Libraries versus Languages in the Reusability of Programming[1]

James M. Neighbors
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
1983

1 The Two Faces of Reusability in Programming

When we talk about the reuse of existing programs we must be careful in describing the goals of the reuse. In some cases a programmer on a project is looking for a program part which can just be "plugged in" without modification. In other cases the programmer is looking for a program part which he can modify before using. This is an important distinction for someone contemplating the construction of a library of reusable program parts. In the first instance only what the program part does need be stored while in the second instance both what the part does and how it does it need be stored.

The reuse of program parts without modification has been terribly successful in the past and has been invisible to the programmers requesting the reuse. The obvious example of this style has been the implementation of compilers by linkage to run-time support routines. The use of classical program libraries such as mathematical subroutines supported by linkage editors is another example of reuse of this type. The library is kept in an encoded form and thus cannot be changed by the programmer. This approach has been most successful in reusing libraries of mathematical functions since the data objects being manipulated by the programs are one of a few different types of number representation. In the author's opinion this type of reuse will not fare well when the data objects being manipulated become more complex than simple numbers.

The reuse of program parts with modification with the aid of a machine has not been investigated very much as such. Most of the work in this area has come from research on automatic programming, program generators, computer-aided software engineering, and specialized language design. It appears to this author that the reuse of program parts with modification without the aid of a machine is the major activity of detailed design and coding. Encyclopedic works such as [Knuth68, Knuth69, Knuth73, Aho74] serve as guides supplying information above the level of programming language code which tells the programmer what the part does and how it does it. The "how" information allows the programmer to adapt the part to the system under consideration.

In the author's opinion significant increases in "programmer" productivity will only come from the reuse of program parts with the aid of a machine. We believe that a major part of the solution to the "software crisis" will be the reuse of software. Further, we believe that in the long term the key to the reuse of software is to reuse analysis and design; not code [Neighbors80a].

1. Unpublished position paper sent to the Workshop on Reusability in Programming, ITT, 1983.

2 Organizing a Collection of Software Parts

2.1 Software Part Libraries

If we have the “what” describing the function of each software part in the collection, then one straightforward way of organizing the collection is to put each part into a library of source code. Potential users of the part would search through the “what” descriptions of the parts of the library and select the appropriate part. This is the scheme used by most source program libraries. The problems encountered by this scheme are:

1. The <u>classification problem</u> of what is an appropriate language or scheme for specifying and searching “what” descriptions.

2. The <u>search problem</u> in that the burden of searching the library is placed on the potential user of a part. Quite often it is easier for a potential user to (re-)build a part from scratch rather than find a part in a library and understand the constraints on its use.

In addition, following the discussion of the previous section, if the potential user is looking through the library for a software part which can be modified, software part libraries will encounter the following problems:

1. The <u>structural specification problem</u> of what is an appropriate language or scheme for specifying “how” descriptions and constraints of usage between software parts.

2. The <u>flexibility problem</u> of what to make flexible and what to fix in the software parts put into the library.

The overall library problem is aggravated by and increases the magnitude of all the other problems. If the parts in the library are to be modified and reused then they must be small to be general, flexible, and understandable. However, if the parts in the library are small then the number of parts in a usable library must be very large. These two objectives are always in conflict. If a library contains many small parts then it lessens the structural specification and flexibility problems at the expense of increasing the classification and searching problems. If a library contains a small number of large parts then it lessens the classification and searching problems at the expense of increasing the structural specification and flexibility problems.

2.2 Specialized Languages

An alternative to using program libraries is to use specialized languages as a surface form to tie together software parts. As a historical example consider FORTRAN not as a programming language but as a surface description scheme for tieing together the software parts which make up the FORTRAN run-time library. Would FORTRAN have been nearly as successful if it had been presented as a “library of interesting and useful numeric input, calculation, and output routines with descriptions”? In the author’s opinion just a library would not have been as successful because the burden of using the library is placed upon each and every potential user of the library. Having a surface language which ties the library together removes this burden. KLONE and PLANNER are more recent examples of this technique in a problem domain far removed from FORTRAN. These are problem-domain specific languages.

3 Draco and the Reusability of Programming

For the past six years we have been working on a program generation system called Draco [Neighbors80b] which uses specialized languages to capture and reuse the analysis of a particular problem domain. Programs in these languages are refined into other specialized languages using very small, very flexible software parts. The software parts (we call components) represent the reuse of detailed design while the use of one specialized language to implement another represents the reuse of architectural design. Eventually the desired system is modeled in a conventional executable language. During the process of restating one specialized language in another (we call this refinement) source-to-source program transformations are used in a specialization role to remove any unused generality. This enables us to use large numbers of very small and very general software components in the development of many systems.

4 References

[Aho74]
Aho, A.V., Hopcroft, J.E., and Ullman, J.D.
The Design and Analysis of Computer Algorithms
Addison-Wesley, 1974.

[Knuth68]
Knuth, D.E.,
The Art of Computer Programming. Volume 1: Fundamental Algorithms,
Addison-Wesley, 1968.

[Knuth69]
Knuth, D.E.,
The Art of Computer Programming. Volume 2: Seminumerical Algorithms,
Addison-Wesley, 1969.

[Knuth73]
Knuth, D.E.,
The Art of Computer Programming. Volume 3: Sorting and Searching,
Addison-Wesley, 1973.

[Neighbors80a]
Neighbors, James M.,<BR>
Software Construction using Components,
Ph.D. Dissertation, Technical Report UCI-ICS-TR160,
Department of Information and Computer Science,
University of California, Irvine, 1980.

[Neighbors80b]
Neighbors, James M.,<BR>
Draco 1.0 User Manual,
Technical Report UCI-ICS-TR157,
Department of Information and Computer Science,
University of California, Irvine, 1980.