Quality and Knowledge in Software Engineering

Mr. Stu Burton

Celite Corporation
2500 Miguelito Canyon Road
Lompoc, California 93436

Mr. Kent Swanson

Andersen Consulting
717-17th Street Suite 1900
Denver, Colorado 80202

Ms. Lisa Leonard

Andersen Consulting
100 S. Wacker Drive
Chicago, Illinois 60606

Abstract

Celite Corporation and Andersen Consulting have developed an advanced approach to traditional software development entitled the Application Software Factory (ASF). The approach is an integration of technology and Total Quality Management techniques which 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 (KBDA) and its inclusion in the Application Software Factory Methodology has significantly reduced module development time, training time, and module and communication errors.

Introduction

The Application Software Factory was initialed jointly by Celite, 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. The 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 simplifications 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 the ASF, a knowledge based system was created to capture development expertise and eliminate the majority of the training and documentation. While first created as a tool of the ASF, the Knowledge Based Design Assistant (KBDA) quickly became a critical enabling component of the entire approach.

Background

The ASF approach is based on utilizing 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 enterprise wide databases. Necessary application features are bundled in reusable modules, called shells, that are used to develop the application system. Each shell encompasses all the functionality 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 may 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 the KBDA, they were responsible for specifying the requirements correctly and consistently using these terms.

Before the 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 may be whether an element is a primary key on a table and whether that 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 performs. 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 exists in the CCR. This is only available if the data passed in the CCR is 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 the CCR. The KBDA chooses the correct syntax for the code as it varies by element type such as alphanumeric, numeric, and date.

In order to correctly relay module functionality without a KBDA, analysts and programmers are required to have a complete understanding of:

- Shell functionality and organization
- Shell rules and constraints
- Functional Requirement language
- Architecture constraints
- Project standards

Before development of the first ASF application systems, all ASF processes were thoroughly documented and training was conducted on application of the shells, programming approach, and standards. The documentation evolved into a four inch thick blue binder filled with shell descriptions, standards, development procedures, and validation rules. Known as the Bluebook, 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, the 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. The QC system was comprised of sixty modules and approximately 250,000 lines of COBOL code; the warranty system contained fifty modules totaling approximately 210,000 lines of COBOL code. While the implementation of these systems using the ASF showed increased productivity over traditional development techniques, several opportunities for improvement were apparent.

Business Challenge

The difficulties encountered in using the ASF in the QC and warranty tracking systems can be categorized as follows:

- Different interpretations and misunderstanding of shell functionality
- Incomplete, invalid and misinterpreted Functional Specifications
- Disregard for shell limitations
- Ignored or misapplied project standards
- 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. Since 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 very well documented, programmers complained that the analysts' compliance to 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 as inappropriate requests and would either spend time trying to accommodate the feature, or would incorporate them incorrectly in the module.

The application of standards is crucial in the ASF to provide a consistent look and feel in 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 . While fully documented, the instructions for how and when to use the parameters and edit subroutines were confusing and very 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 development time and quality of the final product demonstrated the benefits of using the ASF. One of the

key concepts of the ASF philosophy, however, 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" which 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 Artificial Intelligence

In response to these business challenges, the team decided to implement a Knowledge Based System (KBS). A KBS approach was a good fit for the following reasons:

- The decision processes are complex, highly interrelated, and difficult to express in procedural code.
- A wide variety of volatile data needs to be considered in the decision process.
- The ASF constraints, standards, parameter selection, and use of edit subroutines are rule-based in nature.
- A descriptive repository of shell knowledge is desirable.
- The application needed to be developed very quickly.

A procedural code and database solution were 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. KBS 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 KBS learning curve was not a consideration since there were people with KBS skills on the ASF team.

Development and Deployment

The KBS solution, called the Knowledge Based Design Assistant, 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 the Knowledge Based Design Assistant. 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 graphical user interface development tool to eliminate the need to integrate with a separate graphical user interface package.
5. The tool vendor must be flexible enough to allow the developers to experiment with the application before they make a monetary commitment. (Some 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 the ASF and KBDA were developed. Its components were:

- Digital Equipment Corporation's (DEC) VAX hardware
- DEC's relational database Rdb
- Andersen Consulting's CASE toolset, FOUNDATION, which includes the PC-based design tool, DESIGN/1, and the development and run-time architecture INSTALL/1 for the VAX
- DEC's EVE TPU edit subroutines

The KBDA interfaced directly with DESIGN/1 and created instructions and parameters for the edit subroutines. These were ported to the VAX for

Application Software Factory Architecture

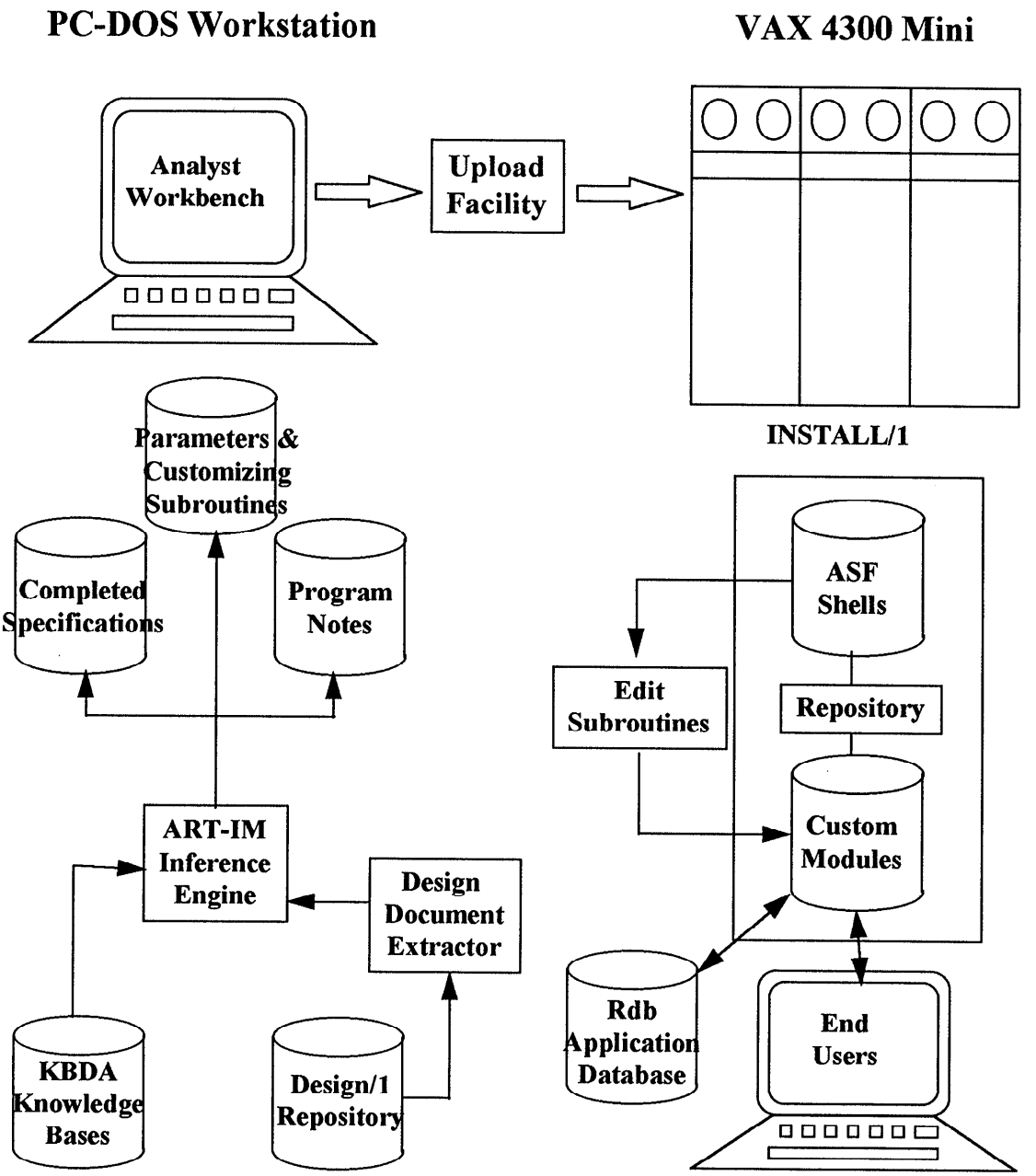

Figure 1. ASF architecture.

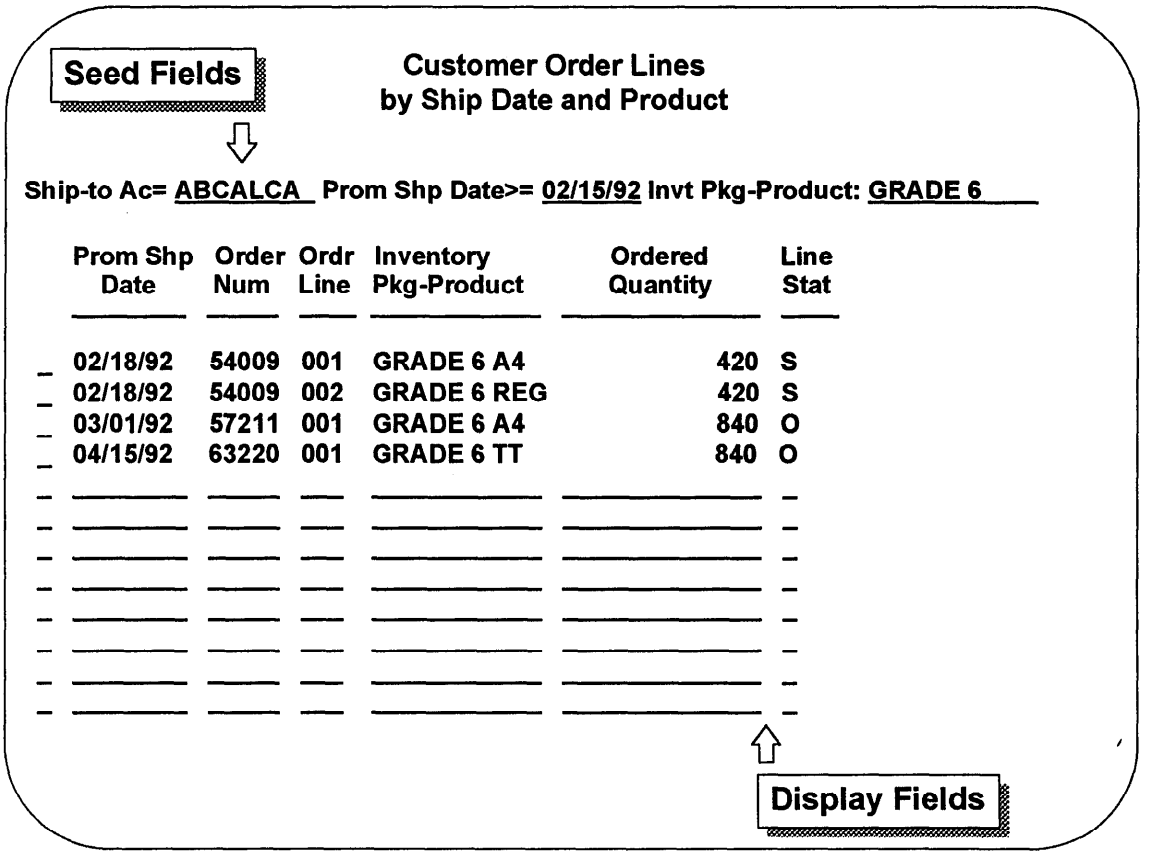

Figure 2. Example inquiry application screen.

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 the QC system, Functional Specification design for the inquiry modules was completed in approximately four hours and programming required an additional eight hours. The same module, using the KBDA, was designed in approximately forty-five minutes and programmed in four hours.

The estimated time to incorporate the remaining shells in the KBDA was approximately 700 hours. It was mandatory that this time be recovered in time savings from the elite distribution system. The distribution system had over four hundred application modules remaining, with completion estimates of twelve to forty hours for each module. A savings of a least two hours per module would provide tangible benefits. Since the majority of modules fell within the twelve to sixteen hour estimate range, and the 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, the 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 the KBDA to perform. The exercise proved highly beneficial to the ASF because it explored new uses of the shells and validated the functionality the shells provided.

Each KBDA knowledge base was iteratively tested during development and sent through a series of test cases when it was complete. The KBDA developers, 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

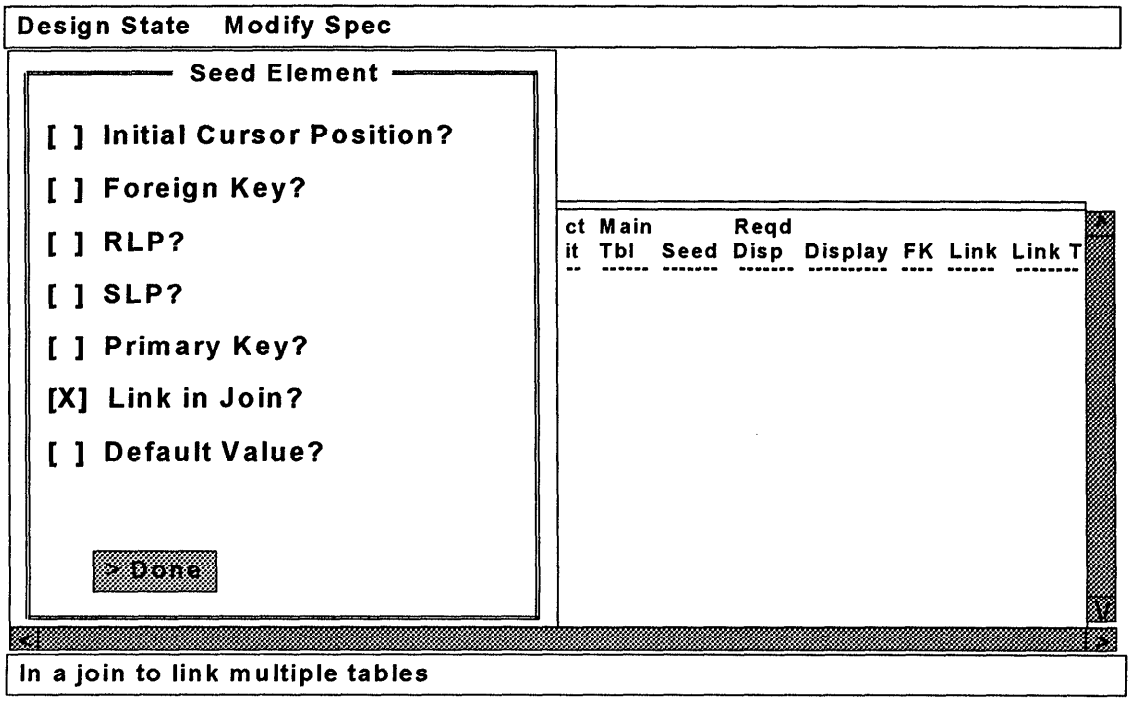

Figure 3. KBDA seed element information entry.

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 the KBDA and the shells. The testing period ranged from one to three weeks per 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 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.

The KBDA has been in production at Celite since June, 1991, and is instrumental in their custom systems development process. During the six month completion of the custom distribution system, a team of eight to fourteen analysts and programmers used it daily. Even end users such as the Shipping Office Supervisory and several Customer Service Coordinators used the KBDA to custom design modules. Now that the distribution system is complete, a team of four analysts use it for enhancements and new system development.

KBDA Functionality

Application analysts employ the KBDA once they have identified the need for a module, and its corresponding development shell. The analyst provides the KBDA with general information about the module such as module name, screen name, SQL access type, and data passing action. When all module information has been entered, the analyst identifies the table(s) accessed and the elements used from those 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 require different element information, have different options, and perform different functions. In a scrolling data display, called an inquiry (see Figure 2), the user enters information in the *seed* elements at the top of the screen. This qualifies an SQL query over one or more database tables. The data retrieved from the query is 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

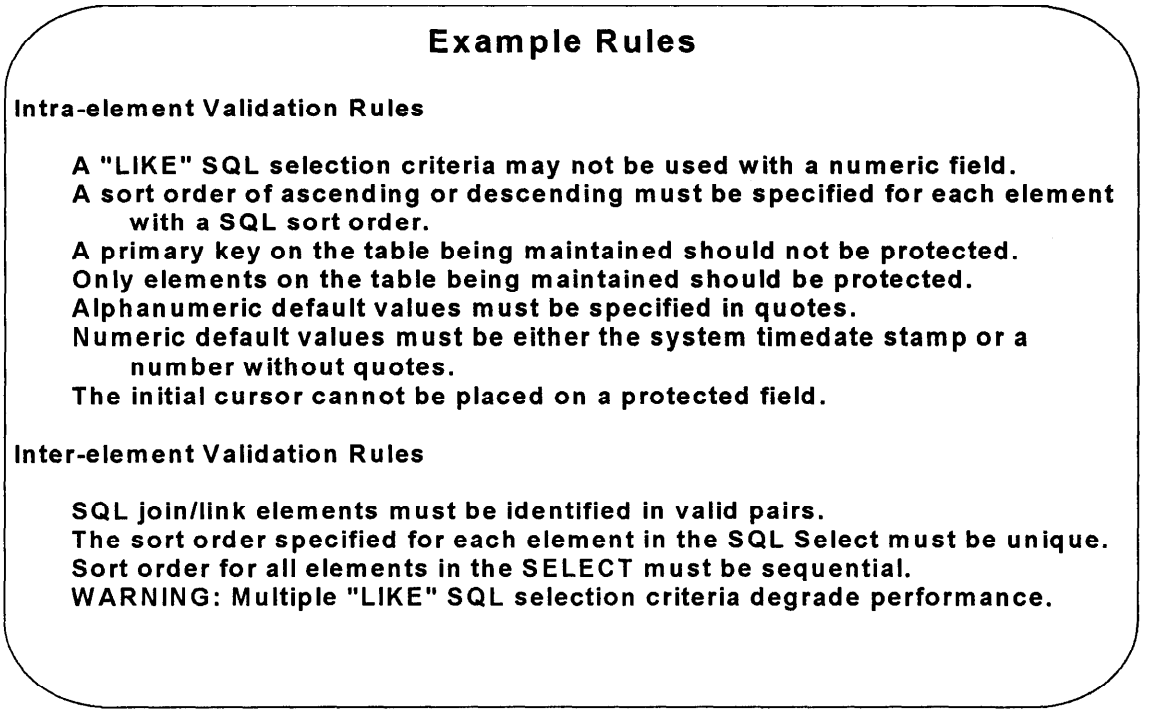

Figure 4. Example KBDA rules.

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 forty other element attributes such as whether it is part of the module selection criteria, processed upon module entry, passed to other modules in a conversation, have default values and are part of the display information sort (see 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 has been 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 insure that an element that is specified as part of the SQL join must be 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 (See Figure 4). In order 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 they are required to fix the problem (See Figure 5).

Once the specification is designed and validated, the analyst can save it and translate it into the edit subroutine instructions that will customize the shell. The KBDA has four outputs:

- The saved specification
- The DESIGN/1 specification documentation
- The edit subroutine customization instructions
- 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

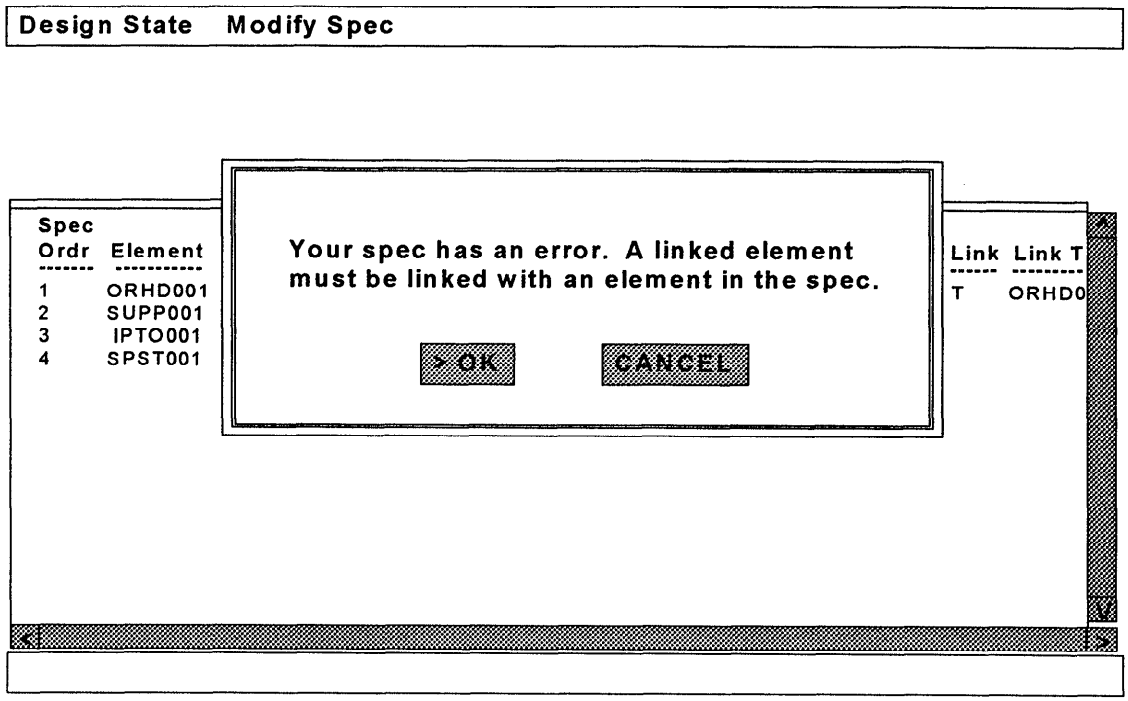

Figure 5. KBDA validation error message window.

programmer instructions. The parameters and edit subroutine customization instructions are ported from the 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.

Benefits

The KBDA provides a level of benefits above and beyond those provided by the ASF approach alone. Its direct contributions include:

- Drastic reduction in module development time
- Significant reduction in rework caused by functional or technical changes
- Iterative application development
- Consistently applied standards
- Elimination of virtually all defects (bugs) in modules
- Elimination of the communication and interpretation tasks that were causing errors
- Reduced analyst training requirements
- Near elimination of the pure programmer role
- Detailed end-user involvement with module development
- Output of a Function Specification document that is used strictly for documentation

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

One of the greatest benefits of the KBDA is the capture of the shell knowledge in one concise repository. While analysts still require training in ASF Methodology, database design, INSTALL/1, DESIGN/1, the KBDA

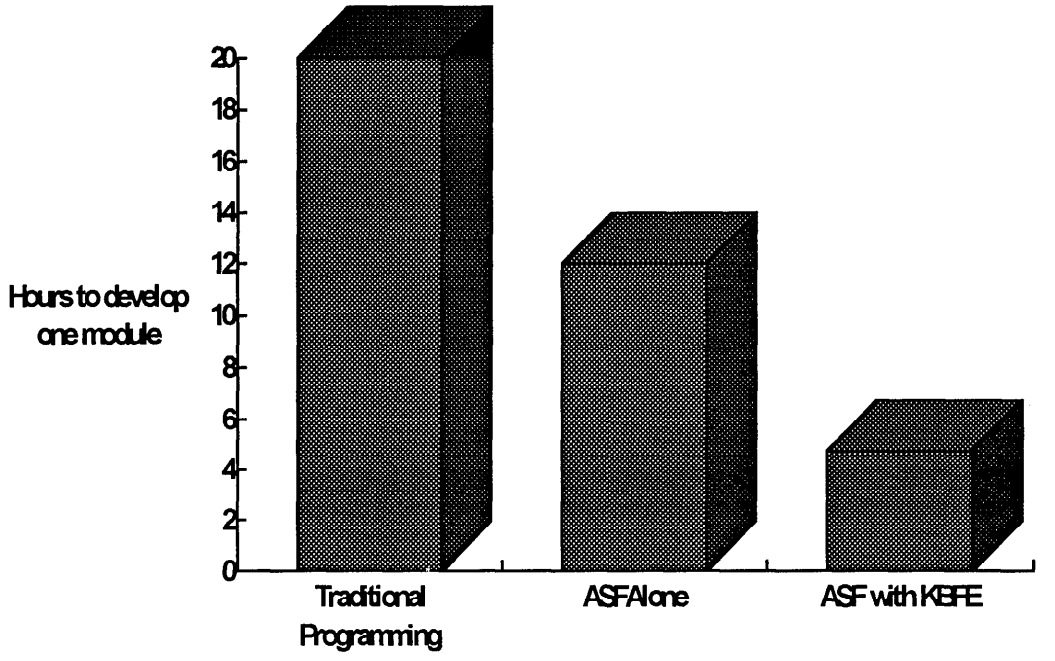

Figure 6. KBDA productivity gains.

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 the 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 the KBDA than it is to teach all the intricacies to the entire team.

Another benefit of the KBDA is the change of skills mix required on a project. With the ASF and KBDA, coding a module generally takes no longer than designing one. Programmer tasks are comprised of screen creation, testing and conversation creation. It is quite feasible to have only analyst/programmers on the team who develop their own specs and then "code" them. As demonstrated at Celite, a system that traditionally would require over 50 people can be developed by a team of approximately 12 people in the same time frame. This further reduces system development cost by reducing the administration necessary for the project.

One unexpected benefit from the 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 the 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 the fact they can design what they want, and if it is accepted by the KBDA, they *know* they will get it in their system and it will work with the other modules in the system. 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 the KBDA is specific to the ASF shells. It will not change unless the shells do. After the inquiry ASF shell was first implemented using the KBDA, several enhancements to the shell were identified and approved. The enhancement time for the KBDA was much less than the enhancement time necessary for the

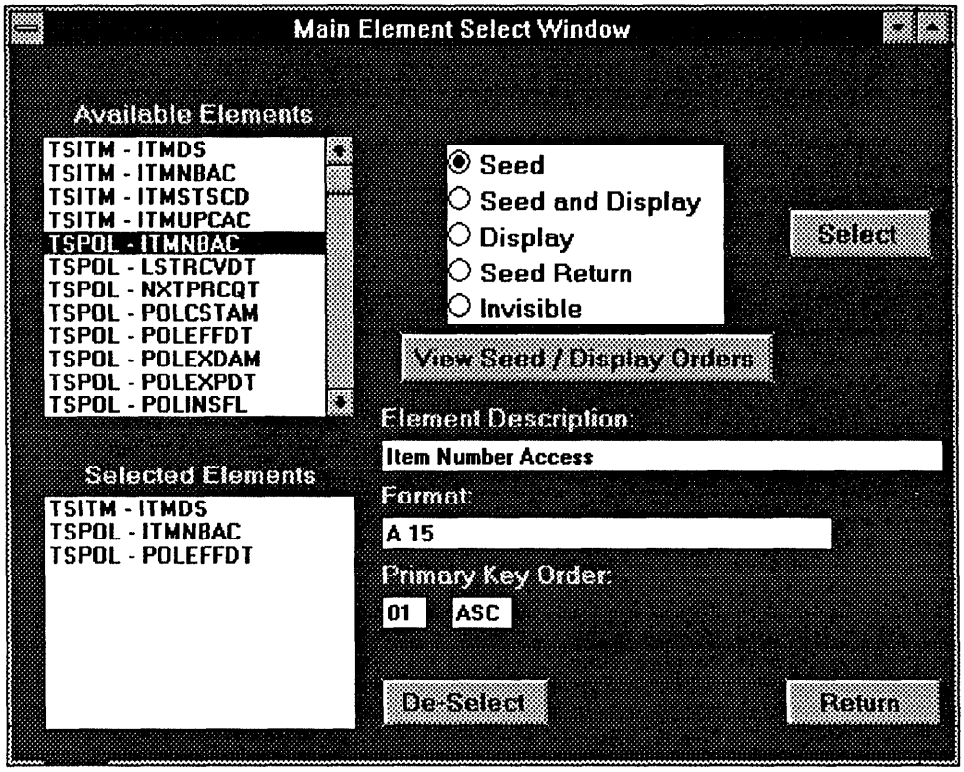

Figure 7. KBDA for the AS/400 ASF user interface

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 the ASF shells or the KBDA have been made.

A second implementation of the ASF and KBDA is also now in production on the AS/400. The AS/400 version architecture consists of:

- IBM AS/400 hardware
- IBM AS/400 relational database
- LANSA CASE tool and run-time architecture for the AS/400
- RDML, LANSA's 4GL
- Andersen Consulting's PC-based CASE design tool, DESIGN/1
- Custom bi-directional bridge between DESIGN/1 and LANSA
- Hypertext ASF Methodology Documentation System
- ART*Enterprise on Windows for the KBDA

Building on what was learned in the first implementation, the shell and KBDA architectures have been improved. The KBDA user interface has been redesigned to take advantage of Windows and ART* Enterprise's object approach (see Figure 7). It creates the module's screen and will draw an example screen (following all LANSA and project standards) for the analyst before any code is generated (see 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 requested by the analyst for a specific module. The KBDA has become a true configurator that knows how to bring objects of code together to support functional requirements. One of the benefits from the new approach is that new and modified subshells and conceptual shells can be implemented without changes to the knowledge base.

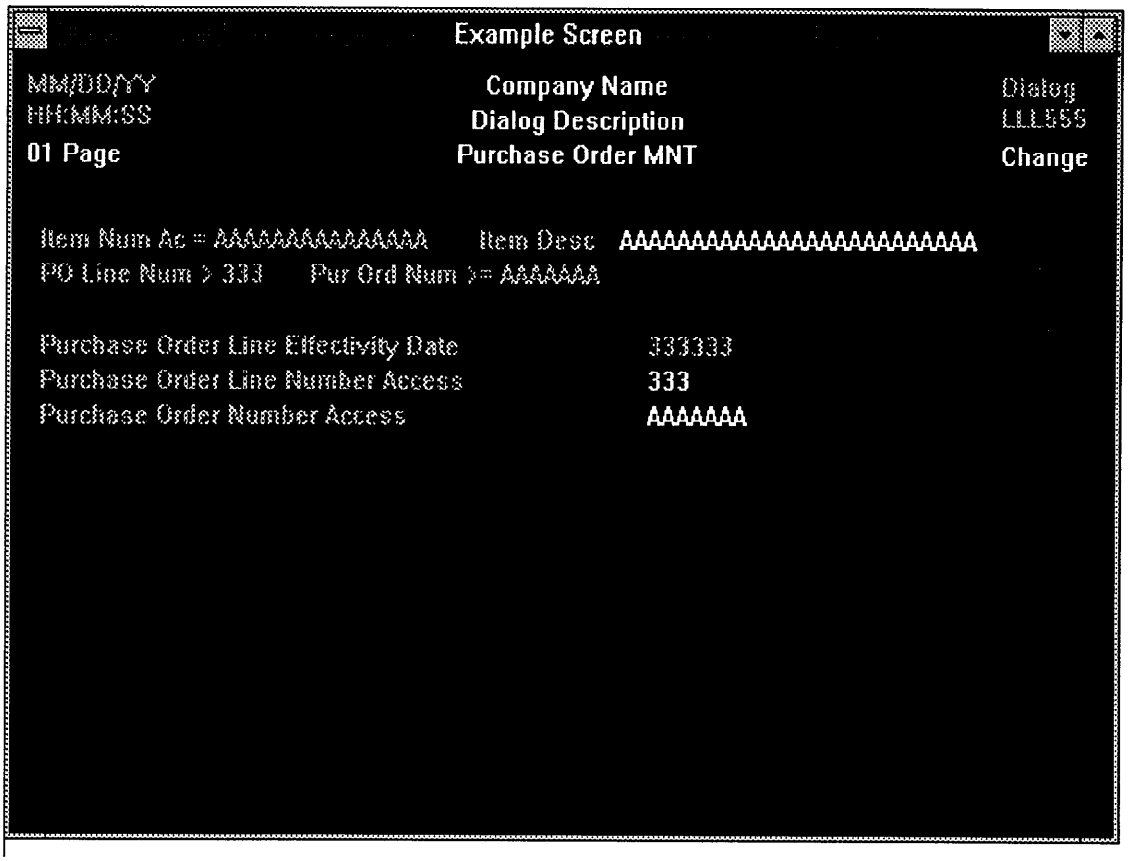

Figure 8. Example screen drawn by KBDA.

Conclusion

The use of Artificial Intelligence technology was a critical success factor in the implementation and maintenance of the complex application of guiding design and coding modules in the Application Software Factory. The KBDA understands the compound, intricate Application Software Factory 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 wish to gratefully acknowledge the contribution of the entire team for their 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 designing the edit subroutine applicator, and Jane Rolston of Andersen Consulting who helped code the Knowledge Based Design Assistant.

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.