Western Michigan University ILL

ILLiad TN: 539107

Borrower: OYR

Lending String: *EXW,DHU,KUK,NBU,CLZ

Patron: greene

Journal Title: Journal of special education technology.

Volume: 12 **Issue:** 2
Month/Year: 1993 **Pages:** 149-163

Article Author:

Article Title: Woodward, J; 'software development...'

ILL Number: 73585537

Call #: LC4023 .J68
Location: ERC

ARIEL: 192.68.202.36)

ARIEL

Charge
Maxcost: 0

Shipping Address:
OREGON CTR FOR APPLIED SCIENCES
LIBRARY ILL ATTN ELAINE SHUMAN
1715 FRANKLIN BLVD
EUGENE OR 97403

Fax:
Email: elaines@ori.org

Software Development in Special Education

JOHN WOODWARD, University of Puget Sound

JOHN NOELL, Oregon Research Institute

Abstract

Over the last ten years, special educators have developed a variety of software programs which attempt to address the unique needs of mildly handicapped students. Much of this work has been funded by the Division of Innovation and Development in the Office of Special Education Programs. This article examines software development techniques used by many of these special educators through the perspective of the software lifecycle. This perspective, which is used intimately by professional software developers, is rarely discussed in the special education technology literature. With rapid advancements in computers, particularly the graphical user interface, the authors argue that traditional programming techniques are becoming less and less viable for software development within the typical funding levels and timelines of federal initiatives. Instead, a trend toward the use of commercial software tools is viewed as inevitable, given these constraints. The article concludes with a brief description of two recently completed, technology-based assessment programs.

For over ten years, the Division of Innovation and Development (DID) in the U.S. Department of Education's Office of Special Education Programs has supported a wide range of initiatives designed to enhance the state of technology use for handicapped individuals. Technology researchers and developers have studied the effectiveness of computer-assisted instructional programs for special education students; probed potential applications of technology currently used in business and the military for the handicapped population; developed an array of compensatory applications for physical, sensory, or cognitive impairments; and designed innovative instructional, assessment, and multimedia systems. There have also been an untold number of other computer-based research and development activities in special education which have not been funded by DID-directed technology competitions.

A sustaining impetus for these activities comes from survey and observational studies conducted during the early 1980's. Researchers found that elementary and secondary mildly handicapped students spend a disproportionate amount of time on drill and practice programs (Becker & Sterling, 1987; Cosden, Gerber, Semmel, Goldman, & Semmel, 1987; Rieth, Bahr, Okolo, Polsgrove, & Eckert, 1988; Semmel & Lieber, 1986). Where technology was not used for drill and practice, the main intent of computer use with special education students was to improve motivation, self-confidence, and self-discipline.

A subtle, but more germane finding in these studies was the modest variation in the *type* of software used (Cosden et al., 1987). It was not uncommon to find students of all academic abilities practicing the same math facts, vocabulary, or decoding software programs. Rieth et al. (1988) attributed part of the

infrequent use of microcomputers at the secondary special education level to the lack of diverse, and hence, appropriate software.

In a broad sense, commercial software commonly available to special education students has failed to address their individual academic needs. Federally sponsored projects, when viewed in this light, have provided alternatives typically unavailable from large software publishing companies. Some special education software programs developed under these monies have reached the commercial market, while others have remained in a prototypic form. Nonetheless, a common theme pervading special education software projects over the last decade has been the incorporation of assessment (Deno, 1990; Fuchs, Fuchs, Hamlett, & Stecker, 1991) or instructional design (Carnine, 1989; Hofmeister, 1990) features which address the *unique* needs of handicapped students.

This article discusses the software tools commonly used to develop computer-based programs for special education students—specifically programs developed for assessment and instruction of mildly handicapped students. The limits of DID support (both in actual dollars and funding timelines), the paucity of commercial developers for the special education market, and the complexity of today's computers make the selection of appropriate software development tools a fundamental issue for the special education technologist. This observation remains valid even if programs are only developed as prototypes. Often the decisions as to which tools are best suited for successful software development are as complex as the assessment and/or instructional design techniques to be embedded in the program. This article concludes with a brief description of two DID-funded technology projects which, in one case, uses tools that until recently were only available in business and the military.

TOOLS FOR SOFTWARE DEVELOPMENT

Programming Languages

Programming languages such as BASIC, FORTRAN, and Pascal are the most traditional method of developing educational software programs. These "high-level" languages enable the programmer to manipulate the basic operations of a computer in a more comprehensible manner than binary or assembly language code. High-level languages, then, represent an important historical step in expediting the development of a software program. They move the programmer closer to the "problem space" (i.e., what software and curriculum developers want a program to do), and further away from the underlying machine hardware. The logic of making programming environments increasingly comprehensible can be extended to even higher-level languages such as authoring languages, authoring systems, and expert systems, all of which will be discussed later.

First and second generation high-level languages such as FORTRAN and BASIC have been particularly useful vehicles for novice computing. They have provided an economical, iterative means of processing large sets of data for relatively simple programs. However, computing in the 1970's and, most certainly in the 1980's, shifted toward more structured programming languages such as Pascal, and C because of their modular structure and transportability across computer platforms. Object-oriented programming is the most recent advancement in high-level languages for enhancing modularity.

Writing instructional and assessment programs in a traditional, high level language—a path followed by many special education technologists—has obvious advantages. It facilitates the development of programs which are closest to educational specifications. CAI programs in math facts (Goldman, Pellegrino, & Mertz, 1988; Hasselbring, Goin, & Bransford, 1988), spelling (Fuchs, Hamlett, & Fuchs, 1990; Hasselbring, 1982), sight words (Lally, 1981), vocabulary (Carnine, Granzin, & Rankin, 1983; Johnson, Gersten, & Carnine, 1987), health education (Carnine, Lang, & Wong, 1983; Woodward, Carnine, & Gersten, 1988) and reasoning skills (Collins, Carnine, & Gersten, 1987; Engelmann & Carnine, 1983) are but some examples of successfully developed and researched special education software. Typically, these programs contain elabo-

rate branching schemes and conditional forms of feedback.

Comparable development work has also occurred in computer-managed instruction. Computer-based implementations of curriculum-based measurement (CBM; Fuchs, Deno, & Mirkin, 1982; Fuchs, Fuchs, Hamlett, & Hasselbring, 1987; Germann, 1985) have significantly reduced its labor intensive aspects. Data management software that automatically graphs data, applies decision rules, and provides feedback statements summarizing those decisions have led to positive teacher attitudes about CBM as an efficient means of formative evaluation (Fuchs et al., 1987).

The Graphic Interface and Increased Programming Complexity

Software development through higher level languages has been feasible insofar as programs were not excessively complex, and they did not demand an extensive use of graphics. However, software trends over the last two decades have been in the direction of greater complexity and graphics as hardware costs have rapidly declined. Graphical user interfaces (GUIs) such as the Macintosh operating system and Microsoft Windows exemplify this movement.

These environments demand a sophisticated understanding of the machine's operating system. They require the use of ancillary programming tools (e.g., MPW for the Macintosh) with long learning curves, thus necessitating a great commitment of time and effort. With few exceptions, creating programs in these environments requires one or more experienced professional microcomputer programmers.

Few programs have appeared in the special education literature which are written for GUI environments (Gleason, Carnine, & Vala, 1991; Slocum, 1988; Woodward & Carnine, 1989; Woodward, Carnine, Steely, Freeman, & Nospitz, 1988). The demands of these environments highlight the importance of the software lifecycle as a framework for conceptualizing software development, a topic rarely discussed in the special education technology literature.

The Software Lifecycle

Most successful software is developed, disseminate, and revised using a multi-step procedure. These steps have been used as common operating procedures in commercial development environments for decades, and they comprise an implicit framework for professional developers. However, the use of a software lifecycle in special education research and development has been far less systematic. Figure 1 below presents the basic steps or phases of the software lifecycle.

Program planning and specification. The first step is to define the purpose of the software, choose which hardware platform is most suitable, and select appropriate software tools (e.g., a high level language such as C, an expert system shell, hypermedia). For the special educator, this phase requires extensive planning and specification. Unlike print curricula development, for example, which can be revised in an interactive process encompassing isolated portions of the materials, software programming initially requires a much more complete initial layout of the problem space (i.e., exactly what a program should do and when different events should be occur). It is rarely possible to "try out" small portions of a program. Educators who decide to make extensive changes in the way a program works after "tryouts" force extensive changes in the entire computer program, which can severely delay or hamper the software development process.

Data flow design. The second step in the software lifecycle is the design or layout of the data flow. Programmers have traditionally used data flow diagrams or flow charts to accomplish this purpose. The sequence of events (e.g., how and what data flow from one

- Program Planning and Specification
- Data Flow Design
- Coding (Computer Programming)
- Alpha and Beta Testing
- Marketing/Dissemination and Technical Support
- Program Updating

FIGURE 1
Common Steps in the Software Lifecycle

step to the next), the branching criteria, and so forth are fully specified. Data flow diagrams are largely historical methods associated with older, more traditional languages (e.g., Pascal). They may be less useful when applied to artificial intelligence languages.

Newer tools built around graphic environments enable the programmer to rapidly generate "shells" of the different screens in a program, much in the way an advertising agency uses a story board in developing television commercials. These shells, which are critical to educational software development, only display the format of what a user will see at each point in the program and have very little code associated with them. Also, no code links one screen to another in the program (i.e., there is no way to "run" the program at this point). However, this aspect of the software lifecycle is crucial to the final product because it previews the program's interface. This is the first major opportunity for developers to reach consensus on the "look and feel" of the program before the intense coding phase begins. The flow of information between user and computer program is at least as important as the internal flow of data within the program.

Coding. After the overall design is finalized, the third step is the actual coding or computer programming. The first part of this step is to lay out the design of the code, which has evolved from the preceding steps. Segments of the program are written, and in the case of larger programs, the work is subdivided so that several teams can work on the project simultaneously. As portions of the code are written, they are checked for bugs (e.g., typos, syntax errors, logic errors). Although many educators tend to think that coding is the last phase of the software development process, it is part of an iterative process and there is much left to be done.

Alpha and beta testing. The program, once written and assembled, is now available for the first level of testing, called "alpha" testing. As with any complex task involving the creation of hundreds, if not thousands or tens of thousands of lines of code, there are bound to be mistakes. Programmers search for bugs at a larger scale, such as errors in the overall design or subcomponents of the program that don't work together. One of the ultimate goals of this process is to insure that the program is robust (i.e., that it performs faultlessly under a variety of uses and conditions). Depending on the complexity of the program and other variables, repeated episodes of alpha testing following by additional programming may be necessary.

Once the alpha testing and re-programming phase is completed, programs are usually sent to actual end-users for a further period of testing called "beta testing." Again, almost always, actual end-users will encounter problems not foreseen or adequately dealt with in earlier phases. This stage also provides a final opportunity to judge whether or not the developer's intentions match those of a wide range of users. Mismatches or "intent bugs" mean that the program must be modified to satisfy the demands of the eventual user. Programs that analyze a complex array of student data but are difficult to use because data entry is confusing represent one example of an intent bug.

Additional programming is done following beta testing to rectify all problems identified to date. The greater the complexity of the program, the longer each of the preceding phases takes. For special educators, this is a difficult task because there is generally not a widely installed base of users who can sufficiently test the robustness of a program.

Marketing, dissemination, and technical support. The program is now ready for dissemination. Some developers sell their products directly, while others—particularly educators—use publishers or distributors. Although more of the purchase price is retained with direct sales, the time, money, and expertise required for successful marketing may make the publisher/distributor option more desirable. A critical element is the need for technical support. Obviously, the more complex the program, the more likely that end-users will want some kind of technical support. This is a regular feature of large

software programs, something which is rarely provided for in special educational software.

Program updating. Over time, programs are continually updated and, in some cases, rewritten entirely. This pattern of development and refinement is evident in the CBM software (Fuchs et al., 1982; Fuchs et al., 1991). Development platforms have shifted from those written in high level languages to ones based on expert system shells.

The process of developing software is usually complicated, time-consuming, expensive, and the magnitude of the task being contemplated is often underestimated. Companies that are in the business of creating software often have very large teams of programmers who spend years refining a product. Development costs can easily reach into hundreds of thousands or millions of dollars. With large potential markets, such costs may be recouped. In special education, the market is generally quite small and software tends to lack a broad-based appeal so that the amortization of large development costs is not a realistic goal. This problem is exacerbated when programs are developed with federal dollars which are allotted over strictly controlled time periods. Figure 2 portrays the problem of special education software development in the context of professional software development and its lifecycle.

Courseware Authoring Environments

It should be evident that creating programs with high level languages requires a considerable amount of pre-planning, expertise, and iterative development *before* they are ever used in classrooms. The cost and timelines associated with this kind of development have led educators to look at authoring languages and systems as serious alternatives for producing simple drill and practice and tutorial programs (Woodward & Carnine, 1983).

Authoring languages. Authoring languages have been advertised as quick solutions to CAI for educators who are intimidated by high-level languages such as BASIC, Pascal, or C. Languages such as *Pilot* (e.g., Apple *Pilot* and Atari *Pilot*) provide a simplified code for creating modules of instruction. Their "code" or set of commands is simpler and supposedly more intuitive, which means that it

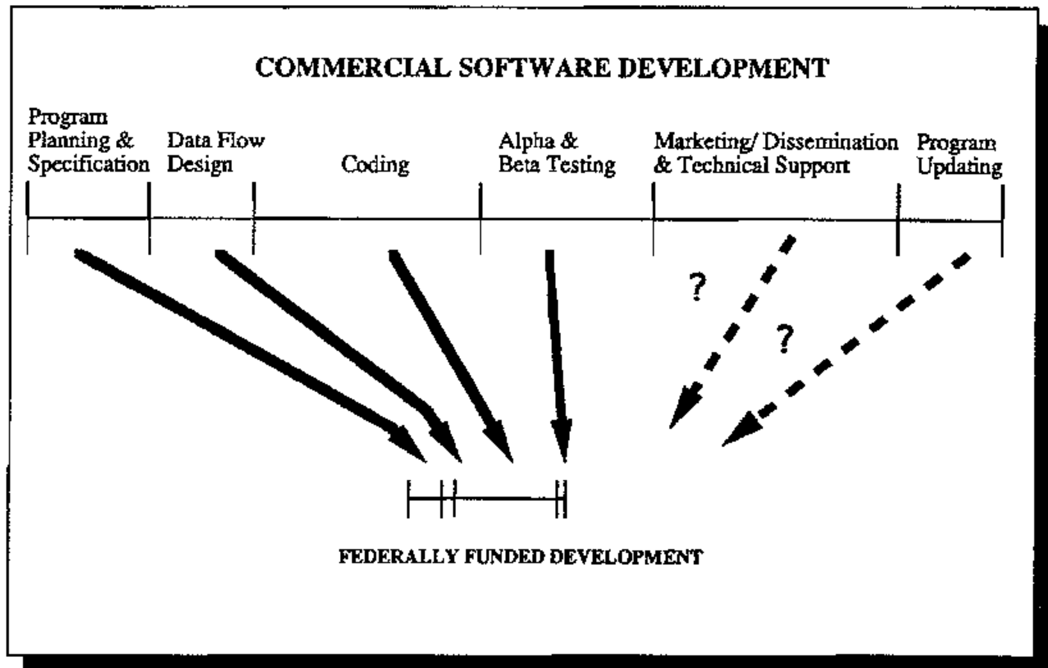

FIGURE 2

is more readily understandable to the casual or novice programmer. For example, "T:" signifies "text" and would be comparable to PRINT in BASIC or WRITELN in Pascal. With this command, one can write statements (or even leave blank lines) to the screen for a presentation. With "A:" the program will "accept" responses from the student. Such answers can be compared and judged correct by an "M:" or match command.

Authoring languages have a distinct appeal for educators and instructional designers. Freed from the minute concerns of traditional programming—particularly intensive data flow design—users presumably can concentrate more on the program planning phase of the development cycle. Segments of code can be strung together in more intuitive, linear segments.

However, a closer look at this courseware design process indicates that the programming phase of the software lifecycle remains a critical issue. What one trades in using an authoring language such as Pilot (as opposed to Pascal, for example) is power for relative ease of use. As Merrill (1982) notes, "the more powerful languages offer the advantage of additional capabilities when the author is ready to go beyond the minimal subset. If an author begins by learning *Pilot* and then desires greater power, he must scrap *Pilot* and begin by learning a new language. Why not begin with a powerful language in the first place?" (p. 76).

The simplified nature of commercial authoring languages leads to further problems. There is nothing in these systems that requires the developer to use a structured or modular approach to programming. For many courseware authors, this may encourage mediocre programming strategies (Merrill, 1982). Furthermore, authoring languages still demand a significant commitment of time for the design of the lesson and its programming.

One study (Woodward & Carnine, 1983) of special educators who were interested in courseware development (but were programming novices) indicated that they were frustrated and often confused by the format and syntax of the popular, "*Pilot*-like" authoring languages when they were directed to develop a simple drill and practice program with feedback. While some of the confusion would subside with increased use and familiarity, the person-hour investment in programming remains. Recent research (Rode & Poirot, 1989) indicates that teachers trained to use authoring languages, especially those who teach non-computer related subjects, do not continue to write software programs for their everyday instruction.

These criticisms suggest that the time educators hope to save in the coding phase of software development—at least for lengthy programs—is misleading. Problems arising in the data flow and coding steps of the project may actually lengthen the time required to achieve a beta version of the program.

Authoring systems. A logical alternative for courseware development by programming novices is the use of authoring systems. These systems allow CAI developers to create higher quality courseware in a shorter period of time than authoring languages or traditional programming languages (Graham, 1983). The user simply answers prompts or questions which are generated by the system. By selecting from a menu that appears in the beginning of the system, a user may include, for example, a graphics or sound component in a lesson. The authoring system programs and formats the input from the user. These systems allow users to create respectable courseware using a specific template.

In this sense, authoring systems solve some of the problems of courseware development where instructional tasks are highly similar (e.g., math facts, lists of spelling words, vocabulary drills). All too often traditional approaches to authoring courseware frequently lead the user to perceive the learning tasks as unique, thus missing "the opportunity to use standard formats" (Brady & Kincaid, 1981–82, p. 117).

Yet as much as authoring systems might save developers time in the program planning, data flow, and coding phases, critics are quick to note the inherent constraints of this approach to courseware development. "Users must recognize the limits of the template, and not try to force all instruction to fit the template. The use of templates would be more

acceptable if a variety of templates were provided for the different types of learning" (Merrill, 1982, p. 77). Thus, what appears to be substantial time and financial savings during the initial phases of the software lifecycle are not realized because the educator is unable to accurately "fit" his or her design into the existing system.

This issue is further complicated when developers construct authoring systems with specific instructional design principles in mind. DIAL (Engelmann et al., 1983), for example, was designed to include a variety of options for feedback, hints, timed responses, and the recycling of missed items, and the system has been used successfully in instructional studies involving mildly handicapped students (Gleason, Carnine, & Boriero, 1989; Grossen & Carnine, 1990).

However, the precise nature of such a system even further constrains alternate instructional strategies for courseware development. Critics have labeled (and dismissed) DIAL-like systems as "electronically programmed textbooks" whose pre-formatted, linear orientation offer very little in the way of sophisticated, "intelligent" instruction (Carbonell, 1970; Merrill, 1987). This characterization is unduly harsh insofar as many academic tasks are best presented through sophisticated drill and practice or tutorial programs. Rather, the criticism raises an issue of "template constraints" which is a persistent concern for a variety of authoring systems (e.g., hypermedia, expert systems).

Newer Tools for Courseware and Assessment Programs

With the growing popularity of GUI systems in industry and education, there has been a commensurate shift in the kinds of tools used to develop instructional and assessment systems in special education. As stated earlier, the GUI environment is exceedingly complex. For GUI-based special education programs to take shape at a reasonable cost and within reasonable time constraints, more sophisticated tools—ones that are already developed and commercially marketed—need to be used.

The advantages of commercial software tools and programming environments is manifold. Commercial versions are often quite robust, having been alpha and beta tested well before their use in designing educational software. Second, they frequently contain explicit links to other software programs or peripheral devices (e.g., hypermedia and its links to CD-ROM or videodiscs). Thus, commercial programs serve as powerful building blocks for complex programs. Recent interest in hypermedia and expert systems in special education bears this out.

Hypermedia. While many authoring languages and systems have been ported over to GUI systems, hypermedia has drawn the most attention. The concept of hypermedia has existed for almost half of a century (Bush, 1945) even though it has only become a viable notion recently, again due to the rapid advances in computing environments over the last two decades. Graphically oriented computers have extended the hypertext environment to visual data or "hypermaps" (Reynolds & Dansereau, 1990). Because of its recency, there are relatively few documented uses to date of hypermedia in special education. Hasselbring, Goin, and Wissick (1989) have developed hypermedia programs for "anchored" instruction. Students work in a multimedia learning lab where reading, spelling, and writing are computer-based activities, and reading words, for example, are keyed to short videodisc segments through a Hypercard™ interface.

As modern authoring languages, hypermedia-based languages are vulnerable to some of the same issues mentioned earlier. The time allocated for coding hypermedia programs easily can be underestimated. Script-oriented languages such as Hypertalk and Supertalk still require a level of competence that exceeds the novice, and the eventual program necessitates extensive design and debugging. This is particularly true when the developer wishes to access external devices such as videodisc players. The lack of built-in connections requires the use of XCMDs and XFCNs (i.e., external commands and external functions). XCMDs and XFCNs can be written in any high-level language, such as C or

Pascal, and they are "called" or invoked from within a hypermedia script (i.e.: program).

Many users who are accustomed to the style of older authoring languages and systems may be surprised to find that hypermedia programs—at least as they are currently designed—do not easily permit highly structured branching schemes which are based on student performance or other linear characteristics associated with traditional CAI programs. These programs tend to be weak in data management and summative profiles of student performance.

There is also a second, more theoretical problem with hypermedia, one associated with its program planning. The idea of a non-linear browsing environment for instruction is a radical departure from the highly structured, "CAI" character of early microcomputer authoring languages. Many researchers (e.g., Conklin, 1987; Heller, 1990) question whether or not hypermedia can be used for effective instruction given its main pedagogical assumptions (i.e., incidental learning). In this respect, hypermedia suffers the same criticism as highly structured authoring systems like DIAL (Engelmann et al., 1983)—as a template for courseware development, it constrains the range of program objectives.

This has been most apparent to users who are accustomed to the style (hence, program specifications) of older authoring languages and systems. Hypermedia programs—at least as they are currently designed—do not permit highly structured branching schemes which are based on student performance or other linear characteristics associated with traditional CAI programs. These programs are also weak in data management and summative profiles of student performance.

However, all of this is seemingly less of an issue when hypermedia is used as a direct link or "front-end" to other programs. It can be used as an interface for CD-ROM or videodisc presentations or for commercial databases. It also has considerable promise as a building block for complex assessment or instructional programs.

Expert systems. Expert system technologies have had a broad impact in business and, to a lesser extent, in special education over the last decade (Hofmeister & Ferrara, 1986). Increasingly, expert systems are chosen over structured programming languages such as Pascal and C because of the need for specific types of representational and reasoning structures.

As Figure 3 indicates, expert systems have received substantial use as development tools in special education. Once relatively obscure technologies, expert systems have evolved into a common choice for many computer based applications, especially in the area of categorical classification and assessment.

Expert systems are highly valued for their "distributed expertise." Once encoded in the software program, expert advice can be applied in a variety of settings where human experts are generally unavailable or too costly. To a large extent, this has been the guiding principle behind the expert systems developed for assessing handicapped eligibility by Hofmeister and his colleagues (Ferrara, Parry, &

Figure 3

Expert Systems in Special Education

SYSTEM	AUTHOR(S)	PURPOSE
Math Test Interpreter	Lubke, 1985	diagnosis/prescription
Class.2	Ferrara, Parry, & Lubke, 1985	classification
Behavior Consultant	Ferrara & Serna, 1985	classification
LD. Trainer	Ferrara & Prater, 1985	classification
SNAP	Haynes & Lubell, 1986	teacher training
Mandate Consultant	Parry & Hofmeister, 1986	regulation compliance
CAPER	Haynes, 1988	planning/placement
TORUS	Woodward et al., 1990	math assessment
CBM ExSys	Fuchs et al. 1991	assessment/ consultation

Lubke, 1985; Parry & Hofmeister, 1986; Prater & Althouse, 1986).

Expert systems also provide a distinct advantage in terms of the software lifecycle. Once the program is designed, the coding phase of the project (i.e., the development of the rules) is relatively brief, Thus, rather than "starting from scratch" and writing an entire program in an artificial intelligence language, such as LISP or Prolog, the developer moves very quickly from the design phase to the alpha and beta test stages. Recent improvements in graphical interfaces for expert systems further enable developers to concentrate on the design and alpha and beta testing phases of a project.

Hofmeister (1986) and others have stressed the importance of the alpha and beta test phases as means of determining agreement between human experts and the computer program. The extent to which alpha and beta testing act as a means of validating principles developed in the design phase (rather than correcting syntax errors and programming logic) underscores an important aspect of expert systems: they are easy to misuse and/or underutilize.

Expert systems, by name, are designed to capture and manipulate expertise on a given topic, and the brief history of these systems indicates a clear shift toward systems which operate on a narrowly defined knowledge base (Bowerman & Glover, 1988; Chandrasekaran & Mittal, 1984; Hayes-Roth, Waterman, & Lenat, 1983). Thus, expert systems are only as good as the knowledge (e.g., its detail, accuracy) they encode, forcing system designers to closely examine and determine if the domain is well-ordered, rich in information, and amendable to clear rules or heuristics (Alvey, Myers, & Greaves, 1985; Bramer, 1982). This also implies that expert systems are often enhanced when linked to databases.

Unfortunately, the lure of expert systems sometimes has misled developers; the technology has often eclipsed an adequate analysis of the knowledge domain. The considerations made at the design phase, particularly the match between an expert system and a given domain, are critical to its success. In this respect, the limits of expert systems are similar to the template constraints mentioned earlier. The developer may attempt to "fit" a poorly designed or conceptualized educational problem into an expert system shell.

Innovation Programs in Assessment and Technology

The two programs discussed below were developed for a recently-funded DID competition in advanced uses of technology in assessment. These programs reflect contemporary trends in software development. The first program, TORUS (Woodward, Freeman, & Howard, 1990), uses an object oriented expert system in conjunction with a database to analyze misconceptions in subtraction. The Social Skills Assessment Program (Irvin et al., 1992), the second program, is a multimedia system, employing a hypermedia front-end linked to a videodisc program.

TORUS

TORUS was designed to build upon earlier mathematical assessment programs, particularly BUGGY (Brown & Burton, 1978; Van Lehn, 1982, 1990). The BUGGY project went well beyond earlier work in simply classifying error patterns (e.g., Ashlock, 1986; Engelhardt, 1977; West, 1971). It demonstrated that even on a seemingly rudimentary task such as subtraction, students *actively constructed* interpretations of what they were taught. A cognitive model of the many deep-seated patterns of misconceptions was coded into BUGGY using the LISP programming language.

BUGGY researchers eventually documented over 3000 possible bugs (Burton, 1981), many of which are extremely unique, if not exotic. The vast array of bugs, some of which were theoretical in nature, stemmed from algorithms in the original BUGGY program which generated a "best possible fit" explanation of response patterns to computational subtraction problems. Lower level bugs were combined to explain the widest range of problems, often yielding highly idiosyncratic bugs (Brown & Burton, 1978). Of these potential bugs, 157 distinct sets of bugs have

been described, with only half of them occurring more than once (Van Lehn, 1982).

TORUS developers pursued a different strategy by closely analyzing data from over 200 students with learning disabilities to determine which bugs were empirically based and common enough to merit incorporating in the TORUS program. Almost half of the project's timeline was devoted to the program planning phase of the software lifecycle. Had the developers then attempted to create the program in a high level language such as C, it never would have been completed, much less alpha tested within the funding period.

Commercial products as "building blocks" for TORUS. To expedite the coding phase of TORUS, the developers chose commercial programs for the Macintosh. Creating movable windows, scroll bars, and pop-down menus for the Macintosh, like any other GUI, requires thousands of lines of code. However, the use of FoxBase™, a widely used commercial database, enabled the developers to quickly create a "Mac-like," menu-driven interface that was tailored to the TORUS system. Coding in FoxBase™ for this portion of the program took approximately two weeks.

Nexpert Object™ was used as the expert system engine for the program. As an object oriented system, Nexpert Object™ enabled programmers to quickly conceptualize and code rule networks. This expert system also can be linked to commercial databases such as FoxBase™ and programming languages like C. Both of these links were established so that a wide range of users could readily enter data into TORUS and the final results could be processed as database reports. Complete details of the TORUS program appear elsewhere (Woodward, 1992).

The role of alpha and beta testing. A common evaluation of expert systems is found in "real world" testing (Bowerman & Glover, 1988; Hayes-Roth et al., 1983; Hofmeister, 1986). This becomes increasingly feasible only if the coding phase of the program is reasonable and rules can be easily adapted to field-test data. Nexpert Object™, through its object orientation and robust design, facilitated field testing with more than 200 middle school students with learning disabilities (Howard & Woodward, 1990).

In the alpha phases of the program, data from many students were run through the program so that developers could refine TORUS's diagnostic scheme to better match human expert judgment. Eventually, a study of 30 students conducted late in the beta phase of development showed a .90 level of agreement between the human expert and the TORUS program. A second study involving TORUS diagnoses and an experienced special education teacher showed agreement on 21 of 22 cases ($\chi^2 = .09$; $p = .76$).

While TORUS remains a prototype of an advanced assessment system, it is currently used as a key dependent measure in several research projects. The extent to which TORUS can be implemented in day-to-day classroom settings is a complicated issue. Theoretically, it is not clear if teachers should use TORUS results to attend to each student's misconception (a.k.a. the "brush fire" approach) or teach mathematics more conceptually and use TORUS results as a broad index of understanding. On a practical level, the runtime versions of Nexpert Object™ and FoxBase™ are relatively expensive. Unless the program is rewritten in a high level language, something well-beyond the financial resources of the current developers, it is unlikely that it will be affordable to end users.

Social Skills Assessment Program

The Social Skills Assessment Program or SSAP (Irvin et al., 1992) was developed to enable teachers to rapidly diagnose specific social skills deficits in mildly handicapped students, especially those entering the mainstream. These students often have difficulties in both peer relations and teacher relations. Efficient remediation of such deficits requires a precise diagnosis. For busy teachers and overburdened school systems, it is often difficult, if not totally impractical, to conduct the behavioral assessments and observations necessary to determine the exact circumstances and factors involved with the problems associated with a specific child. The SSAP

enables teachers to make such assessments more easily and efficiently.

Three basic skill areas are assessed: peer group entry, responding to teasing and provocation, and compliance with teacher directives. The assessment program utilizes video-based stimuli to present typical school situations such as scenes of children playing or teachers looking directly at the subject (i.e. into the camera). The on-screen figures present situations and statements or requests. A student may make a response by touching the video display, which is equipped with a touch-sensitive screen. The timing and location of touches are analyzed by the computer program which then determines the next situation (i.e., video segment) that should be presented. Quick random access to any segment of assessment video is made possible through the use of a computer-controlled videodisc player.

Designed to present stimuli that are as ecologically appropriate as possible, the SSAP is unique in being a stand-alone video assessment system. Previous use of video stimuli for social skills assessment (Dodge, 1987) relied on the concurrent presence of a trained interviewer. The SSAP presents all necessary instructions for use and branches in response to the subject's input. The ability to function without the presence of an adult interviewer or supervisor was considered essential. The demand characteristics placed on a subject when reporting choices and rationales to an adult observer clearly are not the same as those experienced in naturalistic settings.

Creating a stand-alone assessment system meant it was necessary to simultaneously develop appropriate video stimuli and the computer program to control the video presentations. Because coordination of these activities was essential to meeting tight deadlines and fixed budgets, extreme care was given to the program planning and data flow design stages of development. Once initial specifications were set, extensive use was made of logic flowcharts in the design of data flow. These flowcharts were continually compared to initial design specifications and iteratively revised. Once data flow design was completed, the initial coding and video production occurred simultaneously.

Due to the unique nature of both the video stimuli and touch-screen/videodisc player hardware, it was clear that some portions of the program would require a large number of alpha and beta-testing iterations, while others would not. Furthermore, it was important that minor changes be easy to accomplish under field conditions, with limited expertise. Therefore, the researchers chose to use SuperCard (a HyperCard clone) and Macintosh computers. These environments allowed for shortening the iterative development cycles. Shortening the cycles saved time and money; using a simpler environment for simpler tasks avoided the use of an expensive professional programmer time for simple changes.

The tasks that could not be accomplished directly with Supertalk, the Supercard "scripting" (i.e., programming) language (such as controlling the videodisc player, gathering student responses from the touch sensitive screens) were written in Pascal, and invoked from the SuperCard script (as XCMDs and XFCNs). This required the expertise of a professional programmer.

The careful separation of the design and coding responsibilities was a key feature of this project. In traditional software development schemes, involving programmers in all phases of design has always been crucial. In contrast, re-designs of the assessment component (involving major rearrangements) were accomplished without re-writing high-level code (with subsequent debugging of the program). Specific start/stop points for the video stimulus segments were changed in the field by non-programmers with minimal training (e.g., elements of the graphic interface were moved to new positions on the screen). The professional programmer's time was reserved for tasks outside the Supertalk domain, or where speed of execution or complexity of data manipulation was critical and hence demanded the power that only a high level language could provide.

As with TORUS, a hybrid approach to software development, making use of commercially available "building blocks," allowed the SSAP to be designed, coded, alpha-tested and beta-tested in a short time with limited

expenditures. Expert systems, database programs, and scripting environments that allow access to high-level programs but which do not require all code to be created in the high-level language, represent the future direction for software development under conditions of severely limited time and money.

SUMMARY

Software development in special education is constrained by limited federal funding and the lack of a broad-based commercial market, a condition which is unlikely to change in the near future. Recent federal initiatives have reflected a growing realization that sophisticated software programs cannot be developed under one or two-year funding cycles. Instead, development of prototypes has become the most viable avenue for innovation.

Prototypes, by definition, do not place much emphasis on the latter phases of the software development lifecycle (i.e., marketing, dissemination, technical support; and program updating), allowing the special education technologist to devote much more time to program planning, data flow design, coding, and alpha and beta testing. One could further argue that to successfully embed instructional design or assessment principles into the prototype, an inordinate proportion of time is concentrated in the program planning and alpha and beta testing phases. With limited fiscal and/or timeline constraints, commercial software tools such as the ones described in this article are becoming the most viable means of completing federally-funded software projects. These tools function as robust building blocks which move the developer toward a realistic product for the special education population.

Yet a serious problem remains: the relationship of the prototype to an eventual commercial product for the special education market. As previously mentioned, runtime versions of commercial software programs, as well as specialized hardware (e.g., multi-media systems), are often very expensive.

This issue of cost, coupled with special education's traditionally narrow market and the lack of working relationships between most special education developers and commercial vendors, makes the impact of innovations uncertain. While similar problems arise in other areas of technology development (e.g., the National Science Foundation's funding of innovative technology projects), there is little comfort in the hope that ideas which advance the use of computers in education will somehow "trickle down" for eventual commercial use. How special education technologists should face this problem merits further discussion as the field moves toward increasingly sophisticated technology platforms.

REFERENCES

Alvey, P.L., Myers, C.D., & Greaves, M.F. (1985). An analysis of the problems of augmenting a small expert system. In M.A. Bramer (Ed.) *Research and development in expert systems.* Cambridge: Cambridge University Press.

Ashlock, R.B. (1986). *Error patterns in computation.* Columbus, OH: Charles Merrill.

Becker, H., & Sterling, C. (1987). Equity in school computer use: National data and neglected considerations. *Journal of Educational Computing Research, 3*(3), 289–311.

Bowerman, R.G., & Glover, D.E. (1988). *Putting expert systems into practice.* New York: Van Nostrand Reinhold Company.

Bramer, M.A. (1982). A survey and critical review of expert systems research. In D. Michie (Ed.) *Introductory readings in expert systems.* New York: Gordon and Breach Science Publishers.

Brown, J.S. & Van Lehn, K. (1982). Towards a generative theory of "bugs." In T.P. Carpenter, J.M. Moser, & T.A. Romberg (Eds.), *Addition and subtraction: A cognitive perspective.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Burton, R.B. (1981). DEBUGGY: Diagnosis of errors in basic mathematical skills. In D.H. Sleeman & J.S. Brown (Eds.), *Intelligent tutoring systems.* London: Academic Press.

Burton, R., & Brown, J. (1978). Diagnostic models for procedural bugs in basic mathematic skills. *Cognitive Science, 2,* 155–168.

Bush, V. (1945). As we may think. *Atlantic Monthly, 176*(1), 101–108.

Carbonell, J. (1970). AI in CAI: An artificial

intelligence approach to computer assisted instruction. *IEEE Transactions on Man-Machine Systems, MMS*, 11(4), 190–202.

Carnine, D. (1989). Teaching complex content to learning disabled students: The role of technology. *Exceptional Children, 55*(6), 524–533.

Carnine, D., Granzin, A., & Rankin, G. (1983). *Learning vocabulary.* [Computer program], Teaching Wares, Eugene, Oregon.

Carnine, D., Lang, D., & Wong, L. (1983). *Health ways.* [Computer program], Teaching Wares, Eugene, Oregon.

Chandrasekaran, B., & Mittal, S. (1984). Deep versus compiled knowledge approaches to diagnostic problem solving. In M.J. Coombs (Ed.) *Developments in expert systems.* London: Academic Press.

Collins, M., Carnine, D., & Gersten, R. (1987). Elaborated corrective feedback and the acquisition of reasoning skills: A study of computer-assisted instruction. *Exceptional Children, 54*(3), 254–262.

Conklin, J. (1987). Hypertext: An introduction and survey. *IEEE Computer, 2*(9), 17–41.

Cosden, M.A., Gerber, M.M., Semmel, D.S., Goldman, S.R., & Semmel, M.I. (1987). Microcomputer use within micro-educational environments. *Exceptional Children, 53*(5), 399–409.

Deno, S. (1990). Individual differences and individual difference: The essential difference of special education. *Journal of Special Education, 24*(2), 160–173.

Dodge, K. (1987). Social-information-processing factors in reactive and proactive aggression in children's peer groups. *Journal of Personality and Social Psychology, 53*(6), 1146–1158.

Ellis, E.S., & Sabornie, E.J. (1986). Effective instruction with microcomputers: promises, practices, and preliminary findings. *Focus on Exceptional Children, 19*(4), 1–16.

Engelhardt, J.M. (1977). Analysis of children's computational errors: A qualitative approach. *British Journal of Educational Psychology, 47*, 149–54.

Engelmann, S., & Carnine, D. (1983). *Reasoning skills I.* [Computer program], Engelmann-Becker Corporation, Eugene, Oregon.

Engelmann, S., Carnine, D., & Weiss, A. (1983) *Direct instruction authoring language.* [Computer program], Engelmann-Becker Corporation, Eugene, Oregon.

Ferrara, J.M., Parry, J.D., & Lubke, M.M. (1985). Expert systems authoring tools for the microcomputer: Two examples. *Educational Technology,* April, 1985, 39–41.

Fuchs, L.S., Allinder, R.M., Hamlett, C.L., and Fuchs, D. (1990). An analysis of spelling curricula and teachers' skills in identifying error types. *RASE, 11*(1), 42–52.

Fuchs, L.S., Deno, S.L., & Mirkin, P.K. (1982). Data-based program modification: A continuous evaluation system with computer software to facilitate implementation. *Journal of Special Education Technology, 6*(2), 50–57.

Fuchs, L.S., Fuchs, D., Hamlett, C.L., & Hasselbring, T.S. (1987). Using computers with curriculum-based monitoring: Effects on teacher efficiency and satisfaction. *Journal of Special Education Technology, 8*(4), 14–27.

Fuchs, L., Fuchs, D., Hamlett, C., & Stecker, P. (1991). Effects of curriculum-based measurement and consultation on teacher planning and student achievement in mathematics operations. *American Educational Research Journal, 28*(3), 617–641.

Germann, G. (1985). SHERI [Unpublished computer program]. Sandstone, MN.

Gleason, M., Carnine, D., & Boriero, D. (1989). Improving CAI effectiveness with attention to instructional design in teaching story problems to mildly handicapped students. *Journal of Special Education Technology, 10*(3), 129–136.

Gleason, M., Carnine, D., & Vala, N. (1991). Cumulative versus rapid introduction of new information. *Exceptional Children, 57*(4), 353–358.

Goldman, S.R., Pellegrino, J.W., & Mertz, D.L. (1988). Extended practice of basic addition facts: Strategy changes in learning disabled students. *Cognition and Instruction, 5*, 223–265.

Graham, N. (1983). *The mind tool.* St. Paul: West Publishing Company.

Grossen, B., & Carnine, D. (1990). Diagramming a logic strategy: Effects on more difficult problem types and transfer. *Learning Disability Quarterly, 13*, 168–182.

Hasselbring, T.S. (1982). Remediation spelling problems of learning-handicapped students through the use of microcomputers. *Educational Technology, 22*(2), 31–32.

Hasselbring, T.S., Goin, L.I., & Bransford, J.D. (1988). Developing math automaticity in

learning handicapped children: The role of computerized drill and practice. *Focus on Exceptional Children, 20*(6), 1–7.

Hasselbring, T.S., Goin, L.I., & Wissick, C. (1989). Making knowledge meaningful: Applications of hypermedia. *Journal of Special Education Technology, 10*(2), 61–72.

Hayes-Roth, F., Waterman, D., & Lenat, D. (1983). *Building expert systems.* Reading, MA: Addison-Wesley.

Heller, R.S. (1990). The role of hypermedia in education: A look at the research issues. *Journal of Research on Computing in Education, 22*(4), 431–441.

Hofmeister, A. (1986). Formative evaluation in the development and validation of expert systems in education. *Computational Intelligence, 2*(2), 65–67.

Hofmeister, A. (1990). Individual differences and the form and function of instruction. *Journal of Special Education, 24*(2), 150–159.

Hofmeister, A.M., & Ferrara, J.M. (1986). Expert systems and special education. *Exceptional Children, 53*(3), 235–239.

Howard, L., & Woodward, J.P. (1990). Misconceptions in subtraction: An analysis of secondary learning disabled students. (Tech. Rep. No. 90–1). Eugene, Oregon: Eugene Research Institute.

Irvin, L., Wlaker, H., Noell, J., Singer, G., Irvine, B., Marquez, K., & Britz, B. (1992). Measuring children's social skills using microcomputer-based videodisc assessment. *Behavior Modification, 16*(4), 475–503.

Johnson, G., Gersten, R., & Camine, D. (1987). Effects of instructional design variables on vocabulary acquisition of LD students: A study of computer-assisted instruction. *Journal of Learning Disabilities, 20*(4), 206–213.

Jones, K.M., Torgesen, J.K., & Sexton, M.A. (1987). Using computer guided practice to increase decoding fluency in learning disabled children: A study using the Hint and Hunt I program. *Journal of Learning Disabilities, 20*(2), 122–128.

Lally, M. (1981). Computer-assisted teaching of sight-word recognition for mentally retarded school children. *American Journal of Mental Deficiency, 85,* 383–388.

Merrill, M.D. (1987). Prescriptions for an authoring system. *Journal of Computer-Based Instruction, 14*(1), 1–10.

Nelson, T. (1967). Getting it out of our system. In G. Schecter (Ed.), *Information retrieval: A critical view.* Washington, D.C.: Thompson Books.

Parry, J.D., & Hofmeister, A.M. (1986). Development and validation of an expert system for special educators. *Learning Disability Quarterly, 9*(2), 124–132.

Prater, A. & Althouse, B. (1986). *LD Trainer* [computer program]. Logan, Utah: Utah State University.

Prater, M.A., & Ferrara, J.M. (1990). Training educators to accurately classify learning disabled students using concept instruction and expert system technology. *Journal of Special Education Technology, 10*(3), 147–156.

Reigeluth, C.M. (1979). TICCIT to the future: Advances in instructional theory for CAI. *Journal of Computer-Based Instruction, 6*(2), 40–46.

Reynolds, S. & Dansereau, D. (1990). The knowledge hypermap: An alternative to hypertext. *Computers in Education, 14*(5), 409–416.

Rieth, H., Bahr, C., Okolo, C., Polsgrove, L., & Eckert, R. (1988). An analysis of the impact of microcomputers on the secondary special education classroom ecology. *Journal of Educational Computing Research, 4*(4), 425–441.

Roblyer, M.D. (1981). Instructional design versus authoring of courseware: Some crucial differences. *AEDS Journal,* Summer 1981, 173–181.

Rode, M., & Poirot, J. (1989). Authoring systems—are they used? *Journal of Research on Computing in Education, 22*(2), 191–198.

Semmel, M.I., & Lieber, J.A. (1986). Computer applications in instruction. *Focus on Exceptional Children, 18*(9), 1–12.

Slocum, T. (1988). IS graphics: Instructional system for graphic facts. [Computer program], Seattle: Experimental Education Unit, University of Washington.

Van Lehn, K. (1982). Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. *Journal of Mathematical Behavior,* 3(2), 3–72.

Walker, H., Irvin, L., Noell, J., & Singer, G. (1992). A construct score approach to the assessment of social competence: Rationale, technological considerations, and anticipated outcomes. *Behavior Modification, 16*(4) 448–474.

West, T.A. (1971). Diagnosing pupil errors: Looking for patterns. *The Arithmetic Teacher, 18,* 467–69.

Woodward, J.P., & Carnine, D.W. (1989). The *Genisys* program: Linking content area knowledge to problem solving through technology-based instruction. *Journal of Special Education Technology, 10*(2), 99–112.

Woodward, J., & Carnine, D. (1983). *What do today's authoring languages and authoring systems mean to today's educators?* [Unpublished manuscript], University of Oregon, Eugene.

Woodward, J., Carnine, D., & Gersten, R. (1988). Teaching problem solving through computer simulations. *American Educational Research Journal, 25*(1), 72–86.

Woodward, J., Carnine, D., Steeley, D., Freeman, S., & Nospitz, C. (1988). *Genisys.* [Unpublished computer program], Eugene, Oregon.

Woodward, J., Freeman, S., Blake, G., & Howard (1990). TORUS: Computer-based analysis of subtraction. [computer program]. Eugene, Oregon: Eugene Research Institute.

Yin, R.K., & Moore, G.B. (1987). The use of advanced technologies in special education: Prospects from robotics, artificial intelligence, and computer simulation. *Journal of Learning Disabilities, 20*(1), 60–63.

Young, R.M. & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science,* 5, 152–177.