Table Driven Design

A DEVELOPMENT STRATEGY
FOR MINIMAL MAINTENANCE
INFORMATION SYSTEMS

By Wayne Cunneyworth

Data Kinetics

Data Kinetics Ltd.
2460 Lancaster Road
Ottawa ON K1B 4S5

Tel 613-523-5500
Fax613-523-5533

Table of Contents

Acknowledgment

Many people deserve credit for bringing together various pieces of the software design puzzle described within these covers. I am indebted to Sheldon J. Glasser for his pioneering efforts in the field of table driven programming. As author of the table driven summarization algorithm, for example, he reminds us that faster, more powerful hardware and software can be used in new ways, and not simply to do the same old cumbersome things more quickly. Darlene Galipo at Northwestern Mutual Life provided early support in developing educational materials on the subject of minimal maintenance program design. Lorne Doolan and Watson Seto at Revenue Canada Taxation have my gratitude for their input and review of preliminary drafts. Special thanks go to Chris Anstead for his inspiring insight into the practical use of sparse decision tables. And finally, a very sincere thanks to all the adventurous people at Data Kinetics who thrive on change and make our business such a pleasure.

Preface

This document is directed at anyone who is concerned with the rising cost of maintaining today's information systems. This includes all personnel responsible for applications software development and ongoing maintenance or enhancements, at strategic and tactical levels.

- Information Systems Senior Managers
- Project Managers
- Systems Analysts
- Application Programmers
- End Users

Some sections are directed primarily at the more technical audience groups. These sections are designed to be browsed or skipped entirely by the non-technical reader, with minimal effect on the higher level flow of the document.

The objective here is threefold. First, to present general principles and benefits of Table Driven Design within the greater context of corporate Information Technology (IT) policies and directions. Second, to identify the impact and significance of Table Driven Design with respect to administrative groups, administrative functions, traditional application design approaches, standards, data types and processes. Finally, to illustrate certain key principles with reference to particular sample problems and applications, and identify opportunities for Table Driven Design.

The first sections of this document describe general classes of recognized systems delivery problems, and general development directions for addressing those problems. These issues are placed in the context of a typical corporate environment, including contemporary business directions and architectural principles.

Table Driven Design is introduced as a powerful approach to addressing systems delivery problems. Several classes of tables are described, along with their roles in the application life cycle, followed by a discussion of decision control structures and their impact on systems delivery. This leads to a more detailed technical description of decision tables.

Subsequent sections document the implications of Table Driven Design for specific groups, sample projects and platforms. Benefits of the approach are discussed for

selected real world applications. Several topics which relate to the administration of tables are also presented.

Coding examples illustrate key concepts of Table Driven Design in the COBOL II programming language. For better or worse, COBOL remains the language most widely used in business systems today. As a teaching aid for systems design topics, the natural language character of COBOL statements is more likely to be understood by a majority of information workers, regardless of programming background. It should, however, be stressed that the principles of Table Driven Design are independent of any particular programming language or processing platform.

1
Development Challenges and New Directions

Traditional Procedural Design

Early computer application systems were essentially operating procedures manuals translated into programming languages. This traditional procedural design emphasizes sets of instructions executed in a particular sequence to resolve a problem under all possible prevailing conditions (see Figure 1). Certainly, computers have no difficulty executing programs written in a procedural style but business problems are complex. They are defined and resolved only by much careful thought and attention to detail. Unlikely conditions may be missed due to incomplete analysis or they may be ignored or handled inappropriately in order to minimize program complexity. Impossible paths may be coded inadvertently.

Deadlines approach quickly while programmer/analysts struggle with flowcharts and other logic aids designed to help account for all possible operating situations. It has become clear that much of the success of older, manual procedures depended on the common sense of those who used them. Automated systems do not have this luxury. Any special or unusual circumstances, not accounted for in a hard coded procedure, can bring automated systems, and the business itself, to a grinding halt. An adaptable business model is critical to a successful business.

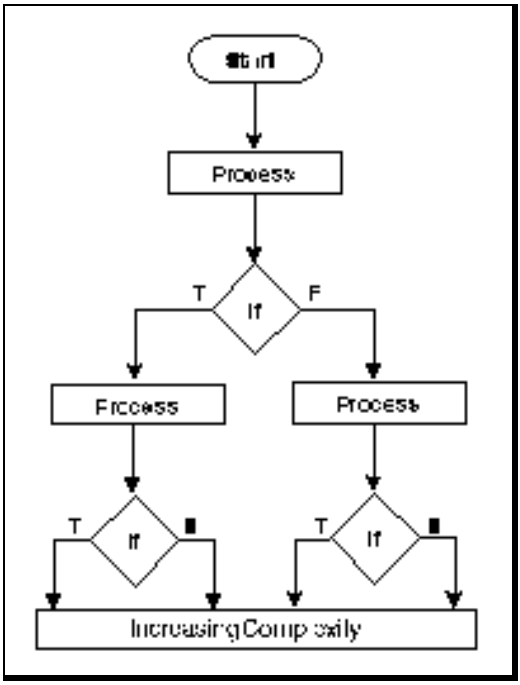


Figure 1 *The Procedural Approach*

Traditional procedural programming imbeds the details of program control flow decisions within the program. For example, a business rule implemented with the COBOL statement:

```
IF  TAXPAYER-TYPE = '01' OR
    TAXPAYER-TYPE = '02'
    PERFORM ACTION-X
```

refers to the detail values '01' and '02', which determine if ACTION-X is to be performed.

Control flow logic must be understood before changes can be made to any specific detail values, and all relevant occurrences of those values must be located in the program. Traditional systems design is often simply not flexible enough to meet the demands of a rapidly changing business world. Since the program code does not follow a normalized structure, the search for relevant detail rules during program enhancement can be time consuming and tedious. According to industry statistics, half of a typical maintenance effort is spent just to determine what has to be changed.

The Legacy

Information systems now in production at major corporations are the products of more than 20 years of development, maintenance, and maintenance upon maintenance. The legacy is a patchwork quilt of duplication, inconsistency and inefficiencies, characterized by:

- **Redundant software**
  Duplication of code
  Inconsistency of implementation
  Inflexible design

- **Redundant effort**
  Duplication of programming effort
  Additional analysis for inconsistencies
  Repeated exposure to inflexible design

To make matters even worse, the very act of solving a problem usually uncovers a new wave of problems to solve. In an interview with Database Programming & Design magazine, Tom Nies, president of Cincom Systems Inc., draws upon a nuclear metaphor:

"In a nuclear reaction, you have a chain reaction of atoms being bombarded by neutrons, [releasing] ... a tremendous amount of energy. ... When you're trying to implement information systems, it's similar: you can never put a problem to rest. Each solution expands demand. Your end users are expanding demand much faster than the data processing community can respond. The demand is expanding exponentially, while IS is trying to expand linearly."

The situation has reached a point where responsiveness to change and opportunities for new development are severely constrained by the demands of inflexible systems and ongoing maintenance. Many organizations estimate that 70 - 90 percent of their Management Information Systems (MIS) budget is spent maintaining existing applications. Recent industry surveys place this figure closer to 50 percent. Either way, legacy systems must be re-engineered if tomorrow's business objectives are to be met.

Many of today's business activities are not simply supported by information systems; business cannot proceed without them. Barriers to systems delivery are barriers to conducting business.

In recent years, the fact that any complex application has been delivered on time, in spite of existing barriers to systems enhancement, is much more often a tribute to the competence and dedication of MIS personnel than it is a reflection of effective development procedures, and with each succeeding year, the challenge becomes greater. The complexity of systems is growing at an exponential rate and future success cannot depend on the status quo.

Development Directions

Effective solutions to the barriers described above require advances on multiple fronts. Efforts must be directed to reduce, reuse and recycle existing software to minimize system complexity. Goals include eliminating inconsistency and duplication, re-engineering for flexibility and empowering the end user.

Inconsistency may be eliminated through strict adherence to naming standards, coding conventions and other systems design directives. A business rule is easier to change if it is implemented in a consistent manner wherever it is invoked in the software model.

Duplication may be eliminated through identification of common sets of conditions, functions and parameters within an application or across systems. No two processes can share a single copy of common code until their common elements have been recognized. Software analysis tools and statistics are now available to assist in building this inventory. A business rule is easier to change if it is implemented in a single location in the software model.

Re-engineering for flexibility may be accomplished through externalized business rules which are not compiled with a program, but rather, interpreted only when they are required at execution time. A business rule is easier to change if the rule is external to the application program, so that the change can be localized in the rule, with no impact on the entire software model.

Finally, empower the end user. If end user demand is expanding faster than the data processing community can respond, then in the spirit of democracy, the end user should be recruited to contribute as much to the solution as possible.

There is much that can be done in support of these goals using existing software and techniques already available in most organizations. Recent advances in hardware and software now provide for dramatic new capabilities which, unfortunately, are often poorly understood and thus under-utilized in most shops.

Larger addressable memories, in particular, should not be viewed as simply more of the same old familiar technology. As we shall see, an entire set of data residing in memory is not just "more data in a larger buffer". In this case, the whole is greater than the sum of its parts. On the subject of extended addressability, IBM publication GC28-1854 states:

"Extended addressability can open completely different solutions to programming problems, ... [solutions which] both improve performance and reduce the effort required for program development."

New technologies often require a re-orientation in thinking. The concepts of Table Driven Design will allow MIS personnel to begin developing design skills which

capitalize on such new technologies immediately, and build upon them gradually to maximize benefits in the future. These skills may be used to re-engineer existing systems or to design completely new systems.

Re-engineering Legacy Systems

The terms "legacy systems" and "heritage systems" have been applied to software applications and design principles that have been inherited as the legacy of an earlier era. They do not necessarily refer exclusively to older systems, but could also indicate new systems which have been designed and developed under that same legacy mindset. Systems which are not in this category, then, are those which have been designed and developed using more contemporary paradigms, including relational databases, object orientation, main memory tables and distributed processing architectures.

Legacy systems often suffer from poor performance and/or expensive maintenance requirements. To reduce costs and promote responsiveness to business changes, they must be re-engineered.

Re-engineering for High Performance

A daily database update process may perform its automated task flawlessly, but if it takes 25 hours to execute, it is a useless exercise. This situation is a real concern to large organizations with rapidly increasing volumes of transactions. Application managers view the approach of this processing wall with understandable trepidation. In many cases, appropriate use of memory resident tables has dropped the elapsed time of an otherwise well tuned process dramatically, for example, from 12 hours to 40 minutes, or from 3 hours to 5 minutes.

Table oriented re-engineering for improved operational efficiency is often a straightforward task, as in, the replacement of tabular data in an external file with corresponding main memory tables of identical design. The purpose here is to minimize I/O by buffering the entire table in memory and, at the same time, reducing the instruction path for accesses dramatically. The same logic extends to replacement of tables for any file organization or Database Management System (DBMS), assuming the data meets the requirements for semi-stable or transient data. (Refer to Chapter 3 - General Table Classifications for details).

This approach does not have functional requirements beyond those already available with standard file or DBMS processing, and thus does not require any change in mindset for the legacy oriented application programmer. It merely requires, for the most part, a simple one-to-one replacement of file or DBMS accesses with the equivalent table load and access logic.

Re-engineering for improved performance is often a critical issue for MIS organizations, but the primary motivation for re-engineering usually focuses on flexible application logic and reduced maintenance. High performance is nonetheless a prerequisite for flexible, Table Driven Design.

Re-engineering for Minimal Maintenance

Maintenance of legacy systems is arguably the major challenge facing MIS shops today. In the wake of today's intense efforts to re-engineer business processes at the highest levels, applications which model those processes must mirror changes smoothly, quickly and accurately. Business Process Re-engineering (BPR) is intended to improve the efficiency of the business by restructuring to minimize required resources and time delays. It must be matched by parallel efforts in legacy application re-engineering.

Re-engineering archaic systems for any reason is no easy task, and re-engineering for Table Driven Design is no exception. The net effect on future maintenance of those systems should indicate a considerable reduction in effort. The problem here is to restructure the existing program code to minimize the impact of anticipated classes of future maintenance. The existing logic must be well understood before it can be manually disassembled and recycled into a new module. A number of vendor products are available to contribute to the automation of this process. The result often aims for a traditionally structured, procedural architecture, not a table driven architecture. General patterns of business rules emerge more readily in structured code than unstructured versions, assisting in the parameterization process, a necessary preliminary to table driven re-design. Rules must be isolated, clarified, stripped of implementation dependencies and prepared for re-use. Software is available in the market place which can help to analyze unstructured logic, provide design related statistics, and identify problematic areas of program code which would benefit from table driven approaches.

More recently, some products have focused on the automated extraction or "capture" of business rules in legacy applications. Business rules are defined, recognized, extracted and classified for purposes of reverse and forward engineering. Rules may be dispersed across a wide variety of applications. Classification consists of gathering similarly structured rules together and loading them into a single, shared table. Rules are associated with data entities or objects which are impacted. Classification extends across hierarchical levels, organizing rules into hierarchies of tables. Effective use of rule extraction products requires that definitions for business rules be communicated to an enterprise wide audience.

Piecewise Versus Holistic Improvements

In Volume 1 of "The Open Society and its Enemies", Sir Karl R. Popper compares two approaches to social engineering, which he calls "piecemeal" and "utopian":

"The piecemeal engineer will ... adopt the method of searching for, and fighting against, the greatest and most urgent evils of society, rather than searching for, and fighting for, its greatest ultimate good. This difference is ... most important. It is the difference between a reasonable method of improving the lot of man, and a method which ... may easily lead to an intolerable increase in human suffering. It is the difference between a method which can be applied at any moment, and a method whose advocacy may easily become a means of continually postponing action until a later date, when conditions are more favourable."

There are good reasons to extend this argument from the social sciences into the more physical engineering disciplines. In particular, it is relevant to software engineering.

For the information systems designer, the often conflicting objectives of "doing it right" and "getting it done" present a spectrum of possibilities for maintaining and enhancing legacy systems, from ad hoc improvements at the detail level to complete rewrites. There are times when a holistic appraisal demands a complete rewrite, but under other circumstances, a piecewise approach is more than satisfactory.

In yet another manifestation of the 80/20 rule, 80% of the benefits of table driven design may sometimes be derived from 20% of a particular application's rule base. Some rules can be expected to change more frequently than others. Major improvements in maintainability may be accomplished by generalizing some simple, yet volatile classes of rules in the program code and placing the detail control values in tables. The remaining, historically more stable sections of code, can be left alone. This approach is not intended to make sweeping changes to the overall design of the application, but it can substantially reduce the pain caused by specific maintenance problems.

The analysis of such legacy systems for piecewise improvements generally proceeds from the bottom up, and the code is converted across a lower level before the analysis continues later at a higher level. This strategy, if carried through to higher levels of the application over a period of time, may result in some redundant effort and obsolete tables as higher level patterns and generalizations are discovered and formalized. The assumption here is that the short term benefits of redesigning problem areas will be preferable to maintaining the status quo. It may well be that such improvements can prolong the life of a system component to the point where a higher level analysis may never even be necessary, or the system is otherwise re-engineered completely from the top down.

Corporate Environment and Directions

Any solutions considered for identified information technology problems must be consistent with established corporate strategic and tactical objectives. They must also be compatible with accepted architectural directives, development methodologies and software environments. An overview of relevant issues follows.

Business Directions

Business rules and procedures are frequently affected by outside influences. As a result, Information Technology (IT) policies and directions must be regularly reviewed and IT plans must be formulated and revised to ensure that information systems continue to support corporate objectives and commitments. Information systems, in turn, must be capable of continual evolution to accommodate change quickly, effectively and economically. Timeliness, survivability, integrity and security are critical success factors in achieving the corporate mandate. Visions of service excellence include the following measurable themes.

- **Responsiveness**
  Improving the accessibility, quality and timeliness of client services.

- **Simplification**
  Streamlining the administrative process.

- **Integrity**
  Ensuring the correctness of system operation and data.

- **Productivity**
  Enabling employees to provide the highest quality service at minimum cost.

A number of environmental forces, or external factors, influence the way a company does business and the manner in which it models that business, including:

- **Changing government legislation**
  Tax reform
  Social program reform

- **Changing business relationships**
  Corporate reorganizations
  Partnership agreements
  Supplier capabilities
  Customer requirements

- **Global trends**
  Changing demographics
  Exchange rates
  Interest rates
  Insurance policies
  Fashions
  Competitive indicators

- **Emerging Technologies**
  Opportunities afforded by rapid changes in technology

Strategic objectives provide impetus for change, and direction for tactical and operational decisions.

Architectural Principles

Contemporary MIS architectural principles include customer focus, employee focus, quality, timeliness, security, flexibility, reusability, consistent methodologies, measurement information and buy vs make.

A customer focus implies the involvement of clients in defining their needs and developing solutions. The client community increasingly shares the responsibility of establishing priorities and plans which allow for the delivery of quality products and services. Either indirectly through market surveys and competitive pressures, or directly through consumer and user groups, customers are becoming more involved in the definition and acceptance of new architectural components.

The employee focus emphasizes the need to access required information and tools. The increasing service orientation and focus on empowerment is shifting the user base from background clerical resources to front-line service delivery staff. MIS must consider various options for distributing information and applications to support dispersed operations and requirements for flexibility. Architectures must reflect an integrated view without restricting future changes in legislation, programs, organizations and modes of service.

The architectural principle of timeliness stresses the pressures exerted by legislation, business relationships, global trends and emerging technologies, which must be addressed by information systems.

MIS architectures must be flexible, facilitating responses to changing business conditions and technology upgrades. Flexible systems will more readily incorporate new capabilities and modifications to existing decision flows. There is a growing need for an ongoing process to manage evolution, and a corresponding recognition that investment in architectural planning is required for longer term gain.

The concept of reusability supports both timeliness and flexibility. Architectures must exploit opportunities for common functions, information, tools and technology. Numerous opportunities exist, within a single department and across the entire enterprise, to perform similar business functions in a similar manner. A commitment to reusability reduces redundant design, development and implementation efforts, and associated costs. In addition, major benefits can be derived through increased

integration and portability of functions. Support for these concepts requires a corresponding investment in management resources.

Applications, information and technology must be planned, developed, implemented and maintained using a consistent set of methodologies and tools, promoting a consistent project management infrastructure across all projects.

2
Introduction to Table Driven Design

Table Driven Design is an approach to software engineering which is intended to generalize and simplify applications by separating program control variables and parameters (rules) from the program code and placing them in external tables.

Design objectives include an emphasis on modularity and decoupling program control data from application logic. Applications are made more flexible by postponing the time when control values and rules are bound to the processes they direct.

The basic principles of Table Driven Design are not new. They were originally developed and implemented in some software systems as early as the 1950's and 60's. However, for a number of reasons, they did not gain widespread acceptance at that time. Main memory was an expensive commodity, and disk access speeds were considered more than adequate when viewed in light of the manual systems that were being replaced. These factors led to an over reliance on disk accesses which proved to be a barrier in the later development of high performance table driven systems. In addition, the average programmer knew relatively little about writing efficient access methods, and there were no canned packages available for effective management of memory resident tables. As a result, data driven systems of the day were often more complex, and more expensive to develop and maintain, than their procedural counterparts. In spite of these difficulties, some of the early efforts were extremely successful. The first macro based operating systems for the IBM 360, for example, were extensively table driven, and they have evolved through a whirlwind of changing technologies and customer demands to support a wide range of applications on the current series of IBM mainframes.

Capabilities of today's operating systems, on-line transaction processing (OLTP) facilities and programming languages provide designers with a variety of alternatives for implementing systems and programs. Design decisions are influenced by experience, standards and performance constraints. Recent advances in hardware and software systems have relaxed constraints and forced designers to revisit business objectives, conceptual approaches and technological capabilities.

What is a Table?

Tables are concise, graphical representations of relationships. They transform information of one kind into information of another kind. The names of two cities, for example, may be transformed into the distance between them. A dictionary is another example of a table, although the graphical aspect is not immediately as clear. Similarly, any collection of conditions may be transformed into a series of actions in a decision table. A Monopoly card deck is an example of this kind of table. If the card is a "Jail" card, then the following actions are implied:

1) Go directly to jail
2) Do not pass go
3) Do not collect $200.

With regard to information systems, the defacto definition of the term "table" is a data structure consisting of a series of rows and columns. The number of columns in a given table is usually fixed, while the number of rows is variable.

A rose by any other name is still a rose. Related terms, such as array, list, stack, queue, index, control block, file, database, graph, and chart may all be used to indicate some form of a table. Sequence lists are simple tables that describe processing sequences, where the next step is related to some action. A reference table is a descriptive table that relates a series of codes to other values, attributes and messages. Systems analysts often use tables to accurately describe processing relationships. Decision tables can be used to describe the combinations of input characteristics used to generate each output result.

Most computable systems can readily be described by tables, and can be implemented through the use of tables. In fact, all computable systems can be described and implemented using a purely table driven approach, but this is not always practical.

Application Development

Table Driven Design applies to every stage of the application life cycle. Tables are used to define, design, develop and enhance the application system. They provide the foundation for a smooth flow through all stages:

- **Analysis**
  Tables provide concise, orderly specifications of the business challenge.

- **Design and Development**
  Tables can be implemented directly from specifications, providing a close link between theory and practice.

- **Enhancement**
  Shared tables allow for single, centralized changes, fast turnaround and high productivity, with minimal attendant risk to existing program code.

Degrees of Decoupling

In the recent evolution of expert systems, the application specific knowledge base (the set of rules which drive an expert system) has been decoupled from the inference engine of more generic program code. There are good reasons for this structure. Jack Smith and Todd Johnson of Ohio State University have advised that, *"the separation of task and search control knowledge makes it easy to modify the system's behaviour by adding new operators or new search control knowledge"*. Expert systems implement this separation to an extreme; more traditional business systems exhibit varying degrees of decoupling.

Within many corporate MIS organizations, there is now a clear understanding of the need to decouple process control parameters from application code to support the development of flexible application logic. Applications may be categorized according to levels of flexibility and shareability, from least desirable (least flexible/sharable) to most desirable.

- Rule details are integrated with high level decision logic in an inflexible procedural coding structure. This group represents the worst case: code which is difficult to modify, with no shared elements.

- Table data is located in a centralized place in the program's addressable working storage, but still integrated with an inflexible procedural coding structure. The potential for sharing control values within the application is only marginally improved.

- Tables are defined in working storage. Table data is centralized and initialized in working storage and shared within the application. Flexibility is improved by decoupling control values from generalized code. That is, each row in the table is interpreted by a given routine, in a processing loop. Periodic overflow of allocated table space is a common problem with this approach. Any change to table data or space allocation requires that the source code be recompiled and relinked to generate a new executable module.

- Tables are defined in working storage, using common copy macros. Both the table data and the structural definition of that data are entered into a common source code library and shared across a number of applications, at compile time. Subsequent changes to the table data or structural definition still require regeneration of all affected executable modules.

- Abstracted rules are kept in sharable Direct Access Storage Devices (DASD). Table data resides on external DASD, and is shared across several applications at execution time. File structure definitions may be copied at compile time or, in the case of database tables, definition information is also abstracted, for maximum flexibility. Programs need not be recompiled when control data in tables is updated. Rules are bound to the program at run time, not compile time. Performance is limited by excessive I/O and lengthy instruction paths for rule access. In addition, limited operators are available for managing rule sets effectively in memory (see Chapter 6 - Table Operators And Support Facilities).

- Abstracted rules are kept in high performance, sharable memory buffers. This level is intended to combine the advantages of sharable DASD, with the added benefit of high performance rule access. However, if memory management facilities are required outside those provided by the programming language, then any additional in-house development effort would have a negative impact on productivity. Vendor products are available to provide these services.

Abstraction of Rules

The abstraction of rules to separate control data from process is initially more intuitive than obvious, representing a continuum rather than a distinct boundary. Some straightforward heuristics are available to guide designers through the processes of engineering and re-engineering to build table driven systems.

Traditional design has, for largely historical reasons, embedded tables of rules within program logic in various forms, such as IF/THEN/ELSE and other branching constructs or declarations of arrays in working storage.

The following business rules:

```
IF TAXPAYER-TYPE = '01' OR
   TAXPAYER-TYPE = '02'
   PERFORM ACTION-X.

IF TAXPAYER-TYPE = '01' OR
   TAXPAYER-TYPE = '03'
   PERFORM ACTION-Y.
```

could be abstracted by placing the detail values '01', '02' and '03' in a table, along with indicators for associated actions, and then checking the table at execution time to determine what actions should be performed for the given taxpayer type. Equivalent table driven code would be of the form:

```
MOVE TAXPAYER-TYPE TO DECISION-TABLE-SEARCH-KEY.
PERFORM TABLE-LOOKUP.

     IF TAXPAYER-TYPE-NOT-FOUND
        PERFORM ERROR-ROUTINE.

     IF ACTION-X-IS-TO-BE-PERFORMED
        PERFORM ACTION-X.
     IF ACTION-Y-IS-TO-BE-PERFORMED
        PERFORM ACTION-Y.
```

This would allow for changes in actions performed for a particular taxpayer type simply by updating the table, without modifying the program code itself. This type of logic construct is sometimes called "action oriented", as opposed to the more traditional "condition oriented" constructs so often found in procedural code.

Figure 2 lists the steps in processing a typical change request for a traditional procedural application.

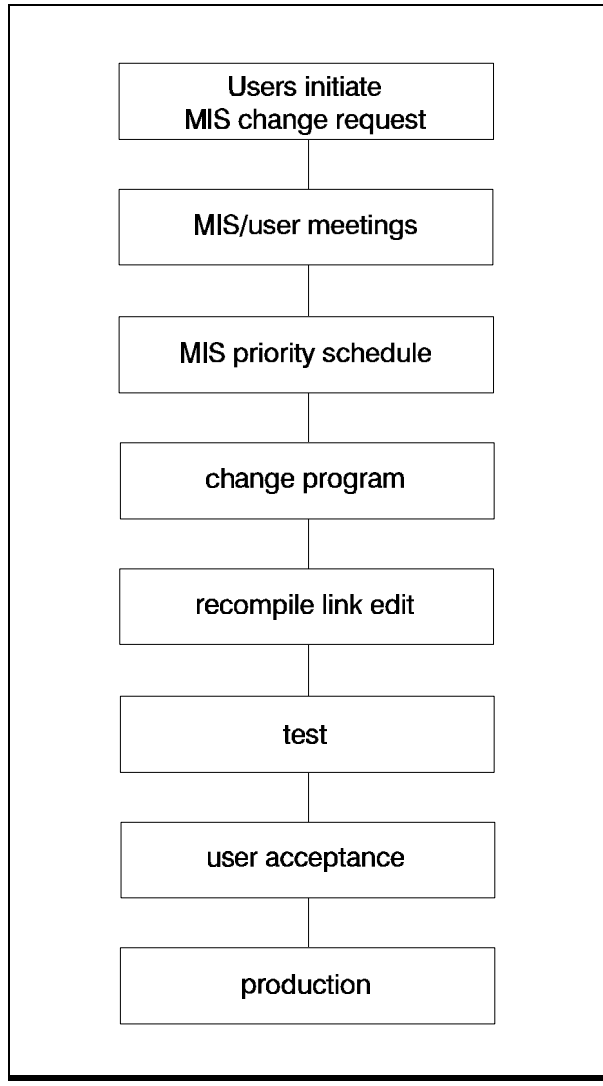


Figure 2 *Traditional Maintenance Process*

Figure 3 lists the steps involved in processing a change request for an idealized table driven application, where the user has direct update access to business rules in tables (see Chapter 8 Administration of Tables - Quality Assurance, for a discussion of quality assurance issues).

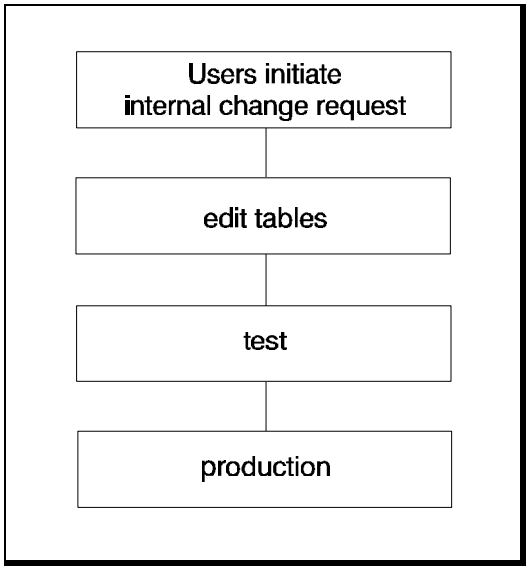


Figure 3 *Table Driven Maintenance Process*

Figure 4 provides a graphical, high level overview of the procedural approach to system maintenance. As the real world business environment evolves, user change requests for a given application often accumulate in the pipeline of the infamous MIS maintenance backlog. Multiple, unrelated change requests are queued and applied collectively to minimize overhead. For change requests which were initiated first, this has the unfortunate effect of introducing additional delays.

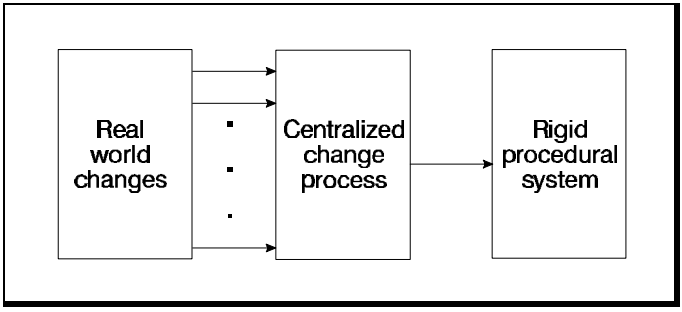


Figure 4 *Procedural Approach to Change*

In the table driven approach of Figure 5, multiple change requests are distributed and applied independently, in parallel, across a series of tables. Each table describes some set of attributes for objects in the evolving business world. Business experts who request changes to business rules are also the ones most qualified to apply those changes directly to the tables which drive the application. MIS overhead is minimized and, in many cases, MIS involvement can be eliminated entirely.

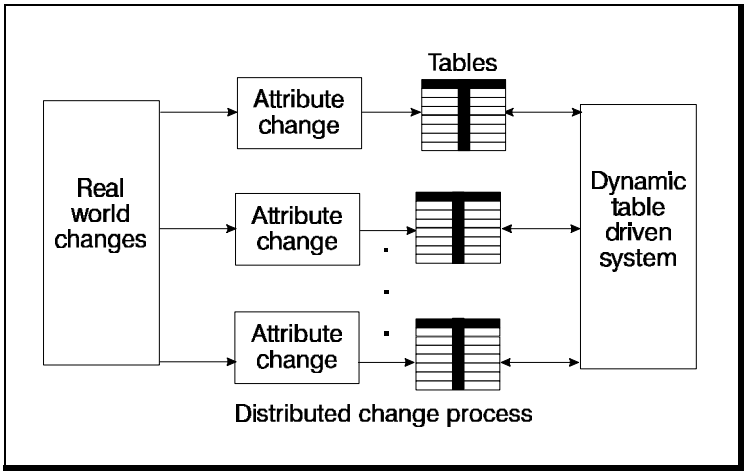


Figure 5 *Table Driven Approach to Change*

The driven aspect of Table Driven Design is already familiar to most programmers. It is commonly found in the traditional concept of a driver program in transaction processing. Logic paths are selected at execution time, depending on the literal value of the current transaction code. Here, though, the transaction code is usually directly associated with a given subroutine in a procedural manner. There is no way to change this association without changing the driver program itself. A truly table driven approach would relate transaction code to subroutine indirectly via an external table.

The problem of inflexible design is inherently related to the problems of duplication and inconsistency. Making the design more flexible reduces duplication and inconsistency; conversely, reducing duplication and inconsistency makes the design more flexible.

The Model and the Business

Any successful design approach, table driven or otherwise, is not simply a question of adhering to a design philosophy, but rather, a question of addressing the business problems at hand. Elements and processes of the application model should accurately identify and reflect corresponding elements and processes of the business problem, including their attributes and relationships. Business rules which drive the model

should be concisely defined and easy to locate. Then, as attributes and relationships change in the real world, descriptions in the model can more readily evolve in a natural and analogous manner.

Insofar as a Table Driven Design approach contributes to solving business problems, the design principles should be applied to business information systems. However, design extremes of any measure should be avoided in favor of hybrid compromises. Rarely is a single perspective sufficient to address non-trivial problems. Table Driven Design is an extension of traditional procedural design and not a replacement for it. In subsequent sections, Table Driven Design shall refer to a class of designs with table driven components, which may themselves be connected by hard coded procedural logic.

3
Classes of Tables

Performance and maintenance concerns should determine the manner in which particular classes of tables are implemented. Tables may be classified even further according to the application objectives they serve.

The information systems definition of tables, as a fixed length data structure consisting of a series of rows and columns, is independent of a particular internal or external storage technology. Tables may be implemented in a number of ways, in memory (also known as main memory, central memory, main storage, or Random Access Memory) and on peripheral DASD. Memory resident tables may be further categorized as internal to the program (statically linked with the program code as part of the program's working storage) or external to the program (memory dynamically allocated outside the application program area).

At first glance, the choice between memory based and DASD based implementations for table structures is not necessarily clear. For example, what are the essential differences in functionality? What kinds of data are best suited to each environment? In attempting to answer these questions, it is helpful to think in terms of two high level categories of table data, process related data and data which is processed.

Process related data is information which tailors the process for a specific set of circumstances. The data is used to set values for parameters which guide or modify a generic algorithm. On the other hand, data to be processed consists of the primary input and output.

Examples of process related data include:

- State/province tables
- Decision tables
- Code translation tables
- Rate tables
- Tax tables
- Message tables

This class of data is accessed repeatedly during execution of an application and for performance reasons, it is usually implemented in the form of memory resident structures.

Examples of data which is processed include:

- Customer transaction data
- Customer master files
- Historical databases
- Financial reports

This class of data is characterized by relatively infrequent access and is usually implemented in the form of files and databases.

Because process related data is inherently a part of the process itself, it may have an impact on large portions, if not all, of the data which is processed. The converse is not true. For example, data in a single row of a state/province table is accessed repeatedly through an entire processing run, affecting many customer transactions, while the impact of a single customer transaction record is generally localized to a single customer master record.

Memory Resident Tables

Tables may be implemented in memory, hard-coded in the application program for optimal performance, as IF/THEN/ELSE condition/action relationships, other conditional branching constructs or array-type data structures in working storage.

Alternatively, tables may be implemented as data structures resident in main memory, but external to the application program. They are kept under the specialized control of a memory management system optimized for table structures, providing the combined advantages of high performance and easy maintenance.

Tables in memory may be initialized at compile time or at execution time. In particular, tables may be loaded at execution time from external DASD storage to allow for changes to table data independent of changes to the program code.

Database Tables

Tables may be implemented as file or database structures on DASD. Here they are maintained under control of an appropriate file access method or database management system.

At a fundamental level, database management systems are file management systems, designed on the basic principle that data is more likely to be on DASD than in memory when it is needed. They allow entire tables to be loaded into main memory buffers only as an afterthought or exception, with the direct result that operator sets have been developed as a reflection and extension of file management operators. No operators are provided to capitalize on the fact that the entire table resides in memory. All operators function within a domain defined by the capabilities and limitations of external peripheral devices. This proves to be restrictive if application requirements indicate that the table could be processed more effectively in a total memory environment.

Database processing is I/O intensive. Relatively long instruction paths and wait times are required for accesses to indexes, data records and log records.

Database tables are typically very large data objects with relatively low volume access patterns. Batch transaction runs may access less than 1% of the data records in any single run. Customer master data falls into this category.

Database indexes are usually either predefined by the database administrator or automatically determined by the database management system at "binding" time, before the application is executed. They are not generated dynamically at run time.

Because of the large volumes of data and the I/O intensive nature of the operation, database tables are not reorganized in parallel with other accesses. They must be reorganized off-line, in order to allow online processing to continue without unreasonably lengthy interruptions or performance delays.

Database tables are not normally used for temporary data structures; they generally have very long retention periods.

Operational Characteristics of Tables

How a table structure is to be processed should ideally determine whether it will reside in memory or on DASD. Operational constraints are characterized by such factors as processing sequence, frequency of access, table size and update activity. All these factors must be weighed in balance to determine how a table should be implemented.

Processing Sequence

Table structures which are processed sequentially in a single pass do not benefit from residence in main memory and are better left on an external device. They may be handled effectively on a piecemeal basis one block at a time, in relatively small buffers. Data structures which are accessed randomly perform significantly faster if all, rather than part, of the data is loaded once into memory, to remain there for the duration of processing. Alternating accesses to table rows in two different blocks, for example, will then require no additional I/O.

Frequency of Access

If the frequency of access is less than once per block in a given processing run, then the table could be accessed effectively as a database. If there are multiple accesses per block, then the table should be entirely resident in memory. Loading data into memory should be avoided if it is not going to be heavily accessed and, conversely, I/O should be minimized for frequent access.

In the case of a randomly accessed table, for example, it is often clear that the same blocks will be repeatedly hit by retrieval requests in a single application. Similarly, multiple applications may make numerous passes through a sequential table. In these situations, the tables should be resident in memory to minimize I/O overhead.

Size

When is a table too large to load entirely into memory? This is a common question which has no simple answer. The question of size must be answered in the context of greater systems considerations.

Benefits of memory resident tables are ultimately dictated by access requirements and performance considerations, not size. Very large tables should be subjected to stringent requirements analysis. If it makes sense for an application to access a table in memory, then the table should be in memory, regardless of size.

Optimal application design, though, may be overridden by system support constraints. Some tables, due to real memory resource restrictions, are simply too large to load entirely into virtual storage. That is, overall system performance would be degraded due to excessive paging. To most analysts, however, these tables may still meet the requirements for table driven, flexible application design. For practical purposes, they may have to reside in database environments until sufficient real resources are available to address the higher level system constraints. The tables can then be moved where they truly belong, in memory.

Update Activity

If there is a real application need for auditing or checkpoint/restart facilities, then a database table is indicated. Memory resident tables are cleared on abnormal job termination (with the possible exception of shared system tables). This will include all updates which may have been applied to the table. Any requirement for logging table updates onto DASD should come under close scrutiny as a costly alternative to efficient processing using memory resident tables. If checkpoint and restart capabilities are in place simply as a time saver to avoid redundant processing in the event of program failure, and not for auditing reasons, the overhead may be counterproductive. Starting fresh periodically is often more economical than logging updates regularly for restart purposes.

Functional Characteristics of Memory Resident Tables

Certain categories of table usage, or functionality, generally imply that the table processing patterns will fit the above guidelines for memory residency. The two major categories include semi-stable data and transient data.

Semi-stable Data

Semi-stable data has a high ratio of read access compared with write access. For Table Driven Design, this includes decision data (program control) and reference data (constants and parameters). The data may be updated daily, but there is the implication that there are high volumes of retrievals between update cycles. This data does not normally need to be in a database. Logging of individual updates is not a requirement.

Program Control Data

This is a special class of semi-stable data which contains decision control information for determining which routines should be executed under a particular condition or set of conditions. Control values of this nature have traditionally been hard coded in application programs for optimal performance, but that approach introduces a level of program complexity which hampers subsequent maintenance efforts. The data properly belongs in memory, separated from program code and accessed by generalized table driven routines. Tables of this type include control, decision, rule, switch, state transition and navigation tables.

See Chapter 5 - Decision Tables, for an introduction to decision table terminology and processor logic.

Constants and Parameters

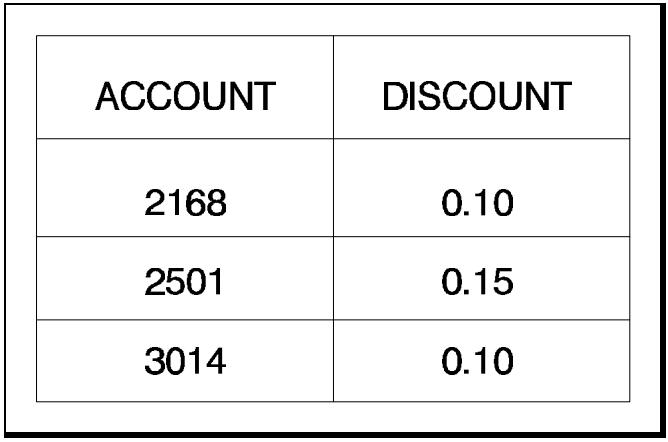
Constants, literals and other parameter values that make up a significant part of a program's working storage are suitable for implementation as memory resident tables. These may be loaded from external storage media for maximum flexibility. This is another special case of semi-stable data, which should be entirely resident in memory, but easy to maintain using data management facilities rather than modifying program code. These tables include reference, parameter, lookup, validation and translation tables.

A comparison of procedural and table driven program code which uses these kinds of reference constants helps to illustrate the pivotal role played by tables. Refer to the procedural example of account processing code shown in Listing 1.

```
IF MASTER-ACCOUNT = '2501'
THEN
   MOVE 0.15 TO ACCOUNT-DISCOUNT
ELSE
   IF MASTER-ACCOUNT = '2168' OR
      MASTER-ACCOUNT = '3014'
   THEN
      MOVE 0.10 TO ACCOUNT-DISCOUNT
   ELSE
      MOVE 0.00 TO ACCOUNT-DISCOUNT.

COMPUTE TRANS-AMOUNT = TRANS-AMOUNT - TRANS-AMOUNT
                     * ACCOUNT-DISCOUNT.
```

Listing 1 *Procedural Account Processing*

Now consider the sample Reference Table in Figure 6, which contains the same master account numbers that are hard coded in the procedure of Listing 1.



ACCOUNT	DISCOUNT
2168	0.10
2501	0.15
3014	0.10

Figure 6 *Reference Table*

The corresponding table driven program code shown in Listing 2 uses the account reference table from Figure 6. Note the table access routine called TBCALL, which is an interface to a memory-based table management system known as tableBASE.

```
  MOVE 0.00 TO ACCOUNT-DISCOUNT.
  MOVE 'FK' TO ACCOUNT-COMMAND-ID
                OF ACCOUNT-COMMAND-AREA.
  MOVE MASTER-ACCOUNT TO ACCOUNT-SEARCH-KEY.

* FETCH (BY KEY) ACCOUNT INFORMATION INTO
                                    ACCOUNT-ROW.

  CALL 'TBCALL' USING ACCOUNT-COMMAND-AREA
                      ACCOUNT-ROW
                      ACCOUNT-SEARCH-KEY.

  COMPUTE TRANS-AMOUNT = TRANS-AMOUNT - TRANS-AMOUNT
                         * ACCOUNT-DISCOUNT OF
                           ACCOUNT-ROW.
```

Listing 2 *Table Driven Approach Using a Reference Table*

Figures 7 and 8 represent high level overviews of the use of control tables and reference tables, respectively, in applications.

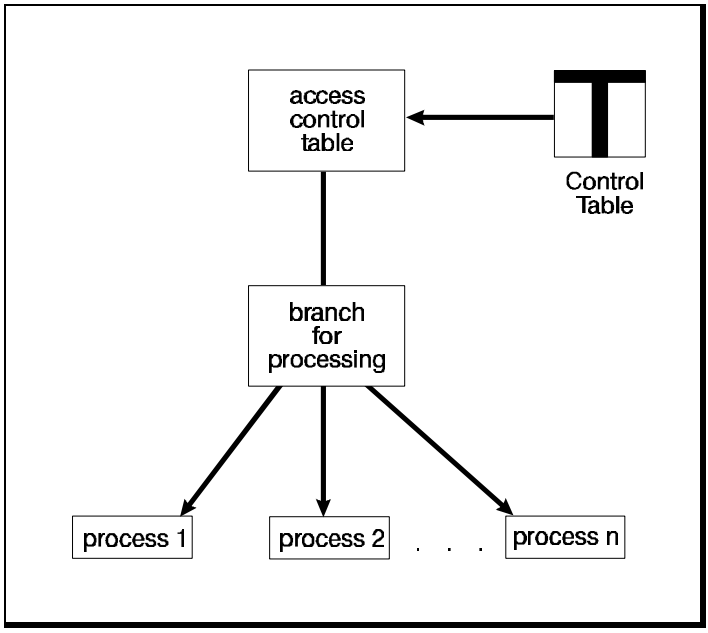


Figure 7 *Using Control Tables to Drive Program Logic*

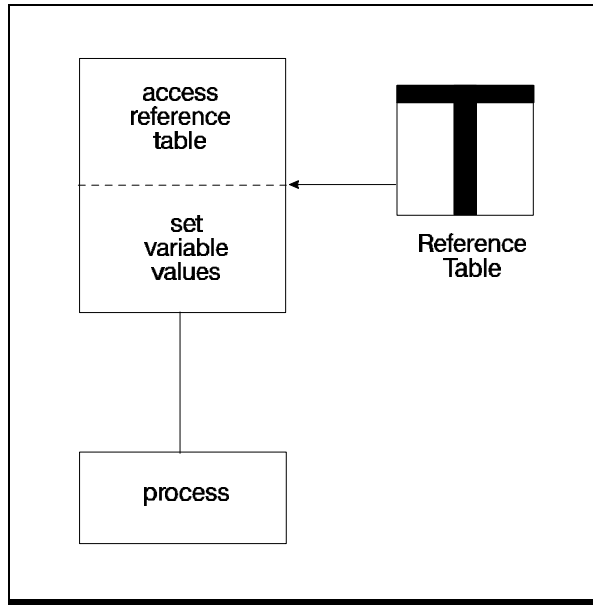


Figure 8 *Using Reference Tables to Drive Program Logic*

Transient Data

Transient data lies at the other end of the spectrum from semi-stable data. It is structured and accumulated specifically for the application, rarely required for sharing, and constitutes a significant portion of working storage in most online and batch applications. Exact volumes of transient data to be stored in tables are often unpredictable. For this reason, such tables have often been implemented as expandable external work files, rather than as fixed size, memory resident arrays subject to overflow. The tables are often built during one processing pass, then used in another pass to drive a subsequent reporting step. Once a process is complete, transient data is not retained. It will typically be regenerated each time the process is initiated. There is no need to create permanent images of the data on disk. This type of data should exist only in memory, to eliminate unnecessary I/O and to streamline access paths.

Probably the most frequent example of transient data in a batch environment is to be found in the summarization of large volumes of data (see Chapter 7 - Data Summarization). Other transient tables may be used in real time environments, where processing rules and decisions are based on aggregates of information which are in a constant state of evolution. There is no permanent version of such data available to the system. These tables provide a foundation for optimization algorithms in areas of performance tuning, effective resource utilization, traffic control and dynamic financial transactions.

Working Storage

The concepts of semi-stable and transient data describe all constants and variables normally declared in working storage. Taken to an extreme, the logical conclusion is that all working storage values are candidates for relocation in external tables. There is no reason that these constants and variables have to be link-edited with the program code if they can be accessed efficiently, in memory, external to the program code. Benefits of this approach extend beyond contemporary maintenance concerns.

Separation of instructions and data supports the goals of shareability and reusability. If a single copy of application program code is to be shared by several tasks executing concurrently, then task-specific (transient) data can be organized in task-specific tables, while common (semi-stable) data in other tables is shared by all tasks. In this way, the effect of any one task is isolated from all other tasks while sharing of components is maximized. This is true even for data which is not obviously tabular or array-like in structure, such as counters, switches and other non-repeating variables. (Not all rules are tables.) All these variables could potentially be collected together as one single record or row, in an external table, to be loaded into memory when the task is initiated, and accessed as required.

Benefits of this extreme approach to the separation of instructions and data ultimately depend upon the degree of sharing desired, and also on the frequency with which working storage variables might be changed as a result of ongoing enhancements to the application.

Grey Areas

How a table structure is to be processed should ideally determine whether it will reside in memory or on DASD. However, the real world does not always provide an ideal playing field. It may be that a particular table structure meets most of the criteria for residence in memory, but also requires additional facilities available only in a database management system. For example, a small insurance rate table may be accessed randomly and frequently, and it may be described as semi-stable data. At the same time, government regulations or corporate directives may well insist that all updates are logged to provide an audit trail. Should this be a memory resident table or a database table? Another larger table may be accessed randomly and frequently by one application, yet processed sequentially in a single pass by a second application. Such tables fall into a grey area between those which are clearly memory resident structures and those which are obvious database structures. There are three ways to deal with these tables.

1. Given that there are advantages and disadvantages to both approaches, the designer could simply choose one or the other and accept a reasonable, but less than perfect solution.

2. The table could be stored externally as a database, to be accessed directly by those applications which require DBMS facilities, but loaded into a transient, memory resident table for high performance processing by other applications.

3. If an application requires the advantages of both approaches simultaneously, then a mirror image of the database could be loaded into memory, and both images could be accessed using whatever facilities are appropriate through a memory resident data manager or through the DBMS. Retrievals would access the table in memory, and updates would be applied to both images.

4

Tables and the Application Development Life Cycle

To paraphrase Robert Burns, "the best laid schemes o' mice an' men go oft' astray". Unfortunately, this has been proven many times in the field of information systems development. Development methodologies are intended to enforce a system of checks and balances for quality assurance to catch problems of poor design at an early stage. Table Driven Design is not a silver bullet, guaranteed to eliminate poor quality. A "tables from Hell" scenario is possible, but it is generally more difficult to produce a bad design with these techniques than with traditional approaches, assuming guidelines are followed. Analyst and designer are brought closer to the business problem, and tabular specifications improve the quality of communications.

Development Methodology

Some specific development methodology is actively promoted within most MIS organizations and there are proven benefits to development methodologies in general. Such traditional guidelines tend to suggest, however, that implementation in production marks a point near the end of the system life cycle. From the perspective of Table Driven Design, production is just the beginning of a much more lengthy and costly period of system enhancements. Maintenance is not addressed in most development methodologies except as a lesser reincarnation of the development process, independent of the original specifications.

Many organizations are moving away from the classical waterfall type of life cycle to more clearly iterative life cycles, such as prototyping. Various modern development methodologies stress the importance of identifying and managing business rules as a foundation for good design.

Table Driven Design is itself not a methodology, any more than traditional procedural design constitutes a methodology. A design approach is a strategy for modelling business problems according to a set of architectural principles and business directions. A development methodology is a procedure for building a particular design and carrying that design through to a finished product. With respect to application development in general, design approaches may be considered strategic, where development methodologies are tactical. For established and emerging development methodologies,

Table Driven Design provides an intellectual framework which is both consistent and complementary.

Analysis

Information analysts are still striving for a paradigm that can bridge the language gap between business and information systems professionals. It is widely recognized that business experts and technology experts must work much more closely together than they have in the past. There are various ways this may be accomplished. Two extreme approaches may be to turn programmers into business experts, or to turn business experts into programmers. Either way, expertise is diluted, and numerous other problems are introduced. These are obviously not practical solutions.

The translation of business language, through any intermediary, to software language introduces a desirable degree of formalism and automation to the business by creating the application. However, all too often, it also introduces errors into the business process. The translation itself cannot be eliminated, but it should be minimized and simplified. Tables of business rules provide a basis for doing just that.

In the design of rule based systems, we are faced with a fundamental change in attitude. Instead of having information analysts try to predetermine the behaviour of an application in a problem space which is continually changing, a more effective approach focuses on designing application dashboards, or control panels. This allows the user to visualize prevailing business constraints (defined in tables) and drive the application through the problem space. Here the term "drive" is more than just a play on words, and the analogy of a dashboard is literal as well as figurative.

Business rules may be defined as a set of constraints placed upon the business. By separating application business rules (tables) from the driving process (sometimes called the inference engine), user specifications are largely expressed in terms that are readily understood by both business experts and MIS personnel alike.

The systems analyst must adopt a particular mindset favoring the development of tabular structures in a table driven application. At the analysis stage, application tables do not necessarily exist. This is the point where table structures are formulated as a description of the business problem. The initial analysis then drives all subsequent processes.

Specification Tables

Specification tables are frequently provided to programmers in today's development efforts. They are then proceduralized according to traditional programming styles. In this case, subsequent enhancements to specifications will also have to be proceduralized

for integration into existing program code. If the original implementation follows the spirit of the specifications as closely as possible, maintenance becomes easier. That is, if the programs are actually driven by physical equivalents of the specifications tables, then subsequent modifications to the specifications should require updates to the physical tables and minimal enhancement to the programs.

Specifications are also frequently provided to programmers in the form of natural language pseudo code. Standardized graphical representations have inherent advantages over corresponding pseudo code for business rule specifications. "One picture is worth a thousand words" is a familiar cliché which reflects only the conciseness of table structures. There are other important benefits of tables over pseudo code. Searching through a normalized set of specifications to locate a particular rule is simpler. Consistency and completeness of the rule base may be determined with relative ease.

These characteristics of tables have profound implications for the maintenance of application systems.

Chaos Theory and Information Systems

"Chaos theory" is concerned with the study of systems and events which appear to operate in a random fashion but, at a higher level of analysis, are well defined and structured according to mathematical constraints. Weather, for example, is chaotic. Although weather patterns may seem to evolve in an arbitrary fashion, they operate according to established laws of physics. Theoretically, it might be possible to predict next month's weather if the entire state of all particles in the universe was known since day one. Unfortunately, Heisenberg's Uncertainty Principle asserts that it is not possible to know both the position and velocity of even a single particle at any time, let alone all particles in the universe. According to one of the more popular tenets of chaos theory, a butterfly flapping its wings in one part of the world, say Los Angeles, will eventually and inevitably affect the weather in places as far away as Montreal or Istanbul. This makes accurate specifications for long range weather prediction impossible. For a comprehensive introduction to chaos theory, refer to James Gleick, "Chaos - Making a New Science".

Real world business systems are chaotic in nature. There is a high level form and structure to business operations which is consistent over time, although the low level factors governing day to day events are often unpredictable. Business information systems are reflections of real world, dynamic systems. Subtle changes in the business environment can often affect details of the business itself in unexpected ways. In such circumstances, maintenance of business systems is unavoidable. Fortunately, although details of original system specifications may change, this usually occurs within well defined, higher level business parameters. The task of the analyst is to separate stable, high level function from more volatile low level details. For example, new district offices might be expected to be established during a corporate expansion. Exactly

which offices these will be is unknown, but they will probably be processed using the same collection of functions that apply to existing offices.

Chaos theory is said to be independent of scale and applies at both microscopic and macroscopic levels. This characteristic has important implications for some types of software models (see Chapter 7 - Resolution Independent Architectures).

Allowing for Change

All business entities and their attributes should be identified in the analysis stage. This includes:

- Primary input and output data
- Processes
- Tables
- Rules

Entities should be organized in hierarchies to facilitate top down development.

A functional decomposition of processes is advantageous in identifying all low level, atomic actions performed. These actions may then be isolated to avoid duplication and maximize flexibility in the event of future change. Software reuse strategies must be integrated into the entire life cycle process to maximize reuse opportunities (see Chapter 4 - Reusability and Shareability, and Generalization).

Software frequently changes in a regular fashion. Over successive changes, however, traditional procedural software loses its original structure unless additional work is done to restore that structure. Table Driven Design is intended to minimize the impact of change. Analysis efforts should include an end user survey to identify entities and attributes which are candidates for potential change. Users should be guided to describe general classes of frequent or otherwise expected system modifications, beyond currently supplied specifications. The objective here is to reduce the need for future backtracking in the application design.

Process control values in tables are subject to the same normalization considerations and benefits as other database types of data, along with other, more logic-specific concerns (see Chapter 5 - Decision Control Structures). Normalization analysis reduces complexity by segregating dissimilar rules and aggregating similar rules.

Analysis for Table Driven Design has much in common with Object Oriented Design (OOD), but excludes such concepts as inheritance and message passing. Object oriented Design, on the other hand, places no explicit emphasis on memory resident control structures. It is important to note that, although the two design approaches are related, they are not equivalent.

From Analysis to Design

The analysis phase produces a set of tabular specifications relating business problem conditions to business solution processes. The need for analyst/designer communication begins during analysis and is reinforced through the design stage.

The design phase takes tabular and procedural specifications from the analysis phase and uses them to develop a robust technical model of the system. Where the analyst focuses on theory, the designer concentrates on practical implementation.

During the design phase, specification tables may be restructured, split and/or merged, based on technical facilities and constraints.

The need for analyst/designer communication remains strong throughout this phase, as it does with traditional design. This helps to ensure that end user requirements are not misinterpreted, and that technical considerations are accounted for in the analysis. For example, such a dialogue helps to correlate the expected business potential for change with the technical impact of associated system maintenance.

Table driven components are linked together at stable, procedural junction points in a tabular/procedural compromise which is visual, formalized and easy to document. Once the design is complete, the application is ready for development and testing.

New Application Development

New application development implies some strategic advance in business operations, providing new functionality for business systems. Existing applications are not normally affected, except at interface points.

There are a number of well defined problem areas to be addressed in the development process:

Interrelationships between Tables

High level logic flows are determined by the sequence of processing individual tables. Table driven processing patterns are connected by procedural logic. High level logic can be coded independent of the results of table lookups, before lower level detail paths are constructed.

Rules in Tables

Low level logic paths are determined by the contents of the tables. At execution time, one unique path is selected, according to prerequisite conditions (the table key), from among a number of available options (rows in the table). In this context, the terms "row" and "rule" are equivalent.

In the case of a reference table, one generic (parameterized) piece of logic in the program is tailored by values in a specific row of the table. For decision tables, separate logic is normally associated with each function identified in the rule. Functional logic may be coded and linked together with selection logic in one module or, alternatively, functional routines may be dynamically loaded at run time.

Table Access Interface

Once the high level logic is in place, common table access routines can be coded for each table to load the table and to select a detail rule at the appropriate time. Table management logic is standard for all tables and can be reliably unit tested. Main memory table management systems, such as tableBASE, are available to automate this process.

Detail Development With Incomplete Tables

Detail logic for each processing path may be coded in the program one path at a time. It may be tested independently of other paths, by entering in the tables only the rows which direct processing along the desired paths. These tables are therefore incomplete. Note that incompleteness requires that logic be included to deal with cases where a rule is not found in the table. Every incomplete table must have a "default rule" to handle this condition. This is not a requirement for complete tables, where all possible condition sets are supposedly guaranteed to be represented in the table, but it may still be advisable to allow for a "not found" condition to deal with potential errors in either the program or the table.

Incomplete tables are particularly useful when very large sets of conditions are involved, but only a small portion of those conditions are normally encountered in day-to-day processing. An application can be designed to add new or rare conditions, along with appropriate actions determined by the user, as they are encountered over the life of the system (see Chapter 5 - Sparse Decision Tables).

Defining Tables

Table organizations and search methods should be designed for maximum access efficiency. Note that language dependent array structures may be limited in the choice of in-memory organization and search method, but memory based table management

systems offer a variety of choices. Alternate view requirements should be taken into account for all tables. Record layouts must be defined for all tables, including all condition (key) fields and action (result) fields.

It should be assumed that table data may be displayed on a screen or printed in hard copy, according to ad hoc requests. Display format information should be included as part of the table definition, and a generic print/display utility should be provided to accommodate ad hoc requests.

Populating Tables

Once the tables have been defined, and associated logic has been coded in the program, tables may be populated for testing. Ideally, this involves creation of a DASD image of the table, to be loaded at execution time.

Reusability and Shareability

Table Driven Design supports the reuse of tables, generalized logic and detail logic, within a single application or across applications.

In some multi-processing operating systems, tables may be placed in special dataspaces to allow for sharing of a single copy in memory, across multiple program address spaces and client workstations. In online transaction processing environments, reentrant generalized programs may be shared in a common memory pool. (Refer to comments on reusability and shareability in Chapter 3 - Working Storage). Tables and programs in DASD libraries may be shared system wide, of course, allowing for creation of private copies in memory.

Table Driven Design approaches to resource sharing are only part of wider corporate software reuse strategies. The MIS group must take steps to ensure that reuse becomes enshrined in the corporate culture. These steps include:

- Specification of the domains where opportunities for reuse exist and identification of criteria to prioritize, qualify and select domains for application of reuse techniques. A domain is the functional area covered by a family of systems, or across systems, where similar software requirements exist. Domain analysis is the study of similarities and differences among related systems within a domain.

- Identification of products, procedures and services which facilitate reuse of software components. A component may be defined as requirement, architecture, design or implementation information.

- Determination of ownership criteria and certification standards for reusable components (see Chapter 8 - Administration of Tables.)

The objective is systematic reuse, not opportunistic reuse. There is no single, all-encompassing approach to reuse. Strategies should include any heuristics which help to avoid re-inventing the wheel. Although a long term goal is to reduce life cycle costs, a significant short term investment in an infrastructure supporting reuse is clearly necessary.

Generalization

Source and executable code may be reduced dramatically by using one adaptable module in a Table Driven Design, rather than several tailored versions of essentially the same procedural module. Generalized procedures are intended to mirror the more predictable high level components of chaotic business systems. Variations in control values at lower levels appear in a concise form as rules in tables instead of the more wordy procedural representations, wrapped in redundant conditional logic. Business rule details are said to be externalized in tables. This has corresponding implications for reducing redundant support efforts and application inconsistencies.

In any generalized process, various actions are executed under various sets of conditions. Processes may be described as condition oriented or action oriented.

A condition oriented process executes a group of actions associated with the given condition set, one after another. References to subgroups of actions may appear several times for different condition sets, resulting in duplication of the code which invokes those actions. Traditional procedural processes are usually condition oriented.

An action oriented process attempts to execute all actions in turn, for a given set of conditions, and depends on a series of flags to indicate whether or not a particular action should be performed for the prevailing condition set. Each action is referenced only once. Table driven processes tend to be action oriented.

Prototyping

Selected rules may be developed and tested in isolation by populating tables for restricted sets of logic paths (see Chapter 4 - Testing). The result is a succession of prototypes, not just in the development phase, but throughout the entire system life cycle. The production version is simply the latest prototype, designed for continual adaptation to change.

Through the development process, new rules and new functional modules may be added to the tables and module libraries gradually, in a top down manner. New

generalized functions may require additional tables to contain the specifications of lower level detail rules. Again, these may be added in a top down manner.

There are two ways to use prototypes. To some, the term "prototyping" implies the construction of a series of sample screen formats, with no associated program code. These prototypes are generally discarded after use. On the other hand, the term "protocycling" is sometimes used to indicate that functioning program code exists and previous prototypes are not discarded, but are built upon to create subsequent iterations. A reusable approach to prototypes is implied in the context of table driven systems development.

Extended Capabilities

Every new paradigm brings new horizons into view. What was not practical from one perspective becomes eminently doable from another. Using tables, for example, allows for a new approach to the automated summarization of data.

The traditional procedural approach to summarization sorts detail data first, then summarizes detail records for reporting. This procedure is so common, it is simply accepted as "the way it's done". That there is another way at all, comes as a surprise to many programmers. A table driven approach reverses these steps. Detail data is accumulated in a memory resident table first, using a high performance search method. This is then followed by an internal sort and report of the summarized data. Details and benefits of this algorithm are described in Chapter 7 - Data Summarization.

In addition, many interpretive algorithms become practical through the use of high performance memory resident tables. Data dictionary tables, for example, can contain all the information required to dynamically evaluate algebraic expressions. Reference to data fields can then be generalized within the program and ultimately resolved in a data driven, rule based manner, using specifications external to the program code. Field name references are interpreted, rather than compiled and link edited.

Maintenance and Enhancement of Existing Systems

The terms "maintenance" and "enhancement" may mean different things to different people. For some, maintenance refers to the correction of program bugs, and enhancement indicates a modification which changes or adds to application functionality. Table Driven Design is intended to address issues in both of these categories, and in subsequent sections, the two terms may be used interchangeably to describe a wide range of modifications to a system. The primary emphasis is on

facilitating anticipated classes of future enhancements. Reduction of program bugs is more of a side effect.

In any case, enhancements to an existing table driven application become, ideally, a matter of updating the appropriate driving tables. This implies that the general type of enhancement required was anticipated in the original design and specifications already exist in tables, ready to be modified or added to. If the general type of enhancement required was not anticipated in the original design, then the program code will have to be modified and, possibly, additional tables may have to be added to the system. This situation is subject to the same considerations as any traditional legacy system.

Testing

Following the logic flow through a generalized procedure does not always help to debug a program in a test environment, particularly if the logic problem is the result of an incorrect decision specification in a table. In the case of one particular rule, a specific path through generalized logic cannot be determined on the basis of the program alone. The detail rule values directing the logic flow reside in the table, not in the program. All table data should be validated for syntax errors at data entry time. What can be done, then, to diagnose semantically erroneous but syntactically correct table data?

Very simply, every table driven application should include some form of trace capability for development and test environments. The trace should be designed to display the results of each table access. This enables the programmer to easily follow particular paths through the logic for any test data to explain the implications of specific rules. As an alternative to application specific trace facilities, vendor supplied table management tools or debugging tools may provide this functionality.

In the same way that Table Driven Design facilitates good, hierarchical documentation by removing control flow specifics from generalized logic, so it also allows for isolated testing of high level control flows, independent of detail logic. At the detail level, individual paths can be selectively tested, through rule-specific logic. This may be accomplished by including or excluding, in tables, the rules which trigger those pathways.

5
Decision Control Structures

There are a number of accepted statistics available to measure the quality and complexity of application software. Many complexity metrics relate directly to decision control logic flows, such as the quantities and characteristics of branching constructs. Some of these characteristics are discussed in detail in the following sections.

Decision Fundamentals: Structure and Sequence

Any sequence of decision points in a program defines a decision tree. Nested IF/THEN/ELSE constructs define a binary decision tree. Only two branches (THEN and ELSE) are possible at a given decision point. Nested case constructs, such as those using the COBOL II EVALUATE/WHEN statement, form n-ary tree structures, where n implies any number of (WHEN) branches at a given decision point. For the following discussion, this distinction is largely irrelevant, since a given n-ary tree can be converted to an equivalent binary tree.

There are inherent difficulties associated with the development of any system which has an excessively large number of logic paths. For example, where does one even begin to specify the decision control structure? What determines the order of IF/THEN/ELSE statements in the decision tree? If processing depends on country, business, division, department, section and transaction, then which of these control values should be examined first? Some ordering must be selected, and traditional procedural design imposes a rigid, hard coded implementation of that choice. The following sections lead to a discussion of normalized, tabular decision structures, where the ordering of decisions and business rules is considerably more flexible.

Duplication of Decision Control Logic

Decision control logic in a program is often duplicated at various points across a given level in the decision tree. The amount of duplication depends upon the level in the hierarchy, and the order and character of decisions up to that point. Application maintenance which includes modification to one instance of a decision construct

requires that all duplicated occurrences of that decision also be located and modified. Generally, the lower the level in the hierarchy, the greater the duplication. Some high level decisions, however, may eclipse the need for lower level decisions.

For example, in the two functionally equivalent procedures, COUNTRY-BUSINESS-PROCEDURE and BUSINESS-COUNTRY-PROCEDURE, shown in Listings 3 and 4, respectively, control variables COUNTRY and BUSINESS are examined to determine appropriate processing. The former examines the value of COUNTRY first, followed by business at the next level. The latter examines values in the opposite order. In both procedures, there are two essential levels in the control hierarchy, one to examine the value of COUNTRY and the other for BUSINESS. In this hypothetical company, CanAm International, the Canadian branch handles banking and telecommunications businesses, while the American branch is also involved in the insurance business.

```
COUNTRY-BUSINESS-PROCEDURE.
     IF COUNTRY = 'CAN'
        IF BUSINESS = 'BANKING'
           PERFORM CAN-BANKING
        ELSE
*          BUSINESS IS 'TELECOM'
           PERFORM CAN-TELECOM
*          NOTE: NO NEED TO CHECK FOR 'INSURANCE'
HERE
        END-IF
     ELSE
*       COUNTRY IS 'USA'
        IF BUSINESS = 'BANKING'
           PERFORM USA-BANKING
        ELSE
           IF BUSINESS = 'INSURANCE'
              PERFORM USA-INSURANCE
           ELSE
*             BUSINESS IS 'TELECOM'
              PERFORM USA-TELECOM
           END-IF
        END-IF
     END-IF.
```

Listing 3 *Country followed by Business*

```
 BUSINESS-COUNTRY-PROCEDURE.
     IF BUSINESS = 'BANKING'
        IF COUNTRY = 'CAN'
           PERFORM CAN-BANKING
        ELSE
*          COUNTRY IS 'USA'
           PERFORM USA-BANKING
        END-IF
     ELSE
        IF BUSINESS = 'INSURANCE'
*          NOTE: COUNTRY IS 'USA' (IMPLIED)
           PERFORM USA-INSURANCE
        ELSE
*          BUSINESS IS 'TELECOM'
           IF COUNTRY = 'CAN'
              PERFORM CAN-TELECOM
           ELSE
*             COUNTRY = 'USA'
              PERFORM USA-TELECOM
           END-IF
        END-IF
     END-IF.
```

Listing 4 *Business followed by Country*

The code for Listings 3 and 4 has been simplified with the assumption that the combination of country and business code values has already been validated.

In COUNTRY-BUSINESS-PROCEDURE, there is one IF statement to check the country code, while in BUSINESS-COUNTRY-PROCEDURE, at a lower level in the decision tree, there are two. Similarly, in the former procedure there are three IF statements required for business code, while the latter procedure contains only two at a higher level in the decision tree. Since there are more nodes of the control hierarchy involved at lower levels of nesting than at higher levels, more IF statements are required at lower levels to check conditions.

The hierarchy of control is perhaps more clearly demonstrated by logically equivalent code using COBOL II EVALUATE/WHEN statements, as shown in Listings 5 and 6.

```
 COUNTRY-BUSINESS-PROCEDURE.
     EVALUATE COUNTRY
     WHEN 'CAN'
        EVALUATE BUSINESS
        WHEN 'BANKING'
            PERFORM CAN-BANKING
*       NOTE: NO NEED TO CHECK FOR 'INSURANCE' HERE
        WHEN 'TELECOM'
            PERFORM CAN-TELECOM
        END-EVALUATE
     WHEN 'USA'
        EVALUATE BUSINESS
        WHEN 'BANKING'
            PERFORM USA-BANKING
        WHEN 'INSURANCE'
            PERFORM USA-INSURANCE
        WHEN 'TELECOM'
            PERFORM USA-TELECOM
        END-EVALUATE
     END-EVALUATE.
```

Listing 5 *Country followed by Business*

```
 BUSINESS-COUNTRY-PROCEDURE.
     EVALUATE BUSINESS
     WHEN 'BANKING'
        EVALUATE COUNTRY
        WHEN 'CAN'
            PERFORM CAN-BANKING
        WHEN 'USA'
            PERFORM USA-BANKING
        END-EVALUATE
     WHEN 'INSURANCE'
*           NOTE: COUNTRY IS 'USA' (IMPLIED)
            PERFORM USA-INSURANCE
     WHEN 'TELECOM'
        EVALUATE COUNTRY
        WHEN 'CAN'
            PERFORM CAN-TELECOM
        WHEN 'USA'
            PERFORM USA-TELECOM
        END-EVALUATE
     END-EVALUATE.
```

Listing 6 *Business followed by Country*

Implied Conditions

Note that in COUNTRY-BUSINESS-PROCEDURE, since the Canadian branch has no insurance functions, it is not necessary to check for insurance business once Canada has been identified as the country. Nor is it necessary in BUSINESS-COUNTRY-PROCEDURE to check for country once insurance has been identified as the business. We could, of course, include these checks anyway, but they would be redundant. They are implied, so they are not normally coded in the procedure. This is quite reasonable from a computer processing perspective, but it is humans who will have to maintain the program when changes to country and business processing are required. Changes to such implied conditions impose complications on the maintenance process. Implicit conditions must first be recognized (and made explicit) before they can be modified. In the context of any realistic, non-trivial piece of application logic, the equivalence of these two variations of procedures is not immediately obvious to a maintenance programmer. The procedures are not normalized (see Chapter 8 - Documentation). Instead, they have the form of pruned tree structures.

According to some established complexity metrics, the presence of implied conditions in a program may be associated with a lower measure of technical complexity. Ironically, this actually reflects a higher degree of comprehension complexity. It is important to remember that such statistics, although valuable, do not necessarily shed light on required maintenance efforts.

Normalized Logic Structures

Normalization is the formal refinement of complex, essentially freeform structures into simpler, standardized structures. The normalization of data structures has long been the subject of study and debate in database management circles. On the other hand, there is relatively less awareness of issues relating to the normalization of logic structures.

In a normalized logic structure, there are no implied conditions; each rule is explicitly and completely specified. The normalization process becomes primarily a matter of relating complete sets of conditions with corresponding sets of actions or functions to be performed. The condition sets are complete in the sense that all conditions are explicitly represented along every path in the tree. Still, due to the textual nature of procedural logic, even well written, normalized code may not be recognized as such without close scrutiny. A table driven approach replaces textual decision trees with graphic tables where all conditions are explicitly represented by a series of rows or columns.

Decision Tables

A decision table defines a process as an ordered set of conditions and their related actions. Tabular approaches to problem solving were developed in the 1960's by General Electric and other organizations. Official decision table components and terminology were standardized by the cooperative efforts of a number of organizations, including the Conference on Data Systems Languages (CODASYL) in 1962 and the Los Angeles chapter of the ACM (1965). The basic format of decision tables remains largely unchanged today. Decision table fundamentals are straightforward. Figure 9 illustrates the four quadrants of a decision table. Condition and action stubs perform a documentary function, while condition and action entries, or values, define the actual control flow for processing rules.

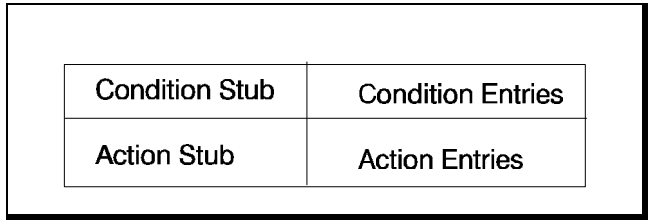


Figure 9 *Decision Table Quadrants*

Limited Entry Decision Tables

Decision tables are graphical representations of logic tree structures, where branching paths and processes are identified by sets of conditions and associated actions. Limited Entry decision tables describe binary logic trees. Condition symbols are limited to the binary values "Y" and "N"; action entries are either "X" or blank. The sample decision table shown in Figure 10 uses these conventions.

	Decision Rules							
Conditions	1	2	3	4	5	6	7	8
Over Credit Limit	N	N	N	N	Y	Y	Y	Y
Times Notified < and Customer History > 2	N	N	Y	Y	N	N	Y	Y
Invoices Older Than 60 Days	N	Y	N	Y	N	Y	N	Y
Actions								
Process Order	X		X					
Refuse Credit						X		X
Refer to Credit Manager		X		X	X			X

Figure 10 *Decision Table Example*

Extended Entry Decision Tables

In extended entry decision tables, condition and action entries are not limited to binary values. They may have a range of valid values. For example, the condition "marital status" may take the values M (married), S (single), D (divorced) or W (widowed). Different conditions may have different numbers of valid values, representing a variable, multi-branch logic structure.

It can be shown that any multi-branch logic tree is equivalent to some binary tree. Theoretically then, any extended entry decision table can be replaced by an equivalent limited entry table, but it would necessarily imply a greater number of decision rules.

Programming with Decision Tables

The original graphical conventions adopted for decision tables display all conditions and actions of a single business rule as one column in the table. To adapt these conventions to tables of a relational model, our decision tables will be rotated so that individual rules appear as rows in the table rather than columns. Taken as a group, the various condition columns become the table key. A given set of condition values represent the state of the application at some point during processing, relative to the decision table. When the table is searched for a prevailing set of conditions, action entries associated with that key become the result fields of the table lookup operation, and determine what processing should occur next. Actions are inferred from conditions, indirectly through the table. See Figure 11 for a high level description of this inference process.

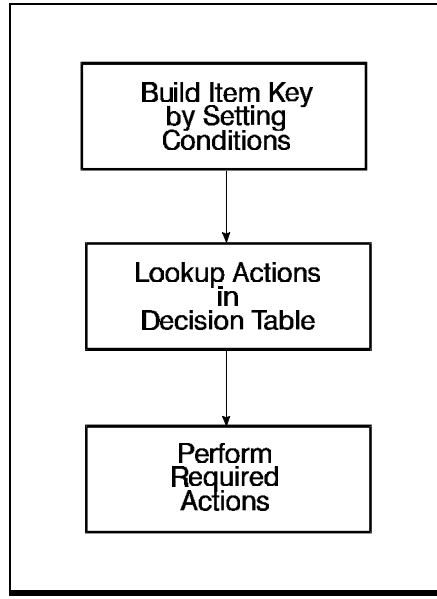


Figure 11 *The Table-Driven Approach Using Decision Tables*

For comparison purposes, examples of procedural COBOL code and equivalent table driven logic appear in Listings 7 and 8, respectively. Listing 7 illustrates a procedural implementation of the specifications for credit processing identified in the decision table of Figure 10. Note that the procedural code combines, or consolidates rules 1 and 3 into a single IF statement by eliminating the check for "times notified" and "customer history". Similarly, rules 2 and 4 are combined into one IF statement by eliminating the check for invoices older than 60 days. Both possible values for these conditions are implied, resulting in a procedural logic structure that is not normalized.

```
* DECISION RULES 1 AND 3

 IF CM-BAL-OWED NOT GREATER THAN CM-CREDIT-LIMIT
    AND CM-OVER-60-BAL-OWED EQUAL TO ZERO
     PERFORM 550-PREPARE-SHIPPING-ORDER.

* DECISION RULES 2 AND 4

 IF CM-BAL-OWED NOT GREATER THAN CM-CREDIT-LIMIT
    AND CM-OVER-60-BAL-OWED GREATER THAN ZERO
     PERFORM 570-PRINT-REFER-TO-MGR-LINE.

* DECISION RULE 5

 IF CM-BAL-OWED GREATER THAN CM-CREDIT-LIMIT
    AND (CM-TIMES-NOTIFIED GREATER THAN 3
         OR CM-CUST-YEARS NOT GREATER THAN 2)
    AND CM-OVER-60-BAL-OWED EQUAL TO ZERO
     PERFORM 570-PRINT-REFER-TO-MGR-LINE.

* DECISION RULE 6

 IF CM-BAL-OWED GREATER THAN CM-CREDIT-LIMIT
    AND (CM-TIMES-NOTIFIED GREATER THAN 3
         OR CM-CUST-YEARS NOT GREATER THAN 2)
    AND CM-OVER-60-BAL-OWED GREATER THAN ZERO
     PERFORM 560-PRINT-REFUSE-CREDIT-LINE.

* DECISION RULE 7

 IF CM-BAL-OWED GREATER THAN CM-CREDIT-LIMIT
    AND (CM-TIMES-NOTIFIED LESS THAN 4
         AND CM-CUST-YEARS GREATER THAN 2)
    AND CM-OVER-60-BAL-OWED EQUAL TO ZERO
     PERFORM 550-PREPARE-SHIPPING-ORDER.

* DECISION RULE 8

 IF CM-BAL-OWED GREATER THAN CM-CREDIT-LIMIT
    AND (CM-TIMES-NOTIFIED LESS THAN 4
         AND CM-CUST-YEARS-GREATER THAN 2)
    AND CM-OVER-60-BAL-OWED GREATER THAN ZERO
     PERFORM 570-PRINT-REFER-TO-MGR-LINE.
```

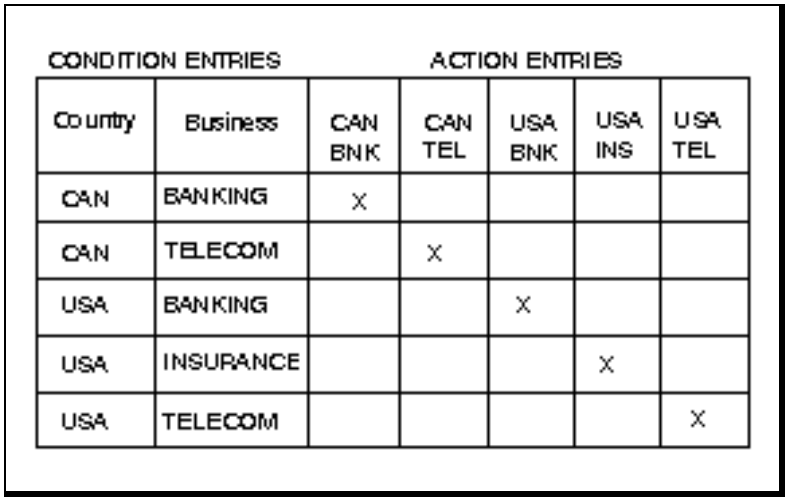
Listing 7 *Procedural Coding for Credit Processing Specifications*

The table driven example of Listing 8 retrieves its control flow information dynamically from the decision table shown in Figure 10. Again, the table driven code uses the table access routine TBCALL as an interface to a memory based table management system called tableBASE. Note that there are no implied conditions in this processing model.

```
 01  RULE-AREA.
     05  KEY-AREA.
         10 CONDITION1                    PIC X.
         10 CONDITION2                    PIC X.
         10 CONDITION3                    PIC X.
     05  RESULT-AREA.
         10 ACTION1                       PIC X.
         10 ACTION2                       PIC X.
         10 ACTION3                       PIC X.

*    SET CONDITIONS FOR SEARCH KEY

     IF CM-BAL-OWED GREATER THAN CM-CREDIT-LIMIT
         MOVE 'Y' TO CONDITION1
     ELSE
         MOVE 'N' TO CONDITION1.

     IF CM-TIMES-NOTIFIED GREATER THAN 3
        OR CM-CUST-YEARS NOT GREATER THAN 2
         MOVE 'Y' TO CONDITION2
     ELSE
         MOVE 'N' TO CONDITION2.

     IF CM-OVER-60-BAL-OWED GREATER THAN ZERO
         MOVE 'Y' TO CONDITION3
     ELSE
         MOVE 'N' TO CONDITION3.

*    FETCH DECISION RULE BY KEY

     MOVE 'FK' TO COMMAND-ID OF COMMAND-AREA.
     CALL 'TBCALL' USING COMMAND-AREA
                         RULE-AREA.

*    PERFORM APPROPRIATE ACTIONS

     IF ACTION1 = 'X'
         PERFORM 550-PREPARE-SHIPPING-ORDER.
     IF ACTION2 = 'X'
         PERFORM 560-PRINT-REFUSE-CREDIT-LINE.
     IF ACTION3 = 'X'
         PERFORM 570-PRINT-REFER-TO-MGR-LINE.
```

Listing 8 *Table Driven Code for Credit Processing Decision Table*

Consider a decision table designed to accommodate the requirements of our hypothetical CanAm corporation. Country and Business may be related indirectly to the routines to be executed through the Extended Entry Decision Table in Figure 12.



CONDITION ENTRIES		ACTION ENTRIES				
Country	Business	CAN BNK	CAN TEL	USA BNK	USA INS	USA TEL
CAN	BANKING	X				
CAN	TELECOM		X			
USA	BANKING			X		
USA	INSURANCE				X	
USA	TELECOM					X

Figure 12 *EEDT for CanAm Processing*

The program code, driven by this table, appears in Listing 9.

```
DECISION-TABLE-PROCESSOR.
    MOVE COUNTRY  TO TABLE-SEARCH-KEY-PART1.
    MOVE BUSINESS TO TABLE-SEARCH-KEY-PART2.
    PERFORM TABLE-LOOKUP.

    IF CAN-BNK = 'X'
        PERFORM CAN-BANKING.
    IF CAN-TEL = 'X'
        PERFORM CAN-TELECOM.
    IF USA-BNK = 'X'
        PERFORM USA-BANKING.
    IF USA-INS = 'X'
        PERFORM USA-INSURANCE.
    IF USA-TEL = 'X'
        PERFORM USA-TELECOM.
```

Listing 9 *Table Driven CanAm Processing - Version 1*

In many applications, a recurring theme of maintenance involves executing the same actions under new combinations of conditions. This kind of maintenance can often be addressed simply by updating the decision table to reflect the new set of conditions and associated actions. The decision table processor code does not have to be modified. This is possible because the conditions and actions of the application are related **indirectly** through the table, rather than **directly** through hard coded procedural logic. If the end user, who is most familiar with the business problem, is empowered to change the decision table without MIS intervention, turnaround time for maintenance requests can be dramatically reduced. Today's end users are frequently limited to

modifying primary data (that is, data which is processed). This direct user control of primary data is accepted in a production environment, within the confines of security and data validation controls. The growing maintenance backlog provides good reason to extend this concept to include the modification of production processes themselves, by qualified end users, through controlled access to decision tables.

Variations in the structure of decision tables are common. Country and Business may also be related to executable routines by the Extended Entry Decision Table shown in Figure 13, where binary action switches have been replaced by multi-valued action codes.

Country	Business	Action Routine
CAN	BANKING	CB
CAN	TELECOM	CT
USA	BANKING	UB
USA	INSURANCE	UI
USA	TELECOM	UT

Figure 13 *EEDT for CanAm Processing Using Action Codes*

The program code, driven by this table, appears in Listing 10.

```
DECISION-TABLE2-PROCESSOR.
    MOVE COUNTRY  TO TABLE-SEARCH-KEY-PART1.
    MOVE BUSINESS TO TABLE-SEARCH-KEY-PART2.
    PERFORM TABLE-LOOKUP.

    EVALUATE ACTION-ROUTINE
    WHEN 'CB'
       PERFORM CAN-BANKING
    WHEN 'CT'
       PERFORM CAN-TELECOM
    WHEN 'UB'
       PERFORM USA-BANKING
    WHEN 'UI'
       PERFORM USA-INSURANCE
    WHEN 'UT'
       PERFORM USA-TELECOM
    END-EVALUATE.
```

Listing 10 *Table Driven CanAm Processing - Version 2*

This kind of table structure has even more advantages for future maintenance concerns. For example, what happens if new functionality for Canadian insurance processing is incorporated into the application at some future date? Certainly, the decision table would have to be updated to include a new rule specifying the conditions CAN and INSURANCE, with a new "CI" action code. There are two ways to add functionality to the program. One involves inserting a new WHEN 'CI' clause to the EVALUATE statement, along with logic for the new paragraph to be performed. A more general method resorts to dynamically loading the routine to be executed if the action code in the table is not recognized among the set of nucleus routines. In the version of the EVALUATE statement in Listing 11, it is assumed that appropriate error checking is performed to ensure that the routine exists on the program library. Preprocessing may be required to derive the program module name from the action code on the table, or the name may be taken directly from the table.

```
    EVALUATE ACTION-ROUTINE
    WHEN 'CB'
       PERFORM CAN-BANKING
    WHEN 'CT'
       PERFORM CAN-TELECOM
    WHEN 'UB'
       PERFORM USA-BANKING
    WHEN 'UI'
      PERFORM USA-INSURANCE
    WHEN 'UT'
       PERFORM USA-TELECOM
    WHEN OTHER
       PERFORM DYNAMIC-LOAD-AND-EXECUTE
    END-EVALUATE.
```

Listing 11 *CanAm Processing EVALUATE Statement*

In fact, none of the action routines need to be coded in the nucleus of the application; they may all be dynamically loaded the first time they are referenced. If the codes for "action routine" correspond to the names of executable modules, then the revised decision processor is streamlined to a minimal maintenance form, as in Listing 12.

```
DECISION-TABLE2-PROCESSOR.
    MOVE COUNTRY  TO TABLE-SEARCH-KEY-PART1.
    MOVE BUSINESS TO TABLE-SEARCH-KEY-PART2.
    PERFORM TABLE-LOOKUP.
    PERFORM DYNAMIC-LOAD-AND-EXECUTE.
```

Listing 12 *Table Driven CanAm Processing - Version 3*

In this approach, condition sets and action sets have been completely decoupled from the remaining procedural code. They are associated only through the external table.

The entire decision control search sequence has been removed from the procedural logic, and is now governed by the sequence of rows in the table. The control structure may be completely resequenced by sorting the table according to some other desired set of condition columns, with no change to program code. In the preceding discussion, for example, our decision table was sorted by Country, followed by Business. It could just as easily have been sorted by Business, then Country. The program code is independent and unaffected.

Consolidated Decision Tables

There are two ways in which decision tables may be consolidated.

1. Two or more tables with different sets of conditions may be combined into a single decision table. This will be referred to as "table consolidation".

2. Two or more rules in a single table may be combined to create a new rule with fewer conditions specified, which covers all of the original conditions. To distinguish from 1 above, this will be called "rule consolidation".

Both consolidation processes are formalized according to the principles of Boolean algebra or equivalent set theory. Harvey Cauthen at Prudential Insurance has described, in detail, the implications of table consolidation for application systems development. Rule consolidation has been discussed in various texts on decision tables, and also in related discussions on minimizing logic designs in switching circuit theory.

There are many variations of the rule consolidation process, but the objective in all cases is to reduce the number of conditions (and thus the number of rules) which must be considered in order to execute a particular action. For the purposes of illustration, we will consider a simplified version, which takes the unconsolidated table shown in Figure 14 through intermediate stages (Figures 15 and 16) to the final form in Figure 17. The basic process can be described as follows:

IF two rules agree in all conditions but one
AND differing entries are "Y" and "N"
AND actions for the two rules are identical

THEN both rules may be consolidated into a single rule by
replacing the differing entries by a "don't-care" symbol
(usually represented by "-").

In this simplified version of the consolidation algorithm, the single new consolidated rule then replaces both contributing rules, and consolidation continues with the remaining rules. Multiple passes through the table act on other conditions to reduce the

number of rules until no further consolidations are possible.

	Decision Rules																													
Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Order & Amount < Credit Limit	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Manual Credit OK	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Special Customer	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Order Size < Shipping Min	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Manual Shipping OK	Y	N	Y	N	Y	Y	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Actions																																
Send Order to Credit Dept.																													X	X	X	X
Send Order to Shipping Mgr.		X				X				X				X				X				X				X						
Ship Order	X		X	X	X		X	X	X		X	X	X		X	X	X		X	X	X		X	X	X		X	X				

5 Conditions ⟶ 2^5 Combinations

Figure 14 *Unconsolidated Table*

Conditions	Decision Rules																
	1 & 3	2 & 6	4 & 8	5 & 7	9 & 11	10 & 14	12 & 16	13 & 15	17 & 19	18 & 22	20 & 24	21 & 23	25 & 27	26	28	29 & 30	31 & 32
Order & Amount ≤ Credit Limit	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N
Manual Credit OK	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N	N
Special Customer	Y	-	-	N	Y	-	-	N	Y	-	-	N	Y	Y	Y	N	N
Order Size < Shipping Min	-	Y	N	-	-	Y	N	-	-	Y	N	-	-	Y	N	Y	
Manual Shipping OK	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	-	-
New Rule Number	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17
Actions																	
Send Order to Credit Dept.																X	X
Send Order to Shipping Mgr.		X				X				X				X			
Ship Order	X		X	X	X		X	X	X		X	X	X		X		

Figure 15 *Rule Consolidation (Step 1)*

Conditions	Decision Rules										
	A1 & A4	A2 & A6	A3 & A7	A5 & A8	A9 & A12	A10	A11	A13	A14	A15	A16 & A17
Order & Amount ≤ Credit Limit	Y	Y	Y	Y	N	N	N	N	N	N	N
Manual Credit OK	Y	-	-	N	Y	Y	Y	N	N	N	N
Special Customer	-	-	-	-	-		-	Y	Y	Y	N
Order Size < Shipping Min	-	Y	N	-	-	Y	N	-	Y	N	-
Manual Shipping OK	Y	N	N	Y	Y	N	N	Y	N	N	-
New Rule Number	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
Actions											
Send Order to Credit Dept.											X
Send Order to Shipping Mgr.		X				X			X		
Ship Order	X		X	X	X		X	X		X	

Figure 16 *Rule Consolidation (Step 2)*

Conditions	Decision Rules									
	B1 & B4	B2	B3	B5	B6	B7	B8	B9	B10	B11
Order & Amount ≤ CreditLimit	Y	Y	Y	N	N		N	N	N	N
Manual Credit OK	-	-	-	Y	Y	Y	N	N	N	N
Special Customer	-	-	-	-	-	-	Y	Y	Y	N
Order Size < Shipping Min	-	Y	N	-	Y	N	-	Y	N	-
Manual Shipping OK	Y	N	N	Y	N	N	Y	N	N	-
New Rule Number	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Actions										
Send Order to Credit Dept.										X
Send Order to Shipping Mgr		X			X			X		
Ship Order	X		X	X		X	X		X	

Figure 17 *Rule Consolidation (Step 3)*

Note that, unlike the procedural consolidation of rules in Listing 7, this consolidation process does not eliminate conditions from the rule. They are still represented, but marked explicitly as immaterial to evaluation of the rule.

More sophisticated, formal algorithms for rule consolidation, such as that developed by Quine and McCluskey, have been applied extensively in switching theory to simplify logic circuitry. These algorithms guarantee a minimal rule set.

Consolidation of an extended entry table is more complex, but still possible. The basic consolidation process is modified to:

IF a set of n rules agree in all conditions but one
AND all (n) valid values for the condition are included
AND actions for all rules are identical

THEN all rules may be consolidated into a single rule by replacing the differing entries by a "don't-care" symbol (usually represented by "-").

Organization of Consolidated Decision Tables

The problem of organizing and searching a consolidated decision table, with some notable exceptions, has received surprisingly little attention among developers of mainstream business applications. The importance of consolidated tables should not be understated. They can help to define and implement any process which deals with complex sets of conditions which must be generalized to a manageable level. Insurance companies, for example, must determine rates to be charged depending on a range of criteria whose numbers are combinatorial in nature. No procedural approach can effectively code for all possible conditions, and an overly simplistic table driven approach results in rate tables which are unnecessarily large and cumbersome to manage.

The difficulty with using consolidated tables centers on searching for key fields which match a value in the table exactly or which match a don't-care value in the table. There are several approaches to this problem, some of which are more elegant and/or efficient than others. The combination of table organizations and search techniques used range from modified serial searches, matching on all conditions at once, to those involving multiple levels of indexing and condition-by-condition comparisons.

In all cases, as unique codes for consolidated fields, don't-cares must be recognized as inherently special conditions. They represent a number of specific detail key values. One consolidated rule represents many unconsolidated rules. All other factors being equal, rules with a high degree of consolidation are statistically more likely to match a given search key than other less consolidated rules. For this reason, techniques have

been developed to sort consolidated rules which cover the most detail rules so that they appear closer to the beginning of the table. Since the sort order is determined after the consolidation process is accomplished, table management capabilities should allow for sorting on non-contiguous keys to accommodate any grouping of sort fields.

In order to effectively sort and search a consolidated decision table, special statistics and flags must be maintained for each rule, in addition to the condition and action columns of the original unconsolidated table. For example, the number of detail rules covered by each consolidated rule should be recorded and don't-care fields should be identified for efficient comparison against a given search key (see Chapter 5 - Searching Consolidated Decision Tables).

When dealing with extended entry decision tables, associated tables of valid values for condition fields are useful for a variety of reasons. They allow for the maintenance of a checklist of values for field consolidation, completeness testing, table update validation or search key validation

Searching Consolidated Decision Tables

In a typical decision cycle, values are established for a set of conditions. That set of values (the search key) is then used to search a decision table for a matching rule. The table key is the corresponding set of conditions for a given rule in the table.

As stated previously, there are several approaches to the problem of searching a consolidated decision table, some of which are more elegant and/or efficient than others. A particular search key field can match a value in the table key exactly or it can match a don't-care value in the table. Either possibility must be accounted for. If don't-cares are explicitly identified in each item, then a minor modification to a serial search can attempt to find a match on the entire search key in a simple, yet relatively elegant algorithm, operating within reasonable performance limits.

The following discussion assumes that don't-cares in the table keys are represented by high values (binary "1"s). Also, there exists, for each row of the table, a set of "don't-care mask" fields, which correspond in structure and length to the set of condition fields in the rule (ie. the table key fields). This set of mask fields, created during the consolidation process, may be physically part of the consolidated rule, or it may appear in a separate table. Each of these mask fields contains high values if the corresponding condition field is a don't-care, otherwise it contains binary zeros.

As with a standard serial search, the search process begins with the first row of the table. Unlike a standard serial search, however, instead of comparing the search key directly against the table key for equality, the search key is first modified by performing

a bitstring Boolean **OR** of the search key and the don't-care identifiers for the rule. In this way, don't care fields are selectively set to high values in the search key. The result is then compared for equality against the table key. This will guarantee a match on all don't-care fields in the table key, as well as other unconsolidated field values.

If a match is not found for the current row in the table, the next row is considered. If the consolidated table is complete and the search key is valid, then a match must ultimately be found in the table. Figure 18 illustrates the difference between a standard serial search and the search process for a consolidated decision table.

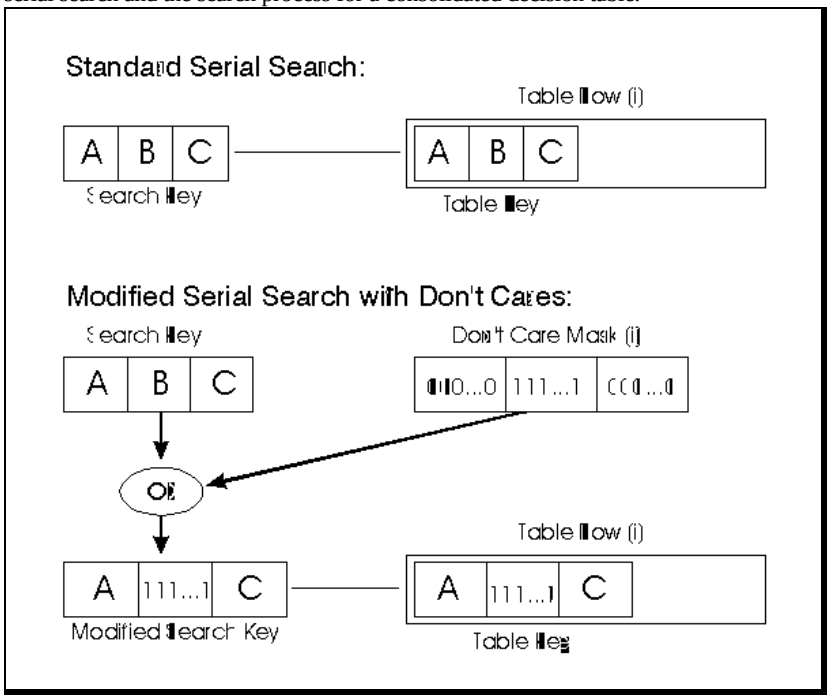


Figure 18 *Searching a Consolidated Decision Table*

Functional Decomposition

Normalization of logic paths is only meaningful with respect to a particular level of detail. High level actions may themselves be further refined as a series of lower level actions executed under appropriate sets of conditions. This functional decomposition of high level actions into lower level actions and conditions could conceivably continue until atomic actions are found which contain no additional decision points. The end result of this decomposition could be a single all-encompassing decision table for the application. In practice, however, it is often easier to analyze and model the problem in

discrete chunks, on a top-down basis. This results in a collection of smaller decision tables, related in a hierarchical manner by connecting procedural logic flows.

In a re-engineering situation, functions or actions may be isolated and extracted from existing procedural code. In this case, an action may be defined as all logic along the path between any two decision points, such that the first decision point marks the only entry to the action and the second marks the only exit from the action.

Context constrains the extent to which actions may be shared and reused. Context is defined by the associated set of conditions under which an action is executed, and the effect those conditions have on the action itself. It is a measure of the appropriateness of an action. Atomic actions are free of decision sub-trees. They may be considered context independent if their behaviour is not otherwise affected by external control variables. Atomic actions are easily shared and reused within an application or across a family of related applications. There is no need to match prerequisite (shared) conditions in the target environment where sharing is to occur, because no such shared conditions exist. The same is true for any higher level, well structured routine which is self-contained. All decision points are dependent only on local control variables.

Sparse Decision Tables

Many business problems involve extremely large sets of potential conditions which might conceivably be encountered during processing but, in practice, only a small subset of these conditions normally surface in day-to-day operations. If there are a million possible logic paths through a problem space, but less than 100 have proven to be relevant to past business, then it would be a futile exercise to code the application to handle all possible paths. The best procedural approach to this situation would allow for only the 100 currently recognized useful paths, and handle all other conditions with an appropriate error message. Unfortunately, as the business evolves, new combinations of conditions are encountered, and the number of these active logic paths increases. The application must then be enhanced, presenting all the attendant problems of traditional maintenance. Even worse, the overall quality of the application is not appreciably improved by the maintenance effort. In the example above, after one new logic path is included, we now have 101 paths accounted for out of a possible one million. The problem here is the law of diminishing returns. Once the application includes all the most common business rules, considerable maintenance effort must be expended for marginal additional improvements. If the business evolves such that new paths become relevant at weekly or daily rates, the maintenance demand quickly outstrips the ability of MIS to keep pace.

The table driven approach to this problem makes use of an incomplete decision table. In effect, there are many more rules missing from the table than there are rules present. The number of conditions accounted for is said to be sparse. Such sparse tables are extreme examples of incomplete decision tables, and have proven to be valuable problem solving mechanisms.

Not Found Conditions

In processing any incomplete decision table, what happens when the search key, or the set of prevailing conditions, is not found in the table? Some default action must be executed, including an appropriate warning or error message. The conditions absent from the table may well prove to be valid operating conditions which have thus far not been included in the application specifications. Alternatively, they may represent another invalid state to be handled by existing error routines. These new conditions can usually be resolved by the end user, by associating them with a series of actions to be performed - actions selected from an existing list which defines the scope of the application itself. This kind of regular, controlled exception processing is the key to continued iterative development of the application.

In this way, the system acquires new knowledge (rules) through training cycles with an expert user. The potential exists, for some applications, to improve upon these user-assisted training cycles with the implementation of more automatic machine learning algorithms. For example, a set of conditions not found in the table may be transformed in some way to match a similar set of conditions which has been encountered in the past and which is represented in the table. A new rule might then be inserted into the table automatically, coupling the new conditions with the inferred set of actions.

Initial Load of Sparse Decision Tables

End user involvement in relating conditions and associated actions begins with the design stage and continues throughout the life of the application. Most relationships between conditions and actions are usually specified early in the design stage, but this critical step may sometimes be deferred until decision table load time. Consider the case where all condition columns and all action columns have been clearly identified in the decision table, and the table processor displays exception messages for application states not found in the table. At this point no rules have yet been entered into the table to associate particular actions with particular conditions. In complex applications, where sparse decision tables are required, it may be unclear, even to an expert user, exactly which sets of conditions should be analyzed and included in the table to begin with. To resolve this problem, an initial batch test run of the application could be made against an empty decision table, using copies of live master and transaction data. The resulting exception list would then identify all prevailing application states which were used for attempted searches in the decision table. With this list in hand, it is a relatively easy matter for an expert user to relate each set of conditions to an appropriate set of actions. The decision table would then be loaded with this information, and testing would continue for verification of the action components.

Subsequent testing should use live transaction and master data from several cycles of processing to ensure that a representative sample of the most common sets of prevailing application conditions are included in the decision table. The first production version of

the application is accepted to be a true prototype. The process of further development and refinement of the initial prototype becomes, in essence, a process of continued updating of the sparse table. For practical purposes, there will never be a final production version. And there is no pretense of creating one.

6
Software Compatibility Issues

Applications developed according to the principles of Table Driven Design are compatible with other software in ways which are both consistent and complementary.

Table Driven Design concepts apply to any programming language. Limitations of any particular language can be overcome by vendor supplied extensions. Some languages, for example, offer only limited support for dynamic allocation of memory, a requirement for table space expansion at application run time. Other languages and software products offer extended features, including the ability to allocate, search and manage tables very effectively in memory.

The following sections discuss a wide range of requirements for table driven applications, with respect to functions provided by table management systems, CASE products and database management systems.

Table Operators and Support Facilities

Much can be accomplished with the use of working storage arrays as a buffer for table data loaded from an external file. No additional tools are required beyond the programming language constructs already available to the average shop, in order to begin deriving benefits from Table Driven Design. Operations on main memory data tables are restricted by the limited memory management capabilities of today's common programming languages. Tables, for example, frequently overflow beyond the limits of static allocations. Inherited from an earlier era, these languages do not allow for the sophisticated manipulation of table data required for optimal table driven architectures.

Memory Resident Rules Support

Ideally, operators for tables and individual rules should be designed to take maximum advantage of the fact that the table is entirely resident in main memory, including:

- Automatic table load/unload facilities
- Efficient main memory data organizations

- A variety of high performance search methods
- Dynamic run time expansion of allocated table space
- Dynamic run time reorganization of tables
- High performance index structures
- Dynamic run time index creation and modification
- Dynamic run time alternate views

Table access should be provided by key and by subscript. The access performance for a table driven application should be in the same order of magnitude as that achieved by compiled procedural code. Dynamic run time reorganization of tables may be required due to modified table structures such as modified keys, reordered columns, update processing, including inserts, deletes, moves, or table space expansion due to overflow.

A menu driven development environment should be available to define and manage table structures consistently across all applications and platforms. Convenient access to table templates may be required, for subsequent customization by an application.

Most of the above support capabilities have never been included in file access methods or database management systems because those systems are primarily DASD oriented. Table driven rules support must be main memory oriented (see Chapter 3 - Classes of Tables).

Other useful support facilities include:

- The compromise ability, under some circumstances, to treat DASD files like memory resident tables, in a manner similar to buffered files being treated like memory resident tables.
- Shared access to memory resident tables across all mainframe address spaces, both online and batch
- Shared access to tables across all platforms in a client/server architecture
- Library support for permanent storage of table data on DASD

tableBASE

Data Kinetics' tableBASE is a memory based data management system specifically designed for memory resident tables. An Application Program Interface (API) allows for high performance access to tables in memory, external to the application code. Historically, tableBASE has evolved from an optimal data access engine into a complete infrastructure for the management of business rules in table driven applications.

An online menu driven system called tablesONLINE provides a consistent user interface for controlled access to tables, table definitions and table utilities, independent of customer written application programs.

CASE Products

Computer Assisted Software Engineering (CASE) is designed to improve programmer productivity in application development and maintenance. CASE is supported by a repository of information about system entities and their interrelationships. Code generating CASE products may select, organize, tailor and integrate source modules to automate much of the coding process. The resulting application source is then compiled and linked to create executable modules.

Application structure and parameters are set, or generated, by such CASE products before the "binding point", at compile time. Compositional mechanisms facilitate reuse of existing application components. Generative mechanisms are used when needed components are not already available.

Parameterized mechanisms are both compositional and generative. In using a parameterized software development CASE tool, an engineer specifies the functionality, behaviour and constraints on a system, by filling out a form or table. The recorded values drive the CASE tool in the creation of the specified component out of pre-existing software and software templates or frames. Parameter driven application generators are useful for domains whose variability is well understood in advance. For domains whose variability is less well understood, other, still more flexible mechanisms are needed.

Other CASE products support the management and re-engineering of application systems by analyzing source code to produce flow diagrams, statistical reports and information for the repository. Software is available to automate the isolation and extraction of business rules from legacy applications. This may be done either statically by scanning source code or dynamically by tracing execution paths based on sets of prevailing conditions. CASE reduces the inevitable maintenance effort by providing useful documentation and by minimizing the number of lines of source code required and managed for application development.

Table Driven Design objectives include generalizing modules to reduce the total volume of source and executable code, reusing executable modules, reusing tables and avoiding regeneration of executable modules whenever possible

Application logic flows and parameters are set, or determined, after the binding point, dynamically at execution time, based on access to external control data (in tables) which describes existing application constraints.

The maintenance effort is minimized by anticipating certain classes of changes, changes which may be accommodated through table updates, with no associated modification or regeneration of source code. Classes of change can be described in terms of the high level parameters which govern the current state and future evolution of particular, chaotic business systems. These parameters are implemented as memory resident tables for reference during system operation.

The major difference here is that CASE focuses on productivity prior to compile time, before the binding point, while the major benefits of Table Driven Design are manifested at execution time, after binding. This has important implications regarding the relative contributions of MIS personnel and end user groups to the change control process.

Figure 19 presents a graphical view of the relationship between MIS, a given application system and the end users of that system.

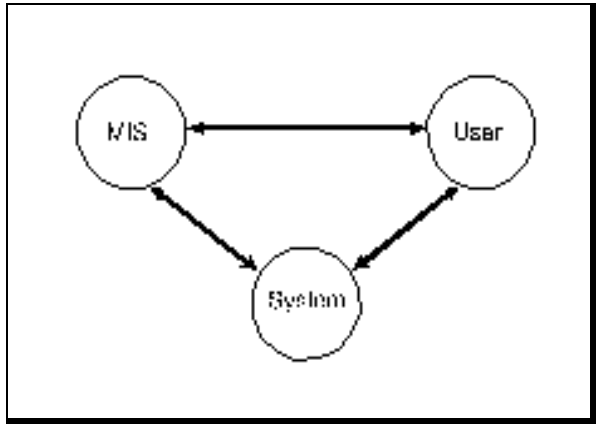


Figure 19 *Traditional Maintenance Relationships*

Figure 20 illustrates a closer, stronger link between MIS and the application system through the use of CASE facilities.

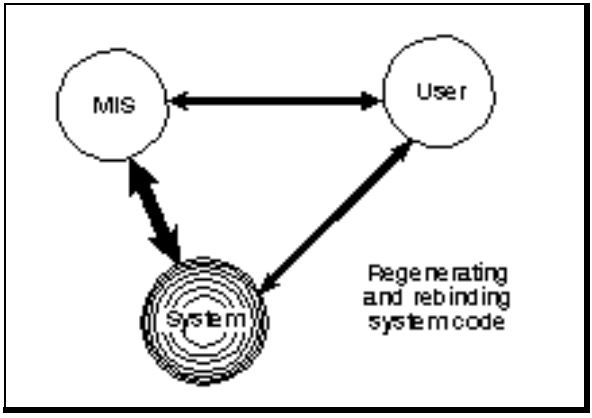


Figure 20 *CASE Accelerates Change Request Process*

In a table driven application, the user has greater control over the behaviour of the system through direct updates to system driving tables, as illustrated in Figure 21. As a privileged user of the application, the MIS organization itself derives similar benefits from a table driven approach.

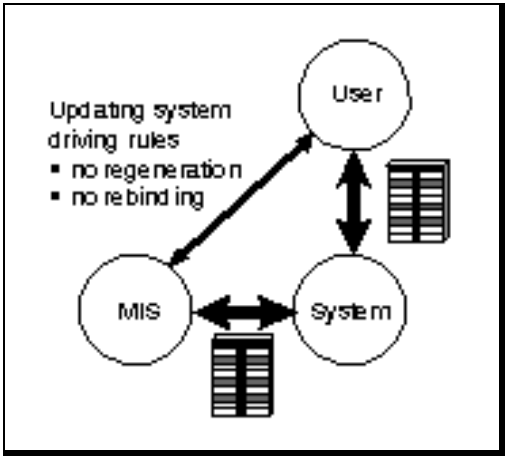


Figure 21 *Tables Accelerate Change Request Process*

Analytical statistics from CASE tools may be particularly useful in helping to identify problematic areas of legacy systems which may be effectively addressed by table driven approaches to re-engineering. CASE products which provide inventories of entities and

relationships contribute to the functional decomposition of legacy systems, an important step in re-engineering for Table Driven Design. Application driving tables may be considered the active part of a complete system repository. Clearly, the two approaches to application development and maintenance offer complementary benefits. As shown in Figure 22, hybrid Table Driven Designs incorporating CASE generated source code, together with execution time driving tables, offer the best of both worlds.

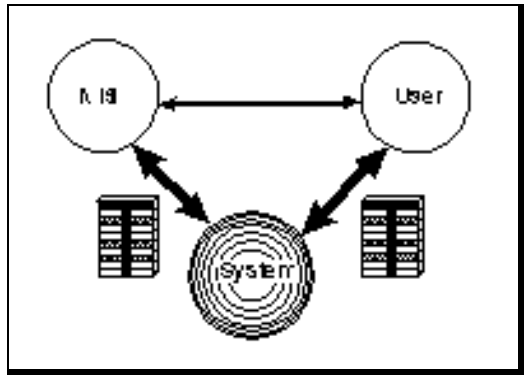


Figure 22 *CASE and Tables Accelerate Maintenance Cooperatively*

Database Management Systems

Some forward thinking DBMS vendors have recently added new capabilities to their products that will permit the loading of entire tables into memory. Effective main memory table management for rule based systems still remains outside the domain of contemporary database management systems. Control data for table driven applications has support requirements which are substantially different from those of a traditional DBMS. See Chapter 3 - General Table Classifications for a more complete discussion of the complementary nature of process related data and data which is processed.

Application Platforms

The concepts of Table Driven Design are widely applicable across a variety of software platforms, including batch and online environments, and client/server architectures.

Tables and programs developed for use in mainframe batch systems should be readily available for reuse in equivalent online applications.

Table driven applications on mainframe systems and PC workstations are capable of extensive interaction in an enterprise wide network for support of distributed application development and client/server architectures.

From the perspective of volumes alone, executable code is reduced considerably in table driven systems. This offers some immediate support for distributed applications, where program and data transfer operations are frequent, representing a potential bottleneck in the system.

Given that the reasoning for memory based, Table Driven Designs can minimize I/O in a mainframe environment, it can also minimize data transfer operations in a networked client/server architecture. From a performance perspective, the table access considerations of network client versus network server are similar to those of main memory versus DASD. Arguments in favor of local copies of tables closely parallel those for memory resident tables. The central problem remains a question of positioning the right data and processes at the right place, for the right reasons.

In client/server architectures, client specific subsets of application driving tables can be distributed to the respective client platforms for optimal access, while common, shared rules remain centralized in tables at the server location. For example, if local retail market descriptions were contained in local tables at all client sites, while centralized wholesaler information drives decisions in more global server processing, then related applications would be truly market driven. Since server delays, due to unnecessary I/O, have a negative impact on all clients, memory resident server tables help to optimize response time across the enterprise.

In addition, hierarchical client-server system architectures are ideal platforms for the development of resolution independent application architectures (see Chapter 7 - Resolution Independent Architectures).

7
Impact Areas

Table Driven Design issues relate directly to various groups, projects, applications and platforms within the MIS organization.

A number of stakeholder groups, including:

- Information Systems Senior Managers
- Project Managers
- Systems Analysts and Architects
- Applications Programmers
- Data Administrators, including Database Administrators
- End Users

must become acquainted, to one degree or another, with Table Driven Design concepts and success factors.

The following discussion of applications draws from live examples which have derived significant benefits from corporate Table Driven Design efforts.

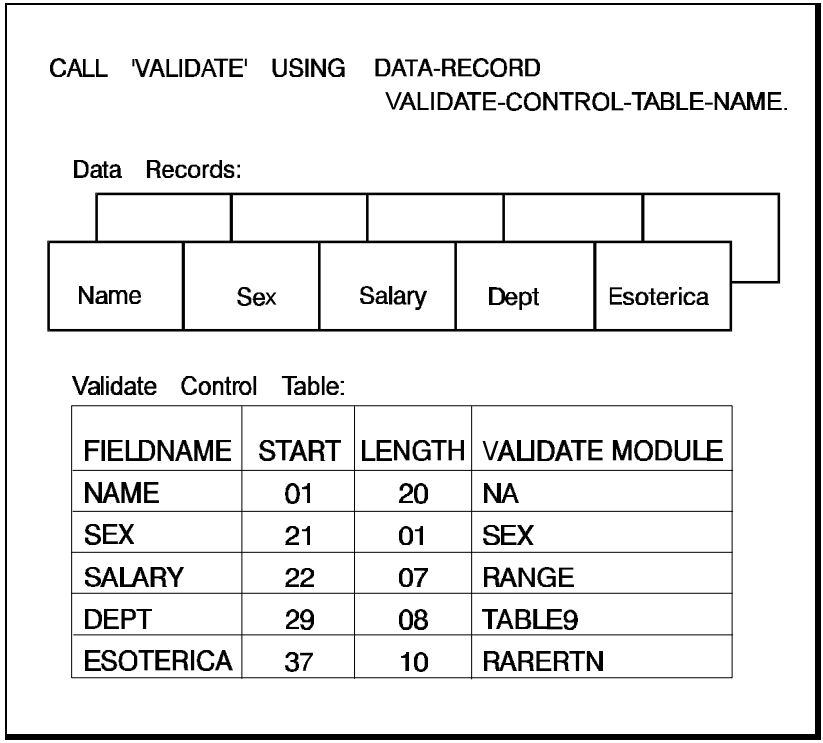
Optimal Table Driven Validation

According to current practice, there are as many validation programs in existence, at a given company, as there are record (or transaction) formats to be validated. This encourages widespread duplication of code and unproductive development and maintenance efforts. Virtually identical numeric edit checks, for example, appear in numerous programs. They differ only in the location of the field to be edited.

A table can readily describe the location and length of each field in a given record, along with the names of associated validation modules for each field. Such validation control tables could be entered into a library to describe all validation requirements for all record formats.

A single executable driver module could be shared by all corporate applications for the validation of fields in any record format. Field level validation modules qualify as atomic functions, easily shared and reused. For all possible types of validation, these

modules can be entered into a load library with no duplication of code or development effort. They may then be loaded in various combinations as required by the driver. Refer to the Universal Validation Program example in Figure 23.



FIELDNAME	START	LENGTH	VALIDATE MODULE
NAME	01	20	NA
SEX	21	01	SEX
SALARY	22	07	RANGE
DEPT	29	08	TABLE9
ESOTERICA	37	10	RARERTN

Figure 23 *The Universal Validate Program*

One validation driver program is capable of validating any record format for which a corresponding table exists. It is simply a matter of identifying which format is about to be validated by supplying the name of the appropriate validate control table in the call to the validate routine. As new record formats are added to the system, appropriate validation tables are added to the table library. If necessary, new field level validation routines are coded and added to the load library over time with a minimum of effort.

With additional analysis, this approach can be shown to be suitable for variable as well as fixed length record formats, and for cross field validation with other fields in the same record or in other records.

As an active component of table driven systems, the field validation information described in these tables should constitute part of the data dictionary. These structures

are ideal for implementation as memory resident tables, to be loaded from the corporate database (see Chapter 8 - The Data Dictionary).

Formatting Processes

Format control tables, similar to the validation control tables of Chapter 7, may be used for dynamically interpreting input files and dynamically formatting output files or reports. A simple format control table for dynamic report formatting is shown in Figure 24. Extensions to this basic theme are common, and may be used to handle literal values and a variety of formatting attributes. Two or more such tables may be used to derive reports from several input files, or to restructure fields from several input files into one or more output files.

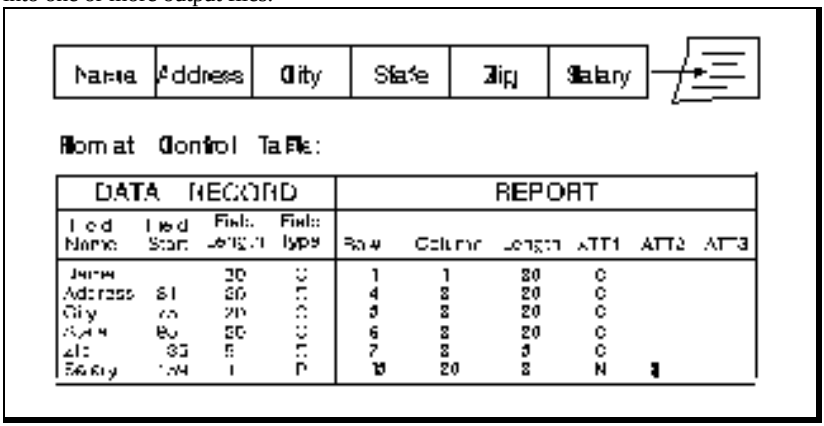


Figure 24 *Report Formatting Format Control Tables*

Following the trends in many industries, banks are implementing new ways of customizing and personalizing service to their clients. One approach adopted by a major bank uses dynamic report formatting to provide its customers with an integrated monthly statement. The statement combines all of the accounts for a customer into one smooth flowing report. Descriptive text is included for each account and for various combinations of accounts. Each customer is sent a single tailored document for all financial transactions, in the belief that this will encourage customers to use the bank as a single source for all financial accounts.

Integrating all of a customer's accounts also provides a basis for target marketing to specific customers. Targeting can be based on a specific profile of accounts, and can be additionally qualified by account balances, activity and demographic data. The integrated statement is a convenient and cost effective vehicle for inserting tailored marketing messages and brochures in customer mailings. Benefits to the bank include reduced paper and mailing costs, and a reduction in mass marketing costs by targeting promotions to those customers most likely to respond.

The statement is custom formatted to the specific, possibly unique, combination of accounts for each customer. The system uses a series of formatting tables and consolidation tables to build the integrated statement. It runs daily, processing 5 to 7 million transactions for an average of 100,000 master accounts.

As an active component of table driven systems, the formatting information described in these tables should constitute part of the data dictionary. Like the validation tables described earlier, these structures are ideal for implementation as memory resident tables, to be loaded from the corporate database (see Chapter 8 - The Data Dictionary).

Resolution Independent Architectures

In an article on chemical communication in the brain, titled "Brain By Design", neurologist Richard Restak comments on the symmetry apparent in nature.

"As successful principles evolve over millennia, nature employs them again and again at various levels of the organism, and so events at one level mirror what is going on several higher or lower orders away".

This symmetry is also apparent across various levels of detail in many components of information systems.

From one perspective, the level of granularity in the analysis of data and processes is related to the degree of normalization of data and logic structures. A granule is a data element describing an atomic object of the system. The degree of data normalization can be thought of as a kind of marker for the end of analysis. In their "DB2 Design And Development Guide", Wiorkowski and Kull state that, for a normalized data structure, at the lowest level of granularity, *"each attribute must be a fact about the key, the whole key and nothing but the key"*. Through ongoing system enhancements, today's granules evolve into tomorrow's assembly of progressively lower level, more detailed granules. Today's key grows tomorrow's pointers to lower level tables. A fundamental symmetry extends through all levels.

For some applications, symmetry provides another perspective on granularity which goes beyond the idea of normalization, a perspective based on chaos theory and recursion. In these applications, the level of granularity is interpreted as a measure of scale, resolution or refinement, of data descriptions and processing rules. As the resolution of these descriptions and rules increases, there is a corresponding measurable improvement in the performance, accuracy or effectiveness of the application. Qualitative improvements are derived from the refinement of data descriptions and processing rules, rather than from enhancements to application code.

This idea is significant in the light of Table Driven Design. In a table driven application, generic procedures can be applied at any given degree of resolution.

Processing is guided by rules which are associated with the current active environment, regardless of scale. Recursive procedures are, by their very nature, driven by tables as they process through successive levels of resolution. There are correlations between resolution, control breaks and table keys which facilitate the development of generic, adaptable systems.

Examples of such resolution based systems include Graphical Information Systems (GIS), generic summarization routines and spreadsheet applications.

Visually oriented Graphical Information Systems are entirely built around two or three dimensional graphical tables. They are driven by pixel descriptions and location specific processing rules. As the population of pixel descriptions and associated rules increases, so does the power of the system, and its value to the user. Various levels of granularity may be maintained simultaneously, through the use of different tables, according to the requirements of the application.

The same conclusions may be drawn for many regionally based types of processing, including marketing or census surveys and other statistical analyses. A major oil company, for example, makes use of a table driven recursive procedure - simple, concise and dynamic - to process legal land descriptions with successive levels of subdivisions for administration of oil exploration areas. Subdivision information has the same granular format as larger blocks of land. No matter how many levels of subdivision are involved, or how often the land base is restructured, no change is required for the application program code. The resolution is implicit in the tables of subdivision data.

Real Time Process Control

A real time process control system monitors a physical process by examining a collection of metered values and event indicators to determine the state of the process at any given moment. Based on this state, one or more actions will be performed. This is a classic application for decision tables. A table of recognized states drives the process to its desired conclusion.

Traditional Table Update

Due to steady growth in regional business, a large enterprise finds it necessary to periodically restructure a single district office into two new and distinct offices, with an appropriate division of territory between the two.

A recent enhancement to a customer application system, for one such District Office (DO) split, required that an additional entry be made in the DO table. In this particular application, the table implementation is tied to the program code in such a way that 74

program modules had to be changed, affecting approximately 250 COBOL paragraphs. Three weeks of programmer time were necessary to apply the changes. Forty-nine pages of specifications were required to describe the requirements for a single DO split. (A second change request arrived in the programmer's hands just prior to moving the first DO split into production, and added volume to the original document.) It's worth noting that this type of change had been applied to this table on seven separate occasions and is expected to happen many more times in the future. In this case, continuation of the status quo is an extremely expensive proposition, inconsistent with strategic objectives and architectural principles of the company.

This represents a worst case scenario for updating a traditional application table. Members of other MIS support teams at the same company report that the DO table is implemented differently in other applications. Specifically, it is implemented in a more table driven manner, and requires only minutes or hours to change entries in one centralized location. The fact remains that the picture of inefficiency painted above undoubtedly applies to a shocking number of applications in many of today's large MIS shops. Paradoxically, it is also one of the easiest problems to rectify.

Table Driven Data Summarization for Batch, Online and Distributed Systems

In the following pages, two table driven approaches to data summarization are presented. Both techniques offer important benefits over the traditional approach. They improve the performance of summary processes in a mainframe batch environment. They also make it feasible to move such applications to online and distributed platforms.

Where the traditional approach to summarization sorts detail data prior to summarizing, the *inverted* approach summarizes unsorted detail data, then sorts the summarized data. The *continuous* approach also summarizes unsorted detail data, but continuously maintains the summary data in key sequence.

A discussion of performance factors and client/server architectures leads to a practical description of benefits for the business.

The Ever Present Application

What is the most popular type of automated business application used in every data processing shop? A few specific applications may come to mind but, certainly, some form of accounting summary process should be among them. From the very earliest batch applications to today's client/server systems, summarization is fundamental to any business. General ledger summaries track the progress of the enterprise for top

management. Payroll summaries maintain accumulations of time and compensation for employees. Spreadsheets extrapolate revenues and expenditures for project planning.

Although the simple addition of numeric values is clearly the most common kind of summary, the term "summarization" refers to a wide range of integrating processes for discrete input values. The list includes: standard accumulations and counts; percentages, multiplications and factorials; adjustment of weights in a priority list, neural network or fuzzy set; and any kind of data collection, generalization or abstraction technique involving a many-to-one compression of data values. These processes define the very essence of "decision support". They convert raw data into meaningful, manageable information.

Detail data to be summarized may be collected and routed to a number of different applications. Time report information, for example, may be integrated into general ledger, payroll and project-specific applications. Every corporate account may be summarized in various ways to meet administrative and decision support requirements.

The Summarization Legacy

Summarization techniques have developed historically in mainframe batch oriented environments. They have traditionally been applied to sequentially organized input files to produce corresponding sequentially organized output reports. Traditional summary processing is efficient if the input detail data is sorted or otherwise synchronized to group successive input values for summarization. Groups of detail values with identical summary keys may be processed one group after another, in a sequential fashion. If the data is not already synchronized according to summarization requirements, then additional overhead is incurred to sort the detail records in a prior step.

Various characteristics of legacy systems, including expensive main memory, limited addressability, large detail files and the I/O intensive nature of sort processes, all tended to keep summary applications in a batch world when other centralized mainframe systems went online in the 1970s and '80s.

The traditional approach to summarization is not appropriate for online systems nor for distributed client/server architectures. Randomly distributed input data cannot be guaranteed to arrive at the summarization server in a sorted fashion. The preliminary overhead of "batching" distributed input and sorting large amounts of detail data at the server site would have an unacceptable negative impact on response time for traditional summary requests.

Today's client/server systems are characterized by much cheaper, larger, distributed addressable memories. I/O and network traffic costs, by comparison, are relatively far more expensive. Successful, cost-effective algorithms depend upon processing approaches which minimize I/O and network data transfers, while maximizing the advantages of available memory.

Traditional summarization is so straightforward and ubiquitous that, until recently, other algorithms have escaped serious consideration by the programming community. Conventional wisdom implies "that's just the way it's done". Given the dramatic advances in hardware and software over the years, it may be time to challenge some of this long-lived wisdom.

Inverted Summarization

Detail data is customarily sorted prior to summarization for two reasons. First, it allows for easy correlation of input data with summary data as the summaries are generated. That is, input detail records are processed in groups, by key, in the same order that matching summary records are created. Second, it produces an output report that is also sorted for easy visual retrieval and ranking by a human client. Sorting prior to summarization, however, is not the only manner in which multiple detail records may be consolidated into a single summary record.

Depending on the particular application requirements, detail data may or may not need to be included in the output report, along with calculated summary values. If specifications demand that the detail data be sorted and reported, in line with summaries, then the traditional approach may well represent an optimal solution.

On the other hand, if there is no explicit requirement to include sorted detail records in the summary output, then there are other approaches to data summarization which should be considered. One involves first summarizing the detail data into a memory resident table, using a high performance access method to correlate input keys with summary keys. The table is sorted for reporting purposes only after all summary values are calculated. The access method used during summarization is a hash search, optimal for random distributions of keys, and tailored specifically for memory resident tables.

As each input record is read, the summary key is extracted and used to search the summary table. If the summary key is not found in the table, then this marks the first time that particular key has been encountered in the input stream. A new table row is initialized with the detail value and then inserted into the table to begin the summary process for that key. If the summary key is found in the table, the summary row retrieved is updated using the new detail value and then replaced in the table.

Once all input data has been processed, a summarized value exists in the table for each summary key, but not necessarily in an order suitable for reporting purposes. The hash summary table is then compressed and sorted, in memory, for reporting values sequentially. This summarize-then-sort process differs from the traditional approach of sort-then-summarize, as shown in Figures 25 and 26. The term *inverted summarization* applies to summary processes which reverse the sort and summarize steps.

The approach is easily extended to multiple levels of summarization. The lowest level summaries are first accumulated in memory, in a single pass through the detail data as described above. Separate summary tables are then created and maintained for each

additional, higher level control break, in a single pass through the lowest level table. Higher level tables can be built sequentially, since the lowest level table is already sorted.

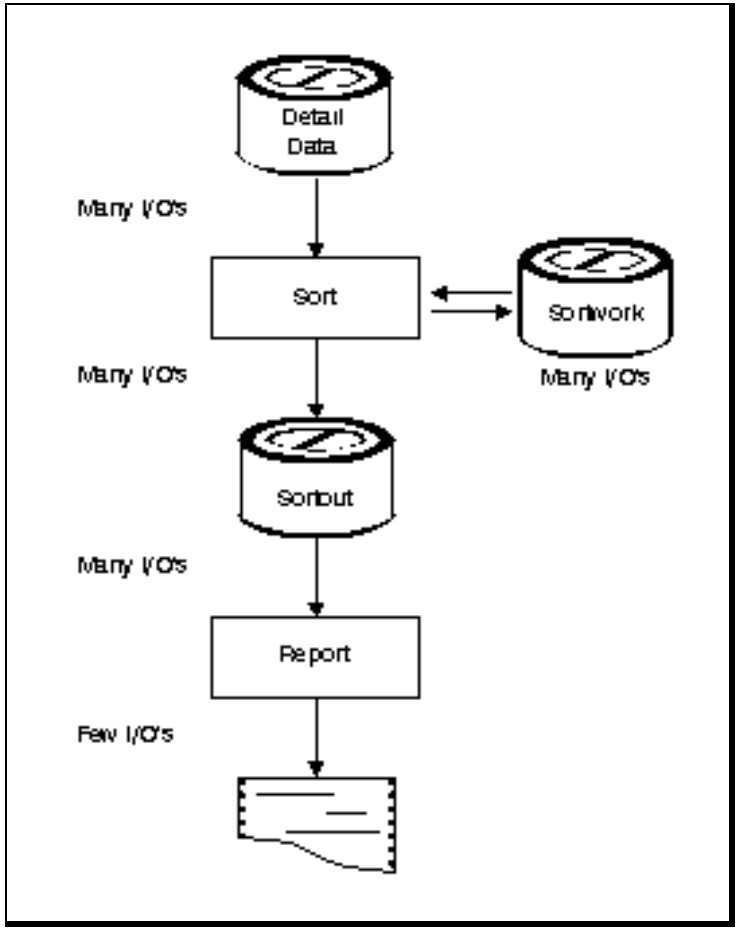


Figure 25 *Sort-Then-Summarize*

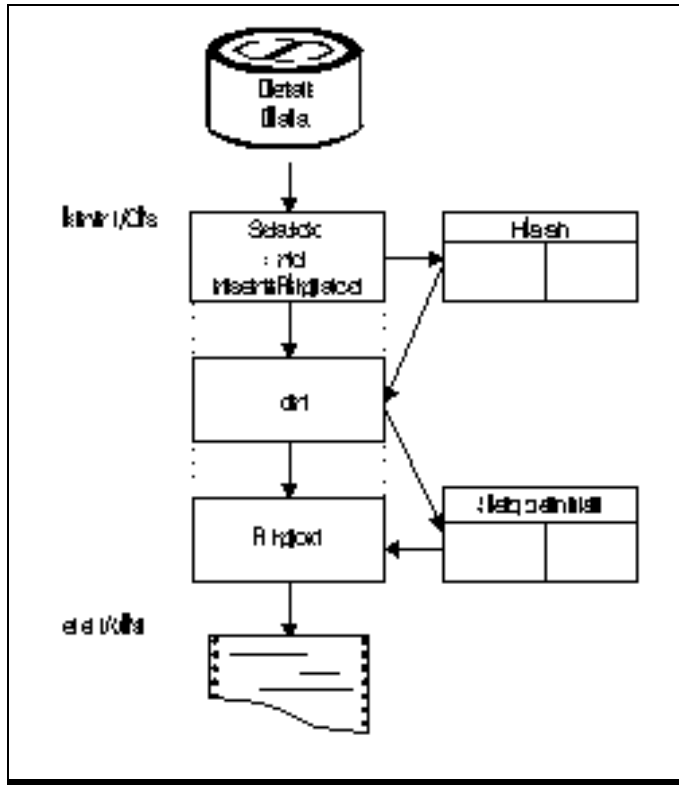


Figure 26 *Summarize-Then-Sort*

Performance Considerations

The inverted summarize-then-sort algorithm is typically more efficient in terms of CPU resources and wall clock time than the traditional approach. A number of interrelated factors affect the relative performance of these two approaches to summarization:

- Organization of input data
- Impact of sorting
- Efficiency of the correlate/summarize process
- Organization of output data

Organization of Input Data

Traditional summarization requires a sorted detail file. If the input data arrives in a random organization, then the data must be sorted according to summary control

breaks. For large data files, this is usually an I/O-intensive process. The inverted summarization approach is designed for detail data that is randomly organized with respect to some key. No preprocessing is required to reorganize input data sequentially. Rather, the summarized output data is sorted instead.

Impact of Sorting

A number of factors influence the overall impact of the sort process, including the choice of internal or external sort, the number of records to be sorted, and the degree to which the input is pre-sorted.

The degree to which input data is pre-sorted certainly affects the sort effort. Unless particular pre-sort conditions are consistently present, however, a more random scenario should be assumed. For the purposes of this discussion, we will assume that the input data arrives in a completely random distribution.

In today's environments, disk I/O due to external sorting has a recognized negative effect on response time and system throughput. This translates, directly and indirectly, into higher costs. By contrast, memory is a relatively inexpensive commodity. Internal sorting eliminates unnecessary disk I/O in exchange for increased memory resources, and has become the favored sort method.

Given extremely large volumes of detail records, the traditional approach with an internal sort still may not be considered a viable option due to memory limitations. For an inverted summarize-then-sort process, the number of unique summary keys determines both the required size of the hash table and the number of records to be sorted in the sort step. Even for very large input files, the number of summary records may be quite small, and memory requirements for reduced volumes of summary information may be quite acceptable.

The ratio of detail records to summary records is called the *consolidation ratio*. In the traditional approach, this ratio has little impact on summary processing, and has been generally ignored. For summarize-then-sort, however, the consolidation ratio is an indicator of sorting effort. Since the sort is deferred until after summarization, a high consolidation ratio implies a corresponding reduction in the number of records to be sorted. In the worst possible case, no consolidation takes place. Conversely, if thousands of detail records consolidate into a few hundred summary records, then the sorting effort is minimal.

Efficiency of the Correlate/Summarize Process

The traditional approach correlates detail data and summary values sequentially and reduces summary processing to a simple read operation, followed by an arithmetic calculation. For summarize-then-sort, correlation of detail data and summary values is more CPU-intensive. In this case, summary processing includes not only the arithmetic

calculation, but also the access method used to search the table, and the subsequent insert/replace operation to update the table. A hashing routine is an optimal access method for random searches. Usually the insert/replace is a direct access data move and does not require a redundant search of the table.

Organization of Output Data

In the days of batch-only processing, printed reports were typically shared by many users. Report data had to be sorted so that a human user could visually locate a particular line of interest. Since the advent of online transaction processing and more recent client/server systems, however, the computer has been enlisted to provide more specific information, tailored for a single user. Indeed, it is now valid to ask, "does the output data need to be sorted?"

If the user requires a list of summaries for various control breaks, or a relative ranking of summary values, then the output data should be sorted to facilitate visual scanning of the information. If there is a requirement for multiple levels of summaries, then sorting may be required for optimal processing. But if the user is interested only in retrieving individual summary records, then the data need not be sequentially organized. In this case, inverted summarization could omit the sort step entirely, for maximum efficiency. This is not possible with the traditional approach.

Balancing Efficiency Factors

In the context of performance constraints discussed above, the consolidation ratio is a key determining factor in the choice between traditional and inverted summarization approaches. The trade-off is between the cost of the sort versus the cost of correlating detail data with summary values. Generally speaking, if the consolidation ratio is low, then sort overhead is high for either approach and the traditional approach wins on correlation efficiency. If, on the other hand, the consolidation ratio is high, then the inverted approach begs a closer examination. Sort overhead is obviously reduced, since there are fewer records to be sorted, but at what point does this compensate for the increased correlation overhead? The answer to this question varies with the number of detail records, the type of sort used, and the degree to which the detail records are pre-sorted. Benchmarking is required to determine a precise ratio, but even this would vary to some degree with different input files.

Continuous Summarization

As described above, inverted summarization offers performance benefits over traditional summarization by avoiding the requirement to sort detail data. There is a variation of the inverted summarization algorithm, which may be referred to as *continuous summarization*, that maintains the summary table in key sequence at all

times, while the detail data is being summarized. The summarize and sort processes are integrated into a single unit.

The difference between inverted and continuous summarization becomes most apparent as new rows are inserted into the summary table. The inverted algorithm inserts new summary items into empty slots in a hash table. Since the table is not maintained in key sequence, no shifting of rows is performed during the insert, and the update is extremely efficient. Of course, a payment is extracted at the end of summarization, in the form of a sort, to report the final summary data sequentially. The continuous algorithm takes a compromise approach and uses an organization which is optimized for both random and sequential access. For example, a linked list binary index would support both kinds of processing very efficiently. When a new summary item is initialized and inserted into the summary table, the index is immediately updated to maintain the table in key sequence. Where the consolidation ratio is high (and insert operations are relatively low compared to replace operations), there is little overhead required to keep the table sorted.

Client/Server Models

While the benefits of both inverted and continuous summarization algorithms can improve the efficiency of batch applications, they may offer much more in a client/server context. Summaries which are not feasible using older techniques suddenly become practical with the new approaches.

Even in a client/server network, detail data can be summarized in several ways. Assume a server process at a regional site waits to receive input. Over a period of time, distributed clients "feed" detail data to the summary server via messages. Using the traditional approach, incoming messages could contribute to a "batch" of detail data at the server site, to be processed in a later summary request. In this case, however, the requirement for sorting detail data at the server site for each summary request might well make the approach impractical.

Alternatively, using either the inverted or continuous approach to summarization, when new detail data is initially loaded or existing detail data is updated by a client, a request could be sent to the summary server to immediately update a corresponding row in a summary table.

Various summaries may exist which draw upon the same detail data. Whenever detail data is entered anywhere in the system, that data should be broadcast to all summary servers throughout the network, so that all related summary tables for other applications may be updated at the same time, in parallel, no matter where they are located. Like an expanding supernova, the broadcast data rides the crest of a wave, leaving a lasting impression at all the summary "planets" it encounters. This would ensure that all summaries across the enterprise reflect the most current state of the business and client requests for summary information may be satisfied, on demand, with minimal pre-processing requirements at request time.

In Summary...

Both the inverted summarization algorithm, and it's continuous variation, improve the performance and scope of data processing's most common application. The potential applicability of these algorithms is enormous.

Table Driven summarization is gaining popularity as businesses are faced with increasingly large volumes of data. Meaningful information, in the form of collections of summary tables, must be available to managers, on demand, for the support of day-to-day business decisions. Ideally, they should be derived from an analysis of all possible business perspectives on the detail data. Architectural principles for today's business processes include timeliness and accessibility of information across the enterprise. The inverted and continuous algorithms allow for practical summarization of detail data in online systems and distributed client/server networks, where the traditional approach does not. In today's competitive business world, success depends on new perspectives for old problems, and fast access to up-to-the-minute information.

8
Administration of Tables

Memory resident tables and table data used for Table Driven Design have neither the same characteristics nor the same administrative requirements as database tables. From the perspective of a Database Administrator (DBA), the administration of a database management system (DBMS) is much more centrally controlled. The definition and reorganization of memory resident tables, on the other hand, are functions which are directly available to application programmers and, in particular, to end users. However, administration of memory resident application driving tables, although more distributed in nature, is still an important issue which must be addressed at an enterprise-wide level.

The most successful implementations of Table Driven Design efforts are coordinated by an administrative individual or group, responsible for providing in-house expertise and control over table structures and processes. This is more than a recommendation; it is a virtual requirement for success. The function includes standards and resource control, plus any related product support and vendor communication. It is independent of traditional database administration, and is often placed in the Data Administration Group or some other closely aligned area. The Data Administration Group, as general custodian of the enterprise's information assets, identifies, collects and classifies all types of data. Regardless of where the function is ultimately located, administration of memory resident tables must be recognized and promoted by senior management.

Standards

If Table Driven Design offers a method of improving the flexibility of applications, then the way must be illuminated by standards. Standards should be set for maximum reusability, naming conventions, quality assurance (training, documentation, testing and migration from test to production), resource utilization and distribution of responsibilities.

Standards vary from one organization to another, but the requirement is persistent. Standards should be investigated, implemented and enforced to maximize consistency across applications and processing platforms. Some guidelines are presented in the following sections.

Reusability Support Requirements

The following prerequisites must be recognized and implemented in order to support reusability. There must be:

- A process for identifying common application requirements and exploiting common solutions
- A mechanism for identifying, cataloging, distributing and controlling ownership of reusable/sharable components
- Standards and approaches for managing consistency, where replication is desirable

This applies to the components of table driven systems as well as other designs. Hierarchies of table libraries should be reused/shared across corporate levels or among applications, as required.

Naming Conventions

Names must be established with the aim of eliminating unnecessary duplication and inconsistency. Program working storage areas associated with particular tables, for example, should be cross-referenced to those tables by the use of common naming conventions. Thus, a row of the 'RATES' table might be retrieved into a working storage structure called 'RATES-ITEM-AREA', while a row of the 'FEES' table would be retrieved into a structure called 'FEES-ITEM-AREA'.

Names for tables and table libraries may include codes to identify administrative or user groups responsible for the resource. In addition, table library names may include codes to differentiate between test and production environments.

Transient tables defined in shared memory pools must follow naming conventions which guarantee unique table names. Table names may be based on a task or process identifier, or they may be generated by a special utility program designed for that purpose.

Training

To be effective, standards must be propagated across the organization. This is best accomplished through corporate training where the training itself meets established standards. Even where personnel have demonstrated high levels of dedication and ability, in the absence of appropriate training in Table Driven Design, maintenance efforts are often unproductive and inconsistent at best.

A smooth transition through the technology transfer stage must be supported by comprehensive training plans. Training must address the implementation, public relations, management and support of table driven and related approaches to application design and development. Without fully qualified personnel, even the best tools and methodologies can produce disappointing results.

Documentation

Removing program control variables and parameters from application logic generalizes and simplifies the application code. It does not eliminate application complexity, but merely moves that complexity to external tables for easier maintenance and sharing of business rules. Such a structured, tabular format also has inherent documentation benefits over embedded rules.

There may be some concern that removing detail control values from the program would make the logic more difficult to understand. Quite the contrary, in many ways, the application as a whole actually becomes easier to understand. For example, the generalized logic is simpler, stripped of conditional logic differentiating one rule from other, similar rules. Locating and extracting individual rules becomes a matter of scanning the appropriate table, where a potentially large group of related rules has a consistent, normalized structure. In addition, general knowledge about an entire class of rules can be derived from the columns of the table without examining individual rules at all. Together with the generalized logic, this information can lead to an effective, high level understanding of the application, independent of detail control flows.

In this situation, the requirement for documentation remains as strong as ever, but the table driven structure of the application forces a hierarchical structuring of documentation, from high level control flows (generalized code and generic table columns) to low level detail rules (specific values in table rows). Documentation is easily segregated according to level of detail. At the lowest level, every field value in a rule can be documented, either manually or as an automated lookup in another related (help) table. Locate the table, individual rule or field, and you locate the corresponding documentation associated with that level of detail. The fact remains, though, that the table must be well documented if all levels of control flow detail are to be understood. It is not sufficient to document the program alone.

Conversely, documentation embedded within a traditional procedural program may or may not be well structured. It is often difficult to determine the quality of such documentation until after detail control flows are located and understood. (This applies equally to the procedural component in a hybrid design.) If documentation is lacking in a table driven application, however, it is immediately notable by its absence.

Application programmers should be aware of the impact of Table Driven Design on documentation. An orientation period may be required to appreciate the benefits of a new style.

Testing Standards

Standards must be established for testing updates to tables, in addition to testing enhancements to programs. Ideally, procedures will be automated for migrating applications from production environments to test, and back again, so that end user groups may assume the bulk of responsibility for updating tables and testing their changes.

In many organizations, tables to be updated are simply copied from production libraries to test libraries. Once updated and ready for testing, the test library is simply added to the library search list of the testing process, so that it is searched before the production library. Updated versions of the test tables will be found in the test library; unmodified tables will be found in their customary location in the production library.

Once testing is completed, the table is copied back to the production library, to replace the earlier version, using the normal production control procedures.

Quality Assurance

Software quality is defined according to a variety of measures, including reliability, availability and effectiveness. Experimental studies suggest that the number of programming errors existing in any given source code, and the time required to find and correct those errors, is related to the number of distinct computational paths through that source code, called the *cyclomatic complexity*. It follows, that the simpler, more generalized procedural code of a table driven application is less susceptible to development errors. As noted in Chapter 8 - Documentation, removing program control variables and parameters from application logic may generalize and simplify the procedural code, but it does not eliminate application complexity. It merely moves that complexity from internal application code to external tables. The set of application driving tables must be a primary focus for quality assurance efforts.

Table driven application design contributes to end user empowerment and responsiveness in the area of change control. Whenever possible, end users should be able to update application driving tables and modify the behaviour of the system. Reliability and availability of any business application can be expected to improve when qualified business experts are updating business rules directly, rather than through an MIS intermediary. This does not necessarily mean the end user is free to alter production tables and applications, however. As always, there must be a system of

checks and balances in place to protect the integrity of application rules in a production environment (see Chapter 8 - Testing Standards).

For table driven systems, migration strategies and controls are similar to those for traditional systems. In most situations, tables are simply considered to be extensions of application code. There may well be application specific cases where driving tables are updated in production, much the same way that databases are updated in production, but this is not yet a widely accepted approach. In any case, the need for integrity control and version tracking applies to application driving tables, as it does to customer data and program code.

Migration requests from production to test and back should be automated to facilitate end user access to tables in test environments.

Security

Security is as much a concern for application driving tables as it is for other kinds of data. Memory resident tables must be subject to the same security controls currently in place for files and databases. Some tables should only be accessed by particular applications and particular users. Table libraries should be identified to the installation's security system for access control.

All data values in tables should be validated at data entry time to ensure the integrity of application driving information.

Resource Management

Information Systems resource considerations have changed dramatically over the past 30 years. Application resources must be managed effectively to take advantage of new opportunities.

The Data Dictionary

No resource can be adequately managed in the absence of an accurate inventory. The importance of inventory control must be recognized. The data dictionary must be formally established as a mechanism to identify, catalog, distribute and control ownership of reusable/sharable components of table driven systems. This concept should be extended to include rules required at execution time, by table driven systems, to create an active data dictionary.

There are currently thousands of registered data items in a typical corporate data dictionary. This can be expected to grow significantly as the trend continues toward more sophisticated repository management.

For table driven systems, resource names and attributes to be found in the data dictionary include fields (columns), rules, tables, table libraries, source modules, source libraries, load modules and load libraries

Memory Resources

If Table Driven Design encourages the effective use of memory resident tables, then system resources must be available to support the increased use of memory. In fact, contemporary memory facilities are currently under-utilized in many shops. Legacy applications generally focus on limiting memory use in favour of I/O intensive approaches.

Historical reasons for the limited use of memory resident tables in legacy systems relate to cost/performance considerations and hardware/software limitations. Because of high volume access to control information in these tables, they are simply not suitable for implementation under I/O bound file access methods or database systems. In the past, however, real memory costs limited the size of resident tables, so many data structures which should have been placed in memory were instead implemented as external files. For similar reasons, the development of sophisticated memory based access methods and operators fell by the wayside. In the absence of these facilities, with some notable exceptions, the promise of table driven programming failed to live up to its potential. In recent years, cost/performance considerations have been turned upside down and hardware/software limitations have been extended far beyond the conceptions of earlier systems designers. Mainframe systems now offer program and data addressability in the order of 16 Terabytes, or 16 million Megabytes. Extrapolating on industry trends, it will not be long before this potential moves to the desktop (see the discussion of table size in Chapter 3). Techniques which depend on larger addressable memories are now clearly more economical when compared with I/O intensive solutions. Today's improved memory management facilities are designed to make more effective use of main memory in these new approaches. In addition, personnel costs and programmer productivity have become highly visible issues. There is every reason to expect that these trends will continue.

It is now in the best interests of the enterprise to exploit available memory resources to the fullest.

9
Preparing for the Future

Over two thousand years ago, in a book of strategy called "The Art Of War", Chinese philosopher-warrior Sun Tzu wrote:

"... a military force has no constant formation, water has no constant shape; the ability to gain victory by changing and adapting to the opponent is called genius."

Sun Tzu bases this quality called genius on some very straightforward and fundamental principles. Table Driven Design is also straightforward and fundamental to flexible, minimal maintenance business applications. The ability to change and adapt is essential to the success of today's information systems, and to the business which employs them.

Table Driven Design promotes shareability, reusability and reduction of application code, resulting in flexible systems which are more readily adapted to change. It is consistent with general business directions and architectural principles. Table Driven Design is compatible with accepted development methodologies and software environments. In addition, it complements and enhances other efforts currently under way to simplify, reuse and share application resources.

Dynamic as it is, though, not all elements of our world are constantly changing. At the confluence of traditional and emerging design ideas, perspective and balance are key words to consider. Procedural and table driven system components are complementary. Not all rules are tables. If the number of decision rules is small and processing is invariant or easily visualized in a procedural manner, then a procedural approach is the preferred choice. Dogmatic extremes should be avoided.

There are two approaches to demonstrating the effectiveness of Table Driven Design. The first approach involves centralizing critical examples of corporate level table data which is frequently accessed by multiple applications. This approach combines a relatively simple conceptual effort with enterprise wide benefits. A single, well chosen table can be selected for corporate wide access, populated with data, centrally maintained, and widely promoted. A single application would be converted to use the table, as a demonstration, with the intent of more to follow once the conversion effort is well understood by other groups. At the level of a single application, subsequent maintenance benefits may well be notable, but the corporate wide effect is shown to be particularly impressive, with minimal effort.

The second approach involves re-engineering a single system for the maximum localized benefits of Table Driven Design. In this case, the full power of the design approach is clearly demonstrated to a relatively isolated audience.

The old adage, "if something works, don't fix it", has lost much of its validity in the context of legacy information systems. The problem of inflexible program logic, together with a proven history of frequent, essentially predictable enhancements, clearly indicates a need for action. Conversely, the lack of immediate perceived pain may seem to imply no tactical need to change and thus no commitment to change on the part of IT management. However, system changes are still required in response to strategic initiatives. To remain competitive, MIS organizations must aggressively pursue design approaches and technologies which promote responsiveness, shareability, reusability and reduction of application code across the entire corporation. This includes, but is not restricted to, Table Driven Design.

The advantages of flexible application logic and Table Driven Design are already appreciated by personnel within many organizations. Ongoing studies and pilot projects are preparing such organizations for an effective implementation of these concepts. Many analysts and programmers are already using Table Driven Design techniques in isolation. A commitment to training can carry this initiative through to all information systems groups.

The strategic importance of this effort may well be understood at various technical levels in the enterprise, but it must be officially recognized and promoted from the top levels of management. Together with a family of emerging technologies, table driven application design effectively addresses today's maintenance burden. It promises to position the enterprise well for tomorrow's new challenges.

Bibliography

Anstead, C., “User Driven Update”, MARK IV User’s Group Conference Proceedings, 1979.

Cauthen, H., "Re-tooling The Applications Development Cycle With Tables", tableBASE User Group (tBUG '91) Conference Proceedings, Data Kinetics Ltd., 1992

Curtis, W., "Measuring The Psychological Complexity Of Maintenance Tasks With The Halstead And McCabe Metrics", IEEE Trans. Software Engineering, vol. 5, March 1979

Glasser, S.J., "Data Processing Concepts And Techniques", University of Toronto, 1977

Givone, D.D., "Introduction To Switching Circuit Theory", McGraw-Hill, New York, 1970

Gleick, J., "Chaos, Making A New Science", Viking Penguin Inc., New York, 1987

Green, T.F. et al, "Program Structures, Complexity and Error Characteristics", Computer Software Engineering, Polytechnic Press, New York, 1976

Hill, F., "Program Understanding And Enabling Technology", Proceedings, 9th International Conference on Software Maintenance and Re-engineering, Washington DC, 1991

Hurley, R.B., "Decision Tables In Software Engineering", Van Nostrand Reinhold Co. Inc., New York, 1983

IBM, "Extended Adressability", MVS/ESA SPL manual no. GC28-1854

Knuth, D. “The Art of Computer Programming”, Volume 3, *Sorting and Searching*, Addison-Wesley, 1973.

Piper, J., "DoD Software Reuse Vision And Strategy", CrossTalk - The Journal Of Defense Software Engineering, no. 37, Software Technology Support Center, October 1992

Popper, K.R., The Open Society And Its Enemies, vol. 1, "The Spell Of Plato", Princeton University Press, Princeton, 1966

Restak, R., "Brain By Design", The Sciences, vol. 33, no. 5, New York Academy of Sciences, September/October 1993

Smith, J.W. and Johnson, T.R., "A Stratified Approach To Specifying, Designing And Building Knowledge Systems", IEEE Expert, June 1993

Stodder, David, "Riding Capitalism's Tumult", Database Programming And Design, October, 1993

Sun Tzu, "The Art Of War", translated by Thomas Cleary, Shambhala, Boston, 1988

"tableBASE Programmer's Guide", Data Kinetics Ltd., 1993

"tablesONLINE/CICS User Manual", Data Kinetics Ltd., 1993

Wiorkowski and Kull, "DB2 Design And Development Guide", 3rd Edition, Addison-Wesley, 1992.