Association for Information Systems
AIS Electronic Library (AISeL)

ICIS 1985 Proceedings

International Conference on Information Systems (ICIS)

1985

Learning from Prototypes

Vasant Dhar
New York University

Matthias Jarke
New York University

Follow this and additional works at: http://aisel.aisnet.org/icis1985

Recommended Citation
Dhar, Vasant and Jarke, Matthias, "Learning from Prototypes" (1985). *ICIS 1985 Proceedings*. 14.
http://aisel.aisnet.org/icis1985/14

This material is brought to you by the International Conference on Information Systems (ICIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in ICIS 1985 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

Learning from Prototypes

Vasant Dhar and Matthias Jarke
Graduate School of Business Administration
New York University

ABSTRACT

Structured methods for the analysis and design of information systems have largely focused on representations and control mechanisms for the *outcomes* of the design process. Prototyping methods are more sensitive to critiques during the design *process* itself but do not preserve knowledge about it explicitly. In this paper, a systems architecture called REMAP is presented that accumulates design process knowledge to manage systems evolution. To accomplish this, REMAP acquires and maintains dependencies among the design decisions made during a prototyping process. It includes a model for learning general design rules from such dependencies which can be applied to prototype refinement, systems maintenance, and design re-use.

Introduction

The process of large systems development is often iterative, involving continuous modifications to programs before a "satisfactory" design emerges. Designers have attempted to use a *prototyping* approach whereby a working prototype system is assembled quickly on the basis of an initial assessment of a a problem situation, and then refined repeatedly in response to critiques from users or design personnel. While this approach may offer significant advantages over "structured" approaches in terms of earlier user involvement, a major drawback is that the initial construction of the system and the process of successive refinement can be haphazard, failing to take cognizance of the rationales for the initial design decisions and for successive changes in these decisions.

This paper employs a case study in the oil industry to analyze these shortcomings in some depth, and presents an artificial-intellignece based architecture called REMAP (REpresentation and MAintenance of Process knowledge) which enhances the iterative design procedure typical for the prototyping approach by the capability of preserving knowledge about the design process, and applying this knowledge in analogous design situations.

The case study has revealed several types of *process knowledge* that appear to be central to systems development. First, the design process consists of a sequence of interdependent design decisions. The *dependencies* among decisions are typically based on general application-specific *rules*; however, these rules are seldom articulated explicitly by users or analysts. Second, when systems are developed in a piecemeal fashion following the prototyping idea, analysts apply *analogies* to transfer experience gained from one subsystem to "similar components" of another. Unfortunately, current development methodologies preserve none of these aspects of process knowledge, making the process of prototype refinement and transfer of experience ad-hoc and susceptible to error.

It appears that the systems development process would benefit greatly if the dependencies among decisons could be represented explicitly, and more importantly, if the general basis for them could be extracted *during* the course of analysis and development. This could lead to a more systematic modification of prototypes and improved maintenance of full-blown implementations. Perhaps more importantly, this knwledge could be used to identify analogous features of different systems precisely, enabling the use of cumulative learning for subsequent designs in the same general application area.

The paper is organized as follows: Section 2 begins with a brief description of the prototyping process; detailed real-world examples are then used to show the need to maintain process knowledge. A formal model of our approach is presented in section 3, along with an overview of a partial implementation of the REMAP architecture. Section 4 provides a discussion relating the model to previous work in systems analysis and artificial intelligence. We conclude with a summary of possible applications which may benefit from the REMAP approach.

The Need for Process Knowledge

REVIEW OF PROTOTYPING

Prototyping is an iterative systems design and development methodology. Figure 1 provides a highly simplified illustration of the main steps involved (Jenkins, 1983). After an initial design has been established, the method follows an assessment/revision/enhancement cycle of *working prototype refinement.* Driven by user critiques, this cyclic process continues until a satisfactory system, the "operational prototype", has been inplemented (right branch of Figure 1). However, if systems *requirements change* subsequently (dashed line in Figure 1), the system leaves the steady state achieved in the "operational prototype" and enters a new refinement cycle. In large systems development, where a single user cannot completely understand the reprecussions of requested changes, designers frequently employ a "protocycling" approach which permits user critiques at multiple levels of a quasi life-cycle approach such as the data flow diagram or the program specification level (Balzer et al., 1982).

Prototype refinement as well as requirments modification frequently involve a reconsideration of the design developed in earlier cycles. It is the purpose of the REMAP approach reported in this paper to accumulate the knowledge gained in every cycle in order to focus and facilitate later revision and enhancement steps of the cycle.

A CASE STUDY

In order to establish a context for the discussion, we shall use an example obtained from the case study of a very large systems analysis and design project. The problem involves the design and subsequent maintenance of a series of sales accounting systems for different products of an oil company, here referred to as OC. OC sells oil and natural gas-based products with different characteristics to its subsidiaries and to outside customers in different parts of the the world. Sales Accounting at OC's Corporate Headquarters requires generating various integrated reports for purposes of audit and control. Input to Sales Accounting is based on invoices generated from transactions in a number of offices in the U.S. and abroad.

For the sake of readability, the system representation is restricted to the Structured Analysis level (DeMarco, 1978; Gane and Sarson, 1979). Note, however, that the problems described here, and our approach to solve them, are not restricted to this level but appear in any prototyping situation.

Systems designs are described in terms of data flow diagrams at various levels of abstraction. A data flow diagram is a network where the nodes represent processes, external entities, or data stores (files), and directed arcs represent the data flows from one node to another. Process nodes are frequently called "bubbles"; each bubble can be decomposed into a lower-level data flow diagram. Bubbles at the bottom level have associated mini-specs on which the program designs are based. Data flow and data store information is managed in data dictionaries. Figure 2 shows the notational conventions used in this paper.

Part of the structured top-down design of OC's Sales subsystem is illustrated in figures 3 through 6. Figure 3 shows level 0 of the system. In this example, since Sales comprises the entire system, this can also be used as the context diagram which depicts the relationship of the system to external entities. Figures 4, 5, and 6 are data flow diagrams for levels 1 and 2 of the sales system. Level 2 (fgures 5 and 6) are the bottom level decompositions of the bubbles 1 and 3. Each of the bubbles at this level have an associated mini-spec (not discussed here).

We now illustrate the problem of design adaptation using three scenarios. Each requires a different extent of modification to the original design, and illustrates the need for a different aspect of process knowledge. All of the examples involve external requirements changes (dashed line in Figure 1) but similar problems also occur during the refinement cycle.

SCENARIO 1: THE ROLE OF GENERAL AND SPECIFIC KNOWLEDGE

"London Sends Formatted Invoices". In the original design, the difference between the New York and London invoices was that the former were accessable *formatted* whereas the latter were received *unformatted,* on magnetic tape. Hence, a minor "convert" operation was required to bring the inputs into a format required by the "verify and correct on line" operation (bubble 1.1).

As a simple change, suppose that the London office begins to send corretly formatted invoices on magnetic tape to central headquarters. What kinds of design modifications are required?

It is clear that the change is not at a high enough level to affect the more abstract parts of the design in figure 4. However, at the next lower level (figure 5), the "convert" bubble is not required anymore since the London invoices should now proceed directly for verification.

In order to be able to assimilate this minor change, the system must know that in the existing design, the convert bubble is dependent on the existence of the dataflows representing London invoices. On recognizing that London

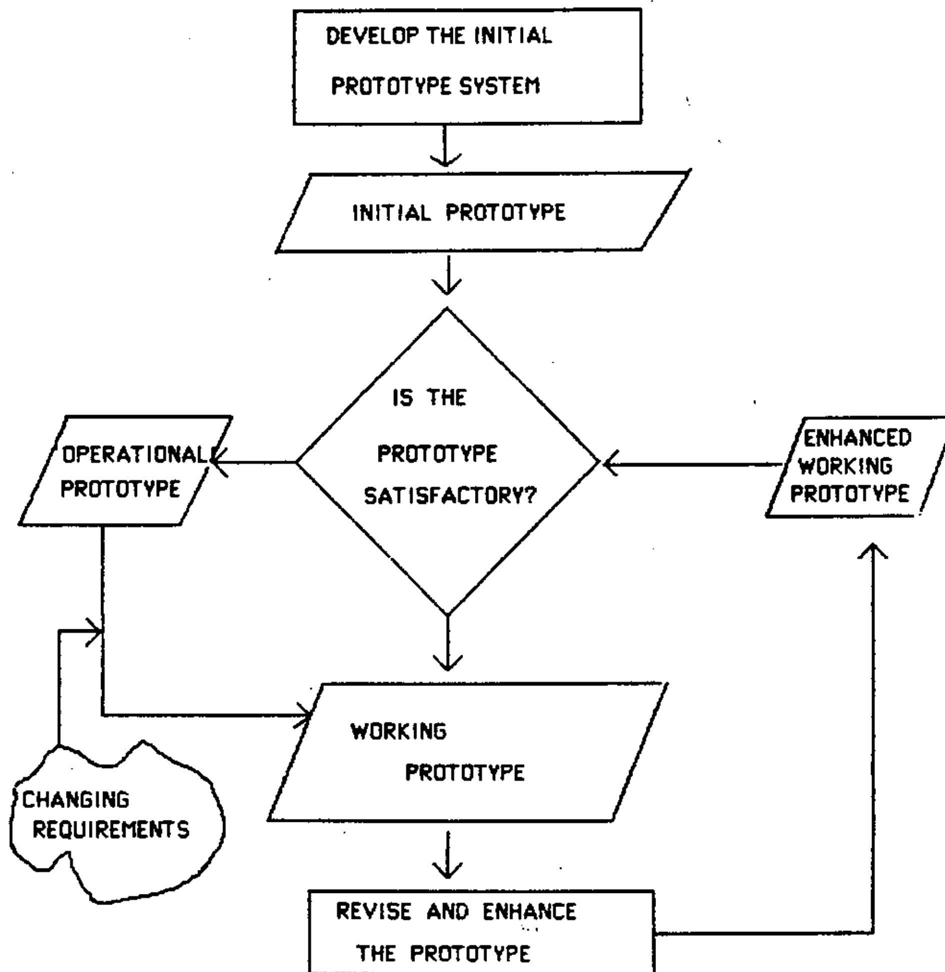


Figure 1

Application System Prototype Model
(Adapted from Jenkins, 1983)

invoices are now not unformatted, it should be able to detect the fact that conversion is unnecessary. Further, it should also know that *in general,* formatted invoices proceed directly for on-line verification. Based on this, it should direct London invoices to the "verify and correct on line" operation.

In summary, we have used two types of knowledge in understanding the existing design and the effects of changes to it: *general knowledge* about domain-specific constraints (i.e., unformatted invoices require conversion), and *specific knowledge* about the purpose of existing design objects in the form of rationales for existing design choices (i.e., the existence of the convert bubble in figure 5 depends on the existence of unformatted invoices).

SCENARIO 2: THE ROLE OF ESSENTIALITY

"London and Tokyo Will Not Sell Fuels Anymore". This represents a more radical type of change than the first. Intuitively, it seems clear that there are likely to be design changes as well as major related modifictions in several section of the code. In this case, lack of invoices from Tokyo obviates the need for a manual add and edit operaton at level 1 (a *manual* input operation was required because these were *paper* invoices). However, the *auto* load and edit is still required because New York invoices must still be processed.

This example illustrates the idea of *essentiality* in design; the tokyo invoices dataflow was an *essential* input for

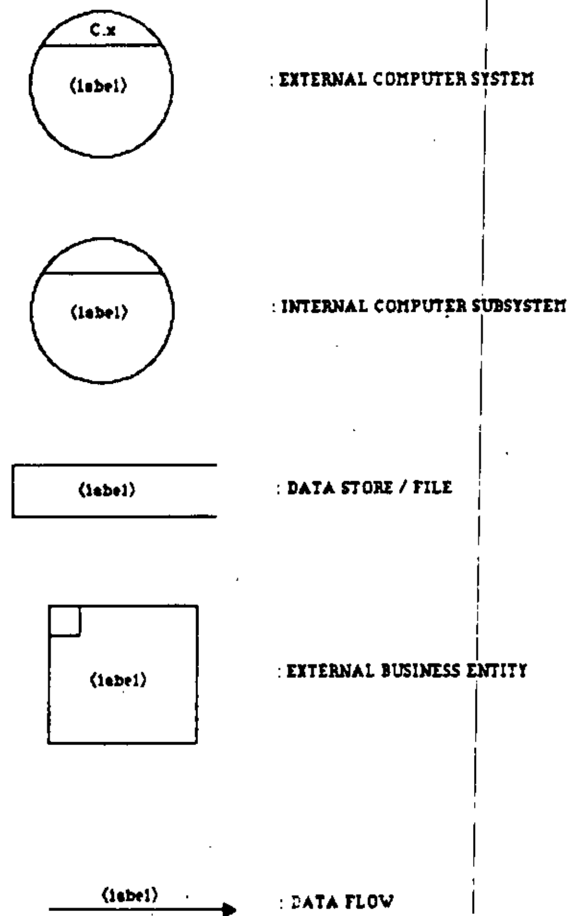


Figure 2

Data Flow Diagram Conventions

manual add and edit. In a more general sense, the *purpose* of a manual add and edit operation was to process paper invoices. The other inputs to it (the discount payable slips, codes and expenses) were *auxiliary,* and in fact *dependent* on Tokyo invoices.[1] In effect, bubble 1 stays (although some of its lower level components corresponding to London operations are removed) while bubble 3 must be deleted. The revised level 1 dataflow design is shown in figure 7.

It should also be noted that although the manual add and edit operation is no longer necessary, some of the lower level operations associated wth it are still required in order to process New York invoices. At the programming level, this means that the code corresponding to those operatons is not deleted since it is shared with the auto load and edit process.

SCENARIO 3: THE ROLE OF ANALOGY

"The Venezuela Office Will Sell Fuels". This corresponds to a high level change that is likely to induce widespread changes into the existing design. First, some additions must be made at level 1. The types of changes, however, depend on the nature of the sales invoices from Venezuela. If the invoices are computerized, an input into bubble 1 is required whereas paper invoices would call for introducing a manual add and edit operation.

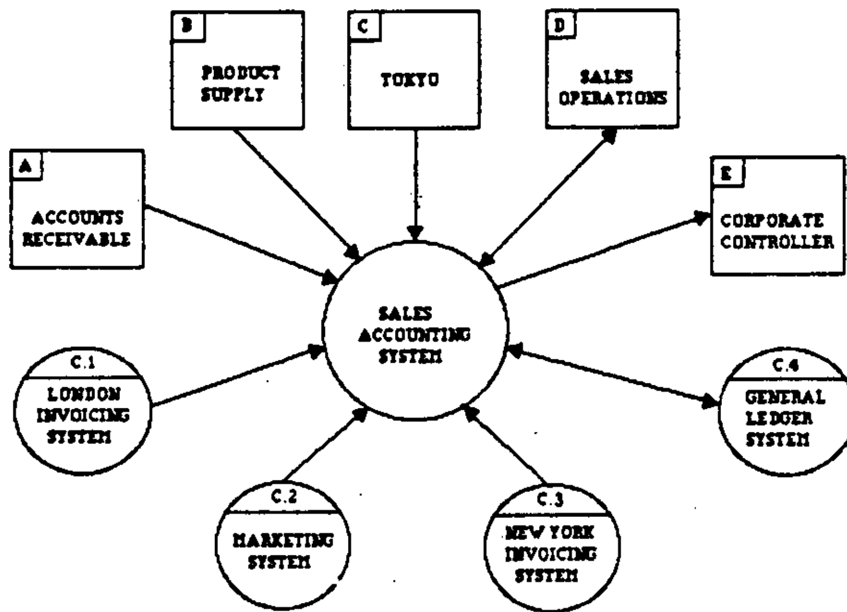


Figure 3

Sales Accounting Systems Context Diagram

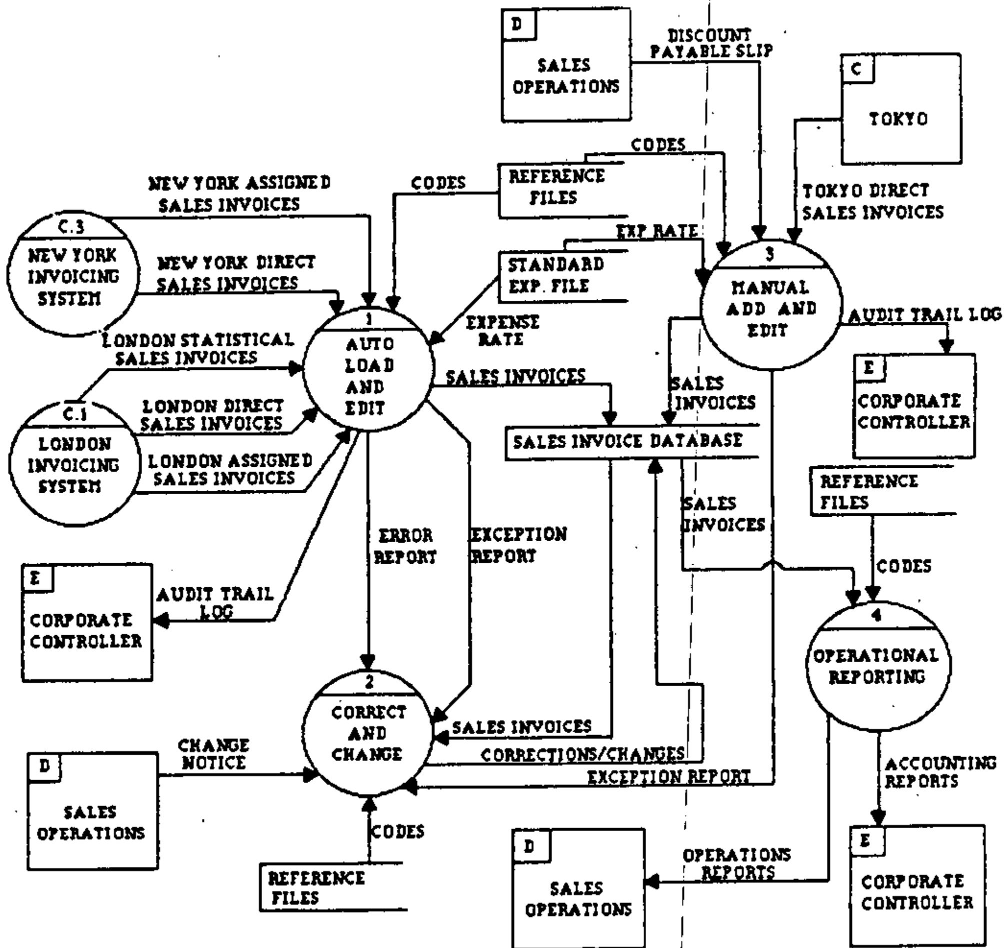


Figure 4

Fuels Sales (Initial)

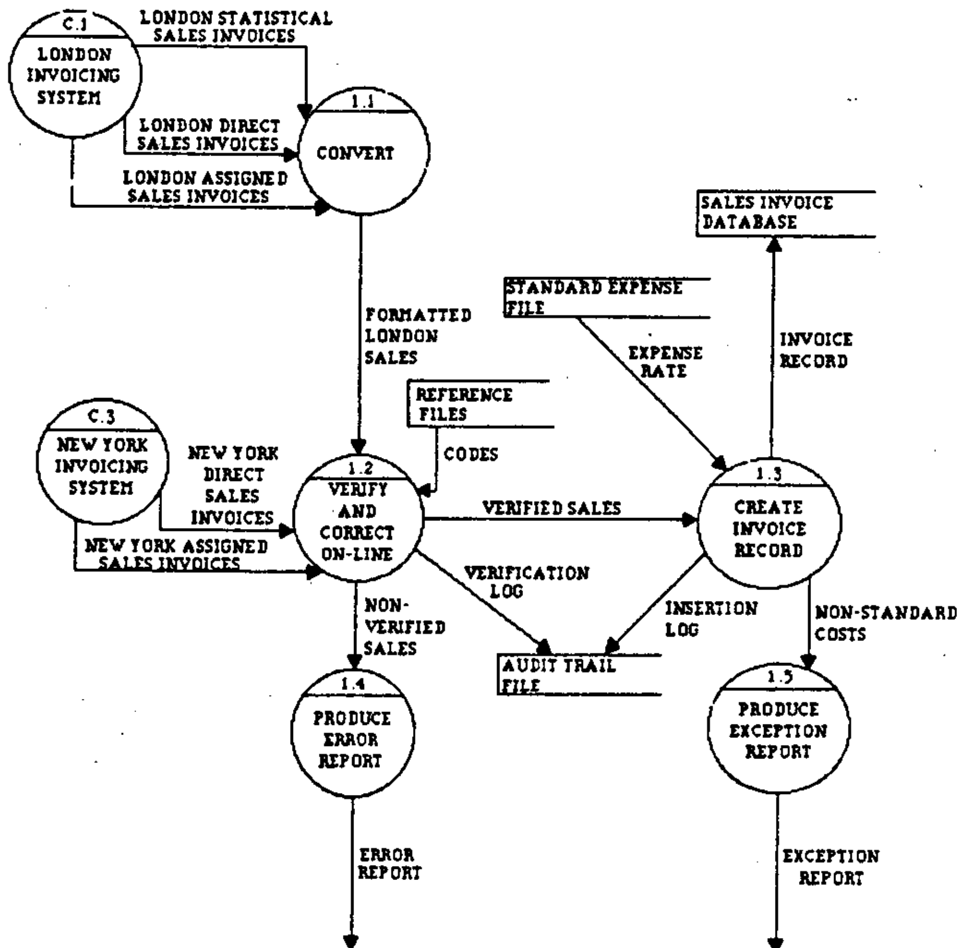


Figure 5

Auto Load and Edit

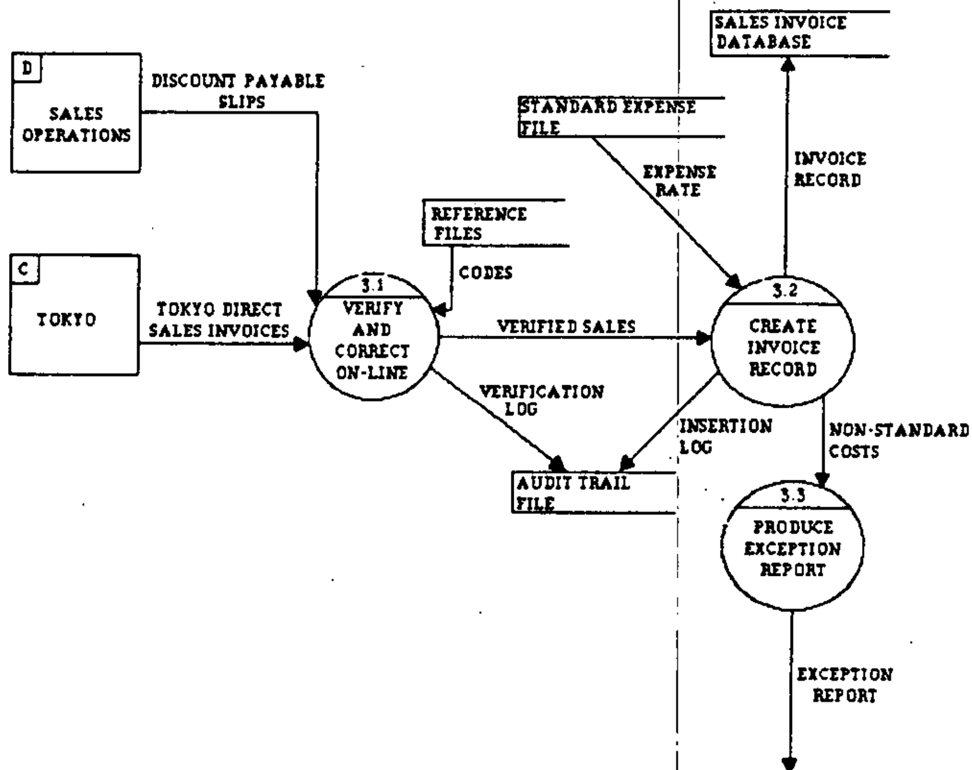


Figure 6

Manual Add and Edit

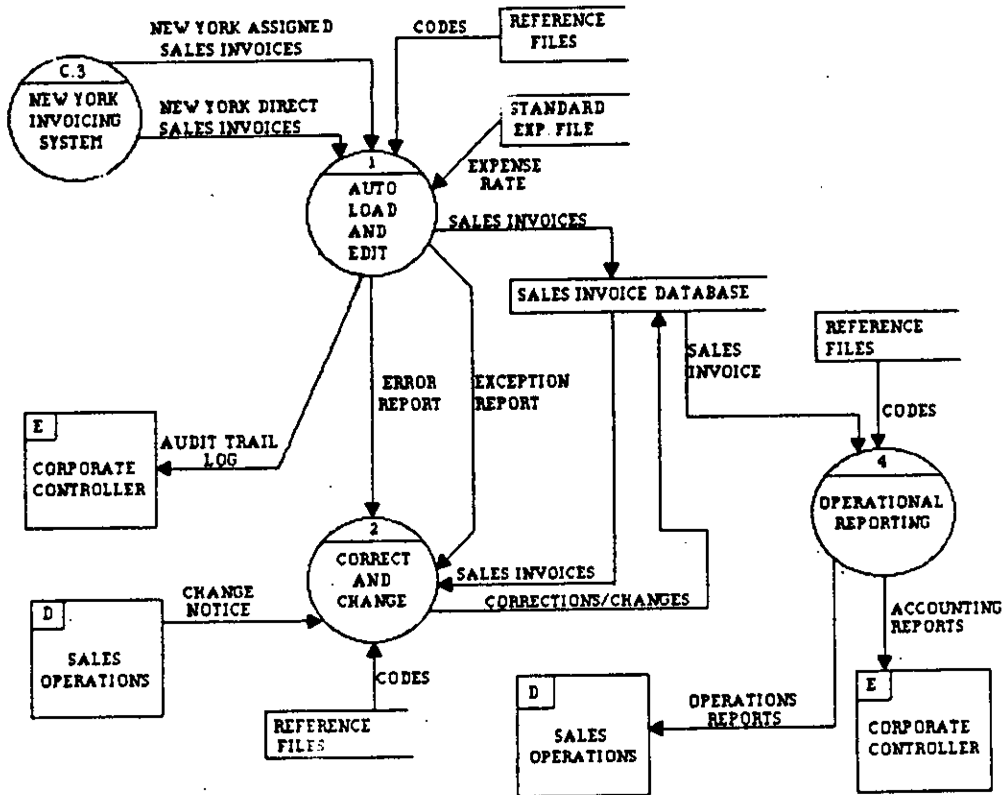


Figure 7

Fuels Sales (Modified)

Similarly, at the next lower level, the operations, required would depend on other, more detailed features of the invoices (i.e. are they formatted, unformatted, etc.).

This example illustrates the use of *analogy* in reasoning about a new situation. Design additions at the various levels depend on how "similar" the Venezuela invoices are to existing ones, and the design ramifications of these similarities and differences. This type of reasoning requires a system to carry out an elaborate match between design parts the system currently knows about, and a new design in order to draw out their analogous features. Specifically, it requires some notion of what the *important* dimensions are in the analogy being sought. In this example, relevant attributes in drawing the analogy are the *medium* of the invoices, that is, whether they are computerized or manual, and whether they are *formatted.* Once the important features are realized, the design ramifications become clear.

SUMMARY: THE NEED FOR TELEOLOGICAL KNOWLEDGE

In walking through the examples, we have attached fairly rich interpretations to the various design components that are *implicit* in the design. These interpretations derive from the *purpose* of the application which cannot be determined form looking at the resulting design alone. Since the design is an artifact (Simon, 1981), its teleological structure is imposed by the *designers'* conception of the problem. This conception may change repeatedly during the evolutionary design process. In other words, there is no *a priori* "theory" relating problems to designs; rather, the rationale for a particular design follows from a subjective world-view of the designer.

If a program is to be able to reason about the types of changes illustrated in the examples, it must have a formal representation for the knowledge that reflects the teleology of the design. Because such highly contextual knowledge about a potential application area is impossible to design into a system a priori, the knowledge must be *acquired* by the system *during* system design. To do this, the program must be equipped with mechanisms that enable it to learn aobut design decisions in an application area that it knows nothing about at the start of the design. It must then apply this growing body of acquired knowledge to reason about subsequent modifications to an existing design, or to construct new designs based on new but similar requirements. In the following section, we describe some broad aspects of an architecture called REMAP that is geared toward the extraction and management of the process knowledge involved in systems analysis and design.

Remap: An Architecture for Process Knowledge

It is apparent from the examples that application-specific knowledge plays a key role in reasoning about a design. This raises an important question, namely, how is this knowledge to be *acquired* by the system?

In most projects involving th construction of a knowledge based system, the system builder constructs the model of expertise by first specifying a representation, and then accreting the knowledge base in accordance with the precepts underlying the chosen representaion. Unfortunately, large scale application developments take place in a wide variety of domains that may have little in common. This uniqueness of each application situation discourages construction of a knowledge base that might be valid for a reasonable range of applications.

If a knowledge based system is to be able to support the process of systems analysis and design, it must have an initial representational framework, and mechanisms to augment this framework with domain specific knowledge that captures the purpose of design decisions and relationships among them. As more is learned, it should be possible to use this process knowledge to reason about design changes, and draw analogies in extending a design to deal with new situations.

A knowledge-based tool needed to support such a process requires four major components:

1. a classification of application specific "concepts" into a taxonomy of design objects, and mechanisms for elaborating this structure as more knowledge is acquired by the system;

2. a representation for design dependencies and mechanisms for tracing repercussions of changes in design;

3. a learning mechanism for extracting general bases for dependencies among design decisions made by the analyst;

4. an analogy based mechanism for detecting similarities among parts of similar subsystems. This mechanism should make use of the classifications in the generalization hierarchy to draw analogies between systems parts.

In the following subsections, we develop a knowledge representation for this process knowledge, and present a model of how it might be extracted and used by the REMAP system architecture.

REPRESENTING DESIGNS USING STRUCTURED OBJECTS

The REMAP model centers around *design objects*. The designer defines *instances* of such objects, whereas the REMAP system maintains a *generalization hierarchy* of object *types*. The structure of an object type definition in the hierarchy is as follows:

```
OBJECT TYPE
   type_name  : <string>
   child_of   : <set of object types>
   parent_of  : <set of object types>
   components : <set of slots>
   operators  : <set of procedures/methods>
```

The "child-of" and "parent-of" components position an object type in the generalization hierarchy. "Components" slots describe typical aspects of an object instance of the given type. As an example, consider the initial top-level definition of a generic object type:

```
OBJECT TYPE
   type_name  : generic_object
   child_of   : ( )
   parent_of  : unknown
   components : (identifier : <string>
                 type       : <string>
                 because_of: <set of objects>)
   operators  : (define, remove)
```

This object type has no parent since it is at the top of the hierarcy, and its children are yet to be specified. The "because-of" slot defines the *raison d'etre* of an object instance and will be further discussed in the next subsection.

A "generic" object provides very little structural information about its semantics. It is therefore useful to *specify subtypes* where additional slots are defined in order to capture the meaning of object instances of such a subtype. This can be represented using a generalization hierarchy of object types as shown in figure 8. Some instances of dataflows and transforms used in the three scenarios of section 2 are shown in figure 9.

In principle, the system could begin with the generic object type and then learn all subtypes from scratch. Since such a procedure would be rather cumbersome for the designer, the system should be provided with a small initial knowledge base. In the Structured Analysis example used throughout this paper, this consists of the definition of object types corresponding to data flow diagram conventions. The five major components are defined below (cf. figure 8):

```
OBJECT TYPE
   type_name  : dataflow
   child_of   : generic_object
   parent_of  : unknown
   components : (part_of    : dataflow;
                 medium     : <string>;
                 from, to   : process)
   operators  : (redirect, nostart, noend)

OBJECT TYPE
   type_name  : transform
   child_of   : generic object
   parent_of  : (process, external, datastore)
   components : (inputs, outputs : <set of dataflows>)
   operators  : ( )

OBJECT TYPE
   type_name  : process
   child_of   : transform
   parent_of  : unknown
   components : (part_of : process)
   operators  : (expand, noinput, nooutput)

OBJECT TYPE
   type_name  : datastore
   child_of   : transform
   parent_of  : unknown
   components : (data_structure : <set of data
                                   elements>)
   operators  : (define_structure, noinput, nooutput)

OBJECT TYPE
   type_name  : external_entity
   child_of   : transform
   parent_of  : unknown
   components : ( )
   operators  : ( )
```

External entities could be further broken down into data source, data sink, and interactor. The slot value "unknown" refers to the fact that the slot values should be, but have not yet been, defined.

As an example of *instance definitions*, consider the following description of the "London" external entity and one of the sales invoice dataflows generated by it (cf. figure 9).

```
{identifier   : London
 type         : external_entity
 because_of   : ( )
 inputs       : ( )
 outputs      : (London-direct-sales-invoices,
                 London-assigned-sales-invoices,
                 London-statistical-sales-invoices)
```

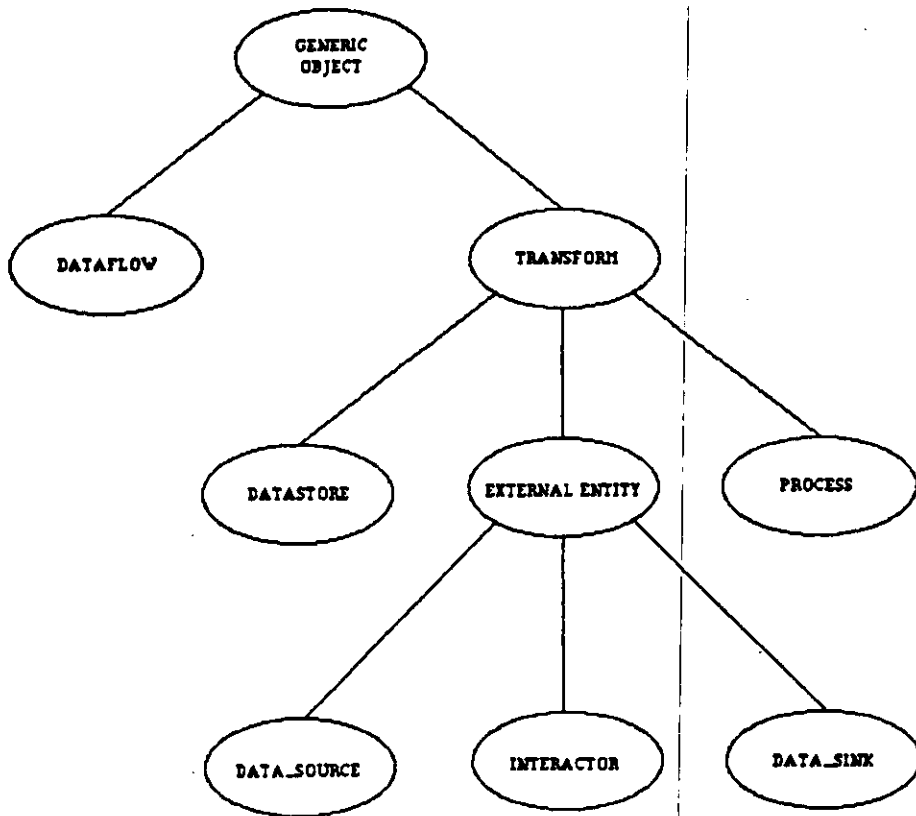
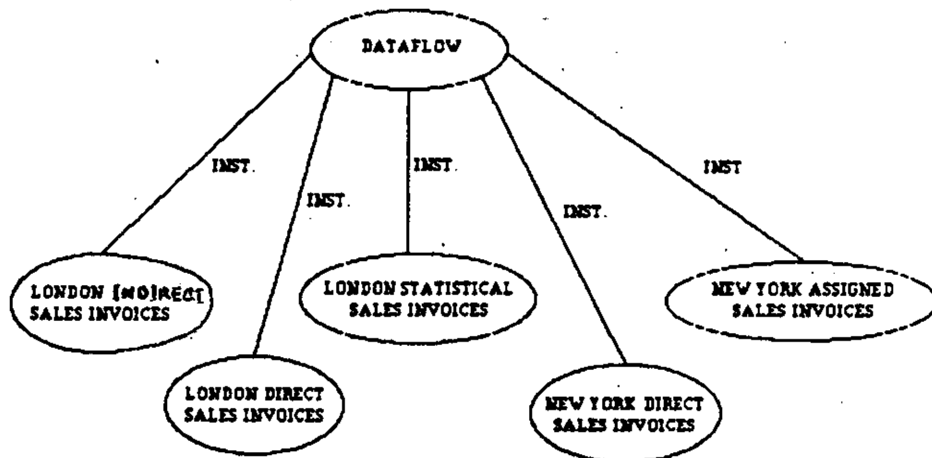


Figure 8

Initial Object Type Hierarchies



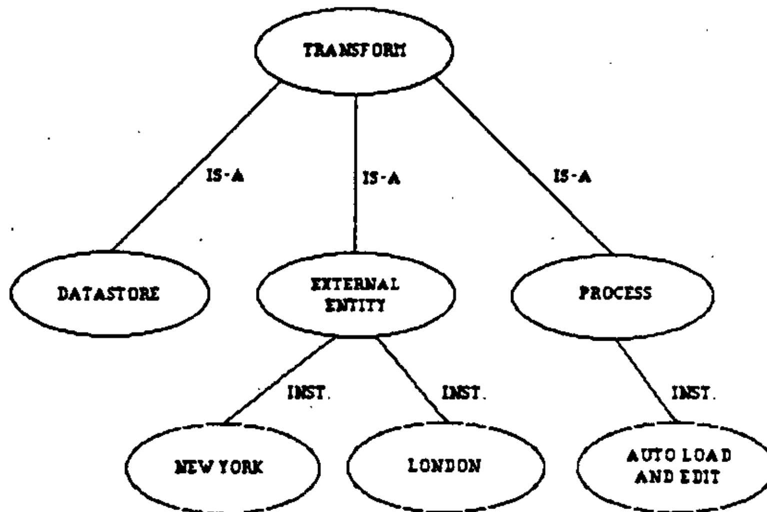


Figure 9

Initial Generalization Hierarchy

{identifier : London-direct-sales-invoices
type : dataflow
because__of : (London)
part__of : ( )
medium : magnetic tape
from : London
to : auto-load-and-edit}

Similarly, instances corresponding to other object types can be defined. Note, that the instance definitions have all the slots defined in their immediate type, as well as inheriting those of their supertypes.

This representation allows us to define data flow diagrams completely. It is also possible to perform "syntactic" consistency checks using information in the hierarchy. As a simple example, if a bubble has no inputs, it must be removed or new inputs must be defined. However, application-specific information is not maintained in this representation. For instance, if London invoices become "formatted", ramifications of this change cannot be assessed using the knowledge in the hierarchy alone (i.e., without using the "because-of" slot). To reason about such situations, additional knowledge structures are required, which we describe below.

REPRESENTING RATIONALES

Design decisions at the Structured Analysis level define bubble and dataflow objects. The *rationale* or *justification* of a decision consists, in turn, of other decisions. To illustrate, consider figure 10 which shows a network of dependencies among a few of the dataflows and bubbles considered so far. Specifically, the auto-load-and-edit is justified by the existence of New York and London invoices, which form its "set of support" (Doyle, 1978) or the cumulative reason for its existence. The convert operation is justified because London sales invoices are not formatted correctly. Similar dependencies can be identified for other decisions.

The complete dependency network corresponding to a design may be viewed as incorporating the overall *purpose* of a set of design decisions. The general form of a dependency is:

(<decision> <justification>)

where <decision> and <justification> are both object instances. In REMAP, each design object maintains a cumulative set of justifications in its *because-of* slot that constitutes its set of support.

In order to demonstrate the usefulness of this dependency network, let us reconsider the first scenario where the London invoices become formatted. In this case, the convert operation in no longer required since its *essential* support elements have been eliminated. Similarly, in the second scenario where the London office does not sell fuels anymore, no more invoices are generated from London. Again, no conversion operation is required. However, the auto load and edit operation is still required because New York invoices are still to be processed.

In general, an existing dependency network such as the one in figure 10 can be used to assess certain ramifications of a change, a process commonly referred to as *belief maintenance* (Doyle, 1978). In the above example, conversion is *not* required for London invoices. However, the dependency network does not indicate how these invoices *should* be treated because this knowledge is not expressed in the network. In order to assess the complete repercussions of the change, additional knowledge of a more general nature is required. For example, to realize that formatted London invoices should be treated like New York invoices (and should proceed directly for verification), it is necessary to know that *in general* formatted invoices are verified directly. This knowledge can then be used to reason about all object instances corresponding to formatted invoices.

RULE FORMATION

Dependency information as indicated in figure 10 is represented in terms of *object instances*. For example, the auto-load-and-edit (bubble 1) is justified by the two kinds of dataflow objects originating from London. An object type corresponding to this invoice dataflow might have slots such as data, amount, or office originating the invoice. However, not all slots are relevant to the justification. For example, the auto-load-and-edit is performed because the invoices are computerized, regardless of their other features. If the system is to be able to learn anything from existing designs, it must also have access to the general *rules on which the dependencies have been based*. In effect, the rules differentiate the important features of the relationship from the incidental.

REMAP allows the designer or user to generalize specific dependencies to design rules *during* the process of system analysis and design. This requires articulation of the justifications for choices, as well as of the general basis for the justifications. A more crucial issue however, is what *form* these rules might take.

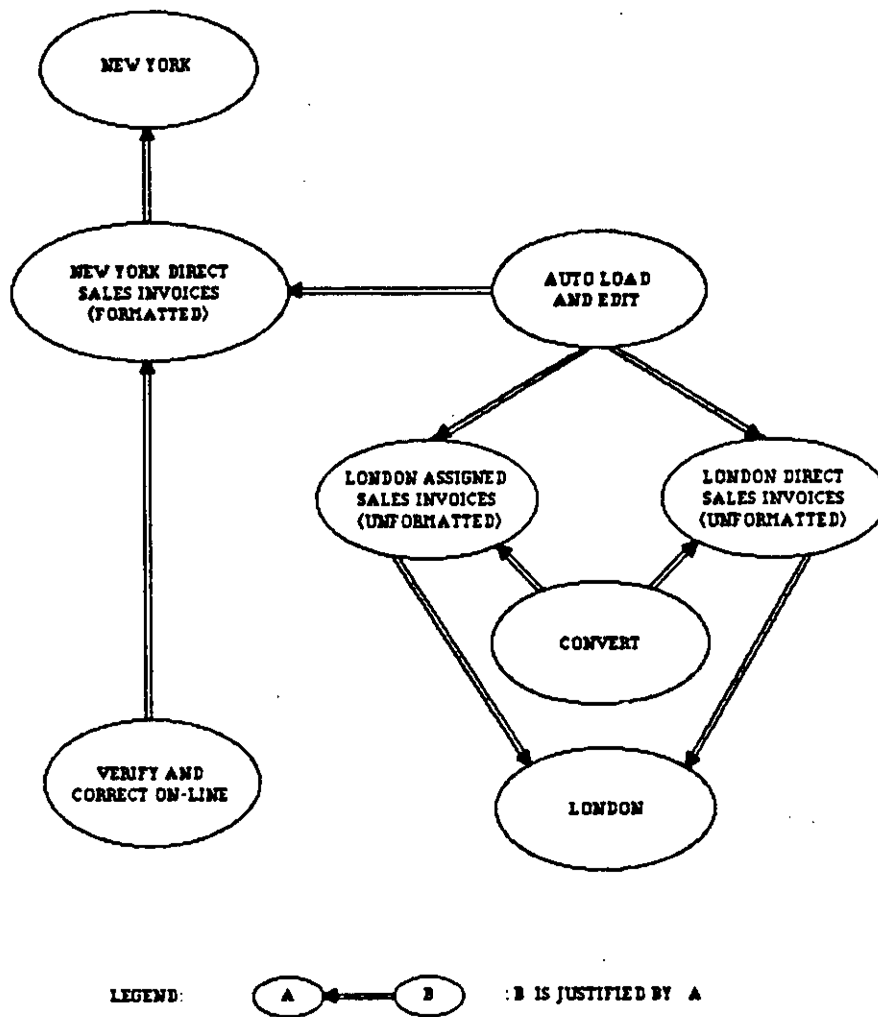


Figure 10

A Dependency Network

On the other hand, the rule can be expressed in terms of objects and their slot values, for example:

{dataflow
medium: computerized} ==> verify on line

{dataflow
medium: paper} ==> perform conversion

If the medium slot has not been defined before, the type definition of dataflow can first be extended to include it. Nevertheless, there is a major problem with this scheme. Recall that so far, the generalization hierarchy for dataflows is extremely shallow including only one type, namely the dataflow (cf. figure 9). Adding additional slots for each rule will soon yield very complex object types. In looking at the different invoices—which are instances of type dataflow—it is apparent that *different* attributes are relevant in describing the various instances. For example, paper invoices might be distinguished by their *color*, an attribute that is irrelevant for describing computerized invoices. Thus, most slots in the extended dataflow type definition would remain unfilled for many objects.

This situation can be expected to occur in the early stages of the system analysis process, when the system is still unfamiliar with the application area. New design decisions could be added and instantiated as instances of an existing type although they differ qualitatively from other instances, and might therefore be better off described in terms of a different bundle of attributes.

When instances vary sufficiently, it is an indication that the generalization hierarchy must be extended to include more specific subtypes. For example, extending the generalization hierarchy in figure 9 would involve creating two new types, namely paper-invoices and computerized-invoices and re-classifying the existing instances in light of this new classification. Further, computerized-invoices can then be broken down into magnetic-tape-invoices and on-line-invoices if appropriate.[2] The reconfigured generalization hierarchy would then appear as in figure 11, and in contrast to the rule representation above, the rule could then be stated in terms of the newly defined object types.

To illustrate, such rules might appear as:

{computerized-invoices} ==> perform auto-load-and-edit

{paper-invoices} ==> perform manual-add-and-edit

It should be possible to use these rule structures in two ways. First, if an operation such as auto-load-and-edit is part of a design and has one or more computerized inputs coming into it, these should be added automatically to the operation's set of support. Second, if no such inputs are in the design, the rule can be used to compare "expected" reasons for the operation to the justifications provided by the user, or to suggest changes in designs that appear "inconsistent" with the knowledge in the rules.[3]

OVERALL CONTROL STRUCTURE

In order to incorporate new knowledge and to reason about user critiques, the model requires an overall control structure that enables it to switch among design support and knowledge acquisition modes. Figure 12 provides a high-level transition network representation of the main modes.

The *add* mode is the usual starting point for a new system. The designer can add a set of proposed new design objects and their associated dependencies. The *belief maintenance* mode is responsible for checking the consistency of proposed changes with respect to existing object types and rules. The *learning* mode interacts with the user in order to establish a generalization of dependencies that are not derivable from existing rules, possibly adding new rules and specifying new object types. The system then moves into the belief maintenance mode in order to check the compatibility and consequences of the newly acquired knowledge.

If there is an existing design to be improved, or reused for another system, the system will start in the *critique* mode. Here, the designer may want to change or add to certain parts of the design. Again, feasibility and possible learning opportunities induced by the change can be studied in the belief maintenance and learning modes. The interaction of these components of the REMAP architecture is described below in "Structured English."

Add-mode:
1. DOWHILE user is entering object instances.
2. Accept object instances.
3. IF enabling conditions of a rule are satisfied by instances
   THEN 3a. Create dependencies generated by rule.
        3b. Invoke belief maintenance.
   ELSE 3c. Accept dependency.
        3d. Invoke Learn-mode

Learn-mode:
1. Extract essential features (slot values) of objects.
2. IF slot value is an object instance
   THEN 2a. Note its type
   ELSE 2b. IF needed slot does not exist
            THEN Create-new-type-mode.
3. Propose generalization (rule) in terms of the identified or defined types.

Create-new-type-mode:
1. Record context (slot values) of object instance.
2. Define new data type corresponding to relevant slot of this instance. Establish an IS-A link to parent-of of the object instance.
3. Create a new instance of the new data type.
4. Assign slot values to the new instance corresponding to the old instance.
5. Destroy the old object instance.

Critique-mode:
1. Accept user critique in the form of negation to existing decision, or addition to design.
2. IF negation
   THEN invoke belief maintenance
   ELSE invoke Add-mode.

Relationship to Previous Work

The REMAP concept attempts to integrate the abstraction concepts of life-cycle methods with the support for user critiques provided by prototypes. It is therefore appropriate to briefly point out the capabilities and limitations of each of these parent areas, as compared with the REMAP approach.

Probably the most advanced of the life-cycle methods are the Structured Methodologies. They offer semi-formal

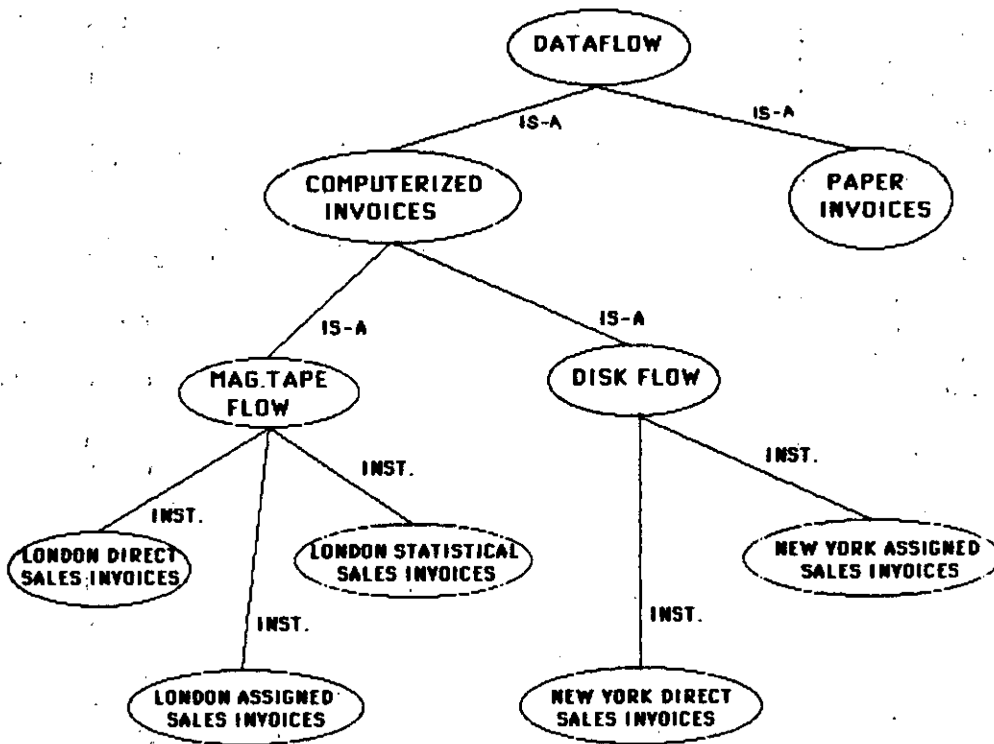


Figure 11

Reconfigured Generalization Hierarchy

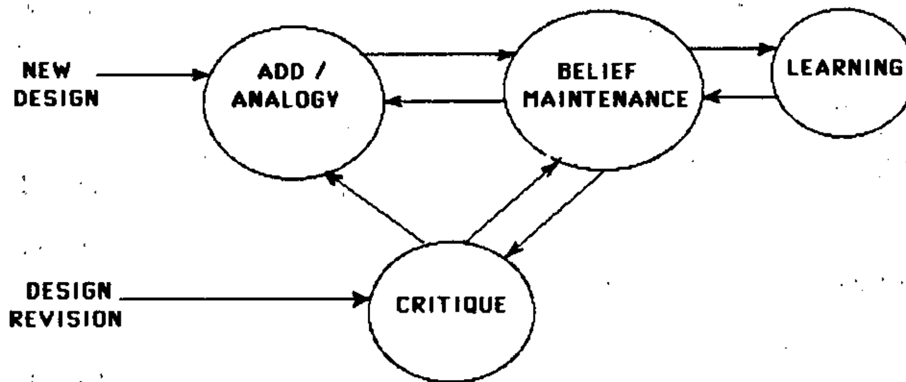


Figure 12

State Transition Network

representational tools (data flow diagrams, data dictionaries, HIPO's, etc.) for top-down strategies in each of the life-cycle stages (Zachman, 1982; DeMarco, 1978; Gane and Sarson, 1979; Yourdon and Constantine, 1978, Orr, 1981). These methodologies were developed in the late 1970's as a generalization of the earlier work on structured programming.

If sufficient time is available for a careful design, life-cycle methodologies result in well-documented original designs from which the programs are constructed. However, subsequent modifications are typically documented only at the program level whereas the design documents remain unchanged. After a few such changes, the program bears little resemblance to the original design. As a response to this problem, some researchers have proposed preserving a computer-based representation of the design. For example, PLEXSYS (Konsynski et. al., 1984; Kotteman and Konsynski, 1984) uses a hierarchy of so-called "dynamic metasystems" to describe designs and detect inconsistencies between the existing design and proposed changes.

Another practical response to the design maintenance problem has been the introduction of design and programming standards in most large organizations. Such standards include naming conventions, design methodologies, structured programming rules, and documentation guidelines. They could, in principle, serve as a knowledge structure for supporting designers and programmers (Jarke and Shalev, 1984) but are currently applied manually, as guidelines for programmers and designers or as evaluation tools for supervisors. It may be difficult to define the set of required knowledge (and thus standards) in advance since requirements and design strategies frequently evolve over time.

None of these improvements adequately address fundamental criticisms voiced against life-cycle methods by the advocates of prototyping. Since they involve a long development time frame, working systems are available for user critique only at a late stage when large parts of the design have been completed and user feedback becomes ineffective (McCracken, 1980; Martin, 1982).

Here lies a major advantage of the prototyping approach (Jenkins, 1983). However, without an appropriate environment, prototyping can result in very brittle programs, especially in complex systems in which the consequences of a change cannot be completely understood by a single user or designer. As a consequence, recent development efforts have attempted to provide a *workbench environment* (Reiner et al., 1984) which is equipped with high level knowledge that can be used to reason about the object domain.

In the general systems arena, Kotteman and Konsynski (1984) have taken the approach of representing application-specific knowledge in terms of an "axiomatic" model that can propagate certain types of changes to the object level where design decisions are represented. This approach is similar in spirit to Davis' (1979) idea of using "meta models" to maintain and reason about object level knowledge contained in the MYCIN system (Shortliffe, 1976). Several other knowledge base management components of AI systems have been structured along similar lines.

While this approach has proven successful in situations where the scope of applications known to the meta-model can be defined in advance, it has fundamental limitations if the application domain is not known a priori. Under such circumstances, the high level model, even in definable, may become general to the point of missing the subtleties involved in an application area. What is needed instead, is a mechanism by which the high level model itself can be synthesized on the basis of experience in the application area. Consequently, REMAP follows an "open systems" approach (Hewitt, 1985) that begins by representing knowledge about relationships among instances in a domain in terms of dependencies, and generalizes some of these into a growing corpus of rules. In this way, the process knowledge involved in building an application can be used for incremental modification of designs, and where possible, to acquire knowledge in terms of application specific rules.

Methodologically, our approach has much in common with the Programmer's Apprentice (PA) project (Shrobe, 1979; Waters, 1982; Rich, 1984). The PA is an intelligent system that is designed to assist expert programmers with the maintenance of large programs. Like REMAP, the PA uses a dependency network of choices in order to represent and reason about evolving programs. However, there are two important differences. Our focus is on the more abstract parts of the design as opposed to the level of coding. More importantly, because of the diversity of applications, we are unable to assume a fixed library of "cliches" or programming constructs, but must build up this knowledge on the basis of application-specific designs. However, once our system has constructed and organized a library of cliches, they could be used to reason about "analogous" situations in a similar manner as the PA.

Conclusions

Some key aspects of the REMAP architecture have been incorporated in a small system intended to test their feasibility. The system contains an implementaion of the object type hierarchy and an initial knowledge base about data flow diagrams. Knowledge is represented using FLAVORS (Moon and Weinreb, 1981), a LISP-based utility that supports object-oriented programming. The

current implementation has the capability to accept data flow diagram object instances, to generalize dependencies to rules, and to expand the generalization hierarchy.

The approach proposed in this paper suggests a novel way of thinking about systems evolution which emphasizes the designer's assumptions and justifications, rather than generally valid "meta-theories" of design. This reorientation is of particular importance in the presence of multiple designers since many apparent "logical contradictions" may arise as a result of different *perspectives*, each based on a different set of assumptions.

From a practical viewpoint, the emphasis on design changes is of particular importance since it is estimated that at least 50% and probably as much as 70% of software costs go into maintenance. Yet, problems of design evolution have not been adequately addressed by previous methodologies, whereas they constitute the focus of our approach. The work reported here is considered a first step towards a process-oriented design environment which is expected to have important applications in at least three areas.

First, the prototyping method of systems development is enhanced by a learning component that prevents the repetition of design errors and supports a better formal understanding of the system's domain. Second, the undesirable practice of just updating program documentaion in the maintenance phase of the software life cycle is replaced by a methodology for maintaining consistent designs; furthermore, the method also provides guidance in the propagation of proposed changes.

Finally, the analogy-based reasoning component of the method supports the reuse of code and designs in systems that are similar to existing ones. It also provides the designer of such systems with access to the rationales for the original design, thus permitting the encapsulation of required design differences and the identification of suitable alternatives. This controlled "cloning" capability is particularly valuable in organizations that have to construct a large number of functionally similar systems for different divisions. If process knowledge is not maintained automatically, such organizations have to rely on the experience and loyalty of a few key individuals.

NOTES

[1]This illustrates the "non-uniform" nature of dataflow diagram entities, that is, relationships among "unconnected" entities, and the design consequences that can emerge due to changes to them.

[2]This raises the following question: how might the program differentiate among situations where the generalization hierarchy should be extended versus those where little is to be gained by extension? Although we have yet to address this question adequately, it appears that a reasonable heuristic for deciding when to extend the generalization might be based on the need for additional slots to differentiate newly defined object instances.

[3]This assumes that the rule is "correct". An existing rule that turns out to be inaccurate, leads to a "contradiction" in which case the rule can be discarded by the belief maintenance machinery, or refined interactively.

REFERENCES

Balzer, R., Dyer, D., Fehling., and Saunders., 1982. Specification-based computing environment, *Proceedings 8th Very Large Data Base Conference*, Mexico City, pp. 273-279.

CGI Systems Inc., 1984. Presenting PACBASE. Systems Development Software from CGI, Pearl River, New York.

Davis, Randall., 1979. Interactive Transfer of Expertise—Acquisition of new inference rules, *Artificial Intelligence*, No. 4.

De Marco, T., 1978. *Structured Analysis and System Specification*, Yourdon Press, New York.

Dhar, V., and Quayle, C., 1985. An Approach to Dependency Directed Backtracking Using Domain Specific Knowledge, in *Proceedings of the 9th Joint International Conference on Artificial Intelligence (IJCAI)*, Los Angeles, California.

Doyle, Jon., 1978. A Truth Maintenance System, AI Laboratory Memo 521, MIT.

Gane, C., and Sarson, T., 1979. *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall.

Greenspan, S., 1984. Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition, Ph.D. Thesis, Technical Report CRSG-155, University of Toronto.

Hewitt, Carl., 1985. The Challenge of Open Systems, *BYTE Magazine*, April.

Jarke, M., and Shalev, J., 1984. A Database Architecture for Supporting Business Transactions, *Journal of Management Information Systems 1*, 1, pp. 63-80.

Jenkins, Milton A., 1983. Prototyping: A Methodology for the Design and Development of Application Systems, Working Paper Number 227, Graduate School of Business, Indiana University, April 1983.

Konsynski, B., Kotteman, J., Nunamaker, J., and Stott, J., 1984. PLEXSYS-84: An Integrated Development Environment for Information Systems, *Journal of Management Information Systems*, volume 1, number 3, Winter 1984-85.

Kotteman, J.E.., and Konsynski, B.R., 1984. Dynamic Metasystems for Information Systems Development, *Proceedings of the 5th International Confer-*

ence on Information Systems, Tucson, Arizona, pp. 187-204.

Martin, J., 1982. *Application Development Without Programmers*, Prentice-Hall.

McAllester, D., 1982. Reasoning Utility Package, AI Laboratory Memo 667.

McCracken, D.D., 1980. A Maverick Approach to Systems Analysis and Design, *Conference on Systems Analysis and Design: Foundation for the 1980s*.

Michie., 1982. The State of the Art in Machine Learning, *Introductory Readings in Expert Systems, D. Michie (ed.,)* Gordon and Breach, United Kingdom.

Moon, David, and Weinreb, Daniel., 1981. Lisp Machine Manual, MIT AI Lab.

Newell, Allen., and Rychener, Mike., 1978. An Instructible Production System, in F. Hayes-Roth and D. Waterman (eds.), *Pattern Directed Inference Systems*, Academic Press.

Orr, K., 1981. *Structured Requirements Specfication*, Orr and Associates.

Protsko, L.B., Sorenson, P.G.., and Tremblay, J.P., 1984. Automatic Generation of Data Flow Diagrams from a Requirements Specification Language, *Proceedings 5th International Conference on Information Systems*, Tucson, Arizona, pp. 157-171.

Reiner, D., Brodie, M., Brown, G., Fridel, M., Kramlich, D., Lehman, J., and Rosenthal, A., 1984. The Database Design and Evaluation Workbench (DDEW) Project at CCA, *Database Engineering*, volume 7, number 4, December 1984.

Rich, Charles., 1984. A Formal Representation for Plans in the Programmers Apprentice, in Brodie, M.L., Mylopolous, J., and Schmidt, J.W. (eds.), *On Conceptual Modeling*, Springer, pp. 239-269.

Shrobe, Howard., 1979. Dependency directed reasoning for complex program understanding, Ph.D. Dissertation, MIT.

Shortliffe, E.H., 1976. *Computer-Based Medical Consultations: MYCIN*. New York: American Elsevier.

Simon, H.A., 1981. *The Sciences of the Artificial*, 2nd ed., MIT Press, Cambridge, Massachusetts.

Stallman, Richard, and Sussman, Gerald., 1977. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence*, volume 9, Number 2, pp. 135-196.

Waters, Richard., 1982. The programmer's apprentice: knowledge based program editing, *IEEE Transactions on software engineering*, number 1.

Winston, P.H., 1982. Learning Structural Descriptions from Examples, in *The Psychology of Computer Vision, P.H. Winston (ed.,)* McGraw Hill, New York.

Winston, P.H., 1979. Learning and Reasoning by Analogy, *CACM*, volume 23, Number 12, pp. 689-703.

Yourdan, E., and Constantine, L.L., 1978. *Structured Design*, Yourdon Press, New York.

An Investigation of the "Tables Versus Graphs" Controversy in a Learning Environment

Gerardine DeSanctis
and
Sirkka L. Jarvenpaa

University of Minnesota
Management Sciences Department
271 19th Avenue South
Minneapolis, MN 55455
(612) 373-5211

ABSTRACT

The study of computer graphics as decision aids has become popular among MIS researchers in the last several years. However, this area of research, like many others in management information systems, has been plagued with methodological problems and contradictory findings. In light of these difficulties, the current study examined the "tables versus graphs" controversy within a learning environment. Seventy-five MBA students were exposed to one of three experimental treatments and asked to develop financial forecasts for fictitious companies over five experimental trials. Following their forecasts for each firm, participants were provided with feedback on the quality of their decisions. The information presentation treatments were as follows: (1) traditional spreadsheet (tabular), (2) graphs using "standard" scaling, and (3) graphs using "nonstandard" scaling. Results suggest that, although graphics may initially demonstrate no advantage over tables, they do show an advantage if decision makers are repeatedly exposed to the novel format and given feedback on their performance. Learning will occur even when improper scaling is used. The implication is that the effectiveness of graphics as decision aids depends on practice. Researchers are encouraged to employ repeated measures, or longitudinal, designs when examining the tables-versus-graphs controversy.

Introduction

How to best display data to decision makers has been a concern to MIS researchers since Mason and Mitroff (1973) first noted the importance of "presentation mode" in the design of information systems. A large portion of this research effort has centered on comparing the relative effectiveness of tables and graphs for the support of problem solving activities in business settings. Interest in "tables versus graphs" comparisons has intensified during the past few years as sophisticated, easy-to-use graphics technology has become incorporated into decision support systems. The underlying assumption in these studies is that graphics should facilitate clearer perception of data relationships and trends over tables.

The empirical research dealing with the effectiveness of graphs as decision aids has been quite controversial. Several studies have found graphs to be easier to interpret than tables; others have found the reverse; and still others report no difference in interpretation accuracy for the two display methods. Similar conflicting results have been found when graphs and tables are compared for their effects on interpretation speed, user preference, and decision confidence (see Ives, 1982; MacDonald-Ross, 1977). Of a total of 7 studies dealing with the impact of graphics on decision quality, only one reports graphs to be superior to tables; 3 conclude that tables are superior to graphs, and 3 have found no difference between the two formats (see DeSanctis, 1984).

Why is it that computer graphics are not proving to be more useful as tools for supporting decision making? Several investigators who have found graphs to be fairly ineffective in improving decision quality have postulated that learning must occur before graphical output becomes meaningful to people (e.g., Lusk & Kersnick, 1979; Vernon, 1946). Business data traditionally has been displayed in tabular form. Consequently, decision makers simply lack the experience needed to properly interpret novel formats. This argument implies that practice in viewing graphs might improve their meaningfulness to