Educational Materials
CMU/SEI-89-EM-2
November 1989

APSE Interactive Monitor: A Software Artifact for Software Engineering Education

Charles B. Engle
Florida Institute of Technology

Gary Ford
Software Engineering Curriculum Project

James E. Tomayko
Video Dissemination Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. These materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS1
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl Shingler
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

Table of Contents

APSE Interactive Monitor: A Software Artifact for Software Engineering Education

Abstract

In 1987 the SEI began a search for a well-documented Ada system, developed under government contract, that could be used in software engineering education. The APSE Interactive Monitor (AIM) was determined to be appropriate for this purpose. This system acts as an interface between a user of an Ada programming support environment (APSE) and the programs that the user executes in the APSE. It provides facilities to support the concurrent execution of multiple interactive programs, each of which has access to a virtual terminal. Educational uses of the system are described, including use as a case study and as the basis for exercises. Software engineering topics that can be taught with the system include software maintenance, configuration management, software documentation, cost estimation, and object-oriented design.

1. Introduction

Educators have long recognized the difficulties in teaching professional software engineering concepts of large systems when students are constrained to work only on small programs. At the first SEI Faculty Development Workshop in October 1986, John Brackett of the Wang Institute (now at Boston University) suggested that the SEI could do a service to software engineering education by finding a suitable, large-scale, professionally developed software system, including development documentation, that could be made available to educators and students. Such a system could be used in many ways, such as a case study in software development or as the basis for exercises in software testing and maintenance.

In January 1987 the SEI identified several criteria for selecting an appropriate system, including that it be coded in Ada, that it include a substantial amount of development documentation, that the SEI could acquire rights to distribute it, and that it not be subject to export controls by the Department of Commerce. Through the SEI's industry affiliates, several candidate systems were located and investigated.

The first system considered was developed for NASA by Computer Sciences Corporation. Detailed study of the system showed that it relied on VAX VMS system calls to implement concurrent tasks, rather than using Ada's tasking capabilities. We decided that this made it less desirable as a teaching tool.

The second system considered was the APSE Interactive Monitor (AIM) developed for the Naval Ocean Systems Center by Texas Instruments (TI). This system made effective use of most of Ada's features and thus seemed more appropriate for education.

The system was brought to the SEI for further study. Our first goal was to determine that the code delivered was complete and usable. The executable image provided by TI was found to execute successfully on an SEI VAX system. We then recompiled all the source code to produce a new executable image, which also executed successfully. At this point, which was in December 1987, we were satisfied that we had a complete system.

An attempt was made early in 1988 to port the system to a UNIX system. This proved to be unsuccessful, but some lessons were learned that can be of use to students and instructors; these are summarized in Section 3.9.

In the summer of 1988, a documentation package was prepared, and the entire system was released to four designated alpha test sites: Boston University, the University of Maryland, Arizona State University, and the University of California at Irvine. Copy were also delivered to The Wichita State University and to the Red River Army Depot, which operates an engineering education center for the U.S. Army. The test sites agreed to investigate the use of the system during the 1988-89 academic year and then report their findings to the SEI.

This report describes the AIM system and some possible educational uses. Section 2 gives an overview of the system, including a description of some of the development documentation that is available. Section 3 describes several ways that the system might be used for software engineering education. Most of this section is based on the experiences of the alpha test sites. The appendices to the report provide additional information about the system and how to get machine-readable copies of the software and documentation. An order form appears at the end of the report.

Instructors who use the AIM system in their classes are encouraged to report their experiences to the SEI, so that they can be incorporated into subsequent packages of educational materials.

2. The APSE Interactive Monitor

The APSE Interactive Monitor was developed by Texas Instruments, Inc., under a contract from the Naval Ocean Systems Center. The project had several goals in addition to the production of the tool itself. First, it was intended to give the developers experience with Ada and object-oriented design, both of which were relatively new when the project began in 1982. Second, careful records were kept of project activities, so that a preliminary cost-prediction model for Ada software could be developed [Baskette87]. Third, after initial development on a Data General system, the AIM was rehosted on a DEC VAX system to investigate transportability issues for Ada software systems.

The original developers described the system in the following abstract:

> The Ada Program Support Environment (APSE) Interactive Monitor (AIM) is a computer program that acts as an interface between a user of the APSE and the programs the user executes in the APSE.

The AIM provides facilities to support the concurrent execution of multiple interactive programs, each of which has access to a virtual terminal. These facilities separate the interactive I/O of multiple programs into disjoint logical terminals, each of which may be displayed on the physical terminal at the discretion of the physical terminal user. The multiprogramming capabilities of modern operating systems are made accessible to the single user while providing a logical separation of interactive I/O. Instead of the typical foreground/background constraints, a user's programs may be separated into interactive/non-interactive categories. Each interactive program under the control of the AIM perceives that it has complete control of the user's interactive device. It is the intent of the AIM to provide a facility that executes within the framework of the two Department of Defense (DoD) Ada Programming Support Environments (the Ada Language System and the Ada Integrated Environment) and can be extended to other APSEs under development.

The AIM user manual chapter titled "AIM System Capabilities" describes the system this way:

Purpose

The APSE Interactive Monitor (AIM) is a computer program that acts as an interface between a user of the APSE and the programs the user executes in the APSE.

General Descriptions

The AIM allows a user to have multiple APSE programs executing while keeping their interactive inputs and outputs separate both logically and physically. Facilities are provided by a simple command language to supplement or replace the standard functions available through the APSE user interface in the area of terminal and program control.

Function Performed

The AIM will interface with page mode computer terminals.

Page mode terminals transmit and receive characters one at a time. When a key is pressed on the keyboard, the character corresponding to that key is transmitted. When a character is received by the terminal, the character is displayed or performs a simple function such as carriage return or line feed.

Additionally, cursor movement and screen editing capabilities such as cursor positioning and character and/or line insertion/deletion are provided.

Images

An AIM user defines structures called *images*, each of which is an analog of the user's display. As such, the length (number of lines) and width (number of character positions) attributes of an image are identical to that of the display. Note that only characters will be supported; no graphic support is provided. Any number of images may coexist at one time, and the user selects which image is "mapped onto" the display. Being "mapped onto" means that any changes to the information in an image are immediately reflected on the display. Only one image may be mapped onto the display at any given time.

Windows

An AIM user also defines structures called *windows*, a window being the analog of the APSE program's view of the terminal. The terminal output of the APSE program is intercepted by the AIM and directed to a structure called a window. There is exactly one window associated with each APSE program executing directly under the AIM.

Viewports

A window is mapped onto an image through a structure called a *viewport*. A viewport is a rectangular area within an image. The width of a viewport is equal to the width of the image. The length is user determined but can be no larger than the image length and no smaller than two lines (this includes a required line for the viewport header).

As many viewports can be defined as will fit on any given image. The user defines the viewports by creating associations between images and windows and defining the relationships between them (position and length). Viewports on the same image are non-intersecting. Horizontal partitioning is supported; vertical partitioning is not. The space on an image that does not contain a viewport is considered dead space and no information is mapped onto it.

A window may be associated with more than one viewport at the same time, but a specific window may only be associated with a specific image once.

The AIM system is structured as 240 separately compilable units, totaling approximately 21,000 lines of code. Approximately 10 major development documents are available, as are three volumes of a final report that documents many of the experiences of and lessons learned by the development team. Several additional minor documents, such as memoranda and presentation transparency masters, are also included. Appendix 1 describes the organization of the source code and documents on the distribution tape, and Appendix 2 presents some more detailed information about the documentation files.

3. Educational Uses of the Artifact

3.1. Ada Code Reading

At the University of Maryland, the artifact proved useful to students entering the software engineering research group as a large example of "real" Ada code. In most cases, the students were not familiar with Ada or had seen only textbook examples. The documentation was similarly used to help familiarize the students with some of the kinds of documents that are produced in an industrial software engineering project.

Of particular interest in a program of this size is the partitioning of the program into Ada packages. Simply reading the package specification can give students insight into the design process for the system. Chapters 3-5 of the report *Design and Implementation Experiences: The AIM* (Volume 2 of the AIM final report) should be read along with the code; these chapters present some of the Ada lessons learned by the original developers. This report is in directory [AIM_ARTIFACT.SOURCE_DOCS.IR2] on the distribution tape.

3.2. Software Maintenance

The AIM artifact was used as the basis for a number of exercises in a graduate course on software maintenance taught at The Wichita State University in the spring of 1989. The goal of these exercises was to teach configuration management as well as corrective, adaptive, and perfective maintenance. Some of the exercises are described below; a complete list of assignments and the syllabus for the course are contained in [Tomayko89].

The artifact runs very inefficiently, mostly because of a design decision that causes the screen to use considerable computational resources to refresh itself, a genuine fault in a windowing system. It has few "real" errors. Most are attributable to inconsistencies between actual behavior and statements in the user manual and test plans. Some "errors" appear when the system is installed where the terminals do not match the original hardware. There is some dead code and other naive Ada code.

What this means is that there are many good opportunities to exhibit to students the range of activities involved in software maintenance.

The following exercises were completed in less than two weeks each (two weeks of normal course time, not real time!) by teams of three students.

Exercise One: Reallocation of function keys

After AIM was installed and compiled, running the acceptance test suite revealed that the defined function keys did not correspond to the keys described in the user manual. Additionally, some of the functions were mapped to the arrow keys on a VT220 terminal. Because these arrow keys are used for line editing under VMS, the students decided to remap some of the AIM functions in a more logical manner and regain the use of the arrow keys. The actual change request and disposition appear in file [AIM_ARTIFACT .WICHITA_STATE.ODD_GROUP.README]CHANGE.RQST on the distribution tape.

Exercise Two: The TERMINATE function

During the execution of the acceptance test suite, the TERMINATE function, which should terminate a program running in an AIM viewport, failed to work. The students' solution made TERMINATE work, but it had side effects, as indicated in this excerpt from the students' report:

> The implementation of this change is not what is expected. The problem is that the TERMINATE function as it now stands will delete a window completely, and it does not check to see that a program is executing in that window. It acts the same as deleting a window. The reason the designed change was not accomplished is due to the way that programs get executed in a window. They do not create a new process (as seen with SHOW PROCESS). The alternative plan was to send a CNTL-C to that window, but a CNTL-C did not work on a program executing in a window when tried manually.
>
> More effort is needed to fix the TERMINATE function to match the proposed design. The way it stands, it does more now than when we first ran the acceptance tests.

Thus this exercise can be effective in showing students that changes are not always as straightforward as they seem.

Exercise Three: AIM leaves leftovers on the VMS command line

When exiting AIM, the last line that the user input remains on the screen on the VMS prompt line. This is not a critical problem but a nuisance. Locating the place for the simple fix was the essence of the solution.

Exercise Four: Code improvements

One perfective maintenance exercise that can be done with AIM is to have the maintainers improve the readability of the AIM code or the cohesion of AIM modules. This was the first large system for which the original developers used Ada, so there are inevitably some rough spots in the code and some immature packaging concepts. These can be fixed by the students if they have some reasonable Ada experience. Even if they are beginners using Ada, they can rewrite some of the code to make it more understandable to them and to gain some Ada literacy.

3.3. Documentation Issues

The AIM system includes a variety of documents that can be used to give students an idea of current industrial practice. Appendix 2 summarizes the documents available in directory [AIM_ARTIFACT.SOURCE_DOCS] on the distribution tape.

Of particular interest is Chapter 8 of the report *Design and Implementation Experiences: The AIM* (Volume 2 of the AIM final report). (This report is in directory [AIM_ARTIFACT-.SOURCE_DOCS.IR2] on the distribution tape.) That chapter identifies particular documents that are required for software developed under U.S. government contracts, and it describes some problems encountered in trying to document the system according to government standards.

The files that make up the documents are not organized particularly well (see Appendix 2). An instructor may wish to assign a student the task of organizing the document files and placing them under configuration management in a consistent manner.

3.4. Cost Estimation

Since one of the goals of the AIM project was to help in developing a cost estimation model for Ada software projects, careful data collection occurred during the development process. Three life-cycle models and three cost estimation models were examined to determine how well they matched the collected data. The result is reported in two documents. First, it is in Chapter 7 of the report *Design and Implementation Experiences: The AIM* (Volume 2 of the AIM final report). This report is in directory [AIM_ARTIFACT.SOURCE_DOCS.IR2] on the distribution tape. It is also reported in the open literature in [Baskette87].

3.5. Configuration Management

Many of the documents and all of the source code on the distribution tape appear in two formats: pure text and the format imposed by the DEC Configuration Management System (CMS) tool. Each directory for which there are two formats contains a subdirectory for each format, named REFERENCE_COPY and CMSLIB, respectively.

The importance of proper configuration management to the success of student projects based on this artifact cannot be overemphasized. For example, when Wichita State used the artifact in a software maintenance course, student groups (the "Even" group and the "Odd" group) were first required to submit plans for maintaining the code and documentation and a defined process for handling change requests. These two plans appear on the distribution tape in [AIM_ARTIFACT.WICHITA_STATE.EVEN_GROUP.TAPE]BUG_REPORT.RNO and [AIM_ARTIFACT.WICHITA_STATE.ODD_GROUP.README]CONFIG.MGMT. Note that the Even group's plan concentrates heavily on change flow and that the Odd group did a better job of specifying the exact location of all the parts of the software. A merge of the best points of each plan would result in an outstanding document.

Instead of giving the students the CMSLIB files, instructors may wish to introduce the CMS software through an exercise that takes the source code text files (in the REFERENCE_COPY directory) and places them under configuration management. This gives the students a baseline configuration before any maintenance exercises are attempted.

3.6. Testing and Quality Assurance

Instructors teaching testing and quality assurance classes can use the AIM test plans as examples. This is an especially valuable use of the AIM, as the test plans and other validation procedures are quite well documented.

The documents that are most pertinent to these topics are the acceptance test plan, the acceptance test procedures, the computer program test specification, the system/integration test plan, and the system/integration test procedures. These appear in directory [AIM_ARTIFACT.SOURCE_DOCS] in the subdirectories listed below.

ATP	acceptance test plan
ATPRO	acceptance test procedures
CPTS	computer program test specification
SITP	system/integration test plan
SITPRO	system/integration test procedures

In discussing the development of a test plan, instructors may want to have students examine the system specification, which is in file [AIM_ARTIFACT.SOURCE_DOCS.PPS.REFERENCE_COPY]PPS.RNO. The difficulties of ensuring adequate test coverage for each functional and performance requirement can be illustrated.

Additional information on testing may be found in Section 3.4 of the report *Transporting an Ada Software Tool: A Case Study* (Volume 3 of the AIM final report). This report is in directory [AIM_ARTIFACT.SOURCE_DOCS.IR3] on the distribution tape.

3.7. Object-Oriented Design Example

The AIM artifact can be used as an example of one form of object-oriented design. The experiences of the developers in attempting to apply this design method are documented in Chapter 2 of the report *Design and Implementation Experiences: The AIM* (Volume 2 of the AIM final report). This report is in directory [AIM_ARTIFACT.SOURCE_DOCS.IR2] on the distribution tape.

Instructors should recognize that the term "object-oriented" has two common interpretations. The term seems to have been widely used in the Smalltalk community in the late 1970s; in that sense it is characterized by some of the concepts of the Smalltalk language, including objects, classes, messages, methods, and inheritance. In the 1980s, the term has been used, particularly in the Ada community, to describe an approach to programming previously called "data abstraction." It is this second sense that is used in the AIM documentation.

3.8. Performance Improvement Project

As an "ultimate" maintenance exercise, a project could be to redesign the system so that it runs more efficiently. Based on a study done by the students in the Wichita State class, it would take at least the amount of time and effort available in a one-semester undergraduate course to accomplish any significant change. This assumes knowledge of Ada and some experience using it. It is more likely that a full-year graduate project course would be needed to fully solve the performance problem.

Before choosing to undertake such a project, the instructor should identify specific performance problems that are to be corrected. This might be accomplished as a side effect of using the system in another setting, such as a class doing the maintenance exercises mentioned in Section 3.2 above.

3.9. Transportability Issues

Because one of the goals of the AIM project was to investigate transportability issues for Ada software, care was taken to record procedures, problems, and solutions during the porting process. These are documented in the report *Transporting an Ada Software Tool: A Case Study* (Volume 3 of the AIM final report). This report is in directory [AIM_ARTIFACT.SOURCE_DOCS.IR3] on the distribution tape.

Some of the issues discussed in this report are important for students. Section 1.2 of the report, for example, discusses issues of designing for transportability. Chapter 2 presents the procedures to be followed in a rehosting effort. Chapter 3 identifies the problems encountered and the solutions or work-arounds that were employed.

An attempt was made at the SEI to rehost the AIM system on a DEC MicroVAX under the Ultrix operating system. Although the code could be moved to the new system and compiled using the Verdix Ada compiler, it could not be made to operate. The reason for this is that, while the AIM is very well designed and easy to port from a *language* point of view, there are

differences in the various implementations of the Ada language that make the AIM non-portable from an *implementation* point of view.

The AIM generates dynamically the tasks that it needs to support multiple windows and images. If each of these tasks is created as a separate process (in the sense of a process known to the operating system), no problems in task interaction will occur when porting the system. If, however, the runtime environment allocates new tasks as subprocesses, then there can be severe difficulties in task interoperation.

The DEC Ada compiler for the VMS system allocates tasks as separate processes that cannot block other tasks. It also has an option (not the default) for requesting that tasks be scheduled preemptively (using a time-slicing algorithm). The Verdix Ada compiler for Ultrix allocates tasks as subprocesses, which means that if one of the tasks blocks, as when waiting for I/O, then all tasks are blocked. There is no means to force preemptive scheduling.

Given the design of the AIM, a runtime system such as the one provided by Verdix is disastrous. The only solution seems to be a modification to the runtime environment (proprietary to Verdix) to allow for preemptive scheduling and separate process allocation for each task. It was thus decided not to complete the port of the AIM to the Ultrix operating system.

A possible student project is to perform an assessment of the resources required to rehost the AIM on a UNIX system. This would require that the students understand the process/subprocess issues outlined above and their relationship to the AIM system requirements. The assertion in the previous paragraph that the only solution is a modification of the runtime environment should be considered. A solution requiring redesign and/or recoding of the AIM, rather than a runtime environment change, might be proposed. The cost in terms of personnel and time should be estimated.

3.10. Subject for Static Analysis Tools

A research project on software reuse at the University of Maryland used the artifact as a test case. A static analyzer and a data binding metrics tool were run on it to help identify potentially reusable components. Unexpectedly, some strange circular data binding relationships were discovered, making it difficult to isolate independent reusable packages.

Students developing or using static analysis tools, including source code metrics tools, may find the AIM system useful as a test case. The Ada style checker (available from the SEI; see [Engle89]) is an example of such a tool.

References

Baskette87 Baskette, J. "Life Cycle Analysis of an Ada Project." *IEEE Software 4*, 1 (Jan. 1987), 40-47.

Borger86 Borger, M. W. "Ada Task Sets: Building Blocks for Concurrent Software Systems." *Proc. Second International Conference on Ada Applications and Environments*. IEEE Computer Society, Apr. 1986, 3-10.

Engle89 Engle, C. B., Jr., Ford, G., and Korson, T. *Software Maintenance Exercises for a Software Engineering Project Course.* Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989.

Tomayko89 Tomayko, J. E. "Teaching Maintenance Using Large Software Artifacts." *Software Engineering Education, SEI Conference 1989; Lecture Notes in Computer Science 376*, Norman E. Gibbs, ed. Heidelberg: Springer-Verlag, July 1989, 3-15.

Appendix 1. Contents of the Distribution Tape

The distribution tape is organized into one top-level directory named AIM_ARTIFACT, which contains three subdirectories:

SOURCE_CODE	all source code and related files
SOURCE_DOCS	all forms of documentation as delivered to the SEI from the Naval Ocean Systems Center and from Texas Instruments
WICHITA_STATE	student-written materials, including modified source code, from the software maintenance course at The Wichita State University

Further elaboration of the documents in the SOURCE_CODE and WICHITA_STATE directories may be found in Appendix 2.

Many of the documents and all of the source code on the distribution tape appear in two formats: pure text and the format imposed by the DEC Configuration Management System (CMS) tool. Each directory for which there are two formats contains a subdirectory for each format named REFERENCE_COPY and CMSLIB, respectively.

The file name extensions (the three characters after the period in VAX VMS file names) used by the original developers can help identify the kind of information in the files. The following extensions are commonly used.

ACT	Action file: used for scratch files; contain information about testing problems, design errors, etc., as seen during the development process.
ADA	Ada source file.
AIS	Compiler generated file: used in the program library; provide information to the compilation system on such things as program unit dependencies, information to support automatic recompilation, etc.
ANS	ANSI (ASCII) files: files copied from a TI990 system that was used in initial development.
CMS	Configuration management system files: files used by the CMS tool.
COM	Command files: contain procedures written in DEC Command Language (DCL).
DAT	Data files: collections of data for various programs.
DIA	Diagram files: source files for diagrams; apparently used by a diagram producing tool; in many cases the essence of the diagram can be determined from reading these files.
DOC	Document file.
DUM	Memorandum file: contain memoranda on needed changes, enhancements, etc., which the designers recorded during the development effort.
ERR	Error file: errors reported by CMS.
FOI	Foil file: documents intended to be used as overhead projector transparency masters.
G	Grammar file: contain the BNF grammar for the AIM.
HIS	History file: contain history information for CMS.
INC	Include file: files included by other files in document preparation.

LIB	Library file.
LIS	Listing file: code listings generated by the compiler.
LOG	Terminal log file.
MEM	Formatted document file: output from the *runoff* document processor.
NOT	Note file: notes and memoranda from the developers.
PRE	Presentation files: presentation documents, similar to FOI files.
RNO	Unformatted document file: input to the *runoff* document processor.
T	Initialization file: apparently initialization information for a table used by the parser generator used to develop the command interpreter for the AIM.
TOT	Consolidation file: a "total" file containing both title page and text files for a *runoff* document.
TTL	Title page file: title or cover pages for documents.
990	TI990 files: files related to the original development on the TI990 system.

Appendix 2. Documentation

Most documents have file names ending in RNO, indicating a file intended to be formatted by *runoff*, a common DEC document processing system.

The original developers' documents are in directory [AIM_ARTIFACT.SOURCE_DOCS] on the distribution tape. The subdirectories and their contents are summarized below:

ATP	Acceptance Test Plan
ATPRO	Acceptance Test Procedures
COVER_LTR	Document transmittal cover letters
CPTS	Computer Program Test Specification
IR1	Final Report on Interface Analysis and Software Engineering Techniques, Volume 1
IR2	Final Report on Interface Analysis and Software Engineering Techniques, Volume 2
IR3	Final Report on Interface Analysis and Software Engineering Techniques, Volume 3
MAINT	Installation and Maintenance Guide
PDR	Preliminary Design Review Documents
PDS	Program Design Specification
PPD	Program Package Document
PPS	Program Performance Specification
RATIONALE	Rationale for the Design of the MIL-STD-CAIS
SITP	System/Integration Test Plan
SITPRO	System/Integration Test Procedures
USERMAN	User Manual

The organization of these files and directories is essentially what was delivered to the SEI from the Naval Ocean Systems Center, which in turn was essentially what was delivered to them by Texas Instruments. There are some confusing aspects of this organization. For example, although volumes 1 and 2 of the final report are in directories [IR1] and [IR2] as described above, all three volumes are in subdirectories [VOL1], [VOL2], and [VOL3] under directory [IR3]. When students are asked to recover a particular document, it will take a little detective work to find exactly the file or files needed.

The documents (and the modified source code) produced by the students in the software maintenance course at Wichita State are in directory [AIM_ARTIFACT.WICHITA_STATE] on the distribution tape. The class was divided into two groups, called the Even group and the Odd group, each of which performed many of the same exercises. The results of the two groups are in subdirectories [EVEN_GROUP] and [ODD_GROUP], which contain the following information:

[EVEN_GROUP]

CPTS	Students' work and excerpts from original documents
CODE_CMS	Source code with students' changes
PROBLEMS	24 software problem reports
TAPE	Students' work including configuration management plan

[ODD_GROUP]

AIMLIB	Source code with students' changes
CMSLIB	Students' copies of original documents
README	Students' work including configuration management plan (see the file README.TXT in this directory for details)

Appendix 3. Network Access to the Software

In addition to distribution via tape, the SEI is considering distributing the AIM over the Internet through a mechanism commonly called “anonymous ftp”. If this proves to be feasible, detailed procedures may be requested from the SEI Education Program or retrieved via the procedure below.

The following transcript of an anonymous ftp session assumes that you are working from a system that supports ftp and that is connected to the Internet. The information typed by you is shown in bold face. Notice that the name of the user must be “anonymous” and that the password should be your normal user name. The file “readme” that is retrieved will contain the detailed procedures for getting the AIM system.

```
% ftp 128.237.2.163

Connected to 128.237.2.163.

220 fg.sei.cmu.edu FTP server (Version 4.174 <date>) ready

Name (128.237.2.163:myname): anonymous

Password (128.237.2.163:anonymous):

331 Guest login ok, send ident as password.

230 Guest login ok, access restrictions apply.

ftp> cd pub/aim

250 CWD command successful.

ftp> get readme

200 PORT command successful.

150 Opening ASCII mode data connection for readme (298 bytes).

226 Transfer complete.

298 bytes received in n.nn seconds (nn Kbytes/s)

ftp> quit

221 Goodbye.
```

AIM Order Form

The AIM software and machine-readable documentation are available from the SEI. These may be downloaded from the SEI over the Internet via the UNIX ftp facility, or they may be ordered on tape directly from the SEI. To order, please select the tape format desired (below) and return this form with payment to:

Education Program
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Checks may be made payable to **Carnegie Mellon University** and should accompany this order form.

Desired format for AIM software and machine-readable documentation:

- ❑ VAX/VMS reel tape $20.00
- ❑ VAX/VMS TK-50 cartridge tape $30.00

Send to:

Name _____

Address _____

_____

_____

_____