SEE
A Software Engineering Environment

Kevin W. Jameson

December 1987

Abstract

The SEE Software Engineering Environment is a practical, portable, software development environment whose tools and concepts are nearly independent of the edited programming language and the supporting host editor environment. The SEE environment is a *software engineering* environment which manipulates components of the software lifecycle, in contrast to *programming* environments which manipulate structured programming languages. Three prominent features of SEE which distinguish it from other environments are the use of a standard software module structure to support lifecycle-oriented software tools, the use of source code as a vehicle for the collection and analysis of project size and time cost data, and the use of tools which preserve the developer's mental train of thought and display screen context. SEE supports the four major project activities of design, implementation, documentation, and project management by providing tools and procedures which simplify or automate many common tasks. The portability of the SEE environment is evaluated based on experiences gained in moving the core of the original environment from a Lisp-based mainframe editor to a C-based microcomputer editor, and the utility of the environment is evaluated on the basis of several commercial, institutional, real-time, and application projects.

Contents

Introduction

Research on the design and implementation of software development environments has increased significantly during the last decade, generally motivated by statistics which indicate that software costs are responsible for a rapidly increasing fraction of computing budgets. For example, Boehm [9] indicates that the overall ratio of software to hardware costs has changed from 15:85 in 1955 to 85:15 in 1985, and that by 1995, a 20% improvement in software productivity will be worth an estimated $90 billion worldwide.

While many approaches to programming support environments can be found in the research literature, only a few have developed active research histories which span more than a few years. Several of the more interesting ones are summarized below. Each approach emphasizes a different method or type of software development, a different set of supported programming languages, a different internal representation of the software, and a different set of tools to support the chosen approach.

One of the most successful approaches to machine-assisted software development is collectively exemplified by the family of Interlisp programming environments [19],[34]. These systems provide extensive integrated support for software development of an incremental and experimental nature in the Lisp programming language. Interlisp editors use a structured tree as the internal representation of software, implying that the primitive functions of the editors are highly coupled to the structure of the Lisp language. A wide variety of user-level tools in these environments support development activities characteristic of experimental Lisp programming. Tools oriented toward conventional lifecycle management (such as support for specification and design) are less plentiful. Interlisp systems are good examples of the state of the art in Lisp programming environments.

A second prominent approach to development environments is characteristized by the family of extensible Emacs editor environments (Teco, Emacs[31], Multics Emacs[22][23], Zwei[38], Zmacs[33], Gnu Emacs[32]).

These environments support a conventional software development model, unlike the experimental model associated with Interlisp systems. Emacs environments differ from Interlisp environments in their internal representation of software; editor buffer contents are represented as a doubly-linked list of text lines, regardless of whether the buffer holds software written in a structured programming language or a document written in English text. An important consequence of the choice to use textual representation is that it allows the construction of language independent editor primitive functions. Thus Emacs editors can effectively support a wide variety of programming languages, in contrast to the Interlisp systems described above and the language-specific syntax directed editors described below. However, since use of a textual representation precludes the structure-oriented editing of buffer contents with editor primitive functions, Emacs editors have traditionally supported a set of optional higher level tools which manipulate the editor buffer contents in a structured, grammar dependent fashion. Lisp-based Emacs environments contain a variety of tools which support the conventional software development process, including special editor modes which support the edit-compile cycle. Tools for experimental programming such as the Interlisp *undo*, *advise*, and *history list* facilities [19] are generally absent in most Emacs implementations, but some can be found in more modern systems [35]. Well-implemented Lisp-based Emacs environments may be viewed as examples close to the state of the art in conventional programming environments [22][23][32][33], and in Lisp environments [33][38][35].

Syntax-directed editing environments [17] [12] [20][21] [26][27][28] have traditionally supported variations of a top-down refinement approach to software development, one in which programs are developed in a top down fashion by expanding syntactic templates. These environments, like the Interlisp family, use a structured tree as the internal representation of software, and thus are highly coupled to the grammar of the language for which they were designed. This limitation has been mitigated by the introduction of generator programs which are capable of constructing completely new environments from language grammar specifications [20] [26]. While most syntax directed environments have provided scarce support beyond the editing task, at least one of them [16] offers wider development support in the form of facilities such as configuration management and source code access control.

The knowledge based development environment described in [29], [36], and [37] supports a conventional development process which emphasizes the construction of software through use of an intelligent programmer's assistant which understands common programming cliches (ie. models) such as sequential search. This environment uses a textual representation of software at the lowest level, as it is implemented in the form of extensions to a Lisp-based Emacs editor [38], but at a higher level, the real work of the knowledge based environment is done using a "plan formalism" or "plan calculus" representation of the software [29] which represents the logical properties of algorithms explicitly [36]. KBEmacs supports Lisp and limited dialects of some procedural languages such as ADA.

Environments based on the automated transformation of software specifications are also subjects of active research. These systems [2][14] [4][6] [11] emphasize the partial or fully automated transformation of program specifications into acceptable executable forms, usually by successive refinement of intermediate representations. The transformation process is completely automated in some cases [6], but requires human assistance in others [2] [11].

Several themes are visible in the environments described above. First, many development environments which see heavy everyday use in educational, research, and commercial applications are Lisp-based systems. This is largely due to the extensibility and power of such environments, but is also a result of long term development in the research community (at least a decade for both Interlisp and Emacs). A large community of knowledgeable developer-users has played a significant role in the growth and power of these Lisp development environments.

Second, the generality inherent in the textual vs. grammatical representation of software is clearly shown by comparing Emacs environments with those which use structured internal representations (Interlisp [19], Cornell Synthesizer[27], Gandalf ALOE[21]). A comparison shows that Lisp-based Emacs environments which use textual representations support many languages, and are easily extended by the user to provide new facilities of arbitrary complexity (eg. KBEmacs[37]), whereas structure oriented environments whose internal representation is based on the grammar of the edited language necessarily support only one language, and in several cases (Interlisp excepted) are not easily extended by the user because they lack good extension languages.

Third, the importance of structure oriented editing tools are illustrated by their presence in all but one (automated transformation) of the environments discussed above, and then only because human manipulation of structured software is not a large part of transformation environments. Rather, the specification language is generally entered once and thereafter manipulated through machine-assisted transformation processes. It is highly probable that such environments could also benefit from structure oriented editing tools in situations where human manipulation of the specification language is required.

In summary, the last decade of research has produced several prominent approaches to the problem of programming environment design. The approaches most visible in the literature are characterized by Lisp based environments for Lisp programming (Interlisp[19]), Lisp based environments for conventional multi-language programming (Emacs[31]), syntax directed environments dedicated to one language (Cornell Synthesizer[27], Gandalf ALOE[21], many others), automated transformation and refinement systems ([2], PSI/SYN [4][6], PDS [11]), and knowledge based 'programmer apprentice' environments (KBEmacs[37]). These environments suggest that Lisp environments are very capable of supporting practical software development environments (Interlisp, Emacs), that the use of a textual, language independent representation adds considerable generality to an environment (Emacs), and that structure oriented editing tools are of significant utility (Interlisp, Emacs, and many syntax directed environments).

Several key issues in the design of programming environments can be identified in the environments summarized above. These issues concern

- the type of software development, and thus the specific development activities, to support (experimental programming vs. conventional application development)
- the type of internal software representation to use (textual vs. structured)
- the generality of the environment (support one language or many languages), and
- the capacity of the environment to accommodate user written functions which extend or modify the functionality of the original environment (extensibility).

Outstanding Problems

While the programming environments discussed above represent very competent solutions for experimental Lisp programming and for manipulating structured languages in general, they do not address many of the outstanding problems commonly cited in the current software literature. A partial list of such problems could include high development and maintenance costs, out of date documentation, lack of standardization in software development, and the difficulty of tracking development progress.

The SEE environment described in the rest of this paper is primarily intended to address these outstanding problems, and differs from previous environments in several fundamental ways:

- The SEE environment contains no editing, compiling, or debugging facilities of its own; its tools and concepts must be supported by a host editor environment. In contrast, other environments typically concentrate on providing these features.

- The SEE environment is not intended to be a *programming environment* for the structured editing and debugging of programs in a particular structured programming language; instead, it is intended to be a *software engineering environment* which manipulates components of the software lifecycle (software, software modules, project documents, and project cost/metric data) throughout a large portion of the lifecyle.

- In contrast to other environments, which emphasize the structure and grammar of a programming language statement for better treatment by the *editor,* the SEE environment de-emphasizes the structure and grammar of statements in the programming language. Instead, the SEE environment emphasizes the meaning of the statement from a lifecycle or software engineering point of view for better treatment by *lifecycle oriented tools.*

- The developer's freedom to choose various programming styles, documentation styles, and project reporting styles in other environments is limited in the SEE environment in favor of standard models of software module structure and project documents.

In what follows, the SEE environment is introduced with presentations of the project design objectives and of the software representation models used by the environment. Subsequent sections of the paper describe the SEE environment from various lifecycle viewpoints and evaluate the environment on the basis of several development projects. The closing sections of the paper discuss the transport of SEE concepts to other environments, the contributions made by this work, and possible future research directions.

Chapter 1

The SEE Environment

1.1 Summary of SEE Design Objectives

The major design objective of the SEE project was to implement an environment of concepts and tools which would address several classic industry problems:

- High production costs
- Lack of product documentation
- High maintenance costs
- Lack of project management information

The SEE environment addresses these problems in the context of projects staffed by one individual, or by a small team of developers (programming in the small).

Secondary design objectives were to

- build a practical, realistic environment which could be understood and effectively utilized on real world projects by individual developers who lacked an extensive educational background in software engineering.
- implement an environment whose concepts and tools could be easily moved to any reasonable machine environment, thus preserving investments in software and training across machine environments.
- implement an environment whose concepts and tools could be used with a variety of modern programming languages, thus preserving investments across programming languages.

- represent software in a standard format which would emphasize the concepts embodied in the software and deemphasize the actual programming language. Such a standard format could be manipulated by software tools and would improve the readability of software by emphasizing important information for human developers.

- construct tools which would directly implement, as a single atomic operation, manual command sequences which represent but one conceptual operation to the developer. Such tools would significantly increase productivity by replacing several commands with one, and would preserve the developer's mental abstraction level by handling the concept implementation details.

The only constraint on the project was a low implementation budget (one developer). This lack of resources forced the implementation approach to emphasize low overhead, low implementation costs, simplicity of representation, and practicality of new concepts and tools. The limitation did not adversely affect the project – instead it focussed effort on productive and useful development activities.

1.2 Summary of the Emacs Editor Host Environment

The SEE environment described in this paper was implemented as a mode package in the Multics Emacs editor environment described below. Some tools were implemented in PL/1 so they could also be used outside of the editor environment.

The Multics Emacs editor is an extensible, Lisp-based editor which was written in 1978 for the Honeywell Multics operating system. The normal capabilities of the editor can be augmented simply by loading new Lisp functions into the Lisp editor environment, where they become indistinguishable from other editor functions. While any Lisp function can be loaded individually, extensions which involve a large collection of related functions are typically placed in one file and loaded as a group.

The term 'editor mode package' is often used to describe such a group of related Lisp extensions in the Multics Emacs environment. Mode package functions typically manipulate the buffer contents in a structured manner based on intrinsic knowledge of the buffer contents, but may also be designed for other non-editing purposes. The generality and flexibility of the Lisp extension mechanism allows the construction of very powerful and knowledgeable tools.

The Multics Emacs editor can support very complex mode package functions. For example, complete electronic mail system interfaces have been written in the form of editor extensions, as have electronic meeting and operating system interfaces. Each mode package allows the user to interact with the mode-supported system through the medium of an editor buffer.

In the domain of programming support, mode packages have been written for (among many others) the Lisp, PL/1, and Fortran languages (eg. lisp-mode, electric-pl1-mode). Each mode package typically provides functions for compiling source code buffers without leaving the editor, for automatic indenting of code as it is entered, and in some cases, for automatically generating parameter lists and attributes in declarations of system subroutines and external variables.

Lisp extension functions written in the Multics Emacs editor environment have access to a large set of primitive cursor navigation and text manipulation functions, as well as to many operating system functions through the Lisp editor's own interfaces. Moreover, the functions have access to the full Lisp environment in which the editor resides. This virtually seamless interface across the editor and operating system is a significant conceptual and practical advantage to the extension writer.

1.3 Summary of the SEE Representation Model

The SEE environment uses a standard software module structure as its internal and external software representation. Stated simply, the standard module structure consists of a normal software module with special 'header comments' and 'task definitions' (or 'task comments') inserted at various places in the text of the module. A sample PL/1 module in standard format is shown in Section A.1 on page 32. Header comments and task definitions defined by the standard format model are listed and described in Section A.2 on page 35.

The main function of header and task comments is to tag conceptually important information in the software module. Such tagging supports both humans and software tools in their attempts to locate and manipulate interesting information.

Header comments, as a class, are generally associated with the declarative information of a module. For example, the Function header tags a textual description of the module function. Similarly, the 'Privileged Init' header tags parameter and variable initializations upon which other initialization statements in the 'Init Storage' section depend.

In contrast to the association of header comments with declarative information, task comments are intimately associated with the executable information of a module. They represent a step by step English description of the module's algorithm, and, ideally, are conceptually associated with the executable code lines which immediately follow them. Thus each line of executable code in the module should be performing some function which is directly related to the description given in the immediately preceding task comment. The next few paragraphs more fully explain the role of task comments in the SEE environment. Afterwards, the paper continues by describing other general features of the standard format.

The physical and conceptual binding of executable code to preceding task comments has several benefits. First, the physical proximity of the two components not only ensures that each section of executable code is accompanied by a conceptual description in the form of task comments, but also helps the developer to match the conceptual levels of code and task comments. A developer who sets the conceptual level of task comments too low finds that a (redundant) task comment is necessary for each line of code; too high, and task comments approach the conceptual level of the function description, covering many lines of code without really describing the algorithmic steps involved.

Second, since the task comments (entered during the detailed design phase) are physically interleaved with the code as it is added in the implementation phase, deviations from the detailed design during implementation are instantly obvious to the developer – the code lines being generated do not functionally support the preceding task definitions. This is an immediate and unmistakable signal to the developer that the current implementation is inconsistent with the intended program design.

Finally, the proximity of a task comment to its related code ensures that maintenance changes to the code are easily compared with, and cheaply propagated to, the associated task comment documentation vehicle. In contrast to the SEE approach, methods which advocate placing a large block of pseudo code at the beginning of a module do not share this ease of comparison and modification, particularly if modification work is done on an 80x25 display screen – associated components of documentation and code are often separated by several display-screens of intervening text. This separation, though seemingly small, is very likely a significant factor in the problem of out-of-date documentation; the SEE approach reduces it to a minimum value.

One generally significant feature of the standard format is that it tends to make most software similar in appearance. Over a wide range of languages, program applications, and machine environments, the location of important conceptual information is relatively constant and clearly marked by header comments.

Another advantage of the standard format is that the header comments serve to mark the *absence* of information as well as the presence of information. Developers are not required to search the entire set of declarations in a module for a particular type of information. The location of information, whether present or absent, is constant and clearly marked. For example, the dependency of a module on system subroutines is clearly indicated by the code (or lack of it) appearing below the System Dependencies header.

Finally, the presence of standard headers in every module stub encourages the developer to complete the template module form by entering appropriate information under the visible headers, thus improving the overall level of product documentation. Software tools provide aid by automatically installing the headers as part of the stubbing process, and by positioning the display cursor on fields which should always be completed.

The general scenario model supported by the standard module structure is one in which a software tool textually searches in the editor buffer for a specific header comment before manipulating the information associated with the header. For example, an editor extension which supports the entry of declaration statements collects declaration information from the user, saves the current cursor location, moves the internal cursor to the appropriate header by means of a textual string search, formats and installs the declaration into the buffer below the header, and finally, restores the internal cursor to its original position. Paraphrased Lisp code for such a tool is shown in Section A.3 on page 37.

One advantage of implementing headers in the form of textual comments is that it provides an effective, near language-independent means of supporting machine assisted manipulation of interesting information. Textual header comments allow simple tools to effectively manipulate information which is spatially associated with a header comment in the editor buffer, whether or not the information can be represented in the programming language grammar. This is a significant advantage, as it allows non-grammatical, *lifecycle-oriented* information to be brought into the realm of machine-assisted processing. In the past, such information has traditionally been carried in the form of unstructured, untagged comments, thus forcing manual processing and increased development costs.

A second advantage is that the approach offers increased portability between language environments. Modifying the headers and associated tools for use with a new programming language typically involves simple changes related to comment syntax instead of large changes related to language grammar. For the most part, tools and headers in the SEE environment are table driven with the particular comment symbols of the edited language. Headers and tools in the SEE environment have been used with several modern programming languages (PL/1, Pascal, C).

An important underlying philosophy of the SEE environment representation is to keep useful information in the source code wherever possible. This includes project documentation, design information, and project time costing data. The major advantage of this philosophy is that embedded information is not easily separated from the executable code – thus investments in documentation and time cost records will not be separated and lost during the product's lifetime. Rather, the information will be preserved for analysis and future reference. A second advantage is that such online information is easily manipulated by software tools for updating and reprinting. In contrast, offline documentation stored in binders is not always updated, and almost never reprinted once it falls out of date. Many graphical design documents fall into this latter category.

On the negative side, the philosophy has several disadvantages. First, embedding information clearly results in larger files which often require 50-100% or more disk space. Second, the development and maintenance of modules in standard format is more costly than traditional methods if no tools are available to manipulate the standard format. (In the SEE environment, the converse is true because extensive tool support is available). Finally, some developers object to the relatively heavy levels of documentation in the module, claiming that the code should be self-documenting and that the header sections and task comments only make it more difficult to understand the code.

1.4 A Note on SEE Design Phase Practices

Not all design phase activities in the SEE development model are the same as those in the conventional lifecycle model. Instead, some activities which are normally associated with the conventional implementation phase are classed as design phase activities in the SEE model. This is because the SEE development model expresses the program design in the form of software module stubs, whereas normal development practices tend to express the program design in a written or graphical non-software form. Thus the creation of module stubs is viewed as a design phase activity in the SEE development model and an implementation phase activity in the conventional model. This practice is feasible because the SEE environment provides mechanisms for automatically generating conventional design documents from the information contained in the module stubs, and is significant in that it provides an alternative to normal design documentation practices. The conventional and SEE approaches both yield comparable design documents.

The next sections of the paper discuss SEE environment support for the project activities of design, implementation, documentation, and project management.

Chapter 2

Lifecycle Support in the SEE Environment

2.1 Support for Design Activities

Three main design activities which are supported by tools in the SEE environment are the creation of module stubs, the organization of those stubs into a hierarchical calling structure, and the generation of calling tree reports from existing collections of modules.

The creation of module stubs in the SEE environment is supported by a tool which generates compilable module stubs in the standard format described earlier. The tool prompts the developer for a module name, a list of parameters, and a list of 'include file' names. Once the necessary information has been obtained, the tool creates the stub and, if necessary, adds declarative statements to describe the module parameters. The developer is not required to specify attributes for parameters whose names and attributes are known to the declaration tools. The tools match each incoming name against a list of known, user-defined names, and for each recognized name automatically insert the appropriate attributes and descriptive comments into the buffer. This approach frees the developer from the tedium of repeatedly documenting common variables and enhances the overall level of product documentation.

After the stub is created, the cursor is left in the Function description section of the module, encouraging the developer to immediately enter a description of the module function. Obtaining this description from the developer is important because the information plays an important role in all subsequent lifecyle phases. General editing of the module stub is possible at this time, allowing the developer to enter more text or otherwise manipulate the module before writing it out to disk storage. The number of keystrokes required to create a standard module stub using this method is easily an order of magnitude smaller than the number of characters in the created stub.

A second design activity supported by the SEE environment is the organization of modules into a connected 'calling tree' which represents the hierarchical calling structure of the program. The developer proceeds by entering declarative statements (external function declarations) into modules which call other modules. This activity makes use of the SEE declaration tools, so attributes for user defined names are automatically entered as the names are recognized. The developer manually enters attributes for unrecognized names. As was the case with the stubbing activity, the size of the keystroke input during this process can also be an order of magnitude smaller than the number of characters in the inserted declarations. Hundreds of cursor movement operations are saved during this process because declarations can be inserted from any point within the file.

Finally, the production of a calling tree report is supported by a tool which attempts to build a tree on the basis of the external declarations contained in the modules which are to be in the tree. The tool obtains a list of all source file names in the current working directory, extracts the declarations in the System and External Dependencies sections of each module, and constructs a tree using the extracted dependency information. Modules which have not been bound into the calling tree with declarations are displayed separately, thus helping the developer to detect omitted declarations. Operating system functions are specially marked in the calling tree to distinguish them from application functions, providing a quick visual indication of the program's operating system subroutine dependencies.

Generation of a calling tree report is possible at any time during the lifecycle, and is always based on the modules in the working directory. Thus interesting partial calling trees may be constructed by controlling the module population of the directory. A calling tree report for the project document is created by this tool as part of the automated project document creation process (described below).

2.2 Support for Implementation Activities

Many implementation phase activities are supported by tools in the SEE environment; several of them are discussed below.

Declaring variables and function calls is a common operation in projects which use modern typed languages, and hence is of significant interest from both the viewpoints of product documentation and development efficiency. A typical conventional declaration sequence requires the developer to move the cursor to the declaration statements at the top of the module or file, enter the declaration, and then reposition the cursor in the original context. Often an initialization statement for the newly declared variable must also be created. The sequence is tedious because of its frequency, high keystroke cost, and strong impact on the developer's train of thought. Thought processes are frozen in stasis while the developer manually searches, searches, for the proper place to insert the declaration.

In contrast, declaration tools in the SEE environment preserve the developer's train of thought and display screen context during declaration activities by obtaining information from the developer via the Emacs minibuffer (message area) before inserting the declaration under the appropriate standard module header. The original display context is undisturbed. Furthermore, executable statements required to initialize newly declared variables are also collected and installed as part of the declaration process. This method preserves the developer's thought and cursor context, greatly reduces the number of keystrokes required to declare a variable, and encourages better documentation by prompting the developer for descriptive comments which are added to the declaration. Paraphrased Lisp code for such a tool is shown in Section A.3 on page 37.

Multiple or group declarations may be initiated with one command in the SEE environment. For example, all functions necessary for simple i/o processing (abbreviated as 'sio' in the SEE environment) may be declared with one command; the keystroke sequence "esc-x dcl sio" inserts ten documented PL/1 declarations for procedures which open, close, read, and write files, for input and output buffer declarations, and for input and output filenames and file variables.

At its best, a declaration sequence in the SEE environment only requires the developer to type, in context, the name to be declared, and then to invoke the declaration tool through its key binding. The tool extracts the word to the left of the cursor from the buffer, compares it against the interesting name list, and completes the declaration automatically where possible.

Where automatic declaration is not possible, the user is prompted for the necessary information before the declaration is inserted. The display screen and mental context of the developer remain unchanged during declaration operations.

The documentation of module modifications to produce a historical record of module development can often provide useful information to maintenance and testing personnel. In some cases, such as for the US National Computer Security Center B2 security rating [42], formal mechanisms for this activity are required. The SEE environment (informally) supports this activity with tools which are modelled after the declaration tools discussed above. The developer is prompted in the minubuffer for a history comment type (eg. Create, Design, Test) and a text description of the modification. Default values are offered for the date, author, and time cost of the activity. Sample history comments are shown in Section A.1 on page 32.

The detection and correction of common programming errors during implementation activities is supported by a simple code auditing facility which can be invoked on the current editor buffer (at any time, but usually just before a compilation attempt). The auditor corrects the errors which it understands, and reports errors which are beyond its ability to fix. The current auditor checks for run-time faults such as uninitialized parameters, variables, and pointers, as well as for documentation faults such as undocumented parameters.

For example, the current auditor obtains a list of parameter names from the Parameters section and then checks the Privileged Init section for statements which initialize each parameter. The developer is notified of any pointer parameters which are not initialized or referenced in the Privileged Init section. To ensure that a module's parameters are documented, the auditor identifies parameter declarations in the Parameters section which are not accompanied by a descriptive comment. The auditor prompts the user in the editor minibuffer for a descriptive comment and places the comment beside the appropriate parameter.

The production of well organized and easily read code is supported by text and code formatting tools built into the SEE environment. One such function sorts declaration statements in alphabetical order by variable name and spaces out task comments in a controlled fashion before passing the module to the operating system code formatter for more complete formatting. After external formatting is completed, the function loads the module back into the original editor buffer and leaves the cursor at (or near) its original position in the unformatted module. A convenient "clean up" tool combines the functions of the code auditor, formatter, and other "fixup" tools into a single conceptual function.

Module cloning activities often take place during the implementation phase when it becomes necessary to create a module which partially shares the structure or function of an existing module. Sometimes cloning operations are used to instantiate template functions. In either of the above cases, the fastest way to create a module B might be to clone a module A and modify it to fit the new situation.

Module cloning is implemented in the SEE environment as a two step process. During the first step, the cloning tools copy most of the contents of the old module A into the newly created module B. Executable code in module A is temporarily left behind. As part of the copying process, the tools perform a series of editing tasks often associated with cloning operations. In particular, the tools will add parameters to the module interface, enter new declaration statements, and perform string substitutions under the guidance of the developer. Once the automated editing tasks are completed, the tools place the cursor in the Function section of the new module so the developer can manually edit the new module. Changes are typically made to the Function, Subsystem Documentation, Notes, and History sections, as appropriate to the new situation.

The second step of the cloning operation deals with transporting executable code from module A to a module B. Once the documentation changes are completed, the algorithm of the new module B can be modified by editing the task comments copied from module A. Task comments are added, deleted, and modified until the new algorithm is adequately described. The second automated step of the cloning process is completed by a function which, for each task comment present in both A and B, copies the code associated with the task comment in A to the corresponding location in B. In this way only the useful code related to B's new algorithm is copied to the new module, and the developer is freed from handling unwanted code while designing the new algorithm in B. The final code, of course, must be inspected for correctness; the cloning operation can leave needed code behind or bring over too much, as the task comments dictate.

Finally, global editing of project files is supported by a tool which invokes an arbitrarily complex Lisp editing function on each file contained in the working directory. The tool sequentially reads each file in the directory into an editor buffer, invokes the indicated function, and examines the 'buffer-modified' flag after execution of the function has completed. If the file has been modified by the editing function, it is written out before the next file is loaded. Otherwise, the unmodified buffer and file are discarded. This tool is extremely useful to application functions which must examine or manipulate the project files as a group. In particular, since the editing function can perform an arbitrary number of tests before it decides to process the current buffer, the developer has the ability to write 'test-before-processing' functions which will selectively process only those modules which meet interesting selection criteria.

In the SEE environment this mechanism is used to invoke a design documentation extraction tool on each project source code module during the project document updating process. The extraction tool gathers design information from the module and stores it in another editor buffer for later addition to the project document. The mechanism is also used to apply the calling tree tool to source code files for calling tree report generation.

2.3 Support for Documentation Activities

The SEE environment supports documentation activities with two types of tools. The first type of tool supports the collection of documentation information during the normal course of development, whereas the second type of tool extracts and presents the information gathered by tools of the first type. An example of a collective tool in the first set is the tool which supports declaration activities, and an example of an extractive tool is the calling tree report generator.

The main operational model of documentation tools in the extractive class is based on the global file editing tool described above. An overseer function uses the global file editing tool to apply an extraction tool to a set of modules, and then processes and formats the results for final presentation.

The major documentation activity in the SEE environment, producing a current project document, is supported by a tool which oversees the construction of a full project document from components stored in the project directories. The tool locates or generates the needed document components, generates or modifies all date-specific information, initiates the processing of the document by a system word processor, and finally assembles the system output files into a unified whole. A current, comprehensive project document can thus be constructed in a few minutes. Since the tool extracts interesting documentation directly from the source code, all function, design, history, time cost, and algorithmic information in the project document is as current as the information contained in the source code.

In particular, this mechanism mitigates the industry problem of out of date design documentation. Since the interleaving of task comments and code makes detailed design information directly accessible to developers at the exact time when they modify the code, keeping the low level design information up to date is a very inexpensive and error free process. As a consequence, it reasonable to expect both that documentation (task comment) changes will actually be made during maintenance, and that the automated extraction process will propagate such changes to the project document.

2.4 Support for Management Activities

Two general types of project management support are provided by the SEE environment. The first kind of support is oriented toward the generation and maintenance of the project document, whereas the second is oriented toward the collection and analysis of project cost and size data as represented by history comment entries and module line counts. Since an overview of the project document facilities has already been presented above, the following sections only describe management supports which concern project metric data.

Project time cost data for interesting project phases is collected and recorded in the History section of the module with the aid of the modification history tool described above. Sample entries are shown in Section A.1 on page 32. At any later time, internal editor tools and external system tools may be used to extract and summarize this information into a simple chart of project activity by time cost. It is also possible to specify particular dates and modules to the tools, allowing developers to determine how much time has been charged against a set of modules by various development activities on any particular day.

Project size metrics in the form of module and source line counts are generated by tools which count lines in the project source code files. The current line counting tool generates counts for modules, prose (multi-line) comments, in-line comments, blank lines, code lines, and task comments. This data can be used to plot source code size against time to provide a visual representation of project code growth, or can be used to update software cost estimations made with line count estimation models such as COCOMO [8].

The value of these management oriented tools is that they provide an inexpensive and efficient means of collecting and analyzing project size and time cost data, and so can be regularly used by developers to monitor project performance. The usefulness of the output is, of course, directly dependent on the accuracy of the input entered by the developer. Metric data can be conveniently collected from the source code at the end of each working session by a batch file containing commands to process the information of interest. The time cost of such an analysis (measured in seconds) is relatively low compared to the value of the information gained.

Chapter 3

Experience with the SEE Environment

The concepts and tools in the SEE environment have been used on a variety of development projects. During its development, a partial SEE environment was used to develop an 11,000 line time management application program written in PL/1. After completion, the full environment was used to develop a 9000 line PL/1 simulation of the commodities futures market [15] and a portion of a commercial real-time X25 networking software product (C, 5000 lines). The development principles, standard module structure, and limited (microcomputer) versions of the environment were later used to develop a simple editor (Pascal, 5000 lines), a simple expert system (Pascal, 5000 lines), and a commercial real-time livestock feeding system (C, 5000 lines).

All developers involved with the projects voluntarily adopted the standard module format quickly, along with all of the declaration tools which were available in the (sometimes limited) environments in which they worked. The limited environments were hosted on microcomputers, and thus did not have the support of a true Lisp-based Emacs editor. Instead, simple tools for stubbing, formatting, declaring, line counting, time costing, and history comment entering were developed either as normal operating system tools, or as extensions to the public domain MicroEmacs editor. No project document tools were developed for use in the limited environments.

3.1 Problems With The SEE Environment

Experience with the SEE environment indicates that there is room for improvement in subsequent versions.

The current calling tree tool generates a true tree in which all subtrees and leaves are expanded, one module name per line. No module function comments or parameter lists appear in the output. A better tool would allow the user to mark subtrees which should not be expanded each time they appear in the tree. For example, the full expansion of subtrees which represent heavily used utility functions greatly reduces the utility of the output; their frequent appearance masks the true program architecture.

The standard module format should be modified to include a short (eg. 40-character) description of the module function. This short form could be printed out beside each function in the calling tree, significantly improving the value of the report in cases where the module function is not obvious from the name alone. Parameter lists printed beside the module name would also improve the utility of the output, although the combined length of indentation, module name, parameter list, and short descriptive comment would frequently exceed 80 columns. Optional parameter lists would facilitate viewing on an 80 column monitor.

Experience has shown that some developers avoid using the time costing tools because they are not convinced of the benefit of time cost data, or because they feel too much labor is involved. In practice, the labor required by the tools is inconsequential on any realistic scale, indicating that the underlying issue is possibly one of desire for performance analysis.

Another significant human factors issue is that users of the environment must relinquish old preferences in coding and module formatting styles in favor of the standard module format which supports the software tools in the environment. Most developers initially resisted the new format because of its apparent verbosity. However, their resistance usually disappeared once they had actually used the tools to improve their productivity and understood how the format supported the tools. It is interesting to note that theoretical discussion alone was almost never successful in this regard; skeptical developers were rarely convinced until they had actually used the tools.

In closing this section, it can be said that most of the tools and methods in the SEE environment have worked well. In particular, the utility of the standard module structure, declaration tools, and history comment tools surpassed initial expectations, and has led to their almost universal acceptance by developers who have actually used them. Other tools may have enjoyed higher usage rates had their operation been properly documented. (This is the one area in which the limited development resources were a significant factor.)

3.2 Extension of SEE Concepts To Other Environments

The portability of SEE concepts and tools across machine and language environments is primarily a result of the textual model of software structure used by the environment. As a consequence, modifying the SEE environment for use with a new programming language usually only requires simple changes in comment syntax.

Effective subsets of the full SEE environment can be implemented on other machines using different editor software, as illustrated by the partial environments created on personal computers using the MicroEmacs editor and external tools. Note that a Lisp-based editor is not required to support SEE concepts (MicroEmacs is written in C), and that many of the tools need not be editor-resident (only the declaration tools were made resident in MicroEmacs). In principle, it should be possible to implement all the tools and models in a non-Lisp, non-Emacs editor environment. However, such a complete editor-resident implementation has not been attempted as of this writing.

Upgrading an editor to use SEE concepts and tools does not affect its regular operation in any way; the editor remains as general as it was in its unmodified state. Moreover, an upgraded editor can be effectively used on projects in progress; new modules can be created in the SEE format without converting the rest of the application. If desired, existing software can be upgraded to the SEE standard format by adding an appropriate set of standard module headers and task comments, and by placing code lines under the appropriate task comment. However, the addition of task comments requires patience in cases where the original commenting style is not close to the standard SEE task comment style, and is a tedious chore in cases where the original module is undocumented.

3.3 Achievement of Design Goals

Three of the four major design objectives have been achieved and strongly demonstrated by the SEE environment. Production costs were clearly reduced by tools which automated many activities throughout the lifecycle, the level of product documentation was significantly increased at low cost by use of a standard module format and documentation tools, and the level of and access to project metric data was improved through simple and inexpensive time costing and project size metering mechanisms.

The fourth objective of improved maintenance productivity has not been demonstrated as strongly. Whereas the attainment of other objectives is easily understood through the environment's visible mechanisms of automation and report production, the environment only provides indirect evidence for improved maintenance productivity.

The objective of increased maintenance productivity is indirectly supported in that systems produced in the SEE environment have a relatively high level of well-organized internal documentation as compared to many other software products known to the author (several commercial operating systems, editors, networking products, graphics systems, business applications, and real time systems). The environment also supports the design objective with its capacity to generate current design documentation from products well into the maintenance phase of their lifecycle. Since accurate product documentation undeniably shortens the learning curve faced by new product maintainers, the environment provides strong, though indirect, evidence for increased maintenance productivity.

All five secondary design objectives were achieved and demonstrated by the environment.

The environment is practical and easy to learn. It has been used and accepted by a variety of developers on different machines, different projects, and in different programming languages. The transportability of the environment across machine environments and programming languages has been strongly demonstrated, as has the usefulness of a standard module structure which can tag and emphasize important conceptual information for humans and software tools. Many tools directly implement high level conceptual operations and thus make significant contributions to increased software development productivity.

3.4 Summary

The SEE environment consists of a practical, portable set of concepts and tools for supporting software development in a variety of editor and programming language environments.

Several important concepts have been introduced and implemented in the SEE environment. First, the work has introduced the concept of a standard module structure, and has illustrated the importance of such a structure in supporting the use of advanced development tools throughout the software lifecycle. In particular, the work has shown that a simple textual model of standard software structure is strong enough to support a wide variety of tools, yet flexible enough to be easily moved between programming languages and editing environments.

Second, the environment has introduced the concepts of representing the system design document in the form of documentation contained in compilable modules and of generating a conventional project document from information embedded in source code files. Assuming developers update interleaved task comments during maintenance (a reasonable assumption given their proximity to the changed code), the extraction method significantly reduces the problem of out of date design documentation.

Third, the environment has introduced a series of tools which demonstrate that it is possible to preserve the developer's train of thought and display screen context in many common situations. For example, SEE tools preserve the display screen context through all declaration operations, and support the developer's mental level of abstraction during stubbing and cloning operations by automating several detailed processes which would normally require the developer's manual attention.

Finally, the SEE environment has introduced and implemented the concept of using the source code as a vehicle for the collection and analysis of project time cost data.

In conclusion, this paper has described and presented the SEE environment in the context of the four project lifecycle activities of design, implementation, documentation, and project management, and has evaluated the portability of the SEE environment on the basis of its partial reimplementation in, and application to, several modern programming languages. The utility of the environment has been evaluated on the basis of several real-life development projects.

The SEE environment was found to significantly improve productivity on a broad, lifecycle-oriented basis in the small projects on which it was used. Several SEE concepts and tools enjoyed near universal acceptance among the developers who tried them.

3.5 Future Research

Future environments which attempt to support a significant portion of the software lifecycle will possibly use increasingly stringent user, software, and project models in order to better support development activities. A key concept is that the support level which can be offered by software tools is directly related to the strength of the supported model. The complement of this assertion is that to be acceptable, increasingly stringent models *must* have increasingly useful tool support. Enforcing a more stringent model without corresponding tool support will cost more, not less, than using no model at all.

Development environments which make no assumptions about programming method or style are generally bound to use the grammar of the edited language as their (minimal) model. This tends to limit the support which can be offered by software tools, as grammatical models have difficulty representing lifecycle activities. Conversely, structural and procedural models which go beyond grammatical models to encompass lifecycle-oriented information offer many more opportunities for software tool support.

3.6 Acknowledgements

The author would like to thank the Department of Academic Computing Services and the Advanced Computing Technology Center of the University of Calgary for providing the computing resources used in this project, and Andrew Ginter for many valuable comments on draft versions of this paper.

Bibliography

[1] Allison, Lloyd. *Syntax Directed Program Editing.* Software Practice and Experience, Vol 13(5) 1983. pp. 453-465.

[2] Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* IEEE T. SE-11 11 Nov 1985, pp. 1257-1268.

[3] Balzer, Robert. *Editorial: Program Transformations.* IEEE TSE SE-7 (1), Jan 1981. p 1.

[4] Barr, A. and Feigenbaum, E, Editors. *Handbook of Artificial Intelligence, Vol 2.* William Kaufman, Los Altos, Calif. ISBN 0-86576-006-3, Chapter X, pp 297-379.

[5] Barstow, David R., Shrobe, Howard E., Sandewall, Erik. *Interactive Programming Environments.* McGraw Hill, New York, 1984. ISBN 0-07-003885-6.

[6] Barstow, David R., and Kant, Elaine. *The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis.* IEEE T. SE-7(4) Sept 1981, pp. 458-471.

[7] Barstow, David R. *Domain Specific Automatic Programming.* IEEE T. SE-11 11 Nov 1985, pp. 1321-1336.

[8] Boehm, Barry W. *Software Engineering Economics.* Prentice Hall, N.J. 1981. ISBN 0-13-822122-7.

[9] Boehm, Barry W. *Improving Software Productivity.* IEEE Computer, September 1987. pp. 43-57.

[10] Budinsky, Frank J. *SRE: A Syntax Recognizing Editor.* Software Practice and Experience, Vol 15(5) 1985. pp. 489-497.

[11] Cheatham,T., Townley, J., and Holloway, G. *A System for Program Refinement.* IEEE Fourth International Conference on Software Engineering, Munich, Germany. September 1979. pp. 53-62.

[12] Donzeau-Gouge, V., Huet G., Kahn G., Lang B. *Programming Environments Based On Structured Editors: The MENTOR Experience.* Interactive Programming Environments, pp. 128-140. McGraw Hill, New York, 1984. ISBN 0-07-003885-6

[13] Feiler, Peter H. *A Language Oriented Interactive Programming Environment Based on Compilation Technology.* PhD Dissertation, Carnegie-Mellon University. May 1982.

[14] Fickas, Stephen F. *Automating the Transformational Development of Software.* IEEE T. SE-11 11 Nov 1985, pp. 1268-1277.

[15] Gillis, K. *Incorporating Interest Rate, Exchange Rate and Barley Futures Into the Hedging Strategies of an Alberta Feedlot Beef Producer.* MA Thesis, University of Calgary. 1986.

[16] Habermann, A. Nico, Notkin, David S. *Gandalf: Software Development Environments.* IEEE TSE Vol SE-12 No. 12, Dec 1986, pp. 1117-1127.

[17] Hansen, Wilfred J. *Creation of Hierarchic Text With A Computer Display.* PhD Dissertation, Stanford University. June 1971.

[18] Hazel, Philip. *Development of the ZED Editor.* Software Practice and Experience, Vol 10(1) 1980. pp. 57-76.

[19] Kaisler, Stephen H. *Interlisp: The Language and Its Use.* John Wiley, New York, 1986. ISBN 0-471-81644-2.

[20] Medina-Mora, Raul. *Syntax Directed Editing: Towards Integrated Programming Environments.* PhD Dissertation, Carnegie-Mellon University. March 1982.

[21] Medina-Mora, Raul., and Notkin, David S. *ALOE Users' and Implementors' Guide.* CMU-CS-81-145, Carnegie-Mellon University. November 1981.

[22] *Multics Emacs Text Editor User's Guide.* Honeywell Inc., CH27-00. December 1983.

[23] *Multics Emacs Extension Writer's Guide.* Honeywell Inc., CJ52-01. July 1982.

[24] Notkin, David S. *Interactive Structure Oriented Computing.* PhD Dissertation, Carnegie-Mellon University. 1984.

[25] Peck, J.E.L. *Chef: A Versatile, Portable Text Editor.* Software Practice and Experience, Vol 11(5) 1981. pp. 467-477.

[26] Reps, Thomas W. *Generating Language Based Environments.* PhD Dissertation, Cornell University. 1983.

[27] Reps, T. and Teitelbaum, T. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* Communications of the ACM, 24:9, September 1981. pp. 563-573.

[28] Reps, T., Teitelbaum, T., and Horowitz, Susan. *The Why and Wherefore of the Cornell Program Synthesizer.* Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981. pp. 8-16

[29] Rich, C. and Shrobe, H. *Initial Report on a Lisp Programmer's Apprentice.* IEEE TSE SE-4 No. 6, Nov 1978, pp. 456-467.

[30] Shapiro, E., Collins, Greg., Johnson, Lewis., and Ruttenberg, John. *PASES: A Programming Environment for Pascal.* SIGPLAN Notices, Vol 16(8), August 1981. pp. 50-57

[31] Stallman, Richard M. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor.* Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981. pp. 147-156. Reprinted in Interactive Programming Environments.

[32] Stallman, Richard M. *GNU Emacs Manual, Sixth Edition, Version 18.* Free Software Foundation, Cambridge, Mass. March 1987.

[33] *Symbolics Text Editing and Processing Manual.* Symbolics Inc., Concord Massachusetts. July 1986.

[34] Teitelman, Warren., and Masinter, Larry. *The Interlisp Programming Environment.* IEEE Computer, Volume 14(4), April 1981. pp. 25-34

[35] Walker, Janet H., Moon, David A., Weinreb, Daniel L., and McMahon, Mike. *The Symbolics Genera Programming Environment.* IEEE Software, November 1987. pp 36-45.

[36] Waters, Richard C. *The Programmer's Apprentice: Knowledge Based Program Editing.* IEEE TSE SE-8 No. 1, Jan 1982, pp. 1-12.

[37] Waters, Richard C. *The Programmer's Apprentice: A Session With KBEmacs.* IEEE TSE SE-11 No. 11, Nov 1985, pp. 1296-1320.

[38] Weinreb, D., and Moon, D. *The Lisp Machine Manual.* MIT Artificial Intelligence Laboratory, 1981.

[39] Wilcox, R.R., Davis, A.M., Tindall, M.H. *Design and Implementation of a Table Driven, Interactive Diagnostic Programming System.* CACM 19(11), 1976. pp. 609-616.

[40] Wood, S.R. *Z: The 95% Program Editor.* Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation. Portland, Oregon. June 1981. pp. 1-7.

[41] Yonke, Martin D. *A Knowledgeable, Language-Independent System for Program Construction and Modification.* USC ISI/RR-75-42, October 1975.

[42] *Department of Defense Trusted Computer System Evaluation Criteria.* CSC-STD-001-83, DOD Computer Security Center, Ft. Meade, MD., August 15,1983.

Appendix A

Examples

A.1 A PL/1 Module In Standard Format

```
bc: proc (p_input, p_output) returns (bit (1));

/* FUNCTION

   This program oversees the determination of the biconnected components
   in an undirected, unconnected graph.
*/

/* SUBSYSTEM DOCUMENTATION

   LIMITATIONS

   Maximum vertex and edge limitations are set in the main data structure
   file which declares the command_environment structure.  Please refer
   to that section of the code for current limit values.

   <other program documentation would go here>
*/

/* NOTES

   This module shows the standard module structure in a PL/1
   context.  Code has been removed to facilitate publication.
*/

/* HISTORY
   Test___ 07 mar 87 (0.20) kj: test with isolated nodes
   Test___ 07 mar 87 (0.20) kj: ok if initial node is a bicon component
   Integ__ 07 mar 87 (2.00) kj: forgot to init stack pointer
```

```
   Test___ 07 mar 87 (0.50) kj: ok on connected Sedgewick graph
   Doc____ 23 feb 87 (1.00) kj: algorithm, limitations, error msg doc
   Incr___ 23 feb 87 (0.30) kj: output formatting
   Design_ 22 feb 87 (0.10) kj:
   Created 21 feb 87 (0.10) KJameson: with stub tool
*/

/* PARAMETERS */
      dcl           p_input        fixed bin;     /* an input parameter */
      dcl           p_output       fixed bin;     /* an output parameter */

/* SYSTEM DEPENDENCIES */

/* EXTERNAL DEPENDENCIES */
      dcl           build_adjlist  entry (ptr, ptr) returns (bit (1));
      dcl           visit          entry (fixed bin) returns (fixed bin);
      dcl           write_out      entry (char (*) var);

/* MISC */
      dcl           FAILS          bit (1) init ("0"b);
      dcl           WINS           bit (1) init ("1"b);
      dcl           i              fixed bin;     /* loop variable */
      dcl           temp           fixed bin;     /* dummy variable */
      dcl           testfile       ptr;           /* contains test suite */
      dcl           testmode       fixed bin;     /* set = 1 for testing */

/* PRIVILEGED INIT */
      p_output = 0;                               /* init output parameter */

/* INIT STORAGE */
      testmode = 0;
      testfile = null ();

/* : allocate and initialize the main data structure */
      allocate cmd_env set(cmd_env_ptr);         /* alloc based storage */
      cmd_env.max_vertices = MAX_VERTICES;
      cmd_env.vertex_id = 0;                     /* depth first search id */
      cmd_env.n_edges = 0;
      cmd_env.n_vertices = 0;

/* : parse and load the command line arguments */
      if bc_parse_args (cmd_env_ptr, testfile, testmode) = FAILS then do;
         call write_out ("Unable to parse command line arguments.");
         return (FAILS);                         /* quit the program */
      end;

/* : with protest, open the input file */
```

```
      if open_i_file (cmd_env.infile, cmd_env.infilename) = FAILS then do;
         call write_out ("Unable to open input file. " ||
            cmd_env.infilename);
         return (FAILS);
      end;

/* : build the adjacency list */
      if build_adjlist (cmd_env_ptr, cmd_env.infile) = FAILS then do;
         call write_out ("Unable to build adjacency list from file:  " ||
            cmd_env.infilename);
         return (FAILS);
      end;

/* : write output header */
      call write_out ("BICONNECTED COMPONENTS (One per line)");

/* : set values for all vertices to zero */
      do i = 1 to cmd_env.n_vertices;
         cmd_env.dfs_order (i) = 0;
         cmd_env.father (i) = 0;
      end;

/* : perform the depth first search on each vertex */
      do i = 1 to cmd_env.n_vertices;
         if cmd_env.dfs_order (i) = 0 then       /* if not visited, */
            temp = visit (i);                    /* visit and print */
      end;

/* : free the storage */
      free cmd_env;

/* : return success */
      return (WINS);

/* INTERNAL PROCEDURES */

/* DATA DEPENDENCIES */
%include bc_cmd_env;                             /* main data structure */
   end;
```

A.2 Standard Module Headers

- **Function** Tags and bounds a textual description of module function.
- **Parameters** Tags and collects module parameter declarations.
- **Subsystem Documentation** Tags and bounds a text description of the major facilities provided by the interface of the module containing the documentation. This header is only followed by text in modules which implement a major subsystem of a larger system, and forms part of a documentation layer at the subsystem abstraction level.
- **Notes** Tags and bounds a text description of information peculiar to the module which contains the header. For example, interface assumptions and operational caveats are placed in this section. In conjunction with the Function section, this section forms part of a documentation layer at the module abstraction level.
- **History** Tags and bounds a text description of the modification history of the module and the time costs charged against the module during various lifecycle phases.
- **System Dependencies** Tags and collects operating system dependencies in the form of declarations of system subroutine interfaces.
- **External Dependencies** Tags and collects external dependencies in the form of declarations of external subroutine interfaces.
- **Data Dependencies** Tags and collects data template dependencies in the form of type and structure declarations contained in include files.
- **Misc** Tags and collects all local variables declared by the module.
- **Privileged Init** Tags and collects all executable statements which initialize parameters or variables which are used in initialization statements in the 'Init Storage' section.
- **Init Storage** Tags and collects regular variable initialization statements.
- **Internal Procedures** Tags and collects internal procedure declarations in languages which support them.

- **Task Definitions** Special comments which document the algorithms and actions of a module. These comments are first entered during the detailed design phase, and are updated during maintenance activities. Each comment is directly and functionally related to the executable code statements which immediately follow it. Task comments are recognized by the colon (:) which follows the comment-start character sequence. ('/* :' for PL/1, '(* :' for Pascal, '– :' for ADA, etc).

A.3 A Lisp Function for Entering Declaration Statements

```
;;;  This paraphrased Lisp function inserts a formatted PL/1 declaration
;;;  statement below a standard header in a PL/1 module.  The caller of
;;;  this function supplies a name to be declared and a header variable.
;;;
;;;  The save-excursion Lisp form preserves the current cursor
;;;  position for the duration of execution of all forms inside
;;;  the save-excursion form.  The cursor position is restored as
;;;  execution exits the save-excursion form.
;;;
;;;  This paraphrased function does not handle variable initializations.

(defun dcl (name hdr-abbrev)
       (save-excursion                            ;restore cursor afterwards
         (go-to-beginning-of-buffer)              ;start at top of buffer
         (let ((hdr (expand-abbrev hdr-abbrev)))  ;generate search string

;;;  Search for the header from the top of the buffer, and if
;;;  found, open up a blank line just below the header.  Indent
;;;  an appropriate amount, and insert a formatted declaration
;;;  based on information obtained from the user.

              (cond ((forward-search hdr)
                     (search-for-blank-line)      ;go to end of dcls
                     (open-space)                 ;open up a blank line
                     (indent-relative)
                     (insert-string
                       (catenate                  ;build formatted dcl
                         "dcl " (get_pname name)
                         SPACE
                         (expand-dcl-abbrevs      ;attrs can be abbreviated,
                           (minibuffer-response   ;eg. ‘fb’ for ‘fixed bin’.
                             (catenate
                                 "Dcl attributes for "
                                 (get_pname name) "? ") NL))
                         SPACE
                         (minibuffer-response     ;get descriptive comment
                           (catenate "Comment? " NL))) SEMI))

;;;  Display an error message and fail safely if no header is found.
                    (t (display-error-noabort hdr " header not found.")
                       (ring-tty-bell)))
              )))                                 ;replace cursor afterwards
```

Keywords

Software engineering, software development, programming environments, module structure, lifecycle, programming style.

Author's Address

Correspondence regarding this manuscript should be addressed to

Kevin Jameson
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, Alberta
Canada T2N 1N4

(403)/220-6015 (UofC Cpsc main office)