Code Reusability Mechanisms

Xavier Castellani and Hong Jiang

CEDRIC-IIE (CNAM) research laboratory, 18 allée Jean Rostand, 91025 Evry Cedex, France
Tel 33 1 69 36 73 50 - Fax 33 1 69 36 73 05, castellani@iie.cnam.fr, jiang@iie.cnam.fr

Abstract

This paper describes standardized reusability mechanisms of program code using a metric and a guideline. These mechanisms find five results of code reusability which allow us to merge programs, to store programs in a library and to include them in other programs, and to store common sequences code of programs in a library and to include them in programs. These mechanisms show how programs can be built in a standardized manner on using a metric and a guideline.

These mechanisms are based on the standardized reusability mechanisms of objects of the object-oriented analysis and design methodology MCO. They can be used with any programming language.

Keyword codes: D.2.3; D.2.6; D.2.8; D.2.10
Keywords: Program design. Reusability. Metrics. Guideline.

1. INTRODUCTION

According to Nauer and Randell [14], *software reuse is the process of creating software systems from existing software rather than building software systems from scratch.*

According to Meyer [13], *reusability is the ability of software products to be reused, in whole or in part, for new applications.*

The software reuse in developing new software is a well known field in software engineering for about twenty years [11]. There is an increasing need for a corresponding maturity in reutilisability, assessment and measurement. The assessment of a component's quality and reutilisability are critical to a successful reuse effort. Components must be easily comprehensible, easily incorporated into new systems. Unfortunately, no consensus currently exists on how to go about measuring component's reutilisability [3].

The success of reuse technology depends on the integration of reuse libraries into the design and programming environments. To reuse software, it is necessary to have specifications of the software. Without these specifications the reuse is impossible. Furthermore, it is very inefficient to look through manually the specifications [6]. When the number of reusable components is quite big, this is not practical. Therefore, reutilisability mechanisms and on-line searching mechanisms are required. With the support of an on-line searching tool [2], software can be shared easily.

The idea of using metrics to normalize program code reusability has been proposed by several specialists. We can mention Cox [7] and groups of the Annual Workshop on Software

Reuse [16]. To our knowledge, it does not exist rational utilization of metrics to normalize the reusability of programs.

This paper presents a solution to normalize the reusability of programs with mechanisms which use a metric and a guideline. These mechanisms find five results of code reusability which allow us to merge programs, to store programs in a library and to include them in other programs, and to store common sequences code of programs in a library and to include them in programs.

The mechanisms of code reusability allow us to find common programs and common sequences in different programs to store them in libraries and to include in programs. The common sequences code and the particular sequences code of two programs are presented in Section 2. The five results of program code reusability are presented in Section 3. Metrics to measure program code are presented in Section 4. The mechanisms of code reusability are presented in Section 5.

2. COMMON SEQUENCES CODE AND PARTICULAR SEQUENCES CODE OF TWO PROGRAMS

A sequence code is a set of lines of code. We distinguish *declaration sequences (DS) which are data declarations and the other declarations: declarations of macros, of "include", of "copy", etc. which can be used, from processing sequences (PS) which can be performed.*

The common declaration sequences of two programs are the data declarations of the same types and the other declarations which are identical: declarations of macros, declarations of "include" or of "copy", etc.

Data declarations of the same types may be declarations of data which have not the same names. Common declarations sequences may be not contiguous sequences and may be not ordered with the same order. Numerous programs have common sequences. It is in particular the case for input/output programs.

The particular declaration sequences of two programs are the declarations which are not common declaration sequences.

The common processing sequences of two programs are processing sequences which provide the same processing.

Two processing sequences provide the same processing if their specifications are syntactically identical or if a tool or an analyst must certify that they ensure the same processing. It is especially necessary to study processing sequences that use named data differently but of the same types, and to study processing sequences not specified with the same statements, for example "case" statements specified with "if" statements.

The common processing sequences may be not contiguous sequences. For example in Cobol, a processing sequence may be defined with statements using "Perform" blocks and these blocks.

Common processing sequences may be not ordered with the same order. For example "Allocation" statements, blocks of the different cases of "case" statements and parallel blocks of statements.

The particular processing sequences of two programs are the processing sequences which are not common processing sequences.

3. THE FIVE RESULTS OF PROGRAM CODE REUSABILITY

In this Section we apply the principle of the object reusability mechanisms [5] to program reusability. These object reusability mechanisms allow the creation of object merges, simple inheritances and abstract objects. The five possible results of program code reusability between two programs P-a and P-b presented in Figure 1 are deduced from the results of object reusability .

1 - P-b is identical to P-a.
2 - P-b is merged with P-a.
3 - P-a is stored in a library and is "included" in P-b.
4 - P-b is stored in a library and is "included" in P-a.

---

5 - One or several common sequences existing in both programs P-a and P-b are stored in a library and are "included" in P-a and in P-b.

Figure 1: The five results of reusability between two programs

For the results 2, 3, 4, the particular sequences of P-b and of P-a (result 2), or of P-a or P-b (results 3 or 4) are defined in case sequences and the program is parameterized to perform these particular sequences.

For the result 5, one or several common sequences existing in both programs P-a and P-b are stored in a library and are "included" in P-a and in P-b. This result allows us to create separately declaration sequences from processing sequences.

The result 5 allows us to define common sequences in two programs. We do not present the study of identical sequences of a same program.

To simplify, we say here that a program is stored in a library and is "included" in another program. According to the used language, this mechanism is obtained with the statements "Include" in C or Pascal, "Copy" in Cobol, or by defining sub-program, or with other mechanisms.

The five results of reusability are defined between two programs which have the most important common sequences. The reusability between several programs is studied successively between these programs considered two by two. For example between two programs P-a and P-b, P-b is stored in a library and is "included" in P-a. After, the reusability is studied between two programs P-a and P-c, P-c is stored in a library and is "included" in P-a.

4. METRICS TO MEASURE PROGRAM CODE

Numerous software experts have proposed metrics to measure software components. Some of them are mentioned by Fenton [8]. The simplest unit to measure source program code length is Lines of Code (LOC). Others prefer the Used Instructions (UI): the COCOMO method (COst COnstructive MOdel) [4] of Boehm for example. Others use the Function Points (FPs) [1]. However, most companies use LOC in spite of their deficiencies [9].

Software engineering specialists have defined other metrics than the sizes of the programs to measure the quality of the software. Among the most used the metrics to measure numbers of operators and numbers of operands, mention the metrics of Halstead [10], the metrics to measure program control graphs, mention the metrics of McCabe [12] and the metrics to measure program call graphs, the hierarchical complexity which is the average number of modules by level.

In this article we use the sizes to measure programs code and we measure these sizes with lines.

5. MECHANISMS OF CODE REUSABILITY

5.1. Guideline of Code Reusability

A code reusability guideline fixes for two programs:

. the minimum size of their common sequences (MinCL),

. and the maximum size of the particular sequences of a program with regard to the other (MaxPL),

in order that these two programs may be merged, or one may be stored in a library and "included" in the other.

For two identical sequences of code, MinCL fixes their minimum size in order that one of them may be stored in a library and "included" in their programs.

- <u>Which code reusability guideline should we choose?</u>

It is not difficult to define a code reusability guideline in an information department because analysts and developers have precise ideas of their values.

Generally, the average size of a written in C, Pascal, Cobol, ..., is comprised between 800 and 3000 lines; some programs even pass beyond 5000 lines. With these languages, the sequences which are stored in libraries to be reused in several programs with the statements "Include" in C or Pascal, "Copy" in Cobol, ..., have generally minimal sizes comprise between 30 and 100 lines according to the programming recommendations: "it is forbidden to store too little sequences in a library". These minimal sizes (30 and 100 lines), are possible values of MinCL. MinCL must be superior to one line. With non object-oriented programming languages the masking is not usable, hence MaxPL must be count in tens lines (and the particular sequences are defined in case sequences).

5.2. Presentation of Mechanisms of Code Reusability

The code reusability mechanisms optimize the reusability of program code with the following criteria:

. a similar program code reused in several programs and satisfying the reusability guideline constraints must be defined once in a library,

. a sequence code reused several times in programs and satisfying the reusability guideline constraints must be defined once in a library,

. the reusability must be made in priority with programs otherwise with sequences,

. and the number of programs and of sequences reused must be minimized.

The mechanisms of reusability of a new program P-b with an existing program P-a which has the most lines in common with P-b, are defined by the algorithm presented in Figure 2. The five results of this algorithm are presented in Section 3.

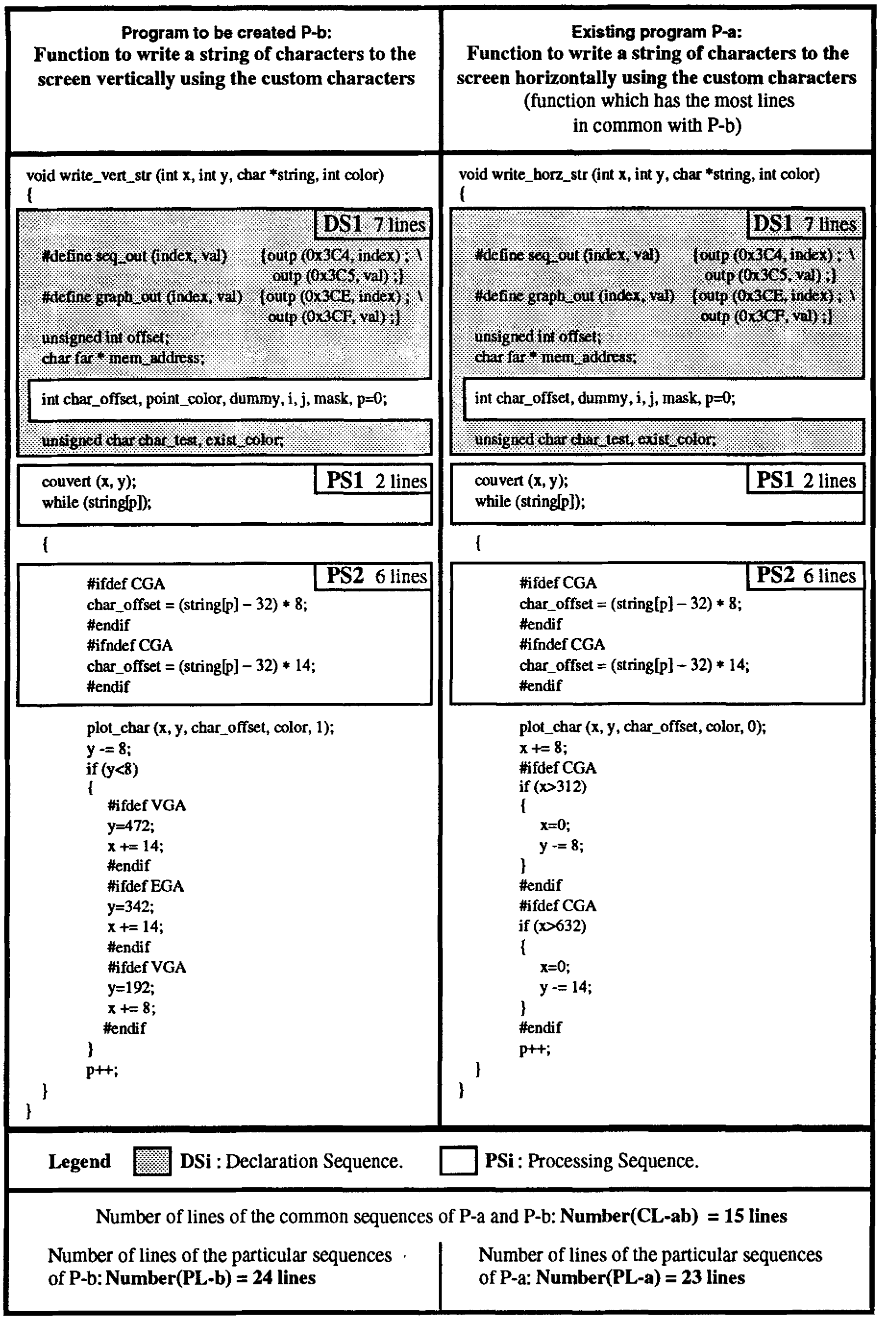
```
If the code of P-a and the code of P-b are identical
   then P-b is identical to P-a
   else If number of lines of common sequences of P-a and P-b: Number(CL-ab)
           is superior to MinCL
           then If number of lines of the particular sequences of P-a: Number(PL-a)
                   is inferior to MaxPL
                   then If number of lines of the particular sequences of P-b:
                           Number(PL-b) is inferior to MaxPL
                           then P-b is merged with P-a
                           else P-a is stored in a library and is "included" in P-b
                        Endif
                   else If number of lines of the particular sequences of P-b:
                           Number(PL-b) is inferior to MaxPL
                           then P-b is stored in a library and is "included" in P-a
                           else For each common sequence of data declarations
                                         or of processing statements of
                                         P-a and P-b: Number(CL-ij)
                                    If Number(CL-ij) is superior to MinCL
                                       then The common sequence will be stored in a
                                            library and are "included" in P-a and in P-b
                                       else No reusability between P-a and P-b
                                    Endif
                                Endfor
                        Endif
                Endif
           else No reusability between P-a and P-b
        Endif
Endif
```

Figure 2: Algorithm to study the reusability between two programs P-a and P-b

5.3. Example of Application of Code Standardized Reusability

Two C functions presented in [15] are in Figure 3:
- a function to be created: Write_vert_str which writes a string of characters to the screen vertically, beginning at a specified pixel position (x, y) and having specified foreground and background colors,
- and an existing function: Write_horz_str which writes a string of characters to the screen horizontally, beginning at a specified pixel position (x, y) and having specified foreground and background colors.

Write_horz_str is the existing function which has the most lines in common with Write_vert_str. It is similar to the previous function. It is a particular case but these functions are short and can be wholly presented in this paper. The common declaration sequences of these functions and their common processing sequences are contiguous except DS1, and are ordered with the same orders. We study the reusability of these two functions on considering the lines of code.



Program to be created P-b:
Function to write a string of characters to the screen vertically using the custom characters

```
void write_vert_str (int x, int y, char *string, int color)
{
                                                        DS1 7 lines
  #define seq_out (index, val)      {outp (0x3C4, index) ; \
                                      outp (0x3C5, val) ;}
  #define graph_out (index, val)    {outp (0x3CE, index) ; \
                                      outp (0x3CF, val) ;}
  unsigned int offset;
  char far * mem_address;

  int char_offset, point_color, dummy, i, j, mask, p=0;

  unsigned char char_test, exist_color;

                                                        PS1 2 lines
  couvert (x, y);
  while (string[p]);

  {
                                                        PS2 6 lines
        #ifdef CGA
        char_offset = (string[p] - 32) * 8;
        #endif
        #ifndef CGA
        char_offset = (string[p] - 32) * 14;
        #endif

        plot_char (x, y, char_offset, color, 1);
        y -= 8;
        if (y<8)
        {
           #ifdef VGA
           y=472;
           x += 14;
           #endif
           #ifdef EGA
           y=342;
           x += 14;
           #endif
           #ifdef VGA
           y=192;
           x += 8;
          #endif
        }
        p++;
  }
}
```

Existing program P-a:
Function to write a string of characters to the screen horizontally using the custom characters
(function which has the most lines in common with P-b)

```
void write_horz_str (int x, int y, char *string, int color)
{
                                                        DS1 7 lines
  #define seq_out (index, val)      {outp (0x3C4, index) ; \
                                      outp (0x3C5, val) ;}
  #define graph_out (index, val)    {outp (0x3CE, index) ; \
                                      outp (0x3CF, val) ;}
  unsigned int offset;
  char far * mem_address;

  int char_offset, dummy, i, j, mask, p=0;

  unsigned char char_test, exist_color;

                                                        PS1 2 lines
  couvert (x, y);
  while (string[p]);

  {
                                                        PS2 6 lines
        #ifdef CGA
        char_offset = (string[p] - 32) * 8;
        #endif
        #ifndef CGA
        char_offset = (string[p] - 32) * 14;
        #endif

        plot_char (x, y, char_offset, color, 0);
        x += 8;
        #ifdef CGA
        if (x>312)
        {
           x=0;
           y -= 8;
        }
        #endif
        #ifdef CGA
        if (x>632)
        {
           x=0;
           y -= 14;
        }
        #endif
        p++;
  }
}
```

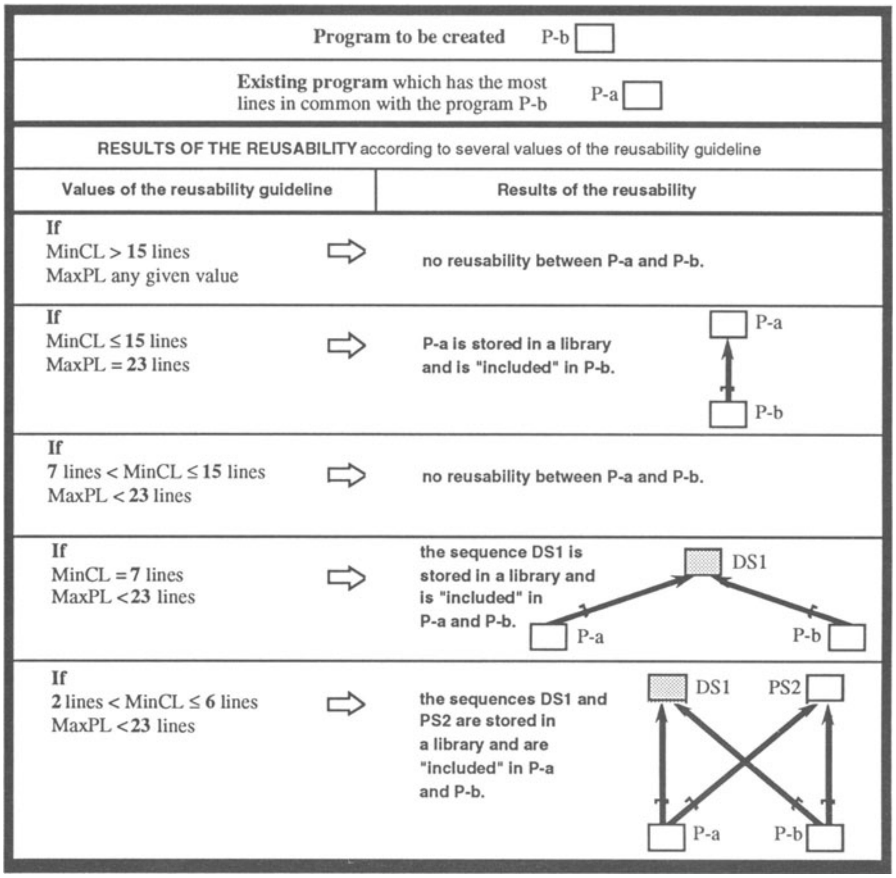
Legend ▨ **DSi** : Declaration Sequence. ☐ **PSi** : Processing Sequence.

Number of lines of the common sequences of P-a and P-b: **Number(CL-ab) = 15 lines**

Number of lines of the particular sequences of P-b: **Number(PL-b) = 24 lines**	Number of lines of the particular sequences of P-a: **Number(PL-a) = 23 lines**

Figure 3: Example of C program reusability study

The results of the program reusability mechanisms are presented in Figure 4 according to several values of the code reusability guideline. The links "is included in" are represented with arrows.



Program to be created P-b

Existing program which has the most lines in common with the program P-b P-a

RESULTS OF THE REUSABILITY according to several values of the reusability guideline

Values of the reusability guideline	Results of the reusability
If MinCL > 15 lines MaxPL any given value	no reusability between P-a and P-b.
If MinCL ≤ 15 lines MaxPL = 23 lines	P-a is stored in a library and is "included" in P-b.
If 7 lines < MinCL ≤ 15 lines MaxPL < 23 lines	no reusability between P-a and P-b.
If MinCL = 7 lines MaxPL <23 lines	the sequence DS1 is stored in a library and is "included" in P-a and P-b.
If 2 lines < MinCL ≤ 6 lines MaxPL <23 lines	the sequences DS1 and PS2 are stored in a library and are "included" in P-a and P-b.

Figure 4: Examples of results of program standardized reusability (obtained with several values of the reusability guideline)

6. CONCLUSION

The reusability guidelines allow comparison of programs to study if they are similar by measuring respectively their common sequences and their particular sequences.

We have adopted the solution which consists in measuring the likeness between programs with guidelines and not with percentages of common and of particular sequences because this last approach does not allow comparison of programs with a reference (a guideline). The use of guidelines supposes that a good metric may be chosen. For program studies we have used the sizes of the declaration and the processing sequences measured with the same metric: the number of lines. It is possible to measure these sizes with numbers of statements or other metrics. Furthermore it is possible to use two different metrics and two guidelines to measure the declaration and the processing sequences.

The reusability mechanisms are mechanisms of pattern recognition because they allow us to compare programs and to study their resemblance. Furthermore they can also create sequences which are stored in libraries of which the reusability must be studied. So the algorithm of the reusability mechanisms is recursive and these mechanisms do not only make pattern recognition.

The reusability mechanisms allow us to find reusable components which can be stored in libraries and reused. So the reusability mechanisms allow us to begin a reverse engineering. They should be implemented in CASE tools, in DBMS and in programming environments of languages.

An outstanding research is on the possibility to take into account programs and sequences code which are syntactically different but which have nearly the same control graph and/or the same call graph.

References

1. Albrecht A. J.: 'Measuring Application Development Effort Productivity', *Proceedings Joint IBM/SHARE/GUIDE App. Symp.*, October 1979.
2. Blaise G.: 'Implantation de l'algorithme de réutilisabilité uniforme d'objets MCO en Smalltalk-80', Graduate thesis engineer cycle C CNAM, 1992.
3. Boetticher G., Srinivas K., Eichmann D.: 'A neural net-based approach to software metrics', *The fifth international conference on software engineering and knowledge engineering*, 1993.
4. Boehm B. W.: 'Software Engineering Economics', Prentice-Hall, New York, 1981.
5. Castellani X.: 'Mechanisms of Standardized Reusability of Objects', *IFIP WG 8.1 Working Conference on Information System Development Process Como*, September 1-3, 1993, proceedings published by North-Holland.
6. Cheng J.: 'Improving the software reusability in object-oriented programming', *ACM Press*, Oct. 1993.
7. Cox B.: 'Planning the software industrial revolution', *IEEE Software*, 7(6), November 1990.
8. Fenton N. E.: 'Software metrics, A Rigorous Approach', Chapman & Hall, 1991.
9. Firesmith D. G.: 'Managing Ada projects; the people issues', *Proceedings TRI Ada'88*, Charleston, WV, 24-27 October 1988.
10. Halstead M. H.: 'Element of Software Science', Elsevier, North Holland, 1975.
11. Krueger C.: 'Software reuse', *ACM Computing Surveys*, 24(2), 1992.
12. McCabe T. J.: 'A complexity measure', *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.
13. Meyer B.: 'Object-oriented Software Construction', Prentice Hall international, Series in computer science, 1988.
14. Nauer P.and Randell B.: 'Software Engineering', Report on a Conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, Belgium, 1968.
15. Stevens R. T.: 'Graphics Programming in C', 1990, M&T Publishing, Inc.
16. WISR'92: 5th Annual Workshop on Software Reuse Working Group Reports, M. Griss and Will Tracz editors, Software Engineering Notes, ACM Press, Volume 18, Number 2, April 1993.