1985

Occidental Versus Oriental I.S. Professionals' Perceptions on Key Factors for Motivation

J. Daniel Couger
University of Colorado, Colorado Springs

Juzar Motiwalla
Institute of Systems Science

Recommended Citation

Couger, J. Daniel and Motiwalla, Juzar, "Occidental Versus Oriental I.S. Professionals' Perceptions on Key Factors for Motivation"
(1985). *ICIS 1985 Proceedings*. 11.
http://aisel.aisnet.org/icis1985/11

Occidental Versus Oriental I.S. Professionals' Perceptions on Key Factors for Motivation

J. Daniel Couger
Distinguished Professor
Computer and Management Science
University of Colorado, Colorado Springs

Juzar Motiwalla
Director
Institute of Systems Science
Singapore

ABSTRACT

A comparison of perceptions of analysts and programmers in Singapore versus the United States identifies many more similarities than dissimilarities. The Singapore sample consisted of 1,179 persons (31% of the entire I.S. population). The U.S. data base is comprised of more than 8,000 persons. Similarities were statistically significant on 9 of 12 factors compared for system analysts and on five of 13 factors compared for programmers. On six of the eight factors where programmers are significantly different, changes underway have a strong likelihood of eliminating those differences. On the most important factor that distinguishes I.S. professionals in the U.S from other U.S. professionals, individual growth need strength (GNS), Singaporean I.S. professionals are not significantly different from their U.S. counterparts. This is the first of six studies comparing American I.S. professionals to I.S. professionals in Singapore, that they perceive motivational issues much like their American counterparts.

Introduction

In January and February of 1985 a motivation survey was conducted of computer professionals in Singapore. Responses were received from 1,179 persons representing 32 organizations from government and industry. Since this constitutes over 31 percent of the total population of approximately 3,800 computer professionals[1], it is an excellent sample. The Computer Board of Singapore took on the responsibility of insuring that the survey was representative. A major possibility of bias was eliminated by taking a 100 percent sample of representative organizations rather than a partial sample of all Singaporean organizations. Careful selection of organizations insured representativeness.

Survey responsiveness was aided by the Singaporean government objective to become the leading software producing country in Southeast Asia. The National Computer Board requested the survey and development of Singapore motivation norms for computer professionals to enable a comparison to the U.S. norms, as an aid in determining if software objectives were realistic.

An earlier study by Couger and Zawacki[2] compared motivational data from seven countries in the far east: Singapore, Taiwan, Korea, Hong Kong, Thailand, Malaysia and Indonesia. The research indicated more similarities than dissimilarities between computer professionals in these countries. It also identified a number of similarities with U.S. computer professionals. However, the data points were too few to develop motivation norms for each country, as has been done in the U.S. by Couger and Zawacki[3].

The massive sample in Singapore enables such a comparison. The objective was identification of significant differences between the computer populations of the two countries. In the conclusions to the paper, we will discuss other factors which relate to the question of the degree to which Singapore is representative of the Southeast Asian motivational environment.

Over the years, some interesting results have been produced by research on the differences in Western versus Eastern hemisphere management styles. The earlier, better-known works were *The Silent Language*, by Hall[4]

and *Natural Symbols: Exploration in Cosmology* by Douglas[5]. In recent years Hofstede is probably the best-known researcher in this area. He has published a series of papers in *Academy of Management Review* on this research. For example, in "Motivation, Leadership and Organization: Do American Theories Apply Abroad,"[6], he identifies far more differences than similarities in managerial styles as a result of comparing data from a variety of countries. Despite the body of evidence to the contrary on management styles, it is our hypothesis that perceptions on motivation of computer professionals have little cultural dependence. Singapore is the first country where this hypothesis has been tested.

METHODOLOGY

The survey instrument used in this research project was the Couger-Zawacki version of the Job Diagnostic Survey (JDS) developed by Hackman and Oldham. The validity and reliability of the JDS was substantiated in 1975[7]. While preserving the integrity of the generic instrument, Couger and Zawacki added sections peculiar to the I.S. field. The modified instrument, called the JDS/DP, was validated in 1977 [(3, Appendix V)]. The Couger-Zawacki, data base is quite rich, containing information on more than 8,000 U.S. computer personnel and over 2,000 foreign computer personnel. The body of research using this data base has produced motivation norms for 18 job types within the computer field.

In addition to a tailor-made program to compute and analyze survey results, the authors utilized the SPSS package. These analytical tools were used to compare the two populations against 27 survey variables. Definitions of the variables whose results are reported in this paper are provided in Appendix II.

Analysis of Results

COMPARING SYSTEM ANALYSTS

The Singaporean system analysts have more in common with their U.S. counterparts than differences. Table I provides a comparison of results on the five core job dimensions, determined by Turner and Lawrence[8] as key to motivation. On two of the variables the difference statistically significant ($p_s.001$). For the other three core job dimensions, the differernces are not statistically significant. However, the responses on the two variables cause the MPS (Motivating Potential Score) to be significantly different; MPS is comprised by the five core job dimensions.

However, the key variable on determining difference between the two populations is GNS (Growth Need Strength). GNS is the need for personal accomplishment and for learning and developing beyond a person's pres-

ent level of knowledge and skills. The U.S. norms show that GNS for system analysts is highest of any occupation ever measured, higher than any of the 500 occupations in the Hackman-Oldham data base. The survey reveals that GNS for Singaporean analysts is not significantly different than that of their U.S. counterparts.

The Singapore survey also shows that system analysts perceive a need for a richer job. Figure 1 depicts that situation. In Figure 1 the U.S. norms are reflected by the horizontal bar at the top of the chart, designed to provide a common scale for the means for the two variables. Thus, GNS and MPS for the U.S. (left side of the figure) align exactly with the horizontal bar. (MPS is a 3-digit figure because it is computed from the core job dimensions.) Singapore norms are shown on the right side of the figure. While GNS for Singaporean system analysts is not significantly different from U.S. GNS; MPS is.

Table II indicated another area in common between the two countries—problems in supervisory feedback. On the two variables feedback in general and feedback specific to goal accomplishment, system analysts in both countries rate their supervisors low (Table II). Nor are differences significant on the two satisfaction variables: general satisfaction and supervisory satisfaction (Table III).

Table IV identifies another area of commonality even though the two norms *are* significantly different ($p_s.001$ level). Social need strength (SNS) of Singaporean analysts (4.83) is higher than that of their U.S. counterparts (4.25). Nevertheless, SNS of analysts in both countries appears to be significantly lower than that of other professionals. The database on U.S. computer personel, compared to the Hackman-Oldham data base on 500 other occupations, identifies this difference. Although there is no data base on other Singaporean occupations, discussions with leaders in that country suggests that Singaporean computer professionals also have much lower SNS than other occupations.

As to why Singaporean analysts (and programmers, also) have higher SNS than comparable U.S. professionals, we must also speculate. In the presentation of these results to more then 150 managers across the Singaporean computer industry, there was a consensus that Singaporeans place great emphasis on teamwork—from childhood on. This early life conditioning appears to raise their SNS above their U.S. counterparts whose conditioning tends to be more along the lines of individual accomplishment.

COMPARING PROGRAMMERS

Although there are differences between programmers in the two countries, an area of similarity is their perception on maintenance of existing applications. The Couger-

TABLE I
Comparison of U.S./Singapore System Analysts on Core Job Dimensions
(All responses on a scale of seven)

Core Job Dimensions	U.S. Norm	Singapore Norm
Skill Variety	5.55	5.22
Task Identity	5.35	5.42
Task Significance	5.75	5.37 *
Autonomy	5.30	5.25
Feedback From the Job	5.20	4.82 *
MPS	155	135 *

*Difference statistically significant at the $p < .001$ level.

TABLE II
Comparison of U.S./Singapore System Analysts on Feedback Variables

Variable	U.S. Norm	Singapore Norm
Feedback from Supervisor	3.95	3.91
Feedback of Goal Accomplishment	4.00	3.85

TABLE III
Comparison of U.S./Singapore System Analysts on Satisfaction Variables

Variable	U.S. Norm	Singapore Norm
General Satisfaction	5.10	4.88
Supervisory Satisfaction	4.65	4.54

TABLE IV
Comparison of U.S./Singapore System Analysts on Social Need Strength

Variable	U.S. Norm	Singapore Norm
Social Need Strength	4.25	4.83 *

*Difference statistically significant at the $p < .001$ level.
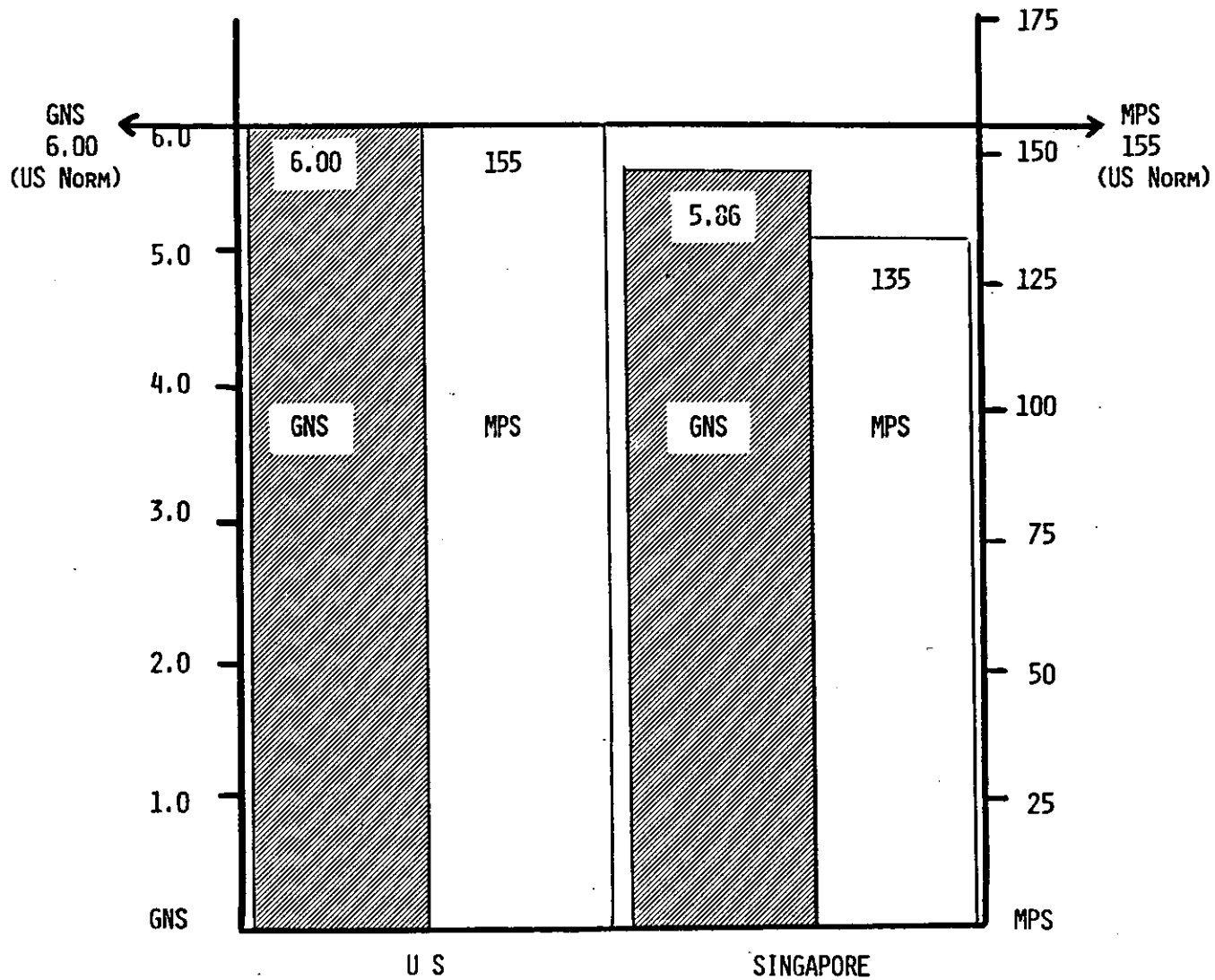
Figure 1

MPS vs GNS for System Analysts

Colter research[9] revealed that MPS of a programming job in the U.S. is highly negatively correlated with amount of maintenance performed. The research showed that MPS for personnel whose work is over 80% maintenance is only two-thirds that of personnel who spend twenty percent or less of their time on maintenance.

Figure 2 shows that Singaporeans have a similar perception. The curves for the two countries are parallel. The reason that the curves are not superimposed is that the norm for MPS for Singaporean programmers is less than that for U.S programmers.

One major difference between programmers in the two countries is shown in Table V. The norms for Singaporean programmers are significantly lower on all five core job dimensions. However, on the key discriminating variable, GNS, the norm for Singaporean programmers is not significantly lower than the U.S. norm. As a consequence, there is a serious mismatch between GNS and MPS. Singaporean programmers have a need for challenging jobs—just as high as their American counterparts. However, they perceive their jobs to be deficient in skill variety, task significance, task identity, autonomy and feedback from the job.

After seeing the survey results, IS managers attending the Singapore presentation generally agreed that they have underestimated the needs of programmers and that they should increase the scope of work. They concluded that both job enlargement and enrichment are needed. After these changes are made, a longitudinal resurvey is planned. With job enlargement/enrichment, it is expected that MPS will be raised to a level similar to that of U.S. programmers. These expectations appear reasonable, and, since GNS of the two populations is already statistically equivalent, the resurvey could be expected to indicate that programming norms of the two countries are as similar as norms for system analysts.

Like analysts, programmers responses for the two countries are not significantly different on the following variables: feedback on goal accomplishment, and supervisory satisfaction. Differences are significant for feedback from supervision and general satisfaction, reflecting the problem of mismatch of GNS and MPS.

Conclusions

The survey results reveal that Singaporean and U.S. analysts and programmers have more similarities than dissimilarities. Similarities were statistically significant on 9 of 12 factors compared for system analysts and on five of 13 factors compared for programmers (the additional factor for programmers was maintenance). On six of the eight factors where programmers are significantly

different, changes underway have a strong likelihood of eliminating those differences. On the most important factor that distinguishes I.S. professionals in the U.S. from other U.S. professionals, individual growth need strength, Singaporean I.S. professionals are not significantly different from their U.S. counterparts.

These similarities are even more surprising when the differences in demographics are considered. The analysis of demographic data reveals some differences between the two populations. Those differences will be summarized here from the table provided in Appendix II. Concerning age, 70 percent of the Singaporean respondents were less than 31 years of age, compared to 36 percent of the U.S. respondents. Seventy-one percent of Singaporean respondents obtained a BS degree or higher, compared to 57 percent of U.S. respondents. Seventy-two percent of Singaporean respondents have four years or less experience in the field, compared to 55 percent of the U.S. respondents. Forty-eight percent of the Singaporean respondents are female, compared to 39 percent of the U.S. respondents. In summary, Singaporean respondents are younger, less experienced, and better educated than their U.S. counterparts. In addition, only 52 percent are male, compared to 61 percent of U.S. respondents. However, the U.S. norms show an absence of significant correlations between GNS and age, education or sex.

On the other hand, Singapore may not be representative of Southeast Asia, despite the similarity in survey responses in the earlier survey of seven Southeast Asian countries. That survey was too small to make such projections. Singapore has some advantages over the other Southeast Asian countries. It has a relatively small population (less than three million persons) and geographic size (less than 240 square miles). It is a prosperous country (favorable balance of trade) and has been governed by a very popular political party for more than 20 years. All these factors facilitate its ability to obtain consensus on national goals, such as the objective to become a software leader. Government is employing considerable financial resources to underwrite this objective. Economic incentives for software companies are in place. But nowhere is the support better illustrated than in the educational area. Students have strong financial support at both the technical school and university level. In addition, a number of students have scholarships to study abroad and most of them are in Computer Science/I.S. programs in the U.S. The same applies to Ph.D. students. An ample supply of these well-supported students insures that the proper teaching cadre will continue to exist at the university level.

Finally, Singapore is the only Southeast Asian country where English is the standard educational language at all levels in the educational system. The ability to easily use U.S. journals, manuals, and books is highly advan-
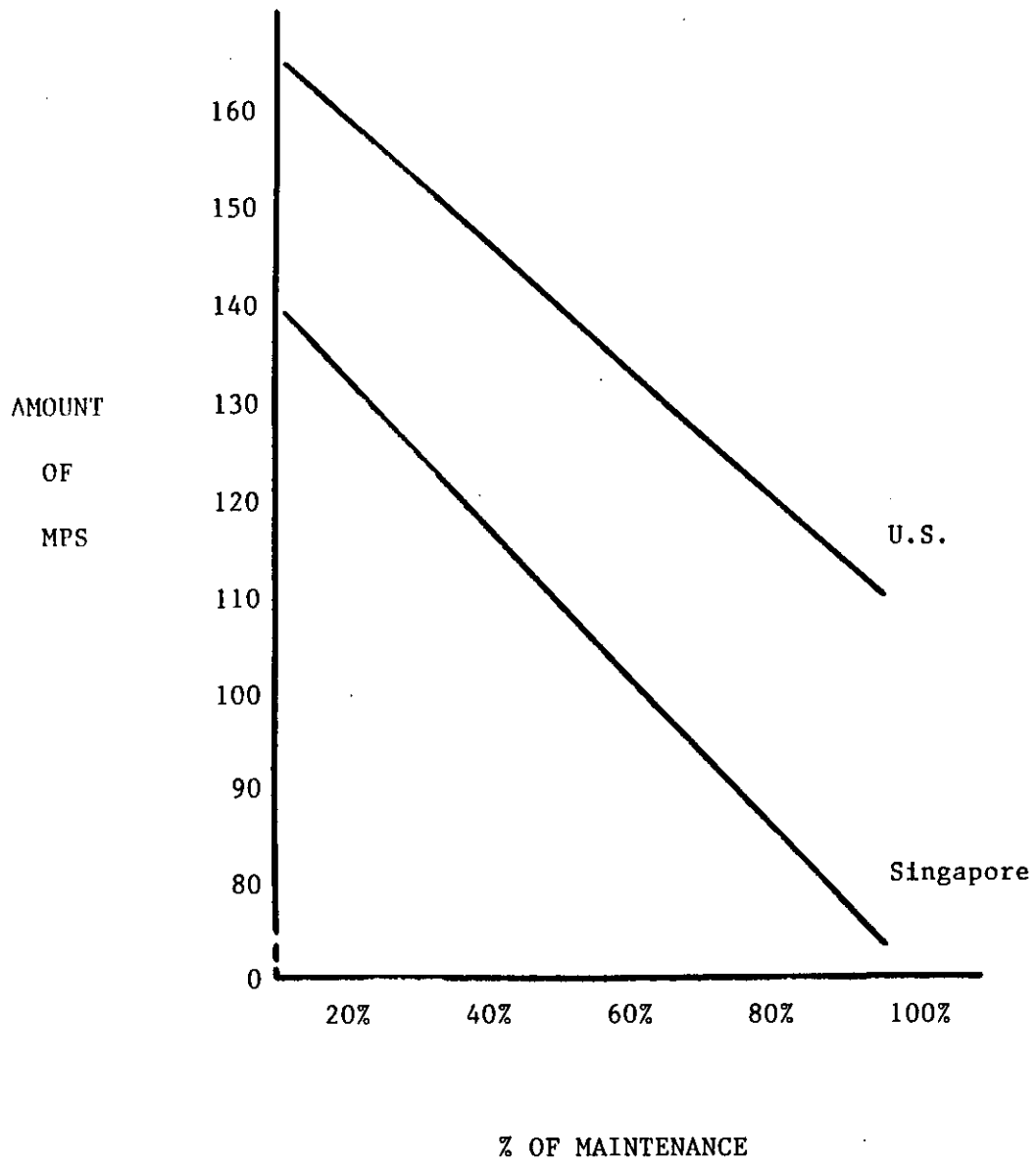
% OF MAINTENANCE

Figure 2

Effect of Percent of Time Spent on Maintenance
U.S. Versus Singapore Programmers

TABLE V
Comparison of U.S./Singapore Programmers on Core Job Dimensions

Core Job Dimensions	U.S. Norm	Singapore Norm
Skill Variety	5.25	4.33 *
Task Identity	5.00	4.52 *
Task Significance	5.45	4.90 *
Autonomy	5.15	4.57 *
Feedback From the Job	5.10	4.48 *
MPS	140	99

*Difference statistically significant at the p < .001 level.

tageous in achieving its software objectives. Communication with American counterparts is greatly facilitated, compared with other Southeast Asian countries. A number of U.S. firms have already located subsidiaries in Singapore and have begun subcontracting software development to Singaporean organizations.

So, while this paper's title is valid in its stated comparison of occidental to oriental motivational environments, Singapore may not be truly representative of either oriental or Southeast Asian motivational environments. Additional research of this type will be conducted in other Southeast Asian countries to clearly determine the answer to that question.

Nevertheless, this research project provides an interesting comparison of computer professionals in different cultures. It is the starting point for the Couger-Zawacki research project to compare motivational norms of computer professionals throughout the world.

BIBLIOGRAPHY

1. Summary of Findings of the Computer Manpower and Salary Survey, Singapore, National Computer Board, January, 1985.
2. Couger, J.D. and R.A. Zawacki, Comparison of SE Asia Computer Professionals on Key Motivational Variables, (UCCS Working Paper), Colorado Springs, 1984).
3. Couger, J.D. and R.A. Zawacki, Motivating and Managing Computer Personnel, New York, John Wiley and Sons, Inc., 1980.
4. Hall, E.T., The Silent Language, New York, Fawcett, 1959.
5. Douglas, M., Natural Symbols: Exploration in Cosmology, New York, Vintage, 1973.
6. Hofstede, G., "Motivation, Leadership and Organization," Organizational Dynamics, AMACOM, Vol. 9, No. 1, 1980, pp. 42-63.
7. Hackman, J.R. and G.R. Oldham, "Development of the Job Diagnostic Survey," Journal of Applied Psychology, Vol. 60, No. 2, 1975, pp. 159-170.
8. Turner, A.N. and P.R. Lawrence, Industrial Jobs and the Worker, Boston, Harvard Graduate School of Business Administration, 1965.
9. Couger, J.D. and Mel A. Colter, Maintenance Programming: Improved Productivity Through Motivation, Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985.

Appendix I
Definition of Survey Variables

1. Key Job Dimensions: Objective characteristics of the job itself.
   A. Skill Variety: The degree to which a job requires a variety of different activities in carrying out the work, which involve the use of a number of different skills and talents of the employee.
   B. Task Identity: The degree to which the job requires the completion of a "whole" and identifiable piece of work—i.e., doing a job from beginning to end with a visible outcome.
   C. Task Significance: The degree to which the job has a substantial impact on the lives or work of other people—whether in the immediate organization or in the external environment.
   D. Autonomy: The degree to which the job provides substantial freedom, independence, and discretion to the employee in scheduling his/

her work and in determining the procedures to be used in carrying it out.

D. *Feedback from the Job Itself*: The degree to which carrying out the work activities required by the job results in the employee obtaining information about the effectiveness of his or her performance.

2. *Satisfaction Measures*: The private, affective reactions or feelings an employee gets from working on his job.

A. *General Satisfaction*: An overall measure of the degree to which the employee is satisfied and happy in his or her work.

B. *Specific Satisfactions*: These scales tap several specific aspects of the employee's job satisfaction:

    B1. Pay satisfaction

    B2. Supervisory satisfaction

    B3. Satisfaction with co-workers

3. *Social Need Strength*: This is a measure of the degree to which the employee needs to interact with other employees.

4. *Goal Clarity and Accomplishment*: These scales measure the degree to which employees understand and accept organizational goals. Further, it taps into the employees' feelings about goal setting participation, goal difficulty, and feedback on goal accomplishment.

5. *Individual Growth Need Strength*: This scale measures the individual's need for personal accomplishment and for learning and developing beyond his/her present level of knowledge and skills.

6. *Motivating Potential Score*: A score reflecting the potential of a job for eliciting positive internal work motivation on the part of employees.

APPENDIX II

Demographics Comparison: U.S. versus Singapore Survey Participants

Demographic Category	U.S. Norm	Singapore Norm
		(Percentages)
Education		
High School	4	12
Some College	39	17
BS	50	64
MS	7	6
Ph.D.	-	1
Experience		
Below 1 year	18	22
1-4 years	37	50
5-8 years	22	16
9-12 years	13	5
13-16 years	5	5
Over 16 years	5	2
Age		
Below 31	36	70
31-40	42	28
41-50	15	2
Over 50	7	-
Sex		
Male	61	52
Female	39	48

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 accessible *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
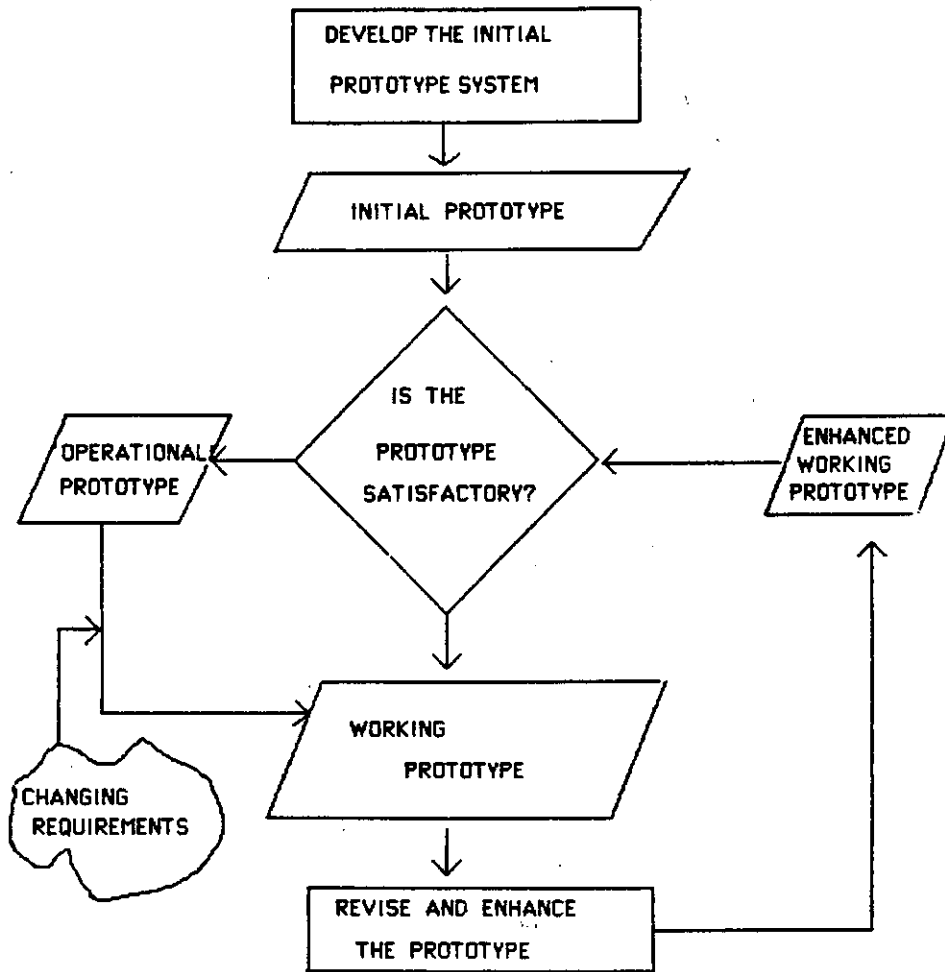
Figure 1

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
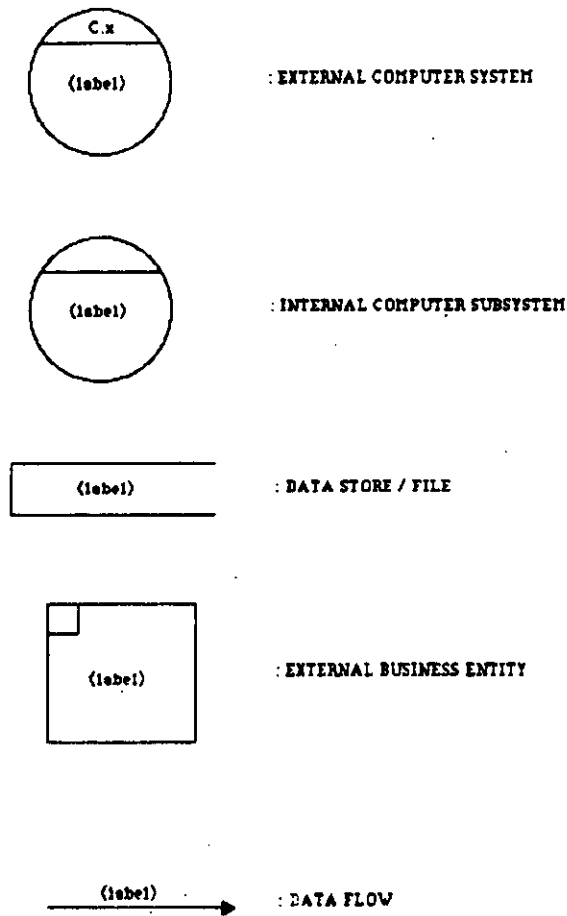
116

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.
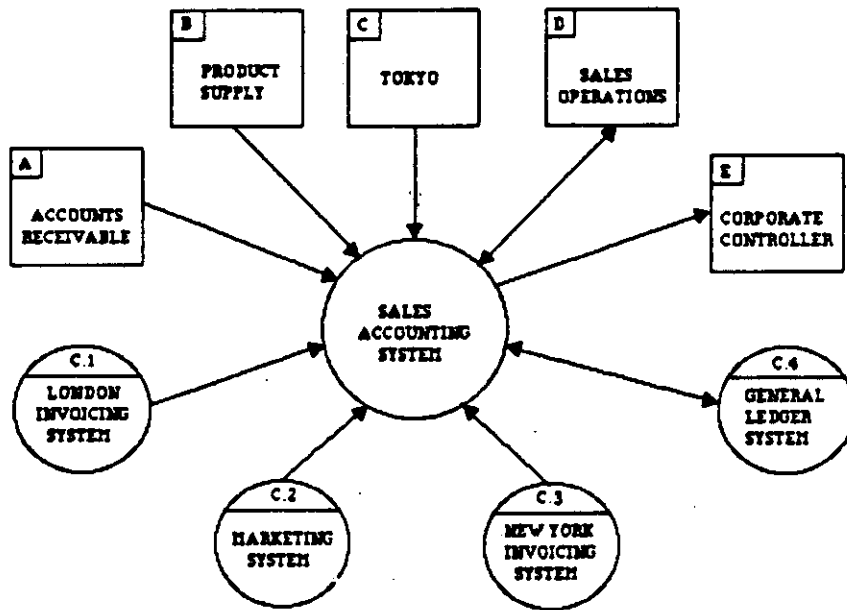
117

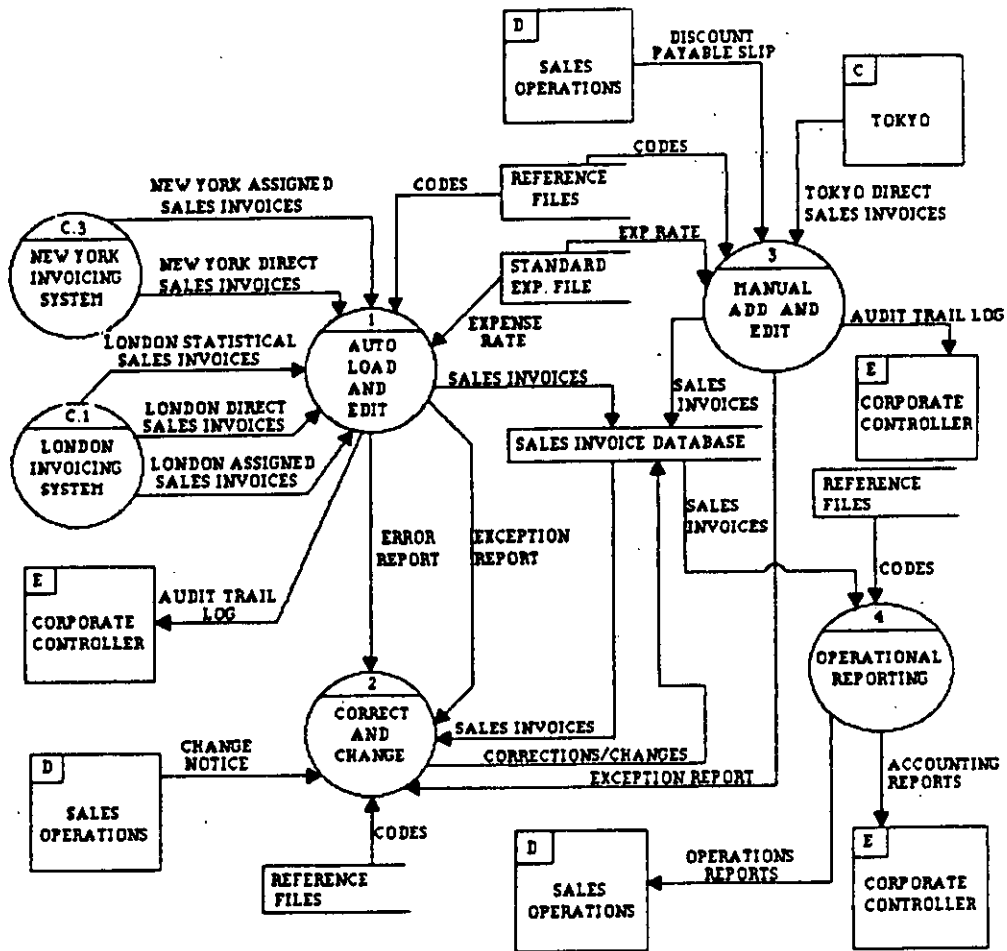**Figure 3**

Sales Accounting Systems Context Diagram
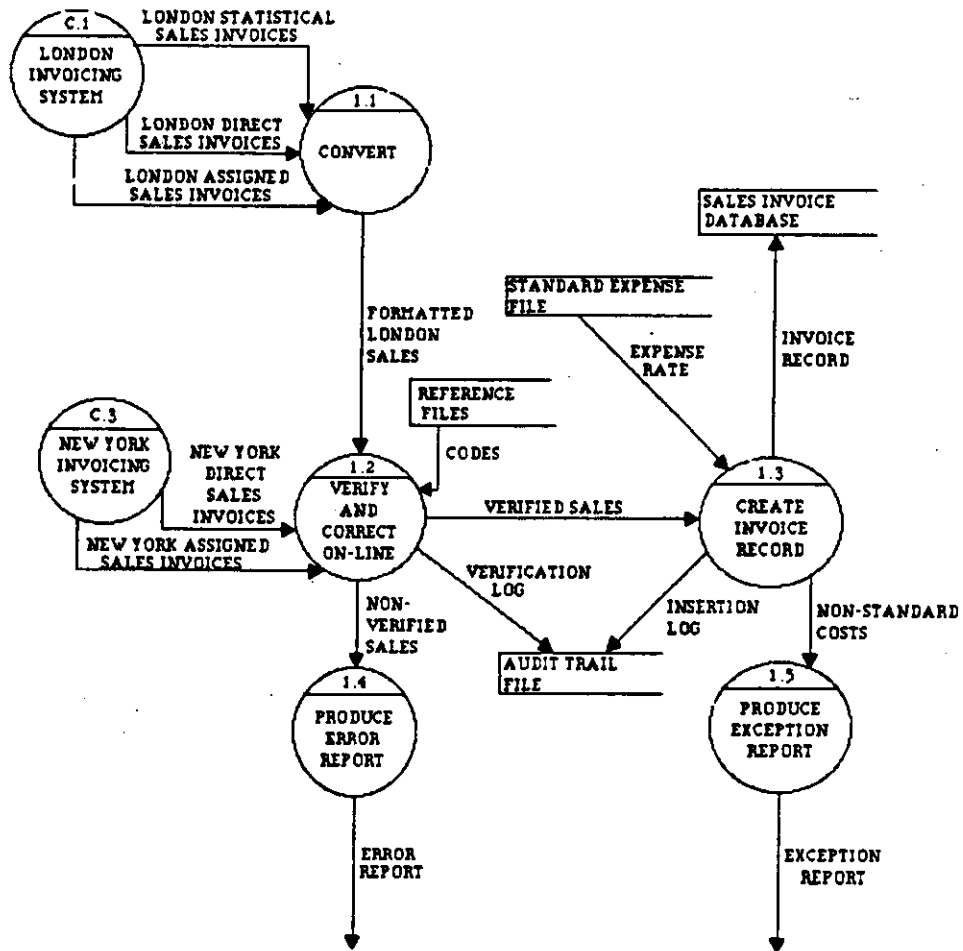
Figure 4

Fuels Sales (Initial)

119

Figure 5

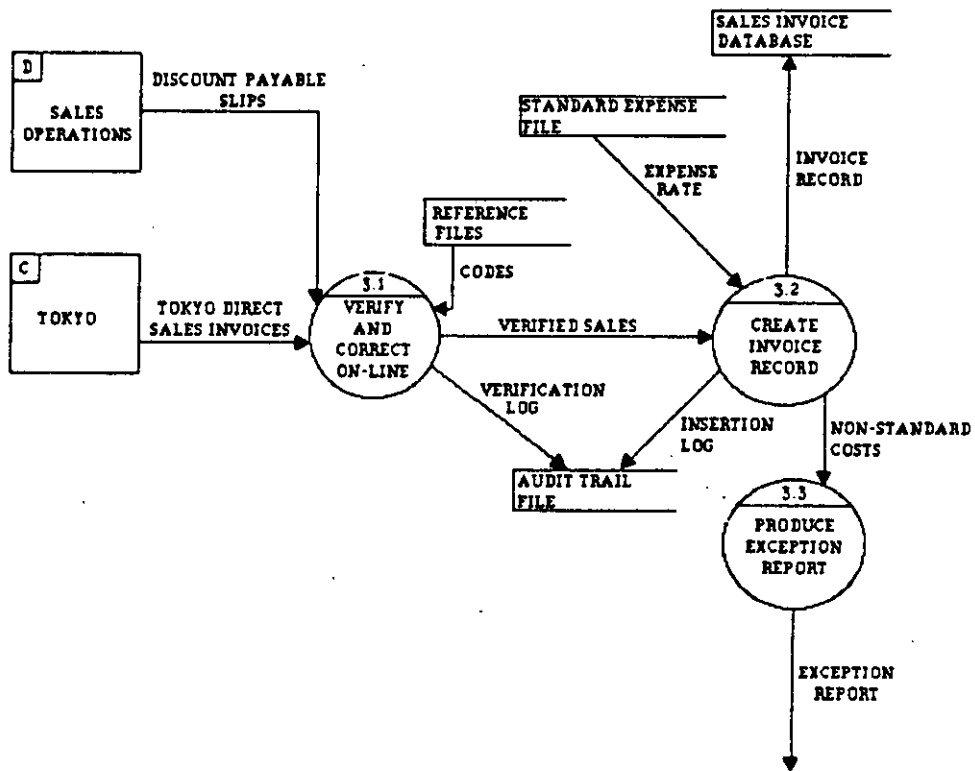Auto Load and Edit

120

Figure 6
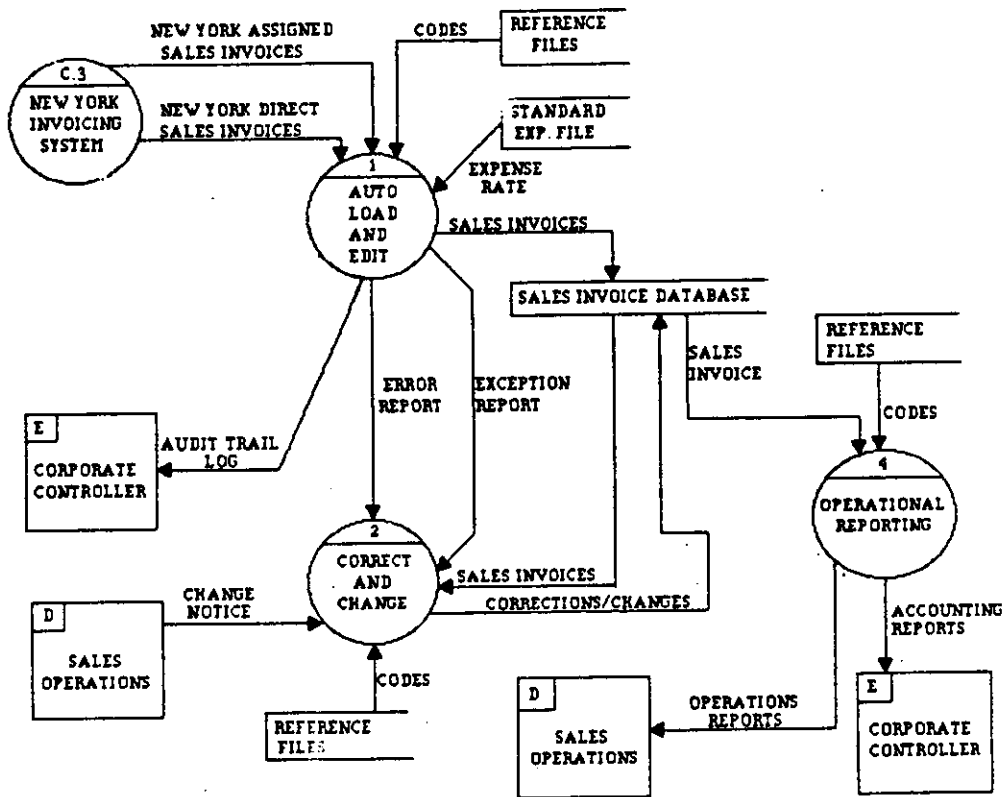
Manual Add and Edit

121

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)
```
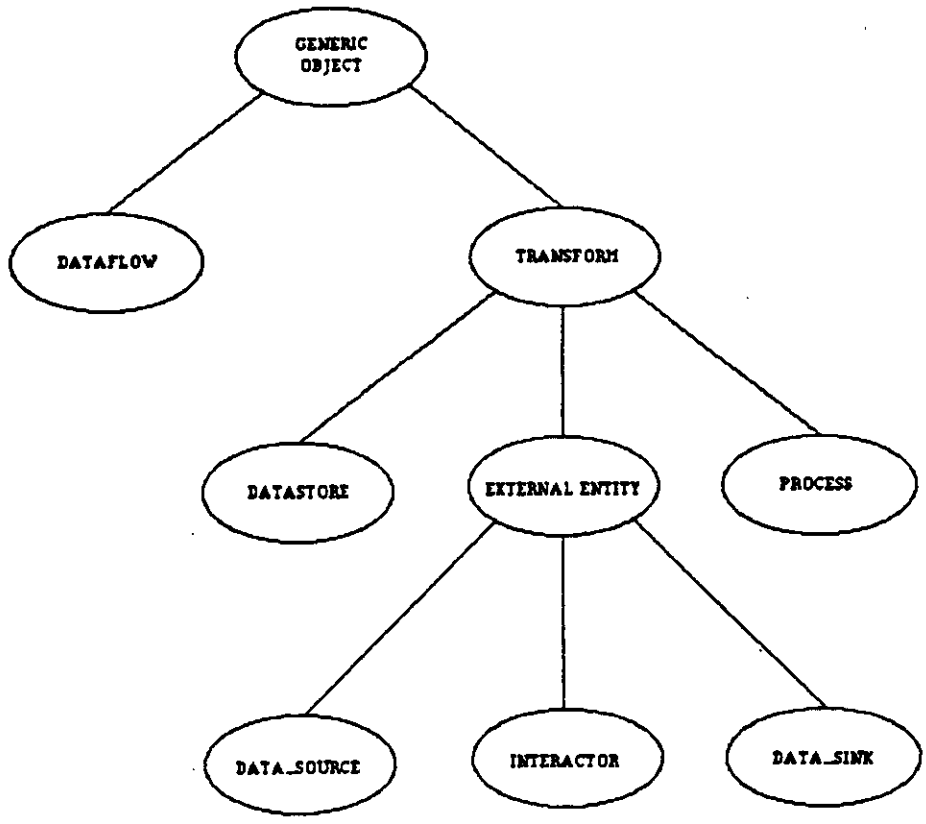
124

Figure 8
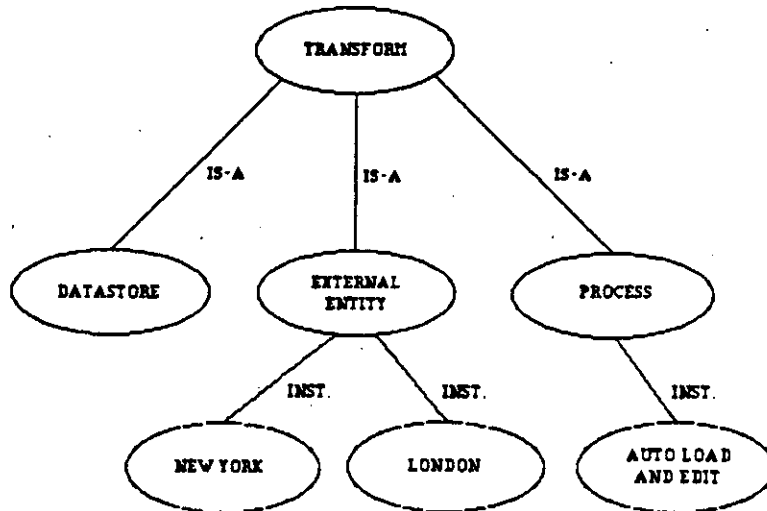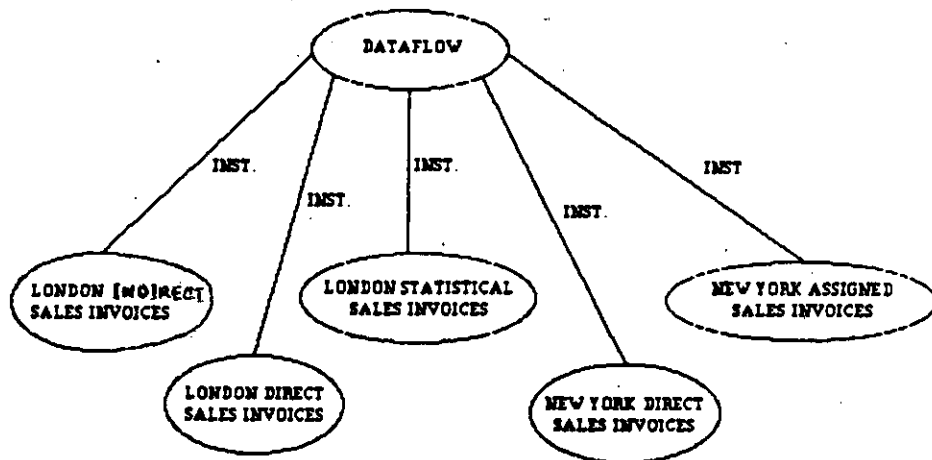
Initial Object Type Hierarchies

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.
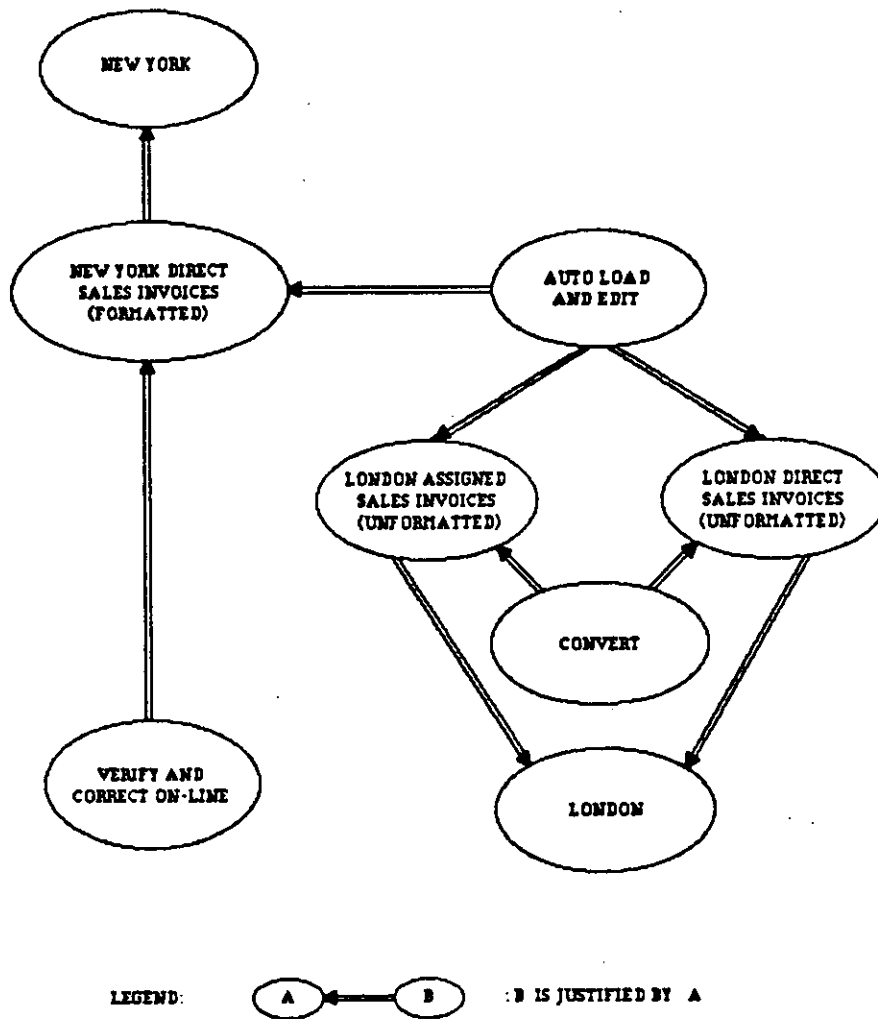
Figure 10

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.

128

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
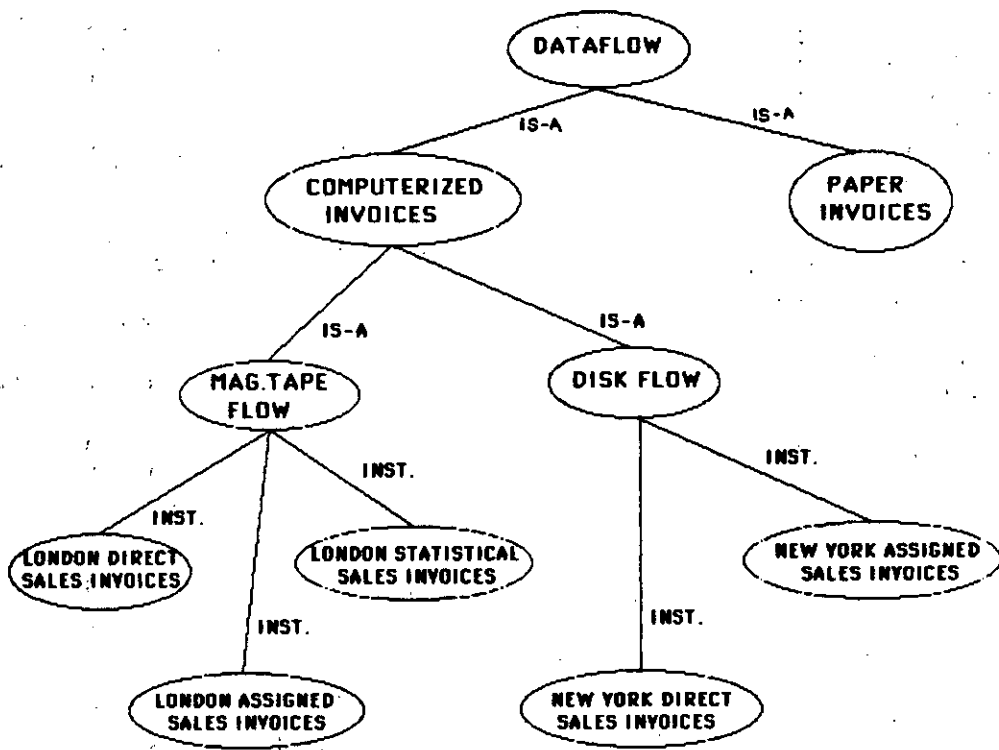
Figure 11

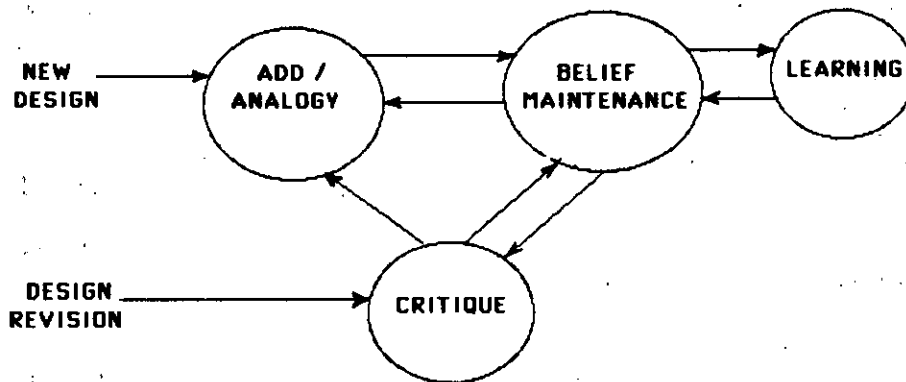Reconfigured Generalization Hierarchy

Figure 12

State Transition Network

representational tools (data flow diagrams, data diction-aries, 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 struc-tured 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. How-ever, subsequent modifications are typically documented only at the program level whereas the design documents remain unchanged. After a few such changes, the pro-gram bears little resemblance to the original design. As a response to this problem, some researchers have pro-posed 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 pro-gramming standards in most large organizations. Such standards include naming conventions, design methodol-ogies, structured programming rules, and documentation guidelines. They could, in principle, serve as a knowl-edge structure for supporting designers and programmers (Jarke and Shalev, 1984) but are currently applied man-ually, 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 fre-quently evolve over time.

None of these improvements adequately address funda-mental 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 be-comes ineffective (McCracken, 1980; Martin, 1982).

Here lies a major advantage of the prototyping approach (Jenkins, 1983). However, without an appropriate envi-ronment, 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 environ-ment* (Reiner et al., 1984) which is equipped with high level knowledge that can be used to reason about the ob-ject domain.

In the general systems arena, Kotteman and Konsynski (1984) have taken the approach of representing applica-tion-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 com-ponents 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 defin-able, may become general to the point of missing the sub-tleties 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 in-stances in a domain in terms of dependencies, and gener-alizes 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 intelli-gent 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. How-ever, 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 diver-sity 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 con-structed 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 feasi-bility. The system contains an implementaion of the ob-ject 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 general-

ization 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.