Used Software

by
R. P. Gabriel
Lucid, Inc.

1. A Story

The student looked out the window and wondered how to speed up his program. He had written a toy Lisp system and its compiler, and he had to speed it up.

His code was simple,

```
(defun fib (n)
 (if (< n 2)
     n
     (+ (fib (- n 1)) (fib (- n 2)))))
```

but all his tricks failed—tail-recursion removal, specially tailored function-call, passing everything in the registers; it was too slow.

He looked out his window and wondered and saw the snow falling in the purple light and his thoughts settled to the ground.

2. Caching

What the student failed to notice was that he was recomputing many values of the FIB function over and over. Unaware that his intermediate results could be saved, he thus did not see the better algorithm for Fibonacci:

```
(defun fib (n)
 (if ((< n 2)
     n
     (do ((p 0)
          (c 1)
          (i 2 (1+ i)))
      ((< n i) c)
      (setq p (prog1 c (incf c p)))))))
```

Although this code may look difficult, it runs much faster than the student's version. The key is that for $2 \leq n$, the value of fib(n) is cached in C and the value of fib($n-1$) is cached in P. The computation of fib(m) for $2 \leq m$ is iterative, using the two cached previous values to compute the next value.

This function could also be written in this way:

```
(defun fib (n)
 (labels ((inner-fib (p c i)
           (if (< n i)
               c
               (inner-fib c (+ p c) (1+ i)))))
  (if (< n 2) n (inner-fib 0 1 2))))
```

'Caching' is used to save previously computed values of functions that might be of interest. Caching is a general technique that enables a programmer to greatly improve the running speed of programs by only computing a frequently used or expensive-to-compute value once. Looking up the stored value in a table is rarely very expensive and is often no more expensive than a hash table lookup.

On many high-performance computers, and even on some medium-performance computers, caching is used to speed up general memory operations. Even paging is a form of caching technique—the disk is regarded as the large main memory, and high-speed memory (RAM) is used to cache the most recently addressed portions of the memory.

In a large, dynamically retargettable Common Lisp compiler, caching is used in half a dozen different places, yielding a total speedup of a factor of 3 or 4 in compiler performance. That is, by adding caches in these places, the speed of the compiler when compiling code improves by up to a factor of 4. In general, the local improvements from caching at the low level are about a factor of 10 over the uncached methods.

The technique of caching can be summarized by the slogan "If you've done it before, don't do it again."

3. The Act of Programming

The act of programming is to take a specification of a task, even a loosely stated specification, and turn it into a program text that, when compiled and executed, performs the specified task.

Expertise in the art of programming is in some sense a cumulative skill. People who program at the expert level have stored away a number of programming cliches, so that large parts of the programming task for them involve nothing more than recalling the appropriate cliche and applying it to the current task. For example, a programmer who has produced many implementations of queue management will find each new queue-management-like program simple to write. Such a task might stymie the novice programmer.

In the more than 25 years that people have been programming computers, many programs have been written. Each can be thought of as a programming cliche and thus if the caching slogan introduced above is followed, we should not write a program if someone has already written a program similar to the one we need. Software that is used in this fashion might be termed "used software."

If I sit down at a terminal to edit a file, I don't write an editor first—I simply use the editor on the computer. This is a program that already exists, and there is no need for me to write it again. If I sit down at a terminal and want to run Lisp, I don't write a Lisp system first—I simply use the one that is already on the computer. However, if I want to run the Lisp system while I am in the editor, using the editor as an input stream to the Lisp, I will probably have to write some code that will enable me to do that.

4. The Perfect World

The ideal towards which we should strive as computer scientists is one in which all pieces of previously written software can be easily re-used. Programmers ought to be able to take two programs and build a third that is the result of combining the two existing programs; if they wish, they also ought to be able to impose some control structure on top of the two programs, perhaps as a feedback loop.

For example, suppose that one of the two programs is a simple spreadsheet that takes data from a data base and that the second is a program that performs a factory-floor simulation. The spreadsheet program reads a model from disk and determines the profit-and-loss situation for a small manufacturing company. The goal is to produce a program that enables a production-line manager to reconfigure his assembly line in ways that positively impact the company's bottom line. There are are set of parameters that can be varied, and the new program is to vary these parameters, watching the bottom line and hill-climbing until the best payoff is achieved.

The two biggest pieces of the new program are already written. But to combine them, the programmer must probably make the two programs interact through files—the spreadsheet program with its model will read a file that is written by the factory-floor simulator and write a file that the controlling program (the hill climber) will read, and so forth.

If the operating system provides features for interprocess communications, the communication between the two programs could be handled by using these features rather

than by using files for communication. This method of combining programs is referred to as 'message-passing.'

In the algorithmic uses of caching, a program that requires previously computed values might look up those values in a table rather than invoke the subroutines or functions that compute them. As an analogy, when programmers are writing programs whose behavior is like that of programs that have already been written, it is preferable to use those programs again intact rather than write them from scratch. If previously written programs can be used intact, the storehouse of existing programs can act as a cache of programs. But unlike value-caching techniques, which are well known, program-caching techniques—how to re-use old software—unfortunately are not well known.

5. Review of Currently Used Software Techniques

Building new programs out of existing programs is possible using known but rather crude techniques. The thrust of new research is to expand these techniques, and new operating systems and hardware will possibly be needed to achieve something close to the ideal. Some modern programming systems do demonstrate the capability of incorporating used software into new programs. This section is an overview of some of the existing techniques.

In the best of cases, the programming cycle is approximately this: Define a formal specification of the program, define the abstract data structures, define the control flow; derive the program from the specification either by hand, by using semi-automatic techniques, or by automatic techniques; compile the program into a running image; and use the program as intended or let it fall into disuse and forget about it.

In the best of cases, all of the artifacts of each part of the cycle are preserved—the specification, the derivation, the source code, and the object code must be saved. When the old, used program is rediscovered, the task of re-using it is easier if all these artifacts exist; the task is much harder if only the running image is available.

There are three basic techniques for creating a new program out of old code and some new code: software engineering techniques, same address space techniques, and object-oriented/message-passing techniques.

5.1 *Software Engineering Techniques*

Software engineering techniques for the re-use of software center on structuring the process of code creation so that the program can be re-used easily. Most of the techniques available focus on the program text or on the derivation of the program text from a formal specification. Source code which is poorly documented and old is nearly as hard to re-use as executable files alone.

To be able to re-use software in a new application using software engineering techniques, programmers must exercise some forethought. Programmers, however, generally do not consider the possible recylcing of their code. Typically, programs are written to stand alone, to exist as if no other program mattered. For example, programmers frequently use global variable names which are not specific to the task and which can, therefore, clash with variable names appropriate to some larger supertask, and programmers similarly often choose subroutine names without regard for name conflicts.

If a program was written using modular programming in which the code is well specified, the interface is documented and well structured, the internals of the code are hidden from view, and the source code is available and usable in the current context, that program can simply be used as a building block in the new application or perhaps used much as a subroutine is used in standard programming practice. That is, the source code for the old program can simply be incorporated into the source code for the new program.

Many programming languages do not have features for abstraction and encapsulation of programs (notable exceptions are ADA, Common Lisp, and Modula). Even if these tools are available, programmers who are writing what they consider to be one-time programs may not wish to make the effort to write their programs so that they can easily be re-used.

There are certain prerequisites for using the modular programming technique. The original programming language must be compatible with the currently desired one; the original formulation of the modules should be suitable; and the compiler for the original programming language must still produce correct code for the program.

If the original code was written automatically or semi-automatically from a formal specification—that is, if the derivation of the program from a formal specification was kept and was usable, the new program can be derived from the union of the old specification and derivation along with the new specification. According to the inferential programming methodology discussed by Scherlis and Scott [Scherlis 1983], all programs should be saved with their derivations for just such a contingency.

This is an ideal methodology because the re-use of the program (source code or source code plus derivation) in some larger context can lead to optimizations to the original program, perhaps resulting in a faster, smaller specialization of the original program.

For example, suppose that the existing program is a text editor, suitable for editing large files, preparing lengthy documents, and writing code in several programming languages. This program is probably quite large and convoluted. Suppose further that the new use is as a line editor in a particular application. The new program will occasionally stop and ask the user a question, which is to be answered by providing a line of text. The applications programmer wants the user to be able to edit his text by using the text editor, rather than by rubbing out previous text and retyping when an error is discovered.

In the obvious uses of this new program, the file-reading, document-preparing, and program-code-creating aspects of the text editor are not needed; this implies that most of the old code is not needed. But if the user of the new program is somewhat sophisticated, the ability to examine files, to extract pieces of the response text from files, and to compose text for input to the program would be useful and important features. If this were the case, the entire text editor might be required.

5.2 *Same Programming Language*

Using existing software in source-code form works best when the currently desired programming language and the original programming language are the same and when the compilers are also the same. The only question then is how to link the various parts of the programs. Several possible problems can occur. The names of functions or subroutines may conflict, the names of global variables may conflict, or the data structures may be incompatible or not used abstractly.

The data structures in both the old code and the new code ought to be used in abstract form in the sense that the underlying implementation of the data structures should not be exposed to the program. For example, if some object is being represented as a vector, the creation of the object, its access, and its modification should all be named with terms appropriate to the object represented and not with terms appropriate to vectors.

Here the typical programming scenario gets in the way. Because programs are infrequently written with re-use in mind, the task of re-coding an existing program for re-use may exceed in difficulty the task of rewriting the existing program unless a carefully written specification is in hand.

5.3 *Programming Systems with Packages*

A powerful technique for re-using code is to encapsulate it into a larger piece of code in such a way that the implementation technology for the old code is hidden and only a narrow, precisely defined interface to it is presented.

Some programming systems have 'modules' or 'packages.' These program-construction constructs enable programmers to write code in such a way that only the external interfaces to their software are visible to other parts of the system. The internal global variables, the procedure names, and the implementation of data structures are hidden. The purpose of these constructs is to reduce the number of possible conflicts between parts of the program written at different times and by different people.

In programming systems that have modules or packages, mixing old existing code with new code is easier than in systems without them. In Common Lisp, for example, old code can be placed in a separate package, which keeps name conflicts from occurring. By using packages with the correct interface code exported to the package with the new code, the problems of mixing old code with new code are minimized.

The advantage of this is that the existing program can be retroactively placed in a separate package, and the interface routines can be exported with the new code, perhaps with a renaming for compatability. By knowing which parts of the code must be exported, the programmer can still use the code effectively even if the original author is not available. This is a tolerable, although not ideal, situation.

5.4 *Translating Programs*

The programming language used by an original author is often not appropriate for a contemplated application. If this is the case, the pleasant situation of working in the same programming language can be attained by translation.

Translation can be done by hand or by machine. Translation by hand is tedious and error-prone. If the original programmer's style is unlike the current programmer's, the task of translating can be maddening.

Translation by machine has only recently become feasible. Several commercial concerns now provide the service between selected programming languages. Translation is like compilation, but while a compiler translates from a high-level to a low-level language, eliminating unnecessary steps along the way, a translator translates a high-level-language (usually) to another high-level one, perhaps with the problem that there is a near-match

between the constructs in the two languages, but the details of the match are difficult to express.

For instance, suppose that the original language has an iteration construct that is inherently parallel—binding forms do not interact—and the new language has an iteration construct that is sequential. The translated use of the iteration will be ugly, perhaps introducing new variables and using side-effects. If the new program is to be maintained by hand or to be altered, the result could be exasperating.

When the translated-from language is a much lower-level one than the translated-to language, the translation is much more difficult. Imagine a large assembly language program in which clever use of registers is made. How could this be easily translated except by mimicry of the original target computer in software?

Translation only allows us to use the same programming language. Without a known specification, the known problems of re-using the code in the context of the new program remain.

5.5 *Object-Oriented Programming*

Object-oriented programming, in its purest form, would be an excellent solution to the used software problem if it were uniformly used and extended to programs as a whole. Object-oriented programming is a technique and a set of programming constructs that enable a programmer to create objects whose behavior is inherited and whose behavior is invoked, usually by message-passing.

Here is a simple example of the inheritance of behavior. Suppose we are writing a program about automobiles operating in various situations. We define a category of thing called an 'automobile.' This category is represented by a data structure whose contents reflect the parts of an automobile of interest. As terminology, we refer to the parts of this data structure as 'slots' and to the contents of the slots as 'slot-values.'

A program can create an instance of an automobile in a stylized manner. The parts of the automobile not specifically assigned by the instance are inherited from the general category. The slots can contain procedures as slot-values. Procedures can be written whose major activity is to invoke other procedures found in slots.

An important use of inheritance occurs in systems that allow multiple inheritance. Suppose that we have an application in which there are automobiles and tanks and that the program we want to write requires the representation of an automobile used as a tank

in a war situation. Inventing a new category of object called 'automobile-used-as-tank' is one way to accomplish this. Another way is to create an instance of both the 'automobile' category and the 'tank' category. This new instance inherits from both of these categories.

A procedure that responds to messages sent to it is one type of procedure that can be invoked on an instance. An object can be sent a message whose contents is a representation of a request for the object to print itself, and the result of sending the message is that the object is printed on some device.

The important point about this is that the programmer does not need to know how to print the object, nor does he need to know the name of the function that will print the object. However, he does need to know the form of the message that will cause the object to print itself.

How to know the forms of messages is a problem whose solution lies in one of two directions. A common language for messages of general use could be created, and any object-oriented system would use this store of messages. The second solution would be to use a variant of message-passing called 'generic' functions.

A generic function is one in which the actual procedure executed depends on the types of objects passed as arguments to it. In the print case, a generic function called 'print' would be defined to print the data structures normally available in the programming language. When the automobile category is defined, an addendum to the definition of 'print' would be made that stated that if the object to be printed is in the category called 'automobile,' some other specified procedure is to be invoked instead of the usual one.

The definition of 'print' can thus be embellished to any degree necessary. And the problem of a standard set of messages is replaced by the problem of a standard set of procedures.

One method for re-using software with minimal rewriting is to treat entire programs as objects; the method of communicating data between programs could be accomplished by message-passing, using an object-oriented programming approach. Remember, we are talking about programs as object code with possibly no existing source code as the ideal definition of a program. Because running programs exist within an operating system, the object-oriented programming constructs needed for the used software problem would have to exist within the operating system.

Currently, object-oriented programming exists as a technique and as a set of programming constructs within a programming language. Most operating systems have only limited versions of this sort of functionality. The two notable exceptions are the Lisp machine operating system, which is simply an extension to Lisp (actually a programming language along with the operating system) and the UNIX operating system, which provides facilities for 'patching' programs together.

5.6 *Lisp Machines*

'Lisp machine,' as used here, refers to a computer whose only programming language is Lisp and whose operating system is an extension of Lisp. Lisp is a programming system, which includes the Lisp programming language plus a large number of utilities. As such, it is easily extendable to an operating system.

If there were only Lisp machines, the problems of used software would be much easier to contend with because the goal of combining a common programming language and a powerful package system to separate programs could be achieved.

Of course there is nothing special about Lisp in this regard. It simply happens to be one of the languages that has been most developed and designed with this goal in mind. Other programming languages are suitable for having a 'machine' around them; Ada is a good example. Insisting on Lisp or on any single programming language is unrealistic. People have programmed in a variety of programming languages, and they will continue to do so.

5.7 *UNIX*

UNIX, despite its drawbacks, has one feature that is of great use in the quest for a means of re-using software—pipes.

Programs generally communicate with people by typing on the screen. Character output is channelled through a single mechanism, a mechanism much like a stream. Similarly, many programs accept input that the user types in. This input is also handled through a mechanism that is like a stream. A pipe is a means of connecting two programs to each other by connecting the output channel of one to the input channel of the other. Instead of typing out to the screen, the first program sends its character output directly to the input channel of the second program, which was written to expect characters typed in at a terminal.

Programs not specifically written to communicate can thus communicate easily. The communication language is a standard one—ASCII text. Other programs in other operating systems can be made to mimic pipes by communicating through files or through other interprocess communication channels provided by the operating system.

6. Common Interprogram Language

The goal is to enable programs to be re-used without extensive reprogramming, translation, or human interaction with the innards of the old program.

One way to solve the used software problem is to define a common interprogram communications language that can act as a protocol between programs. This interprogram language would also serve as the usual means for the program to interact with the outside world. That is, all terminal, keyboard, file, network, and mouse interactions would be channelled through this mechanism.

For example, the programmer writing code that interacts with a graphics system would invoke the interprogram language to get lines and characters (with fonts) onto the screen. Ideally, the programmer would not know how any of the underlying objects were represented. Furthermore, this mechanism could be object-oriented in its implementation, and all slots external to the program as a whole could exist only in the interprogram language part. The programmer would then be able to manipulate the program as a whole by using only the hooks provided by the interprogram language.

The interprogram language would need to include abstractions for all of the interchange media available—characters, strings, lines, textures, bitmaps, mouse events, file names, and many others. This language is not easy to define, and grafting programs together might involve representation changes between programs. For example, if one program is writing lines onto a screen to form closed polygons while a second program is taking lines forming polygons and calculating areas, the first program will 'think' it is still drawing lines on a screen, and the second program will 'think' it is receiving lines as pairs of cartesian co-ordinates. This might require a representation change if the interprogram language is not written at the right level of abstraction.

The notion of pipes can thus be extended to include interactions with graphical objects on a bitmapped screen. In fact, all external behavior of the program could be used as its 'exported' behavior while its internal workings are kept a compiled secret. The benefits of this approach extend beyond used software. For example, a program originally written

for bitmap graphics could be run on a machine with display-list vector graphics, assuming that the proper abstractions are implemented.

7. Object-oriented Operating Systems

To use the interprogram language, an operating system that provided a good programming environment would be needed—an environment that, like the Lisp machine environment, would provide a powerful vehicle for using existing software. Unlike the Lisp machine environment, however, this environment would need both shared and unshared address spaces. Many programs are written that assume that the entire address space is available to them, and the other parts of the program, which are possibly other old pieces of used software, need to be protected from such unruly programs.

This attractive scenario would certainly be an improvement over the current situation. A programmer would only have to manipulate an existing program that interfaced with the outside world through an explicitly managed set of channels, and the existing program would look as if it were written in a modular fashion, ready for use. It would not need to be re-compiled, translated, or manipulated in any way other than by representation changes in the interface between programs.

But the scenario could be much better. Programs are written for very different pieces of hardware. And even if all programs were written using the interprogram language, a new program put together from a number of old programs compiled for different computers would not be able to run on any single computer unless the source code was still available and all the old programs could be re-compiled. How do you run an IBM 370 program on an MC68020?

8. Conclusions

The world of programming is in a medieval stage at present. Different programs are castles in a feudal society of programs, with little communication between them.

We cannot solve all programming problems here but only touch upon some of the directions that research in programming methodology might follow. The pie-in-the-sky goal of taking any program written any time for any piece of hardware and using it to build new programs with almost no programming changes to the original program is too much, perhaps, to ask. However, with the speed of new computers, it may indeed be possible to define virtual processors with acceptable performance.

9. Reference

[**Scherlis 1983**] Scherlis, W., Scott, D., *First Steps Towards Inferential Programming*, IFIP Congress 83, North-Holland, 1983.