On Applying Design Pattern Approach to Reengineering COBOL Programs

Krassimir Manev[†] and Neli Maneva[‡]
[†] New Bulgarian University, 21 Montevideo Str., Sofia, Bulgaria
kmanev@nbu.bg
[‡] Institute of Mathematics and Informatics, BAS, Acad. G. Bonchev Str., Bl. 8, Sofia, Bulgaria
neman@math.bas.bg

ABSTRACT

The Design Patterns (DP) approach is one of the modern techniques in the area of Software Engineering. It has been introduced to facilitate and make more effective the process of design and implementation of software - especially within the object-oriented paradigm (OOP). As usual, each technique that has proven useful for design and development of software is applied, sooner or later, in some other related activities as quality assurance, maintenance, etc. DP approach is actively used in reengineering of existing software systems, too. Unfortunately, when a system is written in an old fashioned language like COBOL (in case of the so called legacy system), which is too far away from the OO paradigm, the use of DP is not the most appropriate solution. Business Rules (BR) approach is another technique introduced for the same reason, which is not so closely related to OOP and seems to be more convenient for reengineering of legacy systems, written in COBOL. The theory and practice of business logic extraction in form of BR and using them for improving the design of the legacy system are still under development. This paper is an attempt to adapt some ideas from DP approach in order to facilitate the process of business logic extraction from programs, written in COBOL.

KEYWORDS

Legacy software, software reengineering, design patterns (DP), DP catalogs, design motif, extraction of design motifs, business rules.

1 INTRODUCTION

There are three main topics, which are the basis for the work presented in this paper – the modernization of legacy software systems, the Business Rules approach for design and redesign of software system and last, but not least, the language phenomenon known as COBOL. Let us briefly present these three topics, giving only of the key information, which is necessary for understanding the concept.

1.1 Legacy Systems

Legacy software systems are usually written in old fashioned programing languages and on old fashioned platforms that are not maintained anymore. Each system that is used long enough becomes, soon or later, a legacy one. But the systems, which have been used for many years, are successful, stable and helpful [1]. That is why it is worth to try to transform them to modern platforms, revealing and preserving the knowledge and business logic, built into them. In [2] the modernization that "… improves capabilities and maintainability of a legacy system by introducing modern technologies and practices ..." is called *reengineering* and 5 kinds of reengineering are identified – *retargeting*, *revamping*, *code reduction*, *code translation* and *functional transformation*.

The variety of situations, in which the modernization is unavoidable, is so rich, that it is impossible to decide in advance which approach is the best one to follow. But it is clear that most of the modernizations of legacy systems will be accompanied with some (smaller or bigger) changes of their data model and/or functionalities.

In [3] we considered the different kinds of reengineering, in order to identify the stages each of them passes through. Our observations have been that all types of reengineering, excluding retargeting (i.e. moving the legacy system into a new hardware platform, having a translator for the source code language and the necessary DBMS), need some *analysis of the source code* of the legacy system [4]. Recently, "digging" existing pro-

gram code becomes so popular that some researchers are inclined to speak for a new subdomain of Software engineering – the *Software archeology* [5].

1.2 Business Rules approach

In the last years of the past century, a new approach for specifying software systems was proposed by the so called Business Rules Group [6]. As main goals of the project have been stated:

- to define how to apply business rules (BR) to information systems design;
- to define and describe BR and associated concepts;
- to provide a rigorous basis for engineering new systems based on formal definitions of BR.

Beside these three main goals the authors of the approach formulated the following goal also:

- to provide a rigorous basis for reverse engineering, using the BR extracted from existing systems.

Nowadays BR is considered as one of the fruitful concepts for software specification.

In order to assure an efficient process of modernization, any changes and updates of the system have to be made in an easy and natural way. *Business rules approach* has the potential to provide such an efficient way for reengineering of the legacy systems. The essence of the idea is the following: Given the source code of the legacy system, to extract automatically the business logic from it in the form of BR. All necessary changes and updates should be made on the extracted rules and then the new system can be implemented, following the BR approach, using the updated BR as specifications.

Nowadays the BR approach is very popular and it is a successful technique for specifying software systems. There are some attempts for automatic extractions of BR, too [7]. But the idea to use BR for reengineering of legacy systems is still under development and is considered as a "blue sky" research topic.

Recently, a team of researchers, including the authors of the paper, is working on a scientific project. The main goal of the project is to propose some feasible methods and tools for automatic extraction of BR from the source code of a legacy system, so as to recover the business logic built in it. Different approaches have been investigated in order to elucidate and clearly define some recommendations how the automation of the BR extraction should be accomplished. One of the directions of our study is the implementation of software systems on the basis of *design patterns* and the possibility to use the design patterns approach in the process of extracting business rules. The current paper comprises some promising results in this direction.

1.3 The phenomenon COBOL

Giving to COBOL the label phenomenon is not an exaggeration. We have very serious reasons for this. Despite the strong, more than 40 years domination of the professional community by C language, and despite the efforts of Java-fans to make J-technology dominating in commercial software, some statistical data shows that 60% of the corporate software all over the world is written in "COBOL". Why we call COBOL phenomenon and even put the name in quotation marks? Let us outline briefly the current status of this software instrument.

The "language" COBOL (Common Business Oriented Language) was created in the end of 50-es of the past century to facilitate development of business software. The creators of the instrument intended to make it a versatile tool, giving the possibility each user to be able to specify the computation she/he needs. The first official standard specification of the language was COBOL 60.

The attempt to make COBOL "language" as close as possible to the natural language led to unusual for wide spread programming languages list of about 250 reserved words (for comparison, in kernel C language these words are about 30). Something more, the authors of compilers from COBOL had the freedom to deviate from the standard in their implementations. COBOL-standards itself have been changed many times. Standard COBOL 60 was soon replaced by COBOL 61 and COBOL 61 Extended. Then many other official standards were launched – COBOL

68, COBOL 74, COBOL 85, COBOL 2002, COBOL 20XX, etc.

Finally, extremely large number of unofficial dialects appeared. For example, the page for COBOL in Wikipedia [8] contains a list of more than 25 dialects. As a result of a brief search in Internet we found at least 11 different dialects of the language, developed only by IBM – IBM OS/VS COBOL, IBM COBOL/II, IBM COBOL SAA, IBM Enterprise COBOL, IBM COBOL/400, IBM ILE COBOL, VS COBOL II, COBOL-370, ENTERPRISE COBOL, COBOL for MVS & VM, and Visual Age COBOL.

It is not realistic to believe that all this variety of dialects could be maintained in a single software company. Working on the project, we find out that one of the most difficult problem in extracting BR from legacy system, written in COBOL, will be to recognize the dialect used and to recover its formal grammar. In [9] we discussed an approach for syntax analysis of programs written in some dialect of COBOL. The goal of the current paper is a bit different and it is stated below.

1.4 Paper content

The main goal of this work is to identify some elements of the design patterns in COBOL for which we will use the notion *motif.* We expect to assign to each such motif some interpretation as element of a BR and to use the identified motifs in the process of extracting BR.

It is impossible to expect that design pattern approach could be applied on programs, written in COBOL, the same way as on the programs written in the modern OO languages (C++, Java, etc.). I.e., we do not expect to elaborate one *catalog* of COBOL patterns, which will be relatively complete and widely accepted by the community. The explanation is that nowadays there are no so many people that are familiar with some versions of the language COBOL and have enough programming experience to suggest patterns. By our opinion, the combination of deep knowledge and practical experience in COBOL use is absolutely necessary in order to elaborate a good, both versatile and efficient, catalog of COBOL design patterns.

But there is a second, much more serious reason for skepticism. The style of programmers of 60^s and 70^s, (if it is possible to speak for programming style in that time!), is rather "artistic" compared with the almost "industrial" style of the OO-programming, applied recently. It is very difficult to shape inside 20-30 patterns the programs of this "romantic" period of computer programming. That is why our ambition is to develop for each application which is a subject of reengineering a set of COBOL design motifs, to assign to each of them a corresponding BR element and to try to compose from such elements the complete business rules.

In Section 2 we will introduce briefly the design pattern approach for development of the software and will discuss the advantages and disadvantages of the approach from the investigated perspective – BR extraction. The notion *design pattern* and *design motif* are introduced, discussed and redefined in accordance with our purposes. Section 3 describes briefly the program code, used for making experiments in our research, and a small software instrument, created especially for this study.

In Sections 4, 5 and 6 we present our point of view on the process of identifying design motifs, influenced by samples of program code. In Section 4 the identification of a design motif is influenced by set of samples from a popular in the COBOL-age book [10].

The samples that influenced the extracted in Section 5 and Section 6 motifs are taken from a real legacy system. In Section 5 a „cyclic" design motif is identified. We consider it a very important motif for digging COBOL source codes, because the language COBOL does not have the modern loop constructs `for`, `while` and `do … while`. In Section 6 another, very frequently used in the software systems design motif is identified. This motif is linked with the execution of sub-programs external for the considered program . It really "hides" the implementation in its semantic and will be crucial for extracting of BR from COBOL code.

For all of the above mentioned three motifs, considered as significant for our study, some parts of

program code are given first, which suggest the structure of the motif. Then each of the motifs is presented with pseudo-code. Finally, for each of the motifs a finite automaton is constructed, which can be used in the process of recognizing of the motif.

Giving some samples of code in COBOL in the text of paper, we will skip the first 6 positions of each line, dedicated usually for numbering of the lines or something else. So, the first shown position of the line is the 7th and the character `'*'` in this position means that the line contains a comment. All parts of the COBOL statements that are important for our analysis are given explicitly. The other parts are omitted and replaced with `"…"`. The reserved words of COBOL are bolded for clarity.

Some results, obtained till now, from the work of the recognizer on a real software system, written in COBOL, are given in Section 6. Section 7 contains some conclusions and a few propositions for future research and development work.

2 DESIGN PATTERNS AND MOTIFS

In the field of Software engineering a *design pattern* (DP) is considered as a general reusable solution to a commonly occurring problem within a given context [11]. The initial idea of *DP approach* is that a design pattern is a formalized best practice, which should be transmitted directly to the software application under development, has been further enriched with many other meanings, appropriately defined for a variety of problems. One possible classification of such problems, addressed by patterns, is according to the number of criteria, e.g. phase, process or a particular activity of software development; application domain, structural relationships, etc.

Essential for the DP approach is the development of a catalog containing a couple dozens of design patterns which are widely used in development of OO systems [12]. Recently many researchers evaluate the constraints of the development process, based on use of design patterns from the catalog of GoF (under this abbreviation are popular the authors of [12]) and consider such development as too restrictive. They propose usage of more simple constructive elements – *design subpatterns* [13] and *design motifs*, dedicated to make the DP approach more flexible.

DP approach seems convenient for extraction of BR, but it is strongly connected with the OO programing paradigm. So it is impossible to use it directly in case of languages such as COBOL which is too far from OOP. We think that it is worth to try to adapt the concepts and ideas of DP approach to our goals. Especially for the activity under consideration – business rules extraction from source code of a legacy program, we will give the following definitions:

Design motif is a continuous piece (slice) of code that is semantically integral and valuable. So it could be a part of a pattern and could be separately searched as an item for further goal-oriented analysis and embedding in the pattern templates.

Design pattern is a code template, composed of two parts. First of them is relatively constant part of code, not necessarily consecutive, identified syntactically but without its own semantic meaning. The second part is a variable set of motifs which have to fill the template of the pattern and to make it semantically valuable. Patterns should be also searched, saved and documented as potential carriers of significant part of one or more business rules, built in the code.

Next sections contain a brief description of our attempts to create a tool for design motifs recognition in programs, written in COBOL.

3 AN EXPERIMENTAL PROGRAM CODE AND A RECOGNIZER

As a base of experiments for described in this paper approach we are using a very large corporative system, written in a dialect of COBOL that could be identified by the mentioned in some of the programs options COBOL2 and DB2. The system consists of 1002 programs and comprises about 2 500 000 lines of code. It will be referred further in this paper as the system.

In order to be able to identify some interesting places in such huge amount of code we developed an instrument called further *recognizer*. The recognizer is a small lexical analyzer that uses a built-in finite automaton as specification of a language. It receives as input programs and give as output code slices recognized by the automaton.

The alphabet used by the different automata, in the beginning, is composed of the reserved words of the COBOL dialect, described in [10] as well as some auxiliary tokens as <label>, <name>, <number>, <string>, <picture>, etc. For simplicity we will replace them in figures with a token <any> which means "not a reserved word". Real automata used in our research are more complex. For simplicity too, on the figures are not shown rejection states as well as the transition that start the search from the beginning.

In the process of work some new reserved words were appended, because they were used in the software system, but were not included in the list of reserved words of Mini COBOL, presented in [10].

For the moment, each specific automaton is embedded in the source code of the recognizer as a two dimensional array. The rows of the array are labeled by the states of the automaton and the columns – by the letters of a specific subset of the alphabet.

4 INPUT DESIGN MOTIF

Our first idea is to use some introductory educational materials for programming as an origin for design motifs. In this activity, even in the early years of programming, could be found elements that are similar to design patterns. It is quite natural for teachers in programming to use in their classes some carefully prepared pieces of code to demonstrate the syntax of the programming language, its semantic, as well as various programming techniques. By analogy with the teaching of piano (or some other musical instruments) , we will call them *etudes*. We suppose that a well prepared *system of etudes* could be a base for elaborating efficient and usable set of design motifs to be used in the process of reengineering of legacy systems, written in COBOL.

Here we will consider a set of etudes from the popular book [10]. The book was famous in the 70^s of the past century, because it contains specification of a subset of COBOL which seems to be common for many COBOL dialects and which is called by the author QUICK COBOL. For our purposes we will use the etudes of Chapter 2 of [10] which the author obviously consider as fundamental for the future COBOL programmers. In this chapter there are two unnamed etudes that we will call UONE and UTWO and other six etudes, named ALPHA, BINGO, CHARLY, DOG, EASY and FOX. We will use these names when we want to refer to the corresponding etude.

4.1 Examples of data input with processing

Looking at the mentioned above etudes we could say that they are unified around chronologically first activity of almost each COBOL program – the input of the external data to be processed by the program, including the processing itself. We will use this fact to demonstrate in detail how we could proceed when trying to identify a motif.

In etudes of Chapter 2 we find the following input *paragraphs*:

```
* UONE PROGRAM
 START.
    READ … .
    IF … GO TO FINISH.
    …
    GO TO START.
 FINISH.

* UTWO PROGRAM
 START.
    READ … .
    IF … GO TO FOUND.
    IF … GO TO MISSED.
    GO TO START.
 FOUND.
 …
 MISSED.
 …

* ALPHA PROGRAM
 START.
```

```
    OPEN INPUT … .
    OPEN OUTPUT … .
 MAIN-LOOP.
    READ … AT END GO TO FINISH.
    …
    GO TO MAIN-LOOP.
 FINISH.

* BINGO PROGRAM
 START.
    OPEN INPUT …, OUTPUT … .
    …
 PROCESS.
    READ … AT END GO TO FINISH.
    …
    GO TO PROCESS.
 FINISH.
    CLOSE … .

* CHARLY PROGRAM
 START.
    OPEN INPUT …, OUTPUT … .
    …
 PROCESS.
    READ … AT END GO TO FINISH.
    IF … GO TO PRINT.
    GO TO PROCESS.
 PRINT.
    …
    GO TO PROCESS.
 FINISH.
    CLOSE … .

* DOG PROGRAM AN EASY PROGRAM
* INPUT PARAGRAPHS ARE SIMILAR
* TO PARAGRAPH OF CHARLY PROGRAM

* FOX PROGRAM INPUT PARAGRAPHS
* ARE SIMILAR TO PARAGRAPHS
* OF BINGO PROGRAM
```

From program parts, shown above we could conclude that data input usually consists of four paragraphs:

- START: opens the input and output files and makes the initialization of necessary variables;
- PROCESSING: performs some processing on the just read piece of data;
- PRINT: outputs some results obtained by the processing the current piece of data;
- FINISH: performs some final processing of the data, outputs the final results and closes the input and output files.

These four paragraphs are interrelated as shown at the Fig. 1.

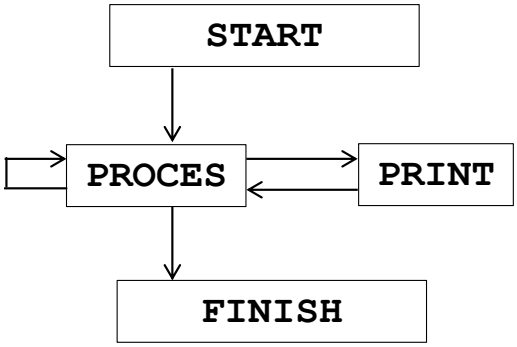

Figure 1. Interrelations of the paragraphs of the motif READ_PROCESS

4.2 The motif READ_PROCESS

Using the outlined structure of the paragraphs related to the input and eventual processing in parallel with reading, we will define the following design motif, called **READ_PROCESS:**

```
<label_1>.
    OPEN INPUT <var>, OUTPUT <var>.
    <init>
 <label_2>.
    READ <var> AT END GO TO <label_4>.
    <process>
    IF <condition> GO TO <label_3>.
    GO TO <label_2>.
 <label_3>.
    <print>
    GO TO <label_2>.
 <label_4>.
   <processing>
   CLOSE <var>.
```

It is obvious that the motif READ_PROCESS, outlined above, is a regular expression over the specific alphabet:

A_1 = {**OPEN,READ,GO TO,IF,CLOSE,**<any>}.

Letter <else> is corresponding to the universal token that eliminates everything not important for the recognition of the expression.

The diagram of the automaton that recognizes this regular expression is shown on Fig. 2. The diagram is simplified as it was mentioned above – it is without rejecting states and transitions that return in the beginning of the search.

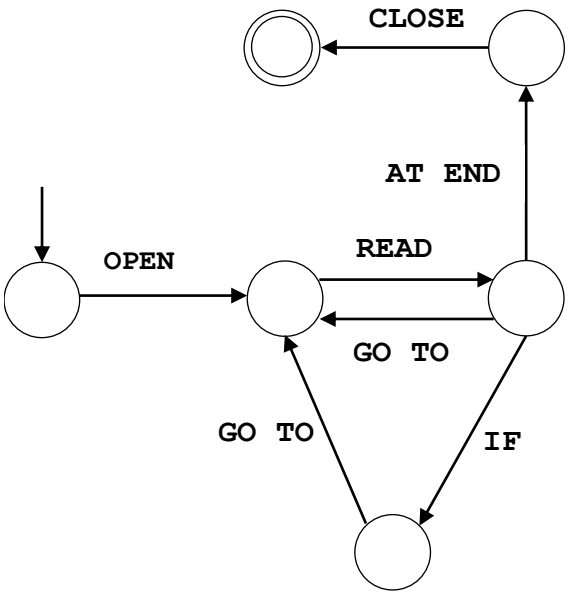

Figure 2. Finite automaton of the motif READ_PROCESS

5 CYCLIC DESIGN MOTIF

Our second attempt to identify a pattern motif is inspired by the fact, that in COBOL there is not any modern loop construct (`for`, `while` and `do …while`). The iterative procedures in COBOL are organized by branching operators – conditional or unconditional (the forgotten in the modern programming operator `GO TO`).

It is well-known that reading and understanding such code ("spaghetti"-code) is a true challenge. This makes the heavy task for extraction of the embedded in such code business logic even harder. The real problem with the loops is very difficult. As we underlined in [14] business rules, as a tool for specification of software, do not contain any cyclic construction. And this is because non-programmers do not use such constructs when describing the things. That is why we have to dedicate special attention to extraction of cyclic slices of the code and their *re-functionalization* (see [15]).

Looking at the considered in previous Section etudes, we could find many cyclic constructions there. We could generalize these cyclic constructions with the following piece of code:

```
* LOOP
 START.
    READ … AT END GO TO FINISH.
    <processing>
    IF … GO TO LABEL1.
    <processing>
    IF … GO TO LABEL2.
    <processing>
    …
    <processing>
    IF … GO TO LABELn.
    GO TO START.
 FINISH.
```

Of course, the operator **`READ`** with the phrase **`AT END GO TO`** is shown separately just because we use such samples. It is quite possible to consider this operator as a special kind of **`IF`** … **`GO TO`** operator and to include it in the sequence of such operators. Where in code are the labels of **`GO TO`** phrases – in paragraphs before `FINISH` or in paragraphs after, it does not matter.

Looking for samples of cyclic constructions in the programs of the system, we found such cyclic piece of code:

```
      * ANOTHER LOOP
       L00.
          READ … AT END … .
          IF …
             …
             IF …
                …
                GO TO L00
             ELSE …
```

The brief look at it shows that it is not significantly different from the outlined above generalization of the cyclic construction of etudes from [10]. So we could consider that this construction:

```
* LOOP
 START.
    <processing>
    IF … GO TO LABEL1.
    <processing>
    IF … GO TO LABEL2.
    <processing>
    …
    <processing>
    IF … GO TO LABELn.
    GO TO START.
 FINISH.
```

identifies the design motif LOOP. This motif is also a regular expression over the specific alphabet:

A_2 = {**GO TO**,**IF**,<any>}.

The corresponding simplified finite automaton is shown on Fig. 3.

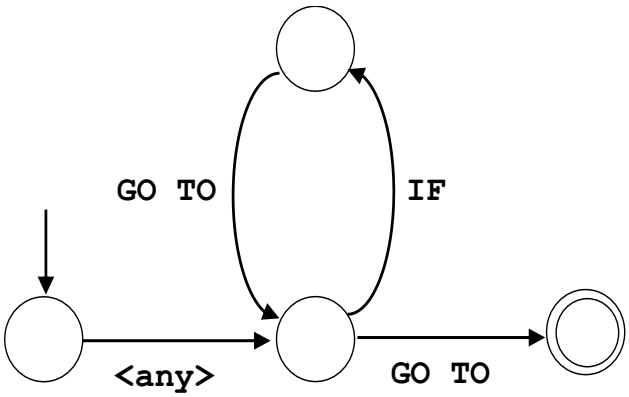

Figure 3. Finite automaton of the motif LOOP

6 CALL EXTERNAL DESIGN MOTIF

Our third attempt to identify a pattern motif is influenced by the fact that we could observe many times almost identical sequences of operators which obviously could be a motif. For example, in a single program we obtained the following three similar sequences:

```
* SEQUENCE 1
 MOVE 'AAAFS15' TO PPNAME WCOM-PROGNAME.
 MOVE SPACES    TO DCWDTEL.
 MOVE 'R'       TO DTEL-TIPORICH.
 MOVE 'G'       TO DTEL-DATATIPO.
 MOVE 'AAAFS15' TO DTEL-SIGLDATA.
 MOVE 'SCADTEL' TO PPCALL.
 CALL PPCALL    USING DCPARM  DCWDTEL.

* SEQUENCE 2
 MOVE 'D44-'    TO WCOM-CODABEND
 MOVE 'LINK'    TO WCOM-FILEFUNZ
 MOVE 'SCBCB10' TO WCOM-FILENOME
 MOVE SPACES    TO WCOM-FILECTRL
 STRING CAMB-CODDIVIS ' ' CAMB-CODCAMBI
        ' ' CAMB-CODDRIFE DELIMITED BY
        SIZE INTO WCOM-FILECTRL
 MOVE 'SCBCB10' TO PPCALL
 CALL PPCALL    USING DCPARM DCWCAMB

* SEQUENCE 2
 IF A531-FLEURUSA = 'U'
    MOVE '.'    TO WNUMVIRG
    MOVE ','    TO WNUMPUNT
 ELSE
    MOVE ','    TO WNUMVIRG
    MOVE '.'    TO WNUMPUNT.
    MOVE 'SCBCNUM' TO PPCALL.
 CALL  PPCALL   USING DCPARM DCCNUM.
```

The operator CALL is not a part of all COBOL dialects. But even in COBOL for AIX v. 2.0 [16] which is very close to the dialect of our experimental system this operator has a different syntax. Any way, it is sure that this is the operator for execution of an external program or a paragraph of external program. Assignments before the call of external programs are passing of the necessary parameters. So, we could specify a design motif CALL_EXTERNAL as follows:

```
* CALL_EXTERNAL
 <passing parameters>
 IF …
    <passing parameters>
 ELSE
    <passing parameters>
 <passing parameters>
 …
 IF …
    <passing parameters>
 ELSE
    <passing parameters>
 <passing parameters>
 CALL  PPCALL   USING DCPARM DCCNUM.
```

This motif is also a regular expression over the specific alphabet:

A_3 = {**IF**, **ELSE**, **CALL**, <any>}.

The corresponding simplified finite automaton is shown on Fig. 4.

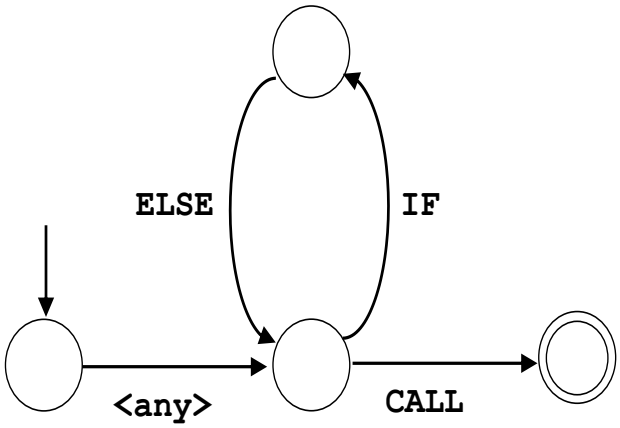

Figure 4. Finite automaton of the motif CALL_EXTERNAL

7 RESULTS OF RECOGNIZING

Using a recognizer written for the purposes of this task we obtained some interesting results. The design motif `READ_PROCESS` was not found at all in the huge amount of code. We could give essential reasons for this. One reason is that etudes in text books, not only for COBOL, are rather elementary and do not correspond to real circumstances and needs of a complex software system.

Especially for the motif `READ_PROCESS` it is obvious that chances to find it in a complex system as the one used for the experiments, are close to zero because data in such systems are kept in data base and are manipulated, including their input, with the tools of the corresponding DBMS (DB2 in our case).

Anyway, design motifs like `READ_PROCESS` could be helpful in reengineering process. They could be found in stand-alone instrumental programs that are dedicated to transform data from one form to another or to perform some ad hoc defined processing of the data that was not included in the system functionality. For legacy systems, which are in exploitation long years, appending of such auxiliary programs during the years is inevitable.

The other two identified above motifs, were found in the system used for experiments because they was influenced by the system. But the frequency of two motifs is absolutely different. The motif LOOP was found only in some of the programs of the system and most frequently once or maximum two times. The reason for this, by our opinion, is the mentioned above usage of DBMS in the system.

A corporative system is not performing heavy numeric calculations of iterative nature, neither sophisticated combinatorial algorithms with too many loops. Practically all procedures of cyclic character are "hidden" in calls of SQL operators. This is a very positive fact because of mentioned above problem with extraction of business logic of code with loops. Lack of too many loops in the legacy code will make the process of extraction of BR easier.

The frequency of meeting of the motif `CALL_EXTERNAL` is too large – 3-5 times in a separate program. This means that the system was developed by a team of programmers that have the possibility to use a powerful library of external subprograms. It is impossible to call them "standard" because there are numerous dialects of COBOL, and probably, they have their own libraries of subprograms. Even more: each team of programmers depending of member's experience and technology used, might create own libraries of external programs. Such libraries could be with entirely different content for the different software projects.

This fact is strongly negative. If the reengineering team possesses only the source code of the system without any description of the used external subprograms, the extraction of the business logic will be impossible.

8 CONCLUSIONS

The current paper presents our investigation of the possibilities to apply ideas from DP approach to extraction of BR from legacy code as well as to propose a practical tool for identifying design motifs in COBOL programs. The results of our work show that:

- It is possible to identify helpful design motifs in the code and to search them automatically. Even a simple instrumental tools as the recognizer used in this research could be very helpful;
- The probability the design motifs identified in one application to be useful in another one, is very low. It seems that identification of motifs for each new reengineering process have to start from the beginning;
- Languages like COBOL give the programmers possibilities to make the code intractable from the reengineering point of view.

Our ideas for further research and development work are:

- To continue development of the methodology for identifying design motifs and search for the identified motifs in the code;
- To continue development of tools for automation of the process of finding design motifs in programming code;
- For a given legacy system to propose some methods and corresponding tools for including frequently found design motifs in potential design patterns;
- To develop an incremental recognition of motifs and patterns in programs, written in different and even unknown COBOL dialects.

9 ACKNOWLEDGEMENTS

This work is supported by the National Scientific Research Fund of Bulgaria under the Contract ДТК 02-69/2009.

10 REFERENCES

[1] Semantic Designs, "Legacy Software Migration", http://www.semdesigns.com/Products/Services/LegacyMigration.html, visited March 2014.

[2] R.C. Seacord, D. Plakosh, and G.A. Lewis, "Modernizing Legacy Systems. Software Technologies, Engineering Process, and Business Practices". Addison Wesley, 2003, ISBN 0-321-11884-7.

[3] Kr. Manev, and N. Maneva, "Using the Source Code for Modernization of a Legacy System", Proc. of 9th Annual Intern. Conference CSECS, Fulda, 2013, pp. 113-118.

[4] D. Binkley, "Source Code Analysis: A Road Map". In: Briand, L., Wolf, A. (eds.) Future of Software Engineering (Proc. of FOSE'07), IEEE-CS Press , 2007, pp. 104—119.

[5] Hunt, A., Thomas, D., "Software Archaeology", IEEE Software, vol. 19, no. 2, 2002, pp. 20-22.

[6] D. Hay, K. A. Healy (eds.). "Defining Business Rules ~ What Are They Really?", GUIDE Business Rules Project Final Report, rev. 1.3., July, 2000.

[7] I. Baxter, S. Hendryx, "A Standards-Based Approach to Extracting Business Rules", Presentation for OMG's Architecture Driven Modernization Workshop, Semantic Designs, 2005. http://www.semdesigns.com/Company/Publications/ExtractingBusinessRules.pdf, visited March, 2014

[8] COBOL, http://en.wikipedia.org/wiki/COBOL, visited March, 2014

[9] Kr. Manev, H. Haralambiev, and A. Zhelyazkov, Approaches to Legacy Software Parsing, Proc. of the ISGT'2013 Conference, Sofia, pp. 269-274.

[10] L. Coddington, "Quick COBOL", McDonalds, London and American Elsevier, New York, 1971.

[11] http://www8.cs.umu.se/~jubo/ExJobbs/MK/patterns.htm visited March, 2014.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of reusable object-oriented software". Professional Computing Series. Addison Wesley, 1995.

[13] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato. "A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition", IEEE Proceedings of ASWEC, Brisbane 2005, pp. 262-269 .

[14] Kr. Manev, N. Maneva, H. Haralambiev. Extracting Business Rules through Static analysis of the Source Code. Proc. of the 41-rd Spring conference of the UBM, 2012, pp.263-270.

[15] Kr. Manev and T. Trifonov, Declarative Semantics of the Program Loops, Proc. of the ISGT'2012 Conference, Sofia, pp.326-337.

[16] "COBOL for AIX. Reference manual", IBM, 2004.