AI Magazine Volume 14 Number 4 (1993) (© AAAI)



# Quality and Knowledge in Software Engineering

*Stu Burton, Kent Swanson, and Lisa Leonard*

■ Celite Corporation and Andersen Consulting have developed an advanced approach to traditional software development called the application software factory (ASF). The approach is an integration of technology and total quality-management techniques that includes the use of an expert system to guide module design and perform module programming. The expert system component is called the knowledge-based design assistant, and its inclusion in the ASF methodology has significantly reduced module development time, training time, and module and communication errors.

The application software factory (ASF) was initiated jointly by Celite Corporation, a worldwide mining and process-manufacturing company, and Andersen Consulting. Their goal was to increase development productivity and improve software quality through reengineering of the software development process and the selective application of automation (Swanson et al. 1991). ASF was developed using total quality-management concepts that stressed improving the design and overall programming process, which would result in higher-quality finished modules.

After applying process-simplification techniques to software engineering, the tasks that remained were still complex and knowledge intensive. The simplified strategy was inhibited by the lack of experience and detailed understanding needed to take advantage of it. Teaching people how to effectively and efficiently develop systems using the new approach required a great deal of training, documentation, and support. After the first two application systems were developed using ASF, a knowledge-based system was created to capture development expertise and eliminate the majority of the training and documentation. Although first created as a tool of ASF, the knowledge-based design assistant (KBDA) quickly became a critical enabling component of the entire approach.

## Background

The ASF approach is based on the use of reusable code, the consistent application of standards, structured database design and abstraction, and detailed project coordination. It is intended to build large, integrated custom applications with enterprisewide databases. Necessary application features are bundled in reusable modules, called *shells*, that are utilized to develop the application system. Each shell encompasses all the functions and variability available for a type of processing. There are shells for scrolling data display, single-screen data display, scrolling table maintenance, single-screen table maintenance, and background process initiation. All the features provided by a shell might not work together, some are required to be used together, and each needs specific application information under different circumstances.

Analysts specify high-level module functional requirements in a one- or two-page *functional specification document*. This document includes information about functional requirements, tables accessed, access methods, elements accessed, and element behavior. All information is conveyed in high-level, functional requirement terms. Each term or set of terms implies a set of lower-level actions. When analysts did not have KBDA, they were responsible for specifying the requirements correctly and consistently using these terms.

Before KBDA was available, programmers had to interpret the functional requirements and translate them into parameters. Each parameter is derived from one or more pieces of technical information and the functional requirements. Technical information might be whether an element is a primary key on a table or whether the table is part of the SQL join. Functional requirements include where an element appears on the screen, whether it has a default value, and what function it per-

forms. The parameters specify the necessary piece or pieces of code within the shell. Specific module and database information attached to the parameters customize the code to work with other chosen pieces of code. The parameters and their respective data are applied to the shell with a set of edit subroutines in a specific order to produce the desired custom module.

All chosen pieces of shell code must be configured correctly to work with other chosen pieces of code. The features and their correspondent code are highly interrelated. For example, there are three options for querying the database with information passed in a conversation control record (CCR) between modules in a conversation. One option is to query the database only if certain data exist in CCR. This option is only available if the data passed in CCR are for a primary key on a table in the query that is being used in a certain manner. If this option is chosen, additional shell code is needed to determine if a value exists in CCR. KBDA chooses the correct syntax for the code as it varies by element type, such as alphanumeric, numeric, and date.

*The application of standards is crucial in ASF to provide a consistent look and feel to the final product.*

To correctly relay module functions without KBDA, analysts and programmers are required to have a complete understanding of shell functions and organization, shell rules and constraints, functional requirement language, architecture constraints, and project standards.

Before development of the first ASF application systems, all ASF processes were thoroughly documented, and training was conducted on the application of the shells, the programming approach, and the standards. The documentation evolved into a four-inch-thick blue binder filled with shell descriptions, standards, development procedures, and validation rules. Known as the *blue book*, it was an essential reference source for every developer on the team, but it could not enforce the standards and rules it contained.

In early 1991, ASF was used to develop a quality control (QC) application for Celite and a warranty-tracking system for Celite's parent company. These systems were robust production applications. QC was made up of 60 modules and approximately 250,000 lines of COBOL code; the warranty system contained 50 modules, totaling about 210,000 lines of COBOL code. Although the implementation of these systems using ASF showed increased productivity over traditional development techniques, several opportunities for improvement were apparent.

## Business Challenge

The difficulties encountered in using ASF in the QC and warranty-tracking systems can be categorized as follows: (1) different interpretations and misunderstanding of shell functions; (2) incomplete, invalid, and misinterpreted functional specifications; (3) disregard for shell limitations; (4) ignored or misapplied project standards; and (5) complex and confusing customization rules and procedures.

It was challenging and time consuming to communicate common shell and functional specification definitions to all the team members. Because analysts often did not realize the functional implications of their requests, they often did not recognize an error until programming was complete. Even though the shells and functional specifications were standardized, precise, and well documented, programmers complained that the analysts' compliance with the specification standards varied greatly in detail and accuracy. Programmers needed many informal sessions with the analysts to clarify specific points about the action of a module.

Analysts also had a tendency to ignore the limitations of the shells and request combinations of functions that could not be implemented. The programmers did not always recognize these requests as inappropriate and would spend time trying to accommodate the feature or would incorporate them incorrectly into the module.

The application of standards is crucial in ASF to provide a consistent look and feel to the final product. Even within a small team, however, it was difficult to communicate and control the use of screen and programming standards. Disregard for the standards and human error required the analysts to review completed modules in detail to ensure standards were followed.

Finally, the programmers were often confused about the parameters and procedures needed to customize a module. Although fully documented, the instructions for how and when to use the parameters and edit subroutines were confusing and complex when used in combination with one another. New programmers needed extensive training, and even more experienced programmers could not apply complex combinations of edit subroutines without help from the shell developers.

Despite the difficulties, the overall improvements in the development time and the quality of the final product demonstrated the benefits of using ASF. One of the key concepts of the ASF philosophy is constant improvement of the process. In this light, the

team was challenged to find a way to automate the complex, high-level decision-making tasks that were causing errors in the first implementations. They needed to create an expert analyst that understood the correct interpretation of the shells, remained within the shell constraints when specifying a module, consistently applied standards, and could correctly apply the complex parameters and edit subroutines.

## Decision to Use AI

In response to these business challenges, the team decided to implement a knowledge-based system. A knowledge-based system approach was a good fit for the following reasons: (1) the decision processes are complex, highly interrelated, and difficult to express in procedural code; (2) a wide variety of volatile data need to be considered in the decision process; (3) the ASF constraints, standards, parameter selection, and use of edit subroutines are rule based in nature; (4) a descriptive repository of shell knowledge is desirable; and (5) the application needed to be developed quickly.

A procedural code and database solution was considered and rejected because the problem was too complex and volatile. Designing a database to account for all the possible variations would be time consuming and difficult to maintain. Procedural code would quickly become far too complex. Knowledge-based system tools simplified the problem by providing a rich and flexible data-representation environment, allowing for autonomous rule declaration and providing an integrated development and deployment environment. The learning curve for a knowledge-based system was not a consideration because there were people with such skills on the ASF team.

## Development and Deployment

The knowledge-based system solution, called KBDA, was implemented during the design and installation of a third ASF application. The application being developed was a distribution system at Celite consisting of production reporting, inventory control, warehousing, customer service, order entry, traffic, shipping, and billing functions. It was estimated to contain approximately 2 million lines of COBOL source code.

Inference's expert system shell ART-IM (PC-DOS version) was chosen to develop KBDA. It best met the following selection criteria identified by the team: (1) the shell must be rule based and provide strong reasoning capabilities such as conflict-resolution strategies; (2) the shell must provide declarative representation capabilities, supporting objects as well as facts; (3) the shell must support portability between the PC-DOS and VAX-VMS environments to facilitate direct integration with the RDB table definitions in the future; (4) the shell must supply a simple graphic user interface development tool to eliminate the need to integrate with a separate graphic user interface package; and (5) the tool vendor must be flexible enough to allow the developers to experiment with the application before they make a monetary commitment. (Some members of the project team had to be convinced of a knowledge-based system's applicability and the feasibility given the short time frame.)

The Celite distribution system dictated the architecture under which ASF and KBDA were developed. Its components were Digital Equipment Corporation's VAX hardware; Digital's relational database RDB; Andersen Consulting's computer-aided software-engineering (CASE) tool set, FOUNDATION, which includes the personal computer (PC)–based design tool, DESIGN/1, and the development and run-time architecture INSTALL/1 for the VAX; and Digital's EVE editor's TPU edit subroutines.

KBDA interfaced directly with DESIGN/1 and created instructions and parameters for the edit subroutines. These instructions and parameters were ported to the VAX for application against the shells within INSTALL/1. (See figure 1).

To prove the feasibility and benefits of a knowledge-based system for this application, a pilot system for the inquiry shell was designed and implemented by one developer in two weeks. The productivity gains and applicability were immediately apparent. In QC, functional specification design for the inquiry modules was completed in approximately four hours, and programming required an additional eight hours. The same module, using KBDA, was designed in approximately 45 minutes and programmed in 4 hours.

The estimated time to incorporate the remaining shells in KBDA was about 700 hours. It was mandatory that this time be recovered in time savings from the Celite distribution system. The distribution system had over 400 application modules remaining, with completion estimates of 12 to 40 hours for each module. A savings of a least two hours for each module would provide tangible benefits. Because the majority of the modules fell within the 12- to 16-hour estimate

*... the team was challenged to find a way to automate the complex, high-level decision-making tasks that were causing errors in the first implementations.*

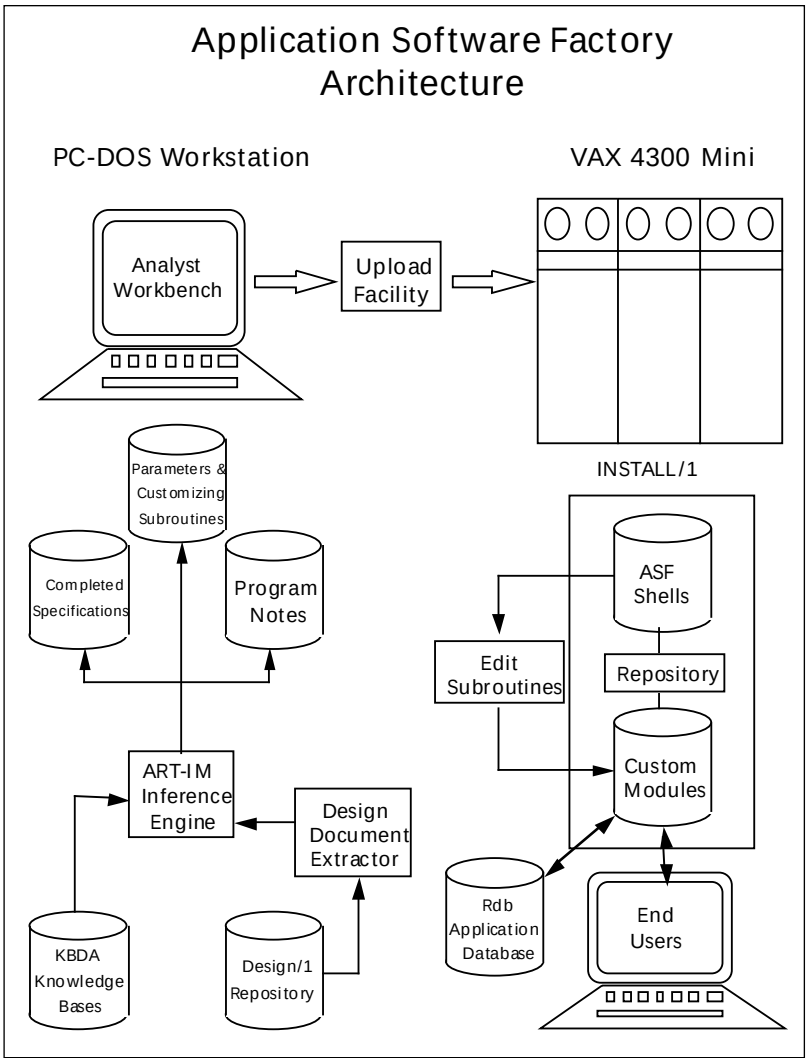


*Figure 1.* ASF *Architecture.*

range, and KBDA had actually cut the completion time of inquiry modules in half, the potential benefits well exceeded the time invested in KBDA development. Two full-time designer-developers were dedicated to KBDA creation for an elapsed development time of approximately two months.

To simplify the design, KBDA was implemented in five knowledge bases, each corresponding to one ASF shell. The primary expert was the shell developer, but application and database analysts outlined the data-entry features and functional validation they wanted KBDA to perform. The exercise proved highly beneficial to ASF because it explored new uses of the shells and validated the functions that the shells provided.

Each KBDA knowledge base was iteratively tested during development and sent through a series of test cases when it was complete. KBDA developers, the shell developer, and application analysts shared the task of validating the test cases. When everyone was satisfied with the results, the KBDA knowledge base was put into production under close observation.

The application analysts would review the output of the first few modules created and bring any discrepancies to the attention of the KBDA and shell developers. Required modifications were identified and made in both KBDA and the shells. The testing period ranged from one to three weeks for each knowledge base, depending on the shell complexity and implementation order (the knowledge bases deployed later were accepted more quickly).

Knowledge reuse between ASF shell knowledge bases was extensive. This reuse enabled subsequent knowledge bases to be developed much more quickly than the first two, even though the shells they were based on were more difficult.

KBDA has been in production at Celite since June 1991 and is instrumental in its custom system development process. During the 6-month completion of the custom distribution system, a team of 8 to 14 analysts and programmers used it daily. Even end users such as the shipping office supervisor and several customer-service coordinators used KBDA to custom design modules. Now that the distribution system is complete, a team of four analysts uses it for enhancements and new system development.

## KBDA Functions

Application analysts use KBDA once they identify the need for a module and its corresponding development shell. The analyst provides KBDA with general information about the module, such as module name, screen name, SQL access type, and data-passing action. When all module information is entered, the analyst identifies the table(s) accessed and the elements used from the tables in the module. One table must be identified as the driving table in the SQL select statement.

The behavior of every element in the module must be described. Each element is classified by one of five element types: seed, display, seed and display, seed return, and invisible. Each of these types requires different element information, has different options, and performs different functions. In a scrolling data display, called an *inquiry* (figure 2), the user enters information in the seed elements at the top of the screen. This infor-

mation qualifies an SQL query over one or more database tables. The data retrieved from the query are displayed on the lower half of the screen in display elements. The seed and display type is for elements that appear in both the seed and display areas. Seed returns display decode fields from tables not in the SQL join. Invisible elements do not appear on the screen but participate in a background function such as the SQL join.

Based on the element type, analysts further define approximately 40 other element attributes, such as whether it is part of the module selection criteria, is processed upon module entry, is passed to other modules in a conversation, or has default values and is part of the display information sort (figure 3). The attributes are limited by previously entered element information. For example, in an inquiry module, only elements that are part of the selection criteria can be processed upon their entry.

Once an element's use is defined, the interface module retrieves element and screen characteristics from DESIGN/1. This information is used to generate the COBOL PIC clause and perform validation on an element.

When the analyst is done, the module is validated to check the compatibility of options associated with the element and test its validity given the other elements in the module. One example validation would be a check to ensure that an element that is specified as part of the SQL join is joined to another element that is also specified as part of the SQL join. Other rules check screen placement, sort order, processing upon entry of multiple elements, decode fields, and primary key and foreign key relationships (figure 4). To allow the analyst to specify the elements in the order most convenient to them and allow for easy module modification, some validation that could have been performed during element entry is postponed until the analyst indicates that the specification is complete. If an error is found in validation, an error message is displayed for the analyst, and he or she is required to fix the problem (figure 5).

Once the specification is designed and validated, the analyst can save it and translate it into the edit subroutine instructions that customize the shell. KBDA has four output: (1) the saved specification; (2) DESIGN/1 specification documentation; (3) the edit subroutine customization instructions; (4) and the programmer instructions, which highlight intricate specification details and assist the programmer with screen creation.

If there are any features of the module that require programmer intervention, they are listed in the programmer instructions. The parameters and edit subroutine customization instructions are ported from PC to the VAX and applied to the shell automatically by the edit subroutine program. The output is an error-free custom COBOL module. It requires no further testing because the shell was extensively tested, and the customization was applied automatically.

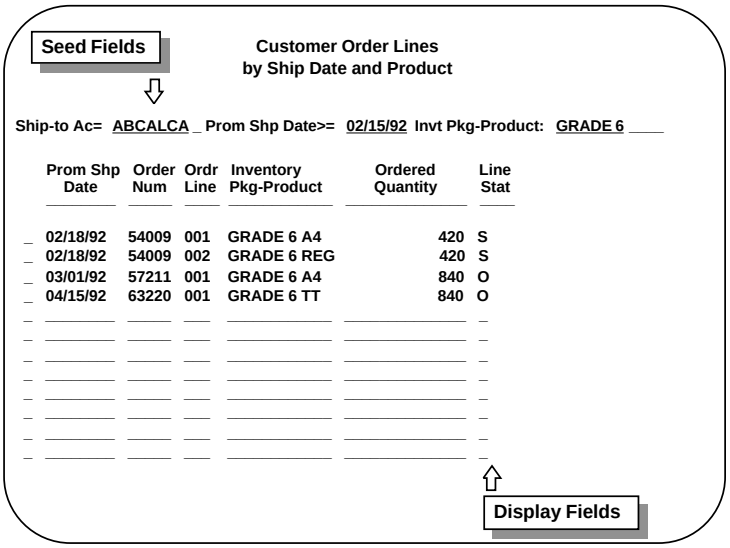


*Figure 2. Example Inquiry Application Screen.*

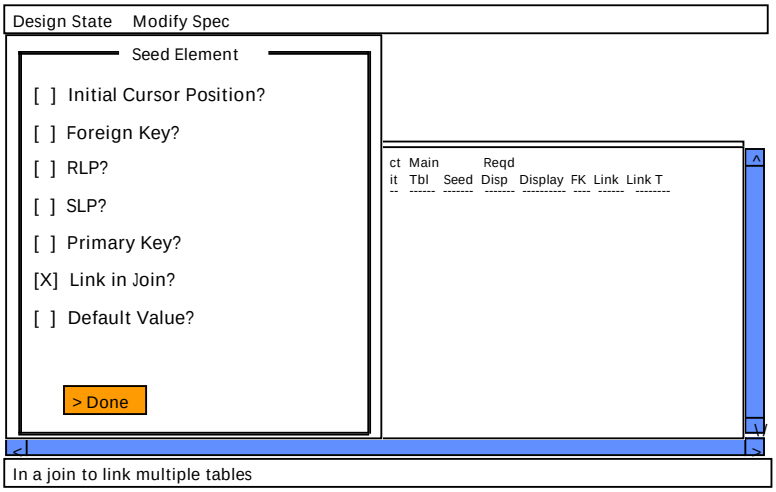


*Figure 3. KBDA Seed-Element Information Entry.*

Example Rules

Intra-element Validation Rules

A "LIKE" SQL selection criteria may not be used with a numeric field.
A sort order of ascending or descending must be specified for each element with a SQL sort order.
A primary key on the table being maintained should not be protected.
Only elements on the table being maintained should be protected.
Alphanumeric default values must be specified in quotes.
Numeric default values must be either the system timedate stamp or a number without quotes.
The initial cursor cannot be placed on a protected field.

Inter-element Validation Rules

SQL join/link elements must be identified in valid pairs.
The sort order specified for each element in the SQL Select must be unique.
Sort order for all elements in the SELECT must be sequential.
WARNING: Multiple "LIKE" SQL selection criteria degrade performance.

*Figure 4. Example KBDA Rules.*

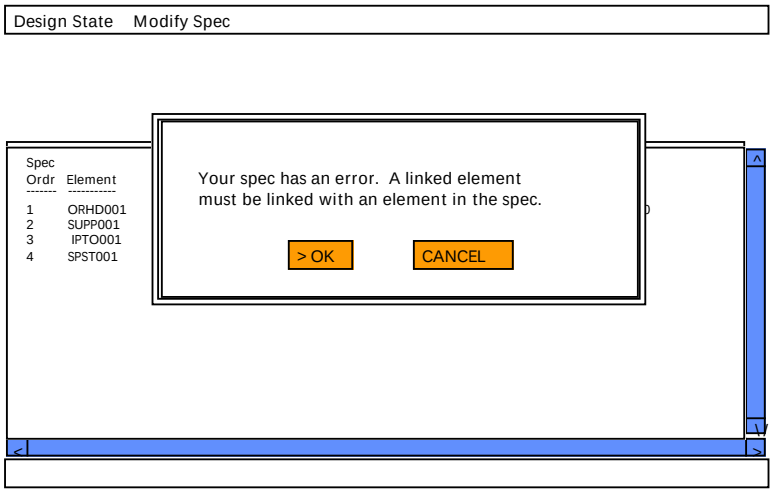


*Figure 5. KBDA Validation Error-Message Window.*

## Benefits

KBDA provides a level of benefits beyond those provided by the ASF approach alone. Its direct contributions include (1) drastic reduction in module development time, (2) significant reduction in rework caused by functional or technical changes, (3) iterative application development, (4) consistently applied standards, (5) elimination of virtually all defects (bugs) in modules, (6) elimination of the communication and interpretation tasks that were causing errors, (7) reduced analyst training requirements, (8) near elimination of the pure programmer role, (9) detailed end user involvement with module development, and (10) output of a function specification document that is used strictly for documentation.

KBDA has significantly reduced module development time beyond the productivity improvements of ASF alone (figure 6). One of the reasons for the improvement is the reduction of rework. If a module needs additional functions, the analyst simply edits the specification and regenerates the output. This procedure is automated and takes from 1 to 15 minutes, depending on the shell, greatly facilitating iterative development. The same procedure done by hand used to take one to five hours. Additionally, modules generated by KBDA are virtually error free, so costly unit testing and bug fixes are simplified or eliminated.

One of the greatest benefits of KBDA is the capture of the shell knowledge in one concise repository. Although analysts still require training in ASF methodology, database design, INSTALL/1, DESIGN/1, KBDA, and other aspects of the architecture, they no longer are required to have such a detailed, precise, and accurate understanding of the shells, their limitations, and their use. The benefits are realized up front when new analysts are brought to ASF and when changes are made to the shells, or new shells are added. It is easier to add new or more detailed knowledge to KBDA than it is to teach all the intricacies to the entire team.

Another benefit of KBDA is the change of skills mix required on a project. With ASF and KBDA, coding a module generally takes no longer than designing one. Programmer tasks include screen creation, any required program (unit) testing and user acceptance testing, and conversation creation. It is feasible to have only analyst-programmers on the team who develop their own specifications and then code them. As demonstrated at Celite, a system that traditionally requires over 50 people can be developed by a team of approximately 12 people in the same time frame. This reduction in team size further reduces system development cost by reducing the administration necessary for the project.

One unexpected benefit from KBDA arose from the ease and speed at which an analyst could develop a module. At Celite, end users from outside the data processing arena (such as the shipping office supervisor and several

customer-service coordinators) are able to use KBDA to design custom modules to be included in the system. They appear to understand and accept standards and constraints much more easily when they are restricted by a tool. The users also like that they can design what they want; if it is accepted by KBDA, they know they will get it in their system, and it will work with the other system modules. Bringing users this close to development helps the application-development team satisfy user requirements and lays the groundwork for a positive system enhancement cycle. If the users know which enhancements are simple and which are difficult, they are more likely to request—and receive—the simple enhancements.

## Maintenance and Enhancements

The knowledge base within KBDA is specific to ASF shells. It does not change unless the shells do. After the inquiry ASF shell was first implemented using KBDA, several enhancements to the shell were identified and approved. The enhancement time for KBDA was much less than the enhancement time necessary for the shell code itself. As new shells are defined by the analysts, it is expected that the expert system's iterative nature will augment the development process. However, since completion of the Celite distribution system, no additions or modifications to ASF shells or KBDA have been made.

A second implementation of ASF and KBDA is also now in production on the AS-400. The AS-400 architecture consists of IBM AS-400 hardware, an IBM AS-400 relational database, and a LANSA CASE tool and run-time architecture for the AS-400; RDML, LANSA's fourth-generation language; Andersen Consulting's PC-based CASE design tool, DESIGN/1; a custom bidirectional bridge between DESIGN/1 and LANSA; a hypertext ASF methodology documentation system; and ART*ENTERPRISE on WINDOWS for KBDA.

Building on what was learned in the first implementation, the shell and KBDA architectures have been improved. The KBDA user interface was redesigned to take advantage of WINDOWS and ART*ENTERPRISE's object approach (figure 7). It creates the module's screen and draws an example screen (following all LANSA and project standards) for the analyst before any code is generated (figure 8). Shells are now only conceptual models of how small pieces of code, called *subshells*, can be customized and placed together. Each subshell performs certain functions that are requested by the analyst for a specific module. KBDA has become a true configurator that knows how to bring objects of code together to support functional requirements. One of the benefits of the new approach is that new and modified subshells and conceptual shells can be implemented without changes to the knowledge base.

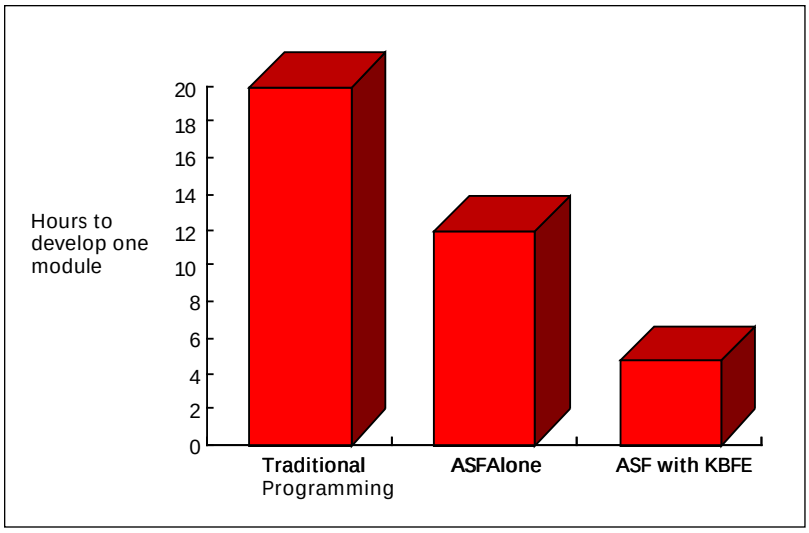


*Figure 6. KBDA Productivity Gains.*

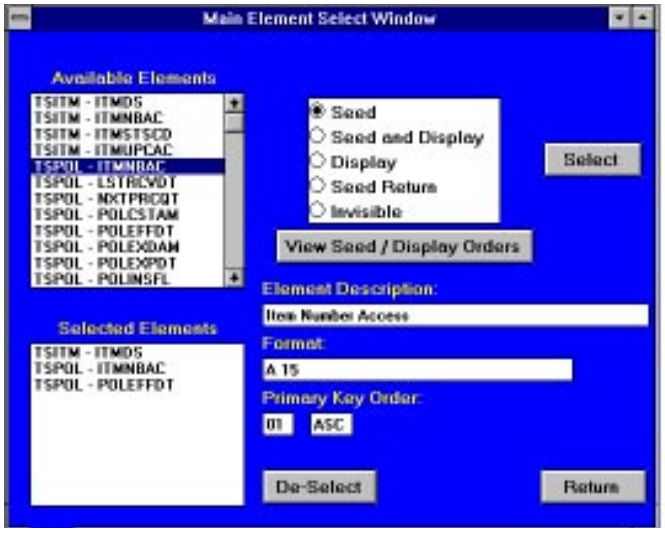


*Figure 7. KBDA for the AS-400 ASF User Interface.*


*Figure 8. Example Screen Drawn by* KBDA.

## Conclusion

The use of AI technology was a critical success factor in the implementation and maintenance of the complex application of guiding design and coding modules in ASF. KBDA understands the compound, intricate ASF shell knowledge that has proved difficult for people to assimilate. Its inclusion in the ASF process has greatly increased the value of the entire ASF methodology.

### Acknowledgments

The authors want to gratefully acknowledge the contribution of the entire team for its dedicated effort and hard work. We especially want to thank Dave McComb of First Principles, Inc., for his vision and insight; Steve McMillian of Celite for his support and encouragement; Mike Dalke of Andersen Consulting for his expertise and patience; Mark Carpenter of Celite for his technical wizardry; Derek Dalpiaz of Manville Corporation for his design of the edit subroutine applicator; and Jane Rolston of Andersen Consulting who helped code KBDA.

### References

Swanson, K.; McComb, D.; Smith, J.; and McCubbrey, D. 1991. The Application Software Factory: Applying Total Quality Techniques to Systems Development. *MIS Quarterly* 15(4): 567–579.

Stu Burton is a programmer-analyst at Celite Corporation in Lompoc, California. He received a B.S. from California State University at Los Angeles in 1986. Burton develops and maintains order-entry, manufacturing, quality-control, warehousing, and distribution applications for Celite to use in its plants and warehouses worldwide.

Kent Swanson is a partner at Andersen Consulting in Denver, Colorado. He received a B.S. from the University of Minnesota in 1967 and an M.B.A. from the University of Chicago in 1969. In the same year, he joined Arthur Andersen & Co. Since then, he has worked exclusively in Andersen's Products Industry Group. Swanson has primarily been engaged in the design and installation of systems covering the manufacturing, materials and management, distribution, logistics, and customer-support functions. He has also supervised numerous total-quality productivity projects for the firm's manufacturing and distribution clients. ASF was developed for one of these clients and accorded him the opportunity to apply the quality and productivity techniques typically associated with the production environment to software development.

Lisa Leonard is a manager at Andersen Consulting in Chicago. She received a B.S. in computer science from Texas A&M University in 1986. Since joining Andersen Consulting in 1987, Leonard has been part of the Knowledge and Technology Group, where she has gained experience implementing systems using traditional, object-oriented, and knowledge-based system technologies. She has designed and installed applications for a wide range of industries, including manufacturing, distribution, materials and management, transportation, and logistics.