

ISS 787

COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT
ISSUES(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA
M K FREY JUN 87

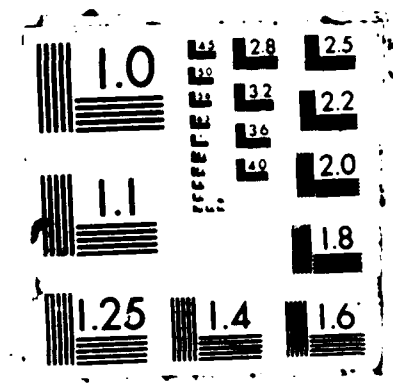
1/1

UNCLASSIFIED

F/G 12/5

NL

END
9 87
DTIC



DTIC FILE COPY

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
AUG 28 1987
S D

AD-A183 787

THESIS

COMPUTER AIDED SOFTWARE ENGINEERING (CASE)
ENVIRONMENT ISSUES

by

Wayne K. Frey

June 1987

Thesis Advisor:

Daniel L. Davis

Approved for public release; distribution is unlimited

87 8 28 024

unclassified

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 is unlimited.	
2b DECLASSIFICATION/DOWNGRADING SCHEDULE		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)		5a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (if applicable) 52	7a ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	
8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (if applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO	PROJECT NO
		TASK NO	WORK UNIT ACCESSION NO
11 TITLE (Include Security Classification) COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT ISSUES			
12 PERSONAL AUTHOR(S) Frey, Wayne K.			
13 TYPE OF REPORT Master's Thesis	13b TIME COVERED FROM TO	14 DATE OF REPORT (Year, Month, Day) 1987 June	15 PAGE COUNT 69
16 SUPPLEMENTARY NOTES			
17 COSA CODES		18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB GROUP	
		Computer Aided Software Engineering (CASE) En- vironment; Software Engineering Environment; Environment Development Principles	
19 ABSTRACT (Continue on reverse if necessary and identify by block number) The rising percentage of system costs attributed to software development and maintenance have resulted in the research by industry and academia into ways to improve the productivity of software professionals in all phases of the software lifecycle. Computer Aided Software Engineering (CASE) environments are one solution being pursued. This thesis attempts to coalesce, from various efforts to date, some general principles for such environments in order to assist decision makers who must procure them. This work is in support of the Missile Software Branch, Naval Weapon Center China Lake, California (MSB), and their investigation of CASE environments to improve productivity. Problems of CASE development and use are discussed in this context. A general problem solving approach through abstraction of resources is proposed with a focus on an individual programmer productivity subset of a CASE environment.			
20 DISTRIBUTION AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS		21 ABSTRACT SECURITY CLASSIFICATION unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. Daniel L. Davis		22b TELEPHONE (Include Area Code) (408) 646-3091	22c OFFICE SYMBOL Code 5204

Approved for public release; distribution is unlimited.

Computer Aided Software Engineering (CASE) Environment Issues

by

Wayne K. Frey
Lieutenant Commander, United States Navy
B.S.(Business Administration), University of Minnesota, 1974

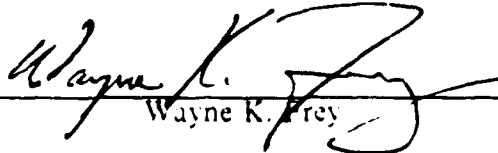
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

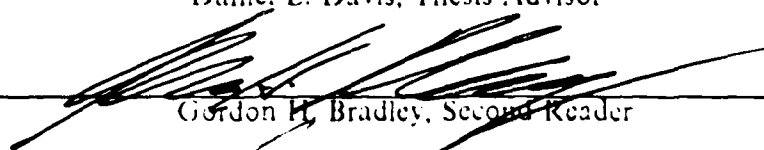
NAVAL POSTGRADUATE SCHOOL
June 1987

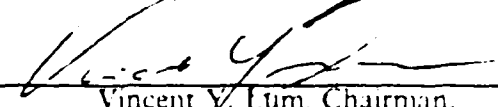
Author:

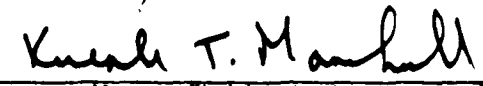

Wayne K. Frey

Approved by:


Daniel L. Davis, Thesis Advisor

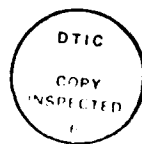

Gordon H. Bradley, Second Reader


Vincent Y. Lum, Chairman,
Department of Computer Science


Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

The rising percentage of system costs attributed to software development and maintenance have resulted in the research by industry and academia into ways to improve the productivity of software professionals in all phases of the software life-cycle. Computer Aided Software Engineering (CASE) environments are one solution being pursued. This thesis attempts to coalesce, from various efforts to date, some general principles for such environments in order to assist decision makers who must procure them. This work is in support of the Missile Software Branch, Naval Weapon Center, China Lake, California (MSB), and their investigation of CASE environments to improve productivity. Problems of CASE development and use are discussed in this context. A general problem solving approach through abstraction of resources is proposed with a focus on an individual programmer productivity subset of a CASE environment. (Theses).



Accession For		
NTIS	CRA&I	<input checked="" type="checkbox"/>
DTIC	TAB	<input type="checkbox"/>
Unannounced		<input type="checkbox"/>
Justification		
By		
Distribution/		
Availability notes		
List		
Special		
A-1		

THESIS DISCLAIMER

The following trademarks are used throughout this thesis:

- Ada® is a Registered Trademark of the U.S. Government (Ada Joint Program Office)
- Apple® is a Registered Trademark of Apple Computer Incorporated
- GEM® is a Registered Trademark of Digital Research
- IBM® is a Registered Trademark of International Business Machines Corporation
- Macintosh is a Trademark of Apple Computer Incorporated
- Macintosh II is a Trademark of Apple Computer Incorporated
- microVAX II is a Trademark of Digital Equipment Corporation
- UNIX® is a Registered Trademark of AT&T Bell Laboratories
- VAX is a Trademark of Digital Equipment Corporation

TABLE OF CONTENTS

I.	INTRODUCTION	8
II.	BACKGROUND OF SOFTWARE ENGINEERING AND ENVIRONMENTS	10
A.	THE SOFTWARE ENGINEERING PROCESS	10
B.	THE SOFTWARE ENGINEERING PROBLEM	13
	1. Quality	13
	2. Quantity	15
	3. Maintenance	17
	4. Management	17
C.	DEVELOPMENT ENVIRONMENT A SOLUTION	18
	1. Structured Methodology	18
	2. Automation	19
	3. The Environment Jungle	20
III.	CASE DEVELOPMENT ISSUES FOR MISSILE SOFTWARE BRANCH (MSB), CHINA LAKE	29
A.	MSB BACKGROUND	29
	1. Mission	29
	2. Problem	30
	3. Organization	30
	4. Current Environment	31
	5. CASE a Desired Solution	32
B.	CASE ENVIRONMENT PROCUREMENT ISSUES	33
	1. Short Term Off-the-shelf Buy Approach	33
	2. Long Term Off-the-shelf Buy Approach	35
	3. Make Approach	36
C.	WHICH WAY FROM HERE	36

IV.	CASE ENVIRONMENT DEVELOPMENT ISSUES	37
A.	SCOPE OF CASE PROBLEMS	37
1.	Evolutionary Development Politically Necessary	37
2.	Requirement Tradeoffs Contributing to Risk	38
B.	FUNDAMENTAL PRINCIPLES FOR CASE ENVIRONMENTS	38
1.	Portable Reusable CASE Resources	39
2.	Integrated CASE Resources	40
3.	Open Environment	41
4.	User Friendly	41
C.	FUNCTIONAL ABSTRACTION AN APPROACH TO SOLVING PROBLEMS	42
1.	Definition of Abstraction	42
2.	Formal Specification	42
3.	Abstraction of Physical Resources	43
4.	Abstraction of Environment Resources	43
5.	Layers	44
6.	Standards Enforcement vs. Encouragement	45
7.	Top Down or Bottom Up	47
V.	FUNCTIONAL REQUIREMENTS ANALYSIS ISSUES	49
A.	SCOPE OF THIS EFFORT	49
B.	INDIVIDUAL PROGRAMMER PRODUCTIVITY (IPP) RESOURCES	50
1.	Physical Resources	50
2.	CASE Resources	53
C.	WHAT ABOUT THE REAL WORLD	57
VI.	CONCLUSIONS	58
A.	INVITATION FOR REVOLUTION	58
B.	FUTURE WORK	59
C.	RECOMMENDATIONS FOR MSB	60
1.	Near Term	60
2.	The Future	63
	LIST OF REFERENCES	65
	INITIAL DISTRIBUTION LIST	68

LIST OF FIGURES

2.1	The Waterfall Model of the Software Life Cycle	11
2.2	The ISTAR Integrated Project Support Environment (IPSE).	24
2.3	The Ada Programming Support Environment (APSE)	26

I. INTRODUCTION

Since its infancy, the software industry has worked to improve the environment in which people work to create software. In general, these efforts were paced by hardware developments and by the way programmers thought about programming. The development of assembly and then higher level programming languages was an environmental improvement (over machine language) because they allowed programmers to think in more abstract, logical terms about the problems their programs were solving. System operators and operating systems relieved the programmer from the burden of managing hardware resources. The move from offline batch interaction to online real time interaction was another major improvement in the environment of programmers. As more and more software resources to improve the programmers environment have been introduced, the hardware designers have provided the speed and computing power necessary to support all of these features, and real work, without bringing systems to their knees.

The hardware advances resulting from VLSI and other technologies have allowed the proliferation of low cost computers throughout modern society, resulting in an explosion in the demand for software. The drastic improvements already being made in software engineering methods have not kept up with this demand.

For the past decade and more, the software industry has expended much effort on the issues of software engineering as a methodology analogous to other engineering fields, and to the development of automated tools and environments to support this methodology and enhance the productivity of software developers and maintainers.

This thesis attempts to coalesce, from various software development environment efforts to date, some general principles for such environments to aid the decision makers who must procure them. We begin by discussing the *software engineering process*, the *software engineering problem* and the issue of environments. We then consider a particular research and development software group, Missile Software Branch, Naval Weapon Center, China Lake, Ca. (MSB), their mission, their need for a Computer Aided Software Engineering Environment (CASE), and some of the issues they face in procuring a CASE.

The concept of an integrated CASE has developed to include the *cradle to grave* life cycle of software. We discuss the general state of technology of software development tools and environments to date and some of their problems. We discuss abstraction of environment resources and standardization of interfaces as potential solutions to problems. To limit the scope of this effort we focus on one of the better developed and understood subsets of a CASE environment, the aspect of individual programmer productivity (IPP), in terms of abstract resources applicable to any CASE. Future areas of study are suggested. Recommendations, for Missile Software Branch procurement efforts, are discussed in terms of general CASE principles in the IPP context.

II. BACKGROUND OF SOFTWARE ENGINEERING AND ENVIRONMENTS

A. THE SOFTWARE ENGINEERING PROCESS

Software engineering has been defined in many ways. Boehm (1981, p. 16) called it "the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation." The focus of such definitions is that software engineering is engineering in the same sense as the traditional engineering fields.

It should be clear that we are not talking about one person programming for his sole individual use. We are talking about the case where more than one person is involved in developing and/or using the software products. In general terms there are at a minimum: a customer (an individual or organization(s) who want something useful done by a computer), a developer (an individual or organization(s) who must engineer the software to meet the customer's need), a user (who may or may not be the same as the customer) and a maintainer (who may or may not be the same as the developer).

An applicable definition of *process* when referring to the *software engineering process* is: "... a series of actions or operations conducing to an end: esp. a continuous operation or treatment esp. in manufacture ..." (*Webster's*, 1966, p. 678). We take the "end" in the software engineering process to be an operational version of a software product including the object code and all attendant documentation (both historical and deliverable) required to recreate it.

The common waterfall model of the software life cycle, Figure 2.1 (Boehm, 1981, p. 36), with minor variations, is often used to capture the major (top level) "series of actions or operations" in the software engineering process. This traditional view appears in the literature as far back as a 1956 paper written by Herbert D. Bennington describing work on the SAGE air defense system software (Bennington, 1956, p. 356).

The IEEE Ninth International Conference on Software Engineering, (30 March - 2 April 1987, Monterey, California, USA) met with the theme of "Formalizing and Automating the Software Process." During the opening Plenary Session, Program Committee Co-chairman Robert Balzer stated that the traditional waterfall model of the software engineering life cycle "is dead". Our purpose here is not to debate *that* issue. However, this work is based on a belief that the waterfall model provides a well

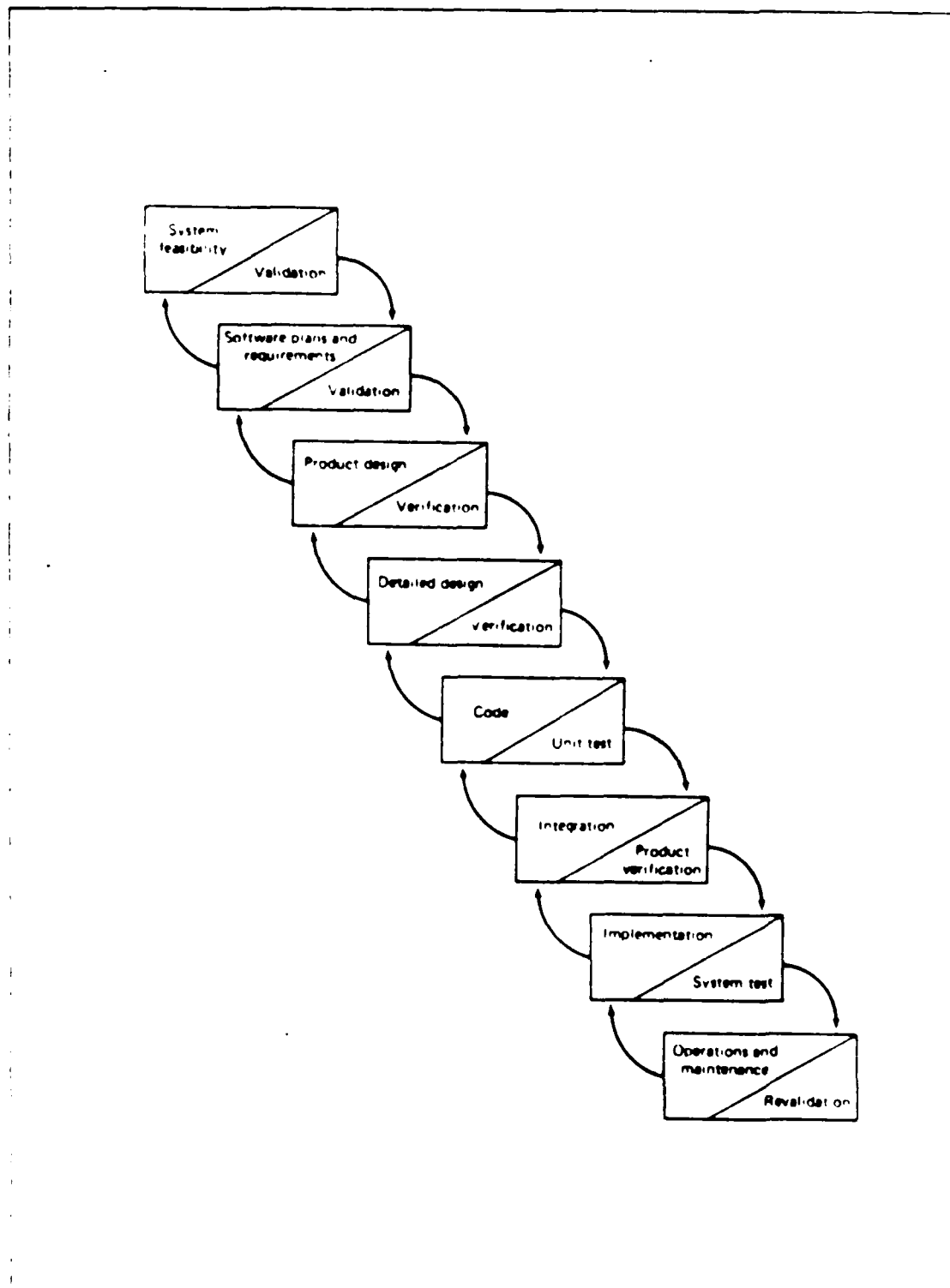


Figure 2-1 The Waterfall Model of the Software Life Cycle.

known terminology and common framework around which much of software engineering to date can be discussed. We believe that the waterfall model represents a top level view of the software engineering process when the process is viewed statically. Such a static view is often attempted, and may be appropriate, in dealing with systems where the problem and solution methods are well known and defined.

When faced with uncertainty in the definition of the problem or methods of solution, understanding evolves throughout the software engineering process and a static view of the whole life cycle is inappropriate. We believe that, in such a case, a waterfall model is still useful, but not as the top level of the process. Instead, major portions of it exist at a lower level of a process which may be categorized as evolutionary prototyping. In other words, a waterfall model applies at many levels of the overall software engineering life cycle process. In our view of such a dynamic process, a waterfall-like sequence of transformations may be executed repeatedly to "conduce" progressively more functional versions of the "end" product. Since the waterfall model imposes no temporal constraints on its phases, an initial version may be *prototyped* rapidly by manipulating not only the functionality of the prototype version, but the complexity and detail of some phases of that version's life cycle. A result can be top down design, with a combination of top down and bottom up implementation, of a family of evolutionary versions. Each new version is evolved by extension of the collective analysis, design, implementation and testing, etc., of some prior version. This approach can deal dynamically with problem solution uncertainty early in a project development life cycle, as well as with continued evolution of the project over time and in the context of technological advance. Lehman (1985) discusses, in detail, the dynamics of software evolution with respect to his now familiar S, P and E program classes. He also discusses the process of iterative transformation through waterfall like phases from topmost (i.e., requirements) specification to final implementation (in this case a prototype or subsequent versions). Lehman's iterative transformation is based on a single canonical *design step* whereby the software engineer creatively chooses a formal lower level linguistic system in which to *model* the higher level model with which he begins each step. Formalism is intended to support a mapping from the higher level model to the subsequent lower level model facilitating calculable (vs. empirical) verification, backtracking and change propagation activities which lead to iteration of the *design step* and support evolution. Such formalism is not in common practical use in industry today. Such a process is intended to help avoid

throwing out large portions of prior development work and having to start over from scratch.

Speaking at the IEEE Ninth International Conference on Software Engineering, Herbert Bennington commented that the successful SAGE development efforts were not "driven" by the waterfall-like model illustrated in his 1956 paper. But, that a prototyping process, based on these activities, was employed to deal dynamically with the uncertainty involved in the projects ambitions and possible solutions. So, these ideas are far from new. They have been studied and gained prominence in the context of past successes and failures. Given these views of the software engineering process, we can discuss briefly our view of the software engineering problem.

B. THE SOFTWARE ENGINEERING PROBLEM

The *software engineering problem* or *crisis* to use a popular cliché, has manifested itself in problems including quantity, quality, maintenance and management.

1. Quality

Software quality is a fundamental issue. Many products simply don't do what the user wants. The causes for this generally reduce to the inherent difficulties with validation, verification and testing.

a. Validation

Validation is the process of determining the fitness of a software product for its operational mission. The developer interacts with the customer, at the start of the software engineering process, to translate the customer's need into a requirements specification. Validation at this stage tries to determine that the right product is being engineered. The customer's description of what the product must do, is inherently imprecise because it is expressed in a semantically imprecise natural language (e.g., English). Within his own organization, the developer translates the requirements description into a requirements specification. This has traditionally been the first step away from natural language toward a more precise representation. Validation of this translation is often complicated because the customer thinks in his natural language, not the developers representation scheme. Next, the developer translates the high level requirements specification into design specifications. Design specifies how the requirements will be met. The high level requirements specification of what does not inherently contain all of the detail required to make explicit design decisions. Ideally, as questions arise, conscious effort would be made to revalidate the requirements specification with the customer so that the requirements specification matches the

questions. Often this does not happen. Sometimes it is not even clear to the designer that his decision cannot be validated from the requirements. Additionally, design representation is generally so far removed from the customer's view and understanding that an inherently imprecise reverse translation is required in order to get any feedback from the customer at this point. After implementation, validation must determine if the right product was actually built. As we'll discuss later, this last phase, testing, is an inherently imprecise process.

b. Verification

Verification assumes that the requirements specification is valid and tries to ensure that the product is built correctly. The developer must translate design specifications into an implementation in object code for the target machine. Since design specifications are a more precise representation, this translation is much more direct. In fact, systems have been implemented in which specifications are automatically translated from design languages to source code languages which are then translated into object code by compilers. However, these programs themselves can not as yet be proven correct, so empirical verification is essential to insure the intent of the design is met by the object code. For two decades, considerable effort has been devoted to proving the correctness of programs (verification). However, testing continues to be the best available tool for verification and validation.

c. Testing

It is generally accepted that exhaustive testing (instrumented execution of a program, in its precise operational environment, over every possible combination of inputs) is not feasible for other than trivial programs. It is also accepted that nothing short of exhaustive testing will infer program correctness. As Dijkstra said, "program testing can be used to show the presence of bugs, but never their absence". (Dahl, 1972, p. 6) So, the primary objective of testing becomes demonstration of the program's operational readiness. Individual tests must be mapped from the design specifications for verification and from requirements specifications for validation. Since exhaustive testing cannot realistically be performed, a reasonable subset of all possible tests must be chosen. With knowledge of the design and code, (by not simply treating modules and programs as "black boxes") tests can be chosen for boundary conditions, legal and illegal inputs, volume, etc. and logical assumptions can be made about continuity of function between boundary conditions. Even if the program executes the tests without errors, and the logic is unflawed, operational readiness is only shown in

the specific environment tested. Was some unforeseen combination of inputs omitted? Is the target machine and its firmware and system software identical to the test machine?

2. Quantity

Proliferation of computers throughout modern society has caused an explosion in demand for software. This demand is a double edged sword. The more software there is, the more software there is to be maintained. A fixed number of software engineers, with fixed productivity, will eventually reach a point where all of their effort is consumed by maintenance. No new software can be developed until some software in maintenance is retired.

While the number of software engineers is not fixed, several industry studies conclude that the number is increasing too slowly to keep pace with increasing software demands. To complicate matters, the lifespan of existing software often exceeds expectations. This is particularly true in the United States Department of Defense (DoD) where capital investment in militarized hardware, logistics systems, training of technicians and operators, etc. all add up to practical and political inertia to keep a working system in place (and in maintenance) long after technology passes it by. Improving the productivity of software engineers appears not only desirable, but essential, to stem the quantity problem.

a. Reuse

The reuse issue is actually a component of the overall issue of improving productivity of software engineers. It is mentioned separately here because it has long been thought to be the key to making an order of magnitude improvement in software production capability. Since the earliest days software engineers have redesigned and reimplemented things which had been built before. The problems of reuse are well known and go far beyond any "not invented here" egomania. Reusable code libraries achieved some early success for discrete functions (like mathematical formulas). It was hoped that higher order languages would make source code reuse a reality for much more complicated functions and programs. However, a general lack of discipline in establishing and adhering to language standards resulted in proliferation of subsets, supersets and generally inconsistent implementations of compilers for varying machines often defeating portability of such code. Additionally, at the larger scope of more complex functions and procedures, the code is too detailed (i.e., data types, data structures, etc.) for easy reuse. As a result, it is generally the abstraction (e.g., a

domain specific model), and not the concrete implementation, that is reutilized. (Standish, 1984, p. 495) In general, even the reuse of the abstraction has been informal at best. Documentation of requirements and design specifications generally lacks standards outside a particular organization (and sometimes within). The *why* of particular design and implementation choices is often unclear in documentation. The level of effort required to understand what an existing design is doing and what, if anything, must be done to adapt it to a new application or new environment, is often seen as more difficult than starting fresh. So, reuse of the abstraction, without methods and tools to reduce the understanding overhead, is usually informal. Individuals use their own prior work (things they understand and retain in their personal toolbox), but they reject organized library resources as too hard to use. This situation is changing as such methods and tools supporting reuse become more available, but formal reuse is still absent in many organizations.

b. Productivity

In classical terms productivity can be defined as units of product delivered divided by cost. Herein lies one of many problems associated with measuring the productivity of software developers. There are no basic units of software. However, various measures have been developed and attempts to instrument and study productivity have been made. In general, we believe productivity in software engineering activity has been worse than it is now. We believe that various efforts to improve the software engineering methodology and environment have improved productivity to its current level. And, we believe further improvements are possible. We don't believe precise measurements of productivity are possible.

Such measurements industry wide are complicated by the lack of meaningful measurement standards and the proprietary nature of statistics. Something as seemingly simple as lines of code per programmer per day cannot be compared unless one defines precisely what a line of code is. Is it assembly code or a fourth generation language? Is there more than one statement per line? Beyond this is the issue of program complexity. Highly modularized code is likely to have more lines than unstructured monolithic unmodularized code. Yet a poor design can yield just as many or more lines of modularized code as a good design. Which represents more productivity?

The most believable claims for measuring software engineering productivity, and productivity increases, come from studies within individual organizations. At least

they measure activity in a relatively consistant environment (source system hardware and software, source language, methodology, etc.), with consistant measurement units (what is a line of code?) and with a relatively consistant group of individuals over the course of the study. With such a semblance of a controlled environment, the impact of introducing new tools or methods becomes more measurable. One consistant source of such reports, for a number of years, has been Boehm at TRW. (Boehm, 1981 and 1983) One can argue that the result of such studies can be generalized for other organizations. One cannot argue that such a generalization offers any precision. But, it is evidence to offset the risk in a decision to invest in similar tools and methods for productivity improvement.

3. Maintenance

Software maintenance is commonly defined as any work on a software system after operational release. It is often subdivided into maintenance to correct errors (corrective maintenance) or maintenance to improve or modify capability (sometimes called perfective maintenance). In either case, maintenance involves changing the program. Without a complete understanding of how the program works and why the designers chose to make it work that way, a maintainer can often introduce totally unexpected errors. Such a change can invalidate all prior testing. Maintainers are often not the original developers, and they must rely on the documentation of the development process for the understanding required to change a software system. They need to be able to repeat testing and compare results to original tests in order to determine the operational readiness of the new (maintained) version. They must conduct and document new tests to demonstrate readiness of new capabilities. Much of development and maintenance documentation for existing software has been inadequate to support efficient maintenance efforts. Often, much of the development effort must be repeated to do maintenance well. In reality, driven by pressure to meet operational deadlines, much maintenance results from efforts closer to trial and error. Needless to say, documentation of such efforts, if any, is seldom beneficial to follow-on maintenance.

4. Management

Management problems have been a major driving factor towards software engineering methodologies and tools. Problems such as: late delivery, over budget, unreliable product, failure of product to meet specifications and product difficult and expensive to maintain are common. They are attributable in part to the quality,

quantity and maintenance issues already discussed. The familiar phrase, "You can't manage what you can't measure.", sums up part of management's woes. Another major contributor to management problems is the chaos inflicted on static plans when the well defined problem or solution unexpectedly evolves into something else. Such attempts to manage, based on measurement of the *wrong* things, are further complicated by the phase in the life cycle when the change is discovered. Changes or errors involving the requirements and design, which are not discovered until late in the development process, are often much more expensive to correct. They can often result in discarding much of the work already done.

In the last decade software engineering methodologies, tools and environments have exploded on the market offering and delivering partial solutions to the software problem. Work and controversy surrounding development environments continues.

C. DEVELOPMENT ENVIRONMENT A SOLUTION

A software development environment is, in general terms, the domain in which the software system is developed. From the view of software engineers this domain consists of methods, tools (computer hardware and software) and other software engineers (the managers, analysts, designers, programmers, etc. who make up the engineering team). In other words, all of the *resources* necessary to engineer software.

1. Structured Methodology

Since the early 1970's, structured methods for managing and developing software have been written about, taught and implemented. The structured methods support the major activities of the waterfall model (Figure 2.1).

By structured methods we mean a collection of procedures and concepts to increase the productivity and effectiveness of the software engineering organization. Elements of the structured methods include:

- structured analysis, guidelines and graphical tools that allow replacing the traditional representations of the requirements specification with one that can be more easily understood by the customer;
- top-down design and implementation;
- structured design, guidelines and methods to help the designer distinguish between good and bad designs;
- structured programming, composition of program logic from sequence, if-then-else and do-while constructs with little or no use of the go-to.

Associated with these methods are aids to implementation such as:

- program librarians, to relieve programmers of clerical tasks and manage version control and archival;
- structured walkthroughs, peer group review of design and implementations to assist in error reduction and schedule pacing between formal inspections

(Yourdon, 1986, pp. 2-3).

Controversy about the value of these and other methods often centers around how much they improve productivity and effectiveness. As indicated earlier, *how much* is a difficult thing to measure and compare with any precision. Yourdon says "In general . . . they double the productivity of the average programmer, increase the reliability of his code by an order of magnitude, and decrease the difficulty of maintenance by a factor of two to ten." (Yourdon, 1986, p. 3) We'll just say that common sense indicates these methods should improve productivity and effectiveness, and our general sense of reports from industry, regarding such methods, is that they do work with substantial benefit.

One of the serious problems encountered trying to use these methods is that a tremendous amount of cross referencing of data and data structures from one phase of the life cycle to another is required. Also, many tasks are cyclic in nature and require a lot of repetitive activity. For instance, validation of a data flow diagram representing a requirement specification might require reiterating the diagram several times with minor changes as the customer and developer narrow down exactly what the customer wants. Each named piece of data on the diagram is a unique entity recorded in a data dictionary. Each new change to the diagram must be checked against the data dictionary to ensure all items are uniquely recorded. Such repetitive or purely mechanical tasks tend to be error prone and slow when done by humans. They are excellent candidates for automation using a computer. (MacLennan, 1983, p. 5)

2. Automation

There are generally three forms of automation supporting software engineering.

a. Tools

Tools are programs that perform a single type of function. A compiler, that generates object code for a target machine from source code in a specific language, is a tool, as are assemblers, linkers, editors, graphic tool boxes, spread sheet programs, etc.

b. Programming Support Environments

Programming support environments are collections of tools to provide support for programming (normally considered the implementation phase of the life cycle). They generally only directly support programmers. They may be a cooperative, interoperable set of tools (what we will call a toolset) specifically designed to work together with a common user interface and common data exchange formats. Or, they may be a set of disjoint tools which are separately executed, each with its own user interface, each performing its task on its own internal data structures, generally with a sequential file of characters as the only external data interface with each other.

c. Computer Aided Software Engineering (CASE) Environments

CASE environments are a relatively new concept. They are an extension of programming support environments to the entire software engineering life cycle. They are intended to provide support to the entire engineering team (i.e., managers, analysts, designers, programmers, maintainers, etc.) for overall product development.

3. The Environment Jungle

We have been automating aspects of the environments in which we engineer software for a long time. At first there were simply collections of whatever software tools were available for the hardware and languages we wanted to use. In general, partly due to the large number of languages being developed, only the most basic tools were available (assemblers, linkers, loaders, compilers) to support production of object code. These environments were based in batch processing techniques. As hardware advances produced teletype terminals for on-line real-time processing, environments gave the illusion (in the user interface) of being interactive with the computer. This was still sequential batch processing (for that user), only the batches were much smaller and turnaround time much faster. Video terminals evolved directly from teletype terminals, still processing lines of characters. The natural data structure to evolve for external interfaces in such environments were files of sequential characters. These are still the most common "standard" data exchange format industry wide. Since there are more than one "standard" character code (e.g., ASCII and EBCDIC), filter programs are employed for portability of files.

As hardware provided inexpensive speed and raw computing power, assisted by operating systems offering virtual memory support, a few languages began commanding a large market share. As software engineers came to grips with the software problem, more complex interoperable toolsets appeared. These interoperable

tools often rely on common data structures (other than simple sequential character files) representing *objects* which can be viewed and manipulated by various functions within each tool. These objects are normally stored in a database accessible by all of the interoperable tools. The database objects are generally only meaningful in the context of their tool or tool set which often must eventually produce a sequential character file for manipulation by tools not integrated with the set. The advent of bit mapped graphic objects has added further complexity to portability of data among tools. Due to storage overhead, and the complexity of *handling* bit images instead of the objects they represent, bit mapped graphic objects are generally compacted into unique, complex, proprietary storage code which constitutes a *recording* of the sequence of resource (tool) calls used to construct the object. These recordings are replayed and edited in order to reconstruct or manipulate the objects. The storage format of such objects is therefore meaningless outside the context of the environment required to replay it.

a. Integrated vs. Disjoint Environments

We use the term *integrated* to describe environments with the following features:

- all resources conform to a consistent user interface;
- all resources are as highly interoperable as possible;
- objects and their interrelationships are in a persistent common data format which is meaningful to all environment resources;

We use the term *disjoint* to describe environments which lack integration:

- inconsistent user interface among resources requiring user to shift modes when moving from one resource to another.
- incompatible data formats among resources

b. Environment Development Efforts

The software *crisis* and technological advances (hardware, operating systems, languages, user interfaces, databases, etc.) have resulted in a booming new market in environments. We easily collected a full file drawer of documentation in the form of books, papers, technical reviews, promotional materials, and conference proceedings describing myriad *environments* under research, or in production or operation. What is generally most common about these environments is that *they have so very little in common.*

(1) *General State of Technology.* Developing a CASE environment is itself a software engineering problem of mammoth proportions. No *standard* requirements

for a CASE environment have been adopted. Since the software engineering process itself is less than mature or stable, top down specification and design of an environment to model it has been deficient. For the most part a bottom up approach has prevailed. While many CASE labels have been hung on projects, at best it is limited integrated toolsets that are being made. The CASE customer who can define his particular software engineering process is unlikely to find a toolset which is a complete CASE environment for his process. Since data portability between independent tools and toolsets is generally limited to sequential character files, assembling a complete CASE environment from off-the shelf products can at best yield a disjoint environment. The majority of what are being called CASE environments today include:

- graphic tools supporting various structured analysis and design methods;
- program design language (PDL) tools supporting prototyping through *executable* specifications;
- programming support environments supporting specific language implementations, debugging, documentation, version control, and archival;
- project management systems supporting a variety of management methodologies and economic models;
- office automation toolsets;
- hardware and software supporting multiple view window interfaces and multitasking.

A prevailing point of view seems to be that it is unlikely that any single organization could, or should, define canonical requirements for some CASE environment and then implement all of the integrated resources to instantiate it. ¹

¹ This may not do justice to a few large software developers who have invested in long term top down development of environments for their own use based on their own software engineering process and methods of operation. However these systems are either generally not available off the shelf, or represent an exorbitant investment, and complete integration within them is dubious. A possible exception, the R1000 Ada Development System from Rational (Mountain View, California), has been developed for the market place and is touted in the literature to epitomize "... the fully integrated CASE environment." (Suydam, 1987, p. 58) However, most would classify the R1000 dedicated hardware architecture and software as an exorbitant investment. Also, we should note that European CASE environment efforts seem more advanced than our own domestic endeavors. ISTAR, from Imperial Software Technology (London, England), is an example. ISTAR's top down design provides for a flexible, open and extensible environment.

It is generally agreed that integration of tools/toolsets (resources) is desirable within an environment for:

- coherence, whereby all the tools behave in a uniform and consistent way (e.g., a common user interface style);
- control, whereby tools behave in a disciplined way (e.g., not allowing unintegrated tools to bypass and subvert a configuration management tool);
- sharing, whereby tools work together by sharing data (data is structured independently of the tools which create and use it)

(Hall, 1987, p. 289). There is a basic conflict between the desire for integration and the desire and need (economic and evolutionary) for environments to accept tools from various sources. We feel the most promising of the current approaches to resolving this conflict is to build around a kernel structure of resources which provide services to the tools for accessing and manipulating objects in a standardized environment database. Once the interfaces to these kernel resources are defined, tool developers who adhere to the interfaces will develop integratable tools. Two such efforts currently underway are the Portable Common Tool Environment (PCTE) which is the base for a number of European environment and tool projects (including ISTAR, see Figure 2.2 (Henderson, 1987, p. 59)) and the Common Apse(Ada Programming Support Environment) Interface Set (CAIS) sponsored by the DoD.

(2) *United States Department of Defense (DoD) Initiatives.* Early in the DoD common high-order language project which spawned Ada, developers recognized that the language alone would be insufficient to combat the problems associated with DoD software projects. DoD sponsored development of requirements, defining the Ada Programming Support Environment (APSE), with the stated objective "to support the development and maintenance of Ada applications software throughout its life cycle, with particular emphasis on software for embedded computer applications." (Stoneman, 1980, p. 1) Fundamental concepts of the APSE included:

- host target environment, where the APSE is hosted on a development machine while the target machine of the software, to be developed utilizing the APSE, may be a different machine;²
- program database, to include all project information (e.g., source and object code, documentation, specifications, etc.);
- extensibility, with all tools written in Ada.

²In embedded systems the target hardware may be so limited in resources (speed, memory, etc.) that it cannot practically support the development environment.

ISTAR FRAMEWORK

ISTAR TOOLSET

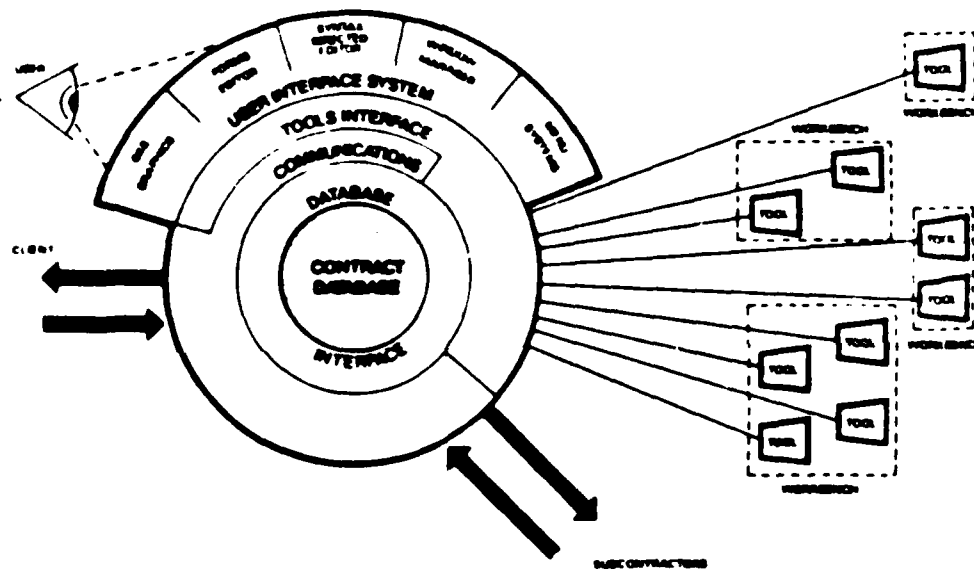


Figure 2.2 The ISTAR Integrated Project Support Environment (IPSE)

The original *Stoneman* representation of the APSE is illustrated in Figure 2.3 (Booch, 1987, p. 409). The host machine resources are provided through the Kernel APSE (KAPSE), which provides the logical to physical mapping. The Minimal APSE (MAPSE) contains some minimal toolset for program development, and the APSE overall represents a *life cycle* environment.

While early *Stoneman* developers may have thought in terms of a single organization developing an APSE or MAPSE under DoD sponsorship, this approach quickly ran into the integration versus independent developers conflict mentioned earlier.³ Commercial developers are competitively pushing the edge of technology in programming support and CASE environment resources. Any standardized integrated toolset from a single developer faces stiff competition in fields (e.g., editors, debuggers, user interfaces, etc.) where few widely accepted standards exist and several potential defacto standards might emerge. To encourage the competitive advance of environment technology in a direction supporting integrated environments, DoD sponsored the development of the Common APSE Interface Set (CAIS).

The CAIS provides interfaces for data storage and retrieval, data transmission to and from external devices, and activation of programs and control of their execution. In order to achieve uniformity in the interfaces, a single model is used to consistently describe general data storage, devices and executing programs ... referred to as the node model. (*Military Standard Common APSE Interface Set (CAIS)*, 1985, p. 11)

The development of CAIS has been a lengthy and methodical process of bounded scope. The version of the specification scheduled for release in the Spring of 1987 is intended to support some transportability interfaces often required by common software development tools, including:

- the node model;
- processes, covering program invocation and control;
- input/output, covering file and device I/O and interprocess communication;
- utilities, list operations for manipulation of parameters and attribute values.

Some CAIS issues and decisions deferred for later versions of the CAIS include:

³The Army Navy Ada Language System (ALS) was such a venture from which the Army has now withdrawn support. The Navy has continued with ALS-N with the specific purpose of directly supporting some major unique Navy embedded architectures. It has been anticipated that such support will not be spontaneous from the private sector due to the extremely limited vertical market.

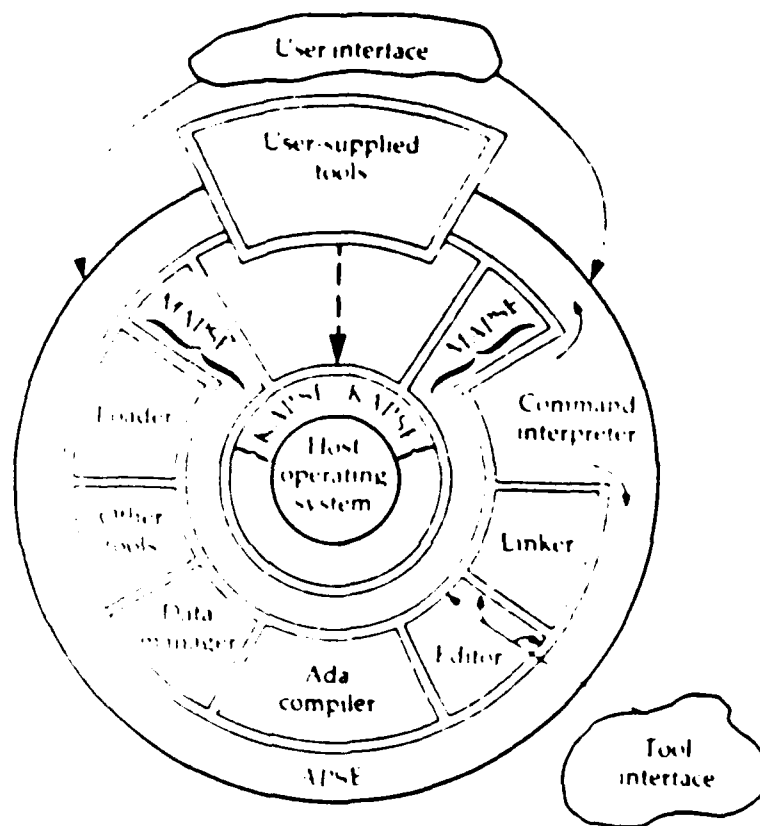


Figure 2.3 The Ada Programming Support Environment (APSE).

- configuration management. CAIS supports resources for configuration management but no specific methodology;
- devices, supports scroll, page and form terminals and magnetic tape drives. (other devices and possibly other ANSI or ISO interfaces (e.g., ISO DIS 7942 Graphical Kernel System (GKS)) are under consideration);
- inter-tool interfaces, are not defined;
- interoperability, only a primitive text-oriented file transfer capability is provided between a CAIS implementation and its host. CAIS does not define external data formats for transfer between environments or between a host and target;
- archiving, a decision on the form that archiving interfaces should take has been deferred

(*Military Standard Common APSE Interface Set (CAIS)*, 1985, pp. 1-2).

The Software Technology for Adaptable Reliable Systems (STARS) program established by the DoD in late 1983 included the STARS Software Engineering Environment (STARS-SEE) task. The early objectives of STARS-SEE were to specify the requirements for a complete life cycle environment which was fully integrated and interoperable, multilingual, utilized state of the art technology and was itself designed to evolve with technology. Early STARS leadership felt that the DoD itself was best capable of analysis and definition of requirements for such an environment. A joint services team composed of uniformed and DoD civilian software professionals, augmented by DoD contractors, analyzed the software engineering process, requirements for the STARS-SEE, and the state of technology. This resulted in generation of a five volume collection of thirty-five preliminary reports, by 1986, in preparation for defining the STARS-SEE software architecture. (Naval Air Development Center, 1986, p. flyleaf) Changes in project management resulted, essentially, in disbanding the STARS-SEE task effort within DoD activities and shifting emphasis to encouragement and support of private sector software engineering environment development efforts.

In addition to such high level DoD environment initiatives as APSE CAIS and STARS, several lower level efforts exist. DoD field activities engaged in software engineering already have an investment in their own unique individual environments which have evolved bottom up (with the evolution of hardware and software technology) as they have done their jobs over the years. They are in various stages of evolution from batch oriented and "interactive" time shared central mainframe mini complexes, to networked "personal" microcomputers with "terminal" access to central computing resources. Individual initiatives to upgrade local

environments with relatively inexpensive "personal" computers and off-the-shelf software engineering tools in such a bottom up fashion are often seen as both blessing and curse. Blessing for their contribution to improving an often otherwise extremely unproductive working environment, and curse because of the lack of interoperability, transportability, consistency, etc. which they represent. ⁴ DoD Ada implementation policy (essentially that Ada is the only authorized programming language for new embedded systems and existing systems entering major revision) has been one point of focus for many of these independent efforts in DoD as well as for the tool and environment developers.

⁴Not all of these efforts are so limited. Some are large and well organized and funded (e.g., the Interactive Ada Workstation being developed under contract by General Electric for the Avionics Lab, AFWAL AAAF-2, WPAFB, OH., and the Software Life Cycle Support Environment (SLCSE) being developed by General Research Corporation under sponsorship of the U.S. Air Force Rome Air Development Center, GAFB, N.Y.).

III. CASE DEVELOPMENT ISSUES FOR MISSILE SOFTWARE BRANCH (MSB), CHINA LAKE

A. MSB BACKGROUND

MSB is a small software research and development group. They are a branch of the Weapons Development Division, Michelson Laboratory, Naval Weapons Center, China Lake, California. Software engineers in the group are domain specialists in onboard, embedded missile software. Prior to current efforts to use Ada, virtually all of their work involved assembly language programs for unique processors with extremely limited resources (e.g., speed, memory) in onboard, mission critical, real-time, embedded missile systems. Working around constraints like limited memory often requires methods (e.g., unstructured design) which subsequently make the software extremely difficult to understand and maintain. Reuse of such hardware specific and unstructured software is virtually impossible. Knowledge of the weapon domain (a major factor itself) is often the only reusable resource in this software engineering process. Hardware advances with the potential to improve the resource (speed, memory, etc.) availability of potential target processors, and the increasing availability of Ada compilers for target processors, have opened the door for MSB to exploit Ada's inherent support for structured methods, object oriented software engineering and code portability.⁵

1. Mission

The basic mission of MSB is to establish and maintain a Navy in-house capability for developing state of the art missile software. As with any research oriented organization, they are considered a resource for exploring new technologies which would likely remain unexplored in the profit oriented private sector. As a development resource they may be tasked to perform some or all of the development of software for specific missile projects.

⁵Onboard embedded software operates in a unique environment. Size, weight, power, and heat dissipation continue to be a major concern, and even today the *memory-is-cheap* scenario may not apply. Efficiency of object code generated by target machine Ada compilers is also of major concern. (Myers, 1987, pp. 71-72)

2. Problem

Hardware advances (e.g., VLSI) have proliferated embedded processors into weapons systems projects at an ever increasing rate. The impact of new technologies on the operational environment of the weapons (e.g., operational deception, electronic warfare, etc.) demands increasing capabilities for mission fulfillment. New technologies make new mission capabilities possible and or essential. As a result, mission critical, embedded system software demand (both upgrade of existing systems and new systems development) is increasing rapidly.

Current federal policy on personnel funding effectively limits any increase in MSB personnel resources in the foreseeable future. Projects rejected (due to insufficient capacity) by MSB must either be done somewhere else (generally private sector contracts) or be abandoned or postponed. For many projects, especially research, the missile software domain expertise of MSB makes them the best equipped for the job. Other considerations, such as security of operational environment intelligence and hardware advances, can make in-house research and development easier, more desirable and less expensive. Our purpose here is not to attempt to quantify capacity shortfall at MSB, or its cost in terms of private sector contracts or unexplored avenues of research. But, to report MSB's own assessment that they are unable to keep pace with demands for their services.

3. Organization

The largest organizational subgroup within MSB is known as the Software Technology (ST) group. This group, currently seven software engineers, is engaged in various projects often involving only one or two people per project. These projects are primarily research oriented (e.g., rapid prototyping for feasibility demonstration). The customer sponsoring such projects is generally the project manager (not part of MSB) for the particular weapon system involved. Also, independent research of a less system specific nature (e.g., developing and benchmarking Ada library packages) may be sponsored by the branch, department or some other activity. Besides the ST group, a team of three software engineers and a program librarian are currently engaged in a development project for the Sidewinder missile. There are three software engineers in the Sparrow missile development group. There is a Software Acquisition Contracting Manager group, of two, who are dedicated to configuration and documentation management for the branch. Finally, there are a Branch Head and a secretary bringing the total to 18 personnel. Development teams are formed from ST group personnel

assets, and return to the ST group when development projects end. This is a general description of the MSB organization and the degree of flexibility in their software engineering process required to meet their commitments.

4. Current Environment

MSB has been actively improving the environment within which they work. Management tailors the software engineering process to the task at hand. Research projects may proceed employing structured methods and top down design for rapid prototyping without pushing the entire bow wave of static sequential life cycle constraints required (e.g., by DoD Standard 2167) when a project enters development. MSB employs structured methodologies espoused by Yourdon and others. MSB is actively engaged in research to demonstrate Ada feasibility for missile software. This work includes performance analysis of object code generated for potential "off the shelf" target processors. They are developing expertise in object oriented design with Ada. They are actively researching missile software domain Ada library packages and working towards reuse of design and code. They have sponsored development of an Ada code analysis metric (Halstead metric) tool, *AdaMeasure*, (Fairbanks, 1987) at the Naval Postgraduate School (NPS). They have encouraged other NPS efforts⁶ including this one, which focus on aspects of CASE resources and development.

To the extent possible with available funding, MSB has upgraded the hardware and software of their host development system. The result to date is a disjoint environment of personal microcomputer workstations with local area network terminal access to their own super-microcomputer and the central site processors. The MSB microVAX II runs the UNIX operating system and hosts various Ada compilers and their run time support tools (e.g., debuggers). Similar resources are available on the central site VAX. The personal computer workstations generally have individualized collections of disjoint tools for word processing, text editing, scheduling, spreadsheets, graphic drawing, etc..

⁶Under development concurrently with this work, these efforts will also result in June 1987 theses. While titles are not yet firm, the works are identified by subject and author:

- An Ada Terminal Interface Package, by Anthony Keough;
- Improved AdaMeasure (Henry Kilar metric), by Paul Herzog;
- Interactive Graphics in a CASE Environment User Interface, by Gregg Singer.

Recognizing not only a need, but an obligation, to remain a viable research and development resource, by remaining competitive in terms of development cost, productivity, and availability, MSB is actively investigating CASE environments.

5. CASE a Desired Solution

In the Fall of 1986, MSB began to actively explore CASE solutions to improve productivity, reduce development costs and improve product quality. Their high level requirements included automated resources supporting the following:

- CASE environment database containing code, documentation, specifications, requirements, transformations, design histories, project summaries and cost projections integrated with graphic design tools;
- library, supporting reusability of source code, documentation, tests and test data and object code;
- documentation generation, supporting their research prototype process and the development process (DoD Standard 2167 and other requirements);
- graphical analysis and design, supporting Yourdon structured analysis structured design methodologies and Ada object oriented design;
- programming support including style guidelines, static and dynamic analysis and source and object code generation;
- office automation, supporting project management.

In addition, they identified hardware resource constraints including support for:

- networked software library (database);
- modern graphics oriented methodologies and tools;
- team approach to software development.

They refined these hardware constraints further to:

- multitasking, supporting parallel simultaneous interaction with environment resources;
- mega-pixel graphics resolution, supporting multiple virtual terminals for parallel simultaneous interaction with concurrent tasks;
- mega-instructions per second, supporting resource-intensive features of the system;
- mega-bytes of main memory, supporting resource-intensive features of the system.

(Missile Software Branch, 1986, pp. 6-8)

B. CASE ENVIRONMENT PROCUREMENT ISSUES

Within DoD, procurement of any large, expensive, complex system of hardware and software (like a CASE environment) is governed by policies and standards which require such things as demonstration of economic feasibility and documentation of the development life cycle in a systematic way for management as the procurement progresses (e.g., DoD Standard 2167 requirements). Our purpose here is not to study the process, but to discuss some general issues which would arise during such a procurement. A fundamental issue is a consideration of make versus buy. By make we mean to make or have made (e.g., under contract) a system which is designed topdown for the unique organization and software engineering processes of MSB. By buy we mean a system composed of existing (off-the-shelf) products which are purchased to assemble a CASE environment. We will discuss briefly two fundamental aspects of the buy option. In the first case one buys a collection of tools or toolsets from a variety of sources, choosing each for the particular functional resource it provides. Because of the general current state of the marketplace in tools (ie: lack of consistent user interfaces, lack of interoperability, etc.) the best result of this approach is a disjoint environment assembled in a bottom up fashion. We call this a short term approach. In the other case, one buys a complete environment (in today's marketplace there are few choices) which has been designed top down as an integrated environment. We call this a long term approach.

1. Short Term Off-the-shelf Buy Approach

a. Advantages

(1) *Immediate Results.* Compared to an environment as a whole, a tool with limited functions is relatively inexpensive. This will often allow funding from lower levels within a bureaucracy with less justification and shorter procurement delays. The tool can be in the working environment much sooner.

(2) *Ease of Extensibility as User Experience and Technology Evolve Requirements.* The relatively small investment in any particular tool in the environment, allows easier justification and funding to enhance the environment by adding a tool which fulfills new requirements better than existing tools, or adds totally new functions.

(3) *Pick Best of Available Tools.* As discussed previously with regard to the dilemma of integration versus a variety of sources, this approach encourages access to the best technology available now or in the future.

b. Disadvantages

The advantages listed above lead directly to some major disadvantages.

(1) *Short Term Solutions Create Long Term Problems (e.g., Creeping Evolution of the Environment)*. Within a software engineering environment, the issue at hand is production of a software product. The product is more than just the object code. Change (maintenance in the traditional view, or evolution in the transformational prototype development view) of software is generally accepted as inevitable. To be able to change object code in an efficient and responsive manner (without starting over from scratch), is a major (if not the major) purpose for the development environment. At a minimum, the environment should facilitate the archival of the product in some durable storage media from which the development process can be recreated exactly and then evolved. Since the product is a direct result of the specific tools used to create it (and the tools themselves are programs which are not provably correct or identical), the only guaranty that a recreation from the archives is precisely the same product is if precisely the environment used in the software engineering process is also recorded in the archival process. If the environment is subject to creeping evolution the task of archiving becomes very complex as multiple versions of a tool, or even totally different tools, may have been used in developing the same or different parts of the product at the same or different times.

(2) *Disjoint Environment*. While each tool may add to overall productivity in a specific way, the additional overhead involved in using a disjoint environment will result in the overall productivity gain being less than the sum of its parts. In contrast, synergistic gains in productivity and quality should be expected from integrated tools (e.g., a debugger which works with an *object* created by an editor, as source code, and is capable of changing the object without forcing the user to shift modes (leave the debugger and re-enter the editor to make the change)).

(3) *Inconsistent User interface*. With rare exceptions, the user interfaces from one vendor of software to the next vary considerably. While many argue that this is only of concern with novice users who must learn a large number of interfaces at the same time, we feel it is a major consideration for expert users as well. The expert user may make fewer mistakes than the novice because he knows which *knobs* operate the system in each of the *modes* of operation imposed by the various disjoint tools. But, there is a cognitive investment, in navigating this modal hierarchy, which must detract from the creative work the user is trying to accomplish in the process. Also,

the training overhead required to create expert users (including acceptance of the new environment by existing users in the first place) will be much higher than with a consistent user interface.

2. Long Term Off-the-shelf Buy Approach

a. Advantages

(1) *Long Term.* Since this approach involves a *complete* environment, we are talking about a major investment in both hardware and software. Once such a system has been procured it is likely to remain quite stable for relatively long periods of time. Because of its large mass as an investment it will tend to have a great deal of resting inertia. The developers' changes will be consistent with the overall design to protect the users investment. Creeping evolution is unlikely, and any evolution is more easily traceable due to reduced complexity in the number of vendors involved.

(2) *Integrated Resources (within this CASE environment).* One should expect synergistic gains in productivity and product quality.

(3) *Consistent User Interface.* A consistent user interface is not guaranteed just because the environment is the product of a single developer. Neither is it prevented if more than one developer is involved. Since there are a variety of possibilities, and no one well accepted standard, it takes a commitment, by the lead developer, to a consistent interface *philosophy*. One relatively successful approach to this is the *Apple Macintosh* interface. While *Apple* themselves followed a consistent interface, they also invested in the future by providing the *toolbox* of resources, in system firmware, which make it easier for application developers to simply conform with the *Macintosh* interface than to invent something new and different.

b. Disadvantages

(1) *High Cost.* This approach requires an up-front commitment to a major hardware software system representing a major investment of funds relative to that involved for individual tools. Local approval and funding are less likely. Justification of the system to a higher level of a bureaucracy is generally more formal and takes longer.

(2) *Sole Source.* Unless his product has a well established market share and the vendor is clearly a healthy business concern, there is a great risk in a major investment in his product. (No one wants to be the first, and possibly only, customer.) This risk is even greater if the product involves a unique hardware architecture required to host the environment. The user may be effectively limited to the vendor's

technological and proprietary vision both for what is included in the environment today and how it will evolve in the future. The resting inertia which makes this a stable long term asset may inhibit extensibility in stride with the advancing state of the art. Also, an off-the-shelf *complete* environment may include resources which are not applicable or useful for the MSB software engineering process. The customer naturally resists paying for something he will not use.

(3) *Incompatible Data Formats (with other development environments)*. It is a natural extension of the idea of interoperability within an environment, to also consider interoperability between different environments. If for example MSB is tasked to prototype a proposed change to an embedded software product developed for a project by some contractor, the process would be significantly enhanced if the MSB CASE environment could accept and operate on the model of the system and all of the objects developed in the contractors original software engineering process and environment.

3. Make Approach

The make approach shares many of the disadvantages of the long term buy approach. In general terms it appears to far exceed existing MSB resources. In Chapter IV, we will discuss some of the inherent risk for any sole development of a CASE environment.

C. WHICH WAY FROM HERE

In order to effectively make decisions which commit scarce resources to developing a CASE environment for MSB, managers in the MSB chain of command must understand, the *software engineering problem* and how it relates to the productivity of MSB, as well as what a CASE environment is intended to be, and do, to improve productivity. An understanding of general CASE environment development issues, and principles for a *good* CASE environment will also help.

IV. CASE ENVIRONMENT DEVELOPMENT ISSUES

A. SCOPE OF CASE PROBLEMS

As previously mentioned there are a number of products on the market which use the term CASE in their descriptions but only amount to tools, or toolsets, limited to a portion of a full life cycle software engineering environment. A life cycle view of CASE entails some major development problems which are reflected in the general state of this technology today.

1. Evolutionary Development Politically Necessary

High risk is the driving force behind evolutionary development of CASE environments. Because of the size and complexity of a CASE environment, and the immaturity, instability or rapid evolution of the fundamental components involved (i.e., languages, database technology, management techniques, software engineering methods, economic models, hardware engineering, graphics, networking, ergonomics, artificial intelligence, etc.), the classic problems of software engineering apply to CASE environment development. Definition of the problem (the software engineering process) is generally incomplete, or inconsistent, and likely to remain so for sometime for the software industry as a whole. As a result, no clear industry wide set of requirements to be satisfied by the environment has emerged. Likewise, no clear agreement on fundamental issues regarding data models and component interfaces within environments or among environments have emerged. Most of domestic industry seem to lack the resources and motivation to undertake a full life cycle CASE environment development project under such high risk conditions (uncertain of the direction each of these technologies will take). As a result, most efforts have continued to chip away at the problem from the bottom up. The risk of changing technology is not likely to suddenly go away. The software engineering process may be well defined for a particular organization in which the majority of projects follow similar life cycles. However, it is likely that several distinct environment markets exist, and commitment to a single life cycle model would constitute a commitment to a single vertical market.

¹European developers seem to be way ahead in top down development of integrated full life cycle CASE environments.

2. Requirement Tradeoffs Contributing to Risk

Among the many tradeoffs involved in CASE requirements analysis are:

- low (e.g., UNIX) vs. high integration;
- closed vs. open environment (extensibility);
- language dependence vs. independence;
- monolingual vs. multilingual;
- partial vs. full life cycle support;
- single vs. multiple methodology;
- single user vs. multiple user;
- hardware dependant vs. independent;
- text vs. graphics;
- system configurable vs. user configurable;
- non-secure vs. secure;
- cost effective vs. cost exorbitant

(Henderson, 1987, p. 48). What is needed is a commitment to a CASE environment development philosophy which will allow evolutionary development of good environments, while minimizing risks from changing software engineering process requirements and continued technological advances. First, let's consider what constitutes a good automated environment for software engineering.

B. FUNDAMENTAL PRINCIPLES FOR CASE ENVIRONMENTS

The background discussion of the preceding chapters included several issues which have influenced the evolution of CASE environment efforts. We did not discover any clear cut study or statistics proving one side of certain issues to be superior to the other. One can get a feel for the trend of developments, user acceptance and the direction of ongoing research, by examining past and continuing work with environments. A strong dose of common sense can then be applied to the issues, and choices can be made which appear to be fundamentally better than the alternatives. An objective study to demonstrate that these choices are superior to their alternatives is certainly a direction for further research, but far exceeds the scope of this work.⁸

⁸It seems that few such studies are ever conducted. Such a study must of necessity follow implementation of the principles involved. Then each of the alternatives need to be applied in parallel to the same problem in an environment where other variables (software, hardware, people, etc.) are controlled (no doubt difficult and expensive). As has often been the case with past developments, if a factor

Leon Osterweil (1981, pp. 36-37) wrote that,

The essence of a software environment is the synergistic integration of tools in order to provide strong, close support for a software job. This environment must have at least these five characteristics: breadth of scope and applicability, user friendliness, reusability of internal components, tight integration of capabilities, and use of a central information repository. A support system must possess these characteristics if it is to merit the name *environment*.

This six year old view of what should characterize an environment has not generally been attacked or disproved, seems to represent the consensus of today's stated goals for environments, and is the essence of what we call fundamental principles for CASE environments.

1. Portable/Reusable CASE Resources

We view environments as a collection of resources. The collection includes:

- physical resources, consisting of computer hardware and system software and firmware;
- CASE resources, consisting of software tools implemented on the physical resources;
- manual resources, consisting of the methods and procedures necessary to the software engineering process but not implemented as CASE resources;
- human resources, consisting of the people who use and facilitate utilization (in the case of manual resources) of the environment.

Our primary focus is on CASE resources. Naturally, the CASE resources imply the minimum physical resources required for their execution. They also define the automation boundaries for a software engineering process in a given environment thereby determining the nature of manual and human resources.

CASE resources should provide the software engineering team (human resources) with a problem solving interface between the real world problem (for which they must develop a software solution) and the manual and physical resources. A broad, shallow, functional hierarchy of resources, is required to support *user friendly*

like productivity (which is inherently difficult to quantify with precision) is noticeably improved by the change, the industry tendency seems to be to accept and exploit the change. If the advantage of the change is not clear, it is resisted and either limps along with a minor market share or dies out by natural selection. In either case there seem to have been few attempts to objectively quantify the relative advantage of the alternatives involved. At best, empirical *order of magnitude* comparisons of similar issues are conducted.

goals (discussed later). By shallow, we mean a hierarchy with very few layers. This facilitates responsiveness by reducing the calling overhead required to descend through the hierarchy in order to use physical resources. Subordinate layers in such a shallow hierarchy will be broad in the sense that they will of necessity contain many resources (if modular design principles of coupling and cohesion are observed). In such an architecture, kernel utility resources (with unique, independent functionality) directly access the hardware resources or the environment data model (the key CASE resources), on behalf of tool resources which provide CASE environment services to the user interface. Such an architecture enhances portability and reusability of software components and extensibility of software systems.

The issues of portability and reusability of CASE resources and extensibility of CASE environments are fundamental to risk management. CASE environment resource user risk will be reduced if their investment is secured well into the future (inspite of hardware, methodology, and other technological advances). CASE environment resource developer risk is reduced if their products reach a broader market (various hardware and methodologies) with greater longevity. Unfortunately, the same competitive market dynamics which encourage technological innovation tend to discourage reusability and portability. Hardware and software developers who rely heavily on the direct linkage of their respective products, to control their share of the market, tend to resist (often in subtle ways) industry standardization efforts which might undermine their market leverage.

2. Integrated CASE Resources

All of the CASE resources in an environment should be integrated to facilitate coherence, control and sharing (see Chapter II) in order to yield a synergistic effect whereby the utility of the environment as a whole is more than just the sum of its parts. Recall that with respect to automated environment tools, in this instance CASE resources, we defined integrated as:

- all resources conform to a consistent user interface;
- all resources are as highly interoperable as possible;
- objects and their interrelationships are in a persistent common data format, which is meaningful to all environment resources.

The consistent user interface and interoperability allow for intuitive access to CASE resources relieving the user of much of the cognitive overhead of navigating among various tools, with various operating controls. The user can devote more of his

attention to the software engineering task at hand. Interoperability, based on manipulation of the common data model by all CASE resources, should allow the user to create or change an object by manipulating any of its displayed forms. (MacLennan, 1987, p. 1-3)

3. Open Environment

To enjoy the benefits of new technology and competitive endeavor, and encourage evolutionary development for multiple environment markets, environments should be open to extensibility. To support reusability of resources, functionality of existing CASE resources should not be diminished by new resources. To reconcile extensibility with the seemingly conflicting principle of integration requires agreement on and standardization of:

- data model used to represent objects and their interrelationships;
- interfaces of CASE resources with the data model;⁹
- interfaces of CASE resources with physical resources;
- interfaces of CASE resources with the user.

4. User Friendly

User friendly is a much overworked term, but we've chosen to use it for consistency with Osterweil. Commitment to an integrated CASE environment composed of CASE resources as described above can facilitate an event-driven user interface philosophy. Such a philosophy is characterized by:

- responsiveness, user's actions have direct results, are intuitive and spontaneous (i.e., no modes);
- permissiveness, the user can do anything *reasonable* at any time, the user decides what to do next, not the individual CASE resource (i.e., no modes);
- consistency, regardless of what CASE resource is in execution, the user's control options and the apparent response to them are consistent with the type of function being performed (e.g., anything that seems like text editing should use identical controls regardless of whether it involves labeling a graphical diagram or generating a textual document).

⁹"The database provides an integrating and unifying medium for interfacing tools without forcing them into a complex structure of interrelationships. Tools obtain their information from the database and return their results to it without having to interface directly with other tools. . . . In order to maintain flexibility it is important to avoid building bridges between pairs of tools rather than bridges into the database" (Howden, 1982, p. 326)

The *feel* of such an interface should be that the environment is waiting to serve the user as opposed to the other way around. This is done by employing an event-driven control structure where user actions are events and the system is always ready to handle them (e.g., as priority interrupts, or by polling for them). The broad shallow architecture, of the CASE resources in the environment, facilitates event handling without modality.

C. FUNCTIONAL ABSTRACTION AN APPROACH TO SOLVING PROBLEMS

The principles may not have changed significantly in six years, but CASE environments embodying these principles are not generally available. Without belaboring the point, we attribute this to the high risk of building on questionable standards in rapidly changing and relatively immature technologies. We propose a strategy, to avert some of said risk, allowing progress towards these principles.

1. Definition of Abstraction

An abstraction is a description of some object which separates the defining properties of the object from the unnecessary details about it. A software engineer is concerned with solving some problem. The tools (CASE resources) in his software engineering environment form a *problem solving abstraction*. The hardware (and some of the software), on which the problem solving abstraction (the CASE resources) are implemented, form a *physical resource abstraction* (Yurchak, 1984, p. 5).

2. Formal Specification

It is generally recognized that the operating system is an abstraction of the hardware system of primary and secondary memory resources, processor resources, and input output resources. Additional abstractions (e.g., video display resources) have also become commonplace. Such abstractions generally exhibit lack of formalism or consistency, a semantic gap, similar to the problems faced by linguists trying to specify the semantics of language constructs. "The vital property of a specification which guarantees that a correct program corresponding to it may be constructed, is *consistency*." (Lehman, 1984, p. 39) The practical problem to be solved involves the portability of software. One must be able to specify resources, in an implementation independent manner, in terms of abstract functional properties they provide. Davis (1984), using concepts developed to specify the semantics of high level language constructs (particularly abstract data types), developed a method for algebraic specification to solve some of these problems. Using such a formal specification as an

external frame of reference, correctness of a program developed from the specification can be viewed as a calculable, instead of empirical, notion (Lehman, 1984, p.39). The implication is that a *correct* implementation of a problem solving resource, layered on top of *correct* implementation of physical resources, will always behave functionally the same regardless of the implementation or hardware details. The way is then clear for development of portable, reusable, functional resources.

3. Abstraction of Physical Resources

Yurchak (1984) used Davis's algebraic formalism to specify AM, an abstract machine (physical resource) from functional requirements. Multiple instances of AM have been successfully implemented, from Yurchak's specification, on different physical hardware at Naval Postgraduate School. Implementation efforts proceed quickly and mechanically without the semantic ambiguity of less formal specifications. Work is continuing testing portability of applications running on AM when hosted by different physical hardware.

Grant (1986) functionally abstracted resources to support graphic user interfaces. He hosted his abstract resources on the Apple Macintosh and Digital Research's GEM (on the IBM PC). Applications, using only his abstract resources, are portable between the two host implementations inspite of significantly different hardware and system software (e.g., differences between color and monochrome are handled by the abstraction by placing colors within a gray scale, from light to dark, causing them to be displayed in logical shades of gray when hosted on monochrome hardware.). There is no noticeable (from a *human* interaction perspective) degradation in the response time of applications using Grant's abstract resources vs. similar native system resources (e.g., mouse tracking) on either host. This is attributed to Grant's adherence to the *broad shallow* architecture principle for portable reusable resources supporting user friendliness. At most two levels of calling overhead are added between an application resource call and the native system resources.

4. Abstraction of Environment Resources

By defining abstractly the basic functionality of CASE resources based on a useful standard data model, and implemented on abstract hardware resources, software developers may be able to drive CASE development with minimal risk from the uncertainties of hardware evolution, language evolution, and even evolution of the software engineering process. One key is agreement on a standard data model capable of representing all of the objects (i.e., real like people, programs and documents) or

imaginary like yet undeveloped programs or unhired people) and their inter-relationships which compose the software engineering environment. CASE resources must assume basic hardware and system capability as specified for the abstract hardware resources. Once a CASE resource is operational on the abstract hardware, it would be portable to any physical hardware capable of hosting the abstract hardware. Given an abstract hardware host, fully integrated environments could be assembled from abstract CASE resources. An environment *builder* could design and implement his own preferred consistent user interface which interacts with the abstract CASE and physical resources. But, ideally he would find it easier to adhere to user interface guidelines making use of CASE resource utilities which directly and efficiently use the abstract physical resources to provide a responsive, permissive, consistent, human engineered user interface. New resources could be abstracted, as technology advances, by adhering to the specified data model and interfaces.

Such an approach is directly pointed to by efforts such as CAIS and PCTE. We believe efforts in this direction hold some promise for bringing order to the current environment chaos.

5. Layers

The question of *efficiency* often comes up in connection with our advocacy of layering abstract problem solving resources on top of abstract physical resources on top of actual physical resources. This is certainly an area of concern since responsiveness is one of our user friendly requisites, and many CASE resources may be physical resource intensive (e.g., manipulation of many interrelated objects in a large project database). A key to this issue is our advocacy of a *broad shallow* hierarchy of CASE resources facilitating responsiveness of event driven user interfaces and resource intensive tools, and providing rapid access to physical resources by avoiding a deep modal hierarchy. Grant's experience indicates that this can be a viable approach for supporting user friendliness in an interactive graphic user interface. The speed of physical resources has been continuously increased by hardware advances, and more recently through multi-processor architectures,¹⁰ so it seems reasonable to argue that

¹⁰As an example, the Multi Backend Database System (MBDS) at the Naval Postgraduate School provides for distributing a database evenly among multiple off-the-shelf backend microcomputers. Database size can be doubled, with no impact on transaction time, if the number of backends is doubled. Or, the response time can be halved by doubling the number of backends while maintaining database size. The number of backends is transparent to the users who deal with MBDS as an abstract database resource which supports multiple data models and multiple query languages.

small efficiency gains in CASE resource implementation, at the cost of portability and reusability, are likely to be wasted in the long run (i.e., if you must scrap non-portable resources in order to take advantage of more significant performance gains offered by technological advances).

In addition to efficiency considerations, a major consideration, in constructing abstract resources is identifying the individual functions to be provided. As Osterweil (1981, p. 37) observed, different application areas will inevitably lead to differences in environments to support them. The bottom layer of problem solving resources should be atomic functions which directly support multiple top layer resources. As an example, an atomic resource might be a parser which is called by pretty printers, error checkers, static analyzers and compilers, etc.. The philosophy for developing environments should use information hiding to protect the integrity of these basic layers. In other words, the users of top level resources only interact with those resources. For instance, the compiler user should only use the compiler. The fact that the parser even exists should be hidden from him. Those abstracting top level resources, know the parser exists, but only access the parser in terms of its abstract functional interface. If the need arises to *jump around* a layer of abstract resources to get at some lower function, then a function which should have been abstracted has been missed. This is one reason why high order languages like Ada or Pascal don't produce portable applications. Abstraction in these languages is at an extremely high level (the programming logic level), and hardware or operating system calls are often required to handle external interfaces (e.g., input output devices). In the case of good program design these *may* be collected into abstract interface packages and documented as requiring change before porting. By abstracting at a lower level, and being committed to a philosophy preserving the integrity of layers of resource abstractions, portability and reusability of environment resources may be achieved.

6. Standards Enforcement vs. Encouragement

One thing the software industry has is plenty of standards. As part of the original STARS-SEE effort, *Institute for Defense Analyses* conducted a study of information interface related standards. They identified 772 existing standards and 422 emerging standards ¹¹ from 77, international, U.S. government, or industrial, organizations. The study focused on standards, in 25 categories (e.g., data interchange,

¹¹The category of emerging standards included both standards oriented development projects and commercial ventures becoming defacto standards by virtue of market share.

project management, graphics, programming languages, etc.), considered of possible relevance in defining integration requirements for the STARS-SEE (Nash, 1985, p. 223).

The fact that so many *standards* exist, and so many *more* are developing suggests that standards are anything *but* standard. Many standards are the result of noble effort by standards organizations. But, adherence to such standards, by developers, can be a high risk proposition. If the standard is something new and different, there is no easily predictable market for a product conforming with it. Success of such a product (its market capture) is determined by a multitude of factors. If the product is measurably or noticeably superior to some existing successful product, or provides some entirely new and highly demanded function, and is targeted for physical resources commanding a significant portion of the likely user group, it will probably be successful. This is risky business, and many standards on paper never become standards in fact. Some *standards of necessity* (e.g., hardware interconnection, external communication protocols, etc.), many of which began as *defacto* standards, are broadly accepted as mutually beneficial to industry as a whole. Other standards, such as those promoting software portability (in this case CASE resources), may be viewed favorably by users, and developers without a vested interest in particular real physical resources. However, much market selection of hardware currently involves issues concerning the breadth and depth of software applications available for that hardware. If software were more readily portable and reusable a major hardware marketing lever would be altered significantly.

As stated earlier, hardware and software developers, who rely heavily on the direct linkage of their respective products to control their share of the market, tend to resist (often in subtle ways) industry standardization efforts. If their market share is large enough, they collect strap hangers seeking some of that market. It is in this way that *defacto* standards arise. Of course, at this point the authors of the *defacto* standard, who have already profitted, may change directions radically in a bid to shake off strap hangers who have not yet recouped their investment. And so, often with different lesser players, the cycle begins again.

In a few cases, such as the DoD Ada initiatives, a particular standard, or set of standards, have been implemented and enforced by management dictate.¹² In the case of Ada, competition for DoD dollars has been the primary industry incentive to

¹²One may argue that Ada is far from being fully implemented, and that management resolve is not perfectly clear.

actually develop the resources required to support the dictated standard. One obvious drawback to this sort of approach to standardization is the fact that few interest groups have the financial *clout* required to pull something like this off. A more subtle, and in the long run possibly detrimental, drawback (to standards by edict) is the possibility that the standard may not be a very good one, but gains momentum by directive, and consumes resources which might otherwise contribute to evolution, through natural selection, of something better. And, once in place, inertia will tend to keep it there. Of course, if the standard is good, or at least acceptable, the advantages of focusing resources and effort should be significant.

The dilemma of standards enforcement vs. encouragement is not likely to be resolved. We favor standards encouragement for CASE resource functional abstraction, interfaces, and data models. Keys to standards encouragement are:

- good design, so there is little incentive to repeat the effort;
- availability, if possible make all foreseeable low level resources sufficiently efficient and readily available so there is little incentive to violate layer integrity (by jumping around it), and little incentive to *reinvent the wheel*
- guidelines, well publicised and justified philosophy of why it is the way it is and how to keep it that way.
- social change, growing recognition that standards promoting *plug compatibility* of CASE resources with each other, users, and physical resources, are also *standards of necessity*.

3. Top Down or Bottom Up

One of our major criticisms of the current state of most CASE environment development has been the bottom up path being followed. We've recognized some of the motivation for this. Commercial CASE developers are avoiding risk and playing to the disjoint off-the-shelf tools market. In order to survive, software developers (CASE resource users customers) in the competitive *trenches* often require immediate support (some of which is available in disjoint off-the-shelf tools). One significant by-product (from the long term view) of this activity has been the generation of experience, with a variety of capabilities, as a base for identifying problem solving resource functions for abstraction.

The top down activity in our CASE environment development strategy begins with the analysis of a basic software engineering process to abstractly specify the data model and interface requirements (which are the infrastructure of the environment), and the functions (at lower layers) and their aggregate (at successive higher layers).

which together with the type of data they manipulate, define the resources of the environment. Design then proceeds hierarchically with more complex resources specified in terms of more primitive resources in the adjacent lower layer. Algebraic formalism associates *meaning* to the specification of each resource, with a rigor which can be used to calculably verify implementations of the resources defined in the specifications (Davis, 1987, pp. 30-2 - 30-7).

V. FUNCTIONAL REQUIREMENTS ANALYSIS ISSUES

A. SCOPE OF THIS EFFORT

In Chapter IV, we discussed CASE environment development issues and their contribution to the existing chaos of disjoint tools, toolsets, and environments. We discussed general principles for good environments and how abstraction of resources and a formal method of algebraic specification may help to achieve those principles and alleviate continuing chaos. We believe that this approach should be developed further to *make* good CASE environment resources which become the foundation building blocks of portable, reusable, interoperable CASE environments.

In virtually all conceivable software engineering processes, starting from the top means analysis of the real world problem to be solved. It is clearly beyond the scope of this work to conduct an in depth analysis of the process required by MSB, and the functional hierarchy of CASE environment resources required to support the process. What we've done so far, falls more in the category of general familiarization. It is potentially useful as a starting point for more directed efforts.

In Chapter III, we outlined three basic alternatives for MSB CASE environment procurement:

- make;
- short term off-the-shelf buy;
- long term off-the-shelf buy.

We also indicated that the *make* alternative very likely exceeds MSB resources and is therefore infeasible. However, we would like to carry the *make* ideas, discussed in Chapter IV, a little further to illustrate some of the top level considerations involved. We are going to skirt the really difficult issues of a *standard* data model (based on the software engineering process whose definition we've also bypassed) and a data exchange interface (at a higher level than sequential character based text files). We will look at some functional design issues for a relatively well understood subset of CASE environment resources supporting individual programmer productivity (IPP). ¹³

¹³This is not intended to appear like the type of bottom up effort we have criticised. We proceed in this fashion because of time constraints, the exploratory scope of this effort, and the extremely broad scope, complexity, and uncertainty of the environment engineering task (which has contributed to the current chaotic state of environment automation in general). Our intended purpose is to advance understanding

It is noteworthy that, with the layered approach we've advocated, most of the low level resources, required to support a subset like IPP, are also required support for other high level tools. As an example, all of the user interface resources, below the CASE tools resource layer, must be in place (as do the user interface guidelines). IPP tool resources will use the user interface physical, CASE, and manual (i.e., the user interface guidelines) resources, just as all subsequent tool resources should use them, as the basis for the consistent, user friendly interface which is one fundamental attribute of an integrated environment. This sort of idea should help one to visualize the potential contribution of our approach towards *open extensible* environments without compromising integration.

B. INDIVIDUAL PROGRAMMER PRODUCTIVITY (IPP) RESOURCES

What follows is a very broad brush treatment of a few of the concerns associated with functional abstraction of CASE resources for a small part of a CASE environment.

1. Physical Resources

One might ask why (given the difficulty of bringing new standards into the marketplace) even attempt to abstractly specify physical resources. For instance, abstracting operating system level resources is tantamount to defining a standard operating system (which has already been done on paper, but has not succeeded in displacing defacto standards such as UNIX). Why not just adopt an existing defacto standard and build on top of it? This is what is generally being done today to achieve some portability and reusability. Problems include:

- lack of formalism in specification of these defacto standards, resulting in less than functionally equivalent instantiations and inherent portability problems;
- knowledge of the underlying operating system layer, encouraging, or at least enabling, undisciplined users to *bail-out* to the operating system, violating the layered functional information hiding structure to produce applications with inherent portability and reuse problems;
- dual functionality (i.e., more than one way to accomplish the same thing, especially if more than one existing standard (e.g., an operating system and a separate graphics kernel) must be combined to get at the hardware, which can

of the problem, and the potential of our problem solving approach, at several levels. Other, more specific, work to demonstrate technical feasibility of functional components of this problem solving approach (some specifically cited in this work and others just commencing or being encouraged) are in progress at Naval Postgraduate School. We hope that our work will provide sufficient background to stimulate continued efforts in an organized top down manner.

lead to implementations of higher resource layers which are affected in different ways, by changes in the components of the physical resource layer, depending on how that particular implementation accomplished something (violation of our unique atomic function interface principle for layers).

- critical functions required but not extendible to existing defacto standards (e.g., If the environment must be a trusted secure system, the very presence of an existing operating system is likely to prevent realizing security which must be designed in from the beginning).

Physical resource functions to be abstracted should be familiar. They include the hardware typically managed by the operating system, graphics interface and database management system.

a. Abstract Hardware Resource Layer

The abstract hardware resource layer represents the hardware virtual hardware ¹⁴ that will host other physical resources (operating system level resources). The challenge at this level is to abstract needed hardware functionality (which can be met with existing hardware technology) in a way that allows extension (e.g., parameterizing the interface to the next higher level in a way that should allow access to future hardware. ¹⁵) without compromising the integrity of the layer. Included should be familiar things like:

- processor(s);
- primary and secondary memory stores;
- archival storage device;
- bit mapped display;
- printer/plotter;
- pointing device;
- keyboard;
- network communications (not strictly required for IPP, but certainly a hinderance to extensibility if not available).

¹⁴The formal algebraic specification of abstract hardware may be implemented as virtual hardware hosted on some existing hardware, (similar to P-coded implementations) or it may be implemented as new physical hardware.

¹⁵Some crystal ball gazing should be beneficial, but even if the result does not allow the most efficient use of all future hardware developments, rehosting virtual devices to new hardware should still capitalize on features such as added speed while effectively porting the entire environment built above it. We see this sort of thing (on a generally smaller scale) in upwardly mobile hardware families where, through emulation, the instruction set of an older machine, runs on the newer machine, allowing porting of object code for the old machine to the new machine.

The abstract hardware resource layer represents the interface between real host hardware and an open, extensible, portable and reusable environment of CASE resources. Of course, this nucleus can be broadened by addition of other devices which must then be reflected back up through the resource hierarchy to (and down through the hierarchy from) the tool resources which can use them.

It is obvious that the physical resources constrain higher level resources, and that higher level resources drive the demand for lower level resources. One should not work independently with either set when defining the functional resource hierarchy. Instead one must begin in one place (either the top or the bottom) and model the desired functionality. Since high level resources generally require an aggregate of lower level functions, one should analyze the situation in a combined top down and bottom up fashion working both ends (required high level problem solving resources vs. available physical resources) towards a meeting point in the middle. The goal is a broad shallow hierarchy with atomic functional resources at the base which are called through the interface to higher layers by resources providing compound (or aggregate) functionality (the combination of atomic functions from below) to the interface with the layer of even more capable resources above them. Working in such a fashion one might continue populating a CASE environment IPP subset resource hierarchy as follows.

b. Abstract Operating System Resource Layer

The name of this layer is virtually self explanatory. However, the layer is expanded, beyond more conventional operating system functions, to handle database management and graphics functions. The resource categories include:

- process management (including multitasking which we consider critical to productivity);
- memory management;
- file system management;
- database system management (the database system is essential to the interoperability aspect of integration in environments);
- input output device management;
- graphics kernel.

As before, additional resources may be added (driven by the balance of requirements from above against capabilities from below). As an example, a security *kernel* might be added (with hooks to security resources added to the abstract hardware layer) and driven from a security manager (in the CASE environment services resource layer) supporting security requirements of the CASE tool resource layer.

2. CASE Resources

These are the problem solving resources. They are intended to interface directly, and only, with the abstract operating system resource layer below, and the user above. There are only two hierarchical layers envisioned (to remain broad and shallow).

a. CASE Environment Services Resource Layer

Resources at this level are the basis for *integration standards* within the CASE tool resources. These resources are the result of the *philosophy* governing such things as the user interface design. Data interface standards are also resolved at this level, and utility service resources (which have broad applicability among tool resources and provide a cohesive functional aggregate of operating system level resources) would also be included.

(1) *User Interface Service Resources.* These resources provide services which directly support the user interface guidelines.¹⁶ Their presence is intended to promote voluntary compliance with the user interface by doing much of the work in advance and giving it to tool resource developers. Included would be:

- *event manager*, the heart of a responsive user friendly interface, reports events (e.g., pointing device movements, keyboard or pointing device key presses) to the user interface and all other consistent CASE tool resources, to which they can respond by forking to event handlers, whereby tool level resources navigate the system resource hierarchy instead of the user who remains free to alter the control flow with new events (whether an event queue is polled, or events are handled as priority interrupts, will be key efficiency considerations for design);
- *window manager* services create and manipulate windows as objects displayed to convey information to the user, classes of windows include system windows (created by the system user interface tool), and tool windows (created by other tool resources), either of which may include dialog or alert windows;
- *menu manager* allows tool resources to create and display menus consistent with the user interface guidelines, and reports menu selections back to the tool (menus allow users to chose options at any time, menu options are imperatives used analogously to commands in more conventional systems (e.g., print, open) or alternatively they may be selections (e.g., font size type), user interface guidelines should provide for menu selection via pointing device or command keys, menus should not be hierarchical (avoidance of modes));

¹⁶Singer (1987) provides an in depth discussion of the user interface philosophy and the resources and guidelines required to achieve it. He also covers some implementation issues and a discussion of the potential of such a user interface for significant productivity improvement when fully exploited by advanced CASE tool resources such as visual programming tools.

- *dialog manager* used to create and control dialog windows when a tool resource *must* have more information from the user in order to continue a task (dialogs are *modal* if the user must respond before doing anything else, or *modeless* if the user can still do other things, dialogs may make use of *controls* standardized by the user interface guidelines and provided by a controls manager, or text entries (e.g., naming a file)), the dialog manager can also generate alert windows (notes, cautions, warnings) when a potentially dangerous situation arises (usually modal);
- *graphics facilities* which manage the *drawing plane* in terms of common parameters, objects, and functions (e.g., two dimensional coordinate system and conventions for defining points, objects, rectangles, regions, bit images, bit maps, patterns, cursors, graphics pens, icons, transfer modes, drawing environments (defining how and where graphics operations will take place), etc.);
- *text facilities* to perform basic text entry and editing, and handle different text characteristics (e.g., text font, face, mode, size, leading, etc.)

(Peatroy, 1986, pp. 4-37).

(2) *Data Model Manager.* The data model manager would provide for manipulation of the chosen environment process data models. The technology exists to provide sophisticated filters for converting to and from models supporting various processes both internal and external to this environment.

(3) *Utility Manager.* In the interest of efficiency and responsiveness, the environment service resources should be resident in memory as are the operating system resources and the user interface tool resource. Utility service resources (handled by the utility manager) and tool resources in general would most likely be in secondary storage. The first call to such resources should bring them into memory until they are either sent back by the user or, the end of the user session. It should be a characteristic of the environment data model that objects created in the environment are tagged with the location of all environment resources used in their creation. The utility manager should be called by a resource recorder checker function (e.g., of the database managers) to locate needed resources and bring them into memory when an object is accessed. When a required resource cannot be found the user should be told, via a dialog, so he can either supply the missing resource, or proceed in some other direction. Utility resources would include:

- *converter*
- *data filters*, various filters might be defined to allow data interchange with external environments (e.g., via the communications network);

- *binding transformers*, to allow *glueing* together parameterized objects with different native contexts (e.g., moving a language dependent code package from a network library into the abstract, language independent representation of the environment data model);
- any of a number of other possible utilities which have broad applicability among tool resources and provide a cohesive functional aggregate of system level resources (e.g., a parser or parsers, for the programming languages and data model supported by the environment, which could be used by a compiler, debugger, pretty printer, etc., or an unparser to reverse the process).

b. CASE Tool Resource Layer

This layer consist of tools which are integrated by their use of the underlying resource layers with adherence to user interface guidelines, the environment data model(s), and the manual and human resources of the environment. It is beyond the scope of this work to complete any particular portion of the abstract function typing for an environment. At this point our purpose is just to indicate the direction of such an effort.

(1) *Environment User Interface*. One might appropriately view the entire CASE environment resource hierarchy as a *super* operating system, with *this* resource providing functionality similar to the command shell or command line interpreter of a more conventional operating system. However, this resource *is* the user interface guidelines incarnate, and it exploits interactive graphic user interface principles to achieve user friendliness and enhance productivity. It is the example for other tool developers who will use the environment service resources and user interface guidelines to achieve the common user interface aspect for integration of their tool into the environment.

(2) *Project Management Support*. This category of resources for the IPP environment might include resources to help an individual manage his time, budget, or other resources. Objects generated here (e.g., schedules reports) should be designed to facilitate aggregation by the project management support resources of a Project Managers environment which is created by extending (by adding resources to ¹⁷) the IPP environment. IPP tools in this category might include:

- *project scheduler*,
- *office automation*.

¹⁷ Extensions would likely include security resources (if not already present) to control access privileges to resources objects.

(3) *System Generation and Management*. This should be a familiar category of tools commonly found in programming support environments. These tools must assist the programmer in a verifiable transformation of the model of a software system, created in the *Designers* environment, into an executable model which must be validated against the model created in the *Analysis* environment. Some of these resources should have broad enough applicability to be useful in the *Managers, Analysts, Designers and Maintainers* environment. For example, the following are necessary to effectively manage all of the various models (i.e., analysis, design, implementation, etc.) within a software project:

- *documentation generator*,
- *version maintainer*,
- *archiver*,
- *backup*.

Other tools, in the system generation and management category, would include programming language specific resources directly supporting transformation of the design model into the executable model. One consideration is exploitation of the available interactive graphics of the user interface (Singer, 1987) and the power of global models of objects (e.g., construction of objects by *selecting templates* and *setting controls* or *filling out* choices in dialogs, manipulation of objects through their displayed forms, multiple simultaneous *views*, animation, etc.) with tools like syntax knowledgeable editors, and interpreting or incremental compiling debuggers, to improve productivity. In other words, use of visual programming techniques. Another consideration is exploration of automated transformation technology to take advantage of formal specification technology and calculable verification techniques in order to deal directly (with some degree of automation) with the system model generated in the *Designers* environment. For the IPP subset we would begin with the more traditional programming support environment resources and exploit the user interface for productivity gains. Tools would include:

- *syntax knowledgeable editor(s)*,
- *compiler(s) interpreter(s)*,¹⁸
- *assembler(s)*,
- *linker librarian*.

¹⁸We prefer the use of an incremental compiler for debugging since it makes verification easier than if an interpreter is used during development of source which will later be compiled.

- *debuggers.*
- *analyzers metrics.*

(4) *System Integration and Testing.* The IPP user must deal with integration of various system modules for which he is responsible. In addition to debugging and verification, he is also concerned with validation of cohesive functional units. Resources to assist him in test set generation, regression testing, etc. are required and also form a logical base for extension to overall system integration and testing.

C. WHAT ABOUT THE REAL WORLD

The foregoing discussion presented an extremely high level view of CASE resource functional abstraction issues in a very limited scope. Hopefully, the benefit of such a discussion (in the context of the current chaotic proliferation of disjoint environments and environment resource options) will be the stimulation of well directed top down efforts to bring order to the development of CASE environments through such techniques. We will conclude with a brief discussion of the major obstacles to the success of such efforts, directions for continuing this work, and recommendations for MSB.

VI. CONCLUSIONS

A. INVITATION FOR REVOLUTION

"Welcome to the CASE revolution," proclaimed the ebullient keynote speaker at a recent symposium covering computer-aided software engineering (CASE). While well meant, those words may not have been well chosen for a technical audience ever watchful of marketing hype and still reeling from the past 'revolutions' of fourth generation languages, relational data bases, structured programming and real-time systems. . . the thought of going through yet another revolution is less than appealing to most. . . appealing about CASE. . . is that its tools. . . do not really represent revolution but rather evolution of tools and concepts. . . already embraced in the systems development lifecycle. (Huling, 1987, p. 73)

Webster's (1966, p.737) defines revolution as ". . . radical and complete change. . . ." We would agree that the CASE *concept* is evolutionary, not revolutionary. In this thesis we've acknowledged the *software problem*, and studied the evolution of software engineering towards solving it. We have little doubt that CASE environments are a natural and needed stage in this evolution. Probably the most compelling evidence of this is the huge demand for, and resultant proliferation of, disjoint CASE tools and fragmentary environments.

We've coalesced, from a variety of sources, spanning several years, a consensus of fundamental principles for good environments. We've reported on, the general state of technology which fails to adhere to these principles, and the technology and market factors which have encouraged such unprincipled bottom up developments.

We've reported on promising research, at the Naval Postgraduate School, involving formal specification of functional (physical and problem solving) resources (abstract function typing). We've proposed a top down strategy for developing integrated CASE environments in an open, extensible, evolutionary manner which could achieve standardization through functional interfaces allowing integration (both common user interface and interoperability) without conflict over advances in hardware and software technology, and supporting multiple processes, models, programming languages, etc., through its extensibility.

We've discussed the major obstacles to such a strategy. The task is difficult because the imperatives include words like *agree* and the descriptors are words like

standard. And, *agreement on standards* implies a required shift in marketing strategies, especially for those hardware and software houses whose symbiotic relationship is the basis for their competitive edge in controlling their market share. Our strategy provides for competition in hardware and software technologies directed not only towards implementing standard functional resources, but also towards defining and implementing new functional resource abstractions which will be integrated with earlier resources. While this would still allow for substantial competitive arenas, they would be different than the current arenas. This *would be* a "radical and complete change". So, it may be argued that our strategy is an invitation for revolution.

Revolutions tend to begin with a small group of protagonists who must gather a following convinced that their cause is just and that revolution is necessary. One goal of this particular revolution is relief from the current *environment* chaos and the dawn of a new age of open, extensible, integrated environments built from portable, reusable functional resources. Another goal is focusing competitive innovation on advancing the state of technology without getting bogged down just trying to cope with the chaos spawned along the way.¹⁹

We believe the only practical means of winning such a revolution is to make it seem like evolution. In our discussion of standards enforcement vs. encouragement, we favored encouragement of standards through *good design, availability, and social change* (realization that the standard is a *standard of necessity*). Social change concerning this issue is already afoot with more and more work focusing on abstraction, rigorous formalism, user interface design and object oriented software engineering. This is a relatively slow process, but it may be accelerated with a catalyst in the form of *availability of well designed* resources. Future work should be directed towards that end.

B. FUTURE WORK

Functional analysis is probably the hardest part of the task. We've discussed a combination top down bottom up process, of balancing high level requirements against physical resource constraints, in order to arrive at an abstract function hierarchy to meet the requirements. The really difficult thing is to do this without letting perceived (but not actual) constraints, derived from the way things are done today (implementations), jaundice the functional abstractions. To arrive at useful

¹⁹For example, the chaotic proliferation of programming languages, by the early 1970's, so saturated development resources and hampered development of new technology that development of anything more than rudimentary programming support tools (compilers, assemblers, linkers and loaders) was considerably delayed.

abstractions, work should progress in the context of a real world environment (keeping in mind the ultimate goal of portable resources). Detailed process and requirements analysis, and understanding, are prerequisites to the high level balancing act required in functional analysis. Working in the real world (e.g., the foundations, say an IPP subset, of a CASE environment for an organization like MSB) demands practical results vs. esoteric discourse. Practical results are the essence of catalysts for *social change*.

Once a minimal functional resource hierarchy is available, abstract resources must be formally specified. Then, parallel efforts can be applied to implementation. Completed implementation of the resources will constitute a prototype version of a CASE IPP environment. Several prototypes should be constructed from the same formal specifications, and testing should be designed to evaluate achievement of the principles for a good CASE environment. By repeating the process from functional analysis through prototype, functional evolution should add CASE resources for direct support of increasing portions of the software engineering lifecycle.

C. RECOMMENDATIONS FOR MSB

Depending on the resources available for such an undertaking, the *make* process described above could take several years (not to mention the time required to win the market revolution and see commercially available resources for constructing working integrated CASE environments). We've also said that MSB lacks the resources to undertake such a project. Other, more practical solutions are of immediate concern to MSB.²⁰

1. Near Term

Given insufficient resources to *make* their own CASE environment, and the inherent disadvantages of the available *buy* options, we decided to consider some sort of hybrid, of the available alternatives, as a potential means of means of acquiring CASE resources while achieving at least some of the advantages embodied in the

²⁰Encouragement of such development using resources which represent DoD sunk costs (e.g., Naval Postgraduate School (NPS) Master's Candidates) and are essentially *free* to MSB has the potential to contribute to the revolutionary effort in the long run, but is not likely to offer practical CASE environment solutions in the near term. Bottom up NPS work on specific tool resources (not incorporated to date under a top down CASE environment development plan) like AdaMeasure can offer limited, more immediate, practical benefits to MSB (while also contributing to their existing disjoint environment).

general principles for a good environment. We've devoted considerable thought to hybrid make buy schemes, and quite frankly there aren't many good choices. ²¹

a. Physical Resources

(1) *Hardware.* We believe that commitment to unique architectures and a proprietarily constrained source of software is a mistake both now and for the future. Flexibility now, and portability and reusability in the future, are best served by a powerful general purpose hardware suite. MSB has already defined reasonable minimum physical hardware constraints (Missile Software Branch, 1986, p. 8). At the time these constraints seemed to dictate the use of relatively high priced (\$25K - \$75K) 32-bit *professional* workstations. Recent market releases of networkable 32-bit *personal computer* workstations, rival the more expensive machines in capability and are driving prices into a far more affordable range (\$8K - \$25K). Such an affordable general purpose hardware base seems to be a reasonable first step to productivity improvement, with the capability to host CASE resources available today and into the future.

(2) *Operating System Resources.* We've already discussed the problems inherent to *standardizing* on top of an existing operating system. A traditional operating system choice is likely to be made based on such considerations as:

- What do we have the most experience with what do we use now? (In the case of MSB the answer would likely be UNIX);
- Is our current operating system adequate?
- What additional capabilities (i.e., graphics, database) are required?
- Which operating system promises to support the broadest selection of off-the-shelf tools (i.e., a defacto standard)?

And so on. We have little to offer here other than this common sense sort of approach to try to ensure the operating system will be adequate and supported until something significantly better comes along. Obviously if UNIX were kept as a defacto standard operating system, a graphics capability would be required (probably best to stick with the ISO GKS standard). Since many of the disjoint off-the-shelf CASE resources to be hosted employ their own database management, a choice on a database management system, to augment the UNIX file system, might either be a non-requirement or be dictated by the tool resources chosen.

²¹One short term option, which we won't discuss, is to concentrate on manual resources and simply wait on better options from CASE development efforts.

b. Problem Solving Resources

So far this near term discussion has sounded like straight *short term off-the-shelf buy*. Environment service resources are the level at which we can see practical potential for compromise between the short term off-the-shelf buy and some portions of the make option. But, look ahead for a moment at the disjoint tools to be bought. The tools of the most interest are likely to be the new CASE tools, offering relatively complete, language independent, support to the early software engineering phases of structured analysis, structured design, and in some cases even generation of source code. (You supply the compile, debug and test functions.) These tools in general have unique internal interfaces for interoperability. They generally have primitive, unprincipled, inconsistent, and highly modal user interfaces which are also unique. These tools generally do not consistently adhere to event driven control vs. hierarchical modality. They generally support a limited set of structured methodologies. The point we're getting at is that there is little common ground on which to base environment service resources. The internal interfaces of these various tools are generally so deeply involved in their design that it is doubtful any monetary incentive (especially something MSB could offer) would entice a developer to re-engineer his tool to interact only through MSB standard data models of objects. That leaves the user interface.

Would it be possible for MSB to develop standard user interface guidelines based on availability of some suitable service package (say GEM, assuming it is supported by the operating system of choice) and then successfully get CASE tool developers (for the tools MSB really wants) to host their tool on the service package with a user interface conforming to the MSB guidelines? Although probably less difficult than the common data model problem, the answer is still probably no. The task would not be trivial,²² and with a market of at most 18 users, a prohibitive pricetag should be expected.

One other possibility, which falls somewhere between the long term and short term off-the-shelf buy option, would be to identify a general purpose computing system meeting the minimum hardware constraints, for which an operating system supporting a widely accepted (defacto standard) well principled, event driven, user friendly interactive graphic user interface, already exists. While the original Apple Macintosh fell short of the minimum hardware constraints, the 68020 based Macintosh

²²For example, few existing tools are implemented using event driven program control, so major restructuring would be required to achieve user interface guidelines based on event driven responsiveness and permissiveness.

II, scheduled for release this summer, will come much closer. The Macintosh user interface guidelines and service resources are well principled and accepted. Originally targeted at a market of unsophisticated computer users, the Macintosh still suffers from type casting as a fancy toy. However, it is in fact a powerful system in its own right. Over a million users later, it presents a lucrative horizontal market to the software developers. Off-the-shelf software is plentiful and the user interface has survived to become a defacto standard for Macintosh application developers, while also influencing the competition. Among the off-the-shelf Macintosh software are, sophisticated syntax knowledgeable editor visual programming incremental compile and debug packages, at bargain basement prices (thanks to the horizontal market). The new Macintosh open architectures promise access to UNIX and MS-DOS. The point here is that, at least to the user interface chaos, there are alternatives. But, it takes a commitment on the part of the customer, to not accept deviation from established user interface guidelines. And, guideline adherence can be a reality if you give developers the tools required to make adherence easier than reinventing the wheel. There is, of course, always a bottom line. In this particular discussion it goes like this. Are the best (functionally, i.e., disregard the kluge user interface) off-the-shelf CASE tools available for Macintosh? What about Ada support? The answers are generally *not yet*. Can MSB alone get a developer to port his product to Macintosh (adhering to the user interface)? Probably not, but the incentive ought to be greater due to a potentially larger market.

Sadly, the bottom line of the whole near term issue would seem to be, if its a matter of survival, join the competition and buy up the disjoint tools of your choice.

2. The Future

We are firmly convinced that the future of CASE environment development lies along the path we've proposed for functional abstraction and formal specification of physical and problem solving resources. Key to this effort are standardization on user interfaces, and interoperability based on manipulation of global objects. Consistency checking, validation, verification, and testing must also be founded on the objects themselves and their interrelationships. Efforts like CAIS within the DoD seem to have a start on this path in an extremely limited and language specific way, and sans rigorous formalism. But, they are a start, and enjoy direct support from a much higher level within not only the DoD bureaucracy, but (due to clout) within the industry as a whole. If MSB wants better choices in the future, we recommend they aggressively

lobby the DoD infrastructure to expand work like CAIS in the direction we've proposed.

LIST OF REFERENCES

- Bennington, H.D., "Production of Large Computer Programs," *Annals of the History of Computing*, v. 5, October 1983.
- Boehm, B. and others, *The TRW Software Productivity System*, TRW, September 1983.
- Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.
- Booch, G., *Software Engineering with Ada*, 2nd ed., Benjamin Cummings Publishing Company Inc., 1987.
- Dahl, O., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming*, Academic Press, 1972.
- Davis, D.L., *A Method for Specifying Computer Resources in an Implementation Independent Manner*, Naval Postgraduate School, Tech. Report NPS52-84-022, Monterey, California, November 1984.
- Davis, D.L., "Interfacing and Integrating Hardware and Software Design Systems," *The Design Development and Testing of Complex Avionics Systems*, NATO Advanced Group for Aerospace Research & Development (AGARD) conference preprint no. 417, April 1987.
- Dijkstra, E.W., "The Humble Programmer," Turing Award lecture, *Communications of the ACM*, v. 15, October 1972.
- Fairbanks, K.S. and Nieder, J.L., *AdaMeasure: an Ada83 Software Metric*, Masters Thesis, Naval Postgraduate School, Monterey, California, March 1987.
- Grant, R.E., *Abstract Specification for a Graphic User Interface*, rough draft, Masters Thesis, Naval Postgraduate School, Monterey, California, December 1987.
- Henderson, P., *Software Development Environments*, tutorial notes, IEEE Computer Society, Ninth International Conference on Software Engineering, Monterey, California, U.S.A., 30 March - 2 April 1987.
- Henderson, W.E., "Contemporary Software Development Environments," *Communications of the ACM*, v. 25, May 1982.
- Huang, J., "Toss in the Trade-Is CASE Really a Cure-All?" *Communications of the ACM*, v. 29, April 1987.

Lehman, M.M., Stenning, V. and Turski, W.M., "Another Look at Software Design Methodology," *ACM SIGSOFT Software Engineering Notes*, v. 9, April 1984.

Lehman, M.M. and Belady, L.A., *Program Evolution: Processes of Software Change*, Academic Press, 1985.

MacLennan, B.J., *Principles of Programming Languages: Design, Evaluation and Implementation*, Holt, Rinehart and Winston, 1983.

MacLennan, B.J., *Programming Tools and Environments: Implementation of a Prototype Programming Environment*, Part 1, notes for course CS-4150 at the Naval Postgraduate School, Monterey, California, Winter 1987.

Military Standard Common APSE Interface Set (CAIS), draft of proposed standard, U.S. Department of Defense, 31 January 1985.

Missile Software Branch, Weapons Development Division, Naval Weapons Center, China Lake, California, *A Point Paper on the Computer Aided Software Engineering (CASE) Approach to Software Engineering*, draft, 4 November 1986.

Myers, W., "Ada: First Users - Pleased; Prospective Users - Still Hesitant," *Computer*, v. 20, March 1987.

Nash, S.H. and Redwine, S.T., Jr., *Information Interface Related Standards, Guidelines, and Recommended Practices SEE-INFO-004*, Institute for Defense Analysis, IDA paper P-1842, July 1985.

Naval Air Development Center, five volume collection of preliminary reports for the Software Technology for Adaptable Reliable Systems Software Engineering Environment (STARSS-SEE) project, 15 July 1986.

Osterried, L., "Software Environment Research: Directions for the Next Five Years," *Computer*, v. 14, April 1981.

Peatross, D.B. and Datatech Publications, *Mastering the Macintosh Toolbox*, Osborne McGraw-Hill, 1986.

Seiden, G., *Interactive Graphics in a CASE Environment User Interface*, rough draft for a Master's Thesis, Naval Postgraduate School, Monterey, California, June 1987.

Standish, J.A., "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, v. SE-10, September 1984.

Technical Report, "Requirements of Ada Programming Support Environments," U.S. Department of Defense, February 1986.

Suydam, W., "CASE Makes Strides Towards Automated Software Development," *Computer Design*, 1 January 1987.

Webster's Seventh Collegiate Dictionary, G & C Merriam Company, 1966.

Yourdon, E., *Managing the Structured Techniques: Strategies for Software Development in the 1990's*, 3rd ed., Yourdon Inc., 1986.

Yurchak, J.M., *The Formal Specification of an Abstract Machine*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1984.

INITIAL DISTRIBUTION LIST

	No. Copies
1 Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2 Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3 Department of Computer Science, Code 52 Naval Postgraduate School Monterey, CA 93943	1
4 Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, CA 93943	1
5 Department of Computer Science, Code 52Dv Attn: Daniel E. Davis Naval Postgraduate School Monterey, CA 93943	2
6 Department of Computer Science, Code 52B7 Attn: Gordon H. Bradley Naval Postgraduate School Monterey, CA 93943	1
7 Commander Naval Weapon Center, Code 3922 Attn: Carl Hall China Lake, CA 93555	1
8 Commander Naval Ocean Systems Center, Code 423 Attn: Patrick Oberdorff San Diego, CA 92152-5000	1
9 Commander Space and Naval Warfare Systems Command Washington, D.C. 20363-5100	1
10 Chief Naval Operations, OP-045 Attn: Director Information Systems Naval Department Washington, D.C. 20350-2000	1
11 J2, Joint K. Fleet USN 2215 Wilson Avenue Ft. Belvoir, VA 22060	5

END

9-87

DTIC