SoftwareMining

Overview of Generated Code

Copyright SoftwareMining Technologies
Version 3.2

Intended Audience:
Java Programmers, Java Architects
C# Programmers and Architect.
# Table of Contents

Intended Audience: ........................................................................................................................................1
Table of Contents ........................................................................................................................................2
Introduction...............................................................................................................................................3
Overview- Layered OO design....................................................................................................................4
  Separation of Data layer from Business Logic ......................................................................................4
  Data Layer Details ................................................................................................................................5
  Object Relational Approach to VSAM Persistence. ................................................................................6
Translation of Business Logic Code ..........................................................................................................7
  Program Calls and Factory Pattern ........................................................................................................7
  Linkage Section items; passing data between programs ....................................................................8
    Calling Other Programs - Execute Method .........................................................................................8
    Passing parameters between programs ............................................................................................8
Memory Cleanup after Program Execution. ...............................................................................................11
Removal of user session ..........................................................................................................................11
GO TO and PERFORM THRU statements. ............................................................................................12
Declarative Section – Conversion to Exception handling .....................................................................13
Embedded SQL Handling ......................................................................................................................14
Report Handling ......................................................................................................................................15
Introduction

This document is an overview of the characteristics of the code generated by SoftwareMining's CORECT translation toolkit. It explains the underlying framework adapted to simplify the generated code, increasing the legibility and maintainability of the new system.

Please note, in order to assist in understanding the structure of the generated code, some of the concepts have been simplified. Whilst the actual code generated from SoftwareMining Tools will have the same characteristics, it may differ to the examples given in this document.
Overview - Layered OO design

Separation of Data layer from Business Logic

COBOL uses WORKING-STORAGE, LINKAGE SECTION and FILE SECTION areas to define variables and records used within each program.

During the translation, the variables declarations are removed from the main body of program, and are placed inside separate Data-Access-Objects (java-Beans) to achieve an Object Oriented and layered design. Every COBOL's 01 level declaration will result in a separate Data-Access-Object.

One of the main features of SoftwareMining translation is "reuse" of the Data-Access-Objects. For example, if two different programs internally defined similar structures – then the system will declare one of them, and use it on both programs. It goes without saying that the data-structures defined in shared copybooks will get reused within all programs.

Consider the following COBOL example:

```
01 COMPANY-INFO.
  03 FILLER                     PIC X(7).
  03 KEY2.
    05 COMPANY-NAME      PIC X(20).
    05 COMPANY-ID       PIC X(20).
    REDEFINES COMPANY-NAME PIC X(20).
  03 COMPANY-DETAIL.
    05 CONTACT-NAME      PIC X(20).
    05 COMP-TEL          PIC X(20).
    05 COMP-FAX          PIC X(20).
```

During translation, the above code segment will be removed from the main body of the program, and placed into a separate data-container class (CompanyInfo). The system will closely monitor the usage patterns of the data-structure, and will generate one of the following types of Data-Access-Object (DAO or Bean)

- When the data-structure contains legacy architectures which the translator cannot remove (e.g. REDEFINES, OCCURS, DEPENDING, Group accessors with numeric fields, … ) – then DAO will have full legacy emulation supports built in (Inherit from AbstractCBLBean).

- Otherwise a simplified DAO will be generated which does not have any emulation support, and hence has a better performance.

In both cases, the DAO’s will have the same variable getters/Setters methods in Java, and Properties in C# - e.g..

```java
public String getCompanyName()
,
public void setCompanyName (String newValue).
```
The translated business-classes will therefore access the data via standard JavaBean notation. E.g.

```java
public class ClientMaintenance extends BaseService {
    CompanyInfo companyInfo = new CompanyInfo (this);

    public void initialize () {
        companyInfo.setContactName("ACME Corporation");
        companyInfo.setCompanyTel("9876543");
        ...
    }
}
```

For C# translation, C# “Properties” will be used to define and the translation will take a more simple format:

```csharp
public class ClientMaintenance : BaseService {
    CompanyInfo companyInfo = new CompanyInfo (this);

    public void Initialize () {
        companyInfo.ContactName = "ACME Corporation";
        companyInfo.CompanyTel = "9876543";
        ...
    }
}
```

**Data Layer Details**

The system translates each record structure (COBOL 01 level declaration) to a separate Data-Access Object (DAO). Where possible, the DAO classes utilize primitive data-types (String, int, double, ...).

However, complicated COBOL structures containing “REDEFINES”, “GROUP” or “RECORD” accessors, or generally when formatting of a field, group or record needs to be maintained, then the generated DAO will reference SoftwareMining specific data-types.

Consider the following example:

```plaintext
01 RECORD-1.
   03 GROUP1.
      05 FIELD-1       PIC X(20).
      05 FIELD-2 REDEFINES Field-1 PIC X(20).
      05 FIELD-3       PIC X(20).
```

The value of FIELD-1 may be set by any of the following means:
To achieve the same functionality, and improve readability - the translated code will use the similar accessors to set the value of fields. E.g.

```java
record.setValue ("new value");
record.setGroup1 ("new value");
record.setField1 ("new value");
record.setField2 ("new value");
```

The system can generate 2 types of data-accessors.

- **Type-1 Accessors** mimics Cobol behaviour. Each record will have an associated character array. All accessors will read and write from the appropriate position in the character array. Record Fields are translated to types "CBLString", "CBLInteger", or "CBLLong" classes. CBL classes act as positional pointer to the record’s character array. CBLGroup provide a similar behaviour for GROUP accessors. Together the CBL and CBLGroup classes will also cater for redefines and special handling of COMP(utational) variable declarations.

- **Type-2 Accessors.** These are generated when the translator can establish no legacy emulation is required (no REDEFINES, GROUP accessors, OCCURS-DEPENDING, Packed-Decimal Formatting, ...) In such cases the translator will generate DAO classes (inheriting from AbstractContainer) – which utilizes Mutable datatypes (MInteger and MString instead of CBLInteger and CBLString). These will have a better performance and lower memory requirement than type-1 accessors.

- **Type-3 Accessors.** These are generated when the translator can establish that in addition to Type-2 requirements, no size information will be required for the DAO fields. For example, if it does not A matter if a String contains “NAME “ or “NAME ”. In such cases the translator will generate DAO classes (implementing BeanInterface) – which utilizes primitive data-types (int and String instead of CBLInteger and CBLString). The bean created for the above example will only contain 2 String fields: field1 and field2. It will also have only 2 sets of accessors for getting / setting the above value.

---

**Object Relational Approach to VSAM Persistence.**

The VSAM File definitions are also translated a “Persistent” Data-Access-Object format. These Persistent-DAOs contain the necessary information to dynamically generate SQL statement (at
runtime) for Create/Read/Update/Delete (CRUD) of data. The translated business logic can then use an Object-Relational API to perform the CRUD operations – e.g.:

```
// initialize Company Record
Company company = new Company(this);

// we are going to look for “ACME CORP”
company.setKey1 ("ACME CORP");

// Find “ACME CORP” (Translation of COBOL START statement)
company.seek(company.getKey1() , CONDITION_TYPE_GREATER);

// alter company tel number
company.setCompanyTel(1234567);

// save company information
company.update();
```

**Translation of Business Logic Code**

**Program Calls and Factory Pattern**

In COBOL, often the target called program is unknown at design time. The following coding standard is often encountered.

```
If ABCD = 1 THEN MOVE "PROGRAM-1" to PROG-TO-CALL-VAR
IF ABCD = 2 THEN MOVE "PROGRAM-2" to PROG-TO-CALL-VAR

CALL PROG-TO-CALL-VAR USING parameters
```

For this reason, the translated programs uses an a Factory Pattern for instantiating target programs. In this design, all Business Logic Programs are subclasses of common parent (BaseClass), which supports passing arguments, and invoked via the `execute` method.

The above call example will be translated into the following code segment:

```
BaseService service;
service = ServicesFactory.createService(progToCallVar);
service.execute(parameters);
```

Another major advantage of this approach is support for distributed system. The Factory design removes necessity for any changes to business logic, when the “called” program needs to be moved another machine. Only the “ServicesFactory” needs to be told of the location of target program.
**Linkage Section items; passing data between programs**

In COBOL the variables passed between 2 programs may have different structures. As far as COBOL is concerned, it is the CHAR-ARRAY representations which will be passed between the two programs. The sending and receiving Structures may thus be different.

COBOL example calling program (Program-1)

```cobol
CALL "PROGRAM-P2" USING STRING-1
```

COBOL example of called program (Program-2)

```cobol
PROCEDURE DIVISION USING PRODUCT-INFO.
```

The SoftwareMining framework, along with the Service Factory pattern described above, provides the necessary supports for the dynamic typecasting and conversion at runtime.

**Calling Other Programs - Execute Method**

Instances of translated classes may be created using calls to ServicesFactory utility class. E.g.

```java
// OPTIONAL : create an instance of BaseService
// used for passing session information,
// screen information, etc.
private BaseService parent = new BaseService() {};

// create an instance of the target class
MyTranslatedClass myTranslatedClass = ServicesFactory.create("com.myCompany.MyTranslatedClass", parent);

// execute the program …
myTranslatedClass.execute();
```

**Passing parameters between programs**

The data wrapper classes can be instantiated as normal objects – i.e.:

```java
com.company.app.idata.MyDataWrapper dataWrapper = new MyDataWrapper (null);
```

Execute method has some overloaded methods to simplify passing of parameters to the target class. The individual fields and groups can be passed (by-pointer) to the called program by:
myTranslatedClass.execute( dataWrapper.getGroup1() , dataWrapper.getField2());

The execute method has been overloaded to take up to 5 parameters.
If passing more than 5 parameters, then parameters setting can be done via “addCallParameter”
method:

```java
MyTranslatedClass myTranslatedClass =
    ServicesFactory.create("com.myCompany.MyTranslatedClass", parent);
myTranslatedClass.addCallParameter(dataWrapper.getGroup1());
myTranslatedClass.addCallParameter(dataWrapper.getField2());
myTranslatedClass.addCallParameter(dataWrapper.getGroup3());
    ...
myTranslatedClass.addCallParameter(dataWrapper.getGroup20());
```

Note that addCallParameter passes value by pointer, i.e. changes to the parameters will be visible to
the calling program.

*Passing Values by Content*
The following method allows passing values by content –

Passing fields / groups from the translated data-objects:

```java
boolean byContent = true;
boolean byPointer = false;
myTranslatedClass.addCallParameter(dataWrapper.getGroup1(), byContent);
myTranslatedClass.addCallParameter(dataWrapper.getField2(), byPointer);
    ...
// finally execute
myTranslatedClass.execute();
```
**Passing the Data-Container (Wrapper) instances.**

There are several overloaded `execute()` methods which can take up to 5 subclasses of `ACMBaseWrapper` classes. E.g.

```java
// create an instance of com.company.app.idata. MyDataWrapper
MyDataWrapper dataWrapper = new MyDataWrapper (null);

// create an instance of the target class
MyTranslatedClass myTranslatedClass =
    ServicesFactory.create("com.myCompany.MyTranslatedClass" ,parent);

// execute the program ...
myTranslatedClass.execute(dataWrapper);
```

An overloaded `execute()` method can take up to 5 parameters:

```java
// execute the program
myTranslatedClass.execute(dataWrapper1 , dataWrapper2);
```

Please note that there are no overloaded `execute()` methods for passing a parameters of different kind – e.g. for passing a mixture of `ACMBaseWrapper` and `CBL/ACMGroup` objects. To pass objects of different types – please use the “`addCallParameter`” method. E.g.

**Passing Other object types.**

The caller and called programs are standard applications. Therefore any type of objects can be

```java
// create an instance of com.company.app.idata. MyDataWrapper
MyDataWrapper dataWrapper = new MyDataWrapper (null);

// create an instance of the target class
MyTranslatedClass myTranslatedClass =
    ServicesFactory.create("com.myCompany.MyTranslatedClass" ,parent);

// pass parameters
myTranslatedClass.addCallParameter(dataWrapper);
myTranslatedClass.addCallParameter(dataWrapper2.getField1());

// execute the program
myTranslatedClass.execute();
```
passed between them.

The first step is to create a ‘setter’ method on the called program – e.g.

```java
Public class MyTranslatedClass extends ... {
    // add new field – and appropriate setter
    private MyNewObject myNewObject;
    public void setMyNewObject(MyNewObject newValue) {
        myNewObject = newValue;
    }
}
```

Now the calling program can set the new value PRIOR to the execute() method is called. I.e.,

```java
// create an instance of com.company.app.idata. MyDataWrapper
MyDataWrapper dataWrapper = new MyDataWrapper (null);

// create an instance of the target class
MyTranslatedClass myTranslatedClass =
    ServicesFactory.create("com.myCompany.MyTranslatedClass", parent);

// pass parameters
myTranslatedClass.setMyNewObject(new MyNewObject());

// execute the program
myTranslatedClass.execute();
```

**Memory Cleanup after Program Execution.**

To mimic the COBOL behaviour, the system will only clean-up the memory for a allocated to program after:

- the program calls the goback() method
- the program calls the quit(); method
- the CALLER program calls the ServicesFactory.cancelService() method.

Otherwise, the system will keep a reference to the particular instance of program.

**Removal of user session**

In a Servlet deployment mode – the method “ServiceFactory.removeSession(…)” will automatically cater for expired sessions.
In a single JVM mode, typically user management will not be required, and on completion of each run – the system will exit. However, ServicesFactory.removeSession(...)” can still be used to control memory management for systems which do not exit on completion of each run.

**GO TO and PERFORM THRU statements.**

Programming languages such as Java and C# do not directly support GOTO, PERFORM THRU, and PERFORM SECTION type statements.

Therefore the translation system will attempt to remove the GO TO statements through use of its pattern matching algorithms. Currently the system can look for and remove some 50 different GOTO usage patterns.

If the Translator has successfully removes all of GO TO type statements, or when COBOL source does not utilize GOTO statement, then generated code will utilize a standard method calls for transfer of control. E.g.

```java
public class businessClass () {  // java class definition
    public void start () {
        // translation of COBOL PERFORM THRU Statement - NO GOTO’s
        // PERFORM INITIALIZE THRU END-INITIALIZE
        initialize();
        endInitialize();
    }

    public void initialize() {  // do something
    }

    public void endInitialize() {  // do something
    }
}
```
However, when the source COBOL programs contain GOTO statements which cannot be removed, a 2ndry method invocation pattern is utilised. This pattern uses reflection and introspection to achieve the same flow control as the original COBOL program. The underlying SoftwareMining libraries provide the necessary implementation of this pattern, and the implementation is kept separate from the translations.

```java
public sealed class businessClass : BaseService {
    @ExecutionOrder(10)
    public void start() {
        // translation of COBOL PERFORM THRU Statement, Plus GOTO's
        //     PERFORM INITIALIZE THRU END-INITIALIZE
        invoke("initialize", "endInitialize")
    }

    @ExecutionOrder(20)
    public void initialize() {
        // translation of GOTO statements
        _goto("exitMethod");
    }

    @ExecutionOrder(30)
    public void endInitialize() {
        // do something
    }

    @ExecutionOrder(30)
    public void exitMethod() {
        // do backup and exit
    }
}
```

Please note that standard Java/C# method can be used in ALL FUTURE ENHANCEMENTS of the new system. I.e. new enhancements do not require the use of ‘GOTOs’ pattern.

**Declarative Section – Conversion to Exception handling**

The different declarative procedures associated with FILE operations are captured during the analysis phase .

```cobol
DECLARATIVES.
    USE AFTER STANDARD ERROR PROCEDURE ON COMPANY-FILE.
    COMPANY-ERROR-PROCEDURE
        MOVE BAD-COMPANY-FILE-STATUS TO WS-FI-FILE-STATUS.
        ...

PROCEDURE DIVISION.
    PROC-1.
    ...
    READ NEXT COMPANY-FILE.
```
The translated code will trap any run time exceptions when dealing with each file, and redirect the control flow to the appropriate translated routine:

```java
public void proc1() {
    try {
        companyFile.next();
    } catch (IOException e) {
        CompanyErrorProcedure(); // or invoke("CompanyErrorProcedure");
    }
}

public void CompanyErrorProcedure() {
    ...
}
```

**Embedded SQL Handling**

The embedded SQL within COBOL programs are partially parsed. The aim of the partial parsing is to re-adjust the SQL Statement for execution within Java/C#. The system does not validate the SQL statement against the database.

Example:

```sql
EXEC SQL
    UPDATE TABLE-1
    SET    FIELD-1 = :F1
    WHERE  ID = :ID
END-EXEC.
```

The above COBOL code is translated into the following code

```java
ESQLProcessor esqlProcessor = new ESQLProcessor (this);
esqlProcessor.prepare("UPDATE TABLE-1 SET A = ? WHERE ID = ?");

// Input variables for embedded sql
esqlProcessor.setParameter(1, anotherRecord.getID() );
esqlProcessor.setParameter(1, "a value" );
esqlProcessor.executeUpdate();
```
Report Handling.

SoftwareMining CORECT translates COBOL Reports Sections into an XML file containing the same information. The XML is then used at runtime together with a set of API to generate the same report. In effect the whole design works like the original COBOL application – but the REPORT SECTION has been moved to an XML file.

For example, consider the following report definition

```
REPORT SECTION.
RD WEEKLY-REPORT
 CONTROLS ARE FINAL, AGENT-ID
 PAGE LIMIT IS 59 LINES
 FIRST DETAIL 6
 LAST DETAIL 56
 FOOTING 59.

01 TYPE IS PAGE HEADING.
03 LINE 1.
 05 COLUMN 8 PIC X(50) VALUE
   "ACME Trading Company - Daily Trading Report".
 05 COLUMN 74 PIC X(4) VALUE "page".

03 LINE 4.
 05 COLUMN 8 PIC X(6) VALUE "Date ".
```

CORECT will generate the following XML definition from above:

```
<reports>
  <report name="weeklyReport" height="59" first="6" last="56" heading="1" footing="59">
    <group bind="pageHeading" name="filler12">
      <entry name="filler2" line="1" column="8" picture="X(50)" > ACME Trading Company - Daily Trading Report </entry>
      <entry name="filler2" line="1" column="74" picture="X(4)" > page </entry>
      <entry name="filler72" line="4" column="8" picture="X(6)" > Date </entry>
  </group>
</reports>
```
Subsequently, the underlying SoftwareMining framework will utilize the XML descriptions in generation of the reports. The SoftwareMining framework is specifically designed to reduce the complexity of within the generated code.

The following COBOL statements

```
INITIATE WEEKLY-REPORT.
GENERATE REPORT-LINE.
TERMINATE WEEKLY-REPORT
```

will hence be translated into the following java or C# code:

```
private ReportProcessor weeklyReport = new ReportProcessor("weeklyReport", this);
weeklyReport.initiate();
weeklyReport.generate("reportLine");
weeklyReport.terminate();
```
The java version of the system uses `java.beans.XMLDecoder` and `XMLEncoder` utility classes to convert the bean classes to/from XML (again – please see generated `com.mycompany.myapp.idata.beans`).

The service provides a single "executeService" method. The method takes 2 string parameters:

- `className` full classname for subclasses of BaseService - eg. `com.softwaremining.examples.loan.LoanService`
- `dataXML` XML representation of an array of BeanInterface - ie BeanInterface[]

This reflects the method in SOAClientUtility class:

```java
// located in com...idata.beans; package/namespace
MyIDataBean lnk = new MyIDataBean();

// assign some values
lnk.setLkInterestRate(6.2);
lnk.setLkLoanLength(15);
lnk.setLkPrincipal(120000); // loan value

// declare an array of BeanInterface
BeanInterface[] beans = {lnk};

// and execute
beans = SOAClientUtility.execute(
   "com.mycompany.myapp.MyBaseClass", beans);
```

**NOTE: DO NOT USE WITH PROGRAMS WHICH CONTAIN SCREEN-IO.**