Software manufacturing techniques and maintenance

by PAUL BASSETT
Netron Incorporated
Toronto, Ontario, Canada

ABSTRACT

"As ye sow, so shall ye reap."

A good solution to the reusable code problem turns out also to provide a solid technical basis from which to understand and deal with the production, quality, and maintenance issues of the software industry. To this end, a software manufacturing methodology has been developed called Computer-Aided Programming. CAP is based on a functional programming concept called a frame. Frames were originally developed as a means of resolving the maintenance problems associated with reusable code.

The introduction explains the necessary background ideas about frames and the types of maintenance that they address. Section two presents the design principles for software that uses frames as subassemblies for program assembly purposes. The components of an existing CAP system are described in section three, and section four discusses the use of CAP as a manufacturing technique. Statistics from a case study are presented to indicate that: (1) production-quality commercial software can be manufactured at rates exceeding 2000 lines of debugged COBOL per man-day (including systems design time), and (2) less than 10% of this code needs to be hand-written or maintained.
INTRODUCTION: THE MAINTENANCE PROBLEM

Software has had a precocious, turbulent childhood, as is typical of newly emerging disciplines. In spite of many important advances, software still remains a hand-made commodity designed in an ad hoc manner with few standards; a product that is almost always late, poorly documented, and difficult to maintain.

Maintenance, more than any other factor, holds the software industry captive, strangling productivity and tying up vital programming resources. The half-life of a typical program is approximately 14 months. The maintenance statistic now approaches 70% and is still climbing.

The central thesis of this paper is that a substantial portion of the maintenance effort stems from the reusable code problem. A good solution to this problem turns out also to provide a solid technical basis to understand and deal with both the production and quality of software and the maintenance issues currently besieging the software industry.

The Reusable Code Problem

In the software industry's current cottage industry style, it is common practice to build new programs by cutting and splicing pieces of old programs together. This approach demonstrates that there is a great deal of potentially reusable code available, and that it is worth the effort to adapt it rather than starting from scratch. Reference 16 shows that unfortunately

1. The programmer does not have any systematic way of isolating just what portions of programs are relevant
2. The customization process is time consuming, tedious, and prone to error
3. Once the process is finished, both old and new programs must be maintained as if each were completely unique, despite the considerable common functionality. Maintenance effort should be proportional to the novelty in the system, not the number of source statements.\(^{1}\)

External Subroutines

It is still widely believed that external subroutines form a satisfactory repository of reusable code. Separately compiled and linked subroutines are obviously useful, but they are limited because there is no graceful or systematic means of effecting local customization of an external subroutine to fit each calling program's particular context of use, or of effecting global evolution of a subroutine when it must change to benefit all future callers of that subroutine without victimizing current callers.

The subtle and often frustrating side effects introduced when common components undergo maintenance directly contributes to the severity of the maintenance problem.

The root cause is that a subroutine is a representation for a single function that is not adaptable at source-program (function) construction or maintenance time. It may have considerable run-time flexibility, but at the time of actually molding the subroutine into the program that must use it, an external subroutine (by its very nature) has no flexibility at all.

Code Generators

Code generators have been around for years (e.g., RPG). Although they offer a potential to drastically simplify the maintenance of large portions of a program, their potential goes unrealized.\(^{2,10}\)

The simplest kind of code generators are those that generate "raw" source code. The problem with such generators is that they are basically "one-shot" tools. Because each generator is expert at only a part of the overall problem,\(^{3,17}\) programmers must supplement and modify the generated source code to suit their own needs. Having adapted the code, they have no means of reusing the generator without destroying all of their manual modifications. This forces the programmer to support the life-cycle maintenance of the program at the more difficult and error-prone level of generated source code, rather than the succinct, declarative level of the original input to the generator.

To be more useful, a code generator must allow some follow-on mechanism that can adapt the generated source code automatically, thus allowing reuse of the generator without the loss of the customizations.

More sophisticated code generators typically supply "user exits" for handling this problem. These provide linkage to separately compiled, external subroutines that usually can be written in a variety of general purpose languages. The trouble is that this is always an additive technique; there is no way to change or remove generated functionality. Also, predefined interfaces often omit information that is essential in the customization (the "black box" effect). In addition, all non-procedural parts of the generated code, such as data declarations, are simply unavailable for refinement. A proper solution requires generators to provide for automatic customization of generated code (not just run-time communication with generated modules).

The Frame Methodology

A frame methodology,\(^{15,14}\) has been developed to address the reusable code problem from the perspective both of pro-
grammers and of code generators. A frame is a machine-processible representation of an abstract data type, with "abstract" meaning functional. Because the data operators are functionals, not functions, frames can accommodate both local customization into an individual program and global evolution to benefit all future embedding programs. Frames are implemented as files containing a mixture of source code (e.g., COBOL) and preprocessor macro commands, but are quite unlike the proposals of Backus or Evans. This mixture is called frame text.

There are just four macro commands whose essential role is to automate the cutting and splicing of programs:

1. COPY-INSERT allows a frame hierarchy to be copied into a program (by naming the frame at the root of the hierarchy), and causes customizing frame text to be INSERTed anywhere into that hierarchy.
2. BREAK-DEFAULT defines a named "breakpoint." Breakpoints mark arbitrary places in a frame where custom frame text can be INSERTed to supplement or replace DEFAULT frame functionality.
3. REPLACE systematically substitutes a specific code string for a generic one (throughout a frame hierarchy). For example, field names and picture clause elements are generic if they tend to vary from program to program.
4. SELECT incorporates into a program one text module from a set of modules in the frame. SELECTs are like CASE statements (with arbitrary nesting), which operate at text construction time. An important use of SELECT is to automate version control (global evolution).

Frames are written both by analysts and by generators. Having code generators produce frames solves the problem of destroying subsequent refinements by automating the cutting and splicing of the customizing frame text into the generated frame text.

All customizing frame text for one program is localized for maintenance purposes into a SPECIFICATION, or SPC, frame. Typically, the size of this file will be less than 10% of the generated source code. An SPC governs the entire process of building the compilable source program from its frame components. As will be seen, a methodology incorporating frames at its heart offers a potential for

THE DESIGN OF SOFTWARE MANUFACTURING TOOLS

In order to realize the potential of frames, especially with regard to maintenance, a software development environment has been created, called Computer Aided Programming. CAP is fundamentally a manufacturing paradigm, in which standard frames are the standard subassemblies, various frame generation steps are the processing operations on basic components (raw materials) to produce fabricated parts, and the CAP processor operating on the SPC frame is the process of final assembly with any custom options.

The Role of Languages

Our industry continues to proliferate languages unabated, and this is both necessary and desirable. The creation of each language is motivated by a desire to reduce the effort of solving, in computer executable form, some class of problems. But does this mean we can eliminate the programming?

In Reference 5 the following definitions were developed: Problem solving is fundamentally a process of finding or composing a suitable function (1) whose domain is the problem's input information, (2) whose range is the goal of the problem (i.e., the desired output), and (3) whose function is consistent with other problem constraints.

Playing chess is an example of problem solving. The domain of a chess function is the set of legal board positions. The range is the set of legal moves associated with each position. The constraints include the time available to select a move, the need to find a "good" legal move, any memory of what moves were "good" in past games and so on.

Programming is a form of problem solving by function composition, in which one must deal with either the order of composition, or the interfacing of component functions, or both. At one extreme, selecting from a menu is an effective way for nonprogrammers to solve their problems. At the other extreme, selecting assembly language instructions will solve an interesting problem only with a great deal of programming effort.

By distinguishing problem solving from programming, it becomes possible, with respect to a given class of problems, to group language expressions into three levels: underspecified, optimally specified, and overspecified.

Optimal specification languages

A language is said to optimally specify a function space (and hence an associated problem class) if and only if:

1. The language is isomorphic to the function space; that is, each distinct function is denoted by only one distinct expression, and only the functions in the space are expressible.
2. The degrees of freedom (constraints) are independent, optimally specified subspaces (of constants, variables, or functions).
3. The language's well-formed expressions are the "most compact" with respect to all languages satisfying (1) and (2).

In practice, this definition is weakened. Part (1) is approximated by first designing the language to be virtually one-to-one, then assuming the function space (implied by the language's semantics) to be what was "really meant" by the solutions of the original, unformalized problem class. Part (2) is approximated first by striving for as much independence as possible, then by applying as many context-sensitive error tests as are practical to any remaining dependent degrees of freedom. Finally, Part (3) is ignored as long as the language users are happy.

It turns out that such "weak optimal-specification" languages are a realistic approach to problem solving without programming. Functions usually can be defined simply by grouping the names of some subfunctions under a new function name, without regard to the order in which these subfunctions are performed and without regard to how these subfunctions must communicate with each other. Their compilers are called code generators because each generator plays the role of a programmer, converting a declarative, optimal specification into procedural, overspecified code, which itself must be compiled. Examples of this type of language as used in CAP are described in this paper. As has been noted, CAP design principles require the generated code to be in the form of frames.

It should be clear that the properties of optimal languages permit maintenance efforts to be minimized, provided that the resulting programs can be produced automatically.

Underspecification

An underspecification language is like an optimal-specification one except that the relationship of well-formed expressions in the language to the possible solution functions is one-to-many. There may be many degrees of freedom that play secondary roles in the structure of the overall function space. There may be several functions, each expressible in a different language, which must be combined, but whose degrees of freedom intersect or are interdependent. In these situations, an underspecification language can be used to quickly "broad brush" the major functional features of the solution. The code generator then employs heuristics to specify one solution function at the optimal level that is reasonable and consistent with any overlapping degrees of freedom.

Thus, the underspecified level is the prototyping level, feeding the optimal level where the life-cycle maintenance efforts are performed. Again, the key requirement is that the software manufacturing tools automate the flow of specifications between levels.

Overspecification language

In an overspecification language, the relationship of well-formed expressions to functions is many-to-one, and properties (2) and (3) of an optimal language do not hold even weakly. Overspecification languages are ubiquitous. For example, every computer's binary or assembly language lacks the syntax to express directly the right degrees of freedom for most of the problem classes to which the machine is applied. So programming, which is often done by a compiler, is inevitable at this final stage of problem solving.

To date, virtually all software maintenance has been performed at the overspecified level (for reasons discussed earlier). This is a significant factor in increasing the maintenance effort required. Provided that the software environment is one where a homomorphic map from the optimal to the overspecified levels exists, an order-of-magnitude reduction in life-cycle maintenance effort can be expected based simply on the reduction of code to be maintained.

To sum up the role of languages, whenever a useful function space can be defined by an optimal specification language, programming can be relegated to the computer. To further enhance problem-solving leverage, multiple underspecification, front-end editor-generator pairs can be built that create optimal specifications. These expressions are processed in turn by editor-generator pairs and create programs at the overspecified level, but maintain them at the optimal level.

Any special-purpose, custom functionality is kept in the SPC frame, which directs the CAP processor in its final assembly tasks of building or rebuilding the complete source program, then compiling and linking it into executable form.

The Role of Frames

Frames are used to formalize the common intermediate stage in the program construction process, prior to the frames being combined and customized into a single program (function). There are two reasons for having this stage. First, recognizing the open-ended nature of problem solving, an extensible library of standard frames and templates, together with generated frames, can support custom programming for any problem. Second, the ability to mechanize the assembly of a program, given the diversity of its components, depends on bringing them to a common notation.

Standard frames

As problems are discovered to be related, a standard frame can be evolved to span the implicit function space. Each frame represents a functional, whose domain defines (using the COPY and REPLACE commands) the degrees of freedom appropriate to the class of related problems, and whose range (all possible instantiations of the frame text) is the corresponding function space. By fixing those degrees of freedom in various ways, various problems in the class can be solved without programming.

This is not to say that programming has been eliminated. Usually real problems refuse to confine themselves to neat, predefined classes. Accordingly, a frame's breakpoints and SELECT clauses constitute open-ended degrees of freedom, where solutions can be arbitrarily extended, if necessary.4

Standard frames are used whenever the function space is too limited in scope or usage to warrant a new optimal
specification language. This approach to problem solving is implemented by using templates. A template is an uncustomized SPC frame, and usually spans a hierarchy of frames. It collects in one linear list (a file) all degrees of freedom appropriate for a useful class of problems. The replacement strings, subfunction selection choices, and insertion points for the frames in the hierarchy constitute a fill-in-the-blank method of customizing the program. Thus, templates and frames together permit problems to be solved in a manner that progressively reduces traditional programming to a minimum, given the open-ended nature of real problems.

To the degree that system design expertise can be stored inside the system, the SPC frame can itself be created by designer tools working at the underspecified level.

Generated frames

Certain function spaces have degrees of freedom too dynamic to be represented by fixed, standard frames. Well-known examples are screen and keyboard interfaces and report definitions. For these cases, optimal languages can be developed in association with frame-writing generators.

By generating frames instead of raw source code, open-ended (programming) degrees of freedom become available. Such degrees of freedom are required in the overall problem class, but should be suppressed in the various optimal specification languages. Further customizing can be specified via an SPC without the hand editing or restrictive user exits associated with conventional generators. Basically what has happened is that the editing that would otherwise be necessary to properly customize the generated code has been mechanized. In so doing, we gain both an assembly line style of constructing programs and an ability to maintain the program using its optimally defined pieces (rather than its overspecified code).

Anatomy of a CAPtool

Figure 1 depicts the flow of specifications from the underspecified or designer level, through the optimally specified or customizer level, down to the overspecified or source and object levels. Life-cycle maintenance is performed with the customizer (special purpose) editors. Please note that where reference is made to screen and report specifications, these are examples of optimal-specification languages with respect to the problems of commercial data processing. A CAP tool may use either, both, or neither of these languages, as well as other notations, if the problems warrant.

AN ACTUAL CAP SYSTEM

At Netron Inc., a CAP system has been developed for use on WANG VS computer systems applied to commercial data processing using COBOL. The following reflects current functionality and some soon to be released tools.

Underspecified Level Tools

1. CAPinput—for building interactive file maintenance and data entry programs
2. CAPoutput—for building report programs based on general data selection criteria
3. CAPfile—for building general file-to-file transforms and interfaces

These three tools are each structured as shown in Figure 1. Specification of a complete program requires that an analyst answer a small number of questions (most of which have defaults).

Optimal-Specification-Level Tools

1. CAPscreen—for designing and maintaining interactive screen and keyboard functionality
2. CAPreport—for designing and maintaining report functionality
3. CAPframes—a library of standard frames

The (weakly) optimal notations are used by designer tools and by analysts, either in conjunction with underspecified-level tools or independently.

A complete description of these languages is beyond the scope of this paper. Very briefly, independence of degrees of freedom is typified by having screen (report) layout facilities completely independent of the attributes of each screen (report) variable. On the other hand, some degrees of freedom are not completely independent. For example, if a variable on a screen is declared as having run-time error checks, and is declared as not being assigned to an internal variable after the operator enters it at run-time, then these two degrees of freedom are in conflict (and the conflict must be resolved).

The tools themselves generate frames from the optimal specification. These frames in turn make extensive use of the hierarchy of available CAP frames. Because the frames are written using general-purpose (but overspecified) COBOL, the programmer has exact control over the “fine tuning” his particular application may need in order to convert a functional into the required function.

The CAPframes are the heart of the CAP system. Each frame implements a useful function space whose patterns have

Specific screen & report specifications
Fill-in-the-blank report & screen customizers
Fill-in-the blank designer
Specific Needs
Generate Custom Frames
Splice Compile Link
Custom Executable Program
Specific frame specifications
Fill-in-the-blank SPC frame customizer
Model
Solution
Frames

Figure 1—CAP flow specifications
been recognized by their appearance in several programs. The frames are organized into a taxonomy that guides the problem solver to the relevant functionality.

DISCUSSION OF TOOL USAGE

Types of Users

The consistent application of the under-optimal-over design principle offers access potential to the industry's three major user groups: end-users, analysts, and programmers. In CAP's current implementation, it is an analyst-oriented software manufacturing system. The focus has been to provide tools that aid in the manufacture of larger, more complex systems.

CAP could be designed for nonprogrammers, but few are inclined to cope with the open-ended applications to building and maintenance that are CAP's main strengths. Most people like driving cars and some even enjoy fixing or rebuilding them. But who wants to design and manufacture them?

Because CAP is a manufacturing paradigm, most of the benefits stemming from the organization of a conventional manufacturing enterprise become available to data processing shops. In particular, the frame-engineering department is quite analogous to a conventional engineering department. A useful division of labor is created. Those responsible for designing and maintaining the organization’s inventory of standard software components (frames) can work independently from those charged with getting the application software products out the door. The benefit of having centralized standards control is obvious.

Rapid Prototyping

While not part of maintenance as such, rapid prototyping is a very desirable feature of any software development system. Moreover, it is important to ensure that rapid prototypes do not lead to maintenance nightmares.

Conventional wisdom, stemming from the software disasters of the sixties and early seventies, has firmly entrenched the hedging policies of preparing exhaustive feasibility studies, formal requirements definitions, structured walkthroughs, and the like. Often, the time and costs to plan a system are greater than the costs of building it. In turn, the specifications are usually out of date by the time they are finally approved, and the end-users still don't really know what they are getting, or if what they get is what they need. Another danger is that it is so easy to specify features that turn out to be much more difficult to implement than they are worth to the user. In short, the institutionalized policies of large data processing groups are no small contributor to the enormous applications backlog.

Conventional wisdom can now be made wiser. CAP tools can write formal specifications that are understood both by people and by computers, and then convert the specifications to equivalent programs. We can now adopt the attitude of “what you see is what you get,” and even let small prototypes constitute part of the design specification.

End-users can “kick its tires” and iteratively guide the specifications. The implementation team can provide specific, detailed arguments as to why certain features should or should not be in the system, and can more accurately estimate the cost of a system’s implementation based on deviations from the organization’s current frame inventory.

Productivity and Quality

Using a tool such as CAPinput typically requires that the user spend a few minutes at the underspecified level. Without further customization, an executable program is available shortly thereafter. The following is the summary from a detailed case study that analyzes the actual use of CAP.

Case study: The manufacture of the Canadiana requisition system

Canadiana Garden Products Inc., is a subsidiary of NOMA Industries Ltd. In March 1983 Canadiana employed Netron Inc., to create a computerized system to replace Canadiana’s manual requisition system. The system was created using CAP and is run on a WANG VS computer using interactive terminals. The system allows requisitions to be created, maintained, displayed, searched, authorized, ordered, recorded, and reported upon.

After the first week, enough of the system had been prototyped that the client recognized serious design problems. The system was subsequently redesigned and put into production by the end of the third week.

Sixteen programs were written using CAP tools to create and control the interaction of the 22 screens and three reports through which the requisition system is operated. CAP tools enabled the author to create the requisition system by writing less than 10% of the total COBOL lines needed.

One method of judging the effect on maintenance with and without CAP tools is to compare the total number of lines of submitted source code in the entire requisition system with the number of hand-written lines. Purely comment lines were discarded.

The results show a more than 10:1 reduction in lines of COBOL to be maintained. Of the 34,000 lines of submitted code contained in the 16 programs of the requisition system, only 3,000 lines were written by hand.

The following table shows, for each of the 16 programs forming the requisition system, the number of lines hand written in the SPC frame, in the generated frames, in standard frames, and in the total submitted to the COBOL compiler.

Quality

Of course, the issue here is not merely to show that there is much less code to maintain. Further analysis of the manufactured programs show that they are more consistent with respect to user-interface and structured program style, more reliable, more functionally complete, and no less efficient than conventional, hand-written programs.
The reason is that the standard frames and frame generators are highly seasoned components in the course of whose evolution many improvements and optimizations have been made. The cumulative effects are capital assets (no pun intended) that yield a return on investment in every incorporating program. Programs hand-written from scratch have no chance to acquire the quality and thoroughness that is the hallmark of a good frame. 15

**Life-cycle Support**

As previously indicated, by storing all source code customizations in one spot, factored away from both standard and generated frames, typical program maintenance is collapsed from 50–60 pages of source listing to two or three pages. By having the code generators emit frame code that can be customized automatically, the declarative specifications also support the life cycle maintenance of the programs in a very convenient manner.

Frame maintenance

As with software, frames change through time. Standard frames tend to be relatively stable since they rapidly become seasoned through frequent reuse. But because they are functionals, they are able to absorb arbitrary amounts of change (including complete rewrites) without risking any previously written program. It is easy to arrange that the range (function space) of a new version of a functional be a superset of the previous version’s range simply by providing a version control parameter governing a SELECT clause.

This still allows the improved functional to recreate all old functional versions. An old program’s SPC, unaware of subsequent changes, references the frame hierarchy with its old version symbol (if any), and gets exactly the same code it has always gotten, even though new programs may get something quite different (the template always contains the latest version symbol).

This does not mean that frames and libraries become more cluttered than in conventional shops. Conventionally, complete copies are kept of all versions (using distinct names), even though only small changes might have been made. Frames keep an automatic audit trail of the version differences, with only occasional rewrites done to eliminate clutter. The obsolete (but still active) versions are placed in a separate library, again to eliminate clutter. Internal version references automate the retrieval of the correct version. Thus, a single external name is common to all versions and less space overall is actually required.

**CONCLUSION**

It is important to realize that programs are models: deliberate approximations to an elusive and ever-changing external reality. Models are useful because they exploit a simplified representation. We know that Newtonian physics is wrong, yet we never use Einsteinian physics when programming everyday calculations. A payroll system has an extremely skimpy model of the human beings on file, but it is quite appropriate for the intended purpose.

From this perspective, development and maintenance are two sides of the same coin. Converging a software model to a useful approximation is called development. But the model also must be updated periodically in light of changing circumstances, and this is called maintenance. The payroll system must quickly incorporate each change to the income tax laws to the extent that its model of those laws becomes invalid.

The recent development of a software manufacturing paradigm has set the stage for changing our cottage industry into a mature technology. By unifying the techniques for program construction and maintenance, each productivity gain can simultaneously benefit both.
REFERENCES


