Writing less code—An approachable ideal

by NAOMI LEE BLOOM
American Management Systems, Inc.
Arlington, Virginia

ABSTRACT

We are being inundated by a sea of unsatisfied user expectations. This growing, and sometimes frightening, backlog of application development requests has been much discussed but little reduced. One almost universal approach to reducing this backlog has been to try to improve the productivity of our scarce technical resources (programmers, systems analysts, etc.). A more promising approach to meeting user application needs may be to substantially reduce the amount of new code needed to satisfy these needs. It takes no great insight to become convinced that, other things being equal, the less code written to achieve a specific level of systems support, the less risk, cost, elapsed time, and frustration must be accepted by the organization. This paper presents a brief survey of some common, and some less obvious, applications-enabling techniques. Two of the most promising techniques, foundation software and adaptable application packages, are more fully described in separate papers.

INTRODUCTION

We are being inundated by a sea of unsatisfied user expectations. This growing, and sometimes frightening, backlog of application development requests has been much discussed but little reduced. And the invisible backlog, described by Martin[1] as the unspoken (and perhaps not yet dreamed of) desires of our users, ensures that this problem is not likely to diminish.

One almost universal approach to reducing this applications backlog has been to try to improve the productivity of our scarce technical resources (programmers, systems analysts, etc.). Productivity techniques, such as structured programming, structured analysis, regression testing, and interactive programming, have been widely adopted, but still the backlog grows. Clearly, even quantum leaps in the productivity of scarce technical resources will not eliminate this backlog.

A more promising approach to meeting user application needs may be to substantially reduce the amount of new code needed to satisfy these needs. Such application-enabling techniques, to use a phrase that seems to have originated within IBM, are intended to reduce the amount of new code written rather than to merely expedite the production of new code. It takes no great insight to become convinced that, other things being equal, the less code written to achieve a specific level of systems support, the less risk, cost, elapsed time, and frustration must be accepted by the organization.

It is important to note, however, that other things are usually not equal. Many of the techniques described in this paper substitute increased consumption of computing resources for reductions in the personnel resources needed to achieve a certain level of user support. As hardware costs and the resulting price performance ratios continue to improve while competent analysts, programmers, and related computer professionals grow more scarce and more expensive, it is a reasonable business judgment to explicitly trade off increased hardware resource consumption for man-hours of development and user time. Such trade-offs must not compromise satisfying user needs and they must be carefully evaluated for each application so that the system overheads associated with various packages and tools do not catch the project team unawares.

This paper presents a brief survey of some common and some less obvious applications-enabling techniques. Two of the most promising techniques, foundation software and adaptable application packages, are more fully described in separate papers by Curtis[2] and Woodward and DiGiammarino.[3] If properly used, the techniques presented here will reduce not only the amount of new code written by any one organization, but also the aggregate amount of new code. However, even if these applications-enabling techniques are fully applied, some new code will have to be written, and that should be done in a highly productive and orderly way. While this paper and those by Curtis and Woodward and DiGiammarino focus mainly on traditional business applications, applications-enabling techniques may be applied equally to the development of scientific, system-oriented or personal applications.

THE SPECTRUM (OR HIERARCHY) OF APPLICATIONS-ENABLING TECHNIQUES

There is nothing very mysterious about finding ways to write less code. You can do any of these things:

1. Convince the user not to want (or to need) a new application.
2. Reuse old code—your own or someone else's.
3. Use simple tools (remember how levers work?) to multiply the work value of any code you do write.
4. Get someone else, perhaps your users, to write the code for you.

The key to successful applications enabling is to build these very simple maxims into your systems development life-cycle methodology. Applications development or even package installation projects should not be initiated, unless the new application is really needed. And in every stage of the life cycle, you must ask yourself what alternatives exist to developing new code. Thus, applications-enabling techniques parallel, in some sense, the applications development life cycle.

In the earliest stage, frequently called the business systems or strategic systems planning stage, you must ask the fundamental question of whether this application is worth doing at all. As the process goes forward, you should be asking the following types of questions:

1. Has this application been developed before? If so, there may be some old code that you can reuse.
2. Does this application lend itself to the use of simple tools? Either tools that someone else has developed, or that you yourself could develop?
3. Does this application lend itself to the end user-written code that is characteristic of many data manipulation and analysis applications?

By asking these types of questions at the appropriate points in the systems development life cycle, you can take advantage of the many techniques available to minimize the amount of new code written. The remainder of this paper explores these tech-

niques in the order in which they tend to present themselves in the life cycle.

DO NOT DEVELOP UNNECESSARY APPLICATIONS!

The most obvious solution to our problem of how to write less code is to eliminate from the backlog all but the essential (translation: justified) applications. Strategic systems planning (also known as business systems planning) is the process by which an organization identifies and prioritizes its major systems development objectives. By explicitly aligning the applications development priorities with the organization's business strategy, we take a critical first step toward reducing the amount of new code to be written.

Although there are many flavors of strategic systems planning described in the literature, the objectives identified by IBM[4] in their business systems planning methodology are representative:

1. To provide management with a formal, objective method for establishing priorities for corporate information systems without regard to local interests
2. To ensure that scarce development resources are committed to those systems that have a long life, thereby protecting the systems investment, because these systems are based on the business processes that are generally unaffected by organizational changes
3. To provide that the data processing resources are managed for the most efficient and effective support of the business goals
4. To increase executive confidence that high-return, major information systems will be produced
5. To improve relationships between the information-systems department and users by providing for systems that are responsive to user requirements and priorities
6. To identify data as a corporate resource that should be planned, managed, and controlled in order to be used effectively by everyone

By ensuring that we develop only those applications whose relevance to the organization and benefits have been rigorously examined, we have made the first breakthrough toward minimizing the backlog of unsupported application requirements. To repeat, if you develop no unnecessary applications, you will not be called upon to write (and maintain!) worthless code.

REUSE OLD CODE—YOUR OWN OR SOMEONE ELSE'S!

Where an application is justified, there are several possibilities for developing it without writing any code or by writing only a small amount of (it is hoped) simple code. Application software packages have been available for nearly 30 years, and many routine business (and system, e.g., sorting) functions are very adequately supported by such packages. In addition, many of your business functions, such as edit routines for specific data elements, have probably been programmed many times within your own organization. Before deciding that an application is so unique as to obviate using any existing code—a common attitude among many in-house analysts and users—consider the many flavors of software packages and reusable in-house code.

Currently available commercial applications software can be divided into three general categories:

1. Traditional software packages, which perform a well-defined set of functions with minimal installation options
2. Contemporary software packages, which perform a well-defined set of functions subject to many table-driven, user-defined, installation-specific options
3. Adaptable software packages, which perform a flexible set of functions subject to many table-driven, user-defined, installation-specific options

Traditional Software Packages

Initially, application packages were really custom software that the developer chose to share, albeit for compensation, with others. Early package vendors often sold their essentially custom systems with minimal documentation and installation support. Installing such a package required the buyer to modify code even to support the most obvious installation-specific requirements, for example, to change report headings to contain the buyer's company name.

The buyer of a traditional software package (and there are many currently being sold) gets some clear benefits: On *short* notice, he is able to obtain and install *debugged* code that performs some *well-defined* set of functions after *minimal* source code modification; and he pays a *far* lower purchase price than he would for equivalent custom development. Needless to say, the italicized adjectives are subject to the buyers' personal evaluation. But, in theory, the risks, cost, elapsed time (and, hopefully, frustration) of purchasing a traditional package are less than in doing the application from scratch.

That's the theory, but the benefits are often not realized in practice. With a traditional package, every user-specific requirement, from report headings and formats to variations on common algorithms, resulted in modifications to foreign (at best) or (more often) incomprehensible and undocumented source code. Although traditional packages remain an appropriate technique for writing less code, their inflexibility can be frustrating.

Contemporary Software Packages

Eventually, modern (that is, scientific) approaches to software design, combined with the recognition that even the most flexible software buyer had some unique requirements, led to a new type of package. Written to be generalized, commercial software products, contemporary packages (my term) have the following:

1. Well-documented source code constructed to provide low-risk user exits, that is, specific points at which user-

written subroutines can be inserted without disrupting the program flow or voiding the vendor's warrantee
2. Reference tables that remove from the source code such frequently customized functions as report headings and, in some cases, formats; message code literals and severity levels; data element names, field lengths, data types, and edit rules, including pointers to other reference tables of valid values and code translations; parameter values, for example, process scheduling dates, current withholding tax percentages, and airline overbooking percentages; calculation algorithms—sophisticated packages exist for which not only the parameter values but also the operators and calculation bases are table-driven; and coding structures, for example, the chart of accounts or organizational structure
3. A formal installation process, including sample conversion programs, job streams, and other code-reducing aids

Like traditional packages, the purchase and use of contemporary application packages generally reduces the costs, risks, elapsed time, and personal frustrations of meeting system support needs. However, there is always a price for flexibility. Sophisticated reference tables can require considerable loading and maintenance effort, although this approach is far less risky than modifying source code. Plus, users can often be roped into taking responsibility for loading and maintaining most of the tables.

More important, from the perspective of containing cost, risk, and elapsed time, the availability of options means someone (usually a cast of thousands) must analyze, document, recommend, evaluate, and (it is hoped) decide on each desired option. But contemporary applications packages go a long way toward meeting organization-specific requirements without developing new code.

One further note before moving into a new area of packaged software. As mentioned earlier, there is usually a hardware resource consumption penalty for using generalized software. Contemporary packages which favor table-driven processes over hard-coded processing, exact a stiffer penalty in this regard than do the traditional packages.

Adaptable Software Packages

One of the most interesting recent developments in software packages is the trend toward building groups of related modules that can be reconfigured to suit various application requirements. One such package was developed to support credit card collection activities (CACS). Recognizing that credit card collections are a specific example of a generic class of applications, that is, case tracking, scheduling, and state-processing functions, the software was developed to automate these generic functions. With a combination of powerful reference tables, including process control tables, and program modules that can be combined in various ways, CACS can be used with minimal source code modifications to support a broad class of user requirements. The paper by Woodward and DiGiammarino[3] describes CACS and the concept of adaptable software in more detail.

Solve Part of the Problem With Old Code

Access to mathematical and statistical subroutines was an early enhancement to many compilers. In contemporary systems, active data dictionaries often drive data element edits from a common or shared subroutine. Indeed, most data processing shops have developed some standard source language components, perhaps as COPYLIB equivalents, that can be reproduced in various applications at minimal risk, cost, and so on. When we discuss using simple tools to leverage the value of any newly written code, one point that we'll develop further is the idea that the design effort must explicitly focus on identifying common processes that could be programmed once rather than needing to be redone in multiple programs or systems or installations.

To take full advantage of existing code (or to identify common processes for initial development), the life-cycle methodology must emphasize answering the following questions at each level of the design:

1. Have we ever automated this function before? Even a relatively minor function, such as a date edit, can be programmed once, even as a generalized routine, at far less cost than having every programmer do his own thing. At a minimum, your effort for the year 2000 will be greatly simplified if you've been smart enough to incorporate a single date routine into all your systems. It's essential to evaluate each process in this way as a potential candidate for the organization's library of standard software.
2. Will we ever need to automate this function again? Date edits, translations of organization codes into their correct names, report headings, and many other common functions appear in nearly every business application. Do them once in a generalized way, at somewhat greater cost initially, and use them forever.

Unless the deliverables at each stage of the development life-cycle explicitly address the issue of standard software (reusable code), many opportunities for writing less code will be missed—now and in the future.

USE SIMPLE TOOLS

There are two general approaches to multiplying the value of any code you do write:

1. Extension software, which uses your (it is hoped) simple code written in the tool's own command language as the input from which it creates (by translation, compilation, assembly or one of several other extension techniques) very substantial functionality; and
2. Conservation techniques, which are a formal set of design techniques that look for the common functional elements in an application in order to develop a single implementation of these common functions for use across the application.

Reusing date routines is a very simple case of conservation. In this section we'll explore more sophisticated examples of the two approaches just mentioned.

Extension Software

When you write JCL to unleash the power of IBM's various operating systems, you are using extension software to minimize the code you must write. My earliest programs in machine language on an IBM 1401 had no such extenders, and we wrote our own tape reads and printer writes. Now, every use of a system utility from within your application, that is, calling the COBOL internal SORT, leverages a few utility commands to perform considerable work.

Thus, the universe of extension software ranges from the old and familiar to the new and still developing:

1. Utility programs that provide system or housekeeping functions
2. Report writers and inquiry languages, including graphics packages
3. Database management systems with which you use simple commands in the application programs to invoke powerful data handling, edit, storage, and access capabilities
4. Screen generators
5. Data management and analysis tools, for example, SPSS and SAS
6. Application generators
7. Very high-level languages.

The boundaries among these tools are not clear-cut, and many of them can be used by nontechnical persons to achieve the ultimate shifting of application development responsibility. All of these tools hold the same promise of providing complex software to leverage simple commands into powerful functionality, and many deliver on this promise.

However, there is a serious fly in the ointment regarding the use of extension techniques. We are now being inundated in a sea of command languages, specialized syntaxes, and easy-to-learn, English-like, languages. There's not even agreement on how commands are delimited! Until considerable standardization occurs, taking advantage of even a small set of these tools will impose a serious training burden on any organization. And many professional programmers and users will resist using these tools because they quite reasonably perceive that the cost of mastering them is too high.

Conservation Techniques

Perceptive analysts and designers have always recognized common functions in their application specifications, but the process of doing so was largely informal. On many business applications, there are a rather large set of common functions that lend themselves to a common software approach. At American Management Systems, Inc., we have incorporated into our life-cycle methodology a quite formal process for searching for these common system elements.

The decision to build an application around a base of common software modules must be made explicit quite early in the design process so that all further effort can be efficiently directed. We call the resulting software, which provides common services to the rest of the application, foundation software. The foundation software approach to developing large application systems is described in detail in the companion paper by Gary Curtis.[2]

GET SOMEONE ELSE TO WRITE THE CODE!

End-user computing is not a new idea. In the beginning of computer history, programming was the adjunct function of scientists, engineers, and mathematicians who were trying to use the great behemoths to calculate ballistic missile trajectories and to develop software for other, equally forbidding problems. In my early days as a programmer, accountants were still developing the first automated payrolls, general ledgers, and banking and insurance systems. Professional programming is less than 20 years old, so why do we now treat end-user computing as a state-of-the-art development?

One reason is that, until now, whoever approached the computer was forced to learn computer-speak—at great personal sacrifice. If we believe the advertisements for various end-user computing tools, the professional programmer may soon focus solely on core production systems and tool development, leaving to the user development of most data extraction and analysis (MIS) systems. But the future has not yet arrived.

Many of the simple tools described in this paper can be used by a nontechnical person after some training, and the growth of information centers attests to the availability of user-friendly tools. Fourth generation languages, for example, RAMIS II or FOCUS, are advertised as powerful tools for developing whole applications from simple commands. The proliferation of personal computers attests to the user orientation of such tools as VisiCalc. Clearly, if the user can directly translate his unspoken (or never clearly spoken) information requirements into a working system, he won't have the DP staff to kick around any more.

CONCLUSION

Computers are worthless without programs, be they software, firmware, or part of the hardware itself. People still write programs, and people are expensive, unpredictable, and fragile. If only to sell more computers, the hardware vendors would welcome (support and probably give birth to) any approach to program development that used more computer resources to free scarce personnel to develop new applications that used more computing resources. Since they develop many of the packages and tools and generally corner the market on really superb professional programmers, software vendors certainly favor the techniques described in this paper. Corporate users and DP managements are also on board the write-less-code bandwagon. So why does the applications backlog continue to grow?

1. In-house programmers would rather write programs (not to mention design whole systems) than load tables for a contemporary package or do report writer setups. Perhaps we need a new category of DP aide or paraprofessional who sees using tools as a desirable job description?
2. Without standardization in grammar or syntax, currently available tools produce a Tower-of-Babel effect wherever they go.
3. Many users have terminal block, not to mention various other phobia, that limit their ability to use any computing tools.
4. Computing resources, while obviously getting less expensive, are *not* free. Their acquisition, which always occurs in large increments, is a more visible expenditure to the organization than is the cost (opportunity cost) of unfulfilled application needs.

Time is clearly on the side of the approaches described in this paper, but I wouldn't yet discharge my COBOL programmers nor declare that all user needs can be satisfied by their new Apples! As in all things, a balanced mix of these new techniques with more traditional application-development strategies will produce the best results.

REFERENCES

1. Martin, James. *Applications Development Without Programmers*. N.J.: Prentice-Hall, Inc., 1982.
2. Curtis, Gary A. "Foundation Software: A Significant Improved Approach To The Development of Large Application Systems." *AFIPS Proceedings of the National Computer Conference* (Vol. 52), 1983.
3. Woodward, Mary, and Peter DiGiammarino. "A Case For Adaptable Applications Software." *AFIPS Proceedings of the National Computer Conference* (Vol. 52), 1983.
4. IBM. "Business Systems Planning—Information Systems Planning Guide," GE20-0527-3, 1981.