An Interface-Based Ada Programming Support Environment

Capt Scott A. DeLoach, USAF
Air Force Institute of Technology
October 1987

Abstract

Programming Support Environments (PSEs) have recently been the focus of much research directed at producing new methods for developing software more efficiently and reliably. The traditional approach to developing PSEs has been to create a number of novel tools, and then integrate the tools together by adding a common database and modifying the interfaces between the tools until they can work together. Recently, however, it has been recognized that the interfaces between the tools and the rest of the environment are critical, and perhaps more important, than the tools themselves. To be useful in solving the larger problems currently associated with software, a PSE must be able to adapt and provide new capabilities, and be available to a relatively large number of users on a variety of machines.

The goal of this research was to determine whether PSEs based on abstract interfaces provide enhanced portability and extensibility over traditionally designed environments. To accomplish this goal, a prototype interface-based PSE was developed and then compared against traditional environments to determine which had better portability and extensibility. First, an interface model was developed to define exactly what types of interface assumptions must be specified to define an appropriate abstract interface. Next, a prototype APSE based on that interface model was designed, specified, and implemented. Finally, this prototype APSE was measured against traditionally designed environments using original environment portability and extensibility metrics.

I. Introduction

The software crisis, characterized by late, unreliable, unmaintainable, and expensive software (Booch, 1983:6), will cost the Department of Defense (DoD) an estimated $30 billion by 1990 (Canan, 1986:46). Programming Support Environments (PSEs), which integrate the tools and data necessary for software development, have been touted as a major step toward solution of this crisis. Currently, however, many PSEs require extensive modification for use on computers other than their original hosts, thereby reducing their potential impact on the crisis (Charette, 1986:71). Also, given their high development costs, it is critical that PSEs be extensible to meet future requirement changes; unfortunately, the means for implementing PSE extensibility are poorly understood (Riddle and Williams, 1986:86). Therefore, the problem is, how to build environments that are portable and extensible?

II. Background

Portability has a number of connotations depending on the context in which it is used. As used with PSEs, portability can be defined as the ease with which a particular software component may be transferred between different solution environments (i.e., computers, operating systems, or PSEs) (Yourdon and Constantine, 1978:322). Likewise, extensibility can also have several different meanings. Riddle and Williams state that environment extensibility is the ability of the environment to adapt to major changes in the environment's capabilities (Riddle and Williams, 1986:86).

From: *Ada Letters* Volume VIII, Number 4, July/August 1988

In an attempt to encourage development of portable and extensible PSEs, the DoD has developed requirements for a so-called Ada[1] Programming Support Environment (APSE) in a document commonly referred to as *Stoneman* (DoD, 1980). Stoneman defines an APSE in terms of tools, interfaces, and its central feature, the database. Stoneman also requires the APSE to be built around a Kernel APSE (KAPSE) with precisely defined interfaces. The KAPSE, in turn, is a small set of functions (whose implementation may be machine-dependent) that provide a machine-independent interface allowing the enclosing APSE toolset to be completely independent of the underlying machine. Unfortunately, as the first prototype APSEs have been developed, the focus has been on providing a minimally acceptable toolset; a true KAPSE and its interfaces have rarely been implemented (DeLoach, 1986:17). APSE efforts that did include a KAPSE often did so as an afterthought and frequently circumvented the KAPSE (i.e., the APSE used host operating system functions or tools directly) to satisfy performance or other objectives (Elliot, 1982; Linski, 1986b). Therefore, the portability and extensibility intended by Stoneman has yet to be realized.

Recently, attention has been focused on internal environment interfaces. The Common APSE Interface Set (CAIS) (DoD, 1985) is an attempt by the DoD to define a common interface to the KAPSE "to promote interoperability and transportability of Ada software across DoD APSEs" (DoD, 1985:1). Similarly, the intent of the UNIX[2] System V Interface Definition (SVID) (Fischer, 1985) is to provide a common UNIX interface, thus promoting portability of tools and applications between UNIX implementations. Other environments such as PCTE (Gallo, 1987) and ISTAR (Dowson, 1987), have developed generic structures into which existing tools could be integrated, thus providing a common database and user interface. These works are significant because they are the first attempts to standardize a set of portable interfaces upon which environments can be built: tools built to the interface specifications are portable to all computers providing the standard interfaces.

One way proposed to build portable and extensible software (Parnas, 1977) is through the use of abstract interfaces which are defined as "a set of assumptions that represent more than one possible interface" (Parnas, 1977:6). Implementation of abstract interfaces for each system component allows part of a system to be modified without affecting other parts as long as the abstract interface between them is not affected (Parnas, 1977:5). Changes may be made to one module without concern for undesired side effects in other modules because all assumptions about that module remain valid in spite of the changes. To help achieve portability, abstract interfaces may be used to hide machine dependencies in low-level modules, while reuse of existing PSE components based on the assumptions defined in their abstract interfaces aids extensibility. Thus portability and extensibility can be facilitated through the use of abstract interfaces.

III. Research Approach

The goal of this research effort was to determine whether PSEs based on abstract interfaces provide enhanced environment portability and extensibility over traditionally-designed environments. The key to developing an interface-based PSE was to develop an interface model stating exactly the kinds of interface assumptions necessary for an abstract interface definition. Next, a prototype APSE based on that interface model was designed, specified, and implemented. Finally, this prototype Interface-Based APSE (IBAPSE) was measured against traditionally-designed environments to determine whether or not an interface-based PSE does, in fact, provide enhanced portability and extensibility.

---

[2] UNIX is a trademark of Bell Laboratories

IV. Interface Model Design

Critical factors of an abstract interface include the module overview, parameters, type definitions, undesired events, system generation parameters, general assumptions, design and implementation issues, and definition of any specialized terms (Clements and others, 1984). The IBAPSE interface model uses the Ada package syntax to specify module names, parameters, types, undesired events, and system generation parameters. Assumptions are specified in comments via condition-action pairs which define information supplied to and from a module, events (normal and failure) reported by the module, how the package state affects the module, and how module operation affects the package state. Condition-action pairs also define where and why an exception (undesired event) may be raised. The condition-action pair syntax is similar to the Ada *case* select statement, as shown below.

when condition => action;

A section of the IBAPSE *low_level_io* interface specification is shown in Figure 1. In this example, the procedure ***put*** places a string of characters to the terminal screen similar to Ada's *Text_IO put* procedure. To determine exactly what the *low_level_io* ***put*** procedure does, look at the lines directly under the the ***put*** procedure declaration which are set off by special comment markers "--:". These comment markers denote that the text following is a semi-formal comment statement that helps define the ***put*** procedure. In the semi-formal comments, quoted strings are used to explain concepts that would be otherwise clumsy or ambiguous if written using a more formal method. Freedom from having to use an exact syntax allows the module description to be presented in an Ada/pseudo-code representation, thus allowing a knowledgeable Ada programmer to understand the module functions without having to learn a formal specification language. The *put* procedure is described via these semi-formal comments; each valid character in the string (see the *valid_character* procedure) is placed on the screen at the current location. The *put* procedure also continues a string on the next line if the last position on the line is used and scrolls the screen up if the the last line becomes full, operations not necessarily evident from the Ada procedure declaration alone.

Therefore, the IBAPSE interface specification defines the procedure calling syntax, parameters, data types, generation parameters, the effect of module execution (i.e., placing characters on the screen), and assumptions that can be made about the module (i.e., when the module will start a new line and scroll the screen). Definition of terms and design and implementation notes are annotated via normal Ada comments preceding the procedure definition.

V. IBAPSE Design

5.1 High-Level Design

As stated above, abstract interfaces may be used to hide machine dependencies in low-level modules, thus allowing the rest of the software to be completely portable to all hosts with implementations of those low-level functions. (A completely portable software component is defined as one which requires only recompilation within a specified environment.) Therefore, by developing a small set of functions *(kernel)* based on abstract interfaces, the remainder of the IBAPSE *(toolset)* can access the underlying host-dependent functions through their abstract interfaces, and thus be completely portable.

To provide toolset extensibility, an approach similar to UNIX, in which tools provide a single function and may be combined together to create larger, more sophisticated tools (Kernighan and Mashey, 1984:193), was taken. Defining each tool via an abstract interface makes it appear as though each tool were a simple procedure call. As each new tool is created, its abstract interface is made available for all other tools to use; thereby, extending the available environment functions and allowing easy integration into new, more sophisticated tools. A tool's abstract interface defines exactly what function the tool provides and how to access it.

```
--
-- Procedure: Put
--   This procedure writes a character string to the
--   user's terminal.
--
-- Definitions:
--       Current Screen Position - the current location of the
--                 terminal's cursor typically in row, column
--
-- Design Decisions: The decision to automatically wrap the
--                 screen was made because not all terminals have
--                 the capability to turn off automatic line wrapping
--                 but it can be simulated on any terminal.
--
-- Implementation Notes: UNIX version 1.0.
--
procedure put (data : in string);
--: for i string'first..string'last loop
--:   when valid_character(data(i)) => "place character data(i) to current screen_position";
--:                                    screen_position := screen_position + 1;
--:                                    if screen_position > end_of_line then
--:                                      if on_last_line then
--:                                            "scroll screen up 1 line";
--:                                      end if;
--:                                      screen_position := "first character of next line";
--:                                    end if;
--: end loop;

function valid_character (item : in character) return boolean;
--: type system_defined is array (integer <>) of character;
--: unsafe_char : system_defined := "character codes that have special meaning";
--:                                 "and shouldn't be written to terminal";
--: when item not in unsafe_char => return true;
--: when item in unsafe_char =>     return false;
```

Figure 1. IBAPSE Interface Specification.

5.2 Detailed Design

To develop a interface-based environment, each IBAPSE component was designed based on its interface to the rest of the environment. As discussed in the last section, there are two basic components of the IBAPSE : a fragmented toolset and a small, machine-dependent kernel. As shown in Figure 2, to support these two basic components, a third component is required: the host computer and its operating system and peripherals.

5.2.1 Host Computer System

IBAPSE was implemented on a VAX-11/780 under the UNIX operating system and was ported to another VAX running VMS. In general, to host the IBAPSE, the underlying operating system must allow multi-processing with reentrant code to support the toolset, which is made up of several distinct processes running in parallel. The host should also have its own Ada compilation system for integration into IBAPSE.

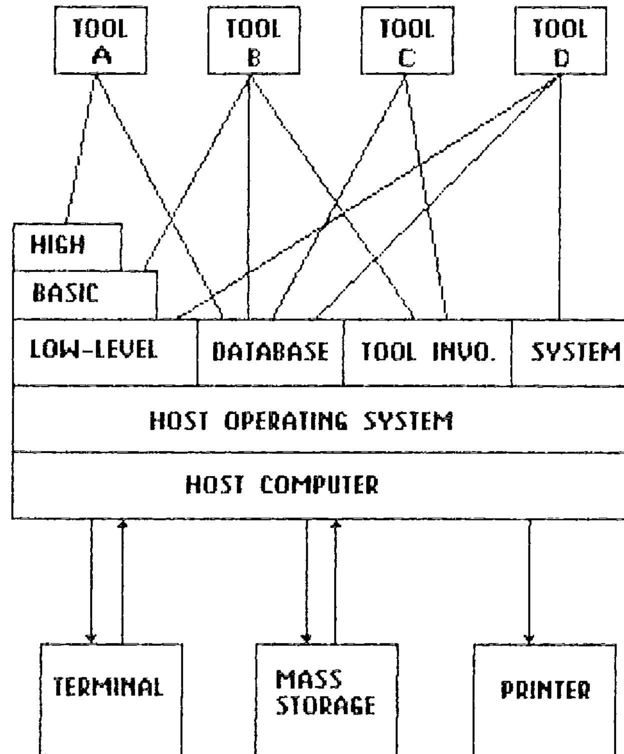

Figure 2. IBAPSE System Overview

5.2.2 Kernel Design

The kernel interface is the key to IBAPSE portability; it defines and specifies a small set of functions (manipulation of the host file system, terminal-independent input and output, and the ability to reuse existing IBAPSE tools) sufficient for the IBAPSE toolset to be completely independent of the underlying host computer system. By defining the kernel interface in terms of an abstract interface, each new implementation of the IBAPSE kernel can be validated against specified functional requirements. As long as each new kernel implementation meets the specified interface, the IBAPSE toolset will be completely portable.

The kernel consists of three distinct interfaces: the *user interface, toolset interface,* and *system-level interface.* The user interface provides toolset portability across terminals and operating systems, and consists of three separate packages: *high_level, basic_level,* and *low_level.* In the *high_level* package, modules to display menus, messages, and lists are provided, while the *basic_level* package provides lower-level functions such as character input, output, insertion, deletion, cursor manipulation, and function key input. By basing the IBAPSE toolset on these abstract functions, the toolset need not know what terminal is being used, thus simplifying the tool design task. The *low_level* interface has two main functions: host-dependent input and output. The *basic_level* and *high_level* packages use the *low_level get* and *put* operations to hide machine dependencies while they hide the individual terminal dependencies. This layered approach also provides greater extensibility than conventional input-output packages by allowing new terminal types to be added without recoding any tools or the user interface package itself. A *terminal capabilities file* (TCF), similar in concept to the UNIX *termcap* file (UNIX Programmer's Manual, 1980:termcap(5)), is used to store all pertinent terminal information; the only change required to integrate a new terminal is to add its definition to the TCF. Therefore, through the use of an abstract user interface, tools can be built that are portable between existing terminals as well as automatically extensible to include new terminal types.

The kernel's toolset interface is critical to both IBAPSE tool portability and extensibility. The toolset interface consists of two packages: the toolset invocation interface, and the database interface. The toolset invocation interface provides abstract functions for invoking and aborting other tools, receiving tool status, passing parameters, and receiving results. Although the underlying operating system mechanisms used to implement these functions vary substantially, the invocation interface provides a single abstract view of the operations. Without a well-defined, abstract tool invocation interface on which to base the toolset implementation, there would be no standard for determining compatibility between host implementations, and therefore, the functions (and the toolset which uses them) might not work identically between hosts. For example, if one implementation of the function *get_status* returns the status *no_existing_process* when a given process has completed while another returns *process_stopped,* the tool receiving the status will react differently on the two different hosts.

By using an abstract database interface instead of a virtual file system approach (as used in CAIS), the IBAPSE database provides a greater degree of portability. A virtual file systems can experience portability problems due to differing file system configuration between PSE installations (files may not be located in the same place on different environment installations thus causing tools that refer to specific locations to run incorrectly); to avoid this problem, the abstract database approach requires all PSE data be kept in a predefined hierarchical structure. By placing all data in an identical structure, tools developed at different installations know exactly how and where to access the data they need. The IBAPSE database structure groups a set of *objects* into *projects,* and *project variants,* while each object (an object is generally an Ada compilation unit) is made up of *versions* and *views.* A view is a particular representation of a data object and is the only place data is actually stored (i.e., source code, object code, executable code, specifications, design documentation, etc.). To keep the rigid database structure from degrading extensibility, the IBAPSE database allows the user to define the level of data decomposition and each installation to define and add new data views. The user can define the level of decomposition by choosing at what level objects will be defined and what views will be required for each object. Adding a new view requires only that an appropriate definition of that view be inserted into the IBAPSE *system definition file* which contains all allowable view definitions.

For example, a similar project may be decomposed differently and contain different data types on two separate installations as shown in Figure 3. As shown, the two decompositions provide two different user views of the project; however, because each branch of the tree must be specified (installation defined system defaults may be implicitly specified), a tool may access the data in exactly the same way regardless of the decomposition. If the user wants to access *main's* source code, he simply selects the desired object *(main)* and object view (source code) and then invokes a tool to operate on that view. For operations such as compilation, the user need only select the object: the object views are implied.

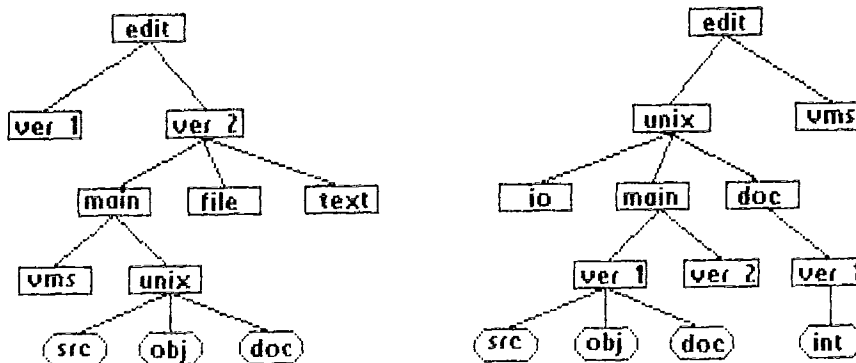

Figure 3. IBAPSE Database Decomposition.

The kernel's *system-level interface* provides an abstract interface for accessing environment variables such as *user name, user expertise,* and *terminal type,* etc. The underlying implementation is free to store these values in files or as operating system variables; however, the abstract interface totally hides this from the toolset.

5.2.3 Toolset Design

The kernel toolset interface along with each tool's own abstract interface make up the IBAPSE toolset interface, which is the key to IBAPSE extensibility. The toolset interface supports IBAPSE extensibility by not only allowing, but actually encouraging tool designers to reuse existing tool fragments in developing their tools. Each tool is invoked as a separate operating system process controlled via the kernel's toolset interface. Even though the IBAPSE kernel allows tools to reuse existing tool functions, the intricacies associated with inter-tool operations do not encourage reuse. To encourage tool reuse, the method for accessing those tools should be simple and straight-forward; therefore, each IBAPSE tool has a separate abstract interface that defines what the tool does and how to use it. Each tool is represented by an Ada interface package with the required data types, procedures, and functions necessary to invoke the tool, thus making the tool appear to be a simple procedure or function call.

As discussed earlier, the IBAPSE toolset is made up of a number of tool fragments that perform only one function. To supplement these fragments, host tools such as an editor, compiler, and linker are integrated into IBAPSE; however, like the resident IBAPSE tool fragments, the host tools are also required to have an abstract invocation interface. These host tool interfaces provide a single abstract interface to the rest of the toolset and convert the tool's host file system requirements into IBAPSE

database calls. Therefore, host tools behave identically to IBAPSE resident tools and can be accessed using the same interface from host to host (e.g., the UNIX *vi* editor interface is identical to the VMS *EDT* editor interface).

An example of a tool specific invocation interface is shown in Figure 4. The IBAPSE *help* tool (a modified version of *Help* available on the ARPANET Ada Software Repository (Conn, 1985b)) can be used via its interface by any IBAPSE tool needing a *help* function. All that is required to integrate the *help* tool into a new tool is to include the *help* interface package and then call the procedure *help:* the underlying function takes care of putting parameters in the right order and invoking the *help* tool through the kernel interface. In the case of the IBAPSE resident line editor (a modified version of the Ada Line Editor *(ALED)* also taken from the ARPANET Software Ada Repository (Conn, 1985a)), the command used is *help_package.help ("edit"),* When the command is executed, the *help* tool is invoked and immediately displays the *edit* menu instead of the main *help* menu. Without the *help* interface package, the person programming a new IBAPSE tool would have to invoke the *help* tool directly through the kernel's tool invocation interface and would have to know the name of the *help* tool, the number and order of the required parameters, and the semantics associated with those parameters. Obviously, a simple function call simplifies, and thus encourages tool fragment reuse.

To enable tools to be easily integrated into the IBAPSE toolset, all tool definitions are placed in a *tool invocation file* (TIF) which is accessed by the kernel's tool invocation interface package. Tools are integrated into IBAPSE by simply adding their tool definition data into the TIF, requiring almost no retesting of existing functions.

VI. IBAPSE Validation

To validate that IBAPSE is in fact more portable and extensible than traditionally-designed environments, IBAPSE was compared against UNIX and ARCADE (a previous AFIT APSE development) (Linski, 1986a) using portability and extensibility metrics developed and validated explicitly for software environments (DeLoach, 1987b). The portability metric measures portability as the size and complexity of the changes required to rehost the environment to another computer system, while the extensibility metrics measures an environment based on the ease of integration of both new tools and new data types. Because no well-accepted definition or standard existed against which to validate the metrics using the traditional means of validation by definition or validation by experimentation, a group of recognized experts were asked to analyze the metrics and answer questions about them (DeLoach, 1987a). Validation by expert opinion revealed that the metrics did measure important aspects of both portability and extensibility, and that the method used to quantify the results were reasonable.

6.1 Portability Metric Design

The most common approach to measuring software portability is simple and straightforward, but does not measure portability in terms of the size and complexity of the changes. This approach involves simply measuring the amount of code changed during the rehosting process, often taking the form of

$$\text{portability} = 1-(R/O)$$

where R is the total number of replacement lines of source code text and O is the original number of source code lines (Bentley and Ford, 1977). It is obvious that this "lines of code" measure does not measure portability as defined by Yourdon and Constantine; for although it does measure the size of the changes it does not measure the complexity of the changes. Therefore, a new metric had to be developed.

Because the "lines of code" metric does measure the size of the required changes, it was extended to include the measurement of change complexity. Change complexity includes many components, most of which are difficult to quantify due to lack of well-accepted definitions or understanding of the concept (e.g., understanding new host operating systems, simulating non-existent host computer

```
--: with tool;
package help_tool is

 procedure help (category : string := "main menu");
--: type help.display_menu  := "initial menu displayed by the help tool";
--: when not(tool.invocation_error)   =>   tool.invoke (help_tool);
--:                                        help.display_menu := category;
--: when tool.invocation_error        =>   raise help_invocation_error;

 help_invocation_error : exception;

end help_tool;
```

Figure 4. HELP Tool Invocation Interface.

functions, etc.). However, one aspect of complexity that is well-understood is the localization of the required changes. If a piece of software requires changes to 500 lines of code, it is easier to rehost that software if the changes are located in five or six modules instead of scattered throughout 50 or 60 modules. Therefore, size of the change will be measured by lines of code, while complexity is measured by change locality (how many modules or packages have to be modified). Using a formula similar to Bentley and Ford's, the size of a change is measured as

$$\text{size} = \frac{\text{number of lines of code modified}}{\text{total number of lines of code}}$$

Change complexity is measured by two measures: modularity and locality. Modularity is defined as the percentage of modules changed, and locality is defined as the percentage of packages changed:

$$\text{modularity} = \frac{\text{number of modules changed}}{\text{total number of modules}}$$

$$\text{locality} = \frac{\text{number of packages changed}}{\text{total number of packages}}$$

To determine how to combine the three individual measures into a single portability metric, the experts were asked how they would weight these three factors. The results, show that complexity is a much more important factor than size; therefore, using a simple average of the expert's normalized weights, overall portability is measured as

$$\text{portability} = 1 - (0.2*\text{size} + 0.4*\text{modularity} + 0.4*\text{locality})$$

For example, two systems A and B, both containing 1000 lines of code, 100 modules, and 10 packages, were rehosted to a different machine. System A required changes to 100 lines of code, 50 modules, and all 10 packages; whereas, system B required changes to 150 lines of code, 10 modules, and only 3

packages. Using the metric defined above, the portability of system A is

$$1 - ((0.2*100/1000) + (0.4*50/100) + 0.4*10/10)) = 0.38$$

while system B is

$$1 - ((0.2*150/1000) + (0.4*10/100) + (0.4*3/10)) = 0.80$$

6.1 Extensibility Metric Design

As defined earlier, environment extensibility is the ability of an environment to adapt to major changes in the environment's capabilities. Although extensibility has been recognized as being a critical factor in PSE success (Riddle and Williams, 1986:86; Reed, 1987:56,58), there has been no real attempt to measure environment extensibility. Extensibility is generally considered part of software maintainability, and is strongly dependent on the structure or modularity the associated software (Glass, 1979:158; McCall and Herndon, 1983:87). The quality of software modularity is described by the well-accepted concepts of ***coupling and cohesion,*** which categorize a module's degree of independence (a characteristic desirable in a well designed, easily maintainable system), into various levels (Pressman, 1982:158). Because PSE extensibility is similar to normal software extensibility (thinking of each PSE component as a module), a categorization similar to those of coupling and cohesion can be used to help determine the quality of a particular PSE's extensibility; the degree of component extensibility can be categorized into various levels. PSE extensibility can be classified into one of two main areas: tool integration, and database extension (Lehman and Turski, 1987; Henderson, 1987:55). To measure the quality of PSE extensibility, each of the two types of extensibility (tool integration or database extension) must be placed into various levels. Obviously, there should be at least three levels of extensibility ranging from a level where little or no modification of the PSE is required to integrate a new tool or data type, (this level is broken into two levels below) to a level that requires almost a complete redesign of a major part of the PSE; the third level, in between the first two, would be for those PSEs that require some, although not major, changes be made to environment components.

Therefore, environment extensibility can be placed is placed into one of the following categories: indirect extensibility, direct extensibility, toolset extensibility, and structure extensibility. By determining and categorizing an environment's methods for integrating new tools and data objects, weights may be assigned relative to the expected ease of extension for each category, and then used to compare extensibility between environments.

Indirect extensibility refers to the ability to integrate a tool or data object through use of data structures external to the PSE components themselves. An analogy to this type of extensibility is indirect addressing, where program code refers to a known location that holds the desired address. This type of extensibility is superior to those described below because it requires no changes to PSE components, yet does not constraint the location of the tool or data object.

Direct extensibility is similar to indirect extensibility because it requires no change to the environment components themselves. However, in direct extensibility, instead of changing an external data structure, the new tool or object is required to meet the PSE's non-modifiable reference scheme (e.g., a tool must be placed in a particular directory). Although direct extensibility may possibly require less effort than indirect extensibility, it is also less flexible. Again, an example for direct extensibility can be made, this time to direct addressing where the actual address is placed directly in the program code. If an address changes, the program must be modified everywhere that address is used.

Toolset extensibility refers to a situation where one or more tools (or kernel functions) must be altered to integrate a tool or data object. A common example of this situation occurs when a PSE's command line interpreter must be changed to recognize a new tool, or when a database tool must be changed to allow incorporation of new object types. This type of change, although possibly small, still requires recompilation and retesting of affected components.

In last type of extensibility, ***structure extensibility,*** the overall structure of the environment must be changed to incorporate new tools or database objects. Whether or not abstract interfaces are used, environment components must make some assumptions about the environment they are in; if these assumptions change (i.e., the structure of the environment changes), the environment components will have to be modified to reflect these changes, thus causing the redesign, retest, and reintegration of numerous environment components.

Development of a quantitative environment extensibility measure required associating values with each level of extensibility discussed above. Again, experts were asked to help determine what weights would be appropriate for each level of extensibility. The following equations, based on the perceived order of magnitude differences between the three original levels, were supplied to the experts for examination:

$$\text{tool extensibility} = 1*Ni + 5*Nd + 20*Nt + 100*Ns$$
$$\text{database extensibility} = 1*Ni + 5*Nd + 20*Nt + 100*Ns$$
$$\text{extensibility} = \text{tool extensibility} + \text{database extensibility}$$

where

Ni = the number of indirect references modified
Nd = the number of direct references satisfied
Nt = the number of tools modified
Ns = the number of structural modifications

The experts generally agreed that weights were of the appropriate order of magnitude. The only suggested changes were that direct extensibility should be weighted two instead of five. More importantly, the experts responses did show that resue of existing tools, was a third important factor of environment extensibility.

For example, consider an extension E to environment A through integration of a new tool and addition of a new data type to the database. Extension E may require changes for two indirect references, one tool, and a change to the overall database structure requiring the redesign of the kernel interface. Using the equations above, the overall environment extensibility = tool extensibility + database extensibility = (1*2 + 5*0 + 20*1 + 100*0) + (1*0 + 5*0 + 20*0 + 100*1) = 122.

6.3 Validation Results

After metric validation, an experiment was conducted to compare the environments for both portability and extensibility. To measure portability, the changes to the source code from an original version to a rehosted version of the environments were measured in terms of size and complexity. (Unfortunately, a second version of UNIX source code was unavailable and was therefore not included in the portability comparison.) The metric results are shown below (1.0 is completely portable).

IBAPSE Portability = 0.8654
ARCADE Portability = 0.4981

Obviously, the results show that IBAPSE is indeed more portable than ARCADE. The major difference was not the size of the changes, but the complexity (which the experts felt should be weighted four times more than the size).

To measure extensibility, IBAPSE, ARCADE, and UNIX were measured for the ease of integration of new tools and new data types. The results are shown below.

IBAPSE Extensibility = 2
UNIX Extensibility = 6
ARCADE Extensibility = 165

Once again, the metric shows that IBAPSE extensibility, even without measuring one of its strongest points, tool reuse, is superior to traditionally-designed environments.

VII. Conclusion

Basing an environment on abstract interfaces does provide enhanced portability and extensibility over traditionally-designed environments by hiding underlying implementation detail and supporting reuse of existing components. By providing a small, machine-dependent kernel defined by an abstract interface, tools may be developed that are completely portable across a wide range of computer systems and terminal types, thereby increasing the availability of quality tools and helping to alleviate the current software crisis.

References

Bentley, J. and B. Ford. "On the Enhancement of Portability within the NAG Project: A Statistical Survey," *Portability of Numerical Software Workshop.* 506-528, Berlin: Springer-Verlag, 1977.

Booch, Grady. *Software Engineering With Ada.* Menlo Park, California: Benjamin/Cummings, 1983.

Canan, James W. "The Software Crisis," *Air Force Magazine,* 46-52 (May 1986).

Charette, Robert N. *Software Engineering Environments: Concepts and Technology.* New York: McGraw-Hill, 1986.

Clements, Paul C. and others. *A Standard for Specifying Abstract Interfaces.* NRL Report 8815. Naval Research Laboratory, Washington D.C., 14 June 1984.

Conn, Richard. *Ada Line Editor, Version 2.1.* Texas Instruments, McKinney, TX. 1985.

Conn, Richard. *Help Tool.* Texas Instruments, McKinney, TX. 1985.

DeLoach, Capt Scott A. *An Interface-Based Ada Programming Support Environment.* MS Thesis-Draft, School of Engineering, Air Force Institute of Technology (AU), AFIT/GCE/ENC/87D, Wright-Patterson AFB, OH, October 1987.

DeLoach, Capt Scott A. *Portability and Extensibility Metrics.* Air Force Institute of Technology Technical Memorandum AU-AFIT-ENC-TM-87-7, August 1987.

DeLoach, Capt Scott A. "Survey of Current APSE Development and Research," in *Essays on Software Environments,* Air Force Institute of Technology Technical Report AU-AFIT-ENC-TR-86-5, December 1986.

Department of Defense. *Military Standard Common APSE Interface Set (CAIS).* Proposed MIL-STD-CAIS. 31 January 1985.

Department of Defense. *Requirements for Ada Programming Support Environments.* February 1980.

Dowson, James. "ISTAR - An Integrated Project Support Environment," *SIGPLAN Notices, 22:* 27-33 (January 1987).

Elliot, J.K., "The ROLM Ada Work Center." *ACM Letters, 2:* 97-100 (Jul-Aug 1982).

Fischer, H. *SVID as a Basis for CAIS Implementation.* Technical Report, Mark V Business Systems, December 14, 1985.

Gallo, Ferdinando and others. "The Object Management System of PCTE as a Software Engineering Database Management System." *SIGPLAN Notices 22:* 12-15 (January 1987).

Glass, Robert L. *Software Reliability Guidebook.* Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

Henderson, Peter B. "Software Development/Programming Environments," *ACM SIGSOFT Software Engineering Notes, 12:* 51-52 (January 1987).

Kernighan, Brian W. and John R. Mashey. "The UNIX Programming Environment," in *Interactive Programming Environments.* New York, McGraw-Hill, 1984.

Lehman, M.M. and W.M. Turski. "Essential Properties of IPSEs," *ACM SIGSOFT Software Engineering Notes, 12:* 52-55 (January 1987).

Linski, 2nd Lt David. *Investigation of the Common APSE Interface Set.* MS Thesis, School of Engineering, Air Force Institute of Technology (AU), AFIT/GCS/MA/86D-5, Wright-Patterson AFB, OH, December 1986.

Linski, 2nd Lt David. *Requirements and Analysis Report on the Ada Language System (ALS).* Unpublished Report, Air Force Institute of Technology, 22 May 1986.

McCall, J.A. and M.A. Herndon. "Quality Assessment: A Missing Element in Software Maintenance," *Proceedings IEEE Computer Software and Applications Conference.* Silver Springs, Maryland: IEEE Computer Society Press, 1983.

Parnas, David L. *Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems.* NRL Report 8047. Naval Research Laboratory, Washington D.C., 3 June 1977.

Pressman Roger S. *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill, 1982.

Reed, Karl. "Practical Software Engineering Environments," *ACM SIGSOFT Software Engineering Notes, 12:* 56-62 (January 1987).

Riddle, William E, and Lloyd G. Williams. "Software Environments Workshop Report," *ACM SIGSOFT, 11:* 73-102 (January 1986).

UNIX Programmer's Manual. 3rd Berkeley Distribution. University of California at Berkeley, Berkeley CA, 1980.

Yourdon, Edward, and Larry Constantine. *Structured Design.* New York, Yourdon Press, 1978.

From: *Ada Letters* Volume VIII, Number 4, July/August 1988