September 2, 2014
RT0962
Computer Science 15 pages

Research Report

Refactoring of COBOL data models based on similarities of data field name

Yohei Ueda, Moriyoshi Ohara

IBM Research - Tokyo
IBM Japan, Ltd.
NBF Toyosu Canal Front Building
6-52, Toyosu 5-chome, Koto-ku
Tokyo 135-8511, Japan

IBM

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Refactoring of COBOL data models based on similarities of data field names

Yohei Ueda, Moriyoshi Ohara

IBM Research – Tokyo

yohei@jp.ibm.com, ohara@jp.ibm.com

Abstract.
When legacy COBOL applications are converted into modern Java applications, the number of converted Java classes tends to become bloated due to the considerable differences in the ways data structures are defined in the source and target languages. To address this problem, we developed a refactoring tool that reduces the number of data structure definitions in a COBOL copybook by unifying redundant definitions so as to generate more compact Java classes. Our tool detects similar data structure definitions based on their memory layouts and the names of their member data fields. The similarity of two field names is calculated as the length of a common substring between the two field names. We tested our tool with three different COBOL applications, and showed that our tool eliminated up to 90% of the converted Java classes. We also confirmed that our refactoring tool rarely unifies unrelated data definitions.

Keywords. Legacy Systems Migration, COBOL Copybooks, Java, Data model, Refactoring

1 Introduction

To lower the costs of application development and maintenance, many enterprises are migrating legacy applications written in old programming languages such as COBOL into new applications written in modern languages such Java. The data model conversions in these migrations tend to be difficult. The record-oriented notation to define data structures in COBOL is quite different from the type-oriented notation in Java, so naive conversions from COBOL data structure definitions to Java leads to inefficient code, producing converted code with poor maintainability and poor runtime performance.

The main difference in how to define data structures in the two languages is whether there is a distinction between a data structure and a named composite type. In COBOL, named composite types are not supported to define data structures, and memory layout of each data structure must be directly specified. In other words, there is no distinction between instances and classes, which are different concepts in Java.

Due to the lack of named data structures in COBOL, COBOL programs tend to have a large number of duplicated data structure definitions that have identical memory layout even where the COBOL source code used a single nested data structure. Converting each data structure into Java classes tends to result in a lot of redundant classes that represent the same composite type. Similar but different Java classes ap-

pear multiple times in converted source code, so developers cannot easily select appropriate one from such redundant classes.

The bloated number of Java classes also impacts runtime performance of converted applications. For example, applications that transfer objects via I/O are significantly incurred due to the high overheads in object serialization. Object serialization is usually very slow operation in Java, so it is a common practice that a specialized serializer is prepared for each Java class. When redundant Java classes that represent the same memory layout are generated from COBOL data definitions, redundant Java methods that serialize the same memory layout are also generated. Java JIT compiler wastes compilation time and memory area to compile such redundant Java methods that actually do the same operations.

To solve these problems caused by the bloated number of Java classes, we developed a refactoring tool that unifies *redundant data definitions*, which represent an identical memory layout. To avoid wrong unification of data definitions that are not intended to represent the same type of information, our tool also compares the names of their member data fields. The similarity of two field names is calculated by the length of a common substring between the two field names. Our tool does not unify the data definitions whose data field names are not similar.

We tested our tool with three different COBOL applications, and showed that our refactoring tool can eliminate more than 90% of the Java classes naively converted from one of our COBOL applications. We also empirically confirmed that our refactoring tool rarely unifies unrelated data definitions.

The rest of the paper is organized as follows. Section 2 describes the related works. Section 3 describes the difference between COBOL and Java data definitions. Section 4 describes the algorithm to unify redundant data definitions. Section 6 describes the experimental results using three COBOL applications. Section 7 shows the performance improvements of COBOL-Java data conversions. Section 8 describes conclusion and future works.

2 Related work

In the context of language translations, many methodologies for automatic, semi-automatic, and manual conversions have been proposed [1, 2, 4, 5, 7, 10, 11].

Terekhov et al. [4] discuss the difficulties in conversions between two languages such as COBOL and Java. The difficulties come from semantic language differences such as supported native types and supported language constructs. Features not supported in the target language must be emulated to keep the semantics equivalence. Translated program code with emulated features is usually less readable, and hard to maintain for programmers that are not familiar with code of the original applications. To improve the readability of generated code, we may use features available in the target language that are relatively close to the features in the source language. For example, we may use a 32-bit integer in the target language for an 8-digit decimal in the source language. This substitution will keep the equivalent semantics unless overflows occur, so we need additional tests to guarantee that such overflows never occur.

In this paper, we employ the second approach in order to focus on model refactoring between the source and target languages. We can extend our work to use the first approach to keep the semantics equivalence, but this enhancement is out of our scope, so we used the second approach to simply our discussion.

Ceccato et al. [5, 9] proposed a precise emulation of legacy language data types in Java. Their techniques can handle union types similar to redefines in COBOL. We can enhance our work to support precise COBOL type semantics including redefines in Java using their techniques.

Type inference for COBOL source code is proposed in the context of program comprehension [3, 8]. In contrast to our work in which we use data definition information in copybooks, they only use the information of the usage of each variable in the program to detect identical data types. This approach provides useful advisory information to infer types of copybook items. The type-based approach and our approach exploit complementing information available in source code, so we can improve preciseness of our approach by utilizing type-based information.

3 Differences between COBOL and Java data definitions

In COBOL, a *copybook* is a code fragment that can be included into a different source code using the COPY statement, which is similar to "#include" in C. Copybooks are mainly used to define data structures, and in this paper we used the word

```
01  MEMBERINFO
  03  NAME
    05  LASTNAME    PIC X(20).
    05  FIRSTNAME   PIC X(20).
  03  BIRTHDAY
    05  YEAR        PIC 9(4).
    05  MONTH       PIC 9(2).
    05  DATE        PIC 9(2).
  03 ENROLLDATE
    05  YEAR        PIC 9(4).
    05  MONTH       PIC 9(2).
    05  DATE        PIC 9(2).
  03 CONTACT
    05  ADDRESS     PIC X(20).
    05  PHONE       PIC X(20).
  03 EMERGENCY
    05  NAME
      07  LASTNAME  PIC X(20).
      07  FIRSTNAME PIC X(20).
    05  ADDRESS     PIC X(20).
    05  PHONE       PIC X(20).
```

Fig. 1. An Example COBOL copybook

"copybook" for a COBOL data definition. In this section, we describe how COBOL copybooks are mapped into Java classes.

Fig. 1 shows a COBOL copybook example. This copybook defines a nested data structure MEMBERINFO. This contains data fields NAME, BIRTHDAY, ENROLLDATE, CONTACT, and EMERGENCY. Each of these data fields is also a data structure. NAME contains two data fields LASTNAME and FIRSTNAME, each of which is a 20-byte string. BIRTHDAY contains three data fields YEAR, MONTH, and DATE. YEAR is a 4-digit decimal while MONTH and DATE are 2-digit decimals.

A copybook defines a layout of a contiguous memory area. MEMBERINFO defines a 176-byte memory block, and the memory region between offsets 0 and 19 is a 20-byte string, and can be assessed using "LASTNAME OF NAME OF MEMBERINFO." "YEAR OF BIRTHDAY OF MEMBERINFO" references 4-digit decimals at the region between offsets 40 and 43.

Note that copybooks define memory layouts and data field names, but do not de-

```
class Memberinfo {
  Name name;
  Birthday birthday;
  Enrolldate enrolldate;
  Contact     contact;
  Emergency emergency;
}
class Name {
  String lastname, firstname;
}
class Name2 {
  String lastname, firstname;
}
class Birthday {
  int year, month, date;
}
class Enrolldate {
  int year, month, date;
}
class Contact {
  String address, phone;
}
class Emergency {
  Name2 name;
  String address, phone
}
```

Fig. 3. Java classes generated without refactoring

```
class Memberinfo {
  Name name;
  Date     birthday;
  Date     enrolldate;
  Contact  contact;
  Emergency emergency;
}
class Name {
  String lastname, firstname;
}
class Date {
  int year, month, date;
}
class Contact {
  String address, phone
}
class Emergency {
  Name name;
  String address, phone;
}
```

Fig. 2. Java classes generated with refactoring

fine named composite data types. For example, NAME OF MEMBERINFO and NAME OF EMERGENCY OF MEMBERINFO contains the same sub data fields FIRSTNAME and FAMILYNAME. There is, however, no explicit notation to specify that these two data structures use the same memory layout. This means that COBOL does not have name composite types.

Naïve conversion of this copybook to Java classes is shown in Fig. 3. The class Memberinfo has data fields birthday, enrolldate, contact, and emergency, whose types are defined in separated classes. This example shows three noticeable redundant notations.

The first one is that every data field has the same name to its type. The name of the data field name is Name, and that of enrolldate is Enrolldate, and so on. This result comes from the fact that there is no distinction between classes and instances in COBOL.

The second one is that the two separated classes are generated for Name. One of the generated classes is renamed to Name2 to avoid name collision. Both Name and Name2 has the same fields with the same types, so these two classes can be considered as identical types.

The third one is that the classes Birthday and Enrolldate have different names, but have the data fields with the same names and types. Thus, both the two classes represent an identical memory layout with identical field names, and can be considered as synonym types.

We call a data structure that converted to identical and synonym types a *redundant data definition*. By definition, a redundant data definition represents the same memory layout to another redundant data definition.

Copybooks in real workloads commonly have redundant data definitions. For example, one of our benchmark workloads has a copybook that represents a 500KB data structure, and contains tens of thousands of data fields. The copybook represents a nested data structure, and thus can be converted into a set of Java classes. Naively converted Java classes are about 8,000 classes, but 90% of them are redundant ones.

Redundant classes degrade both maintainability and runtime performance of applications.

In the maintainability perspective, redundant Java classes spoil the readability of source code. When a programmer updates a class, he/she must check whether another class represents the same data structure, and care about consistency among redundant classes. When a programmer needs to write a new code using a class, he/she cannot easily select appropriate one from many similar classes.

In the performance perspective, loading redundant Java classes consumes a certain amount of internal memory in Java VM to hold metadata of the classes. The Just-in-time compiler also consumes time and memory area to compile the methods of the redundant classes. JIT compiler tries to compile frequently executed methods, but if too many methods are compiled, compilation takes too much time to complete, and degrades runtime performance. Compiling too many methods also wastes memory areas for JIT-compiled execution code. When memory areas for JIT-compiled code runs out, remaining methods cannot be compiled, and are executed in interpreter

mode. Interpreter mode is much slower then JIT mode, so the runtime performance is significantly degraded.

4 Unification of redundant data definitions

To avoid the degradation of maintainability and performance due to redundant data definitions, we reduce the number of generated classes by unifying redundant data definitions. Fig. 2 shows a generated Java classes with redundant classes unified. First, the identical classes Name and Name2 are unified into a single class Name, and the declarations of the data fields in types Name and Name2 are changed to use the same type Name. The synonym classes Birthday and Enrolldate are unified into a new class Date, and the data fields of types Birthday and Enrolldate are changed to use the new unified type Date. Note that only the types of data fields are changed and the names of data fields are not changed.

Unification of redundant classes does not change the behaviors of converted application programs since the memory layout is kept unchanged. That is, unification only changes the classes, and does not change instantiated data. In COBOL, copybooks define memory layout of data, and does not define types, so COBOL programs do not contain type operations that may change program behavior when types are unified. Therefore, the program behaviors of converted application programs are unchanged during type unifications.

This unification process reduces the number of generated classes, from 7 to 5 in this example, and reduces necessary resources for class loading and JIT compilation. Moreover, the result class definitions become clearer for programmers than before. Before unification, both the data fields birthday and enrolldate hold date information, but they use different types Birthday and Enrolldate. There is no explicit notation for the fact that both the data fields represent date information. After unification, both data fields use the same type Date, so programmers can easily understand that both the fields contain the same type of information.

The outline of our algorithm to decide whether given two types A and B can be unified is as follow:

1. If the total size of A equals to that of B, then go to Step 2. Otherwise A and B cannot be unified.
2. If the number of data fields of A equals to that of B, then go to Step 3. Otherwise A and B cannot be unified.
3. For A's i-th data field a_i and B's i-th data field b_i, if the following four conditions are met, then A and B can be unified. Otherwise A and B cannot be unified.
   (a) The size of a_i equals to that of b_i
   (b) The offset of a_i from the beginning of A equals to the offset of b_i from the beginning of B
   (c) Both a_i and b_i represent the same primitive type, *or* both a_i and b_i represent the same composite type
   (d) The data field name of a_i equals or is *similar* to the data field name of b_i

Steps 3 (a), (b), and (c) check whether the two data definitions represent the same memory layout or not. In Step 3 (c), we need to compare two composite types when the two data fields are not primitive types. This means that we need to check data definitions a_i and b_i recursively.

In Step 3 (d), we compare data field name of a_i and b_i after we confirmed that the a_i and b_i represents the same memory layout. This comparison checks whether the two data structures that represent the same memory layout actually represent the same type of information. For example, if A only has a data field of an 8-digit decimal integer that represents a birthday of a customer, and B only has a data field of an 8-digit decimal integer for serial number for a customer, then A and B represents the same memory layout, which is 8-digit decimal integer, but represents different types of information. We can distinguish this difference by examining the data filed names. When the names of the two data fields are exactly matched, we can confidently consider the two fields represent the same type of information.

In the cases when the two names are not exactly matched, we still have opportunities of unification. We found that typical COBOL applications have many unifiable data definitions whose data fields are not matched exactly but are similar. We measure this similarity by detecting the longest common substring [16] of two data field names.

We determine the similarity between the two strings based on a calculated value *similarity ratio*. A similarity ratio is the length of the longest common substring between the two strings divided by the length of the shorter string. If the similarity ratio of the two data field names is more than a given threshold, then we consider that the two data fields represent the same category of information.

5 Implementation of the refactoring tool

Our refactoring tool analyzes given COBOL copybook files, and generates refactored Java classes that represent the same data model defined in the original copybooks. The algorithm of data model refactoring is described in Section 4.

In our implementation of our refactoring tool, we used IBM COBOL compiler to extract information of data definitions in COBOL copybooks. The compiler can generate metadata of COBOL source code in the format of XML Metadata Interchange (XMI) [12].

A generated XMI file contains name and data field definitions of every data definition in a copybook. A data field definition contains its size and offset as well as type information. If the data field is a primitive type, then the details of the primitive type is described. Examples of supported primitive types are binary integers, decimal integers, strings, and empty fillers. If the data field is a composite type, then a reference ID for the composite type is provided. The actual definition of the referenced composite type is available elsewhere in the XMI file.

Our tool reads a generated XMI file from copybooks by using a usual XML parser (StAX [13]), and analyzes the data definitions and applies our unification algorithm to every pair of data definitions in the XMI file.

After unification of data definitions, our tool generates final Java files, each of which contains a Java class of a unified data definition.

Each generated Java class has access methods (getters and setters) for its data fields. Users can add new methods to the generated classes for their own purposes. Automatic generation of method code for generated classes (except for access methods) is out of our scope at the moment. Generating method code requires analysis of COBOL logic code in addition to data definitions. Such automatic generation of Java methods from COBOL source code is discussed in the literatures [2], and integration and enhancement to our refactoring tool is our future work.

6 Experimental results

We conducted experiments to evaluate our unification algorithm by applying our refactoring tool to the copybooks of our COBOL benchmarks. Our benchmark consists of three applications RULE, AUTO, and BANK. RULE is a business rule application that checks validity of items filled in an application form input by a customer. AUTO is an application of an automobile manufacturer. BANK is an application of core banking system. These benchmarks are internal benchmarks developed in our company based on real production applications.

Fig. 4. Percentage of reduced Java classes

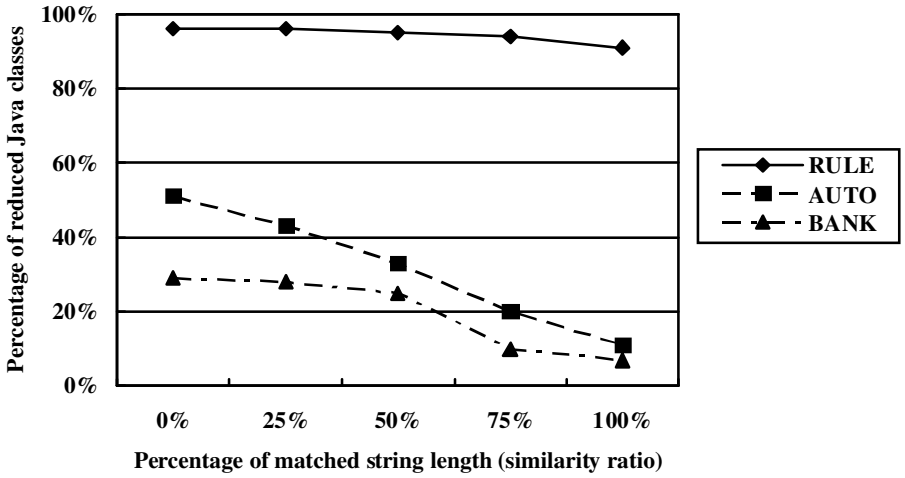


Fig. 4 shows the percentage of the number of redundant Java classes eliminated by our refactoring tool with the similarity ratio changing from 0% to 100%. The similarity ratio 100% is the case that exact matching was used for comparison of data field names. 0% is the case that no comparison of data field names was done.

In the case of RULE with the similarity ratio 100%, we observed 90% Java classes were determined to be redundant, so the total number of classes was reduced to one-tenth of the number of Java classes generated without our refactoring tool. This result indicates that redundant data types frequently appear in the copybook of RULE.

The copybooks in RULE are nested, and some copybooks are imported into multiple copybooks. The data definitions in the imported copybook are expanded into the importing copybook multiple times. This is the main reason why identical data structures frequently appear multiple times in RULE. Other than these cases with nested copybooks, we also found the cases that an identical data structure is explicitly defined multiple times in a single copybook.

As the similarity ratio decreases from 100% to 0%, the percentage of reduced Java classes in RULE gradually increased from 90% to 96% as shown in Fig. 4. This is because not identical but similar data definitions exist in the copybooks.

An example of such similar data definitions is as follows:

```
01 STORE-INFO
  03 MAIN-CONTACT
    05 MAIN-NAME  PIC X(20).
    05 MAIN-PHONE PIC 9(10).
  03 SUB-CONTACT
    05 SUB-NAME  PIC X(20).
    05 SUB-PHONE PIC 9(10).
```

In this example, both MAIN_CONTACT and SUB_CONTACT contain data fields for a person name of 20-byte string. However, the data field names are different even though they have the common suffix “NAME.” The same applies to the data field for a phone number. Exact matching does not recognize these similarities between the data fields. When the similarity ratio is less than 100%, partial marching is used to compare the data field names, and can capture the situation of this example.

The longest common substring between “MAIN-NAME” and “SUB-NAME” is “-NAME”, and its length is 5. The shorter field name is “SUB-NAME”, and its length is 8. Then the similarity between “MAIN-NAME” and “SUB-NAME” is 5/8 = 62.5%. The similarity between “MAIN-PHONE” and “SUB-PHONE” is 6/9 = 66.7%. In the case that the threshold of the similarity ratio is 50%, the ratios of both fields (62.5% and 66.7%) are above the threshold, so the refactoring tool determines that MAIN-CONTACT and SUB-CONTACT are data definitions that represent the same type.

Another example of such similar data definitions is as follows:

```
01 STORE-INFO
  03 CONTACT-1
    05 NAME-1  PIC X(20).
    05 PHONE-1 PIC 9(10).
  03 CONTACT-1
    05 NAME-2  PIC X(20).
    05 PHONE-2 PIC 9(10).
  03 CONTACT-3
```

```
05 NAME-3  PIC X(20).
05 PHONE-3 PIC 9(10).
```

This example shows the data definition of store information that contains three contact points. The data fields of these three contact information have common prefixes and can be unified by our refactoring tool since the similarity ratio of every pair of the data definitions is above a certain threshold.

This kind of repeated appearance of the same data definitions should be represented as an array, but is a common practice in the real world. Our refactoring tool has a functionality that detects this kind of repetition, and replaces it with a Java array.

The 50% similarity ratio achieves 95% reduction percentage. Increasing the similarity ratio more than 50% did not significantly improve the reduction percentage. This means that the 50% similarity ratio is enough to refactor the copybooks of RULE.

In the cases of AUTO and BANK, the reduction percentages were 11% and 7 % respectively at the 100% similarity ratio. Our refactoring tool is also effective for thee two applications, but we observed smaller percentages of AUTO and BANK than that of RULE, mainly because AUTO and BANK do not have nesting copybooks.

As the similarity ratio decreased, the reduction percentages of AUTO and BANK significantly increased, and reached to 33% and 25% respectively at the similarity ratio 50%.

The reduction percentage of BANK did not significantly increase when the similarity ratio decreased from 50% to 0%. This means that the similarity ratio 50% is enough for refactoring the copybooks of BANK.

In contrast to BANK, the reduction percentage of AUTO still increased when the similarity ratio decreased from 50% to 0%. The similarity ratio 0% means that the refactoring tool did not compare the field names at all, and only compared the offset, size, type of each data field. A small similarity ratio achieves a high reduction percentage, but may lead to a wrong decision. The following example shows such a wrong decision in unification:

```
01 BIRTHDAY
  03 YEAR  9(4).
  03 MONTH 9(2).
  03 DAY   9(2).
01 STATUS
  03 CODE   PIC 9(4).
  03 MAJOR  PIC 9(2).
  03 MINOR  PIC 9(2).
```

BIRTHDAY and STATUS are intended to represent completely different types of information, but both data definitions have the same memory layout by coincidence. BIRTHDAY contains one 4-digit decimal integer and two 2-digit decimal integers, and STATUS contains the same data fields in the same order. Only the difference is the names of the data fields.

When the similarity ratio is 0%, BIRTHDAY and STATUS are considered the same type, and are merged as a single Java class. Using a single Java class for variables that represent completely different types of information degrades readability of source code. Note that this wrong unification degrades code readability, but does not lead to an unintentional misbehavior of converted code.

7 Performance of object serialization

As an application of our refactoring tool, we implemented a generator of data conversion code for data communication between COBOL and Java applications. Mission critical COBOL applications are still indispensable in many companies, and it is common practice that newly developed applications written in Java need to communicate with such mission critical COBOL applications. The performance of data conversion between COBOL and Java is very important in such environment.

The Java language provides a default object serialization mechanism [14] that is applicable for objects of any serializable classes. We may use this mechanism for COBOL-Java communication, but we avoid using it since this mechanism is known to be slow in most cases at the cost of its versatility [15]. The default serializer looks up type information of each data fields of the class of a given object since the serializer need to handle objects of arbitrary classes. This runtime type checking is slow. Even when the class of input objects is already known, the runtime type checks cannot be skipped in the default serializer.

To overcome the slow Java's default serialization mechanism, it is common practice to prepare a specialized serializer for each class that requires serialization. Our generator of data conversion code generates serializer and deserializer code in Java for Java classes that generated by our refactoring tool. The generated serializer converts a Java object into a byte stream that represents corresponding data in COBOL. The generated deserializer code converts a byte stream that represents data in COBOL into a corresponding Java object.

Specialized serializers usually provide better serialization performance than the default serializer when the number of classes is small. However, when the number of classes is large, specialized serializers as many as the classes are also generated. As described earlier, loading too many Java classes wastes internal resources of JVM, and are not well handled by the JIT compiler. This leads to performance degradation of COBOL-Java data conversions. Reducing the number of Java classes with our refactoring tool also reduces the number of generated serializer, and improves the performance of COBOL-Java data conversions.

The example code of a deserializer is as follows:

```
// Data structure definition of MyData
class MyData {
  private SubData data1;
  private SubData data2;
```

```
  SubData getData1() { return this.data1; }
  void setData1(SubData data1) { this.data1 = data1; }
  SubData getData2() { return this.data2; }
  void setData2(SubData data2) { this.data2 = data2; }

  // Unmarshaller for MyData
  static MyData unmarshal(byte[] buf, int offset, int
len) {
    MyData myData = new MyData();
    myData.setData1(SubData.unmarshal(buf, offset+0, 20);
    myData.setData2(SubData.unmarshal(buf, offset+20, 20)

    return myData;
}

class SubData {
   private int num;    // 10-digit integer
   private String str; // 10-byte string

   int getNum() { return this.value; }
   void setNum(int num) { this.num = num; }

   int getStr() { return this.str; }
   void setStr(int str) { this.str = str; }

   // Unmarshaller for SubData
   static SubData unmarshal(byte[] buf, int offset, int
len) {
     SubData subData = new SubData();
     subData.setNum(Utils.unmarshalInt(buf, offset+0,
10));
     subData.setStr(Utils.unmarshallStringFromBuffer(buf,
offset+10, 10));
     return subData;
   }
}
```

The class MyData contains two data fields of the class SubData defined subsequently. The class SubData contains data fields of one integer and one string. These two fields are a 10-digit decimal integer and 10-byte string in COBOL respectively. The methods called unmarshal are deserializers of the classes, and generated by our serializer generator. SubData.unmarshal deserializes a 20-byte byte block into a pair of an integer and a string. The utility method unmarshalInt converts a 10-digit decimal integer into 32-bit binary integer. Another utility method unmarshalStringFromBuffer converts a 10-byte EBCDIC string into a Java's Unicode string.

MainData.unmarshal calls SubData.unmarshall twice to converts a 40-byte byte block in COBOL data representation into a Java object contains that holds references to two SubData objects.

The above example only contains deserializers, but serializers that convert data in reverse order are also generated.

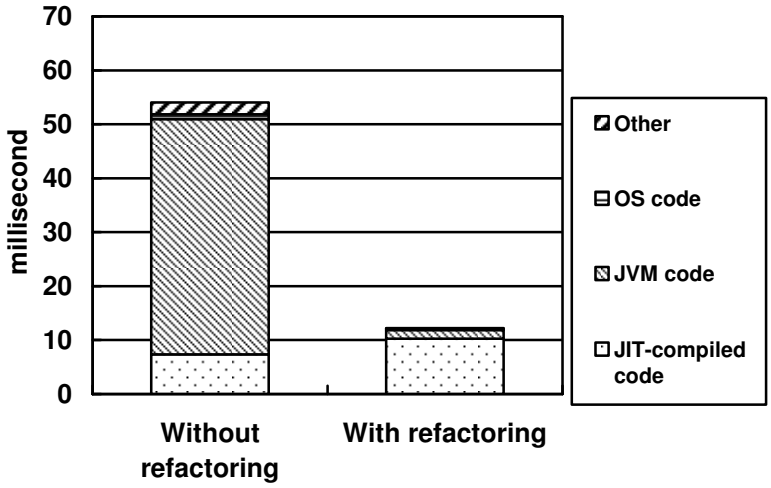
Fig. 5 shows the performance of serialization between COBOL and Java data representations when we applied our serializer generator into the copybooks of RULE.

The graph shows the average elapsed time of round-trip data conversions from COBOL to Java, and back to COBOL. The converted data is a nested copybook in RULE and its size is about 500KB. The experiments were conducted in IBM zEnterprise EC12 with z/OS V1.13. The used Java VM is IBM SDK V8 Beta3.

As the bars in the graph show, our refactoring tool decreased the elapsed time from 54 to 12 milliseconds, and achieved 4.4 times performance improvements in serialization. The bars contain CPU cycle breakdowns collected by a performance profiler. Without our refactoring, CPU time spent in JVM code was dominant, and the portion of JIT-compiled code is small. This indicates that most of method execution is in the interpreter mode. We analyzed this problem, and found out that the memory areas for JIT-compiled code was full, and the JIT compiler failed to compile the methods of generated serializers. We increased the size of the memory areas for JIT compiled code up to 512 MB, but it did not help to resolve the problem.

With our refactoring, the portion of JVM code significantly reduced, and most of the code was executed in the JIT mode. This is the main reason why our refactoring tool improved the performance of data conversions between COBOL and Java.

Fig. 5. Serialization performance of RULE



8 Conclusions and future works

We proposed a refactoring method that reduces the number of data structure definitions in a COBOL copybook by unifying redundant definitions. Our refactoring method is effective for the code bloat problem that typically occurs when data models of legacy COBOL applications are converted into Java data models.

Our refactoring tool detects similar data structure definitions based on their memory layouts as well as the names of their member data fields. The similarity of two field names is calculated by the length of a common substring between the two field names. We tested our tool with three different COBOL applications, and showed that out tool eliminated up to 90% of naively converted Java classes.

Reduction of the Java classes improves readability of source code. We also showed that our refactoring tool improves the performance of data conversions between COBOL and Java.

We introduced a threshold value called similarity ratio to adjust how aggressively the tool unifies similar data definitions. Currently, the tool users have to carefully select an appropriate threshold value by examining the results of our refactoring tool. In our future work, we will develop an automatic mechanism to unify redundant data definitions for a given copybook without tuning a parameter value.

To improve the preciseness of our unification mechanism, we need more information about data items of copybooks. Currently we only use the information available in copybooks, and the information that can be extracted from logic code is not used for refactoring. We will enhance our work so that the refactoring tool analyzes how copybook items are used in source code by using various code analysis methodologies [3, 6, 7].

References

1. Sneed, H.M.: Migrating from COBOL to Java, Software Maintenance (ICSM), 2010 IEEE International Conference on , vol., no., pp.1,7, 12-18 Sept. 2010
2. Mossienko, M.: Automated Cobol to Java Recycling, Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on , vol., no., pp.40,50, 26-28 March 2003
3. van Deursen, A.; Moonen, L.: Understanding COBOL systems using inferred types, Program Comprehension, 1999. Proceedings. Seventh International Workshop on , vol., no., pp.74,81, 1999
4. Terekhov, A.A., Verhoef, C.: The Realities of Language Conversions, Software, IEEE , vol.17, no.6, pp.111,124, Nov/Dec 2000
5. Ceccato, M., Dean, T.R., Tonella, P.; Marchignoli, D.: Data Model Reverse Engineering in Migrating a Legacy System to Java, Reverse Engineering, 2008. WCRE '08. 15th Working Conference on , vol., no., pp.177,186, 15-18 Oct. 2008
6. Høst, E.W, Østvold, B.M: Debugging method names, In European Conference on Object-Oriented Programming (ECOOP 2009)
7. Eierman, M.A., Dishaw, M.T.: The process of software maintenance: a comparison of object-oriented and third-generation development languages, Journal of Software Maintenance and Evolution: Research and Practice archive Volume 19 Issue 1, January 2007

8. van Deursen, A., Moonen, L.: Exploring legacy systems using types, Reverse Engineering, 2000. Proceedings. Seventh Working Conference on , vol., no., pp.32,41, 2000
9. Ceccato, M., Dean, T. R., Tonella, P., Marchignoli, D.: Migrating legacy data structures based on variable overlay to Java, Journal of Software Maintenance, 2010, 22, 211-237
10. Wiggerts, T., Bosma, H., Fielt, E.: Scenarios for the identification of objects in legacy systems, Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on , vol., no., pp.24,32, 6-8 Oct 1997
11. Bisbal, J., Lawless, D., Bing Wu, Grimson, J.: Legacy information systems: issues and directions, Software, IEEE , vol.16, no.5, pp.103,111, Sep/Oct 1999
12. Cover, R: XML Metadata Interchange (XMI), 2001
13. The Streaming API for XML (StAX), `http://stax.codehaus.org/`
14. Java Object Serialization Specification, `http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html`
15. Philippsen, M, and Haumacher, B: More efficient object serialization, Parallel and Distributed Processing. Springer Berlin Heidelberg, 1999. 718-732.
16. Cormen, T.H., Leiserson, C,E, Rivest, R.L., Stein, C: Introduction to Algorithms (3rd ed.), MIT Press (2009)