Object-Oriented Analysis of COBOL

A. Fantechi
Department of Systems and Informatics
Faculty of Engineering, University of Florence
Via S. Marta 3, I 50139 Florence, Italy
fantechi@dsi.ing.unifi.it

P. Nesi
Department of Systems and Informatics
Faculty of Engineering, University of Florence
Via S. Marta 3, I 50139 Florence, Italy
nesi@ingfi1.ing.unifi.it

E. Somma
Department of Information Engineering
Faculty of Engineering, University of Pisa
Via Diotisalvi 2, I 56126 Pisa, Italy

Abstract

The object-oriented paradigm is presently considered the one which best guarantees the investments for renewal. It allows to produce software with high degrees of reusability and maintainability, satisfying in a certain measure also quality characteristics. These features are not obviously automatically guaranteed by the simple adoption of an object-oriented programming language, a process of re-analysis is needed. In this view, several methods for reengineering old applications according to the object-oriented paradigm were defined and proposed. In this paper, a method and tool (C_2O^2, COBOL to Object-Oriented) for analyzing COBOL applications in order to extract its object-oriented analysis is presented. The tool identifies classes and their relationships by means of a process of understanding and refinement in which COBOL data structures are analyzed, converted in classes, aggregated, and simplified semiautomatically. The algorithm is also capable of detecting data structures which can cause problems passing to the next millennium, as demonstrated with an example.

1. Introduction

Software languages are evolving towards new paradigms. As a consequence applications written in the past need to be maintained aligned with the state of the art. This process is often highly expensive since the language evolution proceeds for high steps – e.g., from procedural to object-oriented, from textual to visual, from operational to descriptive, etc. [4]. The literature about the software renewal is quite extensive – e.g., [3], [5], [13]. A solution for minimizing the cost of renewal was proposed by the so-called legacy-techniques. These suggest to reuse old applications by encapsulating them with suitable interfaces. Unfortunately, these techniques are not suitable when the system software must be maintained and updated by modifying internal behavior during its maintenance. This is what usually happens in a great part of the information systems which are used in banks, public administrations, and large firms. A large part of these applications have been traditionally written in COBOL and now there is a strong interest in porting them on new languages and platforms.

The object-oriented paradigm is presently considered the one which best guarantees the investment for the renewal; projects REDO [2] and REBOOT [5] are examples. The object-oriented paradigm allows to produce software with high degrees of reusability and maintainability, satisfying in a certain measure also quality characteristics [1], [9], [10]. These features are not obviously automatically guaranteed by the simple adoption of an object-oriented programming language, though this is a common opinion. The compilation in C++ of an application written in C does not transform it in an object-oriented application (the same statement holds for COBOL and ObjectCOBOL, Pascal and OO Pascal, etc.). In fact, in order to pass from a procedural version of a program/application to an object-oriented version a process of re-analysis is mandatory. The focus must pass from data transformations – i.e., procedure – to data structure, categories and entities, that is, classes and objects. This process leads to the identification of the classes which model the application domain and their organization [1], [12].

In this view, several methods for reengineering procedural applications according to the object-oriented paradigm were defined and proposed – e.g., [11], [7]. More recently, the MOORE approach and tool was presented in [6] for converting COBOL applications into Object COBOL. It works only on pure COBOL85 programs and it is capable of converting COBOL data structures in classes by means of an user-guided process in which MOORE tool proposes solutions while the user selects. In this process only small attention is given to a full analysis of the application and to establishing the optimal relationships among classes in order to minimize the effort of code reuse.

In this paper, a method and tool (C_2O^2, COBOL to Object-Oriented) for analyzing COBOL applications (collections of programs written in COBOL for covering a whole applicative domain) in order to extract its object-oriented analysis is presented. The tool identifies classes and their relationships by means of a process of understanding and refinement in which COBOL data structures are analyzed, aggregated, and simplified semiautomatically. The result is a complete object-oriented analysis of the application domain. The analysis is smart enough to associate structurally similar data having different names as usually happens in COBOL programs. The algorithm is also capable of detecting data structures which can cause problems passing to the next millennium, as we demonstrate with an example.

2. From COBOL to Object-Oriented

The typical COBOL application is organized around a kernel made up by one or more menu modules and by a collection of independent modules which perform the logically separated operations. The organization of modules inside a COBOL application generally reflects the functional decomposition process usually adopted in COBOL to carry out the design. The neat separation between modules simplifies the transformation work, since each module can be seen as a self-standing one with its own well-defined function, different from all the others.

We will deal in the following with the single programs which compose the system. It is possible to classify the COBOL programs in three fundamental types: SUBprograms, BATCH programs and ONLINE programs.

SUBprograms are modules called to perform functions which are too complex or heavy to be included in the main program, or which are used to implement generally used functions or utilities required by several other programs. Roughly, the goal of SUBprograms is to modify the value of some parameters according to some other ones, or to access archives and print reports and presentations.

BATCH programs have as their goal the access to files and/or databases to produce reports and synthetic presentations of the databases. The files, the databases, the reports and the views are the object of program action, though differently from the case of reports or views which structure its control flow, in the case of operations on files or databases, the control flow of the program is established by the operation itself (cancellation, insertion or modification of data).

ONLINE programs process on line transactions. They are built around the transaction they are processing and the structure of such transactions is reflected by the forms of the interface prepared to call the transaction. An on-line transaction can access many files or databases from a single form.

We can indicate as main programs both BATCH and ONLINE modules. For each of these types of programs it is necessary to adopt different conversion strategies which will be clearly specified in the following.

A COBOL program is always structured in four parts (*divisions*):

- IDENTIFICATION DIVISION: reporting the title and author of the program and other information needed to identify the module;
- ENVIRONMENT DIVISION: reporting the configuration sections of the host system, and the definition sections for the file or I/O device control;
- DATA DIVISION: reporting the physical organization of data on both I/O devices and central memory;
- PROCEDURE DIVISION: reporting the sequence of operations to be performed on the defined data, such as the reading/writing access operations on external devices, the numerical computation, controls and alternatives.

Central in a transformation from a COBOL application in an object oriented design is the analysis of the DATA DIVISION, which actually contains all the information needed to create a representation of the data structures of the application domain under examination.

2.1. The Transformation Process

The transformation of a COBOL SUBprogram requires recognizing its specific functionality in the context of the modules which activate it. In particular, the transformation process for a COBOL main program can be performed in the following phases:

- Identification of COBOL data structures, elimination of redundant definitions;
- Identification of COBOL data structures as early classes;
- Identification of class relationships;
- Analysis of data and control referring to classes;
- Reallocation of code to classes.

The first goal of the transformation of a COBOL main program (that is, an ONLINE or BATCH one) is the identification of corresponding classes. This starts by the analysis of all COBOL data structures of the application domain by analyzing all data of modules constituting the application. In this space many data structures are usually repeated. The identification of minimal number of data structures is performed by deleting the redundant definitions by means of several mechanisms.

Once the minimal data structures are identified they can be considered the categories of the application domain, thus early prototypes of classes. According to the object-oriented paradigm the process of analysis continues with the organization of categories in classes by establishing relationships of aggregation, association and specialization. The result is the object-oriented reanalysis of the application domain.

The third phase consists in a transformation process based on the analysis of the accesses to data (e.g. the SELECT, UPDATE, INSERT, DELETE, READ and WRITE operations), with which it is possible to evaluate the relationships between classes and the positioning of the access methods to the class members. In this way, the completion of the static structure of the system is possible by reallocating code to classes and organizing them as methods (fourth phase).

In this paper, only the first three phases are described, since they are those which involve the reanalysis of the application. The last two phases can be in practice semi-automatically performed when classes and class relationships have been identified.

2.2. COBOL Data Structures

The identification of redundant data structures is not at all an easy task; actually, COBOL is extremely clear in the definition of the procedural part, but it is quite obscure in the parts regarding data definition and configuration, since sometimes it leaves space to several interpretations. The motivation for a non-immediate identification of redundancies are due to different causes briefly schematized in the following:

- Since a COBOL programmer is free to define at its will the data structure in the archives, it is not rare that different programmers, in different modules of the same application, give different (maybe similar, but not equal) names to the same data and data structures. This is an example, taken from a real application:

```
05 W-DATE                05 W-DT
   10 DD     PIC 99         10 YEAR  PIC 99
   10 MM     PIC 99         10 MONTH PIC 99
   10 YY     PIC 99         10 DAY   PIC 99

          05 W-D
             10 D PIC 99
             10 M PIC 99
             10 Y PIC 99
```

- Since the visibility of identifiers is global, even for a single programmer there is the need to give different names to the same structure, if it occurs two times in the definition of data in a module. Again an example from the same application:

```
01 FINV
   02 FINV-DATE
      03 FINV-DATE-YY
      03 FINV-DATE-MM
      03 FINV-DATE-DD


01 FBID
   02 FBID-DATE
      03 FBID-DATE-YY
      03 FBID-DATE-MM
      03 FBID-DATE-DD
```

- In the definition of the elementary data the specification of the same format of the PICTURE clause in different ways is possible; this is often used to design logically different data, with the same physical memory occupation:

```
01 W-YEAR PIC 99        01 W-YEAR PIC 9(2)


          01 W-YEAR PIC 9(002)
```

- Having a complete control on the physical representation in memory of data, it is also possible that logically similar data structures are defined as elementary in a module and as compound in others. An example can be given by the representation of the date, for which no international standard is given, but rather a plethora of national or even regional uses of representing a date are in use: this can bring, also inside the same program, to different structures used for representing a date, according to the local use of the date itself.

```
01 W-DATE                        01 W-DATE-N PIC 9(6)
   05 W-YY  PIC 99
   05 W-MM  PIC 99
   05 W-DD  PIC 99
```

In the phase of name recognition, it is possible to uniform the different names which are used for simple data and data structures in different COBOL modules. This process can be performed by establishing a list of synonyms. This mechanism strongly reduces the main causes of redundancy, and thus the number of classes. An example regarding the first reason for redundancy, namely the use of different names by different programmers, is the equivalence: `YEAR = YY = Y`. The static dictionary can answer to the global synonyms problems. More common are local synonyms problems, due to a widely used convention to make identifiers more meaningful.

It is possible to specify other criteria for finding synonyms, for example: (i) discard numeric suffixes, like in: `YEAR-000`; (ii) Hungarian notation: `EV-IN-DT-YEAR` (Event/Initial/Date/Year). The simultaneous use of these filters slows down the conversion, and usually does not provide better results. Actually, it is very likely that every COBOL application uses a single convention for the identifiers. A quick inspection of the source code is enough to recognize the used convention and to select the most useful filter accordingly.

It would also be possible to attempt to further refine the equivalence criteria between identifiers, so to enlarge the automatic recognition cases. In this way a strong dependency is established upon assumptions on the uniformity of the code, uniformity which is not provable and which is often not reasonable: most COBOL applications have been maintained by different people in different times and situations, hence finding uniformity in the code is not justified.

2.3. From COBOL Data Structures to Classes

The identification of classes is performed at two separate levels:

- the identification of elementary classes, as previously discussed;
- the identification and simplification of compound classes.

The identification is performed through the analysis of the definition of the elementary data at the field level (see the PICTURE construct) which, in particular, specifies the visualization and conversion formats. This parametrizes classes, and is used to calculate the memory occupation of numeric or alphanumeric data and to define the default values. For example, the following clauses specify elementary data:

```
10 fiscal-code PICTURE AAABAAAB99A99BA999X
10 YEAR        PIC 99
10 INCOME      PICTURE S9(7)P
10 CARTOON     PICTURE A(7)BA(5) VALUE "GOOFY"
```

The first definition establishes that the `fiscal-code` (i.e., social security number) field is allocated in 16 memory locations of one byte each and is formatted as the example: `GMM MNL 68R10 A509W` (a typical Italian fiscal code).

The identification of the compound classes is performed through the analysis of the layout of the DATA DIVISION, WORKING-STORAGE SECTION and LINKAGE SECTION structures of the main program. The simpler procedure for the conversion of compound data is the transformation of each record in a class and of each field in an instance variable (class member), whose class is specified by the corresponding elementary class if the field is elementary, or by another class, if the record contains a further record in its turn.

The identification of the classes is performed through an iterative strategy based on an algorithm for recognizing the abstractions and simplifying the redundancies. The application of the algorithm provides the automatic recognizing of the structures which form the program, by means of the production of tables and diagrams which allow the user to recognize key aspects of the design or possible areas of refinement of the analysis process. It is then possible to use this information to enrich the basic knowledge on the application and to reapply, iteratively, the analysis algorithm. All the information related to names of variables are collected in knowledge base.

In the following, it is shown, by means of a particularly simple example, how the Automatic Abstraction Algorithm (A^3) works and its tables and diagrams.

The first operation performed by A^3 is the substitution of the synonyms with the base names. In the following a typical example of COBOL data structure definition is reported:

Module ONE

```
01 PATIENT
   05 NAME       PICTURE X(40)
   05 BIRTH
      10 DD     PIC 99
      10 MM     PIC 99
      10 YY     PIC 99
   05 BIRTH-N REDEFINES BIRTH PIC 9(6)
   ...
...
```

Module TWO

```
01 CLIENT
   02 ID   PICTURE XXB999
   02 LAST-ORDER
      03 YEAR  PICTURE 99
      03 MONTH PICTURE 99
      03 DAY   PICTURE 99
    02 NAME      PIC X(40)
   ...
...
```

In the example, it is possible to note how labels YY, MM, and DD in Module ONE are logically equivalent to YEAR, MONTH and DAY in Module TWO. This consideration can be performed in a first reading of the code, and so can be introduced in the knowledge base about the application before the analysis (as we suppose in this case). Alternatively, it can be recorded in the knowledge base after that the abstraction process have produced the first diagrams and results. As it will be shown in the following, once two names are recognized to be synonyms, the secondary name disappears since it is completely equivalent to the first.

The next operation is the building of tables containing important information recovered during the analysis of the application; in particular, the following tables are built:

- Table of system Modules (TM);
- Table of Types (TT): table of formats;
- Table of the Instance Variables (TVI): table of elementary classes, see Fig.1;
- Table of the Classes (TC);

Each module/file of the COBOL application under analysis is stored in TM; in our case TM has the following content:

Table of system Modules, TM		
ModuleID	name	path
1	ONE	/WORK/CO2OC/SBN/
2	TWO	/WORK/CO2OC/SBN/
..	....	...

The only data types which can be directly defined in COBOL are those reported in the following table. Each type is listed with an example of its use and the classes that has been defined as fundamental for any object-oriented application:

Type	example	Class
Numeric	S999V999	CNumber<"S999V999">
Alpha	AABBAAA	CString<"AABBAAA">
Alphanum	A(6)99A999X	CString<"A(6)99A999X">

CNumber<format> and CString<format> are subclasses of CType<format> class, all these classes are provided in a library. These have been parametrized on their format, as shown in Fig.1. In practice, each elementary class defined in the process can be considered as a specialization of parametrized class CType<format> (see for example classes tXX, tX(6), etc., of Fig.1).

Each new format generates an entry in TT, where A means alphanumeric and N numeric:

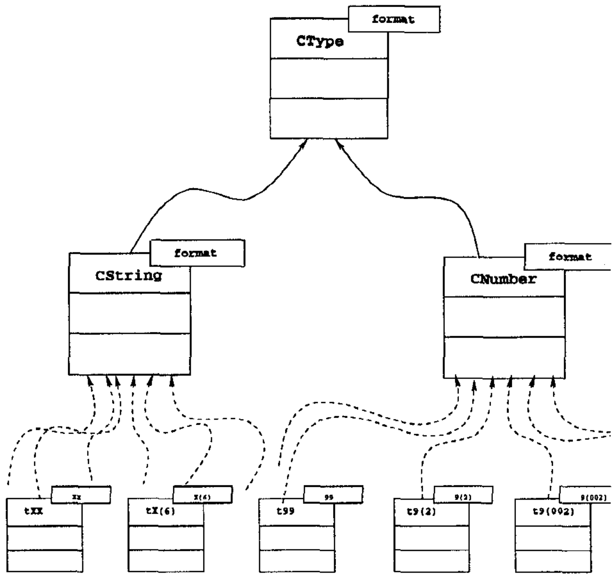

Figure 1. *Relationships among fundamental classes and those which are specifically defined for the application: dashed lines represents in this case the process of class instantiation with parameters, continuous lines are is-a relationships.*

Table of Types, TT			
FormatID	format	Type	Dimension
1	X(40)	A	40
2	99	N	2
3	9(6)	N	6
4	XXB999	A	5
...	...	...	...

Each elementary field label of COBOL data structures generates an entry in TVI. Homonymous labels generate different entries, see for example for NAME:

Table of the Instance Variables, TVI			
TVI_id	LABEL	ClassID	FormatID
1	NAME	1	1
2	DAY	2	2
3	MONTH	2	2
4	YEAR	2	2
5	BIRTH-N	6	3
6	ID	3	4
7	YEAR	4	2
8	MONTH	4	2
9	DAY	4	2
10	NAME	3	1
...	...	..	..

The last column, FormatID, records the references to the formats as indexes into TT. In this table we can also note

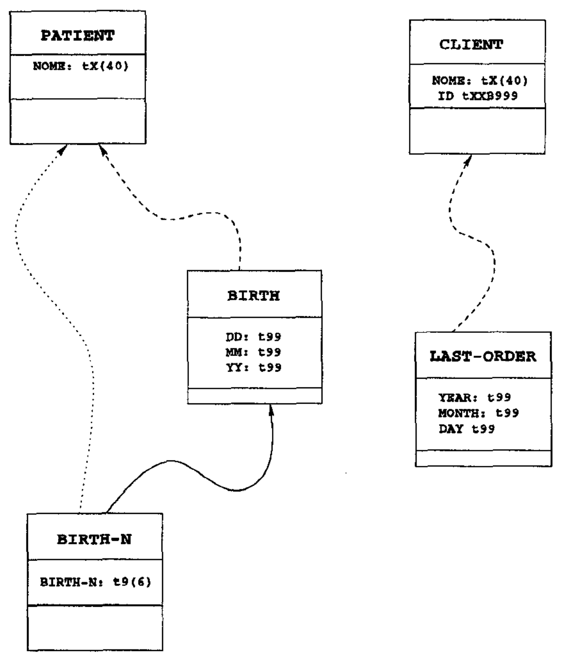

Figure 2. *Relationships among classes after the first phase: is-a as continue lines, is-part-of as dashed lines, is-referred-by as dotted lines. In the bottom some elementary classes are collected.*

that labels YY, MM and DD have disappeared, substituted by their synonyms, as we have seen before. Column ClassID reports the references to the compound classes considered in the analysis of the definitions of records in the DATA DIVISION and thus in TC itself, according to an is-part-of relationship. Each record of the COBOL data structures generates an entry in TC. The relationships automatically obtained after the first phase are reported in Fig.2. The following TC table reports classes and their relationships according to the Object-Oriented paradigm [1], [12]. Each entry of columns is-a, is-part-of and is-referred-by reports the ClassID representing class relationships. In effect, each entry is a list of ClassIDs. In the example reported, only single relationships have been found. In TC the final dimension in bytes of class instances are also reported.

Table of the Classes, TC							
C.ID	class name	M.ID	is-p.-of	is-a	is-r.-by	dim.	cost
1	PATIENT	1	-	-	-	46	46
2	BIRTH	1	1	-	-	6	6
3	CLIENT	2	-	-	-	51	51
4	LAST-ORDER	2	3	-	-	6	6
5	BIRTH-N	1	-	2	1	6	0

Table TC reports also a column reporting an index of the cost of coding according to a simplified version of metrics adopted in [8], [10]. This measure is obtained evaluating the dimensions of locally defined class attributes, and it is useful for having an approximative measure of the effort needed for implementing the system on the basis of the selected class hierarchy, and thus also for reengineering the code. In this case the value of total cost is 99. This value will be reduced by a suitable organization of classes.

In the example, the presence of COBOL clause REDEFINES in the definition of an elementary type means that the same memory area, allocated to a previously defined data structure, is also used by a new data structure. The new data structure could have a different meaning but shares the same memory area. In the example BIRTH-N uses the same area of BIRTH. This mechanism is frequently used by programmers for specifying that a variable is used in the same manner of another variable with a mutual exclusive behavior (e.g., in certain cases the social security number is used in the place of the number of drive licence). This relationship can be regarded as a specialization/generalization since the instances of the subclass can be substantially used as those of an its superclass. For this reason, BIRTH class is considered the superclass of BIRTH-N. Moreover, since PATIENT class presents in the COBOL version two different variables referring the same data area, a relationships of is-referred-by is established with class BIRTH-N. This means that in this specific case each instance of class PATIENT holds an instance of class BIRTH and a pointer to the same instance produced by class BIRTH-N, thus reproducing the early conditions.

2.4. Identification of Class Relationships

The second phase of analysis consists in eliminating redundant instance variables in table TVI and unused classes in TC. This process is drawn on the basis of the relationships among data structures reported in the other tables. As the first result redundant instance variables in TVI are removed and the other tables will be updated accordingly, thus TVI becomes:

Table of the Instance Variables, TVI			
TVI_id	LABEL	ClassID	FormatID
1	NAME	5	1
2	DAY	2	2
3	MONTH	2	2
4	YEAR	2	2
5	BIRTH-N	4	3
6	ID	3	4

Please note that this table already reports the final values in the ClassID column, is-part-of relationships between elementary classes and the other classes. In this phase, each entry of column ClassID of table TVI is in practice a list of ClassIDs.

In the case in which classes with identical layout are detected it is possible to eliminate directly the redundant classes, with the care of maintaining, inside the remaining class, the synonyms collected in the knowledge base of the eliminated classes. In this case the analysis can proceed automatically. On the other hand, this can lead to wrong assumptions since structurally equal classes, which could be considered of the same type, can have very different behaviors. In this case the analysis of the functional aspects should help to take the final decision.

In our example, by forcing the equivalence of the identifiers BIRTH and LAST-ORDER it is possible to consider those classes as unifiable, since they have the same dimension and equal instance variables. The redundant class is deleted from TC.

In the case in which classes are equal "to a certain point" it is possible to create a class hierarchy (is-a relationships) which aggregates in the superclass the common part and in the subclasses the variations and enrichments.

In our example both PATIENT and CLIENT classes share the variable NAME and after the previous step, the BIRTH/LAST-ORDER class. It is possible therefore to create a class (e.g., Super-PATIENT-CLIENT) to which to assign the common contents and to be considered as a superclass of the two previous classes. In the example TC and has been updated as follows:

Table of the Classes, TC							
C.ID	class name	M.ID	is-p.-of	is-a	is-r.-by	dim.	cost
1	PATIENT	1	-	5	-	46	0
2	BIRTH/ LAST- ORDER	1	5	-	-	6	6
3	CLIENT	2	-	5	-	51	5
4	BIRTH-N	1	-	2	1	6	0
5	Super- PATIENT- CLIENT	-	-	-	-	46	46

In TC, the final dimensions in bytes of class instances are reported. These values are substantially different from the theoretical dimension of the class which, by means of the is-a relationship, can be even defined without adding instance variables (such as BIRTH-N with respect to its superclass BIRTH/LAST-ORDER). In fact, with this new class hierarchy a strong reduction of cost of coding has been obtained, thus producing a total cost of coding equal to 57. In Fig.3 the result of the final analysis is reported in the form of a class tree.

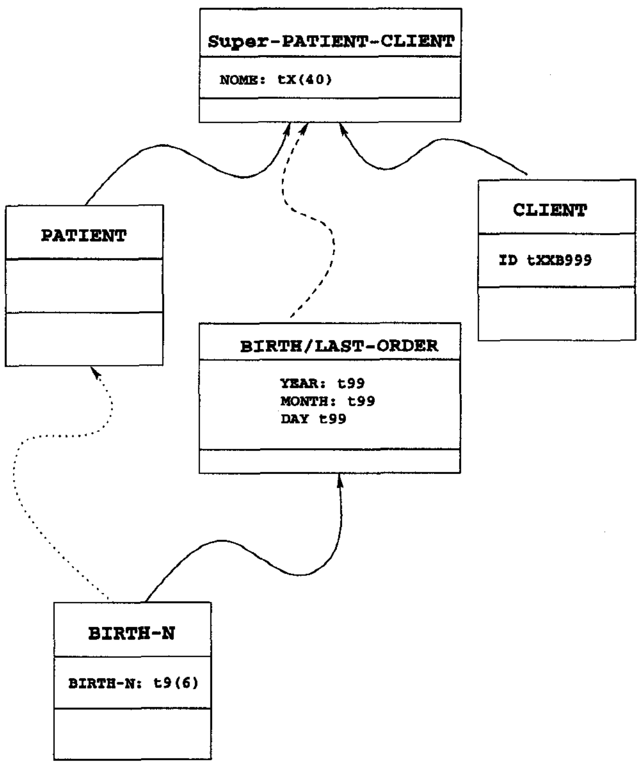

Figure 3. *Final class relationships: is-a as continue lines, is-part-of as dashed lines, is-referred-by as dotted lines.*

3. C_2O^2 : A Tool for Reengineering COBOL Applications

C_2O^2 (COBOL to Object-Oriented tool) is an instrument for analyzing COBOL applications by means of the object-oriented paradigm. The analysis performed by C_2O^2 tool is focussed on modeling the problem domain according to the object-oriented paradigm.

C_2O^2 is based on a Lex/Yacc engine which is capable of processing all COBOL syntax and semantics (the current prototype is able to recognize a particular version of COBOL close to the standard, but provisions have been included to customize it to other versions). Moreover, due to the high number of similar and identical data structures which are usually present in a COBOL application, the process of data structure unification needs to be practically supervised by the user.

This is performed by means of the definition of a knowledge base containing synonyms or by direct assistance from the user. In the case in which similar data structures identified by C_2O^2 are missing in the knowledge base, the final decision about their unification is left to the user.

The objective of the software prototype we have developed, was to show the practical feasibility of the reengineering trajectory sketched in the previous sections.

The prototype is able to perform the recognition, the abstraction and transformation of concepts, to support different re-engineering methods, to support multiple models, to provide visual and textual multiple but consistent representations, and to provide an explicit support to software evolution techniques.

The described features are made accessible to the user by means of two main activities of C_2O^2 : the automatic abstraction, using the A^3 algorithm shown in the previous section and the interactive modelling, which allows the updating of the models of the source code under examination, following an evolutionary strategy, according to the user's needs. The models are collections of meaningful activities, obtained by the analysis of the source code. Once it has been created, a model can be refined by the user by means of commercially available specific evolutionary design tools.

4. C_2O^2 and The Millennium Problem

The Millennium Problem (MP) is substantially a systematic implementation error due to the wrong acquisition of requirements in the early phases of many applications. The first application of C_2O^2 was the re-analysis of the software for managing university libraries on a national basis. It is called SBN (Servizio Bibliotecario Nazionale, National Library Service) and consists of more than 2000 modules for a total of more than ten megabytes of COBOL code. As much as 350.000 lines of obscure and old-fashioned COBOL crafted by more than 30 developers, without a constant evolution plan over eight years of use. In this software, several instances of the MP have been found, in fact all the modules have only two-digit-year in the date definitions. Other systematic errors can be found in procedural code, especially when the date is read from the system clock.

The process of COBOL code analysis has identified a huge amount of data structures containing dates. The typical conceptual process which C_2O^2 follows for enlightening the problem – first the format and then the label unification – has allowed to classify no more than ten data structures as instances of a same class DATE parametrized on the physical storage format.

By using C_2O^2 the passage from the early implementation to an object-oriented description of the domain problem was performed by means of a reasoning-based iterative process.

5. Conclusions

In this paper, a method and tool (C_2O^2 COBOL to Object-Oriented) for analyzing COBOL applications has been presented. The tool is capable of identifying classes and their relationships by means of a process of understanding and refinement in which COBOL data structures are analyzed, aggregated, and simplified semiautomatically. The tool can support different re-engineering methods. It has been used on a large application such as the software for managing all libraries of the University of Florence (where many instances of the millennium problem were found) producing very interesting results. Work is in progress for implementing the mechanism for the automatic association of procedural parts to classes.

References

[1] G. Booch. *Object-Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, California, USA, 1994.

[2] P. T. Breuer and K. Lano. Creating specifications from code: Reverse-engineering techniques. *Software Maintenance: Research and Practice*, 3, 1991.

[3] T. J. Briggerstaff and A. J. Perlis. *Software Reusability: Volume I, Concepts and Models.* Addison Wesley, ACM Press, New York, 1989.

[4] G. Bucci, M. Campanai, and P. Nesi. Tools for specifying real-time systems. *Journal of Real-Time Systems*, pages 117–172, March 1995.

[5] J. Faget and J. M. Morel. The reboot approach to the concept of a reusable component. In *Proc. of 5th Annual Workshop on Software Reuse, WISR'92*, Palo Alto, CA, USA, 26-29 Sept. 1992.

[6] H. Fergen, P. Reichelt, and K. P. Schmidt. Bringing objects into cobol: Moore - a tool for migrating from cobol85 to object-oriented cobol. In *Proc. of the International Conference on Technology of Object-Oriented languages and Systems, TOOLS USA 94*, pages 435–448, 1994.

[7] I. Jacobson and F. Lindstrom. Re-engineering of old systems to an object-oriented architecture. *ACM SIGPLAN Notices*, 26(11), October 1991.

[8] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics, A Practical Guide.* Prentice Hall, New Jersey, 1994.

[9] P. Nesi. *Objective Software Quality, Proc. of Objective Quality 1995, 2nd Syposium on Software Quality Techniques and Acquisition Criteria.* Number 926 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1995.

[10] P. Nesi and M. Campanai. Metric framework for object-oriented real-time systems specification languages. *The Journal of Systems and Software*, July 1996.

[11] C. L. Ong and W. T. Tsai. Class and object extraction from imperative code. *Journal of Object Oriented Programming*, 1993.

[12] J. Rumbaughm, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, New Jersey, 1991.

[13] O. Signore and M. Loffredo. Some issues on object-oriented re-engineering. In *Proc. of ERCIM Workshop on Methods and Tools for Software Reuse*, Heraklion, Crete, Greece, 1992.