OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

ARTIMIS: A MODULE INDEXING AND SOURCE PROGRAM READING AND UNDERSTANDING ENVIRONMENT

Abbas Birjandi

T.G. Lewis

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

(503) 754-3273

TR 86-10-3

Artimis:A Module Indexing and Source Program Reading And Understanding Environment

Abbas Birjandi

T.G. Lewis

Department of Computer Science

Oregon State University

Corvallis, Oregon 97331

(503) 754-3273

Abstract:

Artimis is part of an environment for software reuse consisting of two logically independent portions, 1) the indexing and retreival facility called, *GrabBag*, for storage and subsequent retreival of reusable modules, and 2) a set of tools called *Browsers*, which aid reading and understanding of source programs. GrabBag creates a highly simple and friendly interface for retreival of viable candidates for reuse. Browser's tool set, The Module Interconnection Graph Builder, Procedure Call Graph Builder, and Module Abstractor create different levels of abstraction to help a programmer understand a source program.

Keywords: Programming environment, program transformation, source code mutation, code fragments, code selection, program understanding, program reading, program maintenance.

1 Introduction

The reuse of existing software is seen as a measure of curtailing the high cost of software. The benefits of reusing existing software are: 1) reduction in the cost and development time to produce a new program or system of programs, and 2) an increase in the ease of maintenance and enhancement of existing software systems [Che83]. To reuse existing software one should know what existing software is available and how it can be used in relation to the task at hand.

Artimis is part of an environment for reusing software [Bir86] which provides a programmers database called *GrabBag* [San86] and a set of understandability and abstraction tools collectively referred to as *Browsers*. GrabBage provides a convenient way of locating a module and related documents called attributes. A module is an independent unit of code. Module attributes are known resources of a module such as a documentation file and an interface definition file.

Although other source languages might be used with Artimis, Modula-2 [Wir83] is used as the source language. In Modula-2 , a module has two parts: 1) a definition part which defines the visibility of constants, types, variables, and procedures of the module which can be accessed by other modules and, 2) an implementation part that encapsulates the actual implementation detail of the module. Artimis is written in C and runs on the Macintosh personal computer.

Reusability

In [Ker83] reusability is defined as anyway in which previously written software can be used for a new purpose or to avoid writing new software. This definition covers representation of software at both object code and source code level. However,

reuse of source code in contrast to object code has the advantage of 1) adapting the interface as well as implementation part of a module to a new interface specification, 2) providing an opportunity to tune, optimize, and eliminate unnecessary code, and 3) providing readable code so that a programmer's knowledge of the reusable module is increased.

This allows the possibility of:

1. Source code reuse/replication by reuse of part or all of existing source code or its data structure,
2. Detailed algorithm reuse by reuse of source code from existing programs as an example of how to do a new program,
3. Large-scale structural reuse by selecting and adapting program design,
4. Maintainability/enhanceability by increasing the effectiveness of programmers by enabling them to study programs with the aid of understandability tools,
5. Portability by facilitating the reuse of software across a wide range of hosts, and
6. Optimization by enabling tuning of generated source code.

Reusability Life Cycle Vs. Traditional Life Cycle

When reusable components are used to build a new software system, the traditional software life cycle is altered. Table 1 shows the difference between traditional software life cycle and reusability life cycle. The additional phases in the reusability life cycle indicate how a designer uses existing components rather than implement everything from the beginning.

Traditional Life Cycle	Reusable Life Cycle	Artimis Support
Problem Definition	Problem Definition	None
Requirement Analysis	Requirement Analysis	None
System Design Specification	Find and reuse similar System Design Specification	None
Detailed Design Specification	Find and reuse similar Detailed Design	GrabBag,Browsers
Implementation	Find and reuse existing routines from object code library	None
	Find and reuse (modified) source code from previous systems	GrabBag,Browsers
	Produce Glue Code	None
Testing	Testing	Some help by Browsers
System Integration	System Integration	Some help by Browsers
Maintenance	Reuse of original product	GrabBag, Browsers, None

Table 1: Reusability Life Cycle Stages vs. Traditional Life Cycle

Maintenance may be considered as reusing the original product [Fre83]. In maintenance, problem specification is usually better defined and the product does not have to be located [Fre83]. Problem definition is the phase during which the problem to be solved is formalized as a set of needs; requirement analysis is the process of studying user needs to arrive at a definition of system software requirements; system design specification is the period of time during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirement; detailed design specification is the period of time during which the design of system or a system component is documented; typical contents include system or component algorithms, control logic, data structures,

data set-use information, input/output formats, and interface description; implementation is the period of time during which a software product is created from design documentation and debugged; testing is the period of time during which the components of a software product are evaluated and integrated to determine whether or not requirements have been satisfied; system integration is the period of time during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required; maintenance is the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements

A component is a basic part of a system or program; an interface is a shared boundary to interact or communicate with another system component [Sta83]. Glue code is the minimal extra code that may be needed to bring the reused modules together.

2 Artimis System Components

Artimis has two logically separate components: 1) GrabBag, for adding, deleting and searching for a module and its different attributes, and 2) Browsers, to aid the programmer in reading, inspecting, and understanding the code retrieved from GrabBag (or any other source of program modules).

2.1 GrabBag

In order to reuse existing software there must be a convenient way of locating the viable candidates for reuse. GrabBag is an indexing and retreival system for

finding available modules and their attributes in a Programmers DataBase, (PDB). PDB contains a set of *option lists* that allows the searcher to successively refine the description of the code he is looking for. Option lists are sets of *categories.* Categories are text prompts entered by the PDB builder and are used to lead the searcher to a desired module through a *search path.* A search path is a series of individual categories that lead to a module. Since there are many different ways to describe a module, there could be several different search paths to each module and it's attributes. Figure 1 shows a typical hierarchy of components of a PDB and possible search paths to individual module attributes. In Figure 1 there are

Figure 1: GrabBag Internal Data Model

two levels of option lists. The **Root** is a pseudo starting point of a PDB. The first option list, A, has three categories: B, C, D. Option lists B, C, and D point to some attribute files.

GrabBag Operations

GrabBag supports:

- Creation of new PDBs,
- Searching for a module and its attributes,
- Adding new Categories,
- Addition and deletion of search paths among the categories, and between categories and attribute files,
- Addition and deletion of attribute files and references to them.

The following section is a walk through and explanation of: 1) searching through a PDB to locate a category, leading to a module and its attributes, 2) adding an attribute to a module stored under an existing category, and 3) establishing a search path to a module attribute. We assume that the PDB is already selected and opened.

Searching

Once the PDB is opened GrabBag creates two windows: 1) for the display of search paths being selected in the course of searching, and 2) for display of available categories and attribute files for selection. At the beginning the title in the second window is the name of the currently opened PDB,*UTILITIES DATA BASE*, see Figure 2.

Figure 2: A Category and its Subcategories in a PDB

Selection of a subcategory is made by pointing to the title of the subcategory and clicking the mouse twice. In Figure 3 subcategory *SEARCH ROUTINES* is selected. Each time a selection is made the title of the currently selected category is updated and moved to the *Search Path* window. One can continue navigation along selected

Figure 3: Search Path and Category Windows after a selection

paths to narrow down choices until the desired element is found. Notice if one decides to reverse a selection and backtrack to some earlier category it is only necessary to select the category name from the *Search Path* window. Selection of a category from the *Search Path* window will always make the category the current category. This process can be repeated as long as categories exist. Reusable modules and their attribute files are stored at the end of each Search Path. Once a Search

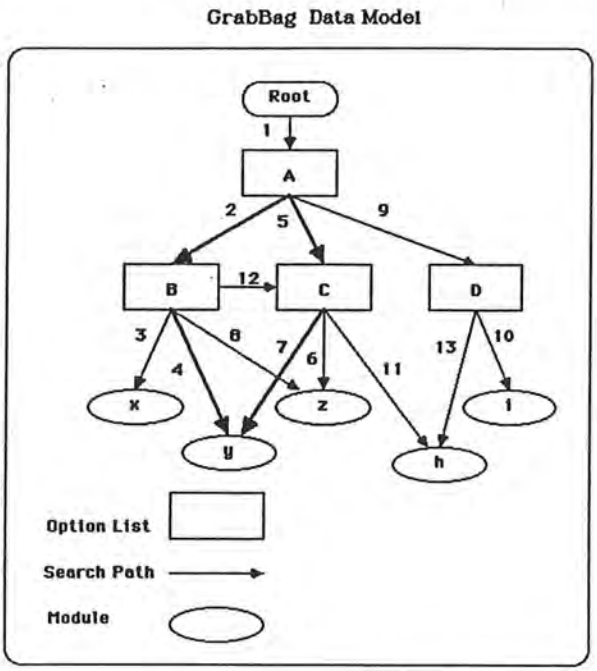

Fig. 1

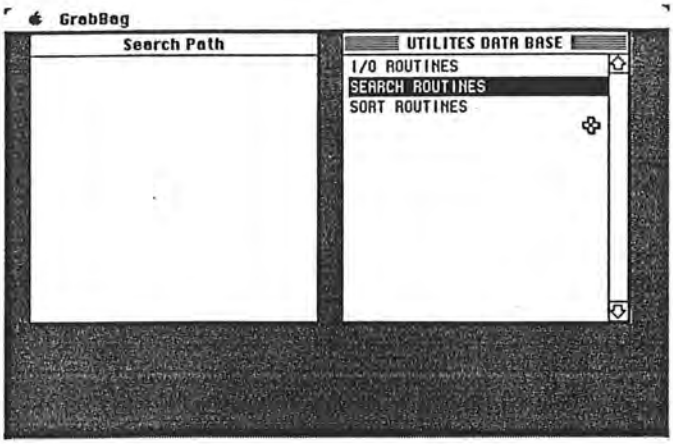

Figure 2

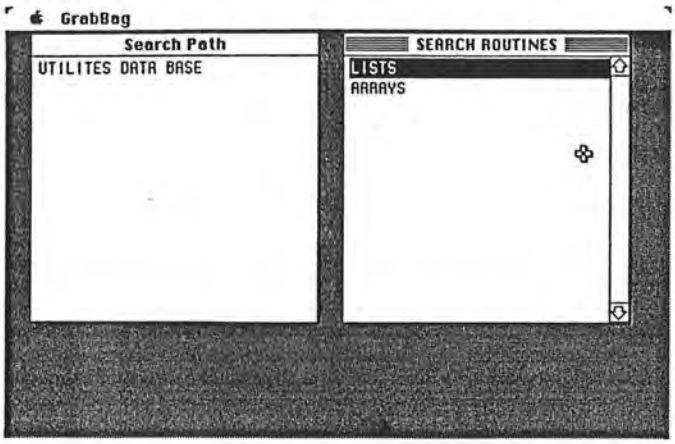

Figure 3

Path is exhausted, attribute file names are displayed in the left–side window as the members of the latest category. In addition, a selection dialog showing available operations is displayed as shown in Figure 4. The *Search Path* leading to attributes for Binary Search and the dialog box containing the available operations is shown in Figure 4. Selecting *Edit* will create an edit window and display the contents of the

Figure 4: Attribute File Selection for Category

selected attribute file (List Binary S.Document) for editing or any other operations that are supported by the editor. Selection of *Copy to* will make a duplicate copy of the file; *Delete* deletes the selected attribute file from the category that it belongs to; and selecting *Cancel* removes the dialog so the search can be resumed.

Adding New Categories

To add a new category to an option list, one first locates the desired option list (the process of locating is the same as searching for a category). Once the desired category is located, *Add Category* must be selected from the GrabBag menu shown in Figure 5. When *Add Category* is selected a dialog showing the category and number

Figure 5: Menu item for Add Category

of subcategories already under it is displayed see Figure 6. The new category title is entered in the *New Subcategory* field. Selection of *Add and Quit* will add the new category as a new subcategory and quits. If there is more than one subcategory to be added, select *Keep Adding* which does the operation of adding and keeps the dialog box for further addition. Notice that the current number of subcategories under a category is also displayed. Selection of *Quit* terminates the process of adding new subcategories and returns to the category and Search Path windows.

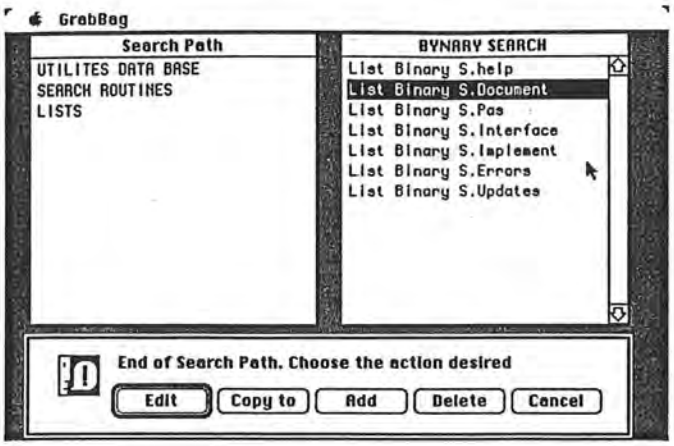

Figure 4

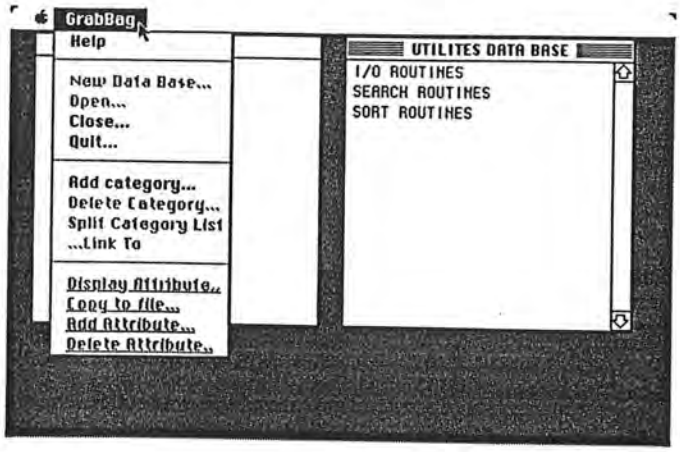

Figure 5

Figure 6: Dialog Box for Adding a New category

Adding A Module Attribute

Adding a module attribute follows the same procedure for narrowing down the category by selection of categories and subcategories. Once the desired category is located the selection of *Add Attribute* from the menu will display the name of the attribute files that can be added, see Figure 7, and 8. Selection of any of these file attributes will add them to the list of available attributes of the selected module.

Figure 7: Menu Item Add Attribute Selection

Figure 8: Selection Attribute File For Addition

Module Attribute Deletion

To delete a module, locate the subcategory which contains the module attribute to be deleted, then select *Delete Attribute* from the menu, see Figure 9. A dialog box will appear as shown in Figure 10 which tells the number of references made to the attribute file. The number of references to the specific attribute file is always shown in order to give some clue to how many active references are to that specific attribute file. One can choose to delete only the reference to the attribute file from the most recent category, or choose to delete all the references to the attribute file. In either case, the actual attribute file may be removed from the PDB by selecting the *Delete Attribute File,too* option

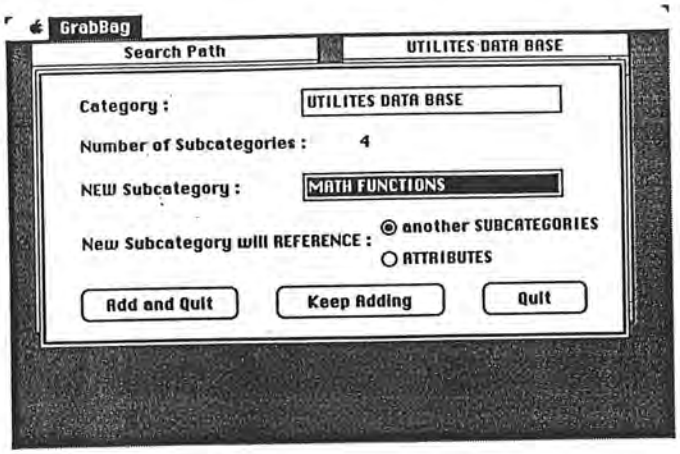

Figure 6

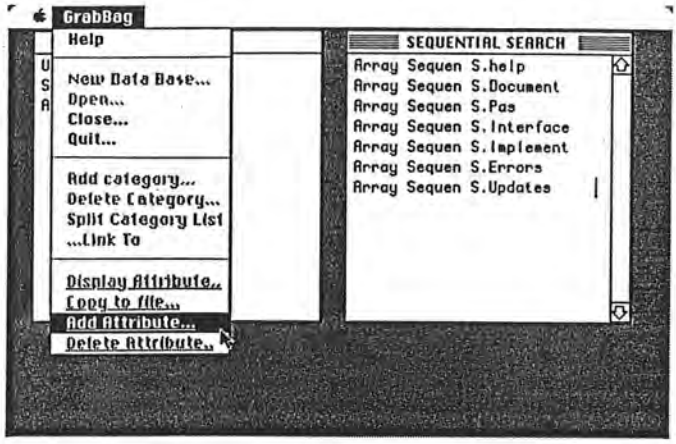

Figure 7

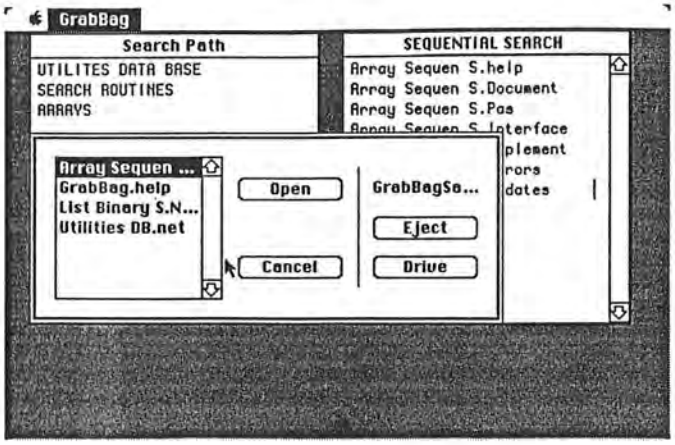

Figure 8

Figure 9: Attribute Deletion Menu Item

Figure 10: Selection Dialog For Deleting an Attribute

Linking Search Paths Among Categories and Module Attribute Files

To establish a link between a category and another category or a module attribute, the title of the *From* category must be selected from the *Search Path* window. Then the *Link From...* item must be selected from the menu to mark the category as the origin of the link, see Figures 11, and 12. Next, the module attribute or the

Figure 11: Selection of Category as the origin of the Link

category to which the link should point must be selected. Choosing the *Link To* from the menu specifies the destination of the link. The dialog shown in Figure 13 will be displayed showing what is linked to what, confirming the action. If the user decides to establish the link the *Ok* button should be pushed, otherwise the *Cancel* button should be selected.

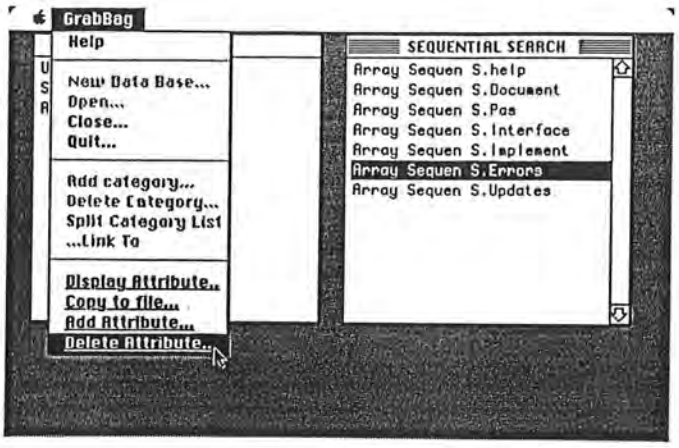

Figure 9

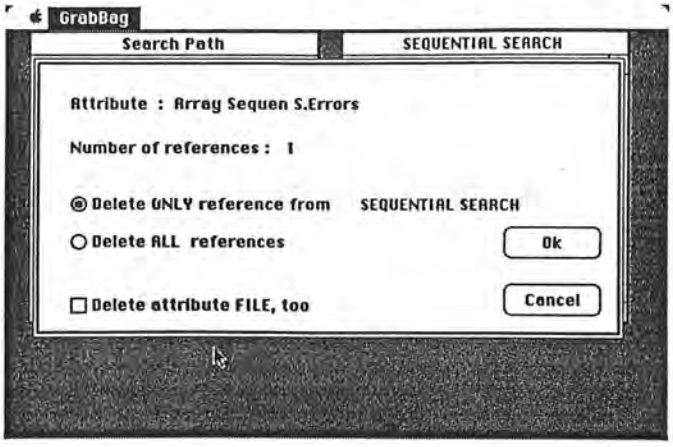

Figure 10

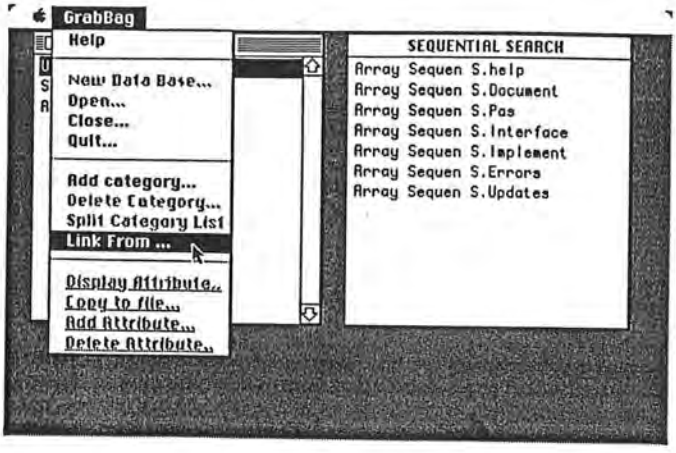

Figure 11

Figure 12: Setting the Link Origin

Figure 13: Dialog For Link Conformation

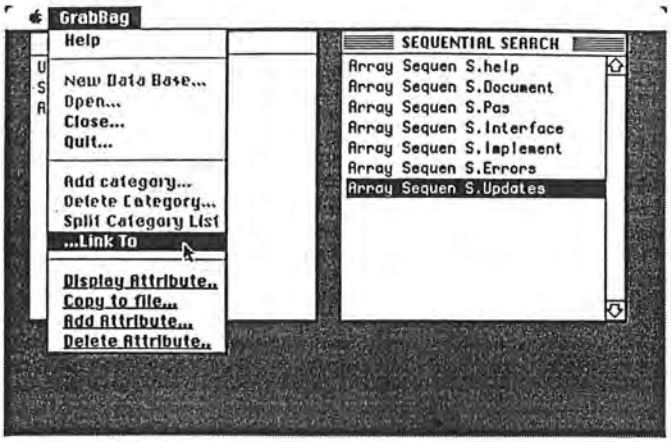

Figure 12

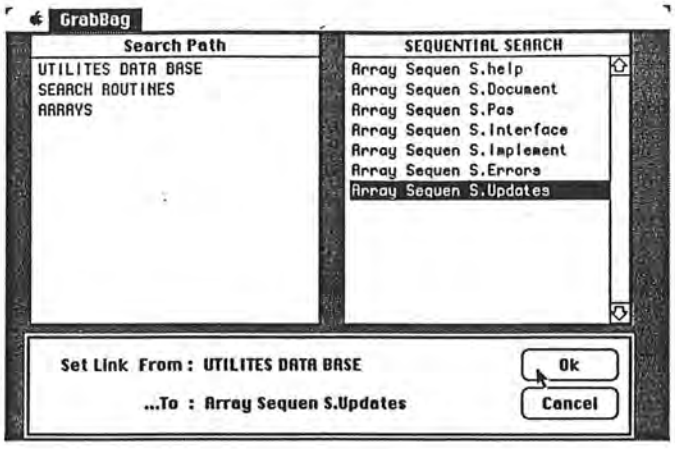

Figure 13

2.2 Browsers

Program Understanding

Browsers are tools to aid in reading and understanding program modules, a module, or parts of a module. Browsers assist the programmer in the process of mental transformation of a system of modules, a module, or parts of a module into an abstraction that summarizes the possible outcomes of the entity under consideration, irrespective of its' internal control structure and data operations.

Recent research in text comprehension [Bar32,SA77,Gra81,BBT79] has shown that schemas can facilitate the processing and storage of information by providing background knowledge or context.

> *Schemas are generic knowledge structures that guide the comprehender's interpretations, inference, expectations, and attention when passages are comprehended [Gra81]*

There is some empirical evidence [Shn76,Ade81,MRRH81] that programmers use schemas in the comprehension of computer programs. Information about the problem, *what it is* , the subgoals necessary to resolve the final goal, the method employed to solve the subgoals, *how it is done* , the level of expertise of the problem solver, etc. can be derived from program text [SE83,SEB82]. Program fragments and data structures can be thought of as schemas and knowledge structures. A program fragment is a piece of source code representing the stereotypic action sequence in programs. Program fragments are different from subroutines. Program fragments are open pieces of source code that are meant to be modified or tuned to the particular task at hand whereas subroutines are purposedly closed entities [SE83]. For example a *while* loop in a sort routine can be considered as a Loop fragment.

For a program to be reused one should know *what it is* and *how it works*. In fact understanding a source program is the basis for: 1) modifying and validating programs written by others, 2) selecting and adapting program design, 3) verifying the correctness of programs, and 4) becoming more effective through study of programs written by others [LMW79].

Abstraction in Reading and Understanding a Module

The object of reading a program or program part is to recognize directly what it does all in one thought, or to mentally transform it into an abstraction that summarizes the possible outcomes of the program under construction irrespective of its internal control structure and data operations. Thus one can regard program reading as primarily a search for suitable abstraction [LMW79]

In [LMW79] it is shown that a program fragment is an ideal component for abstraction. A compound program of any size can be read and understood by reading and understanding its hierarchy of fragments and their abstraction. Artimis uses the idea of *stepwise abstraction* in producing an abstracted version of a module or parts of a module. The process of stepwise abstraction starts at the most detailed level, and replaces each fragment by its equivalent abstraction. Stepwise abstraction is the inverse of stepwise refinement.

2.3 Program Understanding Paradigm

The exact approach and steps taken in reading and understanding a program source depends on the level of expertise of the programmer, clarity and readability of the source code, and availability of documentation. The most common steps typically

taken to understand a program are:

1. Build a picture of the system structure, exposing the hierarchy of interconnecting modules,
2. Examine the interface information to understand the nature and type of information exchanged among the components communicating with each other (e.g. procedures, functions, modules),
3. Start from the main program and trace the execution of the program,
4. Abstract and highlight the program fragments that are crucial to the operation of the program,
5. Comment the highlights and make notes on their operation for later use,
6. Repeat this process until the mystery is solved.

The following is the tool set which implements the steps outlined above in the Browsers of Artimis.

Module Interconnection Graph Builder

The Module Interconnection Graph Builder provides a graphical display of the hierarchical structure of a program containing one or more modules. The graphical display shows, 1) the overall program structure and placement of modules, 2) accessibility of the resources of each module from other modules, and 3) the interconnectivity (or disconnectivity) of modules. Figure 14 displays the Module Interconnection Graph of a set of modules.

The Module interconnection graph is the first order of fragmentation in program understanding. It provides a global view of the modules (fragments) that the program is build around. For example, Figure 14 (MODULES window) shows that module MOD1 has direct access to the resources (variables, procedure definitions, constants, etc.) of MOD2, and MOD3, and possibly has indirect access to the resources of MOD4, and MOD5. In turn MOD2 uses some of the resources defined in MOD4, and MOD5. These information can be used to trace the data and control flow of the module.

The module interconnection graph is also useful in formulating the dependency preserving sequence for correct compilation of the modules(MAKE). For example, in Figure 14 MOD4, and MOD5 should be compiled prior to MOD2 in order to preserve the correct compilation sequence.

Figure 14: A Sample Module Interconnection and Procedure Call Graph

2.3.1 Procedure Call Graph Builder

The Procedure Call Graph Builder shows the subprogram invocations found within a single module. The procedure call graph is the second order of fragmentation in abstraction of a module for readability and understanding purposes. The procedure call graph reveals the textual nested organization of a module [CWW80] that can be used to derive information related to the visibility and scope of entities within a module. It provides an abstract view of the control and data flow among the subprograms. For example Figure 14 (PROCEDURES window) shows call graph of MOD3 in which procedure *HEYYOU* and *MAC* are called from within procedure *FF*. And *FF* is called from within body of MOD3.

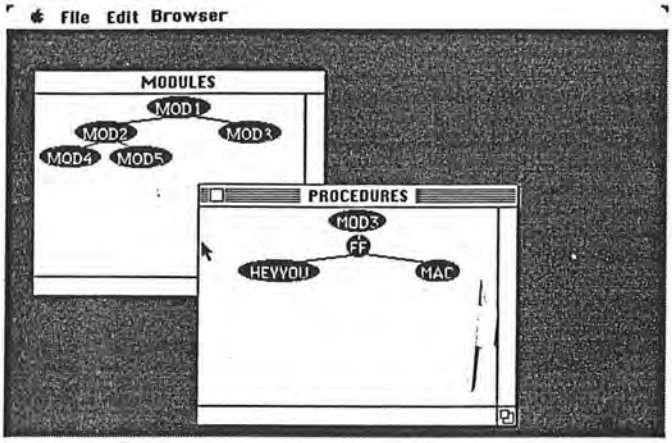
File Edit Browser
MODULES
MOD1
MOD2
MOD3
MOD4
MOD5
PROCEDURES
MOD3
FF
HEYYOU
MAC

Figure 14

Module Abstractor

The Module Abstractor automatically creates an abstraction of code fragments. This tool provides a mechanism for abstracting source code by hiding redundant and unnecessary portions of the code. The programmer can select one or many source fragments for abstraction. The selected source is hidden from view and replaced by either a note provided by the programmer or a default note provided by the abstractor. The abstracted portions of the code can be reversed.

Figure 15, and 16 display a sample program before, and after abstraction of two fragments of the code. In the example the body of *FOR* loop fragment is selected for abstraction.

The selected fragment is replaced by either a default place holder or by a prompt that is supplied by the user. For example Figure 15 shows the body of the *WHILE* loop is selected for abstraction. After the selection the body of the *WHILE* is hidden and replaced by *statement(s)* as shown in Figure 16.

Figure 15: A Sample Module Before Abstraction

Figure 16: A Sample Module After Abstraction

In Artimis, the Module Abstractor can also be used to create internal documentation. *Internal* documentation is the explanation of the algorithmic behavior of the fragments of the code in the module. When abstracting fragments, the user can enter any annotation regarding the fragment to be abstracted. These annotations replace the actual code. This provides the capability of generating internal documentation by enabling one to produce documentation consisting of a mixture of source and annotation, annotation only, or source only.

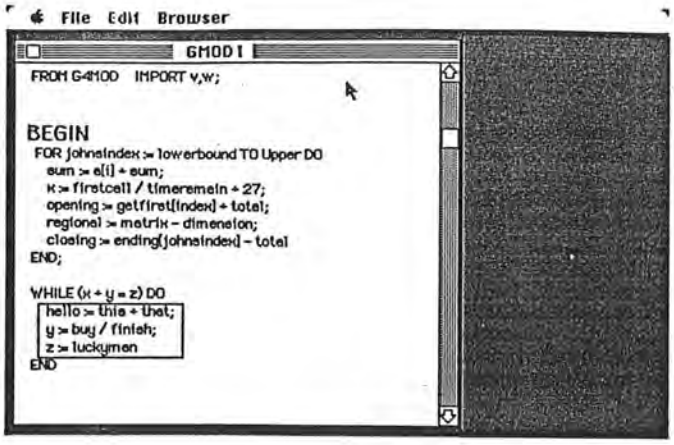

Figure 15

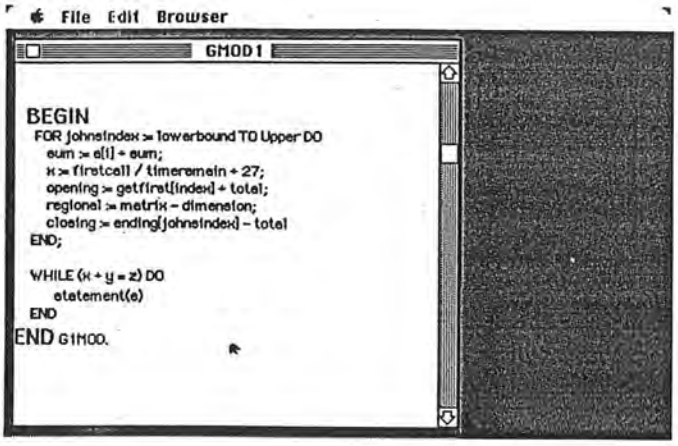

Figure 16

3 Conclusion

It is easier to reuse program fragments than to reinvent them, provided that the time needed for program understanding is less than the time needed for program writing, and provided that the access time for the needed program fragment is sufficiently small. If these two conditions are met then the total programming and debugging time is reduced.

The GrabBag and Browser tools in Artimis provide simple, yet efficient facilities in meeting the above conditions. GrabBag's user interface provides a simple and natural method of searching and locating viable candidates for reuse. The ease of backtracking to previous search selections, and the ability to view all the available categories in one glance creates a highly friendly, and easy environment for locating desired modules. The user interface of GrabBag also makes learning and using it so simple that one does not need to know much about it in order to use it.

The program understanding paradigm was used as a guideline in building tools that are applied to source modules in a non-intrusive manner. The Module interconnection graph and procedure call graph provide a road map and global view of the architecture of a module. The Module Abstractor further hides the non-essentials of the source program and helps to further narrow attention to portions that are essential in understanding the source. Annotating fragments while abstracting them is a natural way to retain the knowledge related to program fragments for further reuse.

References

[Ade81] B. Adelson. Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, 9:422–433, 1981.

[Bar32] F.C. Bartlett. *Remembering.* University Press, Cambridge, 1932.

[BBT79] G.H. Bower, J.B. Black, and T. Turner. Scripts in memory for text. *Cognitive Psychology*, 11:177–220, 1979.

[Bir86] Abbas Birjandi. *A Rule Based Environment for Software Reuse.* PhD thesis, Computer Science Department, Oregon State University, 1986.

[Che83] T.E. Cheatham. Reusability through program transformations. In *Proceedings of Workshop on Reusability in Programming*, pages 122–128, The Media Works, Inc., Newport, RI, September 1983.

[CWW80] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. Nesting in Ada programs is for the birds. In *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, pages 139–145, ACM, Boston, Massachusetts, December 1980.

[Fre83] Peter Freeman. Reusable software engineering:concepts and research directions. In *Proceedings on Workshop on Reusability in Programming*, pages 2–16, Newport, 9 1983.

[Gra81] A.C. Graesser. *Prose Comprehension beyond the word.* Springer-Verlag, New York, 1981.

[Ker83] Kernighan. The unix system and software reusability. In *Proceedings of the Workshop on Reusability in Programming*, pages 235–239, The Media Works, Inc., Newport, RI, September 1983.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming, Theory and Practice.* Addison Wesley, 1979.

[MRRH81] K.B. McKeithen, J.S. Reitman, H.H. Rueter, and S.C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

[SA77] R.C. Schank and R. Abelson. *Scripts, Plans, Goals and Understanding.* Technical Report, Lawernce Erlbaum Associates, Hillsdale New Jersey, 1977.

[San86] Jorge Sanchez. *GrabBag:A Module Data Database.* Master's thesis, Computer Science Department, Oregon State University, 1986.

[SE83] Elliot Soloway and Kate Ehrlich. What do programmers reuse? theory and experiment. In *Proceedings of Workshop on Reusability in Programming*, pages 184–191, The Media Works,Inc., Newport, RI, September 1983.

[SEB82] E. Soloway, K. Ehrlich, and J. Bonar. Tapping into tacit programming knowledge. In *Proceedings of the Conference on Human Factors in Computing Systems*, NBS, Gaithersburg, Md., 1982.

[Shn76] B. Shneiderman. Exploratory experiments in programmer behaviour. *International Journal of Computer and Information Sciences*, 5:123–143, 1976.

[Sta83] American National Standard. *IEEE Standard Glossary of Software Engineering Terminology.* New York, February 1983.

[Wir83] Niklaus Wirth. *Programming In Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, Berlin Heidelberg, 1983.