A STATE OF THE ART REPORT:
SOFTWARE DESIGN METHODS

Contract Number F30602-92-C-0158, CDRL A010
Data & Analysis Center for Software

March 1994

Prepared for:

Rome Laboratory
RL/C3CB
Griffiss AFB, NY 13441-4505

Prepared by:

Kaman Sciences Corporation
258 Genesee Street
Utica, New York 13502-4627

PREFACE

This state-of-the-art review provides an analysis of the status of software design methods. It was researched and published as a service of the Data & Analysis Center for Software (DACS). The DACS is a Department of Defense Information Analysis Center (IAC) whose mission is to support the development, testing, validation, distribution and use of software engineering technologies. The DACS is operated by Kaman Sciences Corporation of Utica, New York under the sponsorship of the Defense Technical Information Center (DTIC), Alexandria, Virginia. It is monitored by the U.S. Air Force's Rome Laboratory in Rome, New York under contract number F30602-92-C-0158.

The topic of software design is an extensive one with a rich history. Views of software design can range from very focused to those which cover the whole spectrum of software development. This report attempts to provide readers with a useful snapshot of software design technology that can be used as a tutorial for the uninitiated, a starting point for detailed research, or a guide for those who will be developing software in the future. The report includes coverage on the nature of design, its evolution, its status, and directions for the future. Section 2.0 covers the nature and history of design. Section 3.1 covers new paradigms for software development and the role design plays in these paradigms. Section 3.2 examines programming paradigms. These paradigms do not derive from development concerns, but do shape the form of program design and software development paradigms. Section 3.3 discusses a selection of specific design technologies that may standalone, or may be definable only in the context of a larger software development methodology. Section 3.4 discusses software design and its place in future integrated CASE environments. Section 3.5 describes the issues related to software design "In the Large" and how these issues are being addressed. Section 3.6 discusses the need to enforce good design within a methodology or software development environment. Finally, Section 4.0 reflects the authors' perceptions of the state of the art of software design as indicated by this research, and some ideas are discussed as to where software design research could lead to from here. Object-Oriented technology and its influences on software design are covered because this technology promises to have a large impact on future software development.

TABLE OF CONTENTS

SECTION PAGE

LIST OF FIGURES

1. INTRODUCTION

1.1 Scope

This state of the art review critiques the technical state of software design methods. Software design is almost universally recognized by software engineering practitioners as a distinct activity required for the achievement of well-engineered software. Well-engineered in this sense means engineered with consideration of quality factors such as reliability, maintainability, and usability, in the same manner that a fine automobile or a computer hardware system would be.

Definitions of design are as diverse as design methods. Coad and Yourdon (1991) define software design as an activity:

> "The practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details."

Webster (1988) defines software design as a representation:

> "In a sense, design is representation -- of an object being created. A design is an information base that describes aspects of this object, and the design process can be viewed as successive elaboration of representations, such as adding more information or even backtracking and exploring alternatives."

Stevens (1991) is more input/output oriented and to him software design is:

> "... a process of inventing and selecting programs that meet the objectives for a software system. Input includes an understanding of the following:
>
> 1. Requirements.
> 2. Environmental constraints.
> 3. Design criteria.
>
> "The output of the design effort is composed of the following:
>
> 1. An architecture design which shows how pieces are interrelated.
> 2. Specifications for any new pieces.
> 3. Definitions for any new data."

As a distinct sequential phase, the Military Standard for Defense System Software Development (DOD-STD-2167A) places design between the *Requirements Analysis*

and *Coding and Unit Testing* phases, and further partitions software design into *Preliminary Design* and *Detailed Design* activities. Preliminary design, which others may classify as system design, architectural design, or top-level design, has to do with the identification of modules that can be logically partitioned into separate functions for purposes of implementation convenience. Detailed design then focuses in on the internal logic of these modules (Jalote, 1991). In Steven's (1991) definition, preliminary design corresponds to output 1 and detailed design corresponds to outputs 2 and 3.

Webster (1988) and Belady (1990) talk about upstream and downstream design. Upstream design is adaptable and abstract and tends to fall within the domains of DOD-STD-2167A's *Requirements Analysis* and *Preliminary Design* phases, while downstream design is concerned with modules, code and documentation and tends to fall within DOD-STD-2167A's *Detailed Design* and *Coding* phases. Upstream activities require a greater component of human expertise and interaction while downstream activities lend themselves more readily to automation. Downstream activities begin when the software designer is confident that his top-level design is correct and complete, and is ready for detailed design. As we shall see later, DOD-STD-2167A's distinct phases are related to the traditional waterfall model of software development that is now being challenged by new evolutionary paradigms and increased automation of the software development process. For example, in software development paradigms like the transformational model, design is always accomplished at the specification level and the paradigm relies on the idea that the specification can be formal enough to be compiled into code.

This review will consider both preliminary and detailed design methods, but they will not generally be distinguished because much of the current thinking on software development does not explicitly make such a distinction. To support this notion, the review will consider methodologies that embed design methods and other activities, within new software development environments. This document will also consider both the conceptual model of the design method and the tools required to realize that method.

1.2 Background

Software design is a problem-solving process whereby the designer applies techniques and principles to produce a conceptualization, or logical model, that describes and defines a solution to a problem perceived in reality. This process is not well structured; the solution set is not limited, the model as built by the designer cannot always accurately represent the problem, and there is no known measure of the effectiveness of the many possible courses of action (Olle, Soland & Tully, 1983).

Since the 1960s, software design methodologies have been evolving; as understanding of the process expands, new and better methods are developed. Computer software design, however, is still evolving and cannot be considered

totally equivalent to other engineering design disciplines. Where early design methods addressed specific aspects of software development, current methods are attempting to fit comfortably within the entire scope of software development. However, many of the current software design methods do not include components that address problem-solving, even fewer consider the human factors aspect, and for some of the newer information systems technologies (robotics, for example) alternative design philosophies are being considered.

Like art objects that can be classified by the age in which they were produced, the many software design methods that have been developed can be classified according to the period in which they were introduced during the evolution of software engineering. Driven by recognized problems, the software community initiated solutions and then developed tools implementing the solutions. Responding to coding and testing problems, early methods dwelt on modularity and top-down development, stepwise refinement, information hiding and levels of abstraction that led to the development of structured languages. Poorly structured designs were dealt with using structured techniques such as structured design, structured analysis and data flow analysis.

Most recently, much of the interest of the software engineering community has been shifted to object oriented (O-O) design. Inefficient designs and development are being addressed with Computer Aided Software Engineering (CASE) tools, fourth generation languages and design languages. Now that the expense involved in automating systems has shifted from hardware to people, there is considerable effort directed toward replacing, or at least augmenting, the familiar waterfall life cycle process model espoused by Boehm (1976), under which most existing software has been developed. It is now thought that this model does not adequately represent the development process of newer classes of software, especially domain-dependent software. If this initiative succeeds, there will be an immense impact on software design methods.

2. OVERVIEW OF SOFTWARE DESIGN METHODS

Each software design method has as its goal to provide the software designer with a blueprint from which a reliable system may be built. This section covers the nature of software design in more detail. It defines the fundamentals which software design should adhere to, design's role as a representational model, and a historic perspective on design.

2.1 Design Fundamentals

Three distinctive aspects of an information system are addressed during its software design. Data design is involved with the organization, access methods, associativity, and processing alternatives of the system's data. Architectural (preliminary) design defines the components, or modules, of the system and the relationships that exist between them. Procedural (detailed) design uses the products of the data and architectural design phases to describe the processing details of the system -- module internals. Software design methods attempt to aid the designer in each of these three aspects; they assist in partitioning the software into smaller components and reducing complexity; they help to identify and isolate data structures and functions; and they attempt to provide some measure of software quality.

Regardless of its specifics, every software design method that has been introduced to the software engineering community is based to some extent on the same proven concepts, and shares common characteristics (Pressman, 1987). They each aid the designer by providing the following:

- A mechanism translating the physical problem to its design representation.
- A notation for representing functional components and their interfaces.
- Heuristics for refinement and partitioning.
- Guidelines for quality assessment.

Fundamental concepts that have remained fairly constant, although the degree to which they are stressed varies considerably, are stepwise refinement, software architecture, program structure, data structure, software procedure, modularity, abstraction, and information hiding.

2.1.1. Stepwise Refinement

Stepwise refinement is a top-down approach where a program is refined as a hierarchy of increasing levels of detail. This process may be begun during requirements analysis and conclude when the detail of the design is sufficient for conversion into code. Processing procedures and data structures are likely to be refined in parallel.

2.1.2. Abstraction

Abstraction is a means of describing a program function, at an appropriate level of detail. At the highest level of abstraction a solution is stated in the language of the problem environment (requirements analysis). At the lowest level of abstraction, implementation-oriented terminology is used (programming). An abstraction can be compared to a model which incorporates detail only to the extent needed to fulfill its purpose.

2.1.3. Software Architecture

While refinement is about the level of detail, architecture is about structure. The architecture of the procedural and data elements of a design represents a software solution for the real-world problem defined by the requirements analysis. It is unlikely that there will be one obvious candidate architecture.

2.1.4. Program Structure

The program structure represents the hierarchy of control. Program structure is usually expressed as a simple hierarchy showing super-ordinate and subordinate relationships of modules.

2.1.5. Data Structure

Data structure represents the organization, access method, associativity, and processing alternatives for problem-related information. Classic data structures include scalar, sequential, linked-list, n-dimensional, and hierarchical. Data structure, along with program structure, makes up the software architecture.

2.1.6. Modularity

Modularity derives from the architecture. Modularity is a logical partitioning of the software design that allows complex software to be manageable for purposes of implementation and maintenance. The logic of partitioning may be based on related functions, implementation considerations, data links, or other criteria. Modularity does imply interface overhead related to information exchange between modules and execution of modules.

2.1.7. Software Procedure

Software procedure provides a precise specification of the software processing, including sequence of events, exact decision points, repetitive operations, and data organization. Processing defined for each module must include references to all subordinate modules identified by the program structure.

2.1.8. Information Hiding

Information hiding is an adjunct of modularity. It permits modules to be designed and coded without concern for the internals of other modules. Only the access protocols of a module need to be shared with the implementers of other modules. Information hiding simplifies testing and modification by localizing these activities to individual modules.

2.2 Design Representation

Webster (1988) defines design as an information base that describes the thing being designed, and design methods as representations of this information. Webster further states that there are two areas which representation must address: one that makes the design method manageable for the human user and one that makes the design method interpretable by the computer. Interpretability has, of course, important ramifications for the ability to automate design and software development in the whole.

Design methods are discussed here in terms of representation because representation is a convenient way to compare the applicability of design methods. In the reference, Webster rated the 44 technologies listed in Figure 1 according to their ability to cover the representational needs of the design process. Of particular interest was Webster's ranking of each technology's scope of application, and its ability to meet both the needs of human processing and machine processing (conceptual complexity versus processable expressiveness). These rankings are best shown in Figures 2 and 3 which have been extracted from his article. In reference to the distribution shown in Figure 3, Webster concludes that, "... complexity seems to go up dramatically with increasing expressiveness, suggesting these methods may be buying expressiveness at an inordinate cost in complexity." He also concludes that object-oriented and frame-based representation show the most promise for overcoming the shortcomings of conventional mechanisms. This conclusion is supported by the placement of these representations in the map in Figure 3.

In considering Webster's results, it is important to remember that he viewed design more as part of a continuous process and the technologies reviewed consist of both design methods and comprehensive methodologies that accomplish design in some way. The concept of upstream and downstream design that is used by Webster (See Section 1.1) to qualify design activities provides a quick and dirty way to place design methods in the general scheme of things. Webster did focus on the upstream leg of design, arguing that the upstream and downstream activities must interface continually and be compatible in representation. The placement of the technologies reviewed in the life cycle can be seen in Figure 2.

Figure 1. Representational Technologies Considered by Webster
(From Webster, D.E. (December, 1988). *Mapping the Design Information Representation Terrain. Computer*, 21(12), 8-23.)

Figure 2. Applications Ranges of Representation Media
(From Webster, D.E. (December, 1988). *Mapping the Design Information Representation Terrain. Computer,* 21(12), 8-23.)

Figure 3. Processable Expressiveness versus Conceptual Complexity.
(From Webster, D.E. (December, 1988). *Mapping the Design Information Representation Terrain*. Computer, 21(12), 8-23.)

2.3 Historical Perspective

During the 1960s the arrival of sufficiently powerful hardware and the wide availability of higher level programming languages resulted in computers being applied to problems of increasing complexity. Reservation systems, personal record systems, and manufacturing systems are a few examples. A significant percentage of these systems was unsuccessful, late, or over budget due to the analysts' inability to sufficiently:

- Translate complex problems to workable software solutions
- Take end-user opinions and practical needs into account
- Take into account the organizational environment
- Accurately estimate the development time and cost, and the operational costs
- Review the project progress with the customers in a regular and consistent manner.
- Anticipate performance/technology tradeoffs (Wasserman, 1979).

This expansion of software development resulted in a widely recognized "software crisis." This crisis made it apparent that a systematic approach was needed and the concept "software engineering" emerged based upon the notion that an engineering-like discipline could be applied to software systems. Meetings

sponsored by NATO in 1968 and 1969 defined the problem more clearly and established some start-up approaches.

During the late 1960s and early 1970s research on software engineering had little direct impact on practical software development, but some of the most important concepts were formed during this period. Among these were top-down design, stepwise refinement, modularity, and structured programming. Structured programming has had a widespread impact and has spawned a family of structured techniques with many variations for different phases of the life-cycle. Structured programming, as represented in its many forms and extensions, is still the most dominant approach to software engineering. Figure 4 illustrates the evolution of structured techniques.

Structured techniques are themselves evolving. Techniques introduced in the 1970s were lacking in many ways, and methodologies were needed that:

- Were more complete.
- Were faster to use.
- Were based on sound data administration.
- Were suitable for fourth-generation languages and application generators.
- Were enhanced for user communication.
- Applied thorough verification techniques
- Solved the severe problems of maintenance
- Were suitable-for computer-aided design with interactive graphics (Martin & McClure, 1985).

2.3.1. Flow Charts

Prior to the structured programming revolution, flowcharts were the predominant method of representing program logic. Flowcharts are limited by a physical view of the system that is improperly applied before overall logical requirements are understood. Flow charts are essentially a detailed design method and have been largely supplanted by such methods as structured English (a PDL as below), Nassi-Sneiderman charts, and action diagrams which can enforce a structured approach (Martin & McClure, 1985).

2.3.2. Program Design Languages

Program design languages (PDLs) are very-high-level programming languages (Caine, 1975). They express the logic of a program in narrative form (Wasserman, 1979). A PDL is principally applied during the detailed design phase and it is best used to describe algorithms and specify interfaces between modules. PDLs impose a language syntax upon their users that makes it possible to use automated tools for validity checking. PDLs were first proposed as a replacement for flowcharts.

Figure 4. Evolution of Structured Techniques. (From Martin, J. & McClure, C. (1985). *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice Hall, Inc.)

2.3.3. Forms

In a forms-driven methodology, standard forms guide the analyst in the collection and analysis of data and in preparing designs. IBM's Study Organization Plan (SOP) method of the 1960s and its 1970s extension called Business Systems Planning (BSP) are examples of early forms-driven software design techniques. While SOP offered the analyst great creative latitude, the BSP version put more emphasis on teamwork, strategic planning and database design. HIPO (Hierarchy Plus Input-Process-Output) was subsequently developed by IBM as a documentation aid and provided a structure for describing and understanding system functions through a visual description of inputs, outputs and processes (Stay, 1976). HIPO's principal advantages are its simplicity and its applicability for both specification and design. It has several severe limitations:

- No representation of control flow information (loop, sequence, etc.).
- No provision for global data structures or databases.
- No differentiation between types of functional modules (common vs. library).
- No provision for user-oriented design (menus).
- Charts were difficult to produce and maintain.

A modernized version of HIPO that attempts to address these limitations is discussed later in this document.

2.3.4. Data Structures

The Data Structure techniques propose a detailed prescription for designing systems based on an analysis of the structure of the input and output data. This is a problem-oriented approach in which the problem is expressed by a mapping of the input data structures to the output data structures. These methods infer that the system is implicit in its data structures. Methods based on this technique are those of Warnier (1974), Jackson (1975), and Orr (1977).

Jackson's Method (JM) is representative. JM was founded on the belief that the structure embodied in systems should correspond to the structure of the data being processed. Jackson espoused four basic constructs to model both programs and data: (1) elementary, (2) sequence, (3) iteration, and (4) selection. The original model was limited to detailed design and did not address the need for a precise definition of the inputs and outputs. The method has since been extended as the Jackson System Design (JSD) into requirements analysis and programming (Cameron, 1986). JSD begins with a model of the real world and the end product is compilable source code. The implementation step is intended to be automatic. The design is transformed from structured text to compilable code, and modification is achieved at the design level. JSD does not provide any guidelines for analyzing the problem.

2.3.5. Data Flow

Data flow techniques are based on analyzing the flow of data through the system in a manner similar to that of data structure techniques. Flow diagrams are used to perform a top-down segmentation of the system into successively simpler levels of modularity. Examples of methods employing data flow techniques are the following:

- Softech's Structured Analysis and Design Technique (SADT) (Ross, 1977).
- DeMarco's Structured Analysis (1978).
- Myers' Composite Design (1978).
- Yourdan and Constantine's Structured Design (1979).

Although based on structured programming, there are significant differences between these popular software design methods. Structured Design (SD) partitions a system into a hierarchical structure of loosely coupled modules within which exists tightly coupled code. Using data flow diagrams (DFDs), a data dictionary, structured English, and decision tables and trees, Structured Analysis (SA) builds a structured system specification (establishing new standards for documenting the analysis phase of system development). The output of a SA phase serves as input to the structured design (SD) phase where a concept known as source/transform/sink (STS) decomposition is applied to the DFDs to identify program modules. A source identifies the bounds of the input data stream, a sink encapsulates the output data stream, and the transform identifies the process of mapping the input to the output.

Structured design is primarily a method for architectural design and its greatest strength lies in the early specification of subsystem interfaces. Its weaknesses lie in its narrow focus. It provides little direct help with specification and detailed design although the development of structured analysis has satisfied the specification side.

Softech's proprietary SADT employs well-trained teams to perform data flow analysis and documentation using the Softech communication tools. SADT is primarily for requirements analysis and definition, and its products do not directly transform into designs or implementation.

2.3.6. Conventional Life-Cycle Model

Of equal importance to the conception and evolution of structured programming was the notion of the software life-cycle. The idea of a software life-cycle was also adopted from the engineering community who recognized as early as the 1930s that all products have a life of finite duration which begins at conception and subsequently passes through phases of specification, design, implementation, maintenance and obsolescence. For software engineering, a life-cycle provides a series of distinct tasks to which engineering methods can be applied and hopefully integrated as a life-cycle methodology. Figure 5 shows the conventional or "waterfall" software development cycle in its best known form. This form was

popularized by Barry Boehm (1976) who gives credit to W.W. Royce (1970) for its original publication.

Agresti (1986a) claims the principle virtue of the waterfall model was its recognition of design as a critical activity. He cites Boehm (1981) as an example of an author who has shown the expensive consequences of premature coding without adequate analysis and design. The waterfall model is being criticized today, if not replaced, because:

- The availability of economic computer power no longer requires rigid planning to accommodate expensive mainframe resources.
- Writing fixed, detailed specifications before coding is not effective and evolutionary paradigms are needed.
- Automation and availability of tools that span several conventional phases have eliminated the need for this strict partitioning.

Figure 5. Conventional "Waterfall" Software Development Life-Cycle Model (From Boehm, B.W. (December 1976). *Software Engineering.*. IEEE Transactions on Computers. c-25(12), 1226-1241.)

On the day an interactive video display first appeared on a user's desk, information systems crossed a threshold. Because users rarely foresee what they either want or can have, the waterfall model seldom delivers the correct solution on time. Because requirements are always changing, it is difficult to execute the stages; as some parts of a system are undergoing testing, new requirements demand changes in the design. Some current thinking declares that there is no best way to develop modern information systems and that the methodology to apply depends on the kind of

system being developed. Systems can be typed as being "database," "decision support," etc., and it begins to appear that quite different methodologies may be appropriate for the design of each (Olive, 1983). There is a large consensus that rather than a design methodology, a holistic development methodology should be employed. Such a methodology would place emphasis in the beginning stages of development, thus preventing and identifying errors at the earliest possible time, and unifying the management of the process. Importance is attached to the need for automated tools and for methods that lend themselves to automation. Quality control measures are becoming increasingly important due to the complexity of today's systems. Most significantly, end-user participation throughout the development process is not only accepted, it is now considered critical to the success of the system.

Section 3.1 of this paper discusses new paradigms for the software development life-cycle. These paradigms attempt to address the above criticisms and take into account opportunities for automation and the recognition that evolutionary development is perhaps the more natural way to build software.

2.3.7. Object-Oriented Technology

Object-Oriented (O-O) software technology is likely the most active area of research today, and this may mean the eventual obsolescence of many of the technologies discussed above. O-O design is fundamentally different from the these traditional functional methods (Booch, 1990). In these methods, decomposition relies on the idea that each recognized module is a major step in the overall process, i.e., these methods are procedural. O-O design, on the other hand, is structured around objects and the functions they perform that exist in a model of reality.

The origins of O-O programming can be traced to the late 1960s and users of the simulation language SIMULA. SIMULA represented a higher order language that introduced class as a means to encapsulate data. During the early 1970s Xerox's Palo Alto Research Center (PARC) developed the programming language Smalltalk from O-O principles. Smalltalk is more than a language. It is a complete graphic user interface (GUI) environment for assembling O-O programs and, as such, demonstrates important concepts for automated O-O software development. As a bonus, rapid prototyping capability is a natural fallout of object-oriented development as implemented by Smalltalk. While Smalltalk plays only a minor role in software development today, it is still the standard that all other O-O languages are measured against, and it is often used as an example to teach O-O principles to beginners.

After all these years, why is O-O software development becoming a force in the software development industry? Coad and Yourdon (1991) suggest four changes that occurred in the last decade:

- "O-O concepts have had two decades to mature and attention has gradually shifted from issues of coding to issues of design and analysis.
- "Underlying technologies for building systems have become much more powerful. It has become easier to think about coding in an O-O fashion with the availability of C^{++} and Smalltalk (and the platforms to support them).
- "The systems built today are larger, more complex, and more volatile. An O-O approach to analysis and design is likely to lead to a more stable system. Today's systems are on-line and interactive. An O-O approach to such systems -- from analysis through design and into coding -- is a more natural way of dealing with such user-oriented systems.
- "The systems built today are more 'domain oriented' than systems built in the 1970s and 1980s. Functional complexity is less of a concern; modeling the data has become a moderate priority; modeling the problem domain understanding and system responsibilities take a higher priority."

3. CONTEMPORARY SOFTWARE DESIGN METHODS

Conventional software development practices are emphasizing the idea that increased effort in the early stages of development will result in a better product. The intent of this up-front effort is the prevention and detection of errors at the earliest possible moment. This coherent methodology concept produces a software engineering environment wherein there exist definitive stages of development characterized by specific products which may be reviewed and measured. The stages and products involved in the process are determined by the specific process model used as the framework for the environment, but they generally will map onto the traditional life-cycle phases of analysis, functional specification, design, implementation, testing, and maintenance. The modern environment additionally includes management and communication modules that unify the process, tying the stages together and generally improving the effectiveness of the process.

Some newly developed process models are not similarly structured. They do not attempt to separate phases of software development, such as specification and implementation, but instead support the concepts of stepwise refinement and program transformation. They still keep unnecessary implementation decisions out of the specification, but distinguish changes made during implementation that arise due to a lack of foresight during initial specification as specification modifications. One upshot of this latest stage in the evolution of software engineering is that software design methods and/or tools can no longer be selected without considering the relationship of design to the other elements in the process. Questions that deal with making the overall development process easier, at less cost, with more quality must be answered.

3.1 New Paradigms

Although software design can be identified and defined as a distinct activity, it must be compatible in both concept and implementation with essential development activities such as analysis and coding that precede or follow it. To achieve this compatibility and to provide a framework for life-cycle automation, a pattern of thought or paradigm must be established. An example of a paradigm is the traditional waterfall model of development, but developers of today's complex information systems find that this paradigm is inadequate in the current environment. In today's circumstances the demand for software is growing at a much faster rate than the pool of adequately trained developers, and strict adherence to the waterfall paradigm requires too much time. At the same time, very large, complex systems are difficult to manage under this paradigm because the relevant information is fragmented, spanning several disciplines, and is often incomplete or inconsistent. The following sections highlight some alternative paradigms that are representative of current thinking and are capable of using new software development technologies to better advantage.

3.1.1. Prototyping

Prototyping has been a hot topic in software engineering for more than a decade. During the early nineteen eighties, when prototyping began receiving serious attention, orthodox life cycle developers, considered it an expensive, time consuming process; computing time was a scarce resource not to be "wasted" in the iterative process of prototyping. Those more open and receptive saw it as an effective way of understanding the users' needs and problems, thereby eliminating costly rework later in the development process. They correctly pointed out that user participants in the development phase found it difficult to understand either textual requirement specifications or the interpretive models produced by structured analysis tools.

Since that time, prototypes have successfully been used as a communications aid between users and developers. After several iterations of the prototype, the developers have a better understanding of the requirements and the users have a better idea about how the system will eventually look and feel. Sometimes the prototype has been used to evaluate performance capabilities or determine the feasibility of a design. Whatever its purpose, within the context of the life-cycle model, the prototype has usually been a "quick and dirty" affair, and, once the sought after information has been obtained, it is discarded and conventional software design ensues. In the prototyping paradigm, the prototype is usually not discarded. Rather than being an adjunct to the life cycle model, the prototype becomes the central focus of the process model. The prototype is scoped, scheduled, allocated resources, and refined as depicted in Figure 6 (Agresti, 1986).

The number of times the prototyping loop is exercised is variable; the developers may hit the mark on the first try, or the users may perceive a need to add-to or change previously stated requirements. Once the prototype is certified by the user as satisfying the perceived requirements, it is transformed into a delivered system. The transformation effort is variable; it may require adding functionality whose need was not previously apparent to the user, or in upgrading or replacing inefficient parts of the prototype to meet required performance criteria. Some critics of this process model suggest that the completed prototype be thought of as a specification written in a *program design language*, to be used as input to the traditional system design phase.

Prototyping usually begins very early in the development process, after some preliminary fact finding establishes the substance, scope, and environment of the proposed automation project. A cost/benefit analysis is also usually performed during this period to obtain required approvals and funding. In the prototyping paradigm, functional specifications are not frozen; on the contrary, users are encouraged to revise and change their requirements. Frequently, users don't really know what they want until they actually see it; they are not even aware of the possibilities until a model of the system is in their hands. The process of

demonstration, review, refinement and expansion of the prototype is one in which the user comes closer to authoring the system.

A major decision to be made when following the prototyping paradigm is whether to build a full sized production prototype or a scaled down model. If the system's proposed functionality is fairly well understood, there are a small number of functions, or, it is a standalone or single user system, a full scale prototype can be built to become a full scale production system. However, if considerable uncertainty about the feasibility of the system remains, or if prototyping tools are not available, a scaled down model is more appropriate. Prototypes can be scaled down by only generating uncertain functions or by reducing database size and/or complexity.

Figure 6. The Prototyping Paradigm and Its Relationship to the Conventional Software Development.
(From Agresti, W.W. (1986b), *What are the New Paradigms?* In Agresti, W.W. (ed.).*New Paradigms for Software Development*, . Washington, DC: IEEE Computer Society.)

Prototyping a software system shares one crucial aspect with prototyping in other engineering disciplines -- when does the iterative modeling process stop? Well-understood guidelines must be established that determine when the difference between iterations is so small as to signal a shift to the next phase of the software life cycle. It is not uncommon for a user to avoid making the decision that the model satisfies the requirements. Strong managers who understand the software

development process, are fluent in a wide area of information systems, and who know when and what to prototype and when to stop are required to successfully apply the prototyping paradigm.

According to Carey & Mason (1983), the benefits of prototyping fall into three categories: improved functional requirements, improved interaction requirements, and easier evolution of requirements. In the first category, the prototype reduces distortions in the developer's interpretation of the user's needs and uncovers errors in functional logic. Improved interaction requirements pertain to the design and use of the user interface; users may have failed to correctly specify how an operation is performed, they may have difficulty understanding terminology used in the interface, or they may need help information. Evolution of requirements is needed where a pattern of use cannot be predicted until a level of experience is achieved.

Design decisions are made in developing the prototype. Detailed, downstream decisions are made in converting a prototype to the final version.

3.1.2. Operational Specification

A traditional requirements specification describes what a system should do; the design process describes how it should be done. The "what" and "how" are separated by the distinct phases of the waterfall model, e.g., requirements analysis and design. This separation has ramifications for both phases; during analysis it is frequently imperative that design and implementation considerations be discussed, and design decisions frequently impact requirements specifications negatively. An operational specification attempts to resolve this problem. It is a model of the system, a refinement of a prototype, that can be executed to evaluate the functional behavior of the system, although not necessarily using the same media used in the delivered system (Figure 7). The emphasis is on resolving problems related to the behavior of the system in terms that are understandable to the user. After the executable specification has been prepared to everyone's satisfaction the implementation-oriented issues can be addressed without the burden of changing requirements.

Benefits of the operational paradigm can be summarized as a clear separation of the development process into problem-oriented and implementation-oriented phases, and the provision of an executable model that can be used to clarify and validate user requirements early in the life-cycle.

3.1.3. Transformational Implementation

Much research attention is being given to the transformational implementation paradigm illustrated in Figure 8. The transformational implementation paradigm adds executable formality to the operational specification. Automation is employed to apply a series of transformations to a specification to produce a deliverable software system. Modifications and enhancements to the system are accomplished

by changing the specification and reapplying the transformations; code modification is eliminated (Agresti, 1986).

Transformational implementation requires both formality, to express the specification in a way that transformation rules may be applied, and understandability, so that users can validate that it meets their requirements. This is a significant challenge. The transformational phase shown in Figure 8 must be level-reducing so that detailed design issues such as data structures and algorithms may be introduced. In implementing a transformational environment there is a tradeoff between the work performed during development of the formal specification and the effort entailed to transform the specification into a real system.

Figure 7. The Operational Paradigm.
(From Agresti, W.W. (1986b), *What are the New Paradigms?* In Agresti, W.W. (ed.).*New Paradigms for Software Development*, . Washington, DC: IEEE Computer Society.)

The formal specification is the system baseline. It is validated as to its representation of user requirements, and maintenance is performed at its level of abstraction. System code is modified as a two-step process. First changes are made to a "formal development" that is a record of the sequence of transformations and decisions that were made to transform the specification into the original code. Second the modified formal development is re-transformed to produce the next generation of the system.

The transformational paradigm produces three products:

- A formal specification.
- A delivered system.
- A formal development record.

Figure 8. The Transformational Paradigm.
(From Agresti, W.W. (1986b), *What are the New Paradigms?* In Agresti, W.W. (ed.).*New Paradigms for Software Development*, . Washington, DC: IEEE Computer Society.)

The three paradigms discussed above are related. The formal specification is essentially equivalent to the operational specification. The transformational process is an automated view of the refinement of the operational specification. The operational specification is in essence a prototype. These paradigms are obviously built around the common themes of delivering an executable model for the users to examine early in the development process, and providing a framework onto which automation can be hung.

3.1.4. Evolutionary Software Development

Yeh (1990) proposes a comprehensive Evolutionary Software Development (ESD) paradigm in which a number of advanced concepts work together to achieve effective software evolution. Yeh's proposal is an elaboration of the above transformational paradigm. He expands on the transformational aspect by distinguishing functional prototyping and performance modeling.

Yeh establishes his paradigm by defining a process model and outlining the methodologies, tools and languages that are needed to support it. The ESD process model is shown in Figure 9. An objective is to support both incremental delivery and maintenance within the traditional phased approach. Fundamental to this model is a "precisely interpretable" functional specification that depends on abstraction to hide details of database structures, the orders of executions, and

algorithms, i.e., to incorporate data, control and functional abstractions. A precise functional specification makes it possible to incorporate rapid functional prototyping, performance modeling, and design testing into the design and development process rather than as attachments to it. Code is synthesized at the highest level of specification by the way of direct transformation, or through the use of previously developed components and subsystem designs. System design occurs as a series of functionality-preserving transformations applied to the original specification with the choice of transform determined by performance, error handling and other objectives. Maintenance is done by modification and then transformation of the original specification.

The ESD methodological framework is distinguished from the traditional waterfall model by its emphases on:

- Risk management rather than document management.
- Mixed-level representation permitting simultaneous development.
- The separation of evolutionary enhancement from maintenance.
- Automatic code generation from the functional specification.

Of particular note is the emphases on evaluation and validation -- risk management. Rapid functional prototyping is incorporated by direct generation of executable code from the functional specification and is used to model and exercise the logical capabilities of the system. Performance modeling is incorporated to provide accurate performance analysis and evaluate different design options.

Figure 9. New Paradigm for Software Evolution.
(From Yeh, R.T. (1990). An Alternate Paradigm for *Software Evolution*. In Ng, P.A. & Yeh, R.T. (eds.). *Modern Software Engineering: Foundations and Perspectives.* New York, NY: Van Nostrand Reinhold.)

The ESD paradigm must be supported by two languages: a specification language and a design documentation language. The specification language must be executable, and must be unambiguous, simple, capable of describing the logical system behavior, and able to accept incomplete specifications. To address simplicity, a graphical syntax is recommended. This would be a natural extension of people's tendency to describe system structures graphically.

The design documentation language is needed to capture all the design decisions made during the design process. It must have the ability to specify alternate designs, module interconnection and hierarchical structure.

Yeh requires that the environment needed to support the ESD methodology must include:

- "A collection of tools for building, modifying, testing and documenting the components of the target system."
- "A user interface through which the designer can create, modify, and view components of the target system."
- "A database system that manages all components that make up the target system."

Yeh talks about the design process but, in effect, the activity of design is done first at the specification level and then at the implementation level. The design is dynamic and is accomplished by successive specification revisions or purposes of logical improvements, performance improvements, extensions and maintenance. There is a specification language for prototyping functions and a design documentation language for performance modeling. The design documentation will be represented in the object oriented database.

It is significant that an object-oriented DBMS is recommended by Yeh as the most suitable choice for the ESD. Yeh's recommendation is based on the fact that an object-oriented database is naturally hierarchical (top down) whereas a relational database can achieve this only through elaborate constructions using joins.

The advantages of the ESD paradigm are:

- Improvements, extensions and repair are handled in the same way. Maintenance does not need to be mixed with evolution.
- Evolutionary development is placed in the context of the traditional phased development.
- Evaluation and validation are also incremental and are incorporated into the design process.
- The design process is automated and code is synthesized from specifications.

3.2 Programming Paradigms

The above paradigms are software development paradigms. That is, they were conceived to address the needs of modern life-cycle software development, and no specific software language dictated their essentials. The following two paradigms are programming paradigms. They are described here because they enforce a model of programming that differs significantly from conventional procedural programming and software development paradigms are needed to accommodate their particular view of design and coding. It is also feasible that a software development environment can be developed around these paradigms that is intended to produce conventional software. This is indeed the case with the knowledge based paradigm that is described below. It is also the case with Yeh's (1990) application of an object-oriented database in his ESD paradigm.

3.2.1. Knowledge-based

Newer software development paradigms tend to rely heavily on tool rich environments in which users and designers can interact with models and prototypes. The Artificial Intelligence (AI) community has long worked in a community with integrated environments supporting the AI programmer. AI languages, such as LISP and Prolog, are often interpretive, and AI programs tend to abolish distinctions between data and programs. These characteristics have led to the blurring of the boundary between the operating system and application programs. The AI style of programming builds tools upon tools. Furthermore, AI applications tend to be processing-intensive; thereby leading to an environment in which each analyst has his own workstation.

These characteristics of AI environments are now prevalent among a wider range of communities, and have much to do with the development of newer software development paradigms. The tools accompanying new software development paradigms can incorporate a varying amount of software engineering knowledge. A particular style of software development using knowledge-based tools constitutes the knowledge-based paradigm. This style can be combined with several software development paradigms, but it fits most comfortably with the transformational paradigm.

In a knowledge-based paradigm, the user's needs are captured in a specification or prototype developed upstream. Knowledge-based tools may assist in developing this specification. These tools would then contain both software engineering knowledge and application domain knowledge. This knowledge can be described as representation knowledge since it is knowledge about the representation of what a system does, but not how it performs these functions. The Knowledge-Based Requirements Assistant (KBRA), a part of Rome Laboratory's Knowledge-Based Software Assistant (KBSA) (Czuchry, 1988) and Refine (Markosian, 1990) are

examples of knowledge-based tools that contain representation knowledge to assist in designing a specification.

In a traditional approach, designers convert specification to source code. Their rationales for the decisions they make are typically not captured in design documentation. The results of design decisions are captured in the source code, but in an often obscure manner. Usually, design considerations cannot be recovered from the source code. Maintenance is performed on the source code. The source, the requirements as embodied in the specifications, and the design quickly diverges, with consequences for cost and quality during both development and maintenance.

In the knowledge-based approach, design decisions and their rationale are captured by the environment. Indeed, the ideal knowledge-based environment would automatically translate the specification to source. Knowledge about how to implement a specification and the design processes are encoded in a knowledge-based environment. The knowledge-based approach is based on the supposition that this transformation must be done by AI tools. Currently, a specification cannot simply be compiled into source in some third generation language such as FORTRAN or C.

This ideal is not yet realizable. Human intervention is needed in the process of converting specifications to source code. Even so, a knowledge-based environment captures these interventions and the design decisions behind them. Human designers and AI programs cooperate in producing the design and source code in a highly complex and interactive manner. Maintenance is done using a knowledge based environment related to the one used during development. Requirements changes are made at the specification level, and the derivation of the source code is replayed.

A knowledge-based approach will have consequences in both development and maintenance. A side-effect of the required formality in translating specifications to source code is a proof that the program meets its specifications. The design will be more "pure." The resulting program will therefore be of better quality.

Since the specifications are changed during maintenance, a single change will be more expensive than under a traditional approach where only source code is maintained. Over a long run of operations and maintenance, the maintenance cost will be lower. The software will retain its design longer, and its entropy will increase at a slower rate as compared to the traditional maintenance process.

The knowledge-based paradigm incurs the majority of development costs up front and presumes a development approach with heavy user involvement in requirements analysis. Requirements are the most troublesome area in modern software developments. The knowledge-based paradigm provides tools to attack this problem area. The designer is provided with substantial support. Many low-level decisions are taken care of automatically, leaving the designer free to concentrate on

the higher-level decisions that have the greatest impact on the end-product. Even here, the environment provides guidance. It also allows the consequences of different design decisions to be more easily examined. Thus, knowledge-based environments, when implemented, will lead to a dramatically different role for design in some software development paradigm quite different from the conventional waterfall one.

3.2.2. Object-Oriented Technology

Object-Oriented (O-O) technology is one of the latest approaches to software development, and it shows much promise in solving the problems associated with building modern information systems. Prototyping is an important aspect of design when systems employing object technology are built but, since it involves a whole new way of thinking about software development, most authorities in the software engineering community consider object technology as a paradigm itself. It is a programming paradigm that can best be applied with development tools that are compatible with it. O-O technology contains these three key aspects:

- **Objects:** Software packages designed and developed to correspond with real-world entities and containing all the data and services required to function as their associated entities.
- **Messages:** Communication mechanisms are established that provide the means by which objects work together.
- **Methods:** Methods are services that objects perform to satisfy the functional requirements of the problem domain. Objects request services of other objects through messages.
- **Classes:** Templates for defining families of objects and all the data and services that are common to them, and providing for the concept of inheritance that makes O-O software easier to modify and maintain than conventional software.

Within the context of the traditional waterfall model, O-O provides a common language throughout the stages, reducing the barriers and enhancing the process (Taylor, 1992). Referring to Figure 10, objects defined during requirements analysis become system objects during design, which are then implemented during programming, and are maintained during evolution.

Object technology, however, is not usually employed following the waterfall model; rather, a prototyping paradigm is more suited. Using layers of "tried and true" modules and very little new construction, new applications are assembled with rapid prototyping. The prototype is the working program, and it is modified, extended and refined until it meets the requirements. The traditional approach with all its beneficial controls is still frequently applied to the generation of each of the classes and modules that make up the libraries. As with moving to any radically different technology, there is a price to be paid. To take advantage of O-O technology new development software must be acquired, development personnel trained,

libraries of components must be assemble (or purchased), and organizations need to be educated about the technology.

There is no real separation between analysis and design when applying object technology. During analysis, potential system objects are identified along with their characteristics and relationships. These objects are maintained through design and coding activities by adding design detail.

Figure 10. A Common Language for All Stages.
(From Taylor, D. (1992). *Object Oriented Information Systems.* New York, NY: John Wiley & Sons, Inc.)

There are few automated tools and no standard notation for accomplishing these steps, although some work is being done (Coad & Yourdon 1991) based on entity-relationship diagramming. Similarly, there is no standard notation for expressing a software design once the requirements are known; however, work is being done in this area and several solutions are being offered that either leverage off entity-relationship modeling, the Ada process model, or are completely new.

The benefits of object-oriented development as claimed by its proponents are many:

- Emphasis is on understanding the problem domain.
- The concept of objects performing services is a more natural way of thinking.
- Internal consistency of systems is improved because attributes and services can be viewed as an intrinsic whole.
- The characteristic of inheritance capitalizes on the commonalty of attributes and services.

- The characteristic of information hiding stabilizes systems by localizing changes to objects.
- Object are inherently reusable.
- The object-oriented development process is consistent from analysis, through design, to coding.

3.3 Examples of Contemporary Design Methods

This section describes a spectrum of recent design tools (or software development methodologies that include design) which have attempted to solve some of the weaknesses of historic software design technologies. Representative of this list is HIPO II which makes no grand claims about new paradigms but is a commendable extension of a 1970's technology, and the Knowledge Based Software Assistant (KBSA) which is an automated, life-cycle development methodology which takes advantage of expert systems technology.

3.3.1. HIPO II

As described by William Roetzheim in his book *Structured Design Using HIPO-II* (1990), Hierarchy Plus Input-Process-Output II (HIPO-II) is a new software design technique that integrates software design products with:

- Project management.
- User requirements for prototypes.
- Programmer requirements for clarity, consistency and convenience.
- Systems analyst requirements for flexibility and power

Roetzheim believes that the advanced CASE tools based on data flow and entity-relationship diagrams are not as easily understood as the HIPO design technique. He concedes that, although IBM's original HIPO had serious flaws that caused it to fall out of favor, HIPO-II competes with the most advanced design methods while maintaining its original simplicity. He asserts that HIPO-II exhibits three essential qualities that must be possessed by any structured approach to software development:

- **Consistent** - yielding predictable results under many circumstances
- **Logical -** based on validated theories, heuristics and algorithms
- **Teachable -** involving step-by-step actions that can be readily understood.

Roetzheim finds fault with software design methods that deal principally with the needs of the systems analyst. He stresses that software design affects four significant groups that must both approve of and benefit from the design method used. These groups are the following:

- **Project Managers -**The method must support the decomposition of the system into work packages from which schedules, work assignments, cost estimates, and risk estimates can be derived.
- **End Users -** The method should produce documentation that is clear and non-intimidating to end users with little or no training.
- **Programmers -** The method must produce clear, unambiguous designs.
- **Systems Analysts** - The method must be powerful, flexible and easy to use; it should support graphical techniques, recognize a wide variety of module types (menu, interrupt, common, etc.), should facilitate rapid creation and modification of designs, and include sufficient level of detail.

HIPO II adds the following enhancements to the original HIPO:

- Different module types are supported. The single class of functional models has been expanded to include menu interaction, interrupt handling, keyboard interaction, common modules, and library modules.
- Built-in Prototyping.
- Control flow graphics including sequence, alteration, iteration, concurrency, and recursion.
- Pseudo-code algorithm descriptions.
- Global database design.
- Input-Process-Output (IPO) charts have been reformatted to match typical module internal headers formats.
- Interface to project management software.
- Low cost CASE support is available.

Vertical hierarchy charts are used in HIPO-II. Figure 11 depicts sample HIPO-II charts for processing modules. These charts are clear, can represent many levels of decomposition, are easy to change, can easily be represented on both computer workstations and printers. The benefits of the hierarchical decomposition used in HIPO-II are these:

- Well suited to stepwise refinement
- Can be presented and reviewed at varying levels of detail
- Well suited to top-down implementation
- Results in programs that are well structured, easy to implement, modify, and test
- Structures are easy to represent on a computer screen

HIPO-II includes built-in support for automatic prototyping. It supports prototyping of user dialogue, data entry screens, and reports and is especially useful when developing systems where the user interface is a significant component. Modeling of accuracy, timing, data flow, etc., is not directly supported by HIPO II. The prototyping capability does not involve any extra work from the designer; after the prototype is approved, it becomes the foundation for all later design work.

Like its predecessor, HIPO-II documents program algorithms in the process portion of input-process-output charts; however, these are now improved with the addition of control flow information in the hierarchy charts. There are five control flow constructs depicting sequence, alteration, iteration, concurrency, and recursion that can be added to the charts.

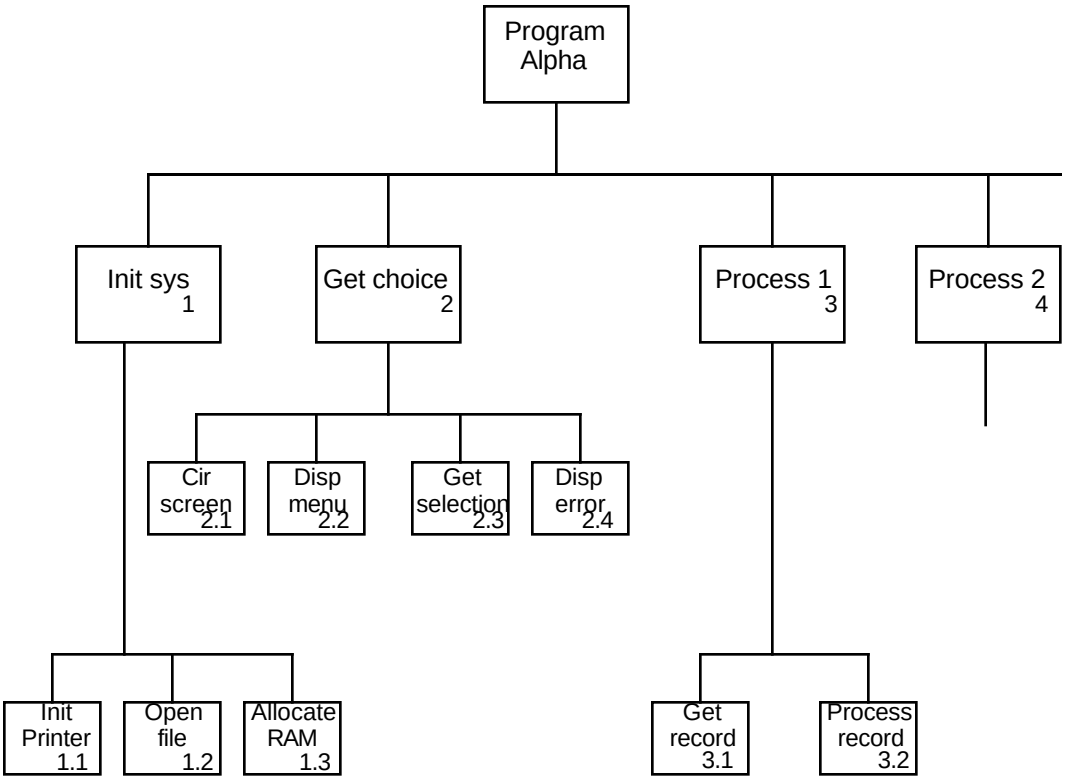

Figure 11. Sample Hierarchy Showing Processing Modules.(From Roetzheim, W.H. (1990). *Structured Design Using HIPO-II.* Englewood Cliffs, NJ: Prentice Hall, Inc.)

Freeform, compact, efficient pseudo-code, or structured English, is used to represent detailed module algorithms. An example, taken from Roetzheim is the following:

> If the track speed is less than 25 Knots, call the routine to update the Kalman filter variable becomes:
>
> If (track.speed < 25) update Kalman filter

For the design of data structures and flow, HIPO-II corrects another deficiency of HIPO by including provisions for designing global data structures. A hierarchical format (similar to those used in functional decomposition) is used to describe variables and data structures accessible from all program elements.

Roetzheim believes that HIPO is the most easily understood design technique ever developed and that the HIPO II enhancements allow it to "compete with even the most extensive 'modern' design techniques while still retaining the charm and simplicity of the original technique." It is a classical structured approach and does not stray far from the conventional life-cycle model, although it does address prototyping at the user interface level. The most important upgrade from HIPO is probably the addition of constructs for human interaction and task management (interrupts). HIPO II can be used for Software *Requirements Analysis* through *Detailed Design* by successive refinements of the design. Webster (1988) would classify it as an upstream design method. The Structured Designer's Tool Box is a low cost CASE tool for IBM compatible PCs that implements the HIPO-II software design methodology.

3.3.2. Object Oriented Design

Coad and Yourdon (1991) introduce a method for Object-Oriented Design (OOD) in their book of the same title. This book is a companion to their previous book "*Object-Oriented Analysis*" (1991). The OOD reference specifically states that the method is for the middle ground between software requirements definition and coding. In the context of the upstream-downstream scale of software design, this OOD method is just downstream.

The OOD method is, of course, an extension of Coad and Yourdon's OOA method. Their OOA method uses a graphic representation with seven constructs as follow:

- **Subject -** A convenient partitioning of a software system into related problem domains. The subject is comparable to the definition of a DoD Computer Software Configuration Item (CSCI).

- **Class -** A description of one or more objects providing the same services (e.g. an automobile moves you from one location to another) and defined by the same attributes (e.g. color of an automobile).

- **Class-&-Object -** A class and all the objects in that class.

- **Gen-Spec Structure** - A Gen-Spec structure is recognized when a Class-&-Object can be specialized by inheriting attributes or services from a more general class, e.g. the class helicopter can inherit properties from the class aircraft.

- **Whole-Part Structure -** A Whole-Part Structure is recognized when both a system and a component of that system are important to the problem domain and can be abstracted as a Class-&-Object. An example of a whole-part structure is an automobile and its engine, i.e., an object in the class automobile may be a brand of automobile and that brand may have several engine options.

- **Instance Connection -** An instance connection is used to represent relationships between objects. For example, a system for tracking automobile registrations must make a connection between a person (the owner) and the owned automobile. In this case the person and automobile Class-&-Objects have their own attributes, but one cannot exist without the other.

- **Message Connection -** A message connection is used when one object may need to request the services of another object.

Coad and Yourdon's graphical representation of these constructs are shown in Figure 12.

To accomplish the act of analysis these constructs are mapped into five vertical layers or activities:

- Identifying Subjects
- Identifying Class-&-Objects
- Identifying Structures
- Defining Services
- Defining Attributes

Coad and Yourdon's methodology uses the same notation for analysis and design, and they point out that this is one of the advantages of the O-O paradigm. For accomplishing the act of design, they identify four horizontal components of software systems:

- The Problem Domain
- Human Interaction
- Task Management
- Data Management

These components should be self explanatory except, perhaps, task management that addresses systems that must deal with multi-users, multi-processors, or external events, including timing signals (e.g., real-time systems).

While the particular design concerns of each of these components are discussed in detail by Coad and Yourdon, the design method is, in essence, a matter of subdividing the constructs identified by the analysis into the four components and adding detail as necessary to begin coding. The result is a matrix filled out with five activities versus four components.

OOA Tool and OOD Tool are drawing and checking tools for this method. These are available from Object International, Inc. of Austin, TX.

Figure 12. OOA Notations.
(From Coad, P. & Yourdon, E. (1991). *Object-Oriented Analysis* Englewood Cliffs, NJ: Yourdon Press.)

3.3.3. Gist

Gist (Balzer, 1982) is an example of a language appropriate for implementing operational prototypes. A specification in Gist, intended to be a "cognitive model" of a system, defines a collection of behaviors of the system. The specification is a closed model, so hardware and human users are described by the specification, as well as the software that will be implemented. Gist provides certain specification freedoms so the description of what a system does is free from considerations related to how it

performs the specified operations. For example, the data model embodied in Gist assumes all needed data is immediately available.

A Gist specification consists of three parts: structural declarations, stimulus-response rules, and constraints. The structural declarations consist of definitions of types, instances of types, and the relationships between objects of each type. These definitions define the state space of a system. They permit the remainder of a Gist specification to use a language appropriate for the application domain. The stimulus-response rules define situations and the range of behaviors resulting from these situations. Gist specifications can be non-deterministic. A specified system is permitted to be in a state in which several stimulus-response rules can be triggered. The constraints prune the state space and the behaviors defined by the other parts of the specification.

A Gist specification is produced upstream in the life-cycle during either system requirements and design, or software requirements. It allows more precise and rigorous requirements analysis. The specification is more likely to be complete and meet the users needs than is the case when non-formal methods are used. Given that the distinction between requirements and high-level design decisions is not precise, the development of a Gist specification can be seen as a part of design. The resulting design product is the Gist specification. Low level design decisions should be easier to make when requirements are more clearly established through an operational specification or prototype.

An interesting consideration is how design decisions abstracted by the specification affect the specification. For example, the Gist specification does not distinguish between those portions of the system that will be implemented in hardware, that will be software, and that will not be implemented at all (for example, user decisions). All of these aspects of the system appear in a closed model of the system. Consequently, the system may be incapable of being implemented if constraints are specified on the user that the implemented portion of the system has no way of enforcing. Similarly, the freedom to ignore details about how data is implemented may lead to an unimplementable system. Requirements and specification decisions are therefore intimately intertwined with tacit design decisions. Since requirements stability is a major problem for both software design and software development as a whole, Gist and the operational specification paradigm address an important software design issue.

3.3.4 Cleanroom Methodology

The Cleanroom methodology (Dyer, 1992) is an iterative, life-cycle approach focused on software quality, especially reliability. Begun by Harlan Mills, it combines formal specifications, structured programming, formal verifications, formal inspections, functional testing based on random selection of test data, software reliability measurement, and Statistical Process Control (SPC) in an integrated whole. SPC, commonly used in manufacturing, involves continuous process monitoring and

improvement to reduce the variance of outputs and to ensure the process remains under control.

The Cleanroom approach fosters attitudes, such as emphasizing defect prevention over defect removal, that are associated with high quality products in fields other than software. Figure 13 shows a possible order for introducing Cleanroom component technologies.

Cleanroom development begins with requirements. Specifications ideally should be developed in a formal language, although the Cleanroom approach allows the level of formality to vary. Cleanroom development is an example of an iterative life-cycle, and incremental releases are used in implementing SPC. The specification is structured to explicitly identify successive releases. Cleanroom-developed specifications also include:

- A target reliability for each release in terms of Mean Time Between Failure (MTBF)
- The operational profile for each increment, that is, the probability distribution of user inputs to the system.

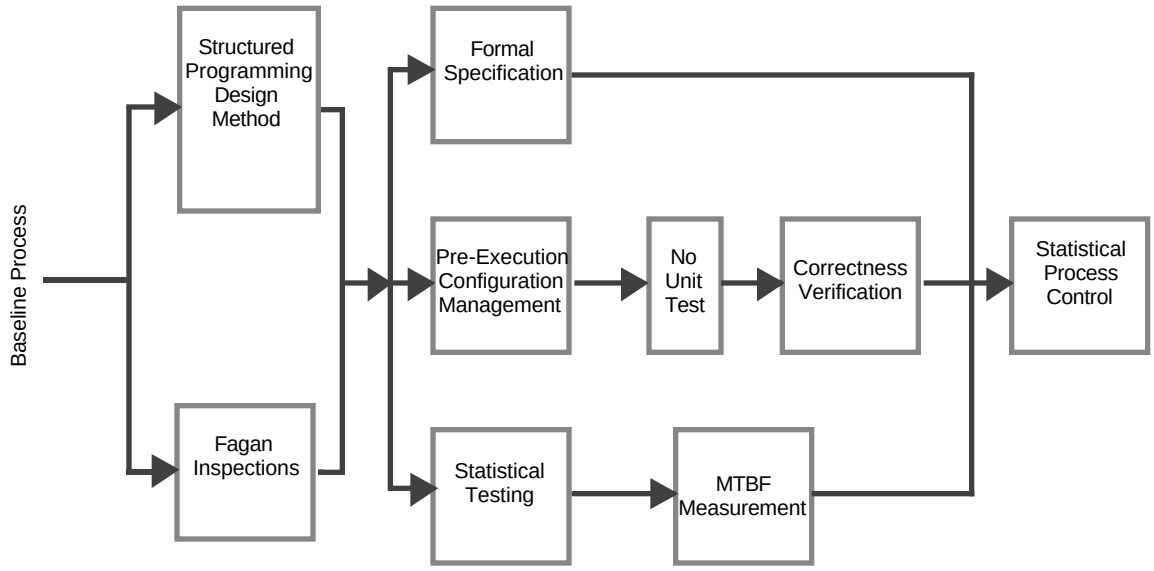

Figure 13. A Road Map for Introducing Cleanroom Technologies

Once a specification is produced, the design, coding, and testing phases are repeated in an iterative style. This process, a combination of the evolutionary prototyping and a type of non-automated transformational paradigm, is illustrated in Figure 14.

The design and coding phases of Cleanroom development are distinctive. Designers develop proofs of correctness, along with their designs and code. These proofs are intended to be human-readable and serve a social role instead of being checked by automated verifiers. Concurrent development of proofs helps guide design and

inspections. Formal inspections (Fagan, 1976) are emphasized, and formal correctness proofs are presented at these inspections. Mills recommends the design follow a formal method based heavily on mathematical functions and sets. Programs are described as a hierarchy of functions, and heuristics are provided to develop designs in a top-down fashion, verify they meet their functional specifications, and recover the function from the design. Mills claims this design method scales up for large systems better than competing formal verification approaches based on assertions, preconditions, and post-conditions (Linger, 1979).

Figure 14. The Cleanroom Life-cycle.
(From Dyer, M. (1992). *The Cleanroom Approach to Quality Software Development*,. John Wiley & Sons, Inc.)

The design process is intended to prevent the introduction of defects. In keeping with this philosophy, the Cleanroom methodology includes no unit or integration testing phases. In fact, coders are forbidden to compile their programs, and programs are placed under formal configuration management before the first compilation. Cleanroom development takes its name from just this aspect of the methodology. Testing is completely separated from the development process, and designers are unable to adopt the attitude that quality can be tested in. Instead, they must produce readable programs that can be convincingly shown correct by proof.

Testing does play a very important role in Cleanroom development. It serves to verify that reliability goals have been met. Given this orientation, testing is

organized differently than in traditional methods. Testing consists entirely of functional testing in which inputs are randomly selected from the specified operational profile. Although bugs are removed when detected, the testing group's responsibility is not to improve the product to meet acceptable failure-rate goals. Rather, testing exists to perform reliability measurement and certification.

When testing fails to demonstrate the desired reliability goal is met, the design process is altered. The level of formality may be increased, more inspections may be planned, or analysts may receive additional training. By combining testing, feedback into the design process, and incremental builds, the Cleanroom methodology tailors SPC to software. As is also evident, the methodology embodies SPC in an institutional structure designed to foster a "right the first time" approach. Methods that have proved themselves in manufacturing quality are thereby adapted to software. The Cleanroom approach has been in existence since 1980 and shown demonstrated effectiveness in trial implementations, but it is not universally accepted among software developers and designers. Inasmuch as Cleanroom draws upon evolving concepts of the best practice in software, it will certainly influence other integrated life-cycle approaches developed in the future. At the very least, future software development methodologies are likely to adapt Cleanroom component technologies, including the design approach of structured programming combined with functional verification and formal inspections.

3.3.5. Leonardo

Leonardo is a comprehensive design support environment being developed under the Microelectronic and Computer Technology Corporation (MCC) Software Technology Program (STP). Leonardo is central to the STP's plan to fulfill their mission of providing an "extraordinary increase in productivity of development and quality of software" for MCC shareholders (Belady, 1990).

MCC performs long term industrial research with an emphasis on continuous technology transfer. Consequently, the STP must account for future trends in software development and not just current events. It is planned that Leonardo will support teams designing large systems and not just individual designers. It is also recognized that networked computers, especially workstations, are becoming ever more pervasive and Leonardo will be oriented towards the design of distributed systems with both hardware and software components.

MCC empirical studies have been used to identify problem areas in design (Curtis, Krasner and Iscoe, 1988). Findings conclude that upstream requirements and design decisions have a significant influence on software productivity, quality, and cost throughout the life-cycle and that successful projects are frequently associated with exceptional designers. These designers are individuals who are able to combine broad application knowledge with an ability to identify unstated requirements, model the interaction of a system's components, and communicate their technical insights in both application and computer science terminology to both the customer

and other analysts. The MCC empirical studies established that productivity and quality are more related to interpersonal communication, team coordination, and design ability rather than to the programming performance of individuals.

Leonardo is being designed to account for the role of upstream activities and their predominant impact on total system cost. It is assumed by Leonardo developers that good design is dependent on human creativity which can be made more efficient with automated support. Leonardo is largely paradigm-independent. It is being built following a bottom-up research strategy that will produce separate components which can be easily transferred to MCC shareholders after completion. The eventual target is a "coherent environment," not a monolithic package that needs to be adopted in one fell swoop. Currently, Leonardo is a research project, not an existing environment.

3.3.6. KBSA

In 1983, it was proposed to the U. S. Air Force's Rome Laboratory (RL, formerly Rome Air Development Center) that a Knowledge-Based Software Assistant (KBSA) be created (Green, 1983). A RL research program was begun, and RL hosted annual conferences on Knowledge-Based Software Engineering to encourage technical interchange among KBSA researchers, the latest being in 1992 (KBSE, 1992). This research program provides an early exemplar of a knowledge-based, transformational approach to software development with an interesting role for design. The KBSA now exists in the form of a demonstration prototype suitable for exhibiting important concepts in the knowledge-based approach to software development.

The knowledge-based software engineering model is illustrated in Figure 15. The KBSA supports an analyst in developing a formal specification. Rules are encoded in the KBSA as is typical in an expert system. These rules record expert knowledge about the domain for which an application is being developed and about software engineering practices. The formal specification is executable, thereby serving as a prototype that can be validated by the user. Once the specification is complete, it is automatically transformed to source code. Any design decisions made by the KBSA in this step are recorded. The designer may have to provide some inputs to tune the final source program. These inputs are also recorded. Any requirement changes to the software during operations and maintenance are made by first modifying the specification and then automatically re-deriving the source code using the decisions recorded in previous transformations.

Design decisions are made at two stages in this process, which is quite different from the traditional waterfall model. Downstream decisions are made in transforming the specification to source code and in tuning the result. The KBSA attempts to automate these decisions as much as possible, thereby minimizing human involvement. The goals of the optimization process (e.g., minimize space or time) must ultimately come from the designer or domain knowledge, but the detailed

implementation of a design satisfying these goals will become increasingly automated.

Less automation is possible for upstream decisions during the requirements analysis phase, but the KBSA gives the expert designer powerful support. The KBSA can automatically deduce the consequences of the interactions of many design decisions. Furthermore, some more obvious decisions will be automated in the KBSA. The KBSA automates lower level details, freeing the designer to concentrate on analyzing the many tradeoffs possible and the implications of alternate designs for a system.

Figure 15. The Knowledge-Based Software Engineering Process.
(From Bailor, P.D. (1992). *Educating Knowledge-Based Software Engineers.* Proceedings of the Seventh Knowledge-Based Software Engineering Conference, McLean, Virginia, September. 20-23.)

3.4 Software Design and CASE Technology

Integrated - Computer Aided Software Engineering (I-CASE) environments provide a coherent set of automated tools to assist software developers in coping with large complex systems and enforcing a disciplined life-cycle engineering approach. I-CASE can be considered an implementation of a software development paradigm.

Since design methods are necessarily a part of I-CASE, it is of value to discuss current trends in these technologies. Mimno (1990) identifies four advances in technology that are converging to produce a whole new generation of CASE tools that are fundamentally changing the life-cycle process. These are:

- Major improvements in human interface capabilities including the use of intuitive command interfaces, elimination of alien syntax, and extensive use of graphics.

- The availability of CASE tools for networks and intelligent desktop workstations including PCs.

- The utilization of front-end graphical design techniques that are amenable to automatic analysis.

- An increasing use of expert system shells and a knowledge base of inference rules.

Mimno further partitions technically advanced CASE products into the following components:

- **Front-end diagramming tools** for Computer Aided Design (CAD) which allow analysts to create, verify and revise drawings on the screen interactively. These tools can support diagramming techniques compatible with a variety of development models including modern techniques based on formal information models and older, manually oriented structured engineering techniques.
- **Design analyzers** that are used to detect internal inconsistencies, ambiguities, and incompleteness in the design specifications.
- **Code generators** that automatically generate code from consistent design specifications. A very tight coupling between the front-end CASE tool for entering and checking specifications and the back-end code generator is required.
- **A central repository** that contains a library of functions, processes, procedures, etc., and a database of pure specifications that can be viewed in a variety of consistent graphical formats. Sufficient detail is maintained in the specifications so that code can be automatically generated from the specification. The repository is also important as a central database for large development teams.
- **Expert Systems** which apply inference processing to a knowledge base containing data and rules about the organization, development process and the application.
- **Methodologies** which enforce a standardized approach to the design and implementation of systems, and guide the analyst through a disciplined application of the CASE tools
- **Life-cycle support** where manual steps are minimized and code generation is tightly coupled with design automation.

No current CASE tools can strongly claim to have all these pieces in place. Most products support only low level code generation facilities like screen maps and data layouts for external languages such as COBOL, C, BASIC and PL/1. While there are code generators that work from user-supplied specifications, they generally do not support the degree of integrated functionality or data base management capability provided by fully integrated CASE tools or 4GL application generators, and they are generally not user friendly. Mimno (1990) discusses a system named CorVision marketed by the Cortex Corporation as an example of a product that supports all phases of the application development life-cycle, but it produces 4GL code rather than an efficient language code.

The direction for advanced I-CASE products appears to be the automatic generation of code from logically consistent and complete graphical specifications. Note that this echoes the idea of the transformational paradigm discussed above. This promises a future where the human component of software design will likely be most prominent at the specification level (upstream), while detailed design activities will be become more automated. There is also an emphasis on visual programming in that a specification is constructed out of graphic symbology that can then be converted directly to code. This uses visual perception, one of humankind's greatest strengths, to greatest advantage.

Advanced I-CASE must also provide prototyping tools for creating models of the system that help define and clarify user requirements. These tools must be integrated with upstream design activities to support direct end-user involvement. The prototype must be at a level of abstraction that the end-user understands, implementing screens, menus, reports, decision trees, procedural logic, and other elements that are recognizable by the user.

3.5 Software Design "In the Large"

When considering software design methods, it is easy to see software development as a personal challenge and to think strictly in terms of a software process model. This is fine for a small system that can be efficiently and correctly developed by a single individual but it is not sufficient for a large system where teams of people must work in a coordinated fashion and eventually integrate their work on different parts of the system. This requires organizational management and its attendant social processes. Software tools designed to aid the individual programmer, even in the context of a large project, do not usually provide the functionality needed to deal with this larger social context. This section focuses on large systems and their influence on software design methods.

In a study conducted for the MCC Software Technology Program, Curtis, Krasner and Iscoe (1988) identify the three most influential problems of software design "in the large" as:

- "The thin spread of application domain knowledge
- "Fluctuating and conflicting requirements
- "Communication and coordination breakdowns"

The study found that a deep understanding of the problem domain was necessary to the success of the project but that this was typically a scarce commodity in the development organization. It was evident that a systems view which sees the computer as an element of a system and not the end product must be accessible to the development team(s). It was proposed by Curtis et al. that access to good domain models is a serious obstacle to the success of automatic programming systems.

The reasons cited for fluctuating requirements were numerous but the concern of system engineers was unresolved design issues and the tools needed for tracking their status. The ratio between issues resolved and issues recorded was thought to be an important measure of design stability and design progress.

The study discovered that many communication activities that appeared to be good software engineering practices were unworkable when scaled up for large projects, and that written documentation poorly served communication needs. In the end, verbal communications served the project best if the development organization was able to develop a common representational convention. It was found that, to foster communications at the project level, the social structure of the project was sometimes factored into system architectural decisions. That is, the system was partitioned in a way to reflect the optimal working arrangements rather than the optimal functional design. A final, important communications issue revolved around artificial organizational or political barriers that hindered the project members' access to customers. This contributed to forgotten information, a shallow understanding of the problem domain, and unstable requirements.

The Curtis et al. study concludes that software development environments must somehow incorporate behavioral processes along with software development processes. These behavioral processes can be encapsulated as three required capabilities: knowledge sharing and integration, change facilitation, and broad communication and coordination. The availability of application domain knowledge must be increased across the entire software development staff. Software development tools and methods must accommodate change as the natural order of things and provide powerful change management features. Software development environments must be designed to be a communications media and integrate information with the tools.

The paradigms and methods described in the sections above support these premises. This is certainly the case with the Leonardo project since the Curtis et al. study served as input to this project. In the section on I-CASE, the requirement for a repository supports both the need for distribution of knowledge about the problem domain, and the idea that the software development environment should be a media for communication. The incorporation of expert systems would also assist

developers in access to information about both the problem domain and the software development process. A project knowledge base could be populated with expert knowledge captured from those rare individuals who understand the project's big picture.

The transformational paradigm manages change at the upstream specification phase rather than the downstream coding phase thus avoiding the situation where changes are needed after the specification is formalized. A history of specification changes is preserved in a formal development record. The emphasis in modern design methodologies on prototyping and iterative development is an acknowledgment that change is inevitable.

3.6 Design Enforcement

Design methods may promote good design but they do not necessarily enforce it. For example, Coad and Yourdon's (1991) method for object-oriented design provides a design framework and guidance for working within this framework, but it can not assure that a good design will be produced by an individual who is uninspired or under pressure. To be sure that a design properly follows the tenets of the method and will have those attributes that predict quality software development, enforcement must be a necessary element of design methods. For development of software systems "In the Large," enforcement helps to ensure that separately designed modules will be uniform in technique and compatible.

Good design is distinguished from correctness. A design that is hard to maintain and difficult to integrate may be capable of producing correct results but its design flaws may eventually haunt the development project in unpredictable ways. Good design cannot be guaranteed because there is no all encompassing definition of a good design, but there are incentives and road markers that can be put in place to assist a designer in keeping on track. Enforcement is especially important if the industry continues toward the transformational paradigm where changes are made at the specification level.

While a level of enforcement could be imposed manually, this would be much easier to subvert for the sake of convenience, and automation is the goal. Automation can inject a level of control that could not be casually bypassed by the software engineer. At the highest level, an automation of the method can enforce the design paradigm. For example, an object-oriented design tool should force the user to define an object before the service provided by that object can be defined. At the next level, design methods should include model checks. This is similar to the design analyzer mentioned in the section on CASE technology.

Coad and Yourdon (1991) distinguish between errors (a rule that cannot be broken) and warnings (a rule that can be broken). Error checking appears to overlap the highest level of enforcement as mentioned above and this kind checking should be

done dynamically as the design is constructed so that errors never enter the design in the first place. Warnings, on the other hand, may be issued dynamically or during a separate process step after the design is completed. Warnings are especially interesting because they imply the use of measures or heuristics. Measures that estimate such characteristics as complexity or size may be embedded in the method to detect possible weaknesses in the design and issue warnings. Likewise rules of thumb that can be used to advise the designer on good practices can be incorporated into a model checker. A more powerful model checker that incorporates knowledge about the application is possible. This idea was implied in the section on software design "In the Large."

Skill and motivation also enforce good design. Those responsible for design must be thoroughly trained in the design method and its principles. They must be convinced of the advantages of the method and that adherence to its tenets will produce a quality product, make for a successful project, and make life easier for them in the long run. Standards such as DOD-STD-2167A, or even stricter in-house standards, can also serve to enforce good design. If an update to a design document is required as part of the design process, the design change becomes public and the designer is forced to review his or her work and make sure that certain questions about the quality of the design have been properly addressed.

In a transformational paradigm where changes are made at the specification level there should be an elaborate check-in procedure for design additions and changes. This is similar to configuration management but it goes further because it makes the designers go through a series of checks before their updates are accepted into the baseline system. Examples of such checks are verification that the model checker has been run and that the project leader has reviewed the design update. Such checks can be supported by automation. A successful run of the model checker may set a flag that clears the design element for entry into the baseline system. Project leaders may have special database write privileges that allow them to update the baseline system with the new design element when they are convinced that the design meets all the requirements of the system and the method.

In an I-CASE environment, enforcement methods as discussed above must be coordinated in a way that makes it very difficult, if not impossible, to execute a poor design. They must be implemented in a way that would force a software designer to have to make a special effort to circumvent these structures.

4. CONCLUSIONS

At this time, software development appears to be at an important juncture. While structured design and its related technologies have dominated in the 1980s; research is now being diverted toward object-oriented technologies. This new direction, whether justified or not, is likely to pull resources away from future advances in structured design and other traditional technologies. The developments of the waterfall model and structured technologies were important breakthroughs that have made possible vast improvements in the way we develop and generally think about software but it is possible that their zenith has passed.

Whether object-oriented technology comes to dominate or not, it appears to be universally recognized that the most difficult phase of software development is in the upstream where the user's requirements need to be pinned down and somehow encoded for implementation. This is apparent in paradigms like the transformational model that hopes to prototype or iterate at the specification level with a very high level language. This is in truth a recognition that this front end of design requires craftsmanship and the user must be kept in the loop, at least at the specification stage. The high-level language requirement comes into the equation as a means to at least partly automate the process and to provide a prototype that is readily interpretable by the client. Interpretability is needed at two levels. At one level is the high level design that is needed by people who will perform verification and validation, or maintenance activities. These people may be part of the client's organization or a third party organization. At an even higher level of abstraction is the end-user's viewpoint. Normally the client's end-user will want to see a realistic representation of a working system and would rarely be interested in interpreting a diagramming language even if it is at a relatively high level of abstraction.

In Yeh's (1990) Evolutionary Software Development paradigm, system design occurs as a series of functionality preserving transformations applied to the original specification with the choice of transform determined by speed, error handling and other performance related objectives. Maintenance is done by modification and then transformation of the original specification. Functional prototyping is incorporated for direct generation of executable code from the functional specification and is used to model and exercise the logical capabilities of the system. Performance modeling is incorporated later to provide accurate performance analysis. In the context of traditional design, functional prototyping represents high level design while performance modeling represents detailed design.

The recognition that software development should be evolutionary is not surprising. All advances are made this way and we would not have expected a modern, aerodynamic automobile to come off Ford's 1927 assembly line. We should also not expect the first version of a software system to come out of the software factory without room for improvement, and with all the features that will ever be needed. That software can be prototyped without a commitment to hardware is its

greatest strength and the prototyping paradigms are simply combining this strength with good engineering practices. It is worth noting that specification prototyping is available today, at least on a small scale. One of the authors has developed applications on a personal computer using the expert system shell LEVEL5 OBJECT (1990). This particular shell provides a multi-paradigm environment in that it supports both rule-based and functional programming in an object-oriented environment. It also provides a set of graphic objects from which the developer can quickly assemble a graphic user interface. Use of this system gives an inkling of the natural way to design software. One starts with the user requirements as they are known at the moment, probably from interviews, site-visits, meetings, etc., and puts together a minimal but working system. It is then shown to the customer and his or her inspired suggestions are incorporated into the next version.

Such cycles of incremental improvements can be repeated as many times as the end-user has time for, or are necessary to capture previously unstated requirements. Software components are abstracted at various levels and the customer is able to see the system at a functional level while the programmer is able to work at the detailed design or coding level. The development environment is self-documenting and the developer can display the architecture of the system or the logic in several different ways. There is no real distinction between designing and coding. Design occurs when the developer creates an object in the context of other objects, and coding occurs when the developer installs the service that the object will perform. The end-user interface is put together with a selection of graphic objects provided with the shell. Changes in the design or code are easily accomplished and minor end-user suggestions could be implemented while he or she observes.

As it is with many expert system shells, this particular system is only practical for developing relatively small, single-user systems. Its interface building tools are limited and it would be of little value for time critical applications because there are no facilities for guaranteeing performance goals. It also has no direct facilities to support projects "In the Large" where many individuals must work in parallel and, at some point, integrate and test their work. Shells such as the one described provide tools to encourage good design but they also have few means to enforce good design. One can easily declare the whole program as an object and fill it with spaghetti code. A trained and inspired programmer with sufficient time is likely to use these tools effectively but a programmer under pressure may not. His or her tendency will be to get the thing working as soon as possible rather than to design first. In summary, existing development shells provide a model for efficient software development, but much needs to be added to be able to cope with development "In the Large."

I-CASE represents an automated implementation of a software development methodology which in itself is a uniform approach for implementing a software development paradigm. An I-CASE implementation is required to enforce the tenets of the methodology and to make that methodology, and consequently the paradigm, realizable for the software development team. Design methods will, of course, be incorporated into I-CASE as an element of the methodology but, as

stressed above, the distinction of design as a stand alone activity may no longer be as valid. In the research above the following themes, which are characteristic of advanced paradigms, were apparent:

- **Front-end diagramming tools:** Upstream design or specification should be given emphasis and be accomplished with diagramming tools and processable graphics. These processable graphics should be automatically checked for errors and inconsistencies, and be transformable into functional and performance prototypes. A variety of diagramming methods that display the specification from different logical perspectives should be available.

- **A customer's vision:** The functional prototype should be at a level of abstraction that allow the system design to be presented to clients as they would visualize it in use. Yeh (1990) separates this aspect of prototyping into functional and behavioral (human interface) components.

- **Performance modeling:** An advanced I-CASE environment should also provide means of transforming the specification into a performance prototype. The performance prototype is a vehicle for verifying that the operational system will meet performance specifications.

- **Consistent internal representation:** The methodology should be founded on an internal representation that is consistent throughout the life-cycle and provides a common basis for storing information about the system under development. The front end diagramming tools should reflect this representation.

- **A repository:** A repository or database is required as part of the implementation to store the design representation, information about the I-CASE methodology, information about the application, and information about the project. The repository should support the multiple graphic representations of the system design or specification.

- **Design "In the Large:"** I-CASE must include distributed tools to support large systems development that requires work to be partitioned among organizations, teams and individuals. The repository should support distributed development by providing a common information archive.

- **Integration:** I-CASE environments must assist in coping with large systems. It must play an active role in coordinating tasks, milestones and change.

- **Features for capturing design issues and tracking status:** The I-CASE environment should have facilities for tracking original implementations and changes. Changes should be tracked from change-order to integration-and-test. Evolutionary development and maintenance should be distinguished.

- **Self documentation:** System documentation should not be developed as a separate step but should derive automatically from actions taken to create or change the software. For example, the graphical specification represents both the specification and the documentation of that specification.

- **Design enforcement:** The methodology should have a infrastructure in place to assist the developer in applying the methodology and to check that the methodology is indeed being applied correctly and uniformly.

These final points can be made about the state of the art and future directions of software design methods:

- Because they have a track record and CASE tools are available, standalone design methods such Roetzheim's (1990) HIPO-II, Yourdon and Constantine's (1979) Structured Design, and Coad and Yourdon's (1991) Object-Oriented Design are still of value and will continue to be used for some time to come.

- Future design methods will be incorporated into I-CASE environments.

- Recent advances in I-CASE environments have been made along the lines of traditional software development methods but there is a wide mix of capability offered and few, if any, uniformly address the entire software life-cycle.

- Advances in user interface and personal workstation technology will drive both the form of future applications software and the form of future design methods.

- Object-orientation will influence software design methods in the future.

- Automation combined with new paradigms like the transformational model will erode the distinctions between requirements definition, design and coding.

- The future emphases on design methods will be at the prototyping and specification level.

- The expert system development shell paradigm with its strong element of prototyping provides a valuable model for future I-CASE environments, but tools for software development "In the large" must be added.

5. REFERENCES

Agresti, W.W. (1986a), The Conventional Software Life-cycle Model: Its Evolution and Assumptions. In Agresti, W.W. (ed.).*New Paradigms for Software Development*, . Washington, D.C: IEEE Computer Society.

Agresti, W.W. (1986b), What are the New Paradigms? In Agresti, W.W. (ed.).*New Paradigms for Software Development*, . Washington, DC: IEEE Computer Society.

Bailor, P.D. (1992). *Educating Knowledge-Based Software Engineers*. Proceedings of the Seventh Knowledge-Based Software Engineering Conference, McLean, Virginia, September. 20-23.

Balzer, R.M., Goldman, N.M. & Wile, D.S. (December 1982). Operational Specification as the Basis for Rapid Prototyping. *ACM SIGSOFT Software Engineering Notes*, 7(5). 3-16.

Belady, L.A. (1990). Leonardo: The MCC Software Research Project,. In Ng, P.A. & Yeh, R.T. (eds,). *Modern Software Engineering: Foundations and Perspectives*. New York, NY: Van Nostrand Reinhold.

Boehm, B.W. (December 1976). Software Engineering.. *IEEE Transactions on Computers*., C-25(12), 1226-1241.

Boehm, B.W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, Inc.

Booch, G. (1990). On the Concept of Object-Oriented Design.. In Ng, P.A. & Yeh, R.T. (eds,). *Modern Software Engineering: Foundations and Perspectives*. New York, NY: Van Nostrand Reinhold.

Caine, S.H. & Gordon, E.K. (1975). PDL - a Tool for Software Design. In Freeman, P. & Wasserman, A.I. (Eds.) (1980). *Tutorial on Software Design Techniques*. Long Beach. CA: IEEE Computer Society.

Cameron,J.R. (1986). An Overview of JSD. *IEEE Transactions of Software Engineering*. SE-12. 220-240.

Carey, T.T. & Mason, R.E.A. (1983), Information System Prototyping: Techniques, Tools, and Methodologies. In Agresti, W.W. (ed.).*New Paradigms for Software Development*, . Washington, DC: IEEE Computer Society.

Cheng, F. & Ng, P.A. (1990). Diagramming Techniques in CASE.. In Ng, P.A. & Yeh, R.T. (eds,). *Modern Software Engineering: Foundations and Perspectives*. New York, NY: Van Nostrand Reinhold.

Coad, P. & Yourdon, E. (1991a). *Object-Oriented Analysis* Englewood Cliffs, NJ: Yourdon Press.

Coad, P. & Yourdon, E. (1991b). *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press.

Curtis B., Krasner, H. & Iscoe, N. (November 1988).A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, 31(11) 1268-1287.

Czuchry, Jr., A. & Harris, D.R. (1988). KBSA: A New Paradigm for Requirements Engineering. *IEEE Expert*. 3(4). 21-35.

DeMarco, T. (1978). *Structured Analysis and System Specification*. New York, NY: Yourdon Press.

Department of Defense (February 1988). *Military Standard Defense System Software Development DOD-STD-2167A*. Washington DC.

Dyer, M. (1992). *The Cleanroom Approach to Quality Software Development*,. John Wiley & Sons, Inc.

Fagan, M. E. (1976). Design and Code Inspections to Reduce Errors in Program Development,. *IBM Systems Journal* 15(3). 182-211.

Green, C. et al. (1983). *Report on a Knowledge-Based Software Assistant*,. Rome Air Development Center. RADC-TR-83-195.

Jackson, M.A. (1975). *Principles of Program Design*. London: Academic.

Jalote, P. (1991). *An Integrated Approach to Software Engineering*. New York, NY: Springer-Verlag.

KBSA (September 1986). *Knowledge-Based Software Assistant Technical Exchange Meeting* . Rome, NY.

KBSA (January 1988). *Second Annual Knowledge-Based Software Assistant Conference*. Rome Air Development Center. RADC-TR-87-243.

KBSA (August 1988). *Third Annual Rome Air Development Center Knowledge-Based Software Assistant*.

KBSA (September 1989). *Proceedings of the 4th Annual Rome Air Development Center Knowledge Based Software Assistant Conference*.

KBSA (September 1990). *Proceedings of the Fifth Conference on Knowledge-Based Software Assistant Conference*. Liverpool, NY.

KBSE (September 1991). *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference*, Syracuse, NY.

KBSE (September 1992). *Proceedings of the Seventh Knowledge-Based Software Engineering Conference*. McLean, Virginia.

LEVEL5 OBJECT (1990). *Object-Oriented Expert System for Microsoft Windows User's Guide*. Information Builders, Inc.

Linger, R. C., Mills, H. D. & Witt, B. I. (1979). *Structured Programming: Theory and Practice,*. Addison-Wesley Publishing Company.

Markosian, L. & Abraido-Fandino, L. & Katzman, S. (1990). Knowledge-Based Software Engineering Using REFINE. In Ng, P.A. & Yeh, R.T. (eds,). *Modern Software Engineering: Foundations and Perspectives*. New York, NY: Van Nostrand Reinhold.

Martin, J. & McClure, C. (1985). *Diagramming Techniques for Analysts and Programmers*. Englewood Cliffs, NJ: Prentice Hall, Inc.

Mimno, P.R. (1990). Survey of CASE Tools. In Ng, P.A. & Yeh, R.T. (eds,). *Modern Software Engineering: Foundations and Perspectives*. New York, NY: Van Nostrand Reinhold.

Myer, G.J. (1978). *Composite/Structured Design*. New York, NY: Nostrand Reinhold.

Olive, A. (1983). Analysis of Conceptual and Logical Models in Information Systems Design Methodology. In Olle, T.W., Sol, H.G. & Tully, C.J. (1983). *Information Systems Design Methodologies*. Amsterdam, The Netherlands: Elsevier Science Publishers B.V.

Olle, T.W., Soland, H.G., & Tully. C.J. (Eds.). (1983). *Information Systems Design Methodologies: A Feature Analysis,*. Amsterdam, The Netherlands: Elsevier Science Publishers B.V.

Orr, K.T. (1977). *Structured Systems Analysis*. Englewood Cliffs, NJ: Yourdon Press.

Payton, D.W. & Bihari, T.E. (August, 1991). *Intelligent Real-time Control of Robotic Vehicles,* Communications of the ACM. 34(8). 48-63.

Roetzheim, W.H. (1990). *Structured Design Using HIPO-II*. Englewood Cliffs, NJ: Prentice Hall, Inc.

Ross, D.T. & Schoman Jr., K.E. (January 1977). Structured Analysis for Requirements Definition. *IEEE Transactions of Software Engineering.* SE-3(1). 69-84.

Royce, W.W. (1970). *Managing the Development of Large Software Systems: Concepts and Techniques.* Proceedings WESCON.

Pressman, R.S. (1987). *Software Engineering: A Practitioner's Approach*,. New York, NY: McGraw-Hill Inc.

Stay, J.F. (1976). HIPO and Integrated Program Design. *IBM Systems Journal.* 15(2). 143-154.

Stevens, W.P. (1991). *Software Design: Concepts and Methods*. Hemel Hempstead, UK: Prentice Hall Intenational Ltd.

Taylor, D. (1992). *Object Oriented Information Systems*. New York, NY: John Wiley & Sons, Inc.

Warnier, J.D. (1974). *Logical Construction of Programs*. New York, NY: Van Nostrand Reinhold.

Wasserman, A.I. (1979). Information System Design Methodology. In Freeman, P. & Wasserman, A.I. (Eds.) (1980). *Tutorial on Software Design Techniques*. Long Beach. CA: IEEE Computer Society.

Webster, D.E. (December, 1988). Mapping the Design Information Representation Terrain. *Computer*, 21(12), 8-23.

Yeh, R.T. (1990). An Alternate Paradigm for Software Evolution. In Ng, P.A. & Yeh, R.T. (eds,). *Modern Software Engineering: Foundations and Perspectives.* New York, NY: Van Nostrand Reinhold.

Yourdon, E. & Constantine, L.L. (1979). *Structured Design*. Englewood Cliffs, NJ: Prentice Hall, Inc.