Abstract Data Type and Mathematical Software Reusability: Two Linked Concepts

Paolino Di Felice
Dip. Ingegneria Elettrica, Università di L'Aquila, 67100 L'Aquila, ITALY

ABSTRACT
Many problems in science are solved by managing problems of linear algebra on sparse matrices. The activity in this latter field is proved by about 20 years now of research and development. In particular, from the practical point of view this activity has produced both efficient programs and large and interesting packages.

In the implementation of new mathematical functionality on sparse matrices, the central issue concerns the possibility of reusing part of the work already done. The adoption of the software reusability technology constitutes the most promising approach to increase both the productivity of programmers and the quality of mathematical software.

In this paper an experience of design and implementation of a reusable Pascal code is reported. The project concerns the solution of the LU-Factorization problem on large sparse matrices. The code carried out is structured into reusable units. The unit design strategy is based on the Abstract Data Type (ADT) notion.

1. PRELIMINARY CONSIDERATIONS
Many researchers believe that the software reusability technology constitutes the most promising approach to increase software productivity and quality. In particular, in [1] Horowitz and munson say that such a technology has the potential for increasing software productivity by an order of magnitude.

Unfortunately, if it is true that there is consensus (among researchers and practitioners) on the importance of building reusable software systems, it is also true that the software reusability technology is still in its infancy. Hereafter, we shortly refer to two different categories of problems concerning software reusability.

From the theoretical point of view, the problem is that at present there is no general agreement upon an approach to reusability. This is confirmed by the excellent foreword to the Special Issue on Software Reusability published by the IEEE Transactions on Software Engineering Journal [2]. In particular, it is not clear what should be reused and how that should be done.

From the practical point of view, the following two problems arise [3]:

a. software systems are often not initially designed for being reused. Therefore, it is hard to find software that can be easily reused;

b. different programmers have different styles of writing a program. If the author of a program is not available to answer questions about his program, then often this becomes the most critical problem in understanding someone else's program.

It is in somehow obvious that to overcome problems a. and b. above we need to impose some discipline on programmers so that they are forced to take care of reusability writing programs.

The aim of this paper is to summarize an experience we did designing and implementing a reusable mathematical software. The project was developed at the Laboratory of Computer Science and was devoted to solve the classical LU-Factorization problem on large sparse matrices (e.g., [4]). The code was split into modules (named units) that allow for separate compilation and information hiding. The code was implemented by different students working on different units. Each unit was designed and implemented for being reused.

As it has been argued, for instance, in [5], the design of the "right" unit is an extremely difficult art, particularly in the situation when the design is based mainly on
the designer's intuition. In our laboratory project, we provided the developers with an abstract-data-type (ADT)-based unit design strategy giving them the "right" reusable units as project specifications. The needed background about the ADT paradigm may be found, for instance, in [6-7]. Major features of ADTs are generalization and encapsulation. In the next section, we give evidence that the ADT paradigm may be used with benefits also in the design of reusable codes.

Details about the code implemented are collected in Section 3. In particular, in such a section it is sketched the structure, in terms of units, of the LU-Factorization code. The features of each unit of the code are also listed.

Conclusions contained in Section 4 end the paper.

2. THE ABSTRACT DATA TYPE SPARSE MATRIX

With respect to the problem $A = LU$, let us introduce the ADT SPARSE MATRIX having the non-zero elements of a sparse matrix as objects and the following set of operations as user defined operations applicable on these objects: $\text{ValueOf}$, $\text{FindPivot}$, $\text{SwapRows}$, $\text{Remove}$, $\text{Insert}$, $\text{RowsCombination}$, $\text{UpdateA}$, $\text{UpdateL}$, and $\text{LUFactorization}$.

In what follows, $A$, $L$, and $U$ are square sparse matrices of dimension $\text{dim}$ represented in some compact way inside the computer [8]. According to the ADT paradigm, we do not need to assume any particular kind of compact representation for sparse matrices $A$, $L$, and $U$. From our point of view, a sparse matrix is a set of 3-tuples of the form $\langle \text{row, column, value} \rangle$. As usual, $A[r,c]$ denotes the element of matrix $A$ in position $\langle r, c \rangle$, where $r$ is a row index and $c$ is a column index. Below, when an operation can be applied to both matrices $A$ and $L$, then, for brevity, we use $M$ to denote them.

Both the ADT SPARSE MATRIX and the semantics of the operations defined on it were given to the students as project specifications. Hereafter, we shortly give the semantics of these operations.

$\text{ValueOf}(A, r, c)$. This operation returns the numerical value of the element $A[r,c]$ in $A$.

$\text{FindPivot}(A, r, c, pivot)$. It returns the biggest element (the pivot) among elements that in matrix $A$ are situated on column $c$ and under row $r$, $r$ included.

$\text{SwapRows}(A, r, c)$. This operation swaps rows $r$ and $c$ of matrix $A$.

$\text{Remove}(M, r, c)$. This operation sets to zero element $M[r,c]$ in $M$ (that is, either in matrix $A$ or $L$). In the context of sparse matrices this means that the element $M[r,c]$ has to be removed from the computer memory.

$\text{Insert}(M, r, c, v)$. It sets the element $M[r,c]$ to the value $v$. In the context of sparse matrices this means that the element $\langle r, c, v \rangle$ has to be inserted into the compact representation of matrix $M$.

$\text{Append}(M, r, v)$. This operation appends at the end of row $r$ the value $v$. The Append operator is a special case of the Insert one.

$\text{SetValue}(M, r, c, v)$. It sets the element $M[r,c]$ to the value $v$. For performing such an operation on sparse matrices a Remove, Insert, or Append operation has to be executed.

$\text{RowsCombination}(A, r1, r2, \text{coefficient})$. This operation performs the linear combination of two rows of $A$. The value of coefficient is evaluated according to the Gaussian elimination strategy.

$\text{UpdateA}(A, \text{dim}, r, \text{pivot})$. This operator performs one step of the Gaussian elimination strategy; that is, it sets to zero all the elements of $A$ below the pivotal element $A[r,r]$.

$\text{UpdateL}(L, \text{dim}, r, \text{pivot})$. It fills out the $r$-th column of matrix $L$ with the coefficients evaluated according to the LU factorization method.

$\text{LUFactorization}(A, \text{dim}, L)$. This operation takes a sparse matrix $A$ of dimension $\text{dim}$ (as input) and returns matrices $L$ and $U$ (as output). Actually, at the end of the factorization process, matrix $A$ is matrix $U$.

3. THE LU-FACTORIZATION CODE

Implementing the operations defined on the ADT SPARSE MATRIX it corresponds (by definition) to implement the LU-Factorization problem. Figure 1 gives the structure of the LU-Factorization code. As programming language, we adopted Pascal because Pascal was the language the students involved in our project already knew; also because by using Pascal we had the possibility of enlarging the library on sparse matrices we started to develop since 1986 (e.g., [9-10]).
In particular, each unit of the LU-Factorization code, alias each operation of the ADT SPARSE MATRIX, was implemented as a Turbo Pascal 4.0's unit [11]. The Turbo Pascal 4.0 supports units separate compilation and information hiding. Information hiding is achieved by structuring each unit into two parts: the interface and the implementation. In this way, Pascal units become equivalent to Ada packages and Modula-2 modules.

To achieve the reusability of units concerning the LU-Factorization code:

- each unit is properly documented. In particular, it contains the information needed by the programmer for reusing the unit;
- the code is independent from hardware, compiler, or operating system features. Moreover, the dependence on global constants, types, or variables is avoided;
- units are parameterized and not just tailored for the LU-Factorization problem;
- the communication between units is allowed through procedure parameters only, while the direct accessibility of data is not allowed. This restriction, with respect to the general definition of units [11], produces a twofold advantage on the final code: to understand a unit's purpose and behavior it becomes simpler; run-time side effects are prevented. The same restriction above is adopted, for instance, by Rajlich in [5];
- the public part of each unit is constituted by the header of the top-level procedure/function inside the same unit, while the other procedures/functions of the unit are hidden. This choice produces the maximum degree of information hiding possible for each unit. Figure 2 shows how the interface of the FindPivot unit looks like.

4. CONCLUSIONS

The ADT-based design strategy reported in this paper is aimed at giving guidelines to the developer for designing and implementing libraries of reusable units. In particular, it facilitates the code reusability both at the functionality level (i.e., the LU-Factorization operation) and at the subfunctionality level (e.g., the FindPivot operation).

Moreover, by adopting such a unit design strategy, the decision about what should be a "right" unit is largely independent from the designer's intuition, on the contrary it is based mainly on the problem-algorithm knowledge. As a consequence, for a given couple problem-algorithm, we have that the number of units is fixed and equal to the number of operations defined on the objects of the ADT "modelling" the target application.

ADT-based code design strategies are equally well suited for programming in the small as well as for programming in the large. In this latter case, the in a sense obvious variation (with respect to what we did) is that the operations to be associated to the ADT modelling the (sub)problem of interest are defined (and implemented) incrementally; that is, as the programmer proceeds in the code development.

In our laboratory project, we observed a
The unit `unitFindPivot` returns the biggest element (the pivot) among elements that in a sparse matrix `A` are situated on column `c` and under row `r`, `r` included.

**Input Data Description.**
- `A`: the sparse matrix
- `r`: the row index
- `c`: the column index

**Output Data Description.**
- `pivot`: the pivotal element

**Other Information.**
- `const dim = ...;
type ValueType = real;
Link = ^Element;
Element = record
  Column: integer;
  Value: ValueType;
  Next: Link
end;
SparseMatrixType = array[1..dim] of Link;
(b) the unit `FindPivot` is written in Turbo Pascal 4.0.

**Interface**

```
procedure FindPivot(var A: SparseMatrixType;
r,c: integer;
pivot: ValueType);
```

**Implementation**

(omitted)

---

**Figure 2.** The interface of the `FindPivot` unit.

Large students' acceptance for the adopted unit design strategy. From their point of view, the major features of such a strategy are the following:

- It is not rigorous. Experience shows that rigorous methodologies often are not well accepted by programmers (e.g., [12]);
- It does not require neither a specific background to be understood nor mnemonic efforts to be remembered;
- Finally, its usability is high since its application is driven by the problem-algorithm knowledge.

To improve the effectiveness of strategies like that reported in this paper in the development of reusable packages, it is important to have around them a software-engineering-based programming environment able to support retrievalability, compositability, and understandability of (passive) reusable units. Genesis [3] is a good example of a software-engineering-based programming environment offering tools and techniques suitable to support these facilities.

In [13], Di Felice describes the basic requirements and the main features of a low cost and easy to use system specifically tailored to support the reusability of an existing large library of units concerning sparse matrices and designed according to the strategy outlined in this paper.

**Acknowledgements.** This work was supported by the Ministry of Public Instruction under project Mathematical Software for Sparse Matrices on Personal Computers.

**REFERENCES**


---

45

