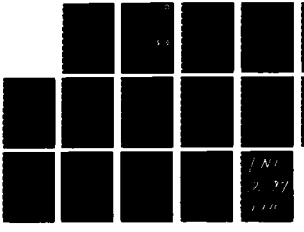
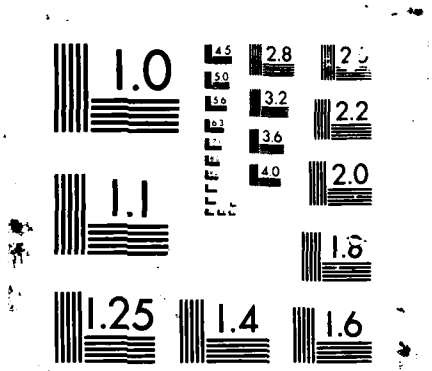
AD-A169 659 AN INVESTIGATION OF EXPERT SYSTEMS USAGE FOR SOFTWARE 1/1
REQUIREMENTS DEVELO (U) NAVAL RESEARCH LAB WASHINGTON
DC V CHIEW ET AL 10 JUN 86 NRL-8968

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

Naval Research Laboratory

Washington, DC 20375-5000 NRL Report 8968 June 10, 1986

An Investigation of Expert Systems Usage for Software Requirements Development in the Strategic Defense Initiative Environment

YI-TZUU CHIEN AND JAY LIEBOWITZ

Navy Center for Applied Research in Artificial Intelligence
Information Technology Division

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 8968	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b. OFFICE SYMBOL (If applicable) 7510	7a. NAME OF MONITORING ORGANIZATION Naval Research Laboratory
6c. ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000	7b. ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Chief of Naval Research 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63223C; PROJECT NO.; TASK NO. BM-1.3.1; WORK UNIT ACCESSION NO. DN155-097

11. TITLE (Include Security Classification)
An Investigation of Expert Systems Usage for Software Requirements Development in the Strategic Defense Initiative Environment

12. PERSONAL AUTHOR(S)
Chien, Yi-Tzuu and Liebowitz, Jay*

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM 5/85 TO 7/85	1986 June 10	31

16. SUPPLEMENTARY NOTATION
*Department of Management Science, School of Government and Business Administration, George Washington University, Washington, DC 20052

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Expert systems; Software design; Software requirements; Strategic Defense Initiative

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The Strategic Defense Initiative (SDI) poses a significant challenge to the United States, namely how to counter the threat of Soviet nuclear ballistic missiles. Part of the solution to this problem involves the development of sophisticated automatic programming techniques. Embedded within these techniques is how to develop and check the consistency of "evolving" SDI-related software requirements.

This report addresses what has been done in developing tools for software requirements determination and what has to be done in developing future software requirements tools for the SDI environment. The use of expert systems is an inherent part of these considerations.

This report first develops a basic foundation on software design. Specifically, its definition, emergence, description, and methodologies and tools are addressed. Afterwards, the SDI environment and its associated problems are described. Then, recommendations for a knowledge-based software requirements development approach are explained.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Jay Liebowitz	22b. TELEPHONE (Include Area Code) (202) 767-2381/676-6969 22c. OFFICE SYMBOL 7510-183:JL:gth

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

CONTENTS

AN INVESTIGATION OF EXPERT SYSTEMS USAGE FOR SOFTWARE REQUIREMENTS DEVELOPMENT IN THE STRATEGIC DEFENSE INITIATIVE ENVIRONMENT

1.0 SOFTWARE DESIGN/REQUIREMENTS

1.1 Software Design: Definition

J. Christopher Jones in his book *Design Methods* [1] cites one-line definitions of design from a number of design philosophers [2]. These descriptions refer to "design" in general and are quite varied; but they do share the common theme and importance of addressing the process of design, not the results [2]:

- Finding the right physical components of a physical structure.
- Decision making, in the face of uncertainty, with high penalties for error.
- A goal-directed problem-solving activity.
- Simulating what we want to make (or do) before we make (or do) it as many times as may be necessary to feel confident in the final result.
- The conditioning factor for those parts of the product that come into contact with people.
- Engineering design is the use of scientific principles, technical information, and imagination in the definition of a mechanical structure, machine, or system to perform prespecified functions with the maximum economy and efficiency.
- Relating product with situation to give satisfaction.
- The imaginative jump from present facts to future possibilities.

In congruence with the above-quoted views of design, Christopher Alexander [3], who is often quoted by those concerned with software design, stresses the desired result of design [2]:

> ... every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble into form and context.

Software design, in particular, deals with producing representations of programs. These representations may be very high-level in detail or very near to actual programs [4]. Software design builds

Manuscript approved November 1, 1985.

coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher levels and on the logical operations involved at the lower levels [4].

Freeman [4] feels that software design, especially at the higher levels, is concerned with the following activities:

- abstracting the operations and data of the task situation so that they may be represented in the system;
- determining precisely what is to be done by the software under design;
- establishing an overall structure of the system;
- establishing interfaces, definite control, and data linkages between parts of the system and between the system and other systems;
- choosing between major design alternatives;
- making tradeoffs dictated by global constraints and conditions in order to meet varied requirements such as reliability, generality, and user-centeredness.

According to Peters [5], seven common ideas pervade the definition of software design:

> First, design is a "good" thing to do prior to implementation. Second, design involves abstraction, including the use of graphics, mock-ups, prototypes, and physical analogies, to strip away detail and to get at the essential character of the system. Third, some rationale is necessary to focus design activity, make it more effective, and ensure that successors will understand what was done. Fourth, design is inexact in that it doesn't lend itself to the use of formulas or precise estimates. Fifth, design is a creative act, uniquely suited to people rather than automated machines, in that people can bring their entire experience to bear on new problems. Sixth, design is a discovery process, in that, as one refines his/her understanding of the problem and enriches one's design to address this new knowledge, one often discovers subtle nuances. Seventh and last, design and analysis (or specification) are inextricably linked and only artificially separable.

Now that software design has been defined, the next section discusses its emergence.

1.2 Software Design: Emergence

One of the earliest uses of the term "software engineering/software design" was in the naming of the first NATO Conference on Software Engineering in 1968 [5]. This meeting and the introduction of the term grew out of concerns on the part of customers and software professionals alike about the cost and quality of the software being produced [5]. These concerns prompted the adoption of many methods and techniques, such as top-down design (this and others are explained in Sections 1.3 and 1.4), each promising to remedy some symptoms of the perceived problem [5].

Other concerns regarding software design were identified at the IFIP (International Federation for Information Processing) Congress in 1971 when Professor Friedrich Bauer [6] believed that an increased application of software engineering/design principles could be of immense benefit to the computer user in solving the following problems:

(a) program duplication—duplication in one's own programming because of ignorance of the work of others, differing languages, change of computing systems or partial change of requirements, and duplication system software, which in the last analysis one has to pay for;

(b) the poor design and implementation of user images and their irrational variation for system to system; and

(c) the management of large application programs—getting them written, used, and maintained.

In the years to follow, other problems were cited for which systematic software was needed. Freeman [4] found the most important reason to design is that the creation of complex systems involves a very large amount of detail and complexity; if this complexity is not controlled, then the desired results will rarely be achieved. Freeman [4] further identifies two more reasons to have proper system design. One such reason is to aid in the discovery of the underlying structure of the problem situation. The other increasingly important reason for design is its impact on system quality. If systems are to contain properties that are global in nature (reliable, user centered, efficient, and portable), then they demand global design decisions [4].

Peters [5] feels that there are three major issues surrounding software design. One class involves technical issues, such as the issue of software design documentation, in which what is practical, not what is theoretically possible, takes the forefront of the discussion. The second major class of issues is conceptual in nature and is related to the more esoteric aspects of software design. The third class of issues relates to the economics of software design, including problems associated with the specification of software designs, the measurement of their quality, and the portability of designs.

To combat these issues and problems, various software design principles and goals have been used to better structure and systematize software designs. Ross et al. [7] give a comprehensive list of principles that should be used to develop software:

- the MODULARITY principle, which defines how to structure a software system appropriately;
- the ABSTRACTION principle, which helps to identify essential properties common to superficially different entities;
- the HIDING principle, which highlights the importance of not merely abstracting common properties but of making inessential information inaccessible (hiding deals with defining and enforcing constraints on access to information);
- the LOCALIZATION principle, which highlights methods for bringing related things together into physical proximity;
- the UNIFORMITY principle, which ensures consistency;
- the COMPLETENESS principle, which ensures that nothing is omitted;
- the CONFIRMABILITY principle, which ensures that information needed to verify correctness has been explicitly stated.

Wasserman [8] adds to this list of principles: reliability, integrity, portability, and adaptability.

These principles are used to affect the process of attaining fundamental goals of software design. Such goals might be [7]: modifiability, efficiency, reliability, and understandability. Modifiability implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained [7]. Efficiency is a much-abused goal, usually because in an excess of zeal it is prematurely permitted a high priority in engineering tradeoffs [7]. Understandability is not merely a property of legibility, but the entire conceptual structure is involved [7].

In the next section, various models of software life cycles are explained, with particular emphasis on the requirements and design phases.

1.3 Software Design: Description

Numerous authors have proposed models to describe the software development life cycle. Three of the most referenced software life models are by Freeman and Wasserman [9], by Metzger [10], and by Boehm [11], as shown in Table 1.1 [5].

Table 1.1 — Three Software Life Cycle Models [5]

<table>
<tr><th colspan="3">Model Authors</th></tr>
<tr><th>Freeman and Wasserman</th><th>Metzger</th><th>Boehm</th></tr>
<tr><td>Needs Analysis</td><td rowspan="2">(System) Definition</td><td>System Requirements</td></tr>
<tr><td>Specification</td><td>Software Requirements</td></tr>
<tr><td>Architectural Design</td><td rowspan="2">Design</td><td>Preliminary Design</td></tr>
<tr><td>Detailed Design</td><td>Detailed Design</td></tr>
<tr><td rowspan="3">Implementation</td><td>Programming</td><td>Code and Debug</td></tr>
<tr><td>System Test</td><td rowspan="3">Test and Preoperations</td></tr>
<tr><td>Acceptance</td></tr>
<tr><td></td><td>Installation and Operation</td></tr>
<tr><td>Maintenance</td><td></td><td>Operation and Maintenance</td></tr>
</table>

These models stress four major phases of the software life cycle. The first phase is system analysis. The objective of the analysis phase is to demonstrate that the customer's problem is understood and to document it in a manner that will aid the design phase. During this phase the customer's problem is externalized, organized, and played back to the customer to ensure that the problem is understood [7].

The second phase of the software life cycle is system design. During this stage, the statement of the problem is addressed through the use of software design methods and techniques to obtain a logical or abstract model of the system software. Implementation issues are not considered, as the goal is a clear perception of a solution concept [7].

System implementation is the third phase and begins with packaging of the logical design. This is followed by implementation of the packaged design in the target programming language and operating system environment, testing of the result, and installation [7].

The fourth major phase deals with system operation. This includes maintenance of the system's performance of original tasks and enhancement to meet changing requirements; the phase leads to the eventual phaseout and replacement of the system [7].

These four phases are the typical common elements found in most software life cycle models. Barry Boehm's model, shown in Fig. 1.1, is one of the most used representations of software life cycles.

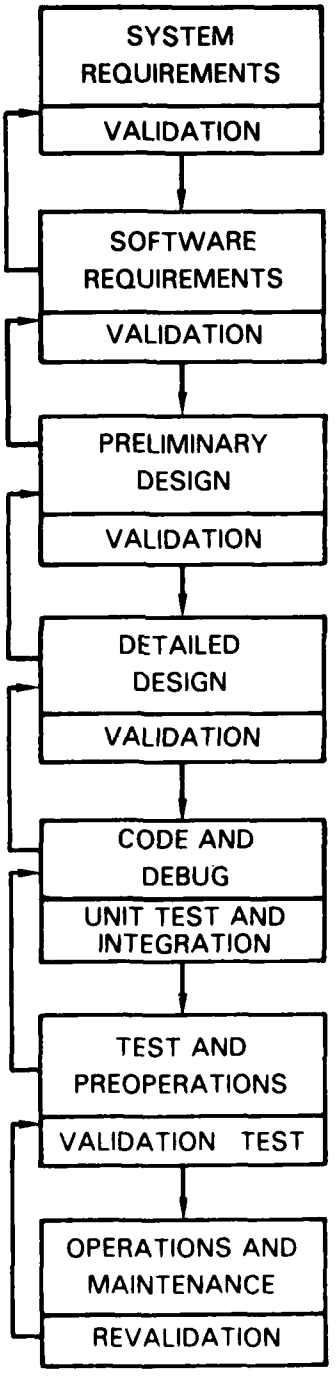


Fig. 1.1 — Boehm's software life cycle model [1]

Boehm's model is best explained by Bruce and Pederson in their book, *The Software Development Project* [12]:

> First, requirements analysis tasks are performed to establish a requirements baseline. This is later used to measure and validate the design products. The requirements are then analyzed and allocated to functional software areas, such as computer programs and modules. This results in a preliminary design that reflects all the requirements and provides the baseline for the detailed design phase. Further analysis and design work on the approved preliminary design baseline results in the detailed design that forms the baseline for the implementation and operation phase. During the implementation and operation phase, the actual coding and testing occur. These activities are controlled through development and test baselines of physical code, programmer documentation, and test data and procedures. After the software is put into the operational environment, the operations and maintenance activities are controlled through version baselines, consisting of specific releases of the software, formal documentation, and test procedures and results. Each baseline is documented and formally reviewed by the development personnel, other project personnel, company experts and, in most cases, customer and user personnel. These documents and reviews provide critical, measurable milestones during the entire software development process.

The documents, referred to in Ref. 12, are shown in Fig. 1.2 [12]. The system and software requirements specifications have the following purposes [12]:

(a) Provide a clear definition of the job to be done.

(b) Allow the project manager and customer to understand what is to be done and agree on the means to do it.

(c) Provide the customer with the option to accept or not to accept the end product(s) by means of a formal acceptance test program.

(d) Provide the test team with the requirements that must be demonstrated.

(e) Provide for approval in writing by the customer. Ideally, once approved, the requirements do not change. If requirements changes are necessary, the impact must be evaluated and the contract changed accordingly.

The software functional design and detailed design documents are used for the following motives [12]:

(a) Provide the project manager and customer with the assurance that the software end products have been systematically defined and designed.

(b) Enable the managers (and customer) to compare the code with the design and to understand project progress.

(c) Are approved by the project manager and are the basis for customer reviews.

(d) Are updated after completion of testing to reflect the "as-built" software product configuration. The updated documents provide the basis for delivery and subsequent software maintenance.

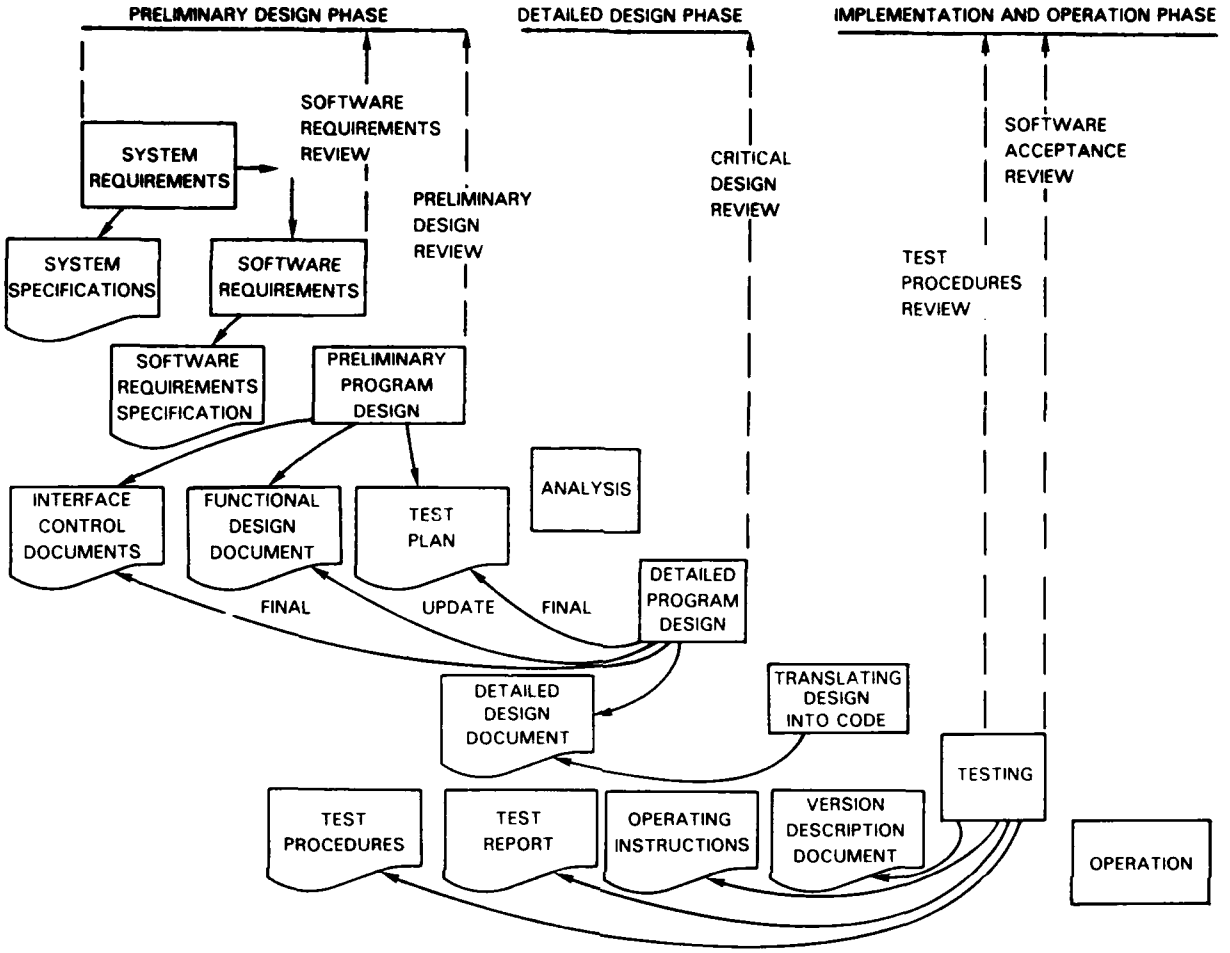


Fig. 1.2 — Software development products [12]

A weakness with Boehm's model, as apparent in other software life cycle models, is that a structured approach is presented only for those situations in which *new* products are to be created [13]. No mention is given to the usefulness of analogous products or the integration of these products with the manager or software designer [13]. To account for this discrepancy, Silverman [14] has developed an analogical view of the software engineering life cycle.

Figure 1.3 shows the analogical view of the software engineering life cycle. Requirements identification, design development, test and integration, operation and maintenance, and "disposal" are steps that are identical to the classical view of software engineering, as in Boehm's model. The major refinement of the classical view is the influence of two added steps—analogous programs and products, and collection. Analogous programs and products are used throughout each phase of the life cycle in which managers and programmers approach new software efforts with what they already know [14,15]. This is especially true, for example, in NASA Goddard's command management system environment where software functional requirements for a new satellite are frequently compared to requirements of similar functions of previous satellites [16]. The collection step, as shown in Fig. 1.3, ensures the updating and accumulation of analogous programs and products in building a corporate memory.

Another software life cycle model that is gaining popularity is rapid prototyping. This approach seems to be the preferred method for expert systems development. Rapid prototyping involves modeling a subset of the problem where systems design, coding, and testing are performed on that subset.

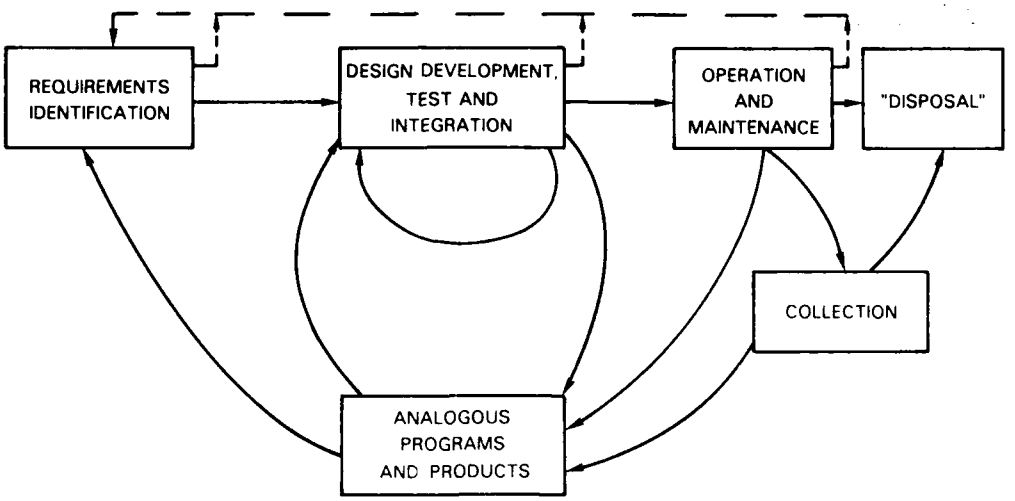


Fig. 1.3 — The analogical view of the software engineering life cycle [14]

Then, through iterative refinements, the subset is modified, enhanced, and enlarged to eventually encompass the total problem application. Here, the complete life cycle process is performed on successive pieces of the problem.

The ability to develop software requirements is a critical part of the software life cycle. Zelkowitz et al. [17] found that defects in software are of two kinds: (a) inconsistency with design or specifications that causes the program to do other than that which is desired by the user; and (b) errors in the program logic that cause the software to operate inconsistently with the written requirements or the intent of the programmer [12]. Shaw and Atkins [18] estimate as much as 50% of the total development effort may go into establishing the requirements for the software [12]. Figure 1.4 by Tausworthe [19] shows what elements go into developing software requirements.

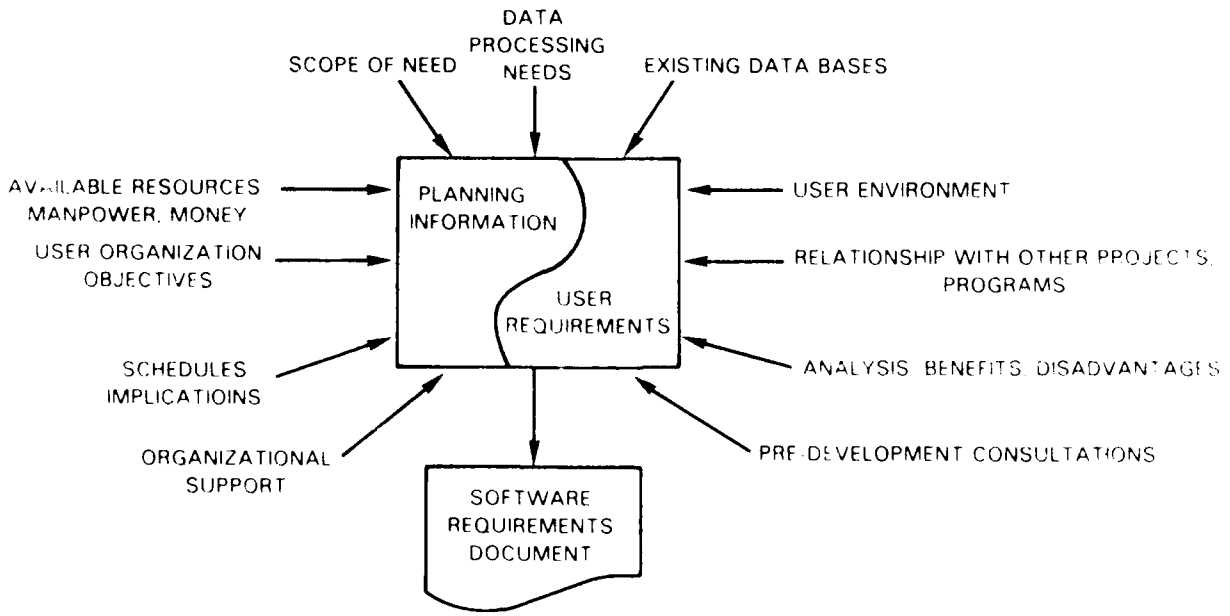


Fig. 1.4 — The software requirement and what goes into developing it [19]

1.4 Specific Software Requirements/Design Methodologies and Tools

To better develop requirements, various methodologies and tools have been used. High-quality surveys by Teichroew [20], Cougar [21], Burns et al. [22], and Reifer [23] provide reporting on these methods [24]. Of particular interest is the survey by Burns et al. which resulted in three major conclusions [24]:

(a) No methodology was defined in sufficient detail that tackles the problem of real-time software requirements and software development. Those methodologies that are applicable (even though partially) are not completely implemented. Scientific application was deemphasized, and intensive real-time computational needs are generally unmentioned.

(b) Systems generally had little emphasis on how to state requirements. Those examined were either a concise description of a methodology without any apparent implementation or some general ideas never formally stated.

(c) Many automated tools were available but were not integrated into a usable system.

Among the systems included in the review were [24]: the accurately defined system (ADS) by NCRS, the time automated grid (TAG) system by IBM, the hierarchy and input process output (HIPO) system by IBM, the information system design and optimization system (ISDOS) developed at the University of Michigan, LOGOS developed at Case Western Research, AUTASIM developed by General Research Corporation, the eclectic Model Driven approach by TRW, and the Systems Optimization and Design Algorithm (SODA). Management approaches such as Computer Sciences Corporation (CSC) Threads and Chief Programmer Team were also included, as well as an additional 19 languages that were reviewed for their ability to support a requirements methodology [24].

Some of the most frequently used methodologies to develop requirements and design software, in general, are: top-down design, outside-in, inside-out, bottom-up, most-critical-component-first, and modular decomposition.

The best-known method is top-down design, also referred to as hierarchical decomposition or stepwise refinement. Under this method, decisions to be made at any point in time are those that affect the greatest possible amount of the total design [2]. The order then follows: if one has grouped decisions into classes or levels, then one makes decisions at the highest level first and then iteratively makes decisions at the lower level [2]. The general caveat in top-down design is to make decisions that take into account as many as possible of the relevant design goals and constraints and that restrict the set of alternatives for lower level decisions as little as possible [2].

Closely related to top-down design is the outside-in method. Basically it is the same as top-down, only the sense of direction is defined in terms of the outside of the system (what the user sees) versus the inside (the implementation) [2]. This approach emphasizes attention to the needs of the end users [2].

The inside-out method is another approach to software design. When using this method, one makes decisions relating to the implementation of the system first before making decisions concerning the external functions of the system [2]. This method tends to define less closely what decisions are to be made in contrast to the top-down and outside-in methods [2].

Another method often discussed is the bottom-up approach, or programming by action clusters. In this method, one makes the lowest level decisions first and gradually builds up the capabilities of the system [2]. The decisions to be made are determined by where one is in the process, starting with decisions concerning basic building blocks and internal functions of the system, and proceeding up to decisions concerning external functions [2].

The most-critical-component-first method is used by designing first those parts of the system whose operation is most constrained [2]. The criteria for making decisions is to make them so that the desired critical parameters are satisfied [2]. Once the critical components are designed, decisions can be made according to some other method [2].

A final frequently used design method is modular decomposition. Software modularity refers to the systematic programming techniques that are used to build reliable software [8]. The program or system is decomposed into pieces, or modules. Modularization is just the implementation of the "divide and conquer" approach to problem solving [25]. Modules can be used to hide design decisions, and through their use, later changes in the design are easy to make [25]. Organizing design decisions according to which are likely to change and then hiding the changeable ones in modules is a specific software design approach called "information hiding."

Apart from the aforementioned software design methods, there are numerous methods and tools specifically geared to software requirements development. These approaches and techniques can be grouped into three major classes: data flow-oriented methods, data structure-oriented methods, and prescriptive methods [5].

1.4.1 Data Flow-Oriented Methods for Developing Software Requirements

Data flow-oriented methods are perhaps the most widely used approaches in industry. These methods and tools advocate the identification and observation of the information that flows through the system; the structure and other characteristics of the resultant flow network are used as the basis for design [5]. Table 1.2 gives a comparison of three commonly used data-flow methods [5]: structured design, structured analysis and design technique (SADT), and systematic activity modeling method (SAMM).

Of these three methods of developing requirements relating to software design, SADT is the most popular technique. SADT is a methodology developed by Douglas T. Ross and is useful for requirements analysis as well as for design [26]. It is a general-purpose modeling technique that is applicable to a wide range of problems, not just computer applications [27]. It has been in use since 1974 by several organizations and is relatively well-known in the software engineering field [27].

SADT consists of three things: a set of methods that assist the analyst in understanding a complex subject, a graphical language for communicating that understanding, and a set of management and human factors considerations for guiding and controlling the use of the methods and language [27].

The methods of SADT are based on several concepts: Top-down decomposition is used to break complex topics up into small pieces that can be understood more readily. Model building provides both a way of communicating and a way of understanding through abstraction from the real world. Establishing and using explicit viewpoints and purposes for each model will help to control and limit the information in a model. Review and iteration are used to ensure the quality of the model. Complementary analysis approaches are used to build on the "activity/object duality" of most situations [27].

A SADT model is an ordered collection of diagrams. The number of diagrams in a model is determined by the breadth and depth of analysis that is required for the purpose of that particular model. The management techniques of SADT have been chosen to coordinate and obtain the best results from the technical methods and tools. Included are document control procedures to keep track of the various stages of a model, review and approval standards for individual diagrams, and project estimation guidelines [27].

SADT supports the basic purposes of the early stages of development since its methods help one understand a subject, and the graphical language provides a flexible way of communicating one's understanding to others. SADT adheres to the explicit design approach. One normally uses it to build several distinct models during development: a model of current operations, a model of functional

Table 1.2 — Comparison of the Data Flow-Oriented Methods [5]

Characteristic	Software Design Method		
	Structured Design	Structured Analysis and Design Technique	Systematic Activity Modeling Method
Current System Modeling	Yes	Yes	Yes
System Specification	Yes	Yes	Yes
System Architecture	Yes	Somewhat	Somewhat
Logical Design	Yes	Yes	Yes
Physical Design	Yes	Potentially	Potentially
Availability of Tutorial Materials	High	Low	Low
Availability of Training Courses	Publicly	By arrangement	By arrangement
Adaptability to Current Management Approach	High	Low	Moderate
Ease of Use (High - Easy to Use)	High	Low	Moderate
Learning Effectiveness	High	Low	Moderate
Communication with Customers	High	Low	High
Hierarchical in Nature	Yes	Yes	Yes
Proliferation Level	High	Moderate	Low
Provision of Objective Evaluation Criteria	Yes	No	Consistency only
Basis of Method	Conceptual	Conceptual/ procedural	Conceptual/ procedural
Degree of Technical Issue Coverage	4 out of 4	3 out of 4	3 out of 4
Support by an Automated Tool	No	Yes	Yes
Support by Qualified Consultants	Yes	Yes	By arrangement
Most Portable Feature (if any)	Coupling and cohesion	Data/ control modeling	Level-by-level consistency

requirements, a model of the system design, and perhaps models of specific topics such as error recovery. SADT permits nontechnical people to express their needs in a form that meets the technical requirements of complex systems analysis [27].

Another popular data flow-oriented requirements method is software requirements engineering methodology (SREM). SREM was developed under the direction of the U.S. Army Ballistic Missile Defense program by several subcontractors and has been described extensively. Its usage to date, however, has been primarily geared to the specification of large missile-defense systems [27].

SREM focuses on techniques applicable to real-time systems; in particular, it incorporates a "stimulus-response" model of real-time systems. It includes a graphical language and utilizes sophisticated graphical displays. It incorporates automated simulation facilities to provide the analyst additional feedback on the characteristics of the system being specified [27].

The language used for stating requirements in SREM is called requirements statement language (RSL); it also has a graphical form called R-nets. These languages permit one to express parallel operation, specify explicit interfaces to other subsystems, and tie validation assertions to particular points in the specification [27].

1.4.2 Data Structure-Oriented and Prescriptive Methods for Developing Software Requirements

Data structure-oriented methods and tools take a similar view of data, as that expressed under data flow-oriented methods, but advocate observing data at rest. The emphasis is on identifying and observing logical relationships between discernible data elements, for these relationships form the basis of the program itself. Table 1.3 shows a comparison of commonly employed data structure-oriented methods [5]: Jackson's method, logical construction of programs, and structured systems development.

The third and last major class of methods and tools used for software requirements development is prescriptive methods [5]. Prescriptive methods generally do not possess an underlying rationale [5]. Instead, the emphasis is on a prescribed regimen, in that these methods dictate procedures the software designer must follow to ensure success [5]. Table 1.4 presents a comparison of frequently used prescriptive methods [5].

One of the more popular prescriptive methods is called Program Design Language (PDL). PDL is designed for the production of structured designs in a top-down manner. It is a "pidgin" language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language). In a sense, it can be thought of as "structured English" [28].

Although the use of pidgin languages is also advocated by others, further steps have been taken of imposing a degree of formalism on the language and supplying a processor for it. Input to the processor consists of control information and designs for procedures (called "segments" in PDL). The output is a working design document that can, if desired, be photoreduced and included in a project development workbook. The output of the processor completely replaces flowcharts since PDL designs are easier to produce, easier to change, and easier to read than are designs presented in flowchart form [28].

Sections 1.4.1 and 1.4.2 cover well-established design methodologies, most of which have been used in practical projects. The next few sections discuss some experimental software design tools and basic research projects, most of which have not been used in any real software developments.

1.4.3 Programmer's Apprentice

Another tool for developing software design and requirements deserves mention. This tool is called the programmer's apprentice (PA) and was developed by Rich, Waters, Shrobe, Hewitt, and Smith at M.I.T. The following paragraphs address the PA.

Table 1.3 — Comparison of Data Structure-Oriented Methods [5]

Characteristic	Software Design Method		
	Jackson's Method	Logical Construction of Programs	Structured System Development
Current System Modeling	No	No	No
System Specification	No	No	No
System Architecture	Yes	Yes	Yes
Logical Design	Somewhat	Somewhat	Somewhat
Physical Design	Yes	Yes	Yes
Availability of Tutorial Materials	Moderate	High	High
Availability of Training Courses	By arrangement	Publicly	Publicly
Adaptability to Current Management Approach	High	High	High
Ease of Use (High—Easy to Use)	Low	High	High
Learning Effectiveness	Low	Moderate	Moderate
Communication with Customers	Low	Moderate	Moderate
Hierarchical in Nature	Yes	Yes	Yes
Proliferation Level	Moderate	Moderate	Moderate
Provision of Objective Evaluation Criteria	Somewhat	No	No
Basis of Method	Conceptual/ procedural	Conceptual/ procedural	Conceptual/ procedural
Degree of Technical Issue Coverage	3 out of 4	3 out of 4	3 out of 4
Support by an Automated Tool	No	No	No
Support by Qualified Consultants	By arrangement	Yes	Yes
Most Portable Feature (if any)	Data structure modeling	Data structure modeling	Data structure modeling

Table 1.4 — Comparison of the Prescriptive Methods [5]

Characteristic	Software Design Method						
	Chapin's Approach	Design by Objectives	Design by Pad	Higher Order Software	Information Hiding	Meta Stepwise Refinement	Program Design Language
Current System Modeling	No	Yes	Yes	No	No	No	Yes
System Specification	No	Yes	No	Potential	No	No	No
System Architecture	No	No	No	No	Yes	No	No
Logical Design	No	No	No	No	Yes	No	No
Physical Design	Yes	Potential	Yes	Yes	Yes	Yes	Yes
Availability of Tutorial Materials	High	High	Low	High	Moderate	Low	Moderate
Availability of Training Courses	By arrange-ment	By arrange-ment	—	By arrange-ment	—	—	By arrange-ment
Adaptability to Current Manage-ment Approach	High	Moderate	High	Moderate	High	High	High
Ease of Use (High-Easy to Use)	Moderate	Low	High	Low	Moderate	Moderate	High
Learning Effectiveness	High	Moderate	High	Low	Moderate	High	High
Communication with Customers	Moderate	High	High	Low	Moderate	—	High
Hierarchical in Nature	Yes	Somewhat	Yes	Yes	Yes	Yes	Yes
Proliferation Level	Low	Low	Low	Low	Moderate	—	Moderately High
Provision of Objective Evalua-tion Criteria	Structured design	Yes	Somewhat	Somewhat	Yes	—	—
Basis of Method	Heuristic	Conceptual/ Heuristic	Heuristic	Mathematics	Conceptual	Conceptual	Conceptual
Degree of Technical Issue Coverage	2 out of 4	2 out of 4	3 out of 4	2 out of 4	3 out of 4	— —	2 out of 4
Support by an Automated Tool	No	No	No	Yes	No	No	Yes
Support by Qualified Consultants	By arrange-ment	By arrange-ment	No	By arrange-ment	By arrange-ment	No	By arrange-ment
Most Portable Feature (if any)	Decomposition heuristic	Conceptual	Not	Axioms	Conceptual	Conceptual	Pseudocode

The PA is a computer-aided design tool whose eventual purpose is to help a programmer deal with program evolution from the initial design phase through the continuing maintenance phase. In this capacity, the PA functions as a "junior programmer" might. Rather than being able to certify the correctness of an entire software system, the PA instead provides the designers and coders with relevant information that helps them to better understand the consequences of modifications. The PA is relatively knowledgeable but not an expert programmer itself. It is able to understand, explain, and reason about programs in terms familiar to a programmer [29]. Its main virtue is its ability to select, from the vast annotation associated with a software system, the small part that is relevant to whatever the programmer is currently conducting [30]. Its intention is that the programmer will do the hard parts of design and implementation while the PA will assist him/her wherever possible. Hence, the PA is designed to be midway between an improved programming methodology and an automatic programming system [31].

The current system is designed in LISP and is composed of five parts: an analyzer, a coder, a drawer, a library of plans, and a plan editor. Figure 1.5 shows the architecture of the current implementation of the PA [30]. Given the text for a piece of a program, the analyzer module can construct a plan corresponding to it. A plan under the PA terminology is a representation for programs that abstracts away from the inessential features of a program and represents the basic logical properties of the algorithm explicitly. The coder module performs the reverse transformation, creating program text corresponding to a plan. The drawer module can draw a graphical representation of a plan. The library contains common program fragments represented as plans. The plan editor makes it possible for the programmer to modify a program by modifying its plan [30].

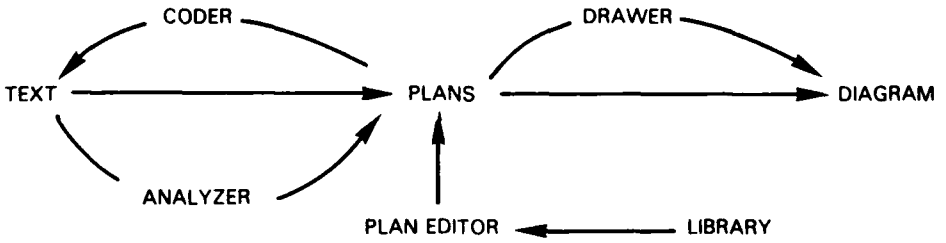


Fig. 1.5 — The architecture of the current implementation of the PA [30]

By creating such a system, several benefits result in which the PA assists the programmer [29]:

- Programming with cliches: The programmer can construct programs faster and with fewer errors by combining and modifying entries from the PA's library of standard algorithm and data structure fragments, as compared with using a standard text or structure editor.
- Flexible display: The PA can display programs at different levels of detail and from different points of view.
- Propagation of design decisions: As the programmer makes more and more design decisions, other decisions become forced. When this happens, the PA makes them automatically.
- Bug detection: Whenever the programmer makes a design decision that contradicts earlier decisions, the PA reports this fact as a bug.
- Automatic modification: The PA has explicit knowledge about many common kinds of modifications and can perform them automatically when requested.
- Automatic optimization: The PA knows how to perform a number of common optimizations and is able to recognize situations in which they should be applied.

- Escape to the surrounding environment: At any time, the programmer can step outside of the PA and use the tools in the surrounding environment.

As there are strengths of the PA, weaknesses are also evident as stressed by Rich and Waters [29]. Thus far, no effort has been expended toward making the PA work with reasonable speed. There are therefore many bottlenecks that need to be removed. For example, many of the existing modules of the PA demonstration systems need to be made more incremental. Currently, the PA does a great deal of unnecessary recomputation. By breaking up internal tasks into smaller units and making greater use of dependencies, it should be possible to improve performance in this area [29].

A related issue is the PA's use of space. This is computationally very convenient but takes up an unreasonable amount of space in memory and in long-term storage. A possible solution is to use a scheme wherein only the knowledge that is currently being used is represented in full, while other parts of the knowledge base are represented more compactly. The key to doing this is segmenting the knowledge and determining a small subset of the information stored in a plan that can be used as the basis for rapidly recomputing the rest [29].

Finally, the modules developed for the feasibility demonstrations need to be made considerably more robust to be suitable for routine use [29, 32]. Some of the modules, such as the one that converts program text into a simple plan, have seen a good deal of use and are approaching reliability. Other modules need to be rewritten. For example, the algorithm used in the coder module is too simple-minded. To reliably produce aesthetic program text, it needs to be replaced by an algorithm that explicitly reasons about trade-offs between readability criteria [29].

Despite these weaknesses, the PA is a sophisticated software design tool and is a step in the right direction to improving the software engineering process.

1.4.4 Innovator

Another tool is currently under development, and it aids software designers in identifying problem areas in software development and in determining a model, for usage, that most closely matches the problem. This automated tool has its roots in analogical reasoning processes and is being developed by Silverman, Nakamura, and Suite [33].

The complete program is menu-driven with four knowledge base areas [33]: military software systems, project management software, artificial intelligence software, and energy and environmental models. After the user selects the knowledge base area, the computer then poses several questions in which the answers identify the problem area. To determine this problem area, a rule-based expert system is employed that uses deterministic selection [33]. This expert system is based upon DIAGNOSE by Winston and Horn [34]. Once the problem area is identified, another expert system is activated in which questions are posed to the user for determining what models from the data base most closely match the problem [33]. This second expert system uses Nakamura's similarity network [35]. This expert system is enriched with fuzzy set logic, analogical reasoning, and psychological similarities for aiding the user in clarifying his/her ambiguous request [33]. After this second expert system discovers what models most closely resemble the problem, another subsystem is activated and applies the models in solving the problem. This last subsystem, called *usage information*, has not yet been developed.

Thus far, the microcomputer-based innovator has been used successfully in the military software systems area. Further work is being conducted to develop the other three aforementioned knowledge base areas.

1.4.5 Intelligent Program Editor

The intelligent program editor (IPE), designed by Advanced Information and Decision Systems, is another tool being developed that is aimed at improving software requirements and design. The editor

will support program development and maintenance activities by providing knowledge-based systems techniques for searching through programs, manipulating programs, analyzing programs for potential errors and good style, and maintaining structured documentation [36]. As a user moves through the various steps of program development, the system will [36]:

- employ templates to aid in the functional definition steps,
- provide access to libraries of typical programming procedures to aid in the development of the detailed algorithms,
- provide general knowledge about the types of data structures that may be employed and some guidelines for the benefits and drawbacks of the various data structures,
- provide some automated checking for coding errors, and
- even partially automate the construction of code once the functional and algorithmic descriptions have been provided.

The IPE consists of a user interface, an editing executive, tools, a programming context model, and an extended program model. The interaction between the system and the user is handled by the user interface. All of the supporting tools exercised by the user are managed by the editing executive. Knowledge about the types of editing contexts in which a programmer might be operating, and the types of tools and information required to support that context, is provided in the programming context model accessed by the editing executive. The description of the actual programs together with all of the associated information (documentation, algorithm descriptions, functional descriptions, etc.) is maintained by the extended program model. The extended program model defines a vocabulary for discussing programs that uses terms that are much closer to the ones that programmers naturally employ. It also contains a knowledge base that documents the composition of particular programs. Embedded within the extended program model are two subsystems: the program reference language (PRL) and the program structures data base (PSDB). PRL's main goal is to take a description of a program fragment, compare it against the knowledge base of the structures present within the program, and return the matches that are found [36]. The PSDB maintains the knowledge by checking for consistency and semantic integrity constraints.

The IPE has the potential for greatly adding to the software development process.

1.4.6 Knowledge-Based Software Assistant

The Psi project at Stanford University was one of the first major activities on automatic program synthesis. The Psi efforts led into the Chi project at the Kestrel Institute. Chi incorporates many of the specifications-based programming principles suggested for large program development projects [37, 38]. The Chi activities are molding into the Knowledge-Based Software Assistant (KBSA) at Kestrel Institute. This knowledge-based software paradigm of the future will provide a set of tools and capabilities integrated into an "assistant" that directly supports the human developers in the requirements analysis, specification, implementation, and maintenance processes [39].

The long-term goal of the requirements part of this assistant is to provide the following [39]: comprehensive requirements management, intelligent editing of requirements, testing of requirements for completeness and consistency, performing requirements reviews, maintaining and transforming requirements in response to changes, decomposing and refining requirements into executable specification languages, and acquiring requirements knowledge.

Short-term goals include [39]:

- analysis of requirements problem definition,
- a formal requirements language,
- smart editing and managing of requirements,
- reviewing requirements definitions for the user, and
- requirements testing.

Some of these goals are quite ambitious, but they are the steps needed to formalize and improve requirements definition, completeness, consistency, and validation. Reasoning Systems (Palo Alto) and Kestrel Institute are performing parts of the KBSA proposal [37].

1.4.7 Other Miscellaneous Work in Knowledge-Based Software Requirements Design

Besides the work previously mentioned, there has been a spattering of other related work performed in using expert systems for software requirements development. Fickas et al. [40, 41] at the University of Oregon have developed a knowledge-based software specification environment, called ORBS. Greenspan [42] at the University of Toronto has created a knowledge representation approach to software requirements definition for requirements modeling. Intermetrics, in Cambridge, Massachusetts, has developed an expert system for compiler code generation [37]. An expert system has also been designed by the University of Waterloo for real-time debugging [37]. MCC, in Austin, Texas, is working on Project Leonardo for using expert systems for automatic generation of requirements. Additionally, the Software Engineering Institute, at Carnegie-Mellon University, is looking at ways of using expert systems for assuring that software engineering practice throughout the mission critical software community achieves and maintains a high level of effectiveness [43]. The DoD STARS (Software Technology for Adaptable, Reliable Systems) program is looking at developing a project manager's tool set and acquisition program manager's assistant. These activities involve the use of expert systems for project management. Liebowitz [16] developed an expert system prototype for determining software functional requirements for command management activities of NASA-supported satellites.

As shown, there has been some work accomplished in employing expert system technology for software requirements development. It appears that more work is starting to be done along these lines. With productivity rising at the rate of only about 5% a year [37], software engineering is a fruitful area for expert system exploration and implementation.

Now that a thorough appreciation for the activities involved in software requirements/design has been gained, the next section describes the Strategic Defense Initiative environment and some of the problems involved in designing expert systems in that domain for requirements generation. Afterwards, some specific recommendations are made for developing expert systems for software requirements generation in the Strategic Defense Initiative environment.

2.0 DESCRIPTION OF THE STRATEGIC DEFENSE INITIATIVE ENVIRONMENT

The Strategic Defense Initiative (SDI) was formed in response to President Reagan's speech on March 23, 1983 in which he urged the United States to investigate whether new technologies could provide the means for countering the awesome threat of nuclear ballistic missiles. The goal would be [44]:

> ... a globe-girdling network of permanently orbiting battle stations capable of shooting down Soviet intercontinental ballistic missiles shortly after they lifted off or of destroying the nuclear warheads that have separated from the booster rocket and are hurtling silently in the vacuum of space over the Arctic.

To develop an effective defense against ballistic missiles, several programs were proposed in the areas of [45]:

- surveillance, acquisition, and tracking;
- directed energy weapons;
- conventional weapons;
- battle management, communications, and data processing;
- systems concepts; and
- countermeasures and tactics.

These programs would help to accomplish the goal of the SDI.

An essential component of the SDI is the use of a layered defensive system. This multitiered defense system would comprise the Ballistic Missile Defense (BMD). These layers refer to the ballistic missile's boost, midcourse, and terminal phases. In the boost phase, the rocket engines accelerate the missile payload through and out of the atmosphere whereafter multiple warheads and penetration aids are released from a postboost vehicle (bus deployment). It is most important to destroy the missiles during the boost phase, because each "kill" would eliminate the need to deal with many warheads and decoys later. A missile remains in the boost phase for about 3 minutes, whereafter multiple, independently targetable reentry vehicles (MIRVs) are deployed by the missile during the next approximate 8 minutes, in which 10 real warheads and 100 decoys could be released. In the midcourse phase, during the next approximate 15 minutes, the warheads and penetration aids travel on trajectories above the atmosphere, in which there are bombs and decoys. Sensors must distinguish live reentry vehicles (RVs) from decoys, destroyed RVs, and debris. During the next 3 minutes, in the terminal phase, the warheads reenter the atmosphere, where they are affected by atmospheric drag. In each phase, a defensive system must perform three basic functions [46]: first—surveillance, acquisition, and tracking; second—intercept and target destruction; and third—battle management.

To destroy the missiles, several weapons have been proposed. Some of these include:

- kinetic-energy weapons ("smart rocks"—a self-guided projectile that homes in on a missile or an RV and slams into it);
- directed-energy weapons (emit powerful beams that can knock a target off course, destroy its electronics, or partially melt it)
  - neutral-particle beam
  - chemical laser (space or ground-based)
  - X-ray laser

Battle stations, perhaps as many as 100, might have to be built. A battle station might contain a weapon concentrating millions of watts of energy in a beam that would bounce off a 30-ft-wide mirror that swirls to aim at a series of rockets and warheads [44].

For this Ballistic Missile Defense system to exist, several critical technologies must be examined. These areas are identified by the Defensive Technologies Study Team [45]:

- Threat Clouds—Dense concentrations of reentry vehicles, decoys, and debris in great numbers must be identified and sorted out during the midcourse phase and high reentry.
- Survivability—A combination of tactics and mechanisms must be developed to ensure the survival of the system's space-based components.

- Boost and Postboost Phases—The ability to effectively respond to an unconstrained threat is strongly dependent on meeting it appropriately during the boost and postboost phases.

- Interceptors—Interceptors must be economical enough to permit attacks on threatening objects that cannot be discriminated.

- Battle Management—Tools are needed for developing battle management software.

An informed decision on system development cannot be made before the end of the decade [45]. Near-term demonstrations of possible technologies in the BMD environment could perhaps be shown. Among these technologies is the use of expert systems in the BMD environment.

In the next section, some of the problems associated with developing expert systems in the SDI environment are discussed.

2.1 Problems with Developing Expert Systems in the SDI Domain

Several difficulties exist in designing expert systems for the SDI environment. One important area is that the expert system must be "evolvable." Since new technologies and SDI-related requirements will be discovered over the next 15 years, the expert system must be able to easily handle the integration of this new knowledge into the old. Thus, a very modular approach to designing the knowledge base is needed, along with a fairly easy semiautomated (or automated) way of acquiring knowledge and integrating it into the knowledge base. A way of checking for consistency of information and possible conflicts of existing and new requirements must be captured in the expert system.

Another area of difficulty lies in developing expert systems where there may be no experts who exist as of the time of building the expert system. Experts may exist for developing parts of the knowledge base; however, the knowledge base may have to capture knowledge in which there are no experts that currently exist. For example, the effects of chemical lasers or neutral particle beams on certain objects are areas that are not known since these weapons have not been built, tested, and used against ballistic missiles. Thus, at the present time, the knowledge base may not be able to include such information, but as time evolves, this information may be obtained and encoded. This suggests that current expert system developments should focus on either a very narrow problem domain, or concentrate on a larger domain but be sure that the knowledge base can be evolvable.

Another possible problem associated with developing expert systems presently for SDI environment is that there are great unknowns. Uncertainty exists such as in the BMD system's architecture, the weapons selected for the BMD system, and the constraints on the computing power. These uncertainties will be lessened as the years progress; however, at the present time, it is not known with surety what the weapons and actual BMD environment will look like in the years 1995 to 2000. Again, it is critically important for the expert system built today for the SDI environment to be able to handle uncertainties and be modular in design for knowledge base evolvability.

3.0 RECOMMENDATIONS FOR AN EXPERT SYSTEM FOR SOFTWARE REQUIREMENTS DEVELOPMENT IN THE SDI ENVIRONMENT

In developing an expert system for software requirements determination in the SDI environment, there are many issues of importance that need to be addressed. Some of these issues include:

- develop formal requirements language—needed for standardization and testing;

- develop techniques for editing and managing requirements;

- review requirements definition from users;

- develop techniques for requirements testing for matching consistency of old requirements with new ones;
- develop ways for capturing and maintaining requirements;
- develop ways of separating the functional requirements from the operational and performance requirements;
- develop ways of breaking down requirements into varying levels of complexity;
- develop ways of accessing analogous requirements that could be used for developing new requirements;
- develop ways of ensuring that the requirements tool gets used;
- develop ways of easily updating requirements;
- develop ways of ensuring proper access to requirements (i.e., security classification);
- develop ways for ensuring completeness, accuracy, and currency of requirements;
- develop techniques to allow distributed access to the knowledge-based tool since the requirements will be developed by different groups in different cities;
- develop ways for determining the feasibility of requirements;
- develop ways of accounting for uncertainty in requirements due to the estimation of new technologies;
- develop backup system for holding the requirements in case of computer failure or tampering;
- develop ways for determining redundancy in requirements; and
- develop techniques for determining maximum use of requirements.

For the scope of this report, three of the most major issues are examined. These are: developing a formal requirements language, developing a technique for editing and managing requirements, and developing a way of accessing analogous requirements that could be used for determining new requirements. Some initial thoughts are made on each of these areas, with an emphasis on various objectives to set forth for accomplishing these goals.

3.1 Developing a Formal Requirements Language

Before a knowledge-based expert system can be built on generating and checking requirements, it is most crucial to have a common vocabulary in order to express requirements. This suggests the need for developing a formal requirements language. To develop requirements over the next 5 to 10 years for the Strategic Defense Initiative project, there must be a set of standardized languages and terms for expressing, validating, and evaluating requirements.

This does not mean that there should be only one requirements language for all application areas in the SDI environment. Rather, there should be one requirements language used within an application area. Thus, an application-specific formal requirements language is needed where the terminology of the application area is suitably incorporated into the syntax and semantics of the language [47]. There should not be only one requirements language for all phases of the SDI because each application area

requires terminology. For example, a language that describes the call processing aspects of a telecommunications switch should employ different terminology from one that describes process control or avionics applications [47]. To generate each application-specific requirements language, some requirements language processor is needed to check for inconsistency, redundancy, and incompleteness in a specification [47].

This requirements language processor should produce a combination of formal specifications and text strings. It might even paraphrase a formal specification into natural language. It should employ formal semantics for error checking, consistency and compatibility analysis, and program transformation.

To achieve this, several approaches might be used. One method stems from a cognitive point of view in which a psychological theory is developed about exactly what it is that the programmer does. Once the way is learned on how requirements are developed, the next step would be to model this approach. A more tangible method is to develop requirements by examples. This involves creating scenarios and if aspects of the new scenario are similar to previous scenarios, then the analogous requirements from the previous scenarios could be applied to the new scenario. To develop new requirements from previous analogous requirements, a mechanism is needed to collect previous requirements and match them with new requirements. An initial cut of designing this mechanism is to review documentation of requirements and select the most used (i.e., common) requirements. Next, an examination of what are the characteristics needed to determine these requirements should be made. For the characteristics that have not been considered in determining these requirements, there should be new requirements generated based upon these new characteristics. For standardization, these requirements could be encoded in rules or frames. The rule structure might look like: IF characteristic1 or characteristic2 THEN requirementA. In a frame-like representation, the following slots might be present:

Name of Requirement:
Definition:
Characteristics:
Synonyms:

The use of analogy is further explored in Section 3.3.

There are several advantages in developing formal requirements languages for certain applications within the SDI environment. One major advantage is that standardization leads to easier and better testability of requirements. This is extremely important in the SDI environment as requirements will be evolving over the next 5 to 10 years. Thus, standardization will lead to improved consistency in generating and testing requirements. It should also reduce the complexity of formalisms for expressing requirements. Redundancy of requirements should be lessened with standardization as several expressions for the same requirement will be reduced to one expression. Another advantage of better defining requirements is that modularity of systems will be enhanced. Since modularity is purpose-driven, the better the requirements are defined, the better the understanding of the purpose one is trying to achieve [48]. More standardized requirements language will improve the requirements definition and hence, the system design.

3.2 Editing and Managing Requirements

Poorly defined requirements are believed responsible for many software project failures [42]. Errors made in requirements and design are the most costly to detect and correct [49], and it is hard to get the requirements right in the first place [50]. This suggests that there is a strong need for thorough editing and managing of requirements.

The ability to easily edit requirements is an important feature in developing an expert system for software requirements determination in the SDI environment. As the SDI architecture emerges and

develops over the next 5 to 10 years, the requirements supporting this architecture will also evolve. This stresses the need for having a capability for editing requirements to account for changes in the SDI environment. An intelligent editor should be built that will first ensure the syntactic structure of the formal requirements language. Then, it would be used to trace through the connections of related requirements to ensure consistency. Later, generic descriptions of requirements would be stored in the knowledge base. To ensure consistency of requirements, one must deal with "over-abstraction." For example, one such over-abstraction or contradiction is: A person's heart normally beats between 70 and 90 times a minute, but this is not so in the case of hyperthyroid patients [51]. One way to deal with these contradictions is to allow the designer to specify exceptional classes or objects for which contradictions are explicitly acknowledged and resolved through "excuses" [51]. For example, the assertion that hyperthyroid patients have high blood pressure excuses the assertion that patients in general have normal blood pressure. Of course, appropriate semantics for excuses must be provided so the final specification will be logically consistent [51]. Through the use of an intelligent editor and "help" facilities, a friendly user interface will be created to facilitate changes in requirements definition and analysis.

Not only is editing an important feature of an expert system for determining software requirements, but also managing of requirements is perhaps even more important. The expert system for developing software requirements should have a knowledge acquisition component. This knowledge acquisition component would help the knowledge engineer explore and manage interactions among rules and, more abstractly, the knowledge that those rules represent [52]. It would aid in developing the expert system into a self-modifying program. It might also evaluate the performance of the expert system, suggesting areas that need a generalization of rules, specialization of rules, or new rules to fill reasoning gaps [52]. From a management standpoint, the expert system should be able to group requirements by certain functions (i.e., categorize requirements). It should also be able to decompose requirements into lower levels and separate functional requirements from operational and performance requirements automatically. The management and learning aspects of an expert system for developing requirements are vital but difficult features to capture. More research is needed to create techniques for handling these elements.

3.3 The Use of Analogy in Determining Software Requirements

All humans beings employ analogy. Analogy is the mapping of a target to a base in order to see if the solution of the base could be the solution of the target. If expert systems are to mimic humans, then they should inherently utilize analogy. In the expert systems environment, the theory of frames for representing knowledge developed partly because humans usually solve problems by first seeing if similar kinds of problems have been solved before. If a similar problem has been solved, then perhaps that solution could be applied to the new problem situation. Here, analogy is the underlying concept where particular descriptions or situations (i.e., frames and scripts, respectively) are used as bases and solutions of bases.

Recent work by Silverman [14,15], Liebowitz [16], and Carbonell [53] indicates that there is great merit in using analogy for expert systems development. The use of analogy in software development leads to better reusability of software, a goal of the Department of Defense. Moreover, it could save in startup costs and enhance commonalities/reuse [14].

Since requirements lists are typically reused time and again [14], it would be beneficial to automatically and efficiently retrieve previous requirements that could be used in a new problem situation. To do this, the characteristics that determine each requirement must be determined. Then, given similar characteristics, analogous software functional requirements could be used as the *new* software functional requirements. Having an expert system to do this, as worked on by Liebowitz [16], would facilitate a first cut at developing software functional requirements. Problems, however, arise in cases where technologies are being invented so there are no analogous requirements, as will partly be the case in the SDI environment. For new technologies, rules could be encoded in the knowledge base of the expert system to anticipate possible novel discoveries. These rules could then carry a certainty factor related to the feasibility of developing the particular technology.

For the analogy method to work, an important feature is needed. This feature is that a mechanism is needed for collecting previous requirements. Thus there must be a central source in charge of building and developing the institutional memory. According to Silverman [14], the areas of greatest support needed are: (a) record/keep track of information that distinguishes good and bad decisions from good and bad outcomes; (b) develop a database of relevant analogy information for performance/training support; and (c) record/keep track of potentially relevant comparison cases. Building this institutional memory on requirements is essential for drawing analogous requirements from similar problem characteristics and for capitalizing on an evolutionary process.

The use of analogy for determining SDI requirements can be shown in an example. Let us say that requirements are obtained relating to existing software requirements for weapon systems. These requirements would be gathered by looking through the software functional requirements documentation for United States' weapon systems. One top-level requirement in the documentation might be to "perform attitude determination." The characteristics that are needed to produce this requirement would then be determined, such as:

rule 1 IF nature of mission = commandable
& weapon = pointer
/ weapon = scanner
THEN level one functional requirement = perform attitude determination <1.0>.

Thus, given the SDI environment, if the characteristics of the proposed SDI architecture fit the above rule, then this software functional requirement (perform attitude determination) would be needed for the SDI work. Here, this analogous requirement would be retrieved to be part of the listing of level one functional requirements for SDI.

4.0 SUMMARY

Having a better way of determining software requirements and of coping with changing requirements is an important part of meeting the goals of the Strategic Defense Initiative. This report provided a general background on software design and presented a survey of work being done in using expert systems for software requirements development. At the Naval Research Laboratory, work is currently being done by McLean [54] and Jacob and Froscher [55] on developing a formal method for determining software specifications. More research is needed, however, in solving some of the problems outlined in Section 3. With some of the initial recommendations taken into account, as described in Sections 3.1 to 3.3, steps could begin to be made in developing an expert system for determining SDI-related software requirements.

5.0 ACKNOWLEDGMENTS

The authors thank the NCARAI personnel, too numerous to mention here, for their helpful suggestions during the development of this report. The authors are especially grateful to Dr. Randall Shumaker and to Dr. Robert Jacob for their technical critiques and reviews of the report.

6.0 REFERENCES

1. J.C. Jones, *Design Methods* (Wiley Interscience, 1970).

2. P. Freeman, "The Nature of Design," *Tutorial on Software Design Techniques*, IEEE (1977).

3. C. Alexander, *Notes on the Synthesis of Form* (Harvard University Press, 1964).

4. P. Freeman, "The Context of Design," *Tutorial on Software Design Techniques*, IEEE (1977).

5. L.J. Peters, *Software Design: Methods and Techniques* (Yourdon Press, 1981).

6. F.L. Bauer, *Software Engineering: An Advanced Course* (Springer-Verlag, 1975).

7. D.T. Ross, J.B. Goodenough, and C.A. Irvine, "Software Engineering: Process, Principles, and Goals," *Computer*, May 1975, p. 17.

8. A.I. Wasserman, "Some Principles of User Software Engineering for Information Systems," *Digest of Papers, COMPCON*, spring 1975.

9. P. Freeman and A.I. Wasserman, *Tutorial on Software Design Techniques*, IEEE Computer Society, Apr. 1977.

10. P.W. Metzger, *Managing a Programming Project* (Prentice-Hall, Englewood Cliffs, N.J., 1973).

11. B.W. Boehm, "Software Engineering," *IEEE Trans. Computers* **C-25**, 1226, Dec. 1976.

12. P. Bruce and S.M. Pederson, *The Software Development Project* (John Wiley & Sons, 1982).

13. B.G. Silverman, "The Software Engineering Paradox: Unexploited Cost Savings and Productivity Improvement," *Test and Evaluation* **V**(1), Jan. 1984.

14. B.G. Silverman, "Potential Software Cost and Productivity Improvements: An Analogical View," *Computer*, May 1985, p. 86.

15. B.G. Silverman, "Analogy in Systems Management: A Theoretical Inquiry," *IEEE Trans. Systems, Man, and Cybernetics* **SMC-13** (6), 1049, Nov.-Dec. 1983.

16. J. Liebowitz, "Determining Functional Requirements for NASA Goddard's Command Management System Software Design Using Expert Systems," George Washington University, Dissertation, Dec. 1984.

17. M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon, *Principles of Software Engineering and Design* (Prentice-Hall, 1979).

18. J.C. Shaw and W. Atkins, *Managing Computer System Projects* (McGraw-Hill Book Co., 1970).

19. R.C. Tausworthe, *Standardized Development of Computer Software* (Prentice-Hall, 1977).

20. D. Teichroew, "A Survey of Languages for Stating Requirements for Computer-Based Information Systems," *AFIPS Conference Proc.* **41**, 1203 (1972).

21. J.D. Cougar, "Evolution of Business Systems Analysis Techniques," *Computing Surveys* **5**, 167, Sept. 1973.

22. F. Burns et al., "Current Software Requirements Engineering Technology," TRW-22944-6921-010, Aug. 1974.

23. D. Riefer, "Software Specifications: A Tutorial," COMPCOM Proceedings **76**, 39, Sept. 1976.

24. G. Davis and C.R. Vick, "The Software Development System," *IEEE Trans. Software Eng.* **SE-3**, 69, Jan. 1977.

25. P. Freeman, "Software Reliability and Design: A Survey," Proc. 13th Design Automation Conference, IEEE (1976).

26. D.T. Ross, and K.E. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Software Eng.* **SE-3**, 6, Jan. 1977.

27. P. Freeman, "A Perspective on Requirements Analysis and Specification," *Tutorial on Software Design Techniques*, IEEE (1980).

28. S.H. Caine and E.K. Gordon, "PDL—A Tool for Software Design," *Tutorial on Software Design Techniques*, IEEE (1980).

29. C. Rich and R.C. Waters, "Abstraction, Inspection and Debugging in Programming," MIT AI Memo 634, June 1981.

30. R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," *IEEE Trans. Software Eng.* **SE-8** (1), 1, Jan. 1982.

31. C. Rich, H.E. Shrobe, R.C.K. Waters, G.J. Sussman, and C.E. Hewitt, "Programming Viewed as an Engineering Activity," MIT AI Memo 459, Jan. 1978.

32. C. Rich and H.E. Shrobe, "Initial Report on a LISP Programmer's Apprentice," *IEEE Trans. Software Eng.* **4** (6), Nov. 1978.

33. M. Suite, "Application of Fuzzy Sets to Analogical Reasoning," unpublished manuscript, George Washington University, Apr. 1984.

34. P.H. Winston and B.K.P. Horn, *LISP* (Addison-Wesley, 1981).

35. K. Nakamura, A.P. Sage, and S. Iwai, "An Intelligent Database Interface Using Psychological Similarity Between Data," *IEEE Trans. Systems, Man, and Cybernetics* **SMC-13** (4), 558, July-Aug. 1983.

36. D.G. Shapiro, B.P. McCune, and G.A. Wilson, "Design of an Intelligent Program Editor," Advanced Information and Decision Systems, Report TR 3023-1, Sept. 1982.

37. K.A. Frenkel, "Toward Automating the Software-Development Cycle," *Commun. ACM* **28** (6), 578, June 1985.

38. C. Green and S.J. Westfold, "Knowledge-Based Programming Self-Applied," *Machine Intelligence* (John Wiley and Sons, 1982), Vol. 10, p. 339.

39. C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a Knowledge-Based Software Assistant," Kestrel Institute, Rome Air Development Center RADC-TR-83-195, Aug. 1983.

40. S. Fickas, D. Novick, and R. Reesor, "An Environment for Building Rule-Based Systems: An Overview," Computer and Information Science Department, University of Oregon, Feb. 1985.

41. S. Fickas, D. Laursen, and J. Laursen, "A Knowledge-Based Software Specification Environment," Presented at the Rutgers Workshop on Knowledge-Based Design, July 1984.

42. S. Greenspan, "Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition," Computer Science Department, University of Toronto, Dissertation (1984).

43. Institute for Defense Analyses IDA-D-49, "Report of Findings and Recommendations: Software Engineering Institute Study Panel," Alexandria, Virginia, Dec. 1983.

44. B. Rensberger, " 'Star Wars' Splits Experts into 2 Camps," Washington Post, Mar. 3, 1985, pp. A-1, A-18, A-19.

45. Department of Defense, The Strategic Defense Initiative: Defensive Technologies Study, Washington, D.C., Aug. 1984.

46. Department of Defense, Defense Against Ballistic Missiles: An Assessment of Technologies and Policy Implications, Washington, D.C., Apr. 1984.

47. M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto, "Requirements-Based Testing of Real-Time Systems: Modeling for Testability," *Computer*, Apr. 1985, p. 71.

48. D. Ross, "Interview: Douglas Ross Talks About Structured Analysis," *Computer*, July 1985, p. 80.

49. B. Boehm, "Software Engineering: R&D Trends and Defense Needs," *Research Directions in Software Technology* (MIT Press, 1979).

50. T.E. Bell, D.C. Bixler, and M.E. Dyer, "An Extendable Approach to Computer Aided Software Requirements Engineering," *IEEE Trans. Software Eng., Special Issue on Requirements Analysis* **SE-3**, 49, Jan. 1977.

51. A. Borgida, S. Greenspan, and J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications," *Computer*, Apr. 1985, p. 82.

52. L. Brownston, R. Farrell, E. Kant, and N. Martin, "*Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming* (Addison-Wesley, 1985).

53. J.G. Carbonell, "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," *Machine Learning: An Artificial Intelligence Approach* (Tioga Publishing, 1983).

54. J. McLean, "A Formal Method for the Abstract Specification of Software," *J. Assoc. Computing Machinery* **31**(3), July 1984.

55. R.J.K. Jacob and J.N. Froscher, "Developing a Software Engineering Methodology for Rule-Based Systems," Proceedings of the 1985 Conference on Intelligent Systems and Machines (1985).

END

2-87

DTIC