Astadia™ THE MAINFRAME TO CLOUD EXPERTS

WHITE PAPER

SAFELY TRANSFORMING COBOL COPY STATEMENTS WITH CODETURN TO JAVA/C#

CONTENTS

TRANSFORMING COBOL INTO JAVA/C#

Organizations relying on legacy systems are under pressure to reduce costs and make their IT infrastructure compatible with technologies that are essential to digital transformation. As COBOL-based business applications tend to be large monoliths and COBOL skills become increasingly scarce, IT leaders are looking for solutions to integrate new digital capabilities and migrate to more modern programming languages without interrupting business operations.

CODETURN

CodeTurn is a powerful tool to automatically convert legacy source code into modern source code that is both very well maintainable and 100% functionally equivalent to the original.

It takes an application's source code and automatically converts it to functionally identical sources, now ready to run within an alternative environment on the same platform, or on a different platform altogether.

1. TRANSFORMING COBOL COPY STATEMENTS

If you've ever used COBOL, chances are high you are familiar with the COPY statement. It allows developers to write chunks of reusable code in a separate file, called a copybook. Such a copybook typically contains common data structures (a data copybook) or shared code for e.g. error handling (a code copybook), but in theory it can contain nearly anything you wish to reuse in multiple COBOL programs.

To include such a copybook in your main program, you use the COPY statement. This statement will simply place the prewritten source code in the compilation unit at compile time. The 'compile time' phrase is an important detail here as it means the COPY statement is a preprocessing statement altering the original source file by replacing the entire COPY statement with the contents of the copybook, so that the compiled program no longer contains the reference to the copybook, but instead contains the entire contents from that copybook.

In addition, a REPLACING clause can be specified with the COPY statement. This clause can be used to replace all occurrences of a specified text by the new associated text. For example, if you want to use a generic data copybook in your program and want to give the data items more meaningful names, you would typically do it as illustrated in the code snippets below.

PROGRAM.cbl

```
...
WORKING-STORAGE SECTION.
COPY PRODUCT.
  COPY PERSON REPLACING ==:PREFIX:== BY ==EMPLOYEE==.
  COPY PERSON REPLACING ==:PREFIX:== BY ==CUSTOMER==.
...
PROCEDURE DIVISION.
  ...
  PRINT-NAMES.
    DISPLAY PRODUCT-NAME.
    DISPLAY EMPLOYEE-NAME.
    DISPLAY CUSTOMER-NAME.
```

PRODUCT.cpy

```
01 PRODUCT-RECORD.
  03 PRODUCT-NAME PIC X(50).
  03 PRODUCT-ID PIC 9(10).
```

PERSON.cpy

```
01 :PREFIX:-RECORD.
  03 :PREFIX:-NAME PIC X(50).
  03 :PREFIX:-AGE PIC 9(2).
```

Throughout this paper we'll build further on this example case. For clarity, we'll restrict the converted code samples to Java, but these are of course equally available in C#.

2. COMPARISON WITH OO LANGUAGES

The COPY statement is the main COBOL mechanism to reach code re-use. At the same time, code re-use is one of the pillar stones of OO. So it may come as a surprise that mapping the COPY statement to an OO equivalent is quite challenging. There are some major hurdles that need to be overcome to reach a conversion design that produces code that is both reusable and blends in with the target language.

2.1. Compile time vs Runtime Expansion

COBOL expands the copybook during compile time into the main program resulting in a full-blown compiled program. This is something you typically do not want to do in OO languages, nor do languages like C# or Java at the time of writing even provide a preprocessor that works in this way. What you do want is to preserve your class structure with program classes and copybook classes after compilation and refer to those copybook classes from your programs during runtime. This is a fundamental difference between these languages!

2.1.1. Introductory Example

For the PRODUCT copybook the solution is rather simple:

Program.java

```
public class Program extends CobolProgram {
  private static class WorkingStorage extends CobolDataArea {
    public final ProductCopybook product = new ProductCopybook();

    @Override
    protected void describeData(DataDescriptionContext c) {
      c.addDataDescriptionsFrom(this.product);
    }
  }

  private void printNames(){
    this.cobol.display(ws.product.productName);
  }
}
```

Product.java

```
public class ProductCopybook extends CobolDataCopybook {
  public final CobolField productRecord = new CobolField();
  public final CobolField productName = new CobolField();
  public final CobolField productId = new CobolField();

  protected void describeData(DataDescriptionContext c) {
    c.add(1, "PRODUCT-RECORD", this.productRecord);
    {
      c.add(3, "PRODUCT-NAME", this.productName).picture("X(50)");
      c.add(3, "PRODUCT-ID", this.productId).picture("9(10)");
    }
}
}
```

As you can see, we can create a separate class for the PRODUCT copybook, then instantiate it in our main program and refer to its fields wherever we need them.

2.1.2. More Complicated

Things get more complicated when we consider the PERSON copybook and add replacing clauses in our main program.

Program.java

```
public class PersonCopybook extends CobolDataCopybook {
  public final CobolField _prefix_Record = new CobolField();
  public final CobolField _prefix_Name = new CobolField();
  public final CobolField _prefix_Age = new CobolField();

  protected void describeData(DataDescriptionContext c) {
    c.add(1, ":PREFIX:-RECORD", this._prefix_Record);
    {
      c.add(3, ":PREFIX:-NAME", this._prefix_Name).picture("X(50)");
      c.add(3, ":PREFIX:-AGE”, this._prefix_Age).picture("9(2)");
    }
  }
}
```

Not only do we include the copybook twice, we also replace the field names with their canonical meaningful equivalents, and we want to do this at runtime. That’s why our COBOL services support library offers a large set of predefined methods capable of intervening in numerous ways when the describeData method is executed at runtime.

Program.java

```
public class Program extends CobolProgram {
  private static class WorkingStorage extends CobolDataArea {
  public final ProductCopybook product = new ProductCopybook();
  public final PersonCopybook person1 = new PersonCopybook();
  public final PersonCopybook person2 = new PersonCopybook();

  @Override
  protected void describeData(DataDescriptionContext c) {
    c.addDataDescriptionsFrom(this.product);
    c.addReplacedDataDescriptionsFrom(this.person1,
        replaceNamePrefix(":PREFIX:", "EMPLOYEE")
    );
    c.addReplacedDataDescriptionsFrom(this.person2,
        replaceNamePrefix(":PREFIX:", "CUSTOMER")
    );
  }
}

  private void printNames(){
    this.cobol.display(ws.person1._prefix_Name);
    this.cobol.display(ws.person2._prefix_Name);
    this.cobol.display(ws.product.productName);
  }
}
```

In our main program we can still create and refer to the PersonCopybook instances, but we make use of the 'replaceNamePrefix' method during the data description to replace the ':PREFIX:' parts with their respective counterparts behind the scenes. It is also worth noting that CodeTurn can be configured to generate field variables based on the replaced names in the main program so you can refer to more meaningful variables instead of the copybook variable references.

```
private void printNames(){
  this.cobol.display(ws.employeeName);
  this.cobol.display(ws.customerName);
  this.cobol.display(ws.productName);
}
```

2.1. Advanced Topic: Tokenization vs String manipulation

Now that we've established how to replace code at runtime, let's have a look at the following slightly adapted example:

```
PERSON.cpy
-----------------------------------------------------------
       01 :PREFIX:-RECORD.
         03 :PREFIX:-RECORD-NAME PIC X(50).
         03 :PREFIX:-RECORD-AGE PIC 9(2).
```

And following COPY statement:

```
       COPY PERSON REPLACING ==:PREFIX:-RECORD== BY ==EMPLOYEE==.
```

Easy enough, you would expect the fields 'EMPLOYEE, 'EMPLOYEE-NAME' and 'EMPLOYEE-AGE' after replacing, right? Unfortunately, the COBOL replacing mechanism doesn't work based on simple textual replacements, but instead uses tokenization to determine possible matches. If we break our fields up in tokens, we would get following result:

```
[:][PREFIX][:][-][RECORD]
[:][PREFIX][:][-][RECORD-NAME]
[:][PREFIX][:][-][RECORD-AGE]
```

And the replacing clause:

```
[:][PREFIX][:][-][RECORD]
```

As you can see, only the tokens from the 01 level match with the tokens from the replacing clause and thus will be replaced. The 03 level tokens however are left untouched! To hit home this point even more, this is the literal end-result of the preprocessing:

```
       01 EMPLOYEE.
         03 :PREFIX:-RECORD-NAME PIC X(50).
         03 :PREFIX:-RECORD-AGE PIC 9(2).
```

In this case, this will lead to a COBOL compiler error. In order to get a compiling program, an additional replacing clause would be needed in our COBOL program to replace the 03 levels.

In our converted OO copybooks, those field names are just regular strings, so also in our cobol services support library we'll need to retokenize those strings according to the same COBOL rules in order to determine whether we may or may not apply a certain replacing clause.

2.3. Advanced Topic: Tokenization vs String manipulation

In COBOL it is also possible to define multiple replacing clauses on a single COPY statement. We'll slightly adapt our example again:

```
       01 NEW-GROUP-FIELD.
       COPY PERSON REPLACING ==01 :PREFIX:== BY ==02 EMPLOYEE==
                               ==:PREFIX:== BY ==EMPLOYEE=.
```

Here we created a new 01 group field in our main program, and with our COPY statement we want to change the 01 level from the copybook to a 02 level and change the prefixes. Also, important to know is that the order of these clauses matters. If we would turn them around, the level replacement would never be matched, as the 01 field would already match the order rule which is more generic.

All of this means that we'll need some way to easily describe and chain multiple clauses in our converted programs and have a notion of the order so we can determine which clauses to match first. Let's adapt our main program and see how that looks in a functional and modern-day Java style.

Program.java

```
public class Program extends CobolProgram {
  private static class WorkingStorage extends CobolDataArea {
    public final PersonCopybook person = new PersonCopybook();

    @Override
    protected void describeData(DataDescriptionContext c) {
      c.addReplacedDataDescriptionsFrom(this.person,
           replaceLevel(1, 2).and(replaceNamePrefix(":PREFIX:", "EMPLOYEE"))
       .or(replaceNamePart(":PREFIX:", "EMPLOYEE")));
    }
  }

  ...
}
```

This example shows different replacing functions can be chained together using the 'and' and 'or' methods. Using the 'and' method means both operators must match for the clause to be applied. In this case, if the level can be changed from 1 to 2 and the data item name has a ':PREFIX:' prefix, the replacement will occur, else we continue with the 'or' function. During conversion, CodeTurn will generate all the or-clauses based on the natural order in which the replacing clauses where defined in the original COBOL source.

This builder styled pattern will make it very easy and flexible to describe all sorts of complex replacing syntax which is not only important during conversion, but also once the code has been delivered and needs to be maintained. It allows for developers to quickly add or change replacing's or create new copybooks altogether.

The Astadia COBOL services support library offers a whole range of functions capable of changing levels, changing any part of data item names, adding or removing additional clauses (REDEFINES, OCCURS, ...), removing a whole level, and so forth...

3. THE CODETURN SOLUTION:

SIMPLICITY FLEXIBILITY MAINTAINABILITY

While this paper proves there are many hidden caveats to account for when transforming the COBOL COPY REPLACE syntax, we've only touched briefly on the possibilities of the COPY REPLACING statement. Although rarely used in real-life scenarios, it is possible to write very abstract copybooks and replacing clauses. You can create copybooks with a combination of data declarations and code paragraphs, you could create a data copybook with partly WORKING STORAGE items and partly LINKAGE SECTION items, you could even write a replacing clause replacing all data items by a code paragraph! The mechanism allows for endless possibilities, just as long as the resulting COBOL program compiles.

CodeTurn has an extensive analysis algorithm capable of detecting the complexity of the replacing syntax and whether it can be automatically converted. When things do get too complex, the analyzer will waive a red flag. CodeTurn can be configured to do one of two things then: either you choose for the copybook to be expanded in the main program source, or you choose for the copybook to be generated standalone but the inclusion of the copybook in the main program will be accompanied with a TODO comment specifying the specific issues, which replacing clauses can or cannot be automatically converted and possibly a suggestion on how to proceed.

In our projects, CodeTurn handles nearly 100% of all COPY syntax. In 100% of all cases does CodeTurn generate compiling programs however, and the resulting code reflects our two main design priorities: code needs to be as simple as possible and have as much resemblance to the original COBOL source as possible. These two combined leads to maximal maintainability, which is why we tuck all the complexity and logic safely away in the COBOL services support library.

For more information

Astadia
info@astadia.com
www.astadia.com
+1 877-727-8234