ISSN 2320 - 2602
Volume 2, No.9, August 2013
International Journal of Advances in Computer Science and Technology
Available Online at http://warse.org/pdfs/2013/ijacst04292013.pdf

Using Feature Direction in Understanding Legacy Programming Code

Dr. Srinivasulu Mannem, Rajyalakshmi S.
Senior Development Lead, Four Soft Limited, Hyderabad, India,
Quality Assurance Specialist, Prime Line Global Solutions, Hyderabad, India,
srinivas.m@four-soft.com, raji_s30@yahoo.co.in.

ABSTRACT

Understanding and maintaining legacy COBOL systems are still a challenging task for both academic research and industry practice. With the development of future direction software engineering, Future direction code comprehension and reengineering for COBOL software systems become a very promising research direction. In this paper, context feature and error handling feature, which are two most important features for COBOL code understanding, are defined. In addition, the advance for feature location and operations in COBOL code is reachable. Program slicing technique is adopted to locate feature code from large COBOL systems. This study/paper reports our experience to date on the application of Feature Direction program understanding in COBOL code.

Key words: Legacy Systems, Legacy programming, Cobol Code.

1. INTRUDUCTION

COBOL applications are widely used and long-lived [10]. According to research firm Gartner, there are roughly 30 billion COBOL transactions processed every day. The expensive issues include the expense associated with running these systems. IBM admits that at least $1.5 trillion has been spent by enterprises to create COBOL/CICS applications [2], and the expense associated with maintaining those applications is increasing rather than decreasing. COBOL applications often run in critical areas of business. For instance, over 95% of finance–insurance data is processed with COBOL. The serious financial/insurance and legal costs that can result from an application failure are one reason for the fear over the software problems.

Feature-Direction Programming (FDP) has been proposed as a technique for improving the separation of concerns in software design and implementation. The field of FDP has focused primarily on problem analysis, language design, and implementation. However, feature direction is applied in understanding COBOL code in our approach by defining the context feature and error handling feature. The approach to understanding legacy COBOL code from feature orientation point of view is presented in this study/paper, which is structured as follows.

Section 2 describes the process to conceptual COBOL code to class diagrams, which is the representation for further feature direction understanding. Section 3 defines the context feature and related operations. Furthermore, section 4 defines the error handling feature and the identification of error handling feature. Section 5 presents a case study to demonstrate our approach. Section 6 presents some related work. Finally, Section 7 concludes the paper with a summary of our experience to date.

2. ABSTRACTION OF COBOL CODE

UML class diagram is adopted to represent legacy COBOL code. This abstraction of COBOL code makes it easier to define the context feature and error handling feature. The approach to abstract COBOL code to class diagram representation is divided into the following steps.

2.1 Dividing Calls into Four Groups

One program PP_s calling another program PP_t is indicated as $PP_s >> PP_t$. One program PP_s not calling another program PP_t is indicated as $PP_s \not> \not> PP_t$. One program PP_s called by another program PP_t is indicated as $PP_s << PP_t$. One program PP_s not called by another program PP_t is indicated as $PP_s \not< \not< PP_t$.

Definition 1: For one program P, its procedures and its functions PP_i, $i \geq 0$, let PP(P) be the procedure and function set of program P, which is indicated as $PP(P)=\{ PP_i | PP_i << P, i \geq 0\}$. PP_n, is called root program element if and only

if $(\forall\ PP_i \in PP(P) \Rightarrow (PP_n >> PP_i))$ AND
$(\forall\ PP_j \in PP(P) \Rightarrow (PP_n \not< \not< PP_j))$

PP_n, is called leaf program element if and only

if $(\forall\ PP_i \in PP(P) \Rightarrow (PP_n \not> \not> PP_i))$
AND $(\exists\ PP_j \in PP(P) \Rightarrow (PP_n << PP_i))$

PP_n, is called node program element if and only

if $(\exists\ PP_i \in PP(P) \Rightarrow (PP_n >> PP_i))$
AND
$(\exists\ PP_j \in PP(P) \Rightarrow (PP_n << PP_i))$

PP_n, is called isolated program element if and only

if $(\forall\ PP_i \in PP(P) \Rightarrow (PP_n \not> \not> PP_i))$ AND
$(\forall PP_j \in PP(P) \Rightarrow (PP_n \not< \not< PP_i))$

In order to identify with the source code as a whole, it is necessary to describe the calling or called relationships of those procedures in program P.

Definition 2: Procedure graph is one graph to describe the calling or called relationships of those procedures in program P, indicated PG. It is composed of nodes and lines. The sequence of procedure graph PG is upper -to-bottom. The procedure the first node represents calls the procedures the next nodes represent. The sequence of the next nodes is the sequence being called in the first procedure.

Definition 3: The procedure layer is one number that represents the depth of one procedure calling other procedures, indicated PL(P). The procedure layer of leaf program elements is 0, the procedure layer of the program elements that only call leaf program elements is 1, the procedure layer of program elements that call the program elements the maximum of whose procedure layers is 2.

Let PP_1 , PP_2, PP_3 be three procedures, and assume that PP_1 is one leaf program element,

($PP_2 > > PP_1$) AND ($PP_2 \not> \not> PP_i$), i≥3,

($PP_3 > > PP_1$) AND ($PP_3 > > PP_2$)

AND ($PP_3 \not> \not> PP_i$), i>3,

then PL(PP1)=0, PL(PP2)=1, PL(PP3)=2.

2.2 Generating Pseudo Classes

Definition 4: For one leaf program element P, PV is its variable set and POP is its operation set. If PV= ϕ , then P is called empty program element. That empty program element is regarded as one class, indicated CLASS Procedure-Name-Empty.

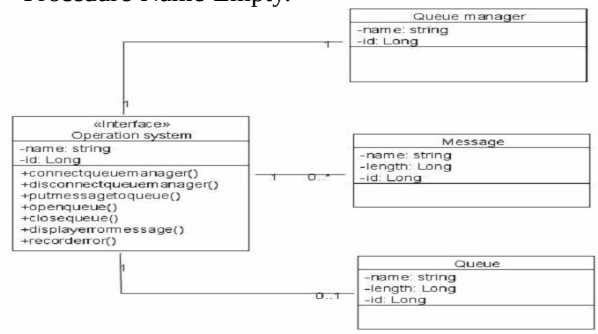


Figure 1: One example of class diagram

The operations in empty program element are transferred into the attributes and operations of that class.

For one leaf program element P and its slicing condition C_i=<p, V_i>, 1≤i≤n, and the corresponding slicing program S_{ci}, PC is the slicing criterion set PC={ C_i }, PCV is the set of slicing criterion variables PCV(P)={ V_i }.

For the first slicing criterion C_1=<p, V_1>, and its slice S_{c1}, PCV(S_{c1}) is composed of the variables of the slicing program S_{c1}. Let V_1 be the first pseudo class, PCV(S_{c1}) is its attributes, POP(S_{c1}) is the operations of that pseudo class. Let V_2 be the second variable, and $V_2 \in$ (PCV(P)-PCV(S_{c1})). For the slicing criterion C_2=<p, V_2>, its slice S_{c2} and the variable set PCV(S_{c2}) are acquired. Let V_2 be the second pseudo class, PCV(S_{c2}) is its attributes, the operations in S_{c2} is the operations of that pseudo class.

The iteration goes on until (PCV(P)- $\sum$ PCV(S_{ci}))= ϕ . Then all the pseudo classes of all the leaf program elements of program P are acquired.

After acquiring all the classes of leaf program elements, node program elements of program P need to be analyzed. Because one leaf program element is one functional module and it is called in the node program elements, it is defined as one class in analyzing node program elements that call the leaf program element.

Definition 5: For leaf program element P, Leaf class is the class with respect to that leaf program element in analyzing the actions and the functions calling it indicated CLASS Procedure-Name-Leaf.

Assume that P is a procedure being sliced and Q is a procedure which is called at statement i in P. The algorithm of inter-procedural slicing CC extended from P to Q is: CC=<n_1^Q, $ROUT(i)_{f\to A} \cap SCOPE_Q$>

where n_1^Q is the last statement of Q, f→A means that the actual parameters will be replaced by formal parameters. $SCOPE_Q$ represents all variables which are accessible in procedure Q.

ROUT(i) = $\cup RIN_C(j)$, where $j \in IMS(i)$.

Assume that the source code has the procedure layer $j=n_0$, C_{ji} represents the i^{th} slicing of the j^{th} layer, and P_j is the procedure whose procedure layer is j. The algorithm computes the pseudo classes which is not empty.

2.3 Generating Real Classes

Pseudo class Vji is slicing-dependent on PCV(Scji). Every pseudo class is one group.

It is necessary to check the validity of the classes and corresponding operations and attributes. If one class is contained in another class, the former class is redundant for the latter.

Definition 6: For one class V_{js}, if $\exists t \neq s$, $\Rightarrow$ (PCV(S_{cjs}) $\subseteq$ PCV(S_{cjt})) AND (POV(S_{cjs}) $\subseteq$ POV(S_{cjt})) then V_{js} is called 'otiose' class. If one 'otiose' class is not leaf class, it is deleted. Then the real classes are generated.

2.4 Defining Relationships among Classes

An association shows a relationship between two or more classes. Associations have several properties:

- A name that is used to describe the coalition between the two classes. Coalition names are optional and need not be unique globally.
- A role at each end that identifies the function of each class with respect to the relations.
- A cardinality at each end that identifies the possible number of instances.

It is necessary to model simplification relationships between objects. Simplification is used to eliminate redundancy from the analysis model. If two or more classes share attributes or behavior, the similarities are consolidated into a super class.

2.5 Pretty Printing Class Diagrams

Consistency of attributes, operations, parameters, and their orders of classes is necessary. The understanding of an interface is performed by the implementation of operations and attributes of a class or a component.

Inheritance is modeled vertically and other relationships horizontally. If two classes interact with each other, some kind of relationship may be needed between them. The transitory relationship is a dependency. In class diagrams, multiplicity between classes is necessary and essential and usually the multiplicity "*" can be replaced by "1..*" or "0..*". An aggregation is a specification of association that depict a whole-part relationship.

3. CONTEXT FEATURE

3.1 Definition of Context Feature

Context feature of legacy COBOL code is the environmental description that introduces the COBOL type [8], the explanation in source code with the notes, and SQL functions. It is presented with UML class diagram that contains three classes: Type COBOL Class, Notes Class, and SQL Class.

3.2 Type COBOL Class

Definition 7: Class of Type COBOL is the class that represents the type of COBOL used in programming. Its name is Type Name, its attribute is the production corporation, and its operation is the extended functionality.

Table 1: Type COBOL class

Class-Name: Type Name
Attribute: Production Corporation
Operation: Extended Functionality

3.3 Note Class

In COBOL code, the notes are useful for explaining the ideas of programming, the structure of program, the precondition of executing statements, the strategy of controlling process, and etc. They are not executable. They describe directly the information for understanding the code and executing what should be done [5].

```
* Initialize the variables for the get call
   MOVE MQGMO-SYNCPOINT TO MQGMO-OPTIONS.
   ADD MQGMO-NO-WAIT TO MQGMO-OPTIONS.
   MOVE VD3-MSGID TO MQMD-MSGID.
   MOVE VD3-CORRELID  TO MQMD-CORRELID.
* Get the chosen message
   CALL 'MQGET' USING VD3-HCONN
                ……
                W01-REASON.
   IF W01-COMPCODE NOT = MQCC-OK
     ……
   ELSE
     ……
   END-IF.
   EXEC CICS IGNORE CONDITION
        MAPFAIL
   END-EXEC.
   ……
   ……
```

In order to describe the notes in comprehending the legacy COBOL code with depicting the context of COBOL code, note class is presented. Note class of legacy COBOL code is one function that contains the notes of one legacy COBOL system to introduce the structure of programs, explain the ideas of programming, present the organization of the system, depict the preconditions of executing statements, illustrate the strategy of controlling process, clarify the anticipated results of the execution in the legacy COBOL code.

Table 2: Note class

Class-Name: Note Class
Line: Line-number
Display: presenting description

3.4 SQL Class

SQL stands for the Structured Query Language. It is one of the fundamental bases of modern database architecture. SQL defines the methods used to create and manipulate relational databases on all major platforms. SQL takes into the programming world many flavors. Oracle databases utilize their proprietary PL/SQL. Microsoft SQL Server makes use of Transact-SQL. However, all of these variations are based upon the industry standard ANSI SQL. All modern relational databases, including MS-Access, Microsoft SQL Server and Oracle regard SQL as their basics. In fact, it's often the only way that can be truly interacted with the database.

Table 3: SQL Command example

Keyword pair	One Example
EXEC SQL	EXEC SQL DELETE FROM HOTEL END-EXEC.
Statement string	IF SQLSTATE NOT = "02000" THEN
Any valid SQL statement	EXEC SQL COMMIT END-EXEC
Statement terminator	ELSE
END-EXEC.	EXEC SQL ROLLBACK END-EXEC

SQL can be used in COBOL programming [13]. SQL statements are identified by the leading delimiter EXEC SQL and terminated by END-EXEC. SQL statements are treated exactly as ordinary COBOL statements with regard to the use of an ending period to mark the end of a COBOL sentence. Any valid COBOL punctuation may be placed after the END-EXEC terminator.

Host variables used in SQL statements must be declared within the SQL DECLARATION SECTION, delimited by

the statements BEGIN DECLARATION SECTION and END DECLARE SECTION. Host variables follow the same scope rules as ordinary variables in COBOL. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for COBOL is the same as a routine.

Table 4: SQL class

Class-Name: SQL COBOL
Organisation: ANSI
Operation: *EXEC SQL… END-EXEC*

SQL is very simple and helpful in COBOL programming. SQL has a limited number of commands and those commands are very readable and easy to understand.

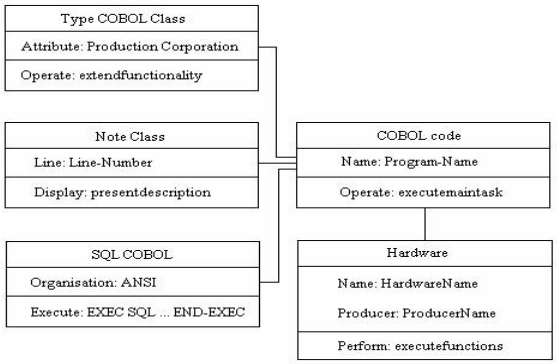


Figure 2: Class diagram of context hardware

4 ERROR HANDLING FEATURE

4.1 Definition of Error Handling Feature

An *error* is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program. The error is a condition dealing with unusual states that changes the normal flow of control in a program. One error in the program may be raised by hardware or software.

The runtime system searches the paragraph for a method that contains a block of code that can handle the error. This block of code is called an error handler. The process and the techniques of coping with the errors in the software system are called error handling.

An exception in the program is any unusual event, erroneous or not, that is detectable by the hardware or software and that may require special processing. An exception is generated when the associated event occurs. The special processing is called exception handling. The code unit that does it is called an exception handler.

4.2 Identifying Error Handling Feature

4.2.1 Candidates of Error Handling

When the program executes its main task, it is conventional to detect the correctness of input and output, the validity of the execution, and the coincidence of the comparison. Only if the error is checked out, the step to cope with the error is performed. Therefore the detection is the first thing of error handling.

Table 5: Example of error handling feature

Normal STOP	IDENTIFICATION DIVISION. PROGRAM-ID. ExampleProgram. …… PROCEDURE DIVISION. …… DisplayInformation. DISPLAY "I did it". …… STOP RUN.
Discussed Case1	…… PROCEDURE DIVISION. …… IF (S NOT GREATER THAN LEVEL1) AND (NOT-ON-ORDER) PERFORM RECORD-ERROR. …… CLOSE FILE1, FILE2, FILE3 STOP RUN. …… END-IF ……
Discussed Case2	PROCEDURE DIVISION. …… PERFORM UNTIL END-OF-FILE …… IF NOT-ON-ORDER …… PERFORM RECORD-ERRORS STOP RUN. ……. END-IF …… CLOSE STOCK-FILE, ORDER-FILE. …… END-PERFORM. ……

COBOL utilizes conditional operations, which indicated as SCD, VERB(SCD)={ IF, IF…ELSE…, EVALUATE, PERFORM…UNTIL (BY)…, CONTINUE, SEARCH } to execute detection tasks. So the set SCD(P) is the candidate of error handling in program P. The discussions below are based on the Weiser's theorem [3], [14].

Theorem 1: Let i be one node of program P. The node i is one candidate of error handling case if

$$ND(i) \neq \phi$$

4.2.2 Error Handling with Termination Keywords

After one program P finishes its task, it stops with COBOL reserved words like "STOPRUN", "GOBACK", "EXIT PROGRAM", or "RETURN". Those are also called Termination Key Words, indicated "STOP".

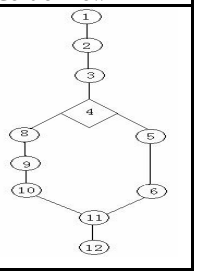
Theorem 2: Let k_0 be the node "STOP", i be one node, t_0 be the last node of program P. One error handling case occurs if $\exists i \in P, (k_0 \in ND(i))$ AND $(k_0 !\in DOM(t_0))$

That is, for one program P, one error handling case happens when the statement STOPRUN is not on every path to the end of program P. The program P stops with not finishing its task. It is called abnormal termination. Its operation set is indicated s_1.

4.2.3 Error Handling with GOTO-EXIT Couple

The statements of GOTO-EXIT couple are used in the error handling. The control body of IF statement includes GOTO statement, and the control of program jumps to the EXIT-PROGRAM.

Table 6: An example of GOTO-EXIT couple

Source Code	Control Flow
1. ACCEPT AA. 2. ACCEPT AB. 3. ACCEPT AC. 4. IF (AA<0) THEN 5. DISPLAY AA. 6. GO TO EXIT-PROGRAM. 7. END-IF 8. ADD AA TO AB. 9. ADD AB TO AC. 10. DISPLAY AC. 11. EXIT-PROGRAM: 12. EXIT	

Theorem 3: Let k_0 be the node "GOTO jump-name", s_0 be the node "jump-name", t_0 be the last node of program P. Assume that i is one node of P. One error handling case occurs if

$$\exists i \in P, (k_0 \in ND(i)) \text{ AND}(s_0 \in DOM(t_0)) \text{ AND } (k_0 !\in DOM(t_0))$$

That is, when one error occurs in program P satisfying one condition in the statement i, the control flow jumps to the flow of the exit of the program P. Its operation set is indicated s_2.

Table 7: Typical mode of error handling

```
BEGIN
INPUT data
<INPUT-ERROR>
<DO input-error-handling>
PERFORM data1
<PERFORM-ERROR1>
<DO perform-error-handling1>
…
PERFORM datan
<PERFORM-ERRORn>
<DO perform-error-handlingn>
OUtPUT data
<OUTPUT-ERROR>
<DO output-error-handling>
END
```

For the example in the Table 6:

k_0 =6; s_0 =11; t_0 =12; i=4;

ND(i)={5,6,8,9,10}; DOM(t_0)={1,2,3,4,11,12}.

$k_0 \in ND(i)$;

$s_0 \in DOM(t_0)$;

$k_0 !\in DOM(t_0)$.

Therefore, the set {5, 6} is the error handling part in the example program.

4.2.4 The Rest of Error Handling

The rest candidates of error handling features in P is indicated s_3: s_3=SCD(P)-s_1-s_2

The error handling is unusual event of processing, and it does not realize the main function of the program except the error handling program. It detects the preconditions in the program and executes special processing. The rest of error handling is extracted with the typical mode of error handling from the rest candidates of error handling in P indicated s_3.

Table 8: Source code-Putting-Message program

```
1     PROCESS-INQUIRYQ-MESSAGE SECTION.
2       IF NOT INITIAL-INQUIRY-MESSAGE
3          MOVE W06-CALL-ERROR TO W06-CALL-STATUS
4          GO TO PROCESS-INQUIRYQ-MESSAGE-EXIT
5       END-IF.
6       MOVE LENGTHOFCSQ4BIIM-MSG TO W03-BUFFLEN.
7       COMPUTE MQPMO-OPTIONS = MQPMO-SYNCPOINT +
8                    MQPMO-PASS-IDENTITY-CONTEXT.
9       MOVE W03-HOBJ-INQUIRYQ TO MQPMO-CONTEXT.
10      CALL 'MQPUT' USING W03-HCONN
11                 W03-HOBJ-WAITQ
12       IF W03-COMPCODE NOT = MQCC-OK
 13         MOVE 'MQPUT'                TO M02-OPERATION
 14         MOVE W06-CALL-ERROR         TO W06-CALL-STATUS
15          GO TO PROCESS-INQUIRYQ-MESSAGE-EXIT
16        END-IF.
17        SET ACCOUNT-QUERY-MESSAGE TO TRUE.
18        MOVE SPACES  TO CSQ4BCAQ-CHARGING.
19        MOVE LENGTH OF CSQ4BCAQ-MSG TO W03-BUFFLEN.
20        COMPUTE MQPMO-OPTIONS = MQPMO-SYNCPOINT +
21                                MQPMO-PASS-IDENTITY-
CONTEXT.
22        MOVE W03-HOBJ-INQUIRYQ TO MQPMO-CONTEXT.
23        CALL 'MQPUT' USING W03-HCONN
24                 W03-HOBJ-CHECKQ
25        IF W03-COMPCODE NOT = MQCC-OK
 26         MOVE 'MQPUT'                TO M02-OPERATION
 27         MOVE W06-CALL-ERROR         TO W06-CALL-STATUS
28          GO TO PROCESS-INQUIRYQ-MESSAGE-EXIT
29        END-IF.
30        MOVE CSQ4BIIM-LOANREQ TO W01-AMOUNT
31       MOVE MQMI-NONE            TO MQMD-MSGID
32        MOVE LENGTH OF CSQ4BCAQ-MSG TO W03-BUFFLEN
33        COMPUTE MQPMO-OPTIONS = MQPMO-SYNCPOINT +
34                    MQPMO-PASS-IDENTITY-CONTEXT
35         MOVE W03-HOBJ-INQUIRYQ TO MQPMO-CONTEXT
36         CALL 'MQPUT' USING W03-HCONN
37                  W03-HOBJ-DISTQ
38         END-CALL
39         IF W03-COMPCODE NOT = MQCC-OK
40            MOVE 'MQPUT   '          TO M02-OPERATION
41             MOVE W06-CALL-ERROR TO W06-CALL-STATUS
42        END-IF.
43     PROCESS-INQUIRYQ-MESSAGE-EXIT.
44        EXIT.
45        EJECT
```

5. CASE STUDY

One COBOL source code, which is named as Putting-Message Program, is presented in Table 8. It is to put one message into the queue written in COBOL 1985. Because it has not notes, therefore its note class is null in context feature. Because,

$ND(2) = \{3, 4, 6, ,7, 9, 10\} \neq \phi$

$ND(12) = \{13, 14, 15, ,17, 18, 19, 20, 22, 23\} \neq \phi$

$ND(25) = \{26, 27, 28, 30, 31, 36\} \neq \phi$

$ND(39) = \{40, 41\} \neq \phi$,

The candidate set of the nodes of error handling in the program is $(ND(2) \cup ND(12) \cup ND(25) \cup ND(39))$. Under the guidance of Theorem-3, the node set of error handling is {3, 4, 13, 14, 15, 26, 27, 28, 40, 41}.

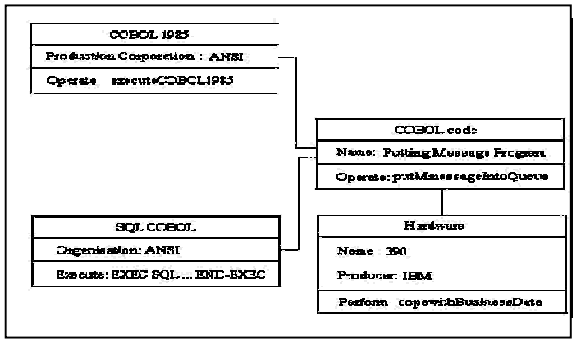


Figure 3**: Generated context feature**

6. RELATED WORK

Because legacy COBOL systems still play an important role in business, a lot of research work has been done to maintain these software systems. [12] Presents an approach to modeling legacy COBOL code with UML collaboration diagrams via a Wide Spectrum Language. [9] Reviews the basic postulates of structured programming as applied to COBOL, and discusses the mechanical transformations archived by an automated restructuring tool. 0 provides a tool taxonomy list which covers more than 100 tools available for working with COBOL. [11] presents an approach to modeling legacy COBOL systems via UML class diagrams and use case diagrams according to the acquisition of domain knowledge.

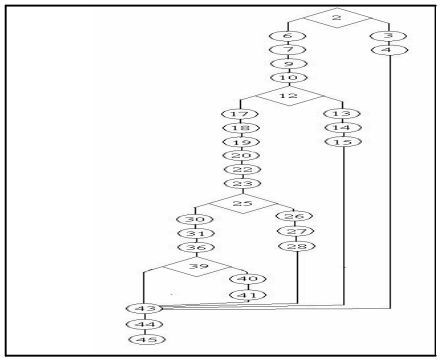


Figure 4**: Control flow of putting message program**

The research of combining FDP and code comprehension is performed in different context. [15] Presents techniques to construct control-flow representations for feature direction programs, and discuss some applications of the representations in a program comprehension and maintenance environment. [6] Proposes several specific techniques such as feature direction or separation of concerns and product maps to assist different RE activities. [3] Points out that some assertions tend to be crosscutting and proposes a modularization of such assertion with feature direction language. [7] Studies FDP in the context of business Programming with COBOL and discusses a typical implementation of feature COBOL.

7. CONCLUSIONS

The approach described in this paper utilizes Feature Orientation to realize the context feature and error handling feature in order to better comprehending the legacy COBOL code. Context feature of legacy COBOL code is presented with UML class diagram. The candidates of error handling features are distilled from COBOL code, and then the error handling feature is derived with the IF-STOPRUN couple, GOTO-EXIT couple, the typical mode. Context feature and error handling feature are greatly helpful for the comprehension and reuse of legacy COBOL code.

REFERENCES

[1] E. C. Arranga, ‘‘COBOL tools: overview and taxonomy”, *IEEE Software*, 17(2), 2012, pp. 59-69.

[2] IBM, *User’s Guide: COBOL and CICS Command Level Conversion Aid for OS/390 & MVS & VM*, Version 2, IBM, 2012.

[3] T. Ishio, T. Kamiya, S. Kusumoto and K. Inoue, ‘‘Future-oriented modularization of assertion crosscutting objects,’’ In *Proc. 12th Asia-Pacific Software Engineering Conference.*, 2011.

[4] J. Jiang, X. Zhou and D. J. Robson, ‘‘Program slicing for C -- the problems in implementation,’’ In *Proc. IEEE Int’l Conf. Software Maintenance*, 1991, pp. 182-190.

[5] C. Jones, *COBOL Programming Course*, available online at http://www.csis/COBOL/Course/capers, 1999.

[6] C. Kuloor and A. Eberlein, ‘‘Future-direction requirements engineering for software product lines,’’ In *Proc.12th IEEE Int’l Conference and Workshop on the Engineering of Computer-Based Systems*, 2009, pp. 98-107.

[7] R. Lammel and K. De Schutter, ‘‘What does Future-oriented programming mean to Cobol?,’’ In *Proc. 4th Int’l Conf. on Future-Oriented Software Development*, 2009, pp. 99-110.

[8] Liant Software Corporation, *RM/COBOL User’s Guide*, Liant Software Corporation, 2003.

[9] J. C. Miller and B. M. Strauss, ‘‘Implications of automated restructuring of COBOL, pp. 76 -- 82.

[10] M. Morach, ‘‘The present and future for past languages -- Cobol,’’ *Database and Network Journal*, 35(1), 2010, pp. 18-19.

[11] J. Pu, R. Millham and H. Yang, ‘‘Acquiring domain knowledge in reverse engineering legacy code into UML,’’ In *Proc. Int’l Conference Software Engineering and Applications*, 2008, pp. 488-493.

[12] N. Stern and R. A. Stern, ‘‘Structured COBOL Programming -- Getting Started with Fujitsu COBOL Update’’, John Wiley & Sons Inc., 2000.

[13] M. Weiser, ‘‘Program slicing,’’ *IEEE Transactions on Software Engineering and methods*, 10(6), 1985, pp. 352--357.

[14] J. Zhao, ‘‘Control-flow analysis and representation for aspect-oriented programs,’’ In *Proc. 6th Int’l Conf. on Quality Software*, 2006, pp. 277-281.

ABOUT AUTHORS

Dr. Srinivasulu Mannem completed his PhD in Computer Science in the area of Software Engineering. Currently he is working as a Senior Development Lead in the one of the world leading product company Four Soft Limited, Hyderabad, India.

Rajyalakshmi completed her MTech in Computer Science and Engineering. Currently she is working as Quality Assurance Specialist with Prime Line Global Solutions, Hyderabad, India.