Automated conversion from Cobol to Java from maintenance point of view

M.A. Mossienko
St. Petersburg State University, LANIT-TERCOM
198504, Russia, St. Petersburg, Bibliotechnaya sq., 2, 1414
Phone: +7 (812) 428 4187 E-mail: maxim@tercom.ru

Abstract

In this paper an approach for automated conversion from Cobol to Java is presented. This approach is aimed for achieving maintainable results, in some cases even at the price of losing complete functional equivalence to the original code. We are discussing applicability of this approach using a large quantity of real-life programs. We also propose to use object wrapping to deal with language conversion difficulties.

Introduction

There are many critical legacy applications that require system renovation. A significant share of these applications is written using Cobol language [14]. They exist in old environment and need constant maintenance. Any new change erodes their structure and makes the following changes even more expensive. Small ability to adapt for changing business needs also represents a serious problem. However, such legacy systems are critical for business due to hidden domain based knowledge and could not be easily replaced with new solution.

One of the possibilities that should be considered in this regard is migration to modern language or environment and Java could be a very attractive choice:

- Java is simple but powerful object oriented language. Use of abstraction, modularity and encapsulation can provide a lot of benefits for large software systems.
- Java represents a new platform, which is hardware and operating system independent. This could be especially useful for old systems, which have hardware and/or software vendor lock-in. Because of this Gartner recommends evaluating Java as a viable alternative for organizations currently using other programming languages [8].
- Java supports open industrial standards (TCP/IP networking, CORBA, JDBC, XML, SOAP etc).
- Java is taught in most universities and thus it is much easier to find new developers skilled in Java than Cobol.

Java is not a silver bullet but it enables solving software problems significantly easier, so migration to Java is considered quite often. At the moment, the most widespread approach is black box modernization (see [4] for discussion of various transformation scenarios), but the results are not always meeting the customer expectations, because language conversion is harder than it seems [22].

This paper describes an approach for automated conversion from Cobol to Java, which strives to achieve functionally equivalent and maintainable target system. When these goals are contradicting each other, the functional equivalence can become slightly compromised in order to produce more maintainable system. Still, we would like to emphasize that in our approach the vast majority of Cobol constructions are converted to the functionally equivalent Java code. We also used object wrapping to hide the data types conversion complexities from the programmer.

The results described in this paper were obtained during development of automated reengineering tool RescueWare [21] and subsequent application of this tool in industrial projects. The paper is organized as follows. The first section describes related works. Section 2 gives a brief overview of the general conversion methodology. Section 3 contains detailed description of our approach to elementary data types support. Section 4 is devoted to the process of classes extraction employed during conversion. Section 5 discusses features that have only partial support in the generated Java code. In Section 6 we provide statistics of usage of several language constructions in real industrial programs and demonstrate how this information could be used to improve the results of the generation. Section 7 contains a discussion of the results. The paper ends with conclusion, which summarizes the results and identifies directions for future research.

I would like to thank Andrey A. Terekhov (LANIT-TERCOM, Russia) for his valuable help in improving this paper.

1. Related works

Language conversion became a hot topic recently, so there are a number of articles on this subject. Migration of Cobol to Ada is discussed in [10]. Similar experiences during conversion from PL/X to C++ are described in [15]. Strategies for conversion from C++ to Java are shown in [17]. General language conversion difficulties are highlighted in [22], while the paper [5] discusses research issues in the whole field of legacy system renovation. The problems of object detection are discussed in [11, 12, 23]. Survey of black box approaches to legacy transformation...
is presented in [4, 18]. Both papers emphasize suitability of object wrapping for software evolution. Automating language conversion is subject of [24]. Possible ways of legacy systems evolution are given in [19, 25].

### 2. A brief overview of the proposed approach

Several strategic decisions should be made before embarking on a language conversion project [22], namely:

- What is acceptable for the customer in terms of maintainability of the converted system?
- Does the customer require the complete functional equivalence of the generated code to the original program?

Only after answering these questions it becomes possible to write down a detailed list of projections to the target language for each construction of the source one.

Maintainability of the converted system is a critical factor, because the most widespread goal of migration is to prolong the lifetime of the system in its new form. One can consider that the process of migration fails, if usual Java developer cannot grasp the generated code in a reasonable amount of time because he/she does not know Cobol operators semantic. This problem was named in [20] as one of the high risks of reengineering. Due to this reason, the emulation of source language data types and statements is generally not suitable for migration [22], even though it could be successfully used in some cases [15, 16]. On the other hand, usage of native types and operations greatly increases the maintainability of the target system.

Functional equivalence is also required for success of migration process. Unfortunately, in some cases maintainability could not be achieved simultaneously with complete functional equivalence. For instance, it is possible to achieve complete functional equivalence by emulating of all data types and statements that are not supported natively in the target language, but the resulting program is quite likely to be unreadable. Note that even in this case hidden errors in original code [20] or hardware dependencies preclude from 100% equivalence.

So it is clear that the optimal solution should employ some combination of emulation (for equivalence of complicated constructions) and native constructions (for clarity of the generated code). In this approach our goal is to automatically produce maintainable code that is mostly equivalent to the original legacy code, and to make sure that this code could be updated more or less easily by the programmers that are carrying out the conversion (for instance, there could be missing functionality to be implemented or some generated templates that should be manually polished). These manual changes could be viewed as a way to extend or refactor [7] the generated code.

It is obvious that we have abandoned complete automation of the process for the sake of the quality [24]. It was a hard decision that was caused by the severe differences between source and the target languages. Below we describe our process of creating the projections for conversion (adapted from [22]).

First of all, we selected the features that reflect the application domain of the Cobol language and paid special attention to them throughout the process of creating projections. It is especially important to have equivalent transformation for these features as well as ease of use; otherwise the results of conversion would contain too many errors or be hard to extend. The main application domain for Cobol is data processing, and the main language features that support this orientation are [9]:

- The capability of heterogeneous, “record structure” data.
- The capability for decimal arithmetic.
- The capability for convenient report generation.
- The capability for accessing and manipulating masses of data (typically made up of heterogeneous data structure records).

Based on our experience, we would also add transaction processing to this list.

Then we began examining the most widely used constructions of Cobol, trying to match them against modern language idioms. To do this, we scanned more than 20 millions LOC and compiled statistics of usages for Cobol constructs. The most popular language constructions were quite simple. But we also found a lot of Cobol constructs that were simulating some behavior, which is natively supported in modern languages [22]. For example, goto statement is used in Cobol not only for usual non-conditional jumps, but also to emulate missing structured constructs (to return from perform, as exception handling mechanism, as loop or break/continue, as conditional statement etc.).

After that, we started to write down mappings between constructs of source and target languages using the patterns captured on the previous stage. Returning to our example with goto, we decided to translate structural gotos into native constructs of target language, and emulate only "truly" non-structured goto by means of loop/switch scheme.

Then we started to refine our projections. Initially, we focused on finding better alternatives for the most popular forms of constructions. Where possible, we wrote the shortened projections to the target languages for these forms. Even though in general case the projection could be quite complicated, the simple forms of construction in the initial language are always converted to the simple forms in the target one.

We also decided to drop complete functional equivalence for features, which compromise portability or usage. For instance, STOP RUN statement terminates execution of all programs in the call stack. This behavior is usually not appropriate in the target languages, because today this is perceived as a barrier for reuse, so we replaced this construction with GO BACK, which terminates only the current program. Of course, this change breaks the
functional equivalence, but in most cases, this is a desirable effect. We also supplemented the converted code with comments describing this situation.

Unfortunately, we cannot sacrifice correctness when dealing with data types. To avoid this and still get more or less maintainable code, we encapsulated all data types into objects with clearly defined interfaces. In a way, this could be considered as an application of object wrapping inside the converted Java program.

During conversion of specific applications we customized our mappings even further, taking into account the peculiarities of the projects at hand to improve the results of the generation (similar work is described in [1]). We also performed preliminary restructuring of fragments written in a difficult programming style to achieve the code that is more suitable for transformation (this process is described in [2]). The real-life projects also gave us a tremendous amount of feedback that we used to enhance our projections.

3. Data items support

One of the most crucial tasks in conversion from Cobol is handling Cobol data types and their operations. We performed extensive research of this problem and came up with a solution that is discussed here (another approach to this problem is proposed in [5]). In Cobol the elementary variables are described by their format string (PIC mask), which defines the allowed range of the values, and type of allocation (USAGE clause), which is responsible for memory representation. While the Cobol documentation distinguishes several combinations of these data attributes, it is important to note that in any case the variables in Cobol have fixed length semantic (padding or truncating occurs during assignment). Thus we need to support these data types as fixed length strings in the target language as well. Another important observation is that unlike the modern languages, in Cobol it is also possible to perform arithmetic operations over elementary pictures with numeric format. Finally, semantics of some Cobol constructions depends on the representation of the variables in memory; this fact should be properly taken care of during conversion of these constructions.

To cope with this complexity we separated representation of the type from its operations (thus making it an abstract data type). Then we identified native Java types, which are able to perform such operations (so-called native data types approach [22]):

- Numeric types without decimal point are converted into int or long (depending on precision) and their operations.
- Types with a decimal point are converted into BigDecimal class, because maintaining precision of the calculations is very important for Cobol application domain [9].
- Internal floating point variables in Cobol (COMP-1 and COMP-2) are converted into float or double.
- All other types are converted into char or String.

We also created a pair of access methods for reading value (get) and modifying it (set) to deal with memory representation issues and hide other implementation details. This property idiom is very natural for Java programs. One more method was required for numeric items to support the possibility of their usage as a string (for example, in STRING statement).

The actual implementation of access methods depends on answers to the following questions:

- Is the representation of this variable shared with other variables? This situation happens with REDEFINES statement or when the variable participates in group assignments.
- Is it important to keep binary compatibility with Cobol version? For instance, Cobol supports some non-portable types such as variables with allocation type COMP-1.
- Should the exact value range be supported? For instance, is it allowed for variable with PIC XX format to contain more (or less) than two characters?

The answer for the first question could be inferred automatically, because it is described in terms of syntax of the original language.

The second question was answered in the following way: we decided to support binary compatibility only if it is needed for external interfacing with non-migrated application parts. Our statistic has shown that less than 1% of variables employ these types (about 40,000 non DISPLAY items was encountered out of roughly 4,000,000), so support of this feature should not be too much of a burden.

Dealing with the third question, we decided to support value ranges for string variables and not support them for numeric ones (except for the case of DISPLAY variable with shared representation, where the support is a side effect of fixed layout). The reason for this was that we need to support decimal arithmetic properly in order to maintain precision [9]. Indeed, overflow effect is not desirable for applications that work with financial calculations. Another option to support this functionality would be to implement it as infinite loops, but of course in this case the code would not be maintainable.

We are not stating that the solution formulated in these answers is applicable in all circumstances – it is quite likely that in other projects the different ones would be required. In the meantime, our implementation of data types with properties allows the code to be virtually independent of these details.

Finally, we made several improvements to get simpler or more maintainable results of the generation:

- We found out that some variables in Cobol are in fact constants so they do not require any access methods. It is sufficient to generate them as simple constants with the corresponding type.
- Access methods for unused variables are generated under comments.

Unfortunately, we found out that some variables in Cobol are actually constants so they do not require any access methods. Instead, it is sufficient to generate them as simple constants with the corresponding type. Access methods for unused variables are generated under comments.
• FILLER variables in Cobol represent anonymous variables, which could not be accessed directly. They represent about 25% of all variables according to our statistics. We removed all uninitialized FILLERS (about 8% of all data items), and collapsed sequence of initialized fillers with same type into one FILLER variable.

• To make resulting program in Java look more natural (and thus easier to use) we implemented conversion of program identifier names, usually written in capital letters, into Java style names. To achieve this, we used tokenization of Cobol identifier names via minus sign. There are other approaches to this problem, for instance, semi-automatic name restructuring that provides better control of renaming is described in [3], but we tried to achieve maximum automation of this process. Special attention was paid to names of paragraphs and sections. In Cobol they were usually named as in the following pattern: [prefix][number-]<some action>-[exit]. After conversion only <action> part remained.

4. Creation of classes

Identification of classes in legacy system is an open research topic in legacy system renovation field [5, 11, 12, 23]. Automatic creation of meaningful classes appears quite rarely. For example, it is possible to create class for each record but this will produce too many classes, so most of the existing methods consider records only as candidates for classes. Authors of [5] propose to use type inference basing on actual use of the record to determine whether the record should become a class.

Our approach was different – we decided that since the record manipulation is very important feature of Cobol [9], it should be preserved and made easy-to-use in object-oriented equivalent of the system after conversion. We also wanted to provide reasonable class interfaces (i.e., set of class fields and methods accessible from outside the class). So the process of creating classes consists of two parts: detection of such interfaces and transforming the program to hide implementation details. After these modifications virtually all program becomes dependent on interfaces. This could be considered as an object wrapping inside the program.

We managed to find and automatically extract the following meaningful interfaces in original programs:

• Interface for the program. It consists of ‘execute’ method with parameters.

• Data record interface. It consists of its elements' get/set methods.

• Simple services interface. Original programs work with files, databases, reports, screens and other entities. This work is performed via external calls (usually represented in CICS, SQL or other embedded language) or built-in operators. For example, we have encountered CRUD (create, read, update, delete) interfaces in original programs working with databases, send/receive methods in CICS or DPS and so on.

• After detection of interfaces we are modifying the program in order to localize the corresponding implementation details. This operation is related to the second and the third interface types and deals with the following issues:

• Hiding implementation details of the properties. For example, if original program contains static aliases (redefines) for declared variables, then we have to use shared representation for both the original and redefined items. To support that, we had to add private instance variable containing serialized representation of the value, because Java lacks memory support. All access to the variables of this type is performed via get/set methods, as was shown in the previous section. We also decided to improve data encapsulation even further by merging the original variable and its redefining copies into one class (if all such items were top level elementary data, then the new class is created). This merging could be regarded as application of type inference equivalence [5] for static redefines relationship.

• Record manipulation support. Cobol natively supports operations that work with entire structure. We treated these operations as operations over implicit field redefining all fields in the structure (it is easy to see that this is semantically equivalent replacement). Thus, we reduced the problem of structure operations to the operations over shared representation, which we just discussed in the previous point. It is quite likely that there could be other improvements in record manipulation support. For instance, we are currently trying to find out whether merging between records, which are related by some manipulation, is beneficial in all cases (in type inference method [5] this causes subtyping relation to appear).

• Localization of simple services (see the third interface type in the previous list). This is a generalization of approach described in [18]. If we find some additional services related to this class, then we add this service as a method (this technique is used in many object identification methods [5]). These methods hide all implementation details, and then all uses of these services are changed in the original program to the calls of the corresponding methods. For example, when we encounter CICS statement READQ TS(item), we create read method in item class (if there is no class corresponding to item, then the special class is created); then we remove EXEC CICS statement and insert the call to this method instead. After that it is possible to modify this method even further in order to support persistence correctly.

• Common data records for the projects (usually declared in copybook files in Cobol) are converted to the public accessible classes (with public interface) so that they could be reused in all programs in the project. This technique is especially useful if the same common data records are used in several programs for accessing external objects – then this work can be localized in one class for the whole application. During analysis of our Cobol programs we found out that there were some 30,000 first level variables acting as shared records out of
total 200,000, so common data accounts for more than 15% of the whole application. Without this technique, all these variables would have been copied to all programs that are using them.

- We left unchanged all the records that were used as parameters of this program or as parameters for calls of other programs, since these records had special meaning in the original program.
- For classes that required shared representation support, we additionally transformed their nested class instances into elementary fields (if needed) and merged the fields of these nested classes with those of the parent class. This process is repeated until the class no longer contains nested records. We performed this operation to localize implementation details even further. Figures 1 and 2 show the initial state of the program and the results of application of this procedure. Note that the inner record was transformed into the field because the parent class required shared representation for its field.

```java
public class YclcraParam {
    private ReprRef ref = new ReprRef(8);
    public final void setLinkBrthYj( int data ) {
        ref.setInt(0,6,data);
    }
    public final int getLinkBrthYj() {
        return ref.getInt(0,6);
    }
    public final void setLinkProcessCymd( String data ) {
        ref.setString(6,2,data);
    }
    public final String getLinkProcessCymd() {
        return ref.getString(6,2);
    }
    public final void setLinkProcessCc( int data ) {
        ref.setInt(6,2,data);
    }
    public final int getLinkProcessCc() {
        return ref.getInt(6,2);
    }
    public final void setYclraParam( String data ) {
        ref.setString(data);
    }
    public final String getYclraParam() {
        return ref.getString();
    }
}
```

**Figure 2. Java class equivalent.**

- All other records are considered to be unused and thus they are removed. For instance, if we find out that the field LINK-PROCESS-CYMD from the program on Fig. 1 is not used, then the result of the transformation is shown on Fig. 3). We conservatively supposed that common items are used.

The proposed approach produces classes that could be easily understood and modified. It also reduces records, which are not used or are acting as pure wrappers. In the meantime, we did not want to lose the first level variables during these optimizations, since in Cobol they are the most important way to structure the application. So we decided to evaluate whether transforming all first level variables to the classes would provide an acceptable solution. We analyzed our statistics and found out that only 200,000 records out of 1,200,000 were top level. Thus it seems that it makes sense to generate classes for all top level variables.

This reduction of the variables brings other benefits as well. For example, in Cobol program one could reference variables by its name instead of the full qualification if this name is unique in the program (like LINK-PROCESS-CC for Fig.1). In Java the complete qualification is always mandatory (so the variable that we just
mentioned is translated to the ugly phrase yclraParam.linkProcessCymd.linkProcessCc. This effect called name balooning was discussed in the paper [10], where it was proposed to remove common prefixes in record/field names. In our approach, this problem is less pressing, since the produced code usually has only one level of qualification, or no qualification at all! The code snippet that we discussed above would produce yclraParam.getLinkProcessCc() or even simply linkProcessCc if yclraParam variable is also removed.

```java
public class YclraParam {
    private int linkBrthYj;
    private int linkProcessCc;

    public final void setLinkBrthYj( int data ) {
        linkBrthYj = data;
    }

    public final int getLinkBrthYj() {
        return linkBrthYj;
    }

    public final void setLinkProcessCc( int data ) {
        linkProcessCc = data;
    }

    public final int getLinkProcessCc() {
        return linkProcessCc;
    }
}
```

Figure 3. Java class equivalent in different conditions.

While one could argue that the converted program containing these newly created classes is not purely object oriented, the proposed approach still seems to be a reasonable choice during transition to Java. This approach is close to the object wrapping and proved to be quite useful for language conversion.

Another interesting observation is that one of the widely used approaches in Java is use of MVC (model/view/controller) design pattern (for example, it is recommended for J2EE applications [13]). This pattern provides means for separation between components based on their functions (the similar pattern was proposed in work [19] for component oriented reengineering). In our point of view, the classes created from records bound to files and databases could be viewed as Model part, the classes related to reports and screen (user interface) as View and standalone programs could be considered as a Controller. With this division of the components, the generated code becomes quite familiar for Java developer, because it employs the widespread patterns of this platform.

### 4. Conversion limitations

There is a strong temptation to support all Cobol features and thus increase the automation level of the conversion process, but very often supporting yet another feature is not really visible during transition while the deterioration of the quality of the generated code cannot get unnoticed. We encountered a lot of features in Cobol, which justified this assertion (this issue is also discussed in work [24]).

We already discussed earlier that we encapsulated implementation details for all data items (including shared representation details). However, several Cobol operations work not with the actual value of the variable, but directly with its memory representation. We always tried to modify this behavior during conversion, because it is confusing and non-portable. For example, initialization of numeric edited data item is performed by direct memory access – our converted version uses simple assignment for this purpose. On one hand, this change affected the semantics of the operation, but on the other hand, all other assignments work exactly this way, so we have unified the operations behavior. Another consideration is that different compilers produce different results in case of initial value being smaller than the size of the item.

In another case, we had to support printing numeric item. This operation is dependent on the memory representation: printing -10 value in S99 picture item yields 1} or 1p, because, by default, the sign is trailing and stored in first four bits of last character (different results of printing occur due to the encoding – the former is printed for ASCII and the latter for EBCDIC). After conversion, this operation prints only the fixed string value (10). Once again, this is not an exact equivalent of the original semantics, but it improves maintainability of the system.

There is a lot of research dedicated to the problem of dealing with abundance of GOTO constructions in COBOL. For example, the paper [2] discusses a method for removing HANDLE CONDITION statements (which are a preprocessor form of gotos) for COBOL/CICS programs. We have encountered a similar problem that is appears when numerous performs are combined with gotos:

The PERFORM statement in COBOL behaves more or less like a usual call, unless the perform ranges are intersecting dynamically. The behavior of Cobol program in the latter situation is undefined, and yet we have encountered programs that used such constructions. The result of the program in this case depends on the implementation of the Cobol compiler. As an example, we refer to the program on Fig. 4. This program really works
differently on different compilers: when this type of PERFORM is supported as call, control flow visits all paragraphs; when PERFORM is supported via non-local gotos, the paragraph C is not visited because second perform is abruptly ended by passing the end of the first one. Moreover, when control flow will pass C (by other means than perform with C paragraph as last one in range) then it will jump to second display statement.

A.
PERFORM B THRU B-END.
DISPLAY 'finished PERFORM 1'.
STOP RUN.
B.
PERFORM B-END THRU C.
DISPLAY 'finished PERFORM 2'.
B-END.
DISPLAY 'in B-END'.
C.
DISPLAY 'in C'.

Figure 4. Example program for abusing control flow via perform/gotos.

While non-local behavior could be supported in Java via extensive use of exceptions, we avoided this solution, because it would produce completely unmaintainable code with too many exceptions and handlers. We suggest performing preliminary restructuring [6] to avoid this problem.

5. Usage based considerations

To create adequate projections for conversion it is necessary to take into account the information about the real forms of the constructions that are used in practice. For example, let us consider Cobol statement STRING that concatenates its operands. There are several options that could be specified with this statement: the amount of data to be taken from operands (DELIMITED BY), from which position result data should be written (POINTER IS) and how overflow situation should be handled (ON OVERFLOW). The general form of the STRING statement is provided on Fig. 5. The mapping of this construction into Java for the described data types model is shown on Fig. 6.

STRING ID-1 DELIMITED BY "*"
ID-2 DELIMITED BY SIZE
INTO ID-3 WITH POINTER ID-4
ON OVERFLOW GO TO OFLOW-EXIT.

Figure 5. The general form of the STRING statement.

String toPut = "";
int ptr = id4 - 1;
String repr = getId3();
boolean overflow = true;

if(ptr >= 0 && ptr < repr.length()) {
    toPut += repr.substring(0, ptr);
    toPut += before( getId1(), "*" );
    toPut += getId2();

    id4 = Math.min( toPut.length(), repr.length() ) + 1;
    overflow = toPut.length() != mem.length();
    setId3( toPut );
}

if (overflow) oflowExit();

Figure 6. Generated Java code for STRING statement.

This projection is semantically equivalent to the initial Cobol code, but the generated code seems to be quite bloated. To improve this code, we used our statistics to find out how often the complete form of STRING is really used. We found 14,294 STRING statements, and the statistics of options was the following:
• delimitation with size was used 13116 times, delimitation with space 522 and custom delimiter was used 722 times;
• pointer option was used 270 times;
• overflow was mentioned only 18 times.
Thus we concluded that the most often used form of the STRING statement is:

```
STRING ID-1 ID-2 DELIMITED BY SIZE INTO ID-3.
```

This statement has very simple equivalent in Java:
```
setId3( getId1() + getId2() );
```

So the impact of the full STRING statement and the corresponding code bloat became statistically negligible.

We used the same statistics approach to assess other complicated features of Cobol. Of course, most of the features that we considered could have been supported in the target language, but we wanted to receive numerical estimation of the trade-off between investment of resources in the development of the corresponding features and the benefits of this support. As another example of this comparison, we mention the varying arrays – in Cobol they are defined with DEPENDING ON option. While it is possible to support this feature in Java, it turned out that this feature is quite rarely used in Cobol – only 9 arrays out of 28,623 that we encountered were varying (0.031%). Thus we decided that it is possible simply to treat varying arrays as static with some maximum bound.

### 6. Discussion

The described approach heavily favors good programming style: conversion of structural and portable program, which uses most typical construction forms and idioms, yields the best result – the program that is both correct and maintainable. However, in cases when the original program is deviating from the usual practices, additional preliminary restructuring is required. Another problem with the proposed approach is that it requires manual efforts for implementation of the missing functionality.

For example, one of our recent projects involved conversion of the Unisys COBOL application (25KLOC). During this project we encountered binary variables, which used 9 bits in byte. Moreover, we have seen addressing of every bit. While this specific could be hidden with our approach to data types conversion, we preferred to replace it with more traditional Cobol data types, because our analysis of the application revealed that this 9-bit bytes were used only for program-screen interaction, which was changed completely anyway.

We also experienced uncovering logic errors in the original programs after conversion (this is mentioned as one of reengineering risks in [20]). This situation appeared because of stricter error checking in Java: index out of bounds during array processing, incorrect values for numeric variables (it is possible to move spaces into numeric variable in Cobol, and this operation will fail only at subsequent arithmetic operation), usage of uninitialized variables and so on.

Manual changes were made in order to support dialogue with the user or data base manipulation. However, we would like to point out that object interfaces (formed from records reflecting user screens and database schema) remained stable. It turned out that the records used for program-screen interactions contained a lot of unnecessary implementation details. If we are changing the program-screen interactions, then it is possible to remove a lot of this specific.

Finally, the current implementation of this approach suffers from some scope limitations – only Cobol and SQL are supported at the moment, even though legacy programs often use other embedded languages, such as CICS or DPS.

We would like to point out that this report represents a work in progress and we continue to improve our reengineering tool. Still, our experience of applying these methods to industrial projects was very positive. We believe that this work on improvement of the results of the language conversion will eventually result not only in a better code, but also in dramatic increase of the customer satisfaction.

### 7. Conclusion

In this article we presented an automated approach for conversion of Cobol applications into Java. We demonstrated conversion methodology, which is aimed for producing maintainable code. In some rare cases the programs produced with this approach are not functionally equivalent to the original one, but they are closer to the paradigm of the target language. When the functional equivalence of the result was completely critical for the usefulness of the application (i.e., dealing with data types support), we used object wrapping.

The result that is described in this paper could be considered as the first step in improvement of the automated language conversion tools, even though the issues discussed in this paper require further research. We also believe that this approach could be applied for conversion of other programming languages.

### References

17. J. Martin, H. Muller “Strategies for Migration from C to Java” // In Proc. of CSMR'01, pp. 200-209.