Computer Aided Instruction Model for Procedural Programming Languages

by

Khalid S. Abdallah

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

March, 1996

COMPUTER AIDED INSTRUCTION MODEL FOR PROCEDURAL PROGRAMMING LANGUAGES

BY

KHALID S. ABDALLAH

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

MARCH, 1996

UMI Number: 1380612

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **KHALID S. ABDALLAH**, under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE.**

Thesis Committee

Thesis Advisor
Dr. Abdallah Al-Sukairi

Member
Dr. Muhammad Shafique

Member
Dr. Mostafa Aref

Department Chairman
Dr. Muhammed Al-Mulhem

Dean, College of Graduate Studies
Dr. Ala Rabeh

26.3.96.
Date

This thesis is dedicated to

my beloved parents, wife, and children.

ACKNOWLEDGMENT

Acknowledgment is due to King Fahd University of Petroleum and Minerals for support of this research.

I wish to express my appreciation to Professor Abdallah Al-Sukairi who served as my major advisor. I also wish to thank the other members of my Thesis Committee Professor Muhammad Shafique and Professor Mostafa Aref.

A very special thank you to Saudi Aramco for giving me the chance to pursue my advanced degree program.

Finally, my profound thanks are due to my parents, wife, children, brothers, sisters, and friends for all their love, affection, support, and well wishes which have always kept my spirits up. It is their sacrifices and farsightedness which made this work possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES ... continued

LIST OF FIGURES ... continued

THESIS ABSTRACT

FULL NAME OF STUDENT:	KHALID S. ABDALLAH
TITLE OF STUDY:	COMPUTER AIDED INSTRUCTION MODEL FOR PROCEDURAL PROGRAMMING LANGUAGES
MAJOR FIELD:	INFORMATION AND COMPUTER SCIENCE
DATE OF DEGREE:	MARCH 1996

Computer Aided Instruction (CAI) is the process of teaching by computer. Computer-based educational systems are valuable tools to improve learning. It is thus worth endowing programming environments with educational capabilities to help students learn programming. Computer Programming is a typical introductory computer science course that is offered to all science and engineering students in an academic environment. The amount of attention traditionally paid to the syntax of a programming language in the first course is excessive, and ought to be replaced with a more balanced introduction to the discipline. Program development process has three distinct phases: understand the problem, design the solution, and code the program. Much of the creativity in programming is concentrated in designing the solution rather than implementing it in a selected language. We have developed a model, called Computer Aided Programming Education (CAPE), that supports the program development process. CAPE consists of problem definition and algorithm construction modules. Algorithm construction module is currently supported with a facility to map an algorithm(s) into FORTRAN or Pascal code. CAPE motivates students to study the algorithmic problems operationally and provides a mechanism for rapid prototyping in various programming languages.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
Dhahran, Saudi Arabia

March 1996

خلاصة الرسالة

إسم الطالب الكامل: خالد سعيد محمد عبدالله

عنوان الدراسة: نموذج تعليمي بمساعدة الحاسب الآلي للغات البرمجة الإجرائية

التخصص: علوم الحاسب الآلي والمعلومات

تاريخ الشهادة: مارس ١٩٩٦ م

التعليم بمساعدة الحاسب الآلي (CAI) هو عملية التدريس بالحاسب الآلي. إن أنظمة التعليم بالحاسب الآلي هي وسائل قيمة لتحسين التعلم لذلك فإنه يستحق دعم بيئات البرمجة بقدرات تعليمية لمساعدة الطلاب لتعلم البرمجة. تعتبر برمجة الحاسب الآلي مادة نموذجية في البيئة الأكاديمية تقدم لجميع طلبة الهندسة والعلوم كمقدمة لعلوم الحاسب الآلي. إن مقدار الإهتمام التقليدي المعطى في هذه المادة لبناء جمل لغة البرمجة هو إهتمام مبالغ به ويجب أن يبدل بمقدمة أكثر توازناً لهذا المجال. إن عملية تطوير أي برنامج تمر بثلاثة أطوار مميزة: فهم المسألة ، تصميم الحل ، وكتابة البرنامج. تركز غالبية الإبداع في البرمجة على تصميم الحل بدلاً من تنفيذه بلغة مختارة. وعليه فقد طورنا نموذجاً يدعى (CAPE) الذي يدعم عملية تطوير البرنامج. يتكون هذا النموذج من وحدة تعريف المسألة ووحدة تصميم الحل الإجرائي مدعمة بوسيلة لوضع التصميم بلغتي (FORTRAN) و (Pascal). من ميزات هذا النموذج أنه يحفز الطلاب لدراسة المسائل الإجرائية عملياً ويمدهم بآلية لإنتاج نموذج أولي سريع بعدة لغات للبرمجة.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران، المملكة العربية السعودية

مارس ١٩٩٦ م

CHAPTER 1

INTRODUCTION

Most educators have become convinced that the computer's role in supporting teaching process is a valid and exciting one. There is at present a considerable interest in the use of computer assisted learning in higher education. It was recognized very early that educating was so complex that computers could only provide help in teaching rather than "replace" teachers. The focus is on using the computer to assist teachers and students in the process of learning. Education involves teaching and learning. The teacher is teaching students, and a student is learning. CAE (Computer Aided Education) system is a computer software that aids in the education of a student, i.e. aiding the teaching and learning of students [4]. The computer facilitates both the teaching and learning aspects of education. CAI (Computer Aided Instruction) emphasizes the instruction side of education where CAL (Computer Aided Learning) emphasizes the learning side of education.

Computer-based educational systems are valuable tools to improve learning. It is thus worth endowing programming environments with educational capabilities to help students learn programming. The system should be designed to teach students how to build abstract models independent of a specific programming language with facilities guiding users to map the model into an effective program in a specific language. Programming is a vehicle to get students to learn problem solving abilities [3]. Learning to program requires modeling capabilities, the ability to understand how to adapt the model to the environment and analyze what properties must be owned by the model in order to be effective.

1-1 Problem Definition

Computer programming is a typical introductory computer science course that is offered to all science and engineering students in an academic environment. The course emphasizes a selected procedural programming language syntax and implementation issues. Much of the creativity in programming is concentrated in designing the solution rather than implementing it. There are a number of programming concepts that are hard to visualize using a black-board only. A number of instructors do an excellent job by simulating operations and representing steps execution. Flowcharting is a famous method that is used to represent program instructions and their flow control. Instructors feel that a higher level of abstraction is required to help in teaching programming.

In a laboratory session, students are asked to develop a program that solves a given problem. Given an initial problem description, students must plan a way in which a computer can be used to solve the problem, and then describe this plan very precisely in some procedural programming language. This process has three distinct phases: *understand* the problem, *design* the solution, and then *code* the program. The problem is presented on a piece of paper with an explanation by the instructor, if needed, on the black-board. To design the solution, we encourage students to sketch an algorithm to plan the solution. Some students neglect the design phase and start coding. We find it hard to emphasize such an important phase for a program development.

To code the program, students are provided with an environment where a selected language's compiler is provided with facilities to code and debug programs, look-up syntax of instructions, and even visualize the execution of the program. The coding environment is not part of our problem since different compiler vendors provide an excellent coding and debugging environment supported with an on-line reference material. Our major problems here are understanding the problem and designing the solution.

To complete a lab session, students usually are asked to submit printed programs supported with test data for a given set of problems. Instructors have to manually go through the programs to validate their correctness. For a questioned solution, the instructor may review the execution of a program with the student. The process is time consuming and may give room for students to copy a solution from another student.

Algorithmic design is an essential component of the preparation of all engineering and science students. Programming requires problem-solving skills that are normally practiced during a laboratory session. A special model is required to effectively support the teaching of procedural programming languages at an introductory level. The amount of attention traditionally paid to the syntax of a programming language in the first course is excessive, and ought to be replaced with a more balanced introduction to the discipline.

1-2 Objectives

The objective is to design a model that will aid in introducing procedural programming languages. The model should fulfill academic requirements and utilize latest computer technology. Students should be provided with an environment where they could understand a given problem, construct its algorithmic solution, and then code the program using a procedural programming language. Instructors should have the ability to register students, define problems, and evaluate students developed solutions.

1-3 Organization Of Thesis

Chapter 2 concentrates on the literature review. It begins by defining CAI and identifying programming environments design approaches, reviewing the structure of teaching programming under an academic environment, and presents three models reported in recent literature. Chapter 3 presents our Computer Aided Programming Education (CAPE) model. It starts with the philosophy behind the model, and then describe its object and dynamic model. Chapter 4 provides a case study using instance diagrams and Windows interface. Chapter 5 concludes our discussion with closing remarks and future directions.

CHAPTER 2

LITERATURE REVIEW

Education and multimedia have a symbiotic relationship; education is shaping the development of multimedia, and multimedia is opening new possibilities in education [14]. Educators have supplied a stream of application ideas that range across all levels of technical sophistication. This process has been important to multimedia technology in terms of understanding applications and the tools and technology necessary to support them. Multimedia is opening new possibilities in education and it is used as an effective communication medium. The aim is to utilize multimedia technology in developing a model that aids in teaching a procedural programming language.

2-1 Overview

Computer aided instruction (CAI) is the process of teaching by computer [11]. This process contains a number of teaching functions that the computer is capable of doing; CAI categories illustrate those teaching functions. Approaches towards designing a CAI model for programming education vary along a line whose extremes are software-engineering oriented systems and cognitively oriented systems. Between the two extremes is found what is called educational oriented programming environments. Researchers have come up with a number of principles about how to build educational environments in academic situations. Those design approaches and principles are discussed in section 2.

Any procedural programming language is characterized by the presence of constructs describing commands issued to a virtual machine. Statement-level control structures refer to language constructs that are used to order the sequence in which individual statements are executed. There are three kinds of control structures: sequencing, selection, repetition. Teaching a procedural language in an academic environment embodies a number of constraints that influence our design. Section 3 discusses those issues and presents a typical course structure used in KFUPM to introduce FORTRAN programming.

Section 4 presents a selected number of models present in recent literature. METKIT CAI system primarily addresses software engineers to provide them with some knowledge in a domain and guidance through it based on didactic strategies. INTELLITUTOR system is an integrated intelligent programming environment to learn Pascal. It attempts to work as a human programming tutor to guide a student in writing a program, detects logical errors, and makes advises to fix them. CAE model provides students with a problem-solving environment where a problem is presented and student actions are recorded and analyzed.

A system can be viewed from three related but different viewpoints, each capturing important aspect of the system. The Object Modeling Technique (OMT), presented in section 5, consists of three kinds of models. The object model describes the static structure of a system in terms of objects and relationships corresponding to real-world entities. The dynamic model describes the control structure of a system in terms of events and states. The functional model describes the computational structure of a system in terms of values and functions. Different problems place different emphasis on the three kinds of models.

2-2 Computer Aided Instruction

Computer-aided instruction (CAI) is the process of teaching by computer. Other definitions include: the system of individualized instruction that uses a computer program as the learning medium or an educational computer program which presents knowledge or skills that can be learned by someone using the computer [11]. CAI systems range from book analogies in which a student "turns pages" to systems that use multiple media to offer intelligent interactive assessment. In addition to text, students would like to have sound, pictures, maps, diagrams, animation and humor [6]. Multimedia extends the power of the computer as a communication medium. This expansion is based largely on the addition of new media, especially of sound and image due to jump of desk-top processing power [14].

Multimedia involves the combination of different representations and offers something new to areas that deal with how ideas are represented and how students explore them. It allows the combination of materials common in educational computing such as simulation, tutorial, and reference materials. Multimedia design involves the grouping of information resources (text, graphics, animation, video) and the controls needed to manipulate them. Now it can be said that CAI can replicate existing teaching practices and, more importantly, can stimulate research and reevaluation of what makes effective teaching [7].

2-2-1 CAI Categories

CAI categories illustrate the teaching functions of the computer. They are drill and practice, page turners, tutorials, and simulation [11]. Many CAI programs can not be neatly classified in one category to the exclusion of all others, so the functionality and technology used will be discussed, rather than looking at specific CAI programs.

Drill and Practice

Drill and practice is the classic mode of CAI and the easiest type to write. These programs do not actually teach, in the sense of explaining new information, but are based on the premise that concepts already have been taught elsewhere. Here the computer's role is to follow up the instruction. They help us to learn and remember by repetition and examples.

Drill is the CAI mode in which the student performs an activity repeatedly in order to memorize facts. The computer compares the learner's answer with the correct answer. If the student answers incorrectly, usually the question is presented again. Sometimes hints or explanations are given to the student after an incorrect response. When the problem is answered correctly, positive feedback is provided followed by another problem to solve. The *practice* part involves using the computer to provide a means of learning processes, procedures, and skills. Practice programs provide feedback to the learner on the correctness of responses and some adjust the level of difficulty correspondingly. They often require negotiation through a complex procedure in order to reach a conclusion.

Page Turners

Page turners use the computer screen as a "page" on which information is displayed, just as in the pages of a book. The primary difference between the printed page and the computer page is that the former is a static display while the computer screen is dynamic. For example, animation can be incorporated into pages of computer displayed text. Instructions that incorporate page turning are often incorporated into many types of software. A relatively small amount of information stored on a computer disk can be more convenient than keeping up with or constantly referring to a manual. Many applications programs that are not primarily CAI programs, such as database, spreadsheet, and word-processing, include instructions on the use of the program.

Page turners are mostly implemented using hypermedia technology. *Hypermedia* is a technology for information delivery based on the notion that information can be divided into nodes and links among them [18]. The content of a node can be a picture, a block of text, or an animation. A node can be a single CAI frame, a series of frames, or an instructional segment that include material from a CD-ROM or videodisks. The nodes can be linked together to form hierarchical or non-hierarchical structures. This environment enables the author to generate complicated data structures that can be easily navigated by the learner.

Tutorials

Tutorials assume the role of teacher for providing new information. A tutor is the one who is in general charge of a person's education. Tutorial CAI programs imitate the human tutor by presenting information in small segments, often in text form. Visual, animation, or sound sometimes employed to clarify new information. Good tutorial programs ask the learner frequent questions and evaluate the responses. Incorrect responses might result in a repetition of the relevant information. Correct learner responses are reinforced by the program much as they would be by human tutor. Tutorial programs may include summaries and drill or practice segments at regular intervals. Diagnostic testing might be provided in the tutorial program to allow the learner to skip over information that is already known.

In most tutorial CAI, the learner controls the pacing and he or she can take breaks or quit at any time. There are two general forms of tutorial programs, linear and branched. In *linear programs* all learners follow the same path of information regardless of their responses. *Branching programs* present material following a dynamic path based on student responses. For example, a student who misses several questions may be directed by the program into a remedial sequence path that is never seen by other learners.

Tutorials can be produced using an authoring system. An *authoring system* is an application program that is designed to make the CAI author's task more simple than using a language [11]. All authoring systems are somewhat restricting by imposing certain limitations on the author. One of the most widely respected authoring systems is Authorware Professional [19]. It is an icon-based object-oriented authoring system. It features answer analysis, behavior-monitoring, and record-keeping functions. If you utilize its capabilities and avoid its restrictions, an effective application with maximum productivity is achieved.

Simulations

Simulations constitute one of the most powerful and potentially valuable applications of CAI. CAI simulation programs are designed to represent the essential elements of some real-life or imaginary event or phenomenon without its attendant inconveniences. There are many times when teachers would like students to explore a situation but find it impossible to bring them into direct contact with it. Using a CAI simulation program, the student makes decisions and sees the consequences of those decisions. Examples of these programs include airplane flight simulator, SimCity, and Navigator [14].

Simulations imply a very different notion of the instructional uses of computers than do courseware tutorials, drills, or practice programs. Rather than directing students toward very explicit outcomes, simulations encourage them to discover important concepts implicitly by exploring the computerized environment, setting events in motion, and watching the results. Authors of simulation programs do not try to anticipate every possible response or question. Rather, they try to create a computerized environment in which students can bring about demonstrations of the relationships among elements in a given environment.

2-2-2 Design Principles

Researchers have come up with a number of principles about how to build educational environments in academic situations [10]. We will discuss important principles that a CAI model should possess.

Learn by doing. Learning should center on a task that requires the skills and knowledge we want to teach. The task should be challenging, but within a student's ability. The student should construct something rather than listening or reading. When students learn by doing, they acquire a case base. They can then use the cases they have experienced or heard about from others to frame their responses in the future.

Problems, then instruction. Students respond best to instructions when what they hear from the teacher relates to problems with which they are struggling. This method will teach students to associate a correct solution with a problem they may encounter in the future.

Find the fun. An instructional designer's job is to make learning fun, which means that students should enjoy what they are doing. Constructing something or viewing an animation that conveys information in an educational manner is considered to be fun.

Power to the student. The student should control the educational process. The recommended path might be marked, but students should possess the power to determine or change the next step.

Learn by exploring. When students become involved with a subject, they naturally generate questions. Students should be able to navigate around an information base to easily discover its contents and find answers. It is the instructional designer's task to provide effective navigational tool.

2-2-3 Design Approaches

Tools are designed following approaches which vary along a line whose extremes are software-engineering oriented systems and cognitively oriented systems. Between the two extremes is found what is called educational oriented programming environments. Today, work is largely oriented toward the design of education oriented programming environments [3]. Attention is focused on tools which unite two capabilities: that of offering facilities to develop software and that of helping students learn programming concepts by assisting them in using these facilities.

Software-engineering

Software-engineering orientation indicates that the design of a tool is mainly based on the experiences in the field of systems for software development; e.g. Turbo Pascal 6.0. Graphical representations and pictures have come into play in the programming process. Visual programming, the use of graphics in the process of programming, has gained momentum in recent years and is progressing in two major directions [16]. In one direction, graphical techniques and pointing devices are used to provide visual environments for program construction and execution, information retrieval and presentation, and software design and understanding. In another direction, languages are designed to handle visual information, support visual interaction, and actually program with visual expressions.

Cognitive

Cognitive orientation systems are designed based on some hypothesis on the learning of problem solving, and analyses of the structure of programming knowledge on that basis. They model the organization of knowledge on programming, and use this model to supervise the student's activity. For example, ACT (Adaptive Control of Thought) [3] is used for program writing where programming knowledge is modeled by a set of production rules, and it is used to build the student model.

2-3 Teaching Procedural Programming Languages

Learning to program has become a common subject of study from a number of different perspectives [5]. Computer scientists aim at improving the programming environment with regard to the design of programming languages, interfaces, and program development tools. Cognitive scientists tend toward using programming as an example of the use of cognitive skills in general, whether problem-solving, learning, or correcting mistakes. Research into learning to program within the field of education is concerned with the development of good methods of programming instruction. The research in this thesis intersects those perspectives to provide young students with a rich educational environment that forms a solid background in procedural programming and problem-solving techniques.

Reports show that FORTRAN was used by 81% of most engineering programs in 1984 [17]. This percentage has dropped to 57% in 1987. Other languages have shown some popularity, with Pascal and C being the leaders. Introduction to a programming course is usually offered under an academic environment in conjunction with a programming language. At KFUPM, FORTRAN remains the favorite language since it provides engineers with a simple language that is specifically designed for engineering work.

The amount of attention traditionally paid to the syntax of a programming language in the first course is excessive, and ought to be replaced with a more balanced introduction to the discipline. Introduction to a programming language is defined as a separate knowledge unit to allow different courses to introduce programming language in conjunction with different knowledge units. We should not focus on the syntax of the selected language at such a level, instead we should emphasize on problem solving skills using a programming language [20].

2-3-1 Language Characteristics

Any procedural programming language, such as FORTRAN, C, or PASCAL, is characterized by the presence of constructs describing commands issued to a virtual machine [13]. The construct that makes programming languages most prominently imperative is the statement. In commonly used languages, programs are sequences of statements, each describing a set of actions to be performed. Programs use explicit control structures to specify precisely the order in which their instructions must be executed. Statement-level control structures refer to language constructs that are used to order the sequence in which individual statements are executed. There are three kinds of statement-level control structures: sequencing, selection, and repetition [15].

Sequencing is used to indicate that the execution of statement (B) must follow the execution of another statement (A). FORTRAN adopts a line-oriented format that uses the end of the line implicitly to separate instructions and impose a sequencing mechanism among them. *Selection* control structures allow the programmer to specify that a choice is to be made among a certain number of possible alternative statements. The logical IF statement of FORTRAN is an example of a selection statement that specifies the execution of a statement according to a Boolean expression. *Repetition* involves repeating a number of actions. FORTRAN provides the DO statement, with which a number of iterations can be specified by introducing a counter (the loop control variable). Repetition over a finite set of values can be used to model the frequent case in which the number of iteration is known in advance, such as in the processing of arrays. FORTRAN does not support condition-driven repetition structure, but it could be simulated using IF and GOTO statements [12].

2-3-2 Course Structure

Introduction to a programming course is usually offered to all science and engineering students at the freshman level. The course requires the student to attend a lecture where programming concepts are explained and how they are implemented using a procedural language. The course has a laboratory session where students practice coding, compiling, and executing programs using a selected language.

Lecture Session

Although lecturing is not the only teaching method, it is the dominant one in many disciplines and in many educational institutions. Programming is traditionally taught using a bottom-up approach [9], where details of syntax and implementation of data structures are the predominant concepts. The following is a list of topics introduced in a typical introductory programming course [12]:

- Constants and variables data types
- Arithmetic and logical operations
- Assignment statement and simple I/O
- Selection construct

- Subroutine and function
- One and two-dimensional arrays
- Repetition construct
- Output design
- Sort and search applications

Laboratory Session

The course has a laboratory session where students practice coding, compiling, and executing programs using a selected programming environment. They are provided with a simple problem that requires a coding of a program. It has been noted that most students start coding the program without attempting to sketch the solution. They debug their program on a trial-and-error basis. At the end, they submit their solution on paper for the instructor to evaluate. This scheme is not helping students develop their problem solving skills, instead, they fear attending such laboratory sessions.

The program construction process encompasses activities necessary to transform the problem specification into an executable program. Programming can be regarded as a problem solving process [8]. Any course which attempts to teach computer programming must require its students to actually write programs and run them on a computer. It is then helpful for those who teach the course to see the results of running these programs. Traditionally, this is arranged by asking them to submit printed listings of the programs along with the results obtained from running them. Unfortunately, this arrangement is not very satisfactory with large classes, where the volume of paper to be handled can become inconveniently large.

2-4 Previous Work

A number of tools and models reported in recent literature vary in complexity and environments intended for. A review of a number of these models has encouraged us to proceed in defining our new model. This section presents three models: the METKIT CAI, INTELLITUTOR system, and CAE model.

2-4-1 METKIT CAI

The METKIT CAI system [1] primarily addresses software engineers, however it is flexible in that it can be used for other domains. The system integrates concepts of hypertext with those of computer aided instruction, providing different types of access to conceptual knowledge about certain domain. It provides users with some knowledge in the domain with a facility to browse through the knowledge base. It also provides novice users with guidance through that knowledge based on didactic strategies. Figure 1 lays out the architecture of the system.

Domain Knowledge Component

The domain knowledge represents knowledge about the concepts of some domain. The domain editor is used to build up and edit the domain knowledge by defining the sets (topics), elements (concepts), and relations. Expert users can access the knowledge using either dictionary or network-browser. Dictionary presents the named items in alphabetic

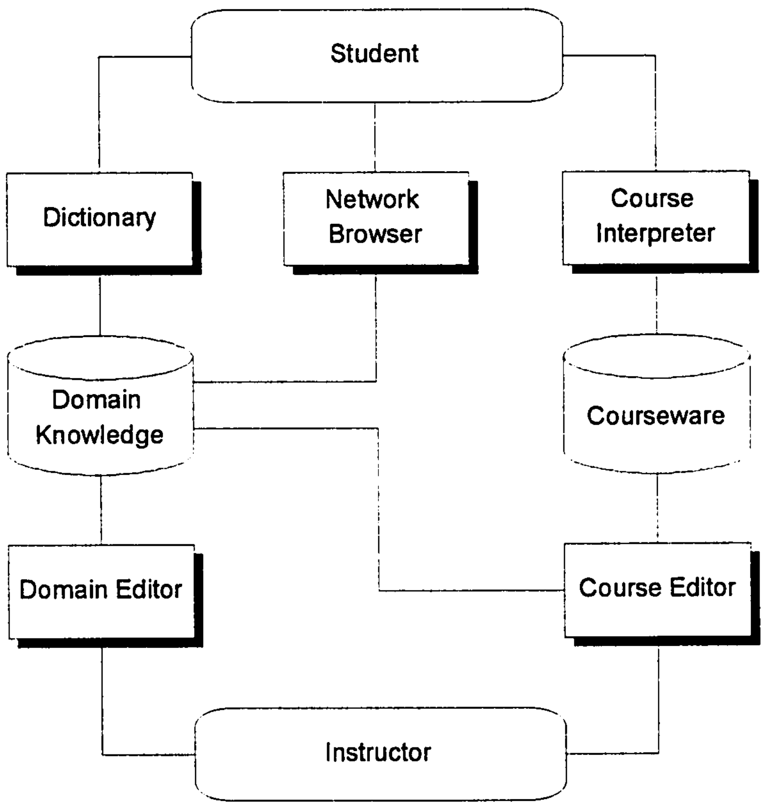


Figure 1. METKIT System Components

order for direct access to the term. Network-browser allows the user to navigate through the networks defined by the relations of concepts which represent the structure of the domain.

Courseware Component

The system provides the instructor with a course editor to select parts from the domain knowledge and attaching guiding information to create a courseware. Developing a courseware for a given topic is done by defining the courseware items, the target audience and the teaching goals, selecting appropriate pieces of information from *Domain Knowledge*, putting these into a convenient sequence, and developing adequate presentations and interactions. The courseware is accessed by beginners via course interpreter. After selecting a courseware item, the user may get a text to read, a list of lower level courseware items for further study, or a list of questions to be answered.

2-4-2 INTELLITUTOR System

INTELLITUTOR system is an integrated intelligent programming environment for learning Pascal programming [2]. It attempts to work as a human programming tutor to guide a student in writing a program, detects logical errors, and makes advises to fix them and lets the student notice his misunderstanding. Teaching is done based on the student model. The system is implemented in the frame-based knowledge engineering environment ZERO and working on a UNIX workstation for system evaluation. Figure 2 provides an overall view of the system.

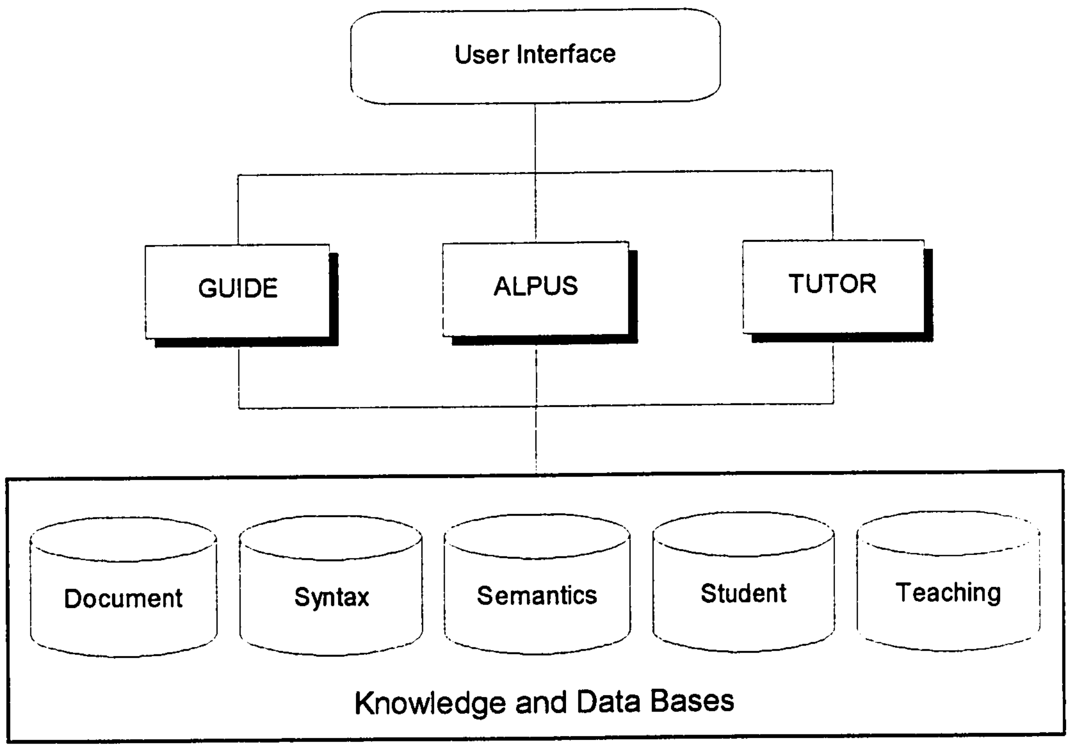


Figure 2. An Overall View of INTELLITUTOR

Knowledge and Data bases

The knowledge bases consist of the knowledge on syntax based on Pascal syntax graph, knowledge on semantics, the knowledge on students which is represented as a student model, and knowledge on teaching. The document database includes Pascal programming manual, the language specification in BNF and syntax graph, and program examples. The document database is organized in a form of structured hypertext and linked together to display it easily during a coding or tutoring session. Although the current system deals with Pascal programs, major knowledge may be used for other procedure-oriented programming languages.

ALPUS Module

The ALPUS module is a knowledge-based program understander. It uses four kinds of programming knowledge: knowledge about algorithm, programming techniques, programming language, and how students make errors. The knowledge about algorithm is represented in a hierarchical data structure called Hierarchical Procedure Graph (HPG). HPG is an abstract representation of knowledge about an algorithm. Each node of HPG represents a chunk of operations of the algorithm. Knowledge about programming techniques and errors is attached to associated nodes. Program comprehension is achieved by means of semantic pattern matching method. The pattern matching method limits program comprehension since the knowledge base does not include every patterns written by students. This can be supported by the student modeling module of the TUTOR system. HPG-based knowledge structure provides a simple reasoning method.

GUIDE Module

GUIDE is a guided editor for easy coding using the information of a manual and syntax graph. There are three modes available: a text editor mode, a syntax editor mode, and an HPG-based editor mode. The text editor mode does not provide any guide and represents a normal text editing session. In a syntax editor mode, guide is provided by presenting the syntax graphs of the language. At each place of writing a new statement, a list of possible instructions is listed for the user to choose from. In HPG-based editor mode, the program coded by the student is converted into an internal representation and passed to ALPUS system, where logical errors are detected and intentions are inferred. Based on the analysis, advice statements are presented to the student.

TUTOR Module

The TUTOR subsystem consists of student modeling module and a teaching module. It receives information about a student from ALPUS, generates the student model for him, and teaches him appropriate programming knowledge based on student model. So teaching of programming is done by both ALPUS and TUTOR. TUTOR has a skeleton of structured programming knowledge for building a *student model*. The skeleton is defined to cover a very wide range of programming knowledge. Model generation is achieved by marking simple information onto associated links of the skeleton. Algorithm-related knowledge are marked with "understand" and "do-not-understand". Technique-related knowledge are marked with "know", "do-not-know", and "error". According to the information received from ALPUS, initial marking is done. The scope of the student is then decided based on the given problem. If there exists unmarked links which are related to "unknown" or "buggy" knowledge within the scope, interactions are requested by the system to the student. Then, teaching of programming knowledge is made according to the student model. Summary reports can also be generated for the instructor to evaluate education progress of each student.

2-4-3 CAE Model

An interesting Computer Aided Education (CAE) model was published last January 1995 [4]. The model aids both the student and the teacher in performing their different educational roles. A high-level schematic overview of the model is shown in Figure 3. The student interacts with the system via the problem-solving environment called the *virtual world.* The student attempts to solve the problem by interacting with the objects to obtain information about the problem and eventually he should obtain enough information in order to solve the problem.

Knowledge and Cases Logbook

The CAE system has access to several sources of knowledge such as knowledge on the problems in a particular application area, as well as how to determine the solution of the problem. Another source of knowledge is the knowledge on problem-solving skills and strategies. With this knowledge the system is able to analyze the attempt of a student, provide a diagnosis of his problem-solving skills and strategies, and suggest possible solutions to weaknesses detected. Cases logbook consists of student's problem-solving attempt, the configuration of the problem attempted, and the analysis and diagnosis produced by the system.

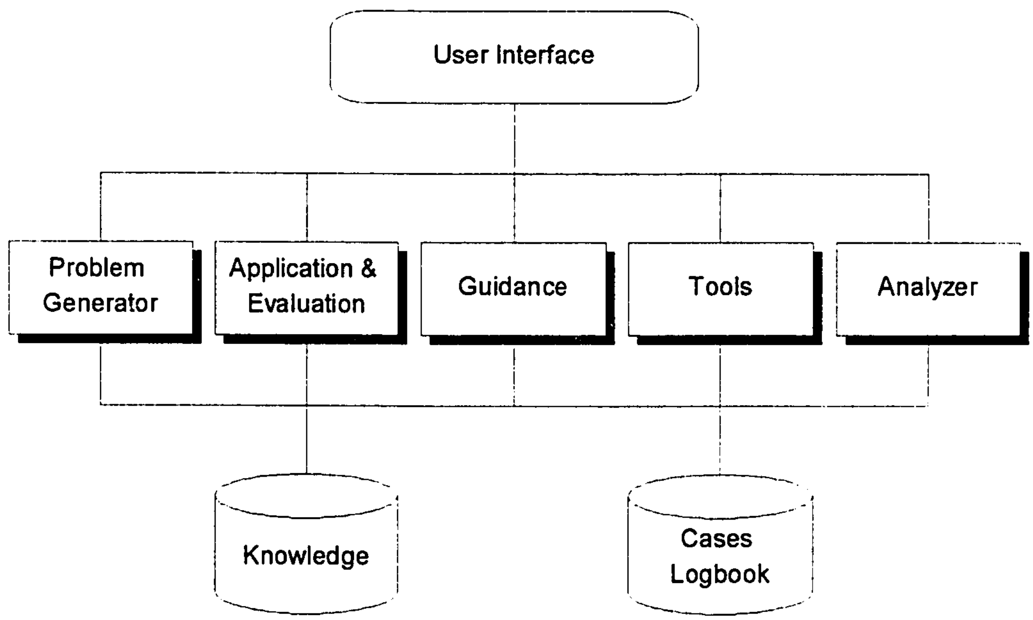


Figure 3. The CAE System Model

Problem Generator Module

The problem generator is considered to be the heart of the system. It is capable of selecting an appropriate problem and setting up the virtual world. It monitors students activities while solving the problem. Initially, it obtains the identity of the student then accesses the database of cases to build a history of the student. Then activates the application area and evaluation module to select a suitable problem. Once the problem has been selected, the tool module is activated to select appropriate tools needed by the problem. The problem is then presented via the virtual world for the student to find the answer. The analyzer module is activated to present the teacher interface and stores relevant data upon completion.

Application & Evaluation Module

This module has access to a database of problems. Each problem has its own description, representation, the set of tools needed to solve the problem, and the solution to the problem. This module determines if the student's solution is correct or not. Guidance module requests this module to evaluate student partial solution to be able to provide useful hints.

Tools Module

This module presents the student with a set of tools, or objects that can be manipulated in the virtual world, that are necessary to solve the problem. Each problem requires its own set of tools. The student has total control of them and can inquire about their static information by a right-click of the mouse.

Guidance Module

This module provides intelligent help that promotes the cognitive learning processes of the student. Depending on the situation at hand and the state of the solution, relevant clues are provided without spelling out the solution. Guidance module knows what the problem is, and how to determine the correctness of the solution. It also knows what tools are available in the virtual world and their relation to the problem. It has access to knowledge on problem-solving skills and strategies.

Analyzer Module

This module is capable of interpreting student's problem-solving attempts and provides a diagnosis of his strategies. It analyses the activities recorded in activities logbook. Rules is a popular representation scheme where the knowledge is encoded in the form of IF-THEN rules. The task of the analyzer module is to analyze the activities of the student, as recorded into the activities logbook, and to produce an interpretation and diagnosis about the problem-solving skills and strategies of the student.

2-5 Object Modeling Technique

A model is an abstraction of something for the purpose of understanding it before building it. Abstraction is a fundamental human capability that permits us to deal with complexity. A good model captures the crucial aspects of a problem and omits the others. A system can be viewed from three related but different viewpoints, each capturing important aspect of the system. The Object Modeling Technique (OMT) [22] consists of three kinds of models. The *object model* describes the static structure of a system in terms of objects and relationships corresponding to real-world entities. The *dynamic model* describes the control structure of a system in terms of events and states. The *functional model* describes the computational structure of a system in terms of values and functions. Different problems place different emphasis on the three kinds of models, but all three are necessary for any large system.

2-5-1 The Object Model

The *object model* describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations. The object model provides the essential framework into which the dynamic and functional models can be placed. The object model is represented graphically with object diagrams containing object classes. Classes are arranged into hierarchies sharing common structure and behavior and are associated with other classes. Classes define the attribute values carried by each object instance and the operations which each object performs or undergoes.

Objects

An *object* is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: they promote understanding of the real world and provide a practical basis for computer implementation. Decomposition of a problem into objects depends on judgment and the nature of the problem. There is no one correct representation. All objects have identity and are distinguishable. The term *identity* means that objects are distinguished by their existence and not by descriptive properties that they may have. A *derived object* is defined as a function of one or more objects. The derived object is completely determined by the other objects and thus it is redundant, but may be included in an object model to ease comprehension. Similarly, there are also *derived links* and *derived attributes*.

Object Class

An *object class* describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. By grouping objects into classes, we abstract a problem. Abstraction gives modeling its power and ability to generalize from a few specific cases to a host of a similar cases. Common definitions are stored once per class rather than once per instance. Operations can be written once for each class, so that all the objects in the class benefit from code reuse.

Object Diagrams

Object diagrams provide a formal graphic notation for modeling objects, classes, and their relationships to one another. Object diagrams are concise, easy to understand, and work well in practice. There are two types of object diagrams: class diagrams and instance diagrams. A *class diagram* is a schema, pattern, or template for describing many possible instances of data. An *instance diagram* describes how a particular set of objects relate to each other. Figure 4 shows a class diagram and an instance diagram of one-to-many association and corresponding links. Each association in the class diagram corresponds to a set of links in the instance diagram. In the example, each company has a set of employees where *Employs* is the name of the association. A given class diagram corresponds to an infinite set of instance diagrams. Instance diagrams are useful for documenting test cases and discussing examples.

The symbol for a class is a box with class name in boldface, association is a line between classes, and object instance is a rounded box with the class name in parentheses at the top of the object box. A derived object/attribute is defined as a function of one or more objects/attributes. Slash or diagonal line on the corner of a class box is used for derived objects and in front of an attribute for a derived attribute. Figure 4 includes object modeling notation for classes. A class is represented by a box which may have as many as three regions. The regions contain: class name, list of attributes, and list of operations.

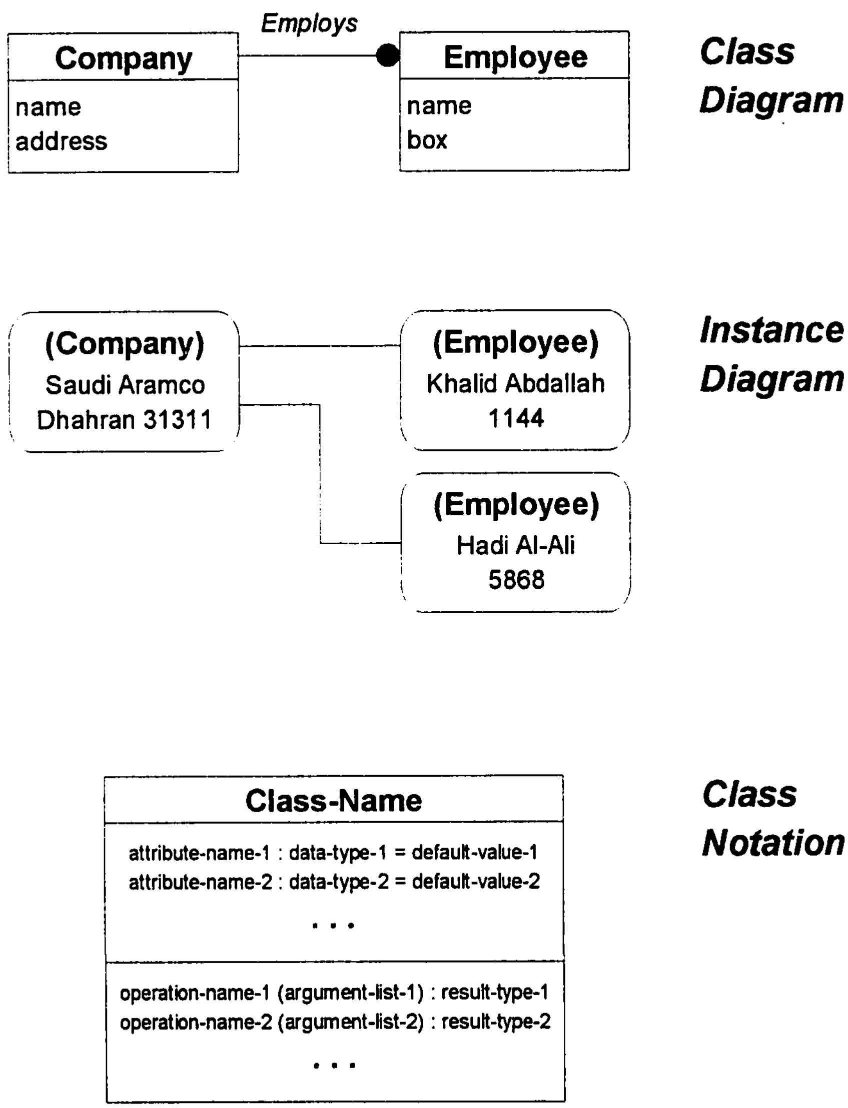


Figure 4 : Class and Instance Diagrams

Attributes

An *attribute* is a data value held by the objects in a class. Different object instances may have the same or different values for a given attribute. Each attribute name is unique within a class (as opposed to being unique across all classes). An attribute should be a pure data value, not an object. Explicit object identifiers are not required in an object model. Each object has its own unique identity. Attributes are listed in the second part of the class box. Each attribute name may be followed by optional details, such as type and default value. Depending on the level of detail desired in the object model, you may choose to omit showing attributes in class boxes.

Operations

An *operation* is a function or transformation that may be applied to or by objects in a class. All objects in a class share the same operations. Each operation has a target object as an implicit argument. The behavior of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation. The same operation may apply to many different classes. Such an operation is *polymorphic*; that is, the same operation takes on different forms in different classes. A *method* is the implementation of an operation for a class. An operation may have arguments in addition to its target object. Such arguments parameterize the operation but do not affect the choice of method. The method depends only on the class of the target object. When an operation has methods on several classes, it is important that all methods have the same *signature*—the number and types of arguments and the type of result value.

Operations are listed in the lower third of the class box. Each operation name may be followed by optional details, such as argument list and result type. An argument list is written in parentheses following the name with arguments separated by commas. The name and type of each argument may be given. Operations may be omitted from high-level diagrams.

Modules

A *module* is a logical construct for grouping classes, associations, and generalizations. A module captures one perspective or view of a situation. An object model consists of one or more modules. Modules provide an intermediate unit of packaging between an entire object model and the basic building blocks of class and association. The same class may be referenced in different modules. Referencing the same class in multiple modules is the mechanism for binding modules together.

Links and Associations

A *link* is a physical or conceptual connection between object instances. An *association* describes a group of links with common structure and common semantics. All the links in an association connect objects from the same classes. Modeling a link as a pointer disguises the fact that the link is not part of either object by itself, but depends on both of them together. Figure 5 shows an example of modeling a company that employs many persons and owns many computers. Each employee is assigned zero or more computers; some computers are for public use and are not assigned to anyone.

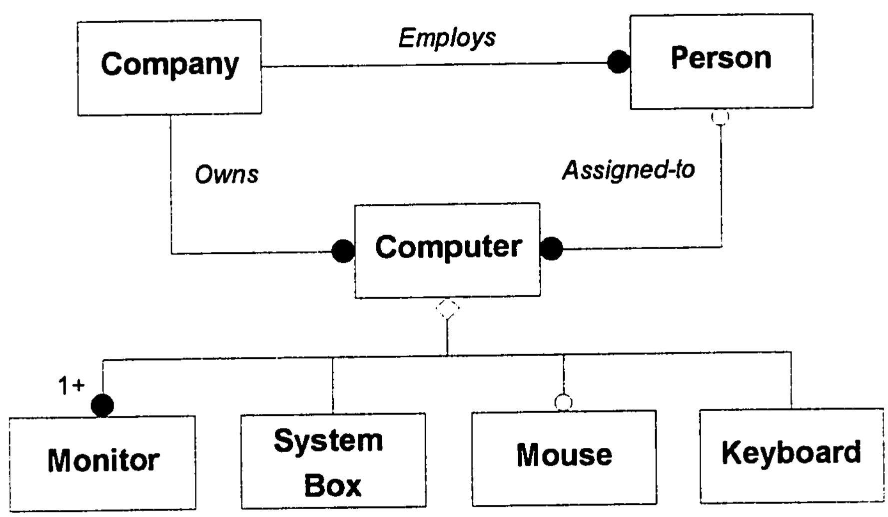


Figure 5 : Links and Association Example

The notation for an association is a line between classes. *Multiplicity* specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity is indicated with special symbols at the ends of association lines. It can be specified with a number or set of intervals. There are special line terminators to indicate certain common multiplicity values. A solid ball is the symbol for "many" meaning zero or more. A hollow ball indicates "optional," meaning zero or one. A line without multiplicity symbols indicates a one-to-one association.

Aggregation is "a-part-of" relationship in which objects representing the *components* of something are associated with an object representing the entire *assembly*. Aggregation is drawn like association, except a small diamond indicates the assembly end of the relationship. In our example, a computer is composed of one or more monitors, a system box, an optional mouse, and a keyboard.

Generalization and Inheritance

Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences. *Generalization* is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass*. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to *inherit* the features of its superclass. Generalization is sometimes called the "is-a" relationship because each instance of a subclass is an instance of the superclass as well. Figure 6 shows a simplified object model of a shape to demonstrate generalization and inheritance.

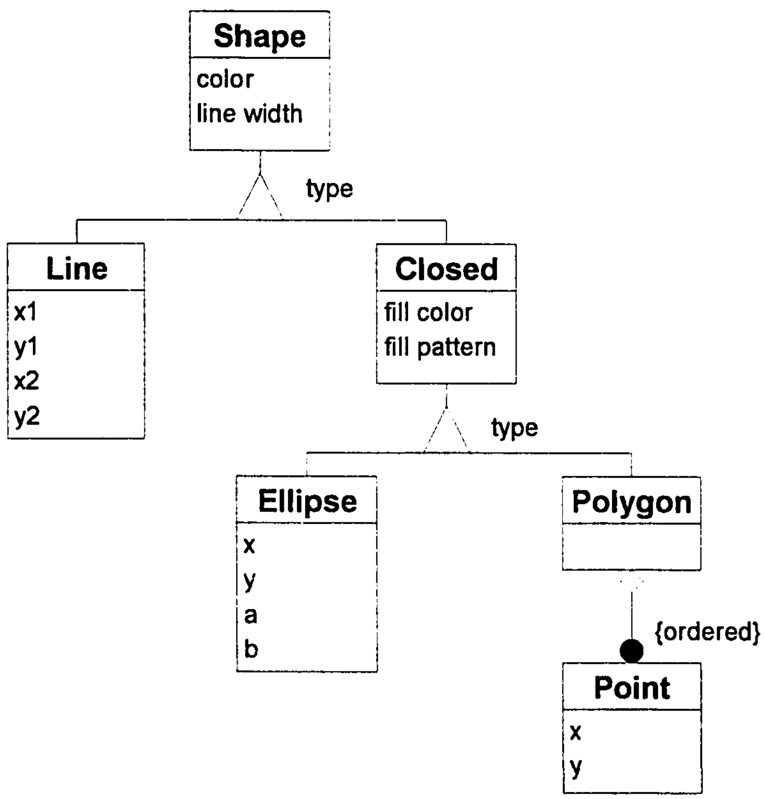


Figure 6 : Generalization and Inheritance Example

The notation for generalization is a triangle connecting a superclass to its subclasses. The superclass is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle. The words written next to the triangles in the diagram are discriminators. A *discriminator* is an attribute of enumeration type that indicates which property of an object is being abstracted by a particular generalization relationship. Only one property should be discriminated at once.

In our example, all shapes have a color and line width. Shapes can be lines, ellipses, or polygons, each with their own parameter. A polygon consists of an ordered list of vertices, shown as an aggregation of many points. Ellipses and polygons are both closed shapes, which have a fill color and a fill pattern. Lines are one dimensional and cannot be filled. Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered. The *ordering* is an inherent part of the association. An ordered set of objects on the "many" end of an association is indicated by writing "{ordered}" next to the multiplicity dot for the role.

2-5-2 The Dynamic Model

The *dynamic model* describes those aspects of a system concerned with time and the sequencing of operations. The dynamic model captures *control*, that aspect of a system that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented. The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and event sequences permitted in a system for one class of objects. Actions in the state diagrams correspond to functions from the functional model. Events in a state diagram become operations on objects in the object model.

An *event* is something that happens at a point in time, such as a user double-clicks a button. An event is a one-way transmission of information from one object to another. It is not like a subroutine call that returns a value. An object sending an event to another object may expect a reply, but the reply is a separate event under the control of the second object. Events are grouped into *event classes* and give each event class a name to indicate common structure and behavior. This structure is hierarchical, just as object class structure is hierarchical. The time at which an event occurs is an implicit attribute of all events.

2-5-3 The Functional Model

The *functional model* describes those aspects of a system concerned with transformations of values—functions, mappings, constraints, and functional dependencies. The functional model captures what a system does, without regard for how or when it is done. The functional model is represented with data flow diagrams. Data flow diagrams show the dependencies between values and the computation of output values from input values and functions, without regard for when or if the functions are executed. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the object model.

Data Flow Diagrams

The functional model consists of multiple data flow diagrams which specify the meaning of operations and constraints. A data flow diagram (DFD) shows the functional relationships of the values computed by a system, including input values, output values, and internal data stores. A data flow diagram is a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects. A data flow diagram contains *processes* that transform data, *data flows* that move data, *actor* objects that produce and consume data, and *data store* objects that store data passively. Figure 7 shows a data flow diagram for the display of an icon on a windowing system. The icon name and location are inputs to the diagram from an unspecified source. The data flow diagram shows the sequence of transformations performed, as well as the external values and objects that affect the computation.

Processes

A *process* transforms data values. The lowest-level processes are pure functions without side effects. An entire data flow graph is a high-level process. The functional model does not uniquely specify the results of a process with side effects. The functional model only indicates the possible functional paths; it does not show which path will actually occur. A process is drawn as an ellipse containing its name which is a description of the transformation. Each process has a fixed number of input and output data arrows, each of which carries a value of a given type. The diagram only shows the pattern of inputs and outputs. The computation of output values from input values must also be specified. A high-level process can be expanded into an entire data flow diagram. Eventually the

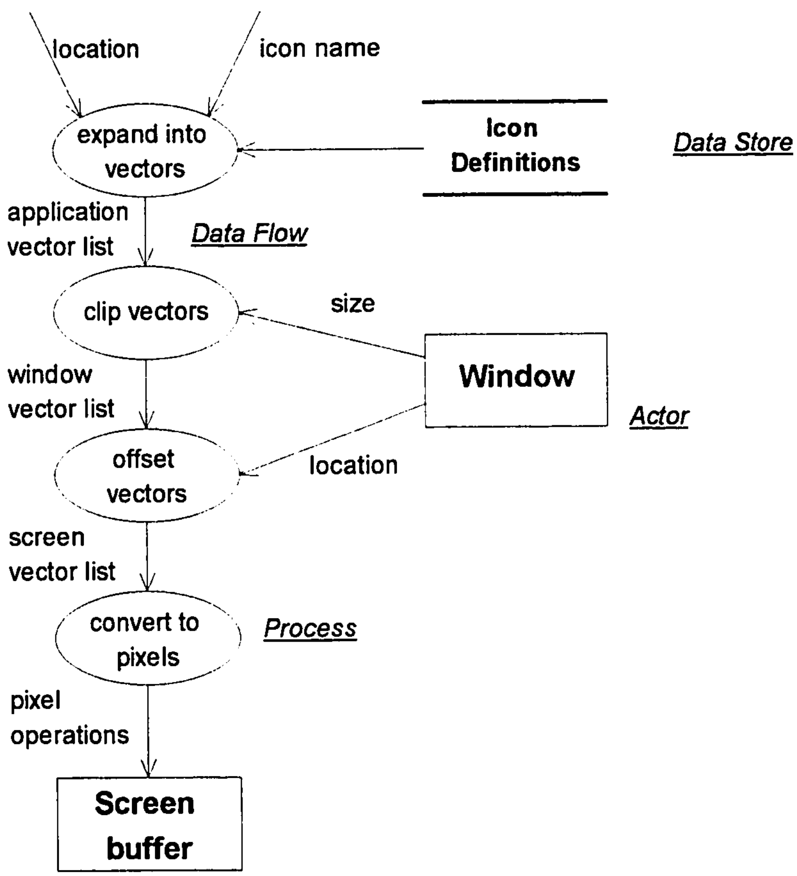


Figure 7 : Data Flow Diagram Example

recursion must stop, and the atomic processes must be described directly, in natural language or by some other means. Processes are implemented as methods of operations on object classes.

Data Flows

A *data flow* connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. The value is not changed by the data flow. A data flow is drawn as an arrow between the producer and the consumer of the data value. The arrow is labeled with a description of the data, usually its name or type. The same value can be sent to several places; this is indicated by a fork with several arrows emerging from it. Sometimes an aggregate data value is split into its components, each of which goes to a different process. This is shown by a fork in the path in which each outgoing arrow is labeled with the name of its component.

Actors & Data Stores

An *actor* is an active object that drives the data flow graph by producing or consuming values. Actors are attached to the inputs and outputs of a data flow graph. The actors lie on a boundary of the data flow graph but terminate the flow of data as sources and sinks of data, and so are sometimes called *terminators*. An actor is drawn as a rectangle to show that it is an object. A *data store* is a passive object within a data flow diagram that stores data for later access. A data store responds to requests to store and access data. A data store is drawn as a pair of parallel lines containing the name of the store.

CHAPTER 3

CAPE MODEL

Computer Aided Programming Education (CAPE) is a system model that, we believe, aids students and instructors of an introductory course to procedural programming languages. Teaching a programming language is not our main goal; rather we are interested in providing a computerized environment where students are aided in a program development process. We need to teach problem solving and program methodologies that can be used in dealing with almost any procedural programming language.

We approach the discussion from program development process view which is supported by the model. Section 1 discusses the philosophy behind our model. The object model is presented in section 2 using Object Modeling Technique (2-5-1). CAPE consists of two modules: a Problem Definition module and an Algorithm Construction module. The problem definition module consists of a description component and a set of exercises. The algorithm construction module consists of a block of statements, storage requirements, and a programming language code representation. CAPE dynamics are highlighted with its storage declaration and code derivation methods. The algorithm's dynamic model is presented in section 3 using our own developed convention.

3-1 Model Philosophy

Program development process is used to approach CAPE's philosophy. The model addresses each phase of the process by providing a separate module for each one. The model is designed for an academic environment where programming concepts are introduced in lectures and programming development process is performed during laboratory sessions. The model is also designed for a general purpose procedural programming language, currently defined to support Pascal and FORTRAN.

3-1-1 Program Development Process

The task is to write a program that solves a given problem. Given an initial problem description, a plan must be developed in which a computer can be used to solve the problem, and then describe this plan very precisely in some procedural programming language. This process has three distinct phases: understand the problem, design the solution, and code the program. Although these phases should occur roughly in the order listed, there is generally a good deal of overlap and backtracking each represents a distinct function that must be performed. If the problem is clarified and a strategy is designed, then writing the program statements is quite straightforward. It may still be time-consuming but it will not be the critical phase.

Understanding Phase

Understanding the problem and clarifying it is a major phase. It would seem reasonable that a clear and precise statement of the problem would be given, but it is easy to slightly misunderstand the exact requirements. You may end up writing a program that solves the wrong problem. A problem may be stated by someone who (presumably) understands what can be programmed and chooses problems to be only interestingly difficult. A substantial fraction of the clarification is concentrated on two key issues: *input* and *output*. What is the input data limit and source ? What is the content and order of output and its destination. A sample of input and output always help to understand a problem.

Designing Phase

Designing the solution phase involves programming creativity. The important thing is to separate the process of planning a solution from the task of describing that solution in a programming language. The plan for a program is generally called an "algorithm". An algorithm is a sequence of steps, whose execution solves a problem. Algorithms are written in whatever language and notation will make them understandable to the reader. Algorithms are written for people to read, rather than for computers and are therefore generally not written in a programming language.

Coding Phase

Coding the program is straight forward if designing phase has been adequately done. You have a detailed comment outline describing the action of each segment of the program. All what remains is to translate these detailed specifications into statements of whatever programming language is being employed. However, as details of program statements are developed, certain structures may not be quite right, thus back to design phase. Rarely a complete detailed translation would evolve without returning at least once to design phase.

3-1-2 CAPE Approach

A program is an algorithm that has been translated into a programming language. We regard a program as a specific implementation of an algorithm. The point is that the algorithm is written first, in an informal (but still precise) English-like language. The program comes later, by a process of translation rather than creation. The hardest part of program development is the planning and design of the program, rather than the translation of that design into the statements of a programming language. Students who have difficulty in programming often believe their problem is unfamiliarity with the language, when actually it is inability to find a suitable design for the program.

Most of the problems that are given to students to code during a laboratory session are classical programming problems. Their expected behavior is well known with different implementation techniques. For our purposes, a program is "correct" if there exists some set of test data for which the program will yield the correct answer. Instructors could apply the problem's test data to students developed solutions and review the output against expected results. An instructor can review solutions with incorrect results by reviewing the constructed algorithm or implemented code to identify the problem. Errors may be caused due to misunderstanding of the problem requirements, failure of the algorithm underlying the program, or misunderstanding the programming language.

When focus is changed to the coding and debugging of programs, it is advisable to map a skeleton code into an external environment that is intended for that purpose. CAPE provides the required environment to describe the problem and construct the algorithmic solution and then map it externally. There are a number of coding environments that are equipped with facilities to compile, execute, and debug programs written in a selected procedural language. CAPE is not intended to follow the same structure; instead it provides an environment where the emphasis is on designing a procedural solution that could be executed easily to reflect the desired abstracted actions.

Problem Definition

A problem can be described by presenting a sequence of textual or animated steps. Each problem has its own approach that makes it unique to be classified as a programming problem. Classical programming problems include average calculation, sorting numbers, and matrix multiplication. A number of exercises can be generated from a single problem. For example, calculate the average of ten numbers, of *N* numbers, or of numbers delimited by a zero, are exercises of calculating average problem. The essence of the problem is in every exercise with differing parameters. An exercise can have one or more sample input data with their expected results. A sample of input and output always help to understand a problem. An evaluation mechanism to execute a set of programs using the exercise sample's data would allow instructors easily identify incorrect solutions.

Algorithm Construction

To aid students in the planning and design of a program, we should provide them with a tool where they could construct such plan. Flowcharting is normally used in most cases, but it does not abstract language specific instructions. If we use a procedural programming language to express an algorithm then we are not abstracting the solution. Flowcharting is not considered as an effective tool to represent algorithms. Students can learn more by having an algorithmic environment that separates solution design from implementation issues. CAPE has four command statements: Set, Call, Get, and Put; and three structure statements: Simple, Select, and Repeat. We believe that these statements are sufficient to represent algorithms for introductory level students. The focus is towards using these statements in expressing steps of execution for every algorithm. Instructors can also use it during a lecture to explain algorithms and to have a consistent mechanism in developing programs.

Code Specification

Once the algorithm has been constructed, then comes the coding phase. CAPE is currently designed to map an algorithm or a set of algorithms into Fortran or Pascal code. The way CAPE is structured allows the incorporation of other procedural languages, such as C or PL/I, in the future. We are addressing coding by providing a facility to map an algorithm into a skeleton code of a selected programming language. The generated code is not expected to be free of syntax errors since the specification of algorithm's expressions is not constrained to a specific language. The algorithm designer would follow a valid language specific method in coding expressions or naming variables while inserting statements. Gained benefit is to generate code that is free of syntax errors. There is a requirement that students must be able to practice their coding skills at certain times. CAPE allows the mapping of a selected set of statements as a comment for the student to complete. The derivation of code is totally based on the algorithm specification. An exercise solution might consists of multiple algorithms mapped into one source code. Constructing a solution allows the selection of algorithms to be merged together into an exercise solution coded in a selected language.

3-2 The Object Model

CAPE treats itself as a course consisting of a set of problems and a set of students. Each problem consist of a description and a set of ordered exercises. Students are asked to develop a solution for a given exercise. The exercise's general problem can be reviewed by going through a number of steps that contains information describing the problem. Each exercise has a set of sample input/output data to help students in understanding the requirements. To develop a solution, students must construct an algorithm, or a set of algorithms, that solves the exercise. These algorithms are mapped into a derived code object representing the solution in a selected programming language. Figure 8 provides a high level object diagram of CAPE model.

The object model is the main frame work around which the design is constructed. We have converted the actions and activities of a dynamic model and the processes of a functional model into operations attached to classes in the object model.

3-2-1 Data Dictionary

Course—is the introductory course to programming we are supporting. A number of students could be enrolled in the course that has a set of programming problems associated with it. Problems are ordered based on level of difficulty where students need not be ordered.

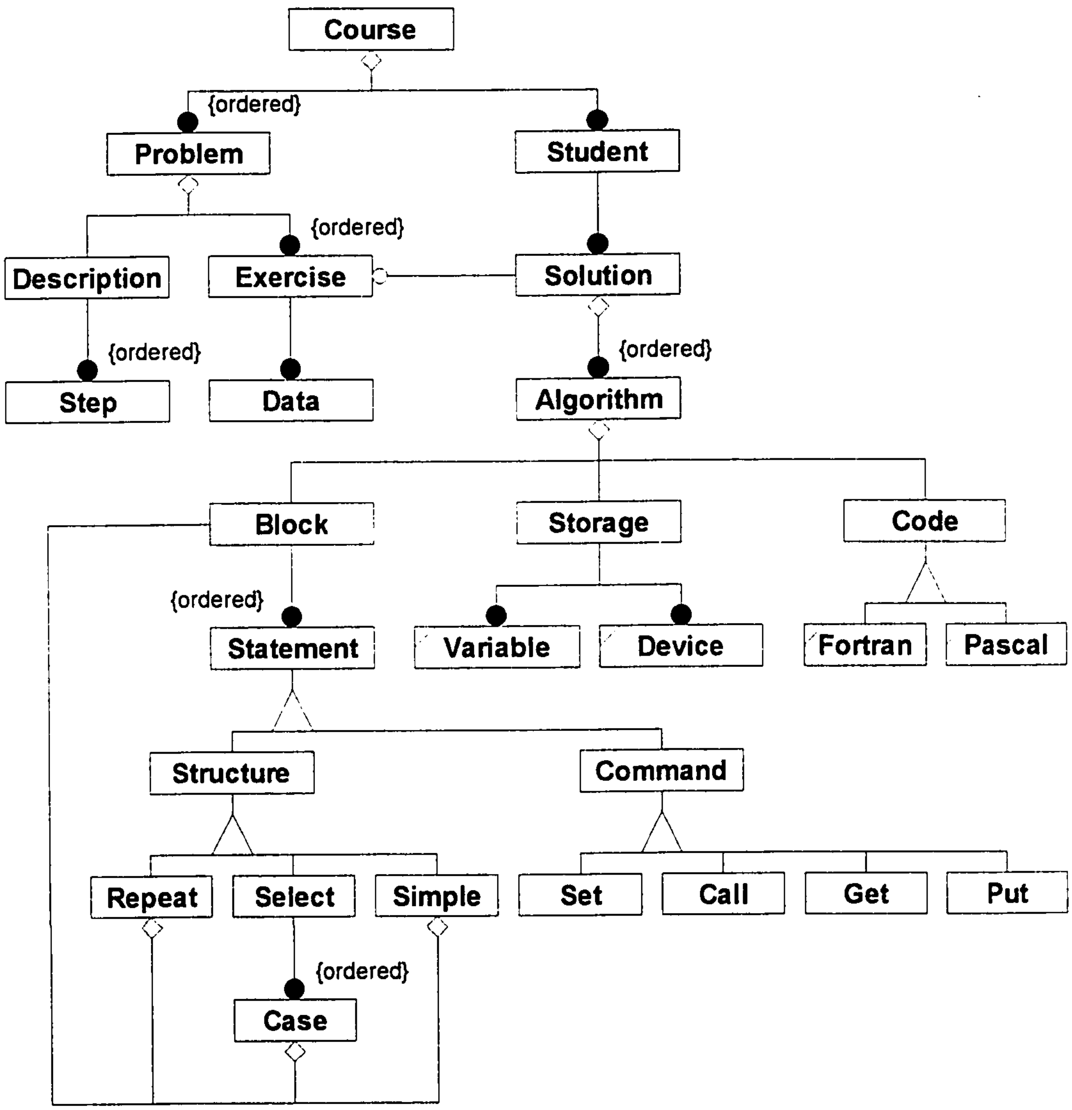


Figure 8 : CAPE Object Model

Problem—is a general or classical programming problem. It consists of a description and a set of exercises. The exercises are ordered by difficulty level.

Description—contains an outline of the key concepts and facts to be included for the problem. A description is made up of pedagogical steps, whose successive unrolling will constitute the covering of the problem's concept.

Step—refers to a pedagogical point that a professor is making to explain a concept. A step can be a mixture of text, graphics, and animation.

Exercise—has the general nature of the problem but with additional requirements. The exercise is for the student to solve where a problem is for the student to review. It contains its description and a set of sample data. There is no order between the data.

Data—is used to specify a sample input/output data for an exercise. Each sample input data has a corresponding expected output data.

Student—a learner who is enrolled in the course. He develops solutions for given exercises. During a course session, a student may own a set of solutions consisting of algorithms.

Solution—is a code representation of a program that consists of a main algorithm and possible number of algorithms used as functions or procedures. A solution could be associated to a single exercise or non. Code is represented in a single attribute derived by the content of the associated algorithm(s).

Algorithm—consists of a block of statements, storage definitions, and a derived code representing the algorithm's in selected procedural language. An algorithm could be used as main, function, or procedure when mapped to code.

Block—contains an ordered set of algorithmic statements. Structure statements contain their own block.

Statement—is an algorithmic construct that indicates the operation to be performed by the computer. It can define its storage requirements and map its code representation when requested.

Command Statement—a subclass of statement that does not contain a block. A *Set* command is used to assign an expression to a variable. *Call* is used to call external algorithm by passing a set of expressions to receive back a set of variables. *Get* is used to read values of variable(s) from an external device. *Put* is used to send an expression(s) to an external device.

Structure Statement—a subclass of statement that contains a block. *Simple* structure consists of a block that is executed once. *Repeat* structure consists of a block that is executed repeatedly for a controlled number of times. *Select* structure consists of a set of cases. A *Case* contains a condition and an associated block that gets executed when the condition is logically true.

Storage—is an object containing devices and variables used in the algorithm's block. These objects are derived by algorithm usage and its block of statements.

Variable—is a storage location used by the algorithm. A variable is created when a statement requests its declaration. A variable has a name, source, size, and type. A variable source could be Local, Device, Input parameter, or Output parameter. Input/Output parameters are defined for an algorithm used as a function or procedure.

Device—is an external source used for Input or Output. A device attributes include name, usage, type, and file name. 'Input' and 'Output' names are used to indicate terminal device.

Code—is a representation of the algorithm in a procedural programming language. A code object has header, declaration, and body attributes. These attributes are derived based on the algorithm's block and storage objects. *Fortran* is a subclass of Code that represents the code using FORTRAN language. *Pascal* is a subclass of Code that represents the code using PASCAL language.

System Modules

A module is a set of classes that captures some logical subset of the entire model. The same class may be referenced in different modules which is the used mechanism for binding modules together. CAPE consists of two modules: problem definition and algorithm construction modules. The problem definition module provides students with an environment to review and understand a given problem and its exercises. The algorithm construction module contains the required objects to construct an algorithmic solution to a problem. Once the design is completed, a mapping mechanism is applied to specify the code in a selected language. The code could be evaluated using the exercise's input data.

3-2-2 Problem Definition Module

A problem definition module consists of a set of classical programming problems, such as calculating average, sorting, or matrix multiplication. A problem consists of a description component and a set of exercises. The description container has an ordered set of pedagogical steps used in explaining the problem. A number of exercises can be created from a single problem. The essence of the problem is in every exercise with differing parameters. Figure 9 shows Problem Definition module.

Description Component

The manner in which the problem is described reflect the teaching strategy of the professor. To effectively support various strategies, we provide further decomposition of the description level. A description is made up of pedagogical steps, whose successive unrolling will constitute the covering of the concept. Pedagogical steps are referring to points that a professor is making to explain a concept. A step can be a mixture of text, graphics, and animation. Developing Step objects is left open for external environments that are designed to develop courseware and on-line experiments. For educators and researchers who would like to create sophisticated courseware, Authorware Professional is a very powerful tool that merits serious consideration [19]. Authorware Professional is an icon-based, object-oriented authoring system.

The description contains an outline of the key concepts and facts to be included for the problem. It states in simple terms what the student should be able to do after the problem has been reviewed. This may be done either in the form of simple statements or questions

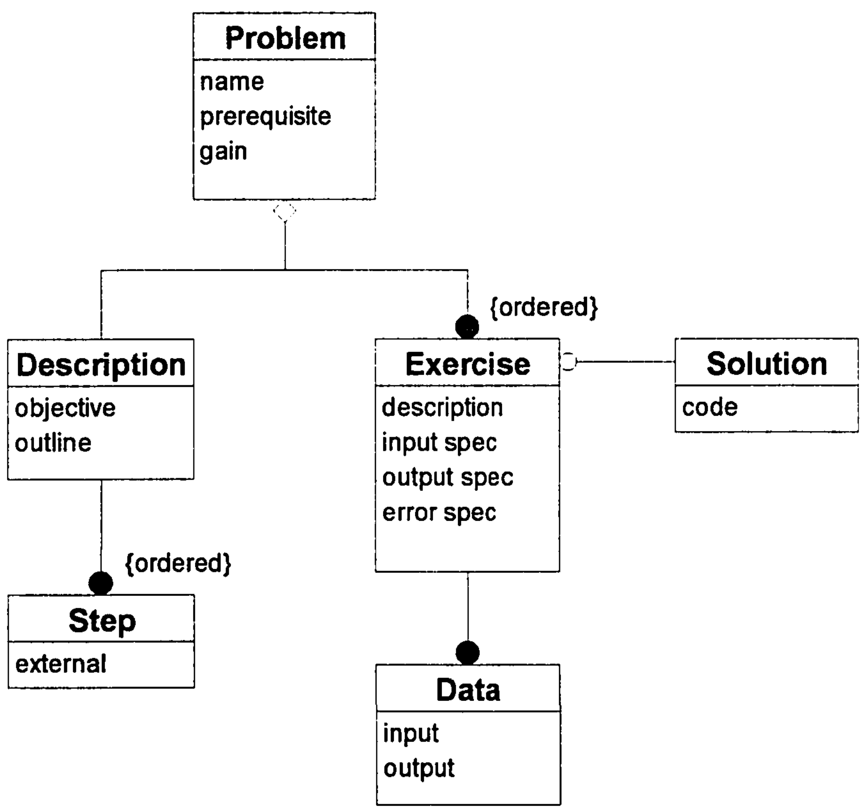


Figure 9 : Problem Definition Module

posed to challenge the student. Sometimes a simple reminder of previous problems or programming technique will help in understanding the problem. Brief example and analogies from material the student is likely to know are helpful. Present the information regarding the problem in organized, systematic, small segments.

Exercise Component

An exercise has the inherited nature of the problem with an added constraint or requirement. For example, average calculation problem may have an exercise that requires a constant number of input, a variable number of input, or a set of numbers delimited by a constant value. The exercise itself has its own description and the specification of input, output, and error handling. To clarify the requirements further, a set of sample input data is provided with their expected result. The sample data should be designed to test and increase the understanding of details of the problem. Input specification provides information about the nature of input. What is the input format and order? What are the limits of volume and how will the end of the input be recognized? What are the limits on input value? These question derive the content of input description. Output specification provides information about the nature of output. What is the content, format, and order of output? What titling is appropriate? What limits on volume may be expected? Errors specification provides editing requirements and handling. What types of errors both in input and in processing must the program guard against and what action should be taken when they are encountered?

3-2-3 Algorithm Construction Module

Algorithm construction module provides an environment to design a procedural solution for a specific task. The module separates the process of planning a solution from the task of describing the solution in a programming language. The plan for a program is generally called an "algorithm" which is a sequence of steps whose execution solves a problem. CAPE has a simple set of statements used to express steps of execution. We believe that these statements are sufficient to represent algorithms at an introductory level. The focus is towards using these statements in expressing steps of execution for every algorithm. CAPE statements could be used in a lecture in explaining some programming concepts.

Figure 10 shows the algorithm construction module. An algorithm object consists of a block of statements, derived storage object, and a derived code object. A block object contains a set of ordered statements that represent steps of execution. A statement can be either a command or a structure that contains another block. These statements derive the content of storage container by sending requests to declare variables or devices used to support the statement. A derived object is indicated with a slash or diagonal line on the corner of the class box. A storage object contains a set of devices and variables used by the algorithm.

Algorithm Object

Algorithm object attributes include name, description, usage, date created, and owner. The name and short description are sufficient to identify an algorithm. Date created and owner

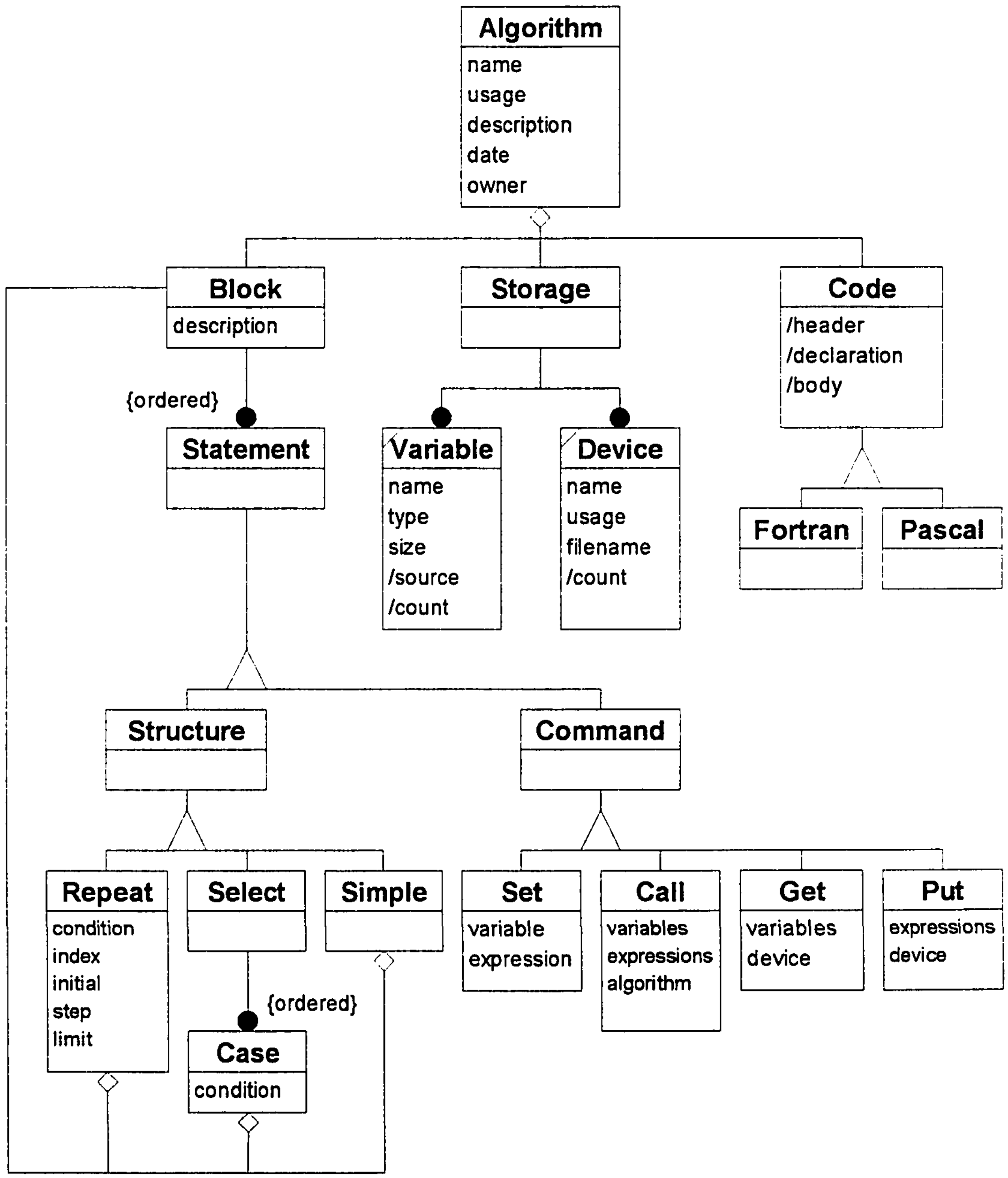


Figure 10 : Algorithm Construction Module

attributes are taken from the run-time system when the algorithm object is created. Algorithm usage indicates whether the algorithm is used as main, function, or subroutine. Using an algorithm as main indicates that there are no input or output parameters passed to the algorithm. When the algorithm is used as a function or a procedure it might receive a set of input/output parameters. Input and output parameters are represented as variables in storage with source set to 'Input' and 'Output' respectively. An algorithm used as a function would have only one variable named after the algorithm defined in storage with source set to 'Output'. A procedure might have multiple input/output parameters.

Block Object

A block contains a set of ordered statements which represent algorithm steps. A block *description* abstracts its statements operation. A statement can either be a command, which is the lowest statement of a block, or a structure, which contains another block.

Command Statement

A command statement can either be a Set, Call, Get, and Put commands. These commands are the lowest level for an algorithmic statement. *Variable* attribute contains a name of a single variable where *Variable(s)* attribute indicate a list of variable names separated by commas. A variable name may contain '(' when used as an array. An *expression* could be composed of constants, variables, functions, or binary expression separated by an operator. *Expression(s)* is a list of expressions separated by commas. Each command has an affect on storage requirements when manipulated. Conditions set on variables and devices usage's are maintained by storage object.

Set Command

A Set command is used to assign an expression to a single variable. Both attributes must be specified to have a valid command. When the statement is added to a selected block, a message is sent to storage object to declare *variable* with 'Local' source. An expression may not be set to a variable declared earlier with 'Input' or 'Device' source.

Call Command

Call is used to communicate with other algorithms by passing them a list of *expression(s)* and receiving back values for a list of *variable(s)*. The called *algorithm* name must be provided where the other attributes are optional. The list of variables or expressions must be separated by commas if more than one is specified. When the statement is added to a selected block, a message is sent to storage object to declare variable(s) with 'Local' source. The list of variables may not contain a variable declared earlier with 'Input' or 'Device' source.

Get Command

Get is used to scan the input stream to get the values of listed variable(s). *Device* attribute is used to indicate the source of the command. 'Input' device name is recognized as run-time or terminal device where other names indicate a read operation from an external file. When the statement is added to a selected block, two messages are sent to storage object: one to declare *variable(s)* with 'Device' source and the other to declare *device* with 'Input' usage. The list of variables may not contain a variable declared earlier other than 'Device' source.

Put Command

Put is used to transfer a list of expression(s) to an external device. *Device* attribute is used to indicate the destination of the command. 'Output' device names is recognized as run-time or terminal device where other names indicate a write operation to an external file. When the statement is added to a selected block, a message is sent to storage object to declare *device* with 'Output' usage.

Structure Statement

A structure statement can either be a Simple, Select, or Repeat structure. A simple structure consists of a block that is executed once. A select structure contains a set of cases, each consists of a block that is executed when the case's condition is logically true. A repeat structure consists of a block that gets repeatedly executed while a condition is logically true or for an indexed number of times.

Simple Structure

A simple structure is used to group a set of statements within a block. The block's description abstracts the operation of its statements. There are no messages sent to store objects since no control parameters are used.

Select Structure

A select structure consists of a set of ordered cases. Each case has a *condition* attribute and an associated block that is executed when the condition is logically true. A case object with condition set to blank indicates the 'Otherwise' condition in an IF construct.

Repeat Structure

A repeat structure consists of a block that executes a repeated number of times. Repetition controls are the attributes of the structure. If a *condition* is specified then the block is executed while that condition is logically true. A block may be repeated while varying an *index* from an *initial* value to a *limit* value incrementing by *step*. Initial, limit, and step may be any expression. When the structure is added to a selected block, a message is sent to storage object to declare *index* variable with 'Local' source.

Storage Component

A storage object consists of a set of variables and devices used by the algorithm. They are dynamically created when a statement is inserted into a block. Viewing these derived entities is useful while constructing the algorithm. They are also used as a base in deriving the solution code in code specification module. Before a variable or a device is declared, a check is made to determine if the declaration was already in place and whether its usage is consistent. A derived attribute, *count*, is used to keep track of the number of statements have declared the object. When a remove-variable request is received, the count is decremented to accommodate other statements using it. When a count reaches zero, then the object is deleted. A number of methods, discussed in the dynamic model, are used to maintain the declaration of storage.

Variable Object

A variable object is created when a statement requests storage object to do so. Declaration request would not indicate the type, a default value '{type}' is used. The owner of the algorithm may access the list of variables to define their type at a later stage on the development such as saving the algorithm or mapping it to code. Variable source can be either Local, Device, Input, or Output. Local source variables are the ones used in Set, Call, and Repeat statements. Device source variables are the ones used in a Get statement. Input source variables are input parameters passed to the algorithm when used as a function or a procedure. Output source variables are the returned variable parameters when the algorithm is used as a procedure or a function. In the case of a function, only one returned variable is created, named after algorithm name, with Output source. Variables written to a device using a Put statement are not classified as output variables. In fact, no message is sent to declare variables from a Put statement, only the declaration of its output device.

Device Object

A device object is created when a statement requests storage object to do so. Get and Put statements request the creation of a device object. Device name of 'Input' and 'Output' are recognized as standard terminal device. Other names indicate an external device, *filename* attribute is used to identify it. Device usage is either 'Input', for Get statement, or 'Output', for Put statement. The owner of the algorithm may access the list of devices to define their filename at a later stage on the development, such as saving the algorithm or mapping it to code.

3-5 Algorithm's Dynamic Model

The object model has described the possible patterns of objects, attributes, and links that can exist in it. The attribute values and links held by an object are called its *state*. Over time, the objects stimulate each other, resulting in a series of changes to their states. For example, adding a statement to a block changes the block's state. An individual stimulus from one object to another is an *event*. An event flow diagram summarizes events between objects without regard for sequence. The event flow diagram is a dynamic counterpart to an object diagram. Paths in the object diagram shows possible information flows where paths in the event flow diagram show possible control flows. Figure 11 shows the algorithm's dynamic model.

We have developed our own convention in describing an object method. This convention is presented next. Part 2 discusses the handling of an inserted or deleted statement that affects storage declarations. Variables are created dynamically in storage object with a default type and size to be specified by the owner. Now that CAPE has a set of statements and storage requirements, a programming language code could be derived. Code object has header, declaration, and body attributes representing the corresponding algorithm's code. Part 3 describes code generation events, regardless of the selected programming language. Part 4 and 5 specify CAPE's objects methods used to generate Pascal and Fortran code respectively. Part 6 discusses the method used to construct code using multiple algorithms.

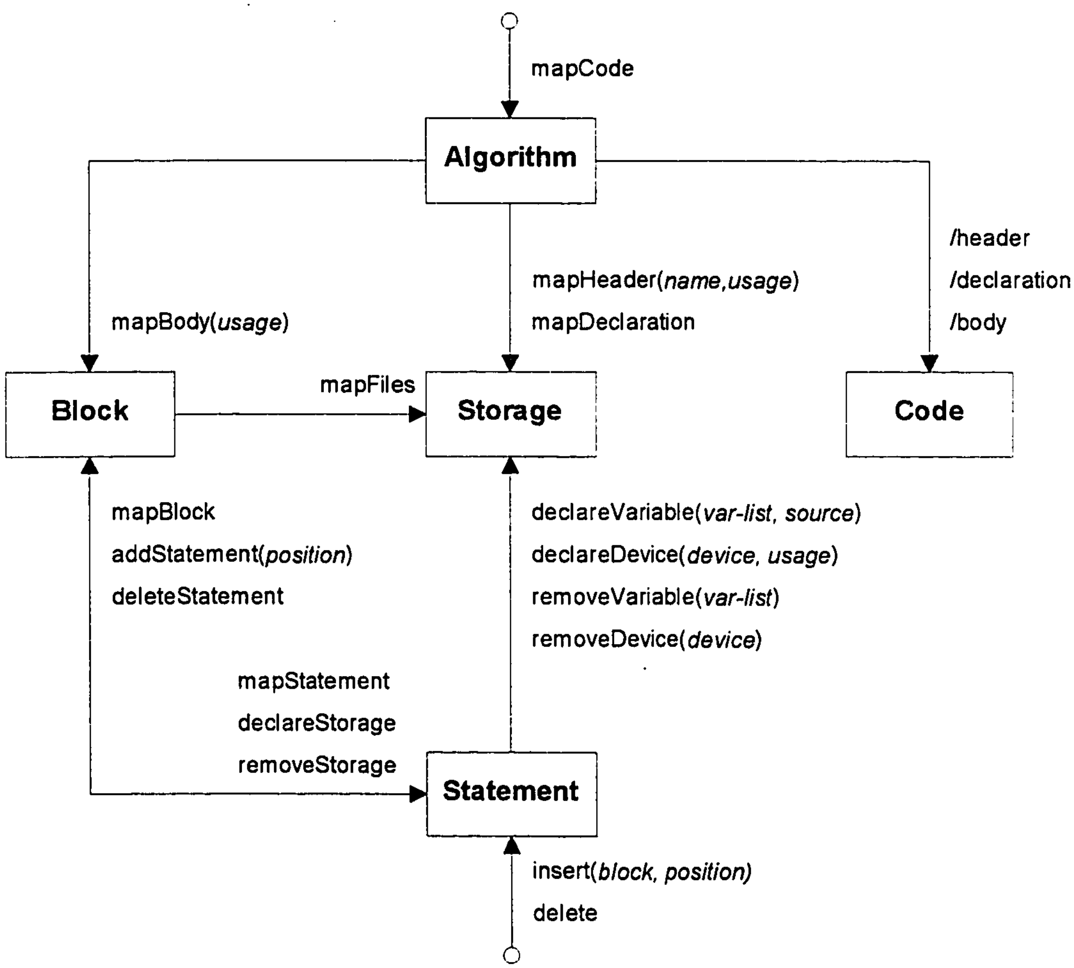


Figure 11 : Algorithm's Dynamic Model

3-5-1 Method's Diagram Convention

We have developed our own convention in describing an object method. Mapping algorithm objects into code requires the exchange of events that return text code in one or more lines, which is the reason why we had to develop our own convention. The convention is also used in describing control methods used largely by storage derivation process. Figure 12 shows our used symbols and their structure.

A method is identified by its, underlined, object name and method, followed by input parameters, if any. The start and end of the method are identified using a triangle and an inverted triangle respectively. Repetition control is indicated next to the start symbol. The only repetition control used in CAPE is 'For every ...', indicating a scan of objects. Repetition is indicated by connecting start and end symbols. Operations and conditional flow is indicated using the standard rectangle and diamond shapes. Messages sent to self or other objects are indicated using a shaded operation symbol.

Methods used to map code are expected to return text code, one or more lines, representing their objects. These methods are identified with '*text*' symbol after the method's name. Each statement has a skeleton representation in every language. We retain the syntax of each statement in our mapping methods to help graphically identify differences between languages. A syntax of a language contains certain operators and keywords, which are represented as a circle and a rounded rectangle. The method's returned text is constructed by concatenating returned values from an operation, message, keyword, or operator. Flow between terminals is indicated with a plain line, or with a line

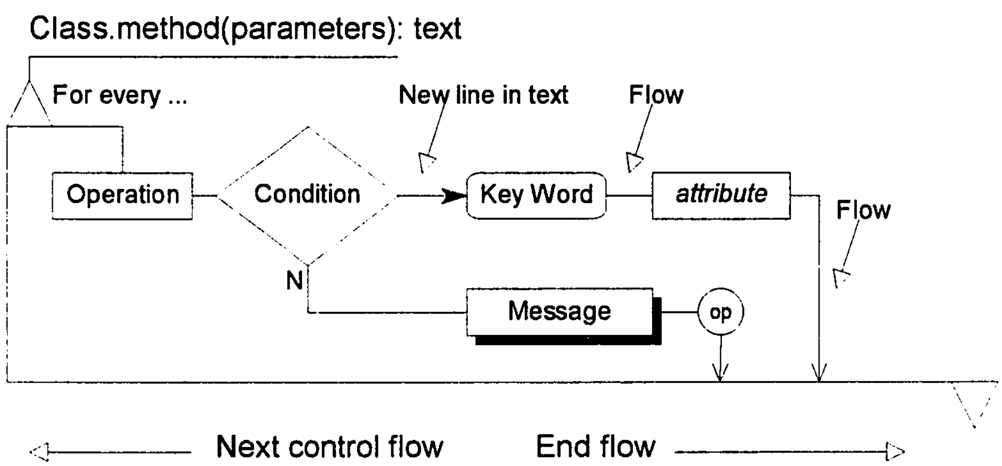


Figure 12 : Method's Diagram Convention

that ends with fine arrow head for added clarity. A new line is inserted in returned text when a flow line ends with a dark arrow-head.

3-5-2 Storage Derivation

Storage object consists of devices and variables used by the algorithm that are maintained dynamically. The content of storage is dependent on statements contained within the algorithm's block. Adding or deleting statements affect the content of storage by sending declare or remove messages to it. Storage derivation is not dependent on the selected language. Figure 13 shows the event flow diagram of storage derivation. In our next discussion we will show how a statement is triggered to communicate with storage object and then show how storage object handles declare and remove requests to maintain its set of variables and devices.

Adding and Deleting a Statement

Statement object sends *addStatement* message to a block object to add itself into a given position. Block object adds the statement to its container and sends back a *declareStorage* message to the statement object to derive its storage requirements. Similarly, to delete a statement a *deleteStatement* message is sent to block object which in turn sends back a *removeStorage* message to remove statement's related declarations. Figure 14 shows block methods to add and delete a statement. The only statements responding to declare or remove storage messages are: Set, Call, Get, Put, and Report. Figure 15 shows statement methods to declare storage followed by Figure 16 that shows statement methods to remove storage.

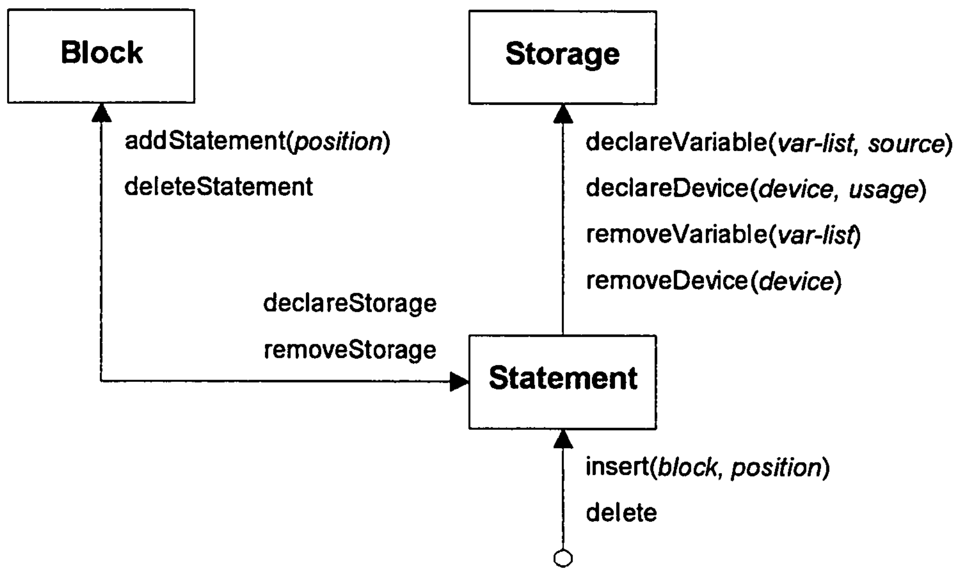


Figure 13 : Storage Derivation Events

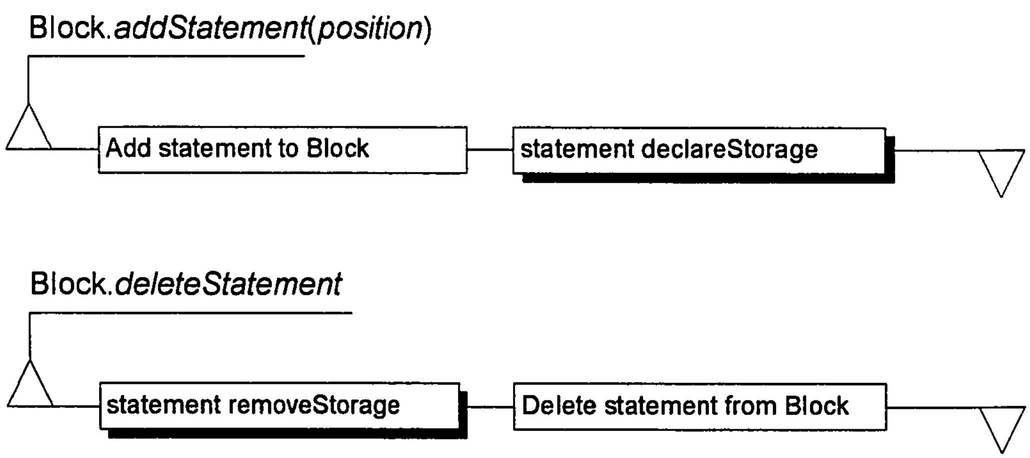


Figure 14 : Block Methods to Add/Delete a Statement

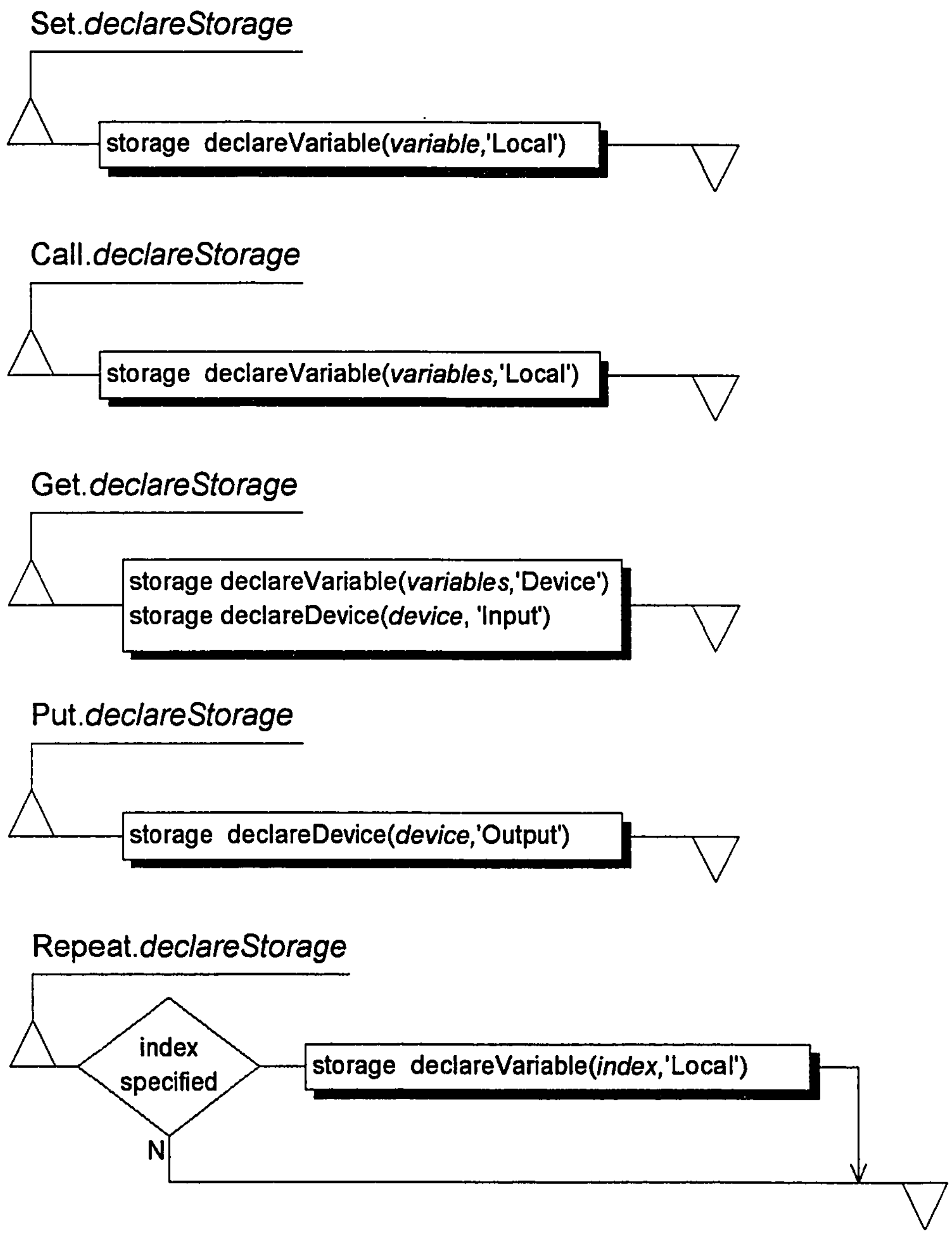


Figure 15 : Statement Methods to Declare Storage

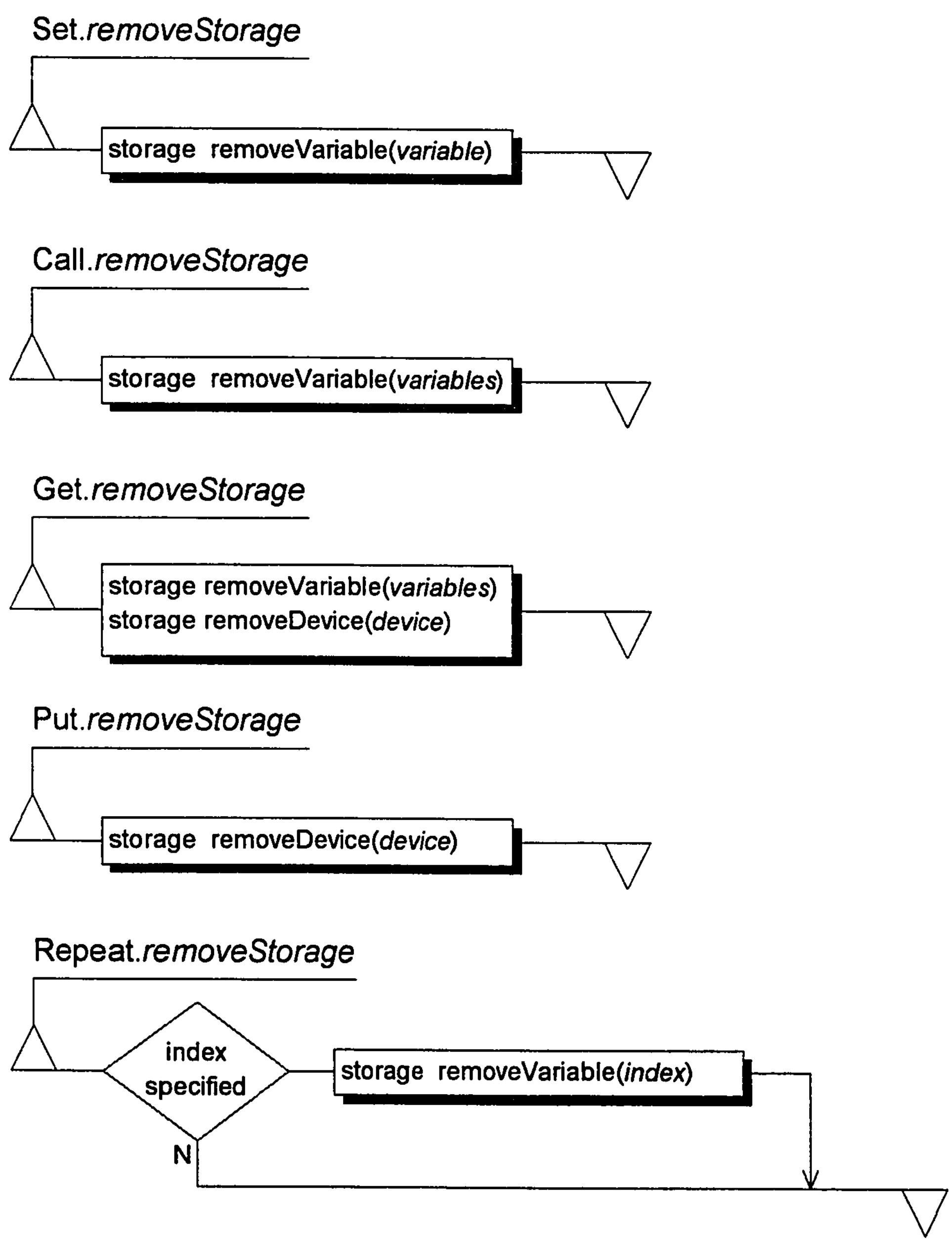


Figure 16 : Statement Methods to Remove Storage

Declaring Variables and Devices

Variable declaration method receives a variable or a list of variables separated by ',' to be declared as *source*. The list is examined to separate each provided variable to be processed individually. When a variable name contains '(' it indicates that it is an array with a size > 1 (or +1). Before adding the variable object, a check is made to verify if the variable has been declared. If the variable was declared earlier, its *count* is incremented by 1 and a validation message is returned if the variable's current usage is not consistent with the declared size or source. If the variable did not exist, then add it with default size, type, and count. It is acceptable to receive a request to declare a 'Local' variable that was declared earlier as 'Output' parameter. This case happens when an expression is set to a variable which is declared as an 'Output' to be returned by the algorithm. Figure 17 shows storage object *declareVariable* method.

Device declaration method receives a device and a usage indicator to be declared as a Device object. Usage specifies whether the device is used for input or output. A device name of 'Input' or 'Output' indicate a standard Terminal device where other names indicate external files. Before adding the device, a check is made to verify if the device has been declared. If device was created, then increment its counter by 1, otherwise create a new device. Figure 18 shows *declareDevice* method.

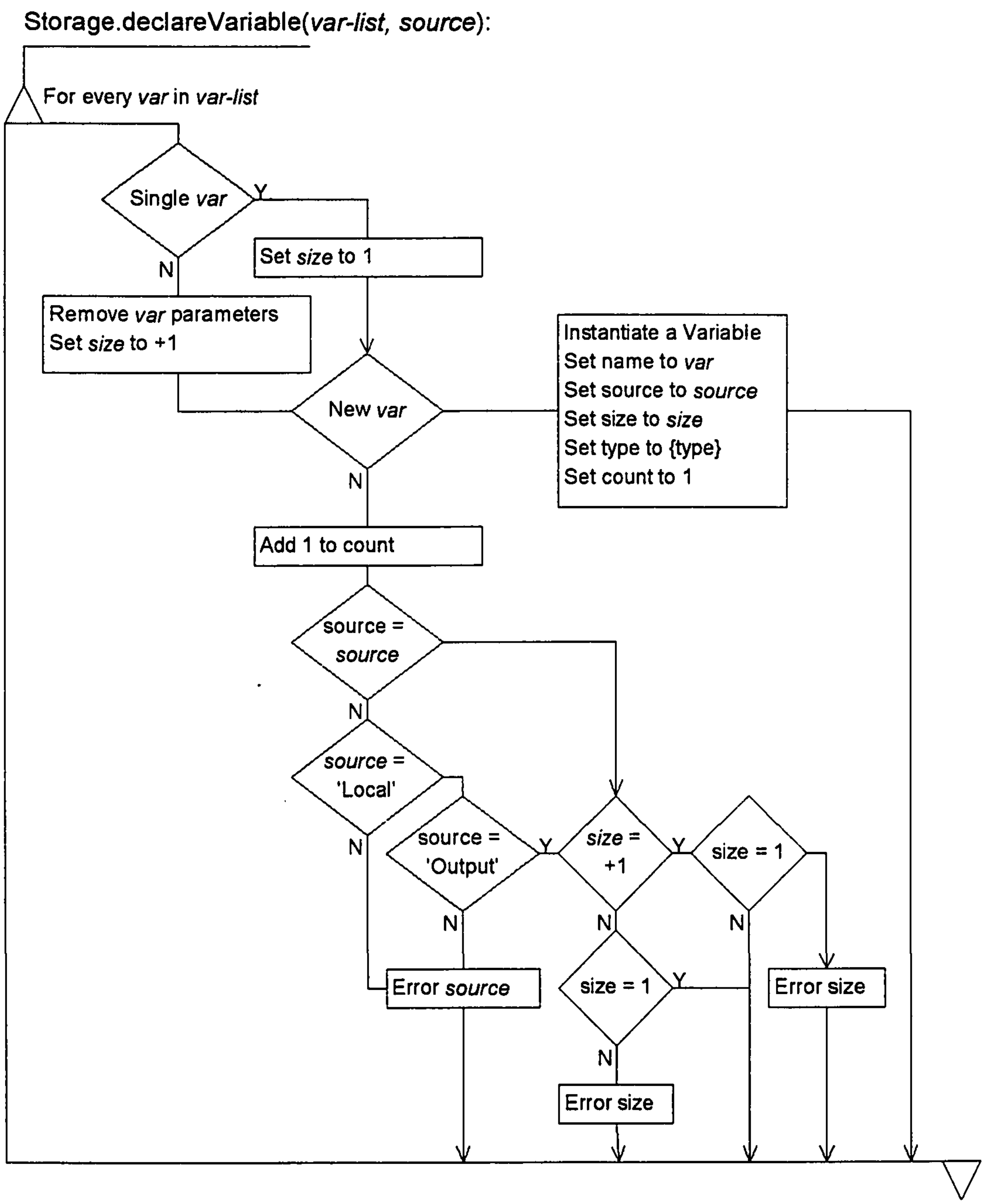


Figure 17 : Variable Declaration Method

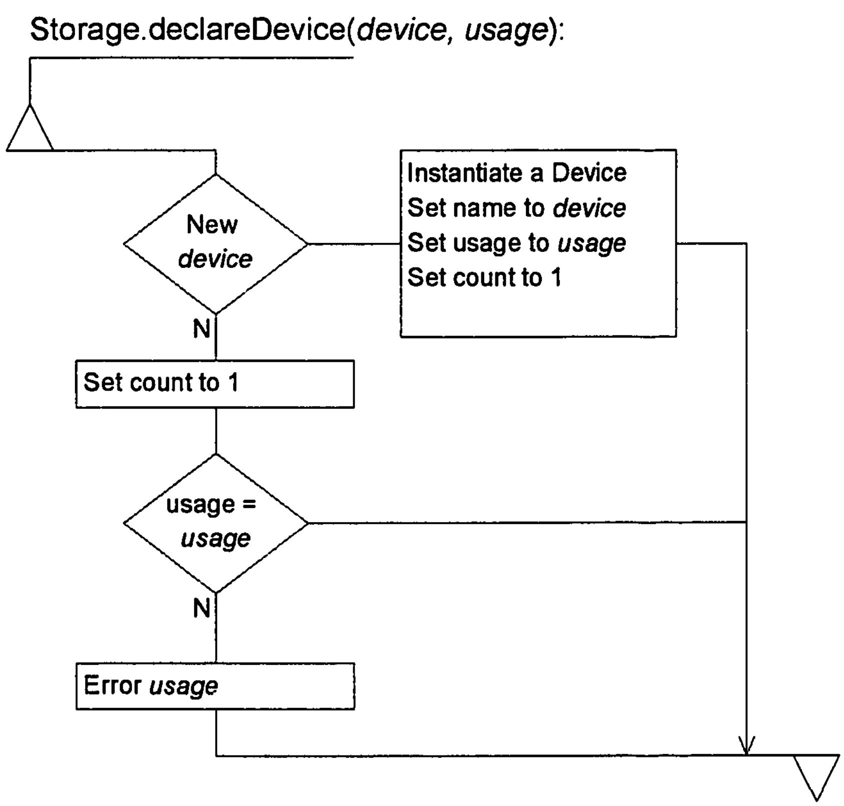


Figure 18 : Device Declaration Method

Removing Variables and Devices

Variable remove method receives a variable or a list of variables separated by ',' to be removed from storage. The list is examined to separate each provided variable to be processed individually. When a variable name contains '(', the enclosed expression is removed to identify the actual variable name. A declared variable has a counter that indicates the number of statements created the variable. To delete a variable, its counter is decremented by 1 and if it became zero then the variable object is removed, otherwise the variable is used by other statement(s). Figure 19 contains *removeVariable* method.

Device remove method receives a device to be removed from storage. A declared device has a counter that indicates the number of statements that are using the device. An existing device is deleted by decrementing its counter by 1, if it became zero then the device object is removed, otherwise the device is used by other statement(s). Figure 19 contains *removeDevice* method.

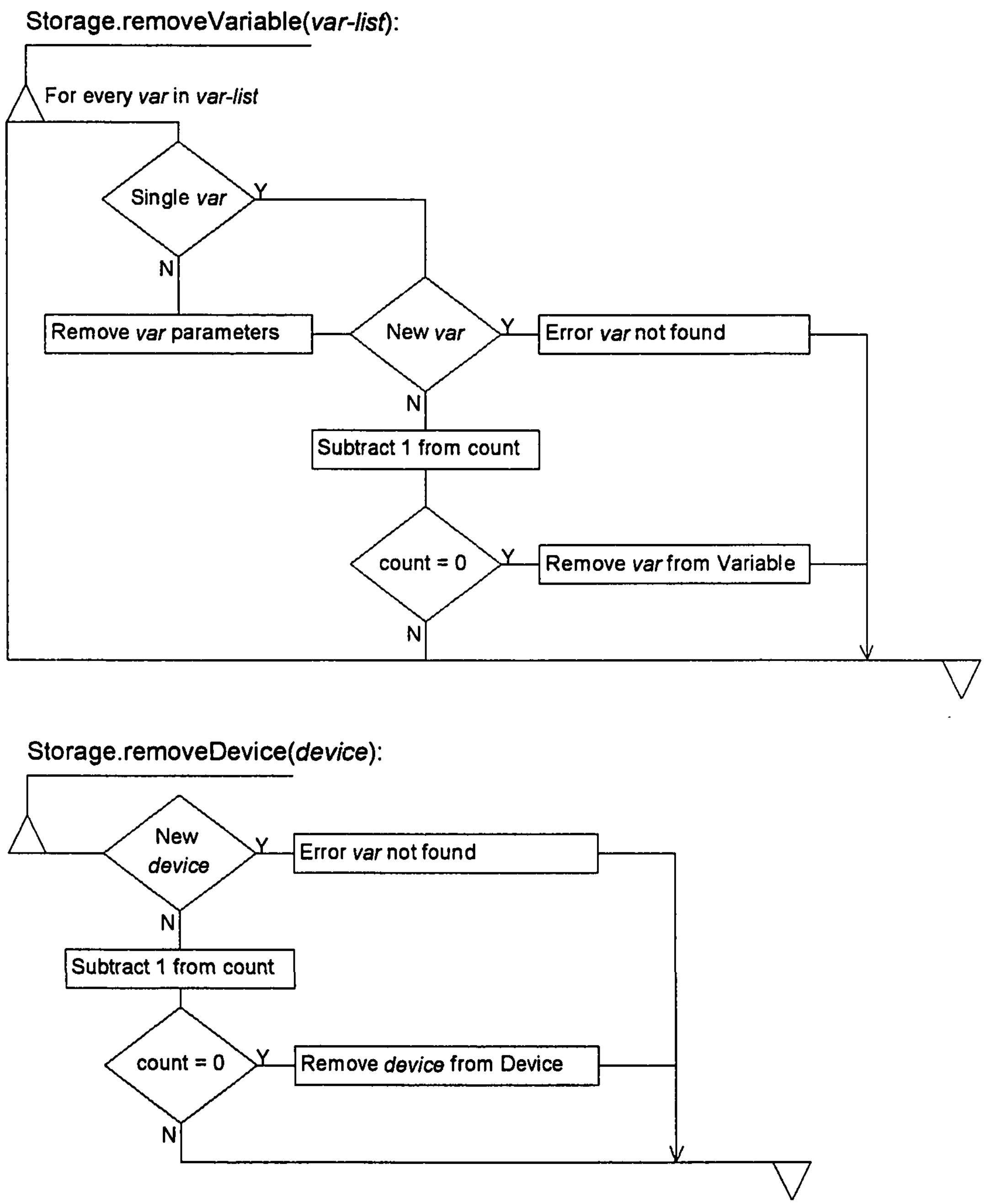


Figure 19 : Remove Variable and Device Methods

3-5-3 Code Derivation

Once the algorithm has been constructed, storage derivation process had created the required variables and devices with certain default settings. It is assumed at this stage that these attributes are accessible by the owner of the algorithm object. This is not a requirement for code derivation to proceed, but it would minimize the editing required to specify variable types, for example, manually in the generated code.

When the algorithm object receives a *mapCode* message to derive its code, it sends messages to storage and block objects to return the required texts. Figure 20 shows code derivation events and algorithm's map method. Storage object derives header and declaration attributes where block object derives body attribute. The algorithm's block object sends a message to its statements to return their code representation. A structure statement sends a message to its associated block.

Code Header

Storage object performs header derivation when *mapHeader(name, usage)* message is received. Header attribute is a single line that identifies the program. If the algorithm is used as a main program, certain identifications may be required by some languages. Pascal, for example, requires a program heading that identifies its name and list of device used, where Fortran does not. When the algorithm is used as a function or procedure, the algorithm name is used as the subprogram name followed by input and output parameters. A function algorithm contains a declared variable named after the algorithm's name with

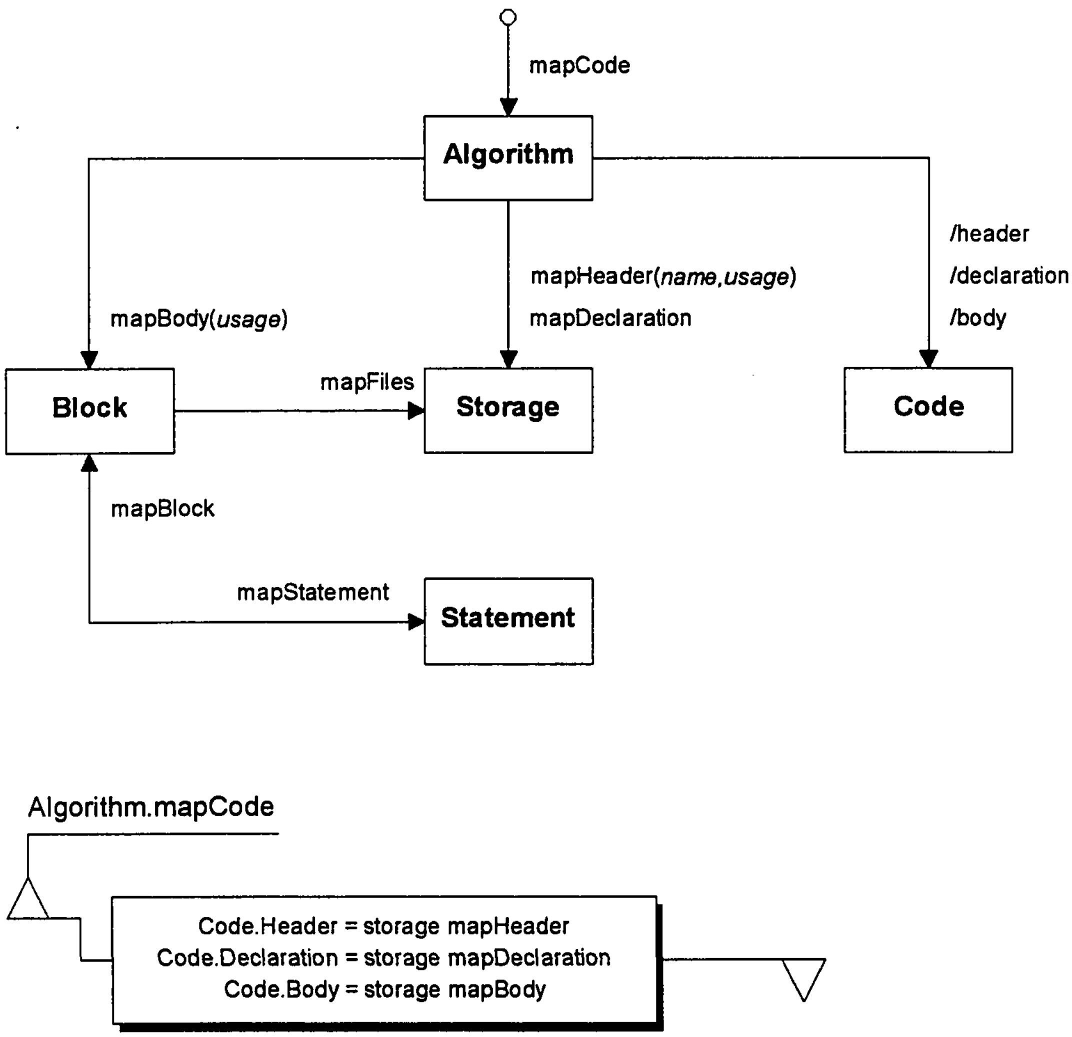


Figure 20 : Code Derivation Events and Method

source set to 'Output'; that variable's type is used as the function's type. Convention followed is to list input parameters prior to output parameters. Storage object uses four local methods to carry out header derivation. *deviceList* returns a list of all devices names. *varType(name)* returns the type of a given variable name. *inputList* returns a list of variable names and their types for variables with source='Input' where *outputList* returns a similar list for variables with source='Output'.

Code Declaration

Storage object performs declaration derivation when *mapDeclaration* message is received. It does not need to know about the algorithm's usage or name since it only derives the required declaration of variables and devices. Declaration attribute consists of multiple lines. Pascal requires the declaration of all devices other than 'Input' and 'Output', where Fortran does not. Pascal does not require the declaration of input/output variables since their declaration is included in the header. Fortran, on the other hand, requires the declaration of all used variables in the declaration section. Declared variables are either single or one dimensional array. Storage object uses two local methods to carry out declaration derivation. *decDevices* returns a list of devices other than input/output device. This method is not used in Fortran. *decVariables* returns a list of variables type declarations.

Code Body

The algorithm's block object performs body derivation when *mapBody(usage)* message is received. It first sends *openFiles* message to storage object to return any necessary devices initialization. A *mapStatement* message is then sent to every statement of the block to return its code representation. Command statements simply return their code representation to its calling block. On the other hand, Structure statements return their code representation after sending a *mapBlock* message to their associated block. The block builds its representation by sending *mapStatement* message to its statements. Select structure statement contains a set of cases that are mapped individually by sending *mapCondition* to each case in their specified order. Case objects send a *mapBlock* message to their associated block and returns the pieces of the selection construct to its Select statement.

3-5-4 Pascal Code Methods

The following is a list of figures describing methods used in deriving header, declaration, and body attributes of Code object using Pascal Language:

Figure	Methods Title
21	Header Derivation
22	Header Derivation Support
23	Declaration Derivation
24	Code Derivation
25	Command Statement Mapping
26	Structure Statement Mapping
27	Case Object Mapping
28	Block Object Mapping

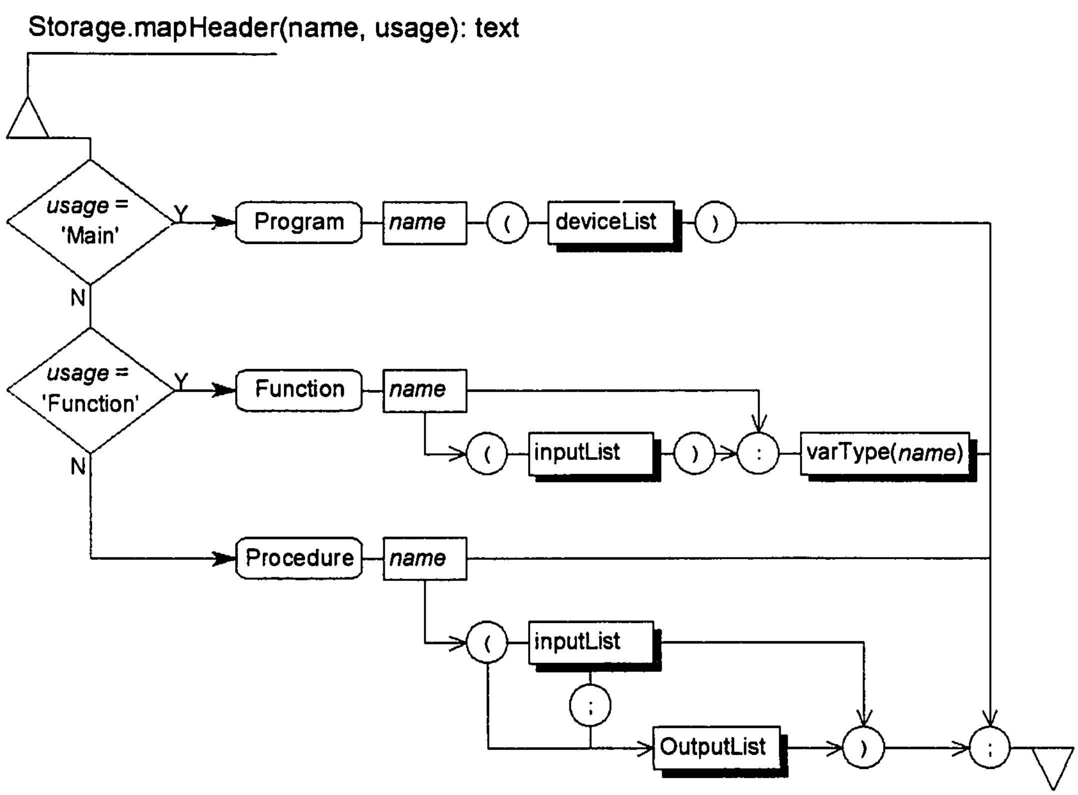


Figure 21 : Header Derivation Method (Pascal)

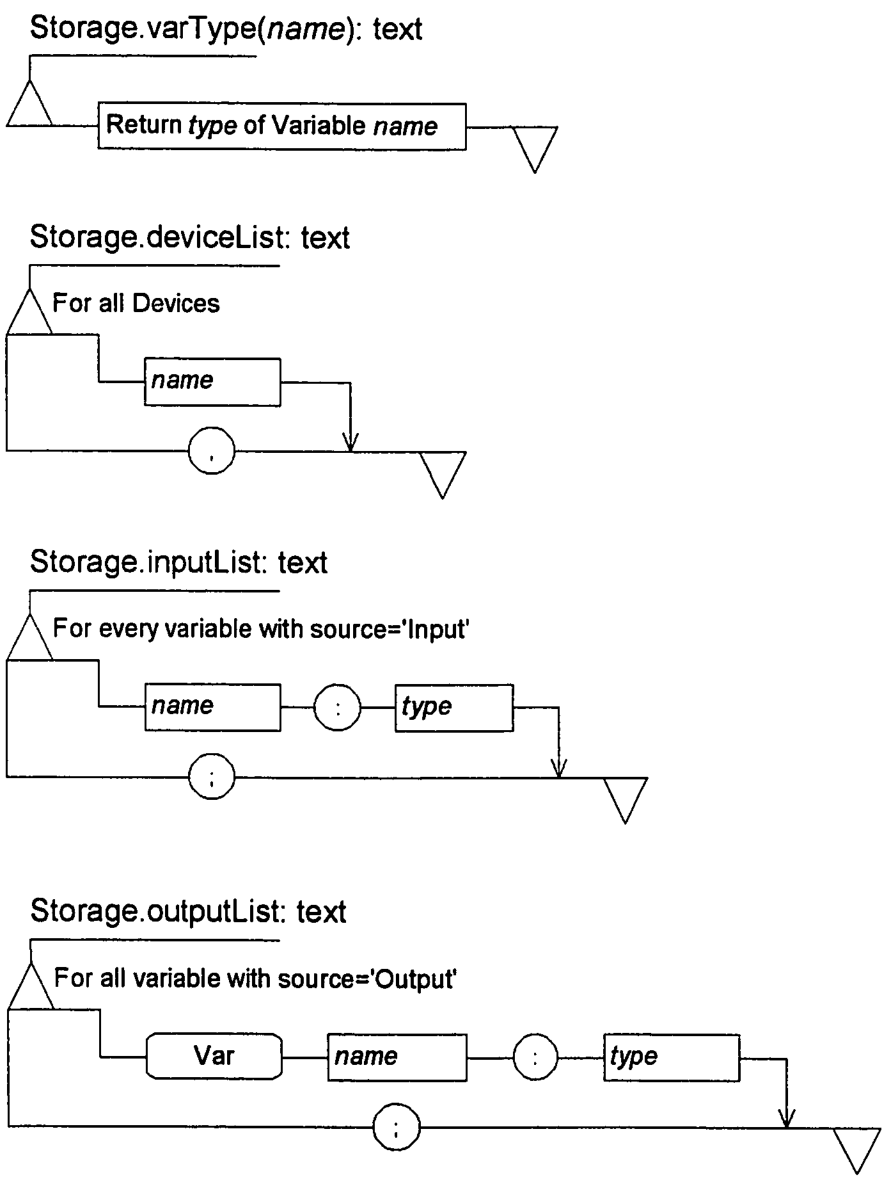


Figure 22 : Header Derivation Support Methods (Pascal)

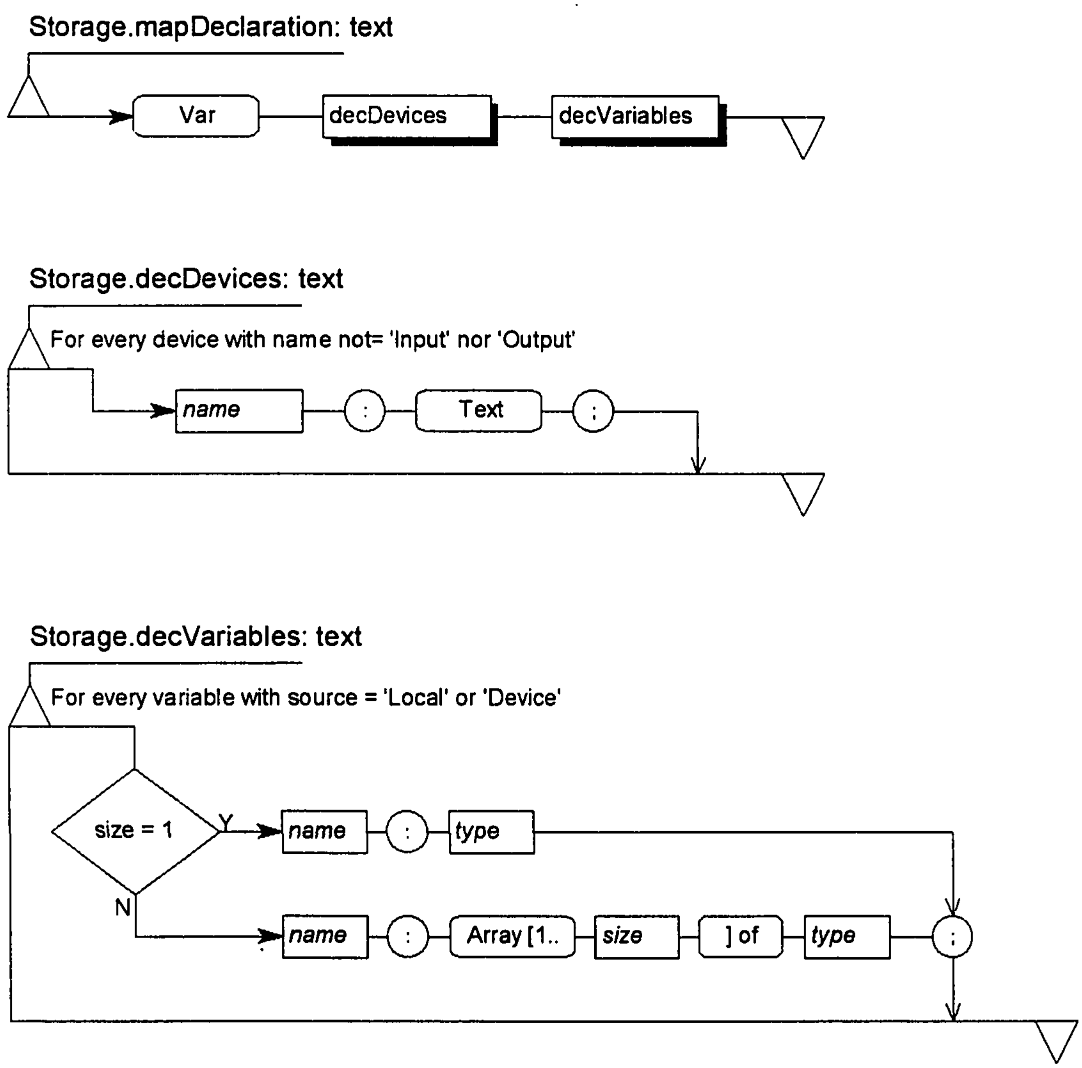


Figure 23 : Declarartion Derivation Methods (Pascal)

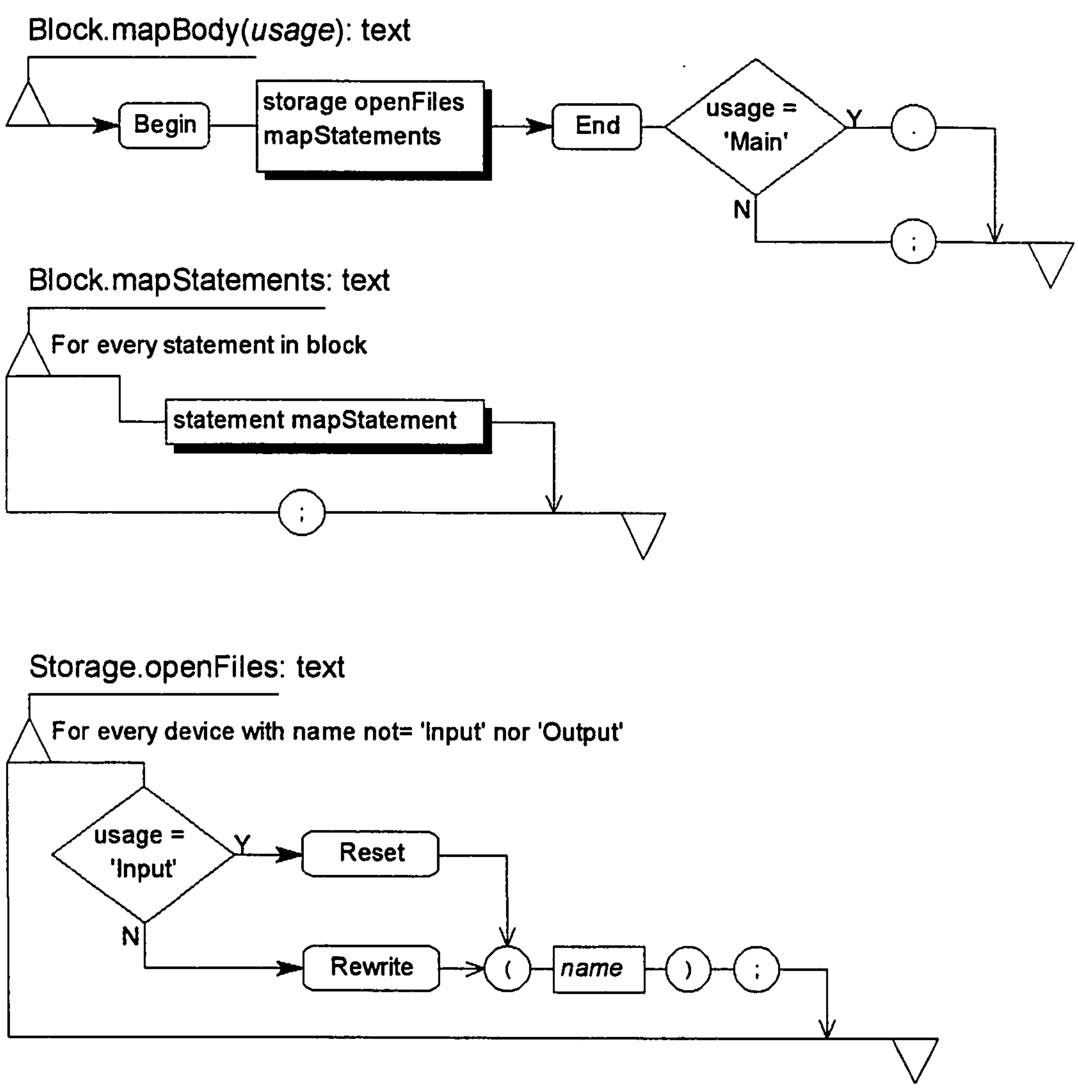


Figure 24 : Code Derivation Methods (Pascal)

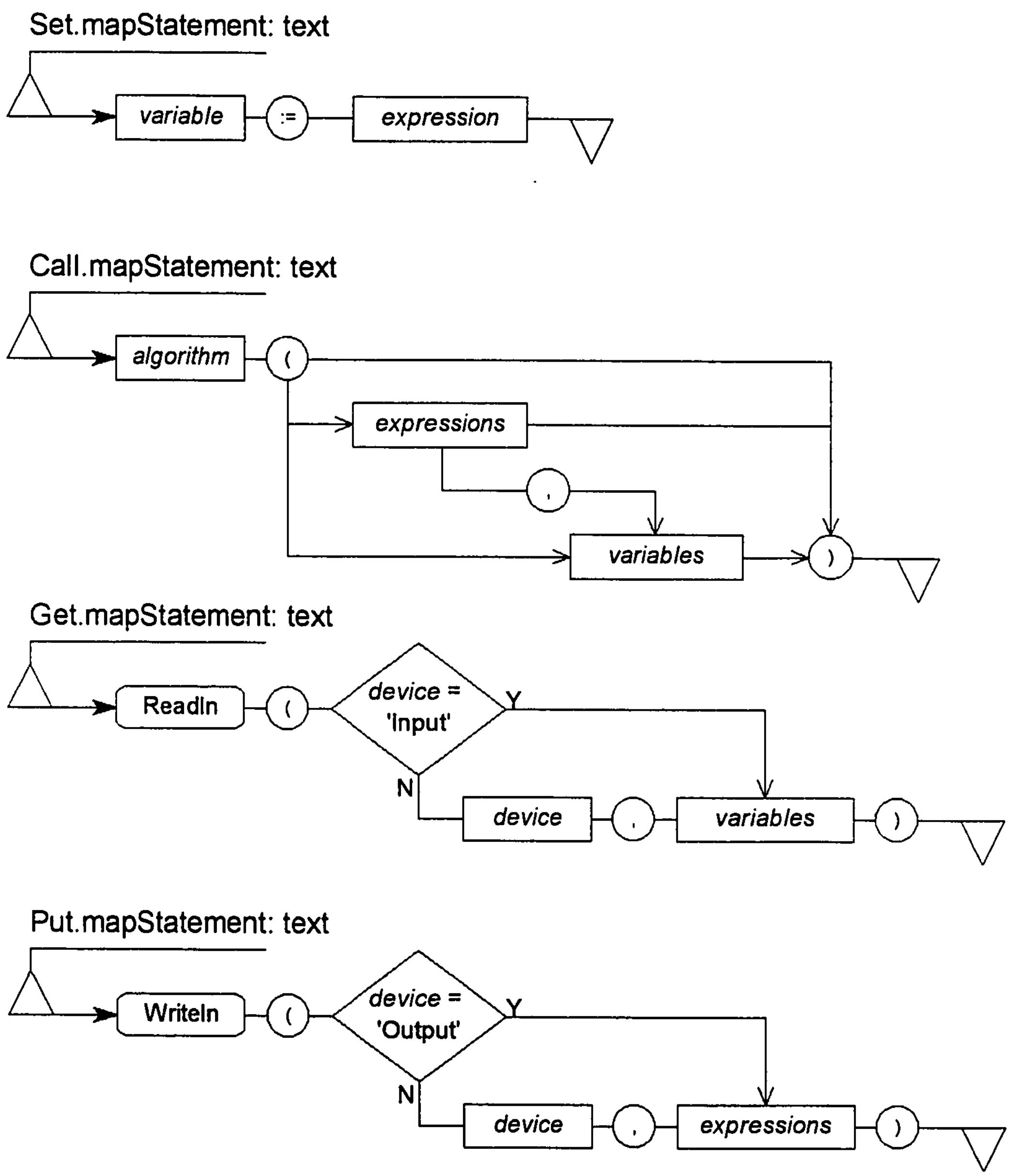


Figure 25 : Command Statement Mapping Methods (Pascal)

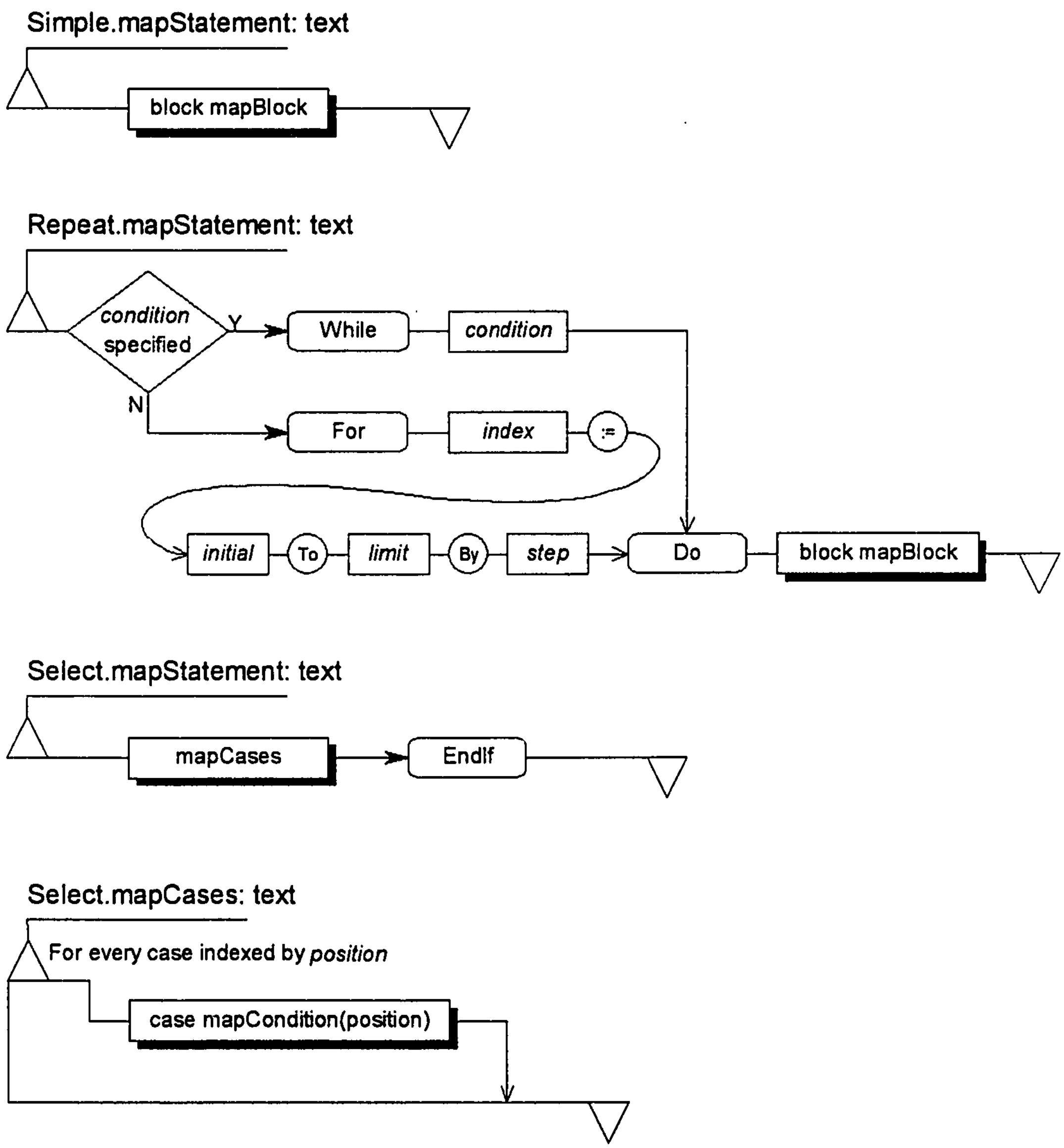


Figure 26 : Structure Statement Mapping Methods (Pascal)

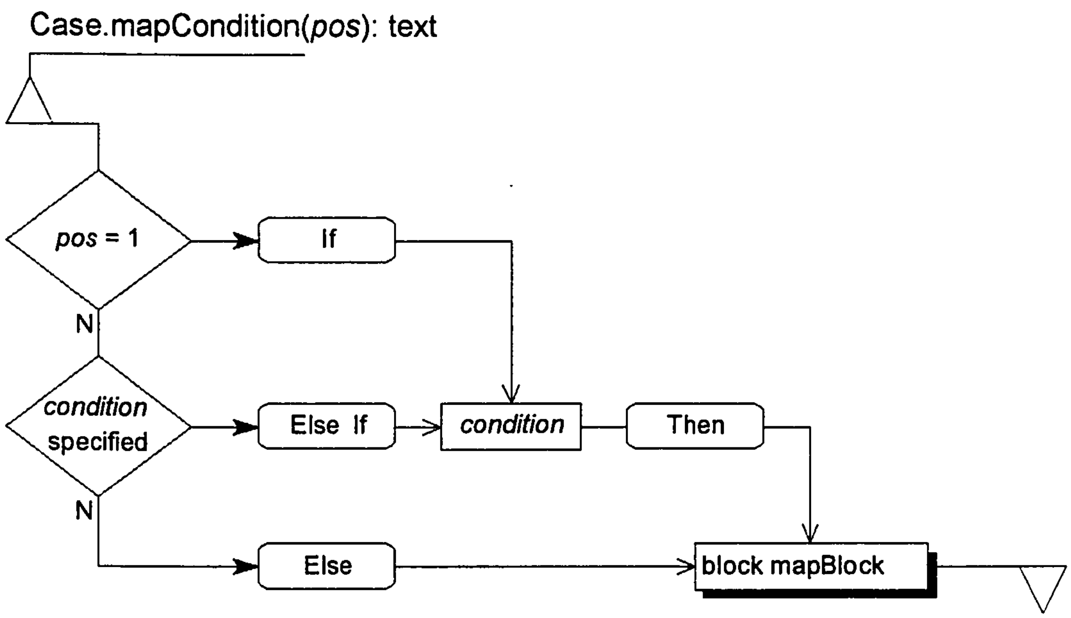


Figure 27 : Case Object Mapping Method (Pascal)

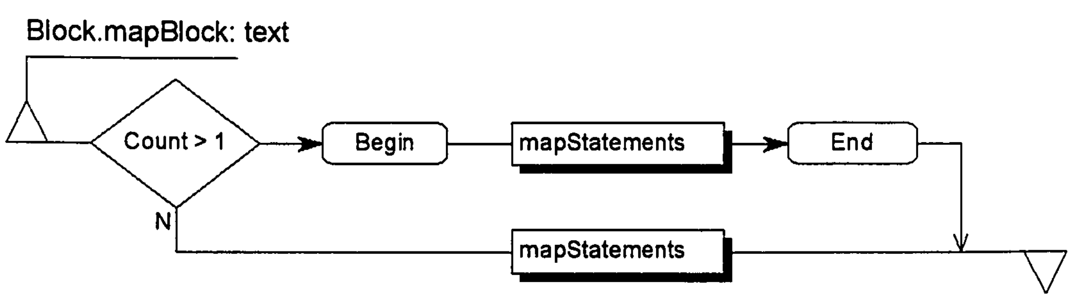


Figure 28 : Block Object Mapping Method (Pascal)

3-5-5 Fortran Code Methods

The following is a list of figures describing methods used in deriving header, declaration, and body attributes of Code object using Fortran Language:

Figure	Methods Title
29	Header Derivation
30	Header Derivation Support
31	Declaration Derivation
32	Code Derivation
33	Command Statement Mapping
34	Structure Statement Mapping
35	Case Object Mapping
36	Block Object Mapping

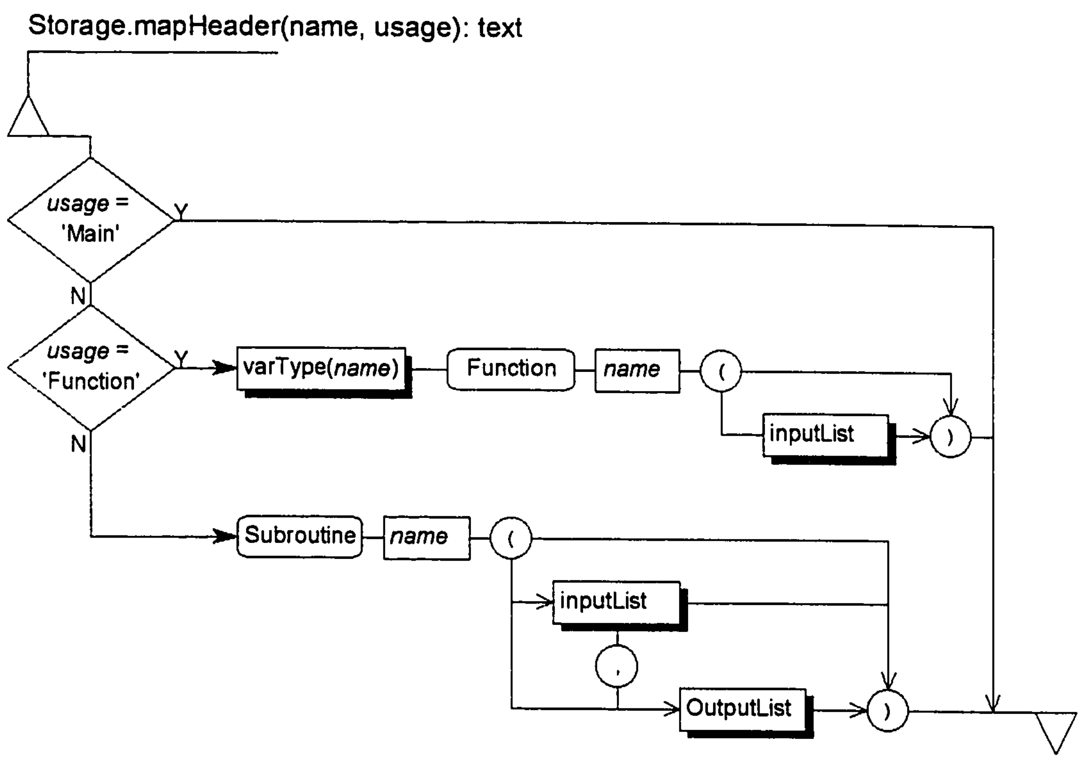


Figure 29 : Header Derivation Method (Fortran)

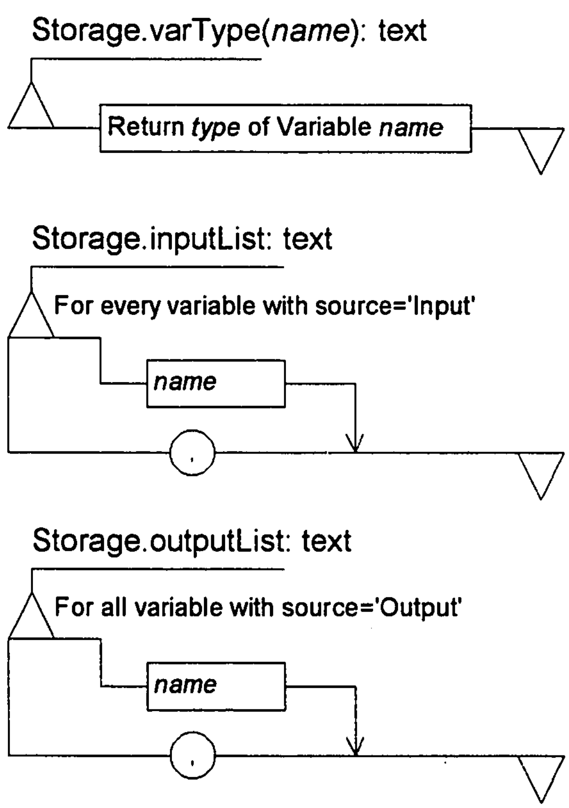


Figure 30 : Header Derivation Support Methods (Fortran)

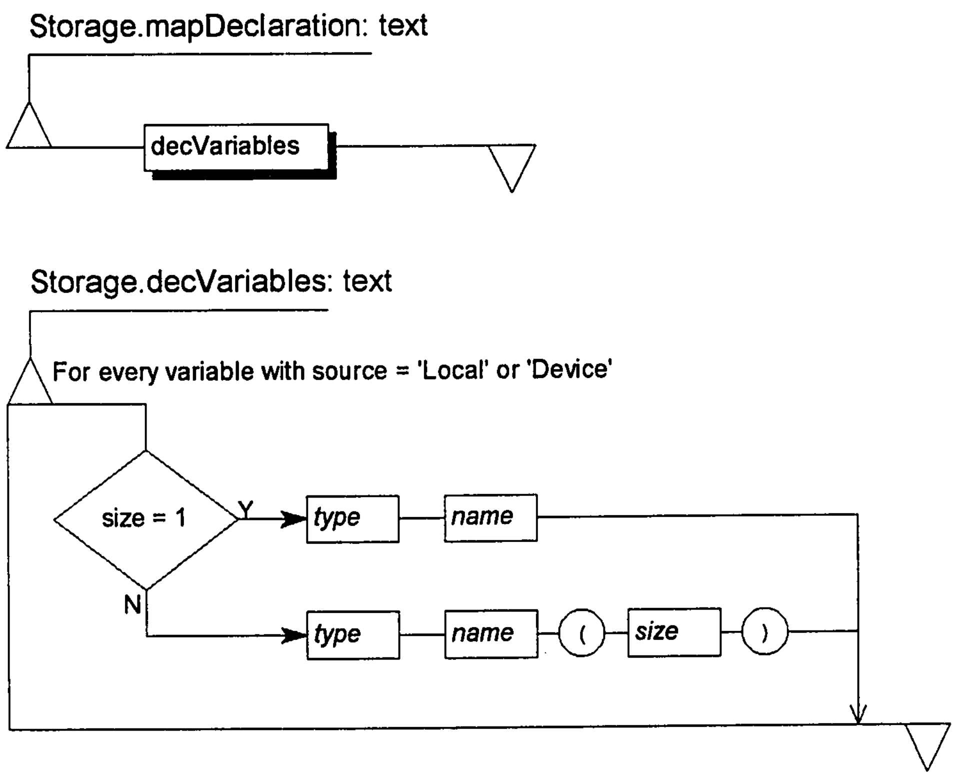


Figure 31 : Declarartion Derivation Methods (Fortran)

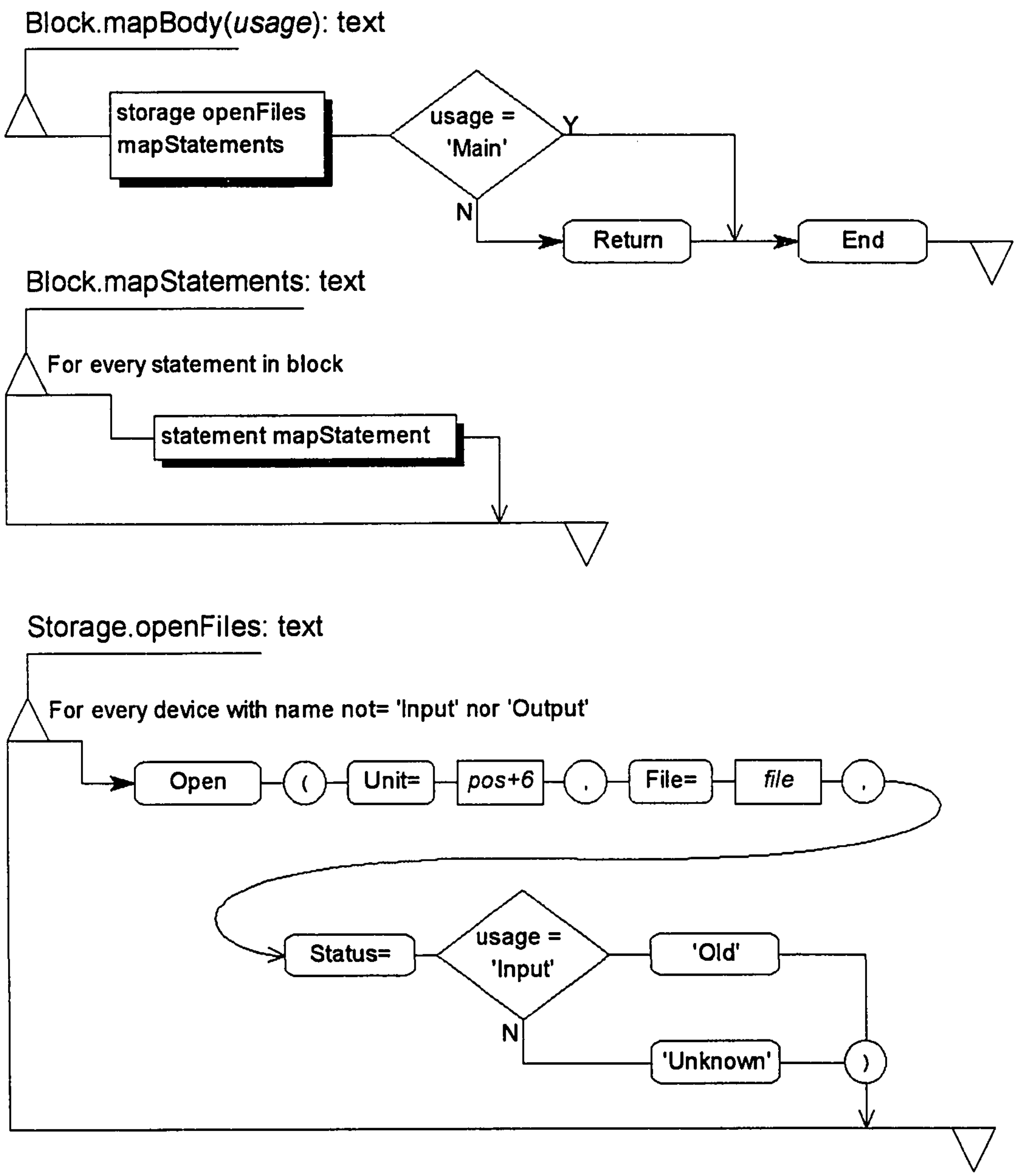


Figure 32 : Code Derivation Methods (Fortran)

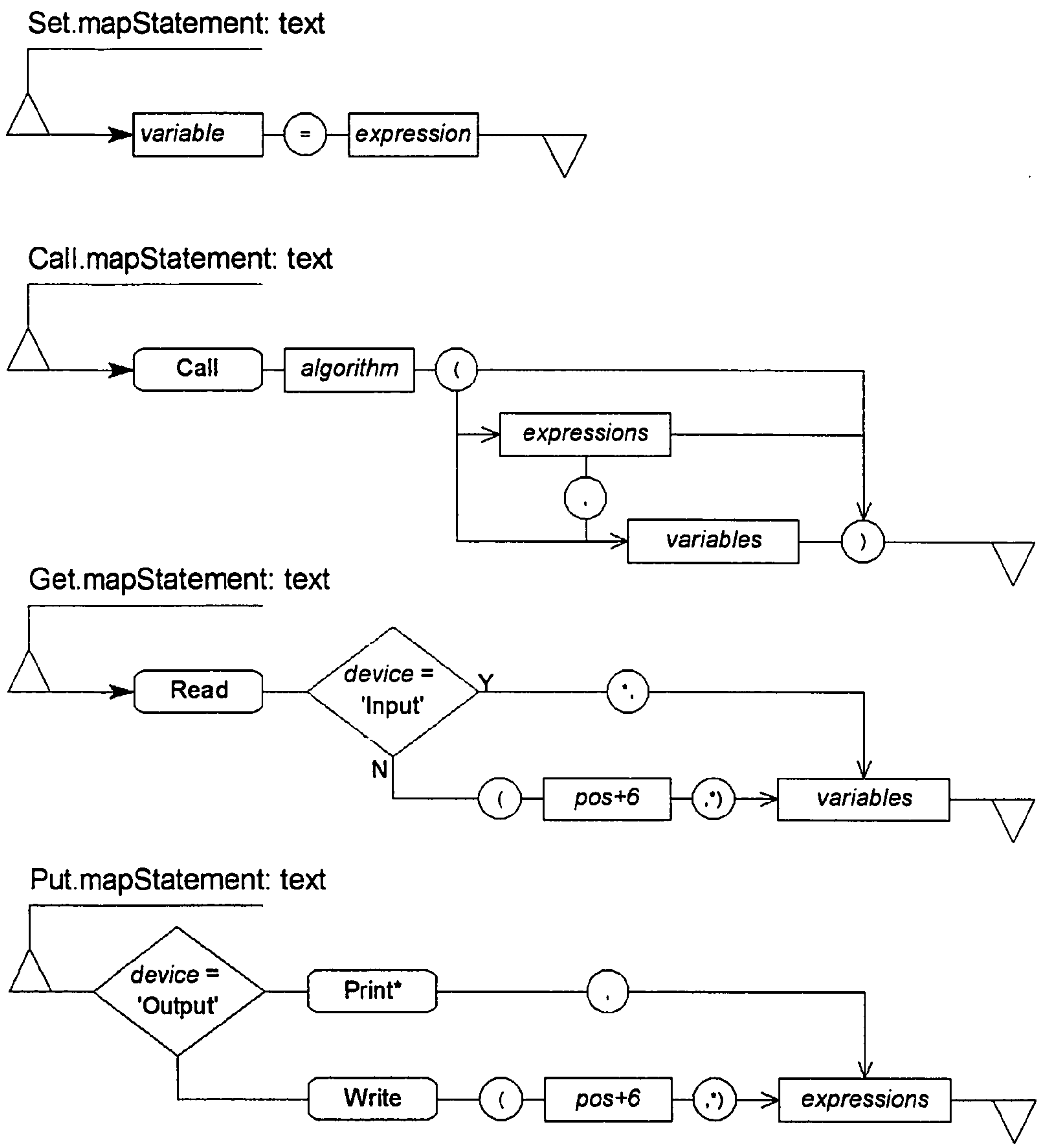


Figure 33 : Command Statement Mapping Methods (Fortran)

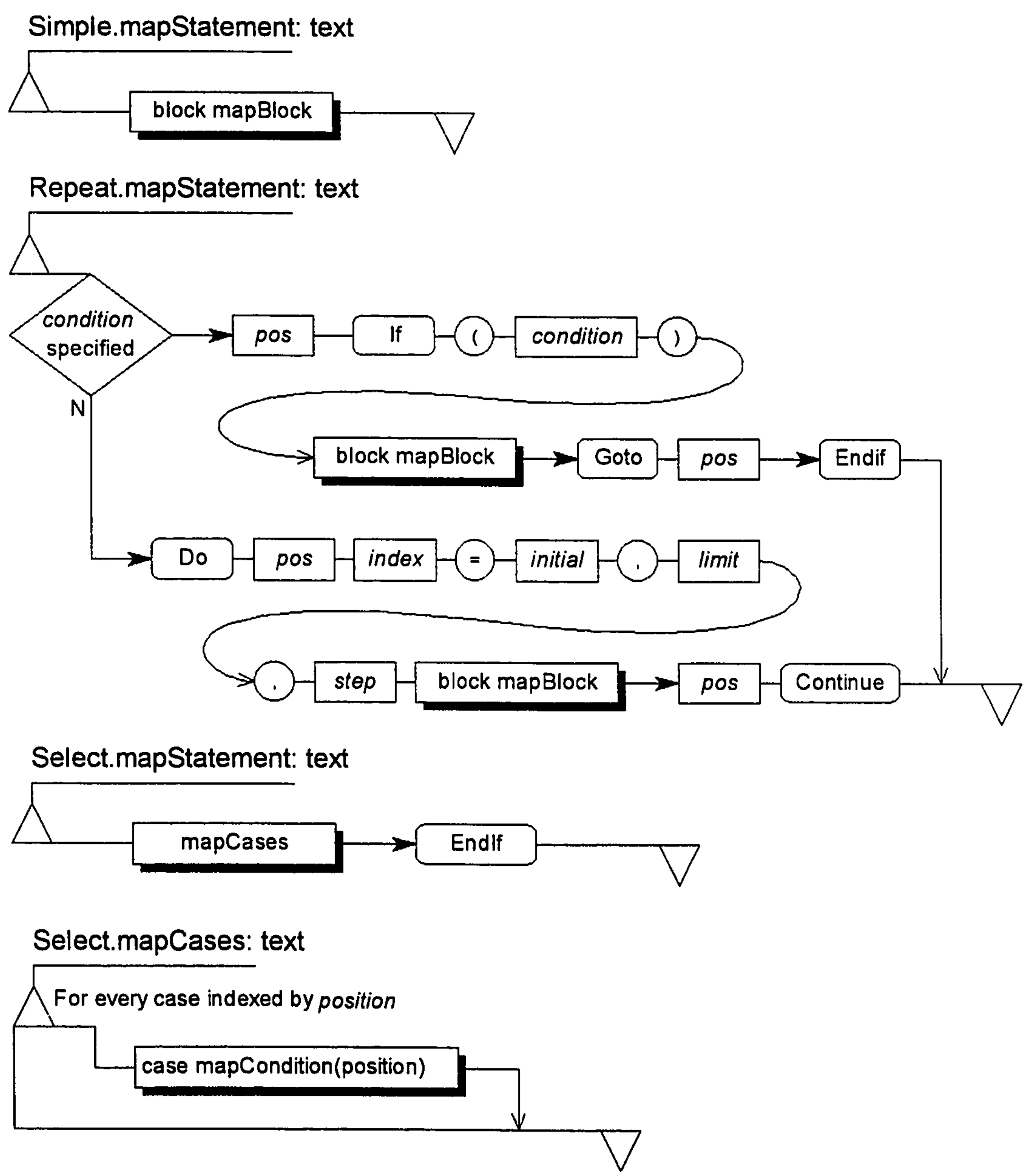


Figure 34 : Structure Statement Mapping Methods (Fortran)

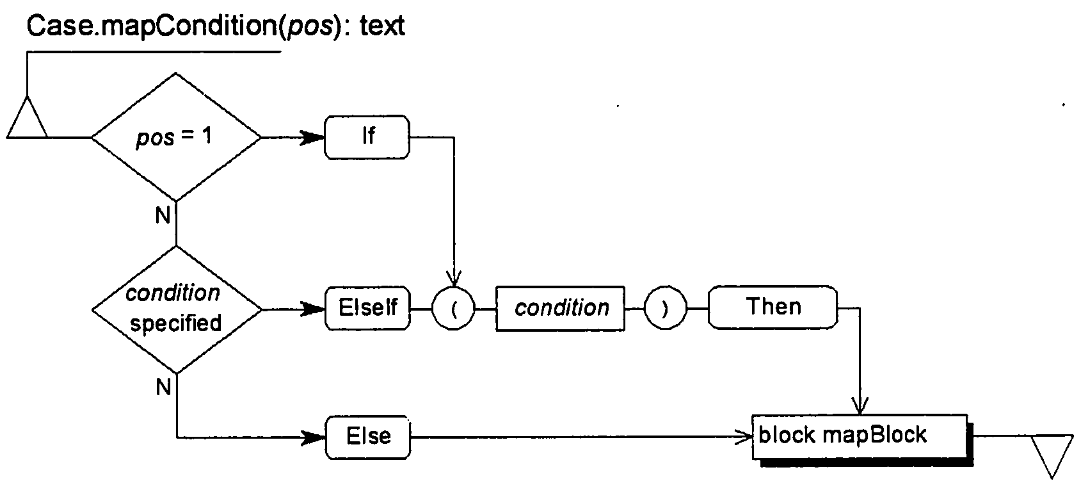


Figure 35 : Case Object Mapping Method (Fortran)

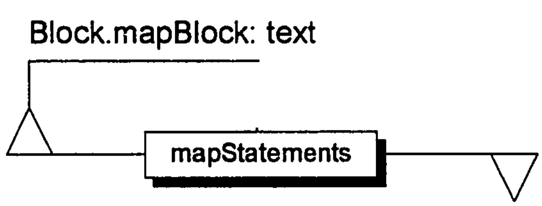


Figure 36 : Block Object Mapping Method (Fortran)

3-5-6 Multiple Algorithms Solution

Certain solutions might require the use of more than one algorithm to perform the task, one of which is used as a main program. CAPE provides a solution object which consists of a set of algorithms and could be associated to an exercise. The solution object has a method that handles the derivation of a code attribute that represents the code for all contained algorithms. Figure 37 shows solution object mapping method.

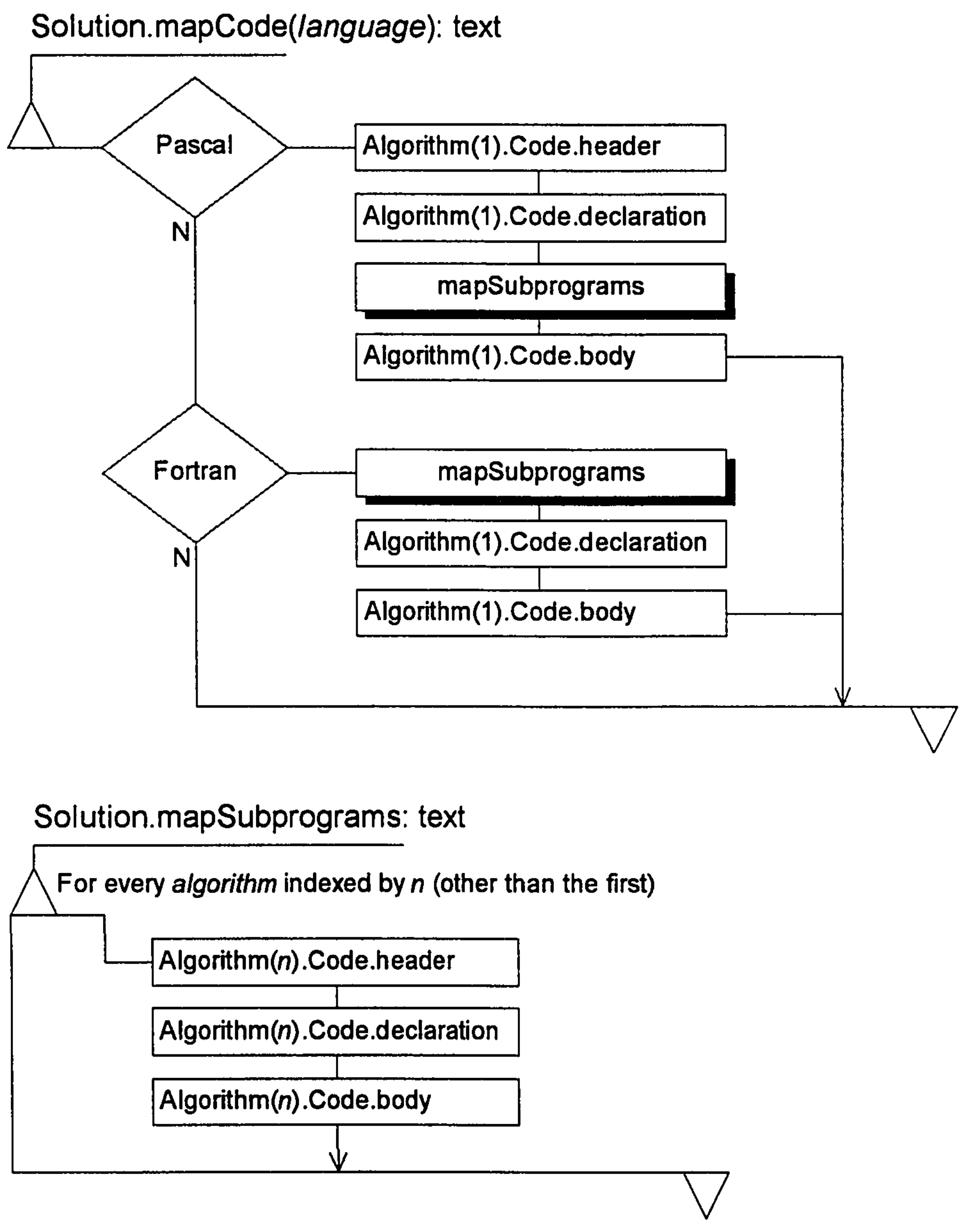


Figure 37 : Solution Object Mapping Method

CHAPTER 4

CASE STUDY

CAPE model could be implemented in various ways. This chapter examines one possible implementation using instance diagrams and user interface screens. We have selected 'Average Calculation' programming problem to be used in describing its CAPE representation. Section 1 describes the model using instance diagrams. Section 2 contains a proposed user interface for CAPE modules.

4-1 Instance Diagrams

The problem definition and algorithm construction modules were described using class diagrams in the previous chapter. A class diagram is used as a template for describing many possible instances of data. An instance diagram describes how a particular set of objects relate to each other. Instance diagrams are useful for documenting test cases and discussing examples. A given class diagram corresponds to an infinite set of instance diagrams. The case study includes a set of instance diagrams describing problem definition and algorithm construction modules. The following discussion describes an instance of a problem definition object and its related components, solve one of the problem's exercises using the algorithm construction module, and then show how the algorithm is represented in Pascal and FORTRAN.

4-1-1 Problem Definition

Figure 38 shows an instance diagram for problem definition object. Problem name, prerequisite, and gained knowledge is indicated in the problem object. The description component, as discussed earlier, is open to support various explanation strategies. Description object contains the objective of the problem's lesson and the outline of steps to be presented. Calculating an average could be described starting with two numbers then more numbers to show the need for an accumulator. The last step is to show the usefulness of repetition construct to implement accumulation process.

The problem has two exercises as shown in the figure. An exercise has its own description, input, output, and error specification. The figure shows a short description of an exercise. The first exercise requires the development of a program that reads a count of grades followed by their set. Three sample data is assigned for that exercise. The first data checks the basic requirements, the second is used to validate the execution using only one input, and the third is used to check if reading will go beyond the specified count. The second exercise is similar to the first, except now the data is delimited by a given value and additional error handling is required.

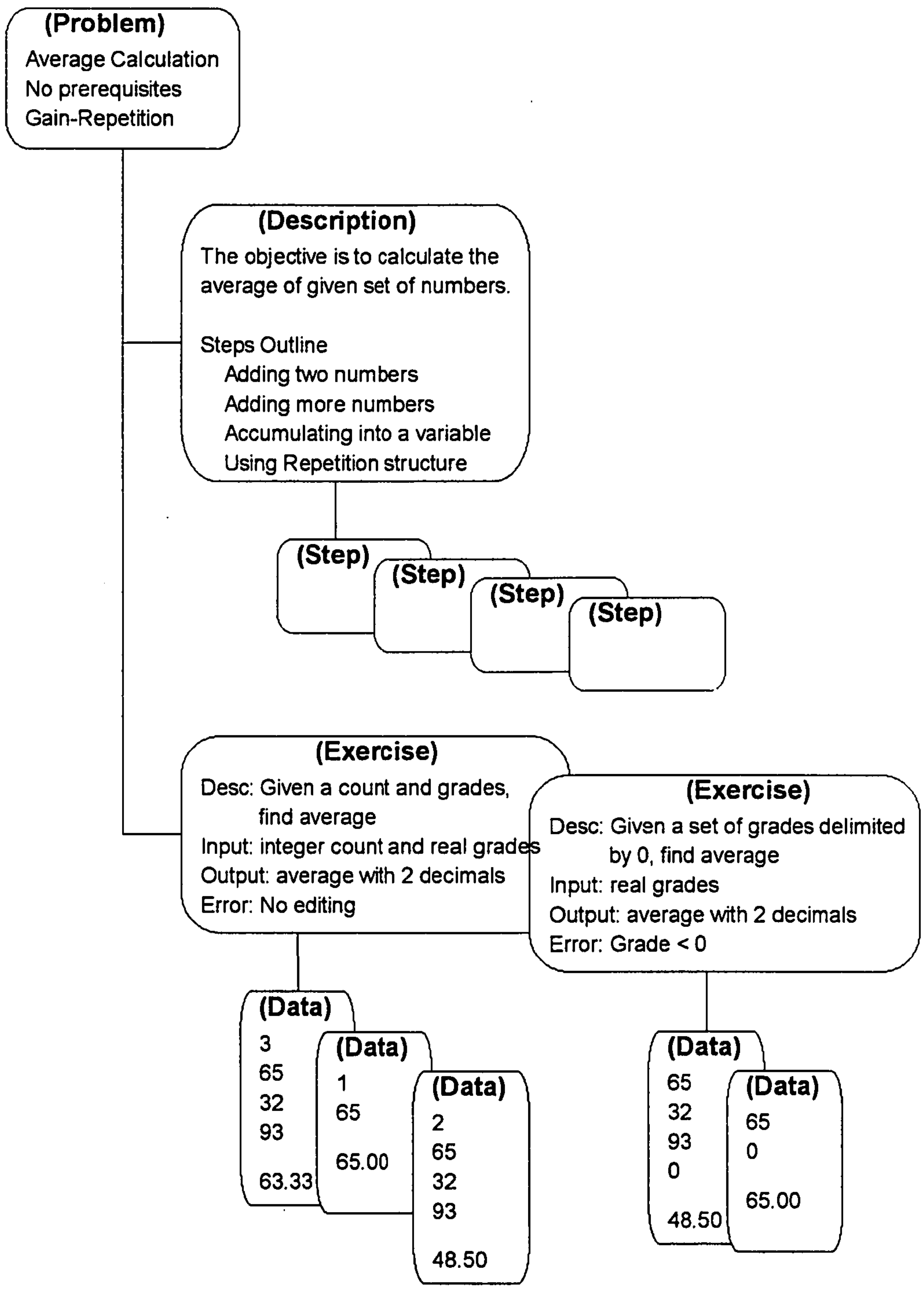


Figure 38 : Problem Definition Instance Diagram

4-1-2 Algorithm Construction

Figure 39 shows an instance diagram of the constructed algorithm that solves exercise one of the defined average calculation problem. The constructed algorithm is named 'Avg1', used as a 'Main' program, with a short description of what it does. The date and owner of the algorithm is maintained by the system. The algorithm consists of a block object, a derived storage object, and a code object which is discussed in the following section. The algorithm's block, with 'Main' as description, has six statements. The accumulator *Sum* is set to zero followed by a read of the count *N* of grades from *Input*. A repeated block *Total grade* is executed for an index *I* varying from *1* to *N* by *+1* steps. The repeated block has two statements to get *Grade* from *Input* then add it to *sum*. After repetition, *Average* is calculated in statement four. A select statement is used to conditionally put a status message. The first condition *a* where calculated average > 90 simply put an 'Honor' message. The second condition *b* where average is between 90 and 70 reports that the average is upgraded by 10 points. After this select statement, the average is reported to *Output* device.

Storage variables and devices objects were derived from the algorithm's block statements. A variable has a name, type, size, usage, and a derived count attribute. When a variable is created, a default type of *{type}* is used with its count set to one. Note how variable *Sum* is counted twice due to its dual usage. A device has a name, usage, filename, and a derived attribute of count. In our example the only device used is the input/output terminal. Output device is used in three statements where input device is used twice.

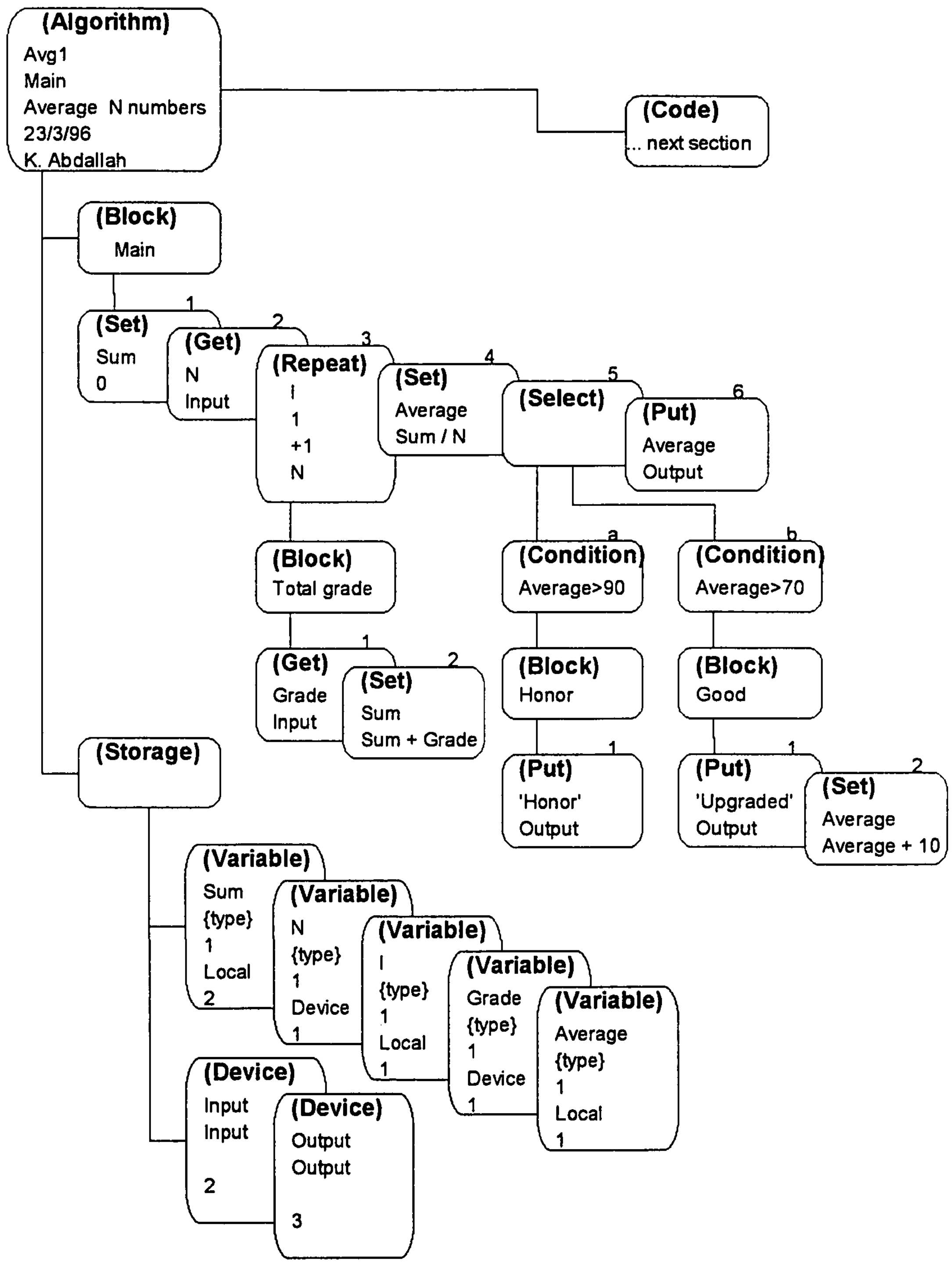


Figure 39 : Algorithm Construction Instance Diagram

4-1-3 Code Derivation

Once the algorithm has been constructed, the owner may complete the specification of defaulted attributes, such as variable type. The choice of language generates the corresponding code, as shown in Figure 40. The constructed code has a header, declaration, and body attributes. Storage object derives header and declaration content where block object derives the body content. Indentation is used to clarify the generated code since it was not explicitly specified in the model.

The generated code in both languages helps students view a simple representation of the constructed algorithm in different languages. Every language has its own naming convention and expression syntax. The code could be directly modified to have a valid compile run but these changes are not reflected in the algorithm. There are certain requirements that can not be represented using the algorithm construction, such as formatting an output or indenting the code. Our suggestion is to get the heart of the algorithm working first and then generate code that will be modified to include final touches.

FORTRAN does not recognize the specified 'If' conditions since they include '>' sign where it should have been 'GT' for greater than. Expressions specified in the algorithm are not validated by CAPE, the compiler does. After using the language for a while, methods of specifying expressions will eventually be consistent with what the language requires. The other note is that variable name 'Average' has more than seven characters, the compiler would not accept it.

(Code-Pascal)

```
Program avg1(Input, Output);

Var
  I : Integer ;
  N : Integer ;
  Sum : Real ;
  Grade : Real ;
  Average : Real ;

Begin
   Sum := 0 ;
   Readln ( N ) ;
   For I := 1 To N By +1 Do
   Begin
      Readln ( Grade ) ;
      Sum := Sum + Grade
   End
   Average := Sum / N
   If Average > 90 Then
      Writeln ( 'Honor' )
   Else If Average > 70 Then
   Begin
      Writeln ( 'Upgraded' ) ;
      Average := Average + 10
   End
   Endif
   Writeln ( Average ) ;
End .
```

header

declaration

body

(Code-Fortran)

```
Integer I
Integer N
Real Sum
Real Grade
Real Average

Sum = 0
Read*, N
Do 3 I = 1, N, +1
   Read*, Grade
   Sum = Sum + Grade
3  Continue

Average = Sum / N
If ( Average > 90 ) Then
   Print*, 'Honor'
Elseif ( Average > 70 ) Then
   Print*, 'Upgraded'
   Average = Average + 10
Endif

Print*, Average
End
```

Figure 40 : Algorithm Code Object

4-2 User Interface

CAPE model could have a number of various implementations. This section presents one possible user interface using Windows-like screens. Problem definition module is discussed first to show how an instructor defines a problem and how the student reviews it. A sample algorithm that solves a selected exercise is presented next to highlight algorithm construction module.

4-2-1 Problem Definition

The problem definition module provides an environment for the designer to define programming problems to the system and for the student to review a problem. The interface is consistent but capabilities are different. Figure 41 shows problem definition interface. Each problem consists of description, discussions, and a set of exercises.

The interface allows the designer to create a new problem or open an existing one for maintenance. Once the problem is selected, the designer can choose any of its components to work with. When a component is selected, its content is displayed in the work-sheet for the designer to work with. The interface is equipped with standard menu commands, such as File and Edit. File commands reflect actions on the problem on hand, such as open or close a problem, where as edit commands reflect actions on problem's elements, such as copying or deleting a text. Students access problems via the same interface.

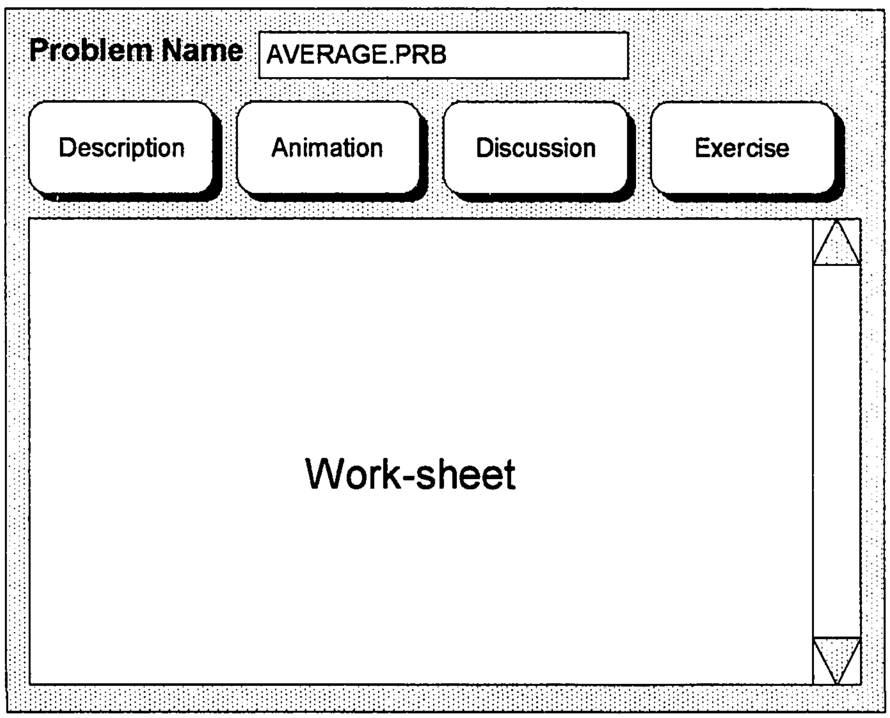


Figure 41. Problem Definition Interface

Description Component

Selecting description component allows the designer to specify a textual description of the problem. Figure 42 shows the description component of the problem. The cursor is placed inside the work-sheet to describe the problem and any related mathematical background that might be useful.

Animation Component

Animation component, Figure 43, consists of a reference to selected ready-to-play animation files. These animation present to the user interesting material that represent execution of a program or steps involved to deal with a problem. This is the area where students will enjoy watching how algorithms process data to achieve certain results. Students can scan through problems at an early stage just to explore what they may contain. The designer can use any preferred animation tool to construct the desired animation file and then associate it to the problem.

The figure shows an example of two animation files. To add another record in the animation list, you can simply insert a record in the list and specify its file name and description. The interface can be enhanced to allow the designer select an animation file from the library and view/play their content before assigning it to the problem. When students access the animation, file name will not show and double clicking the title will start playing the animation file.

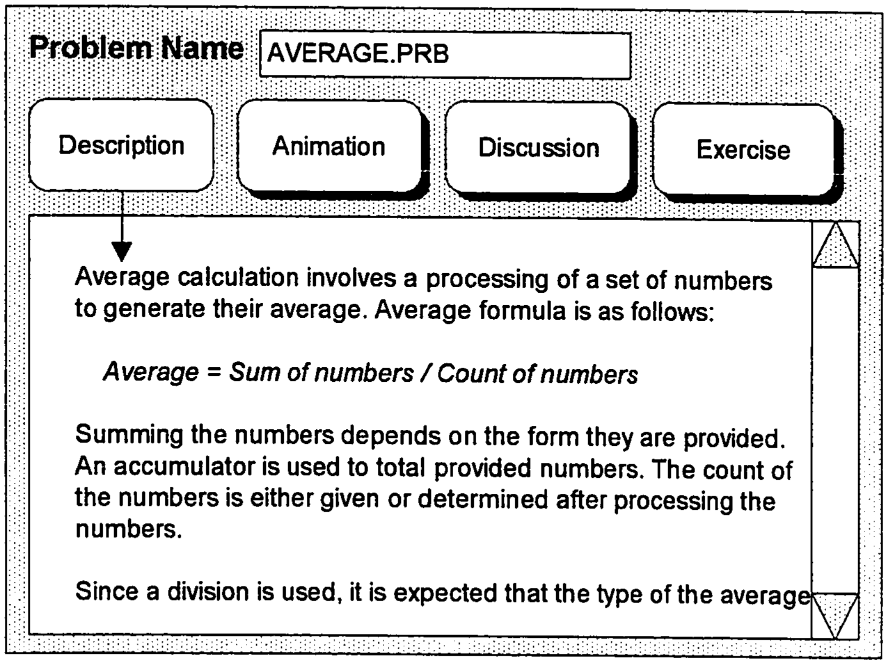


Figure 42. Description Component Interface

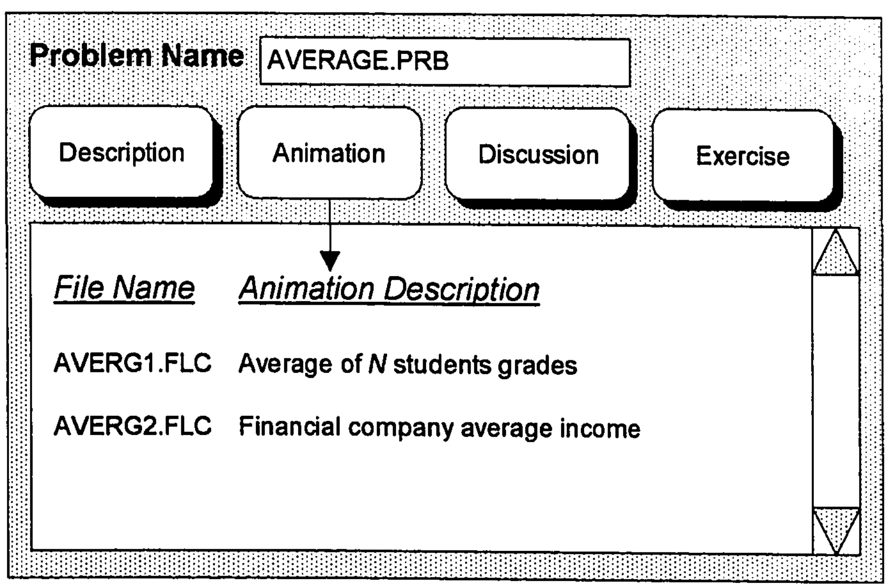


Figure 43. Animation Component Interfac

Discussion Component

Discussion component provides means of evaluating student understanding of the problem using informative questions. Student responses to the questions are not recorded but used to direct presented material. The designer can develop the instructional material using Authorware [19] or hypermedia technology and then associate those instructional materials to the problem, just as we did for animation component. Figure 44 shows the content of average problem discussion component. To add another record in the list, you can simply insert a record in the list and specify its file name and description. The interface can be enhanced to allow the designer select a discussion file from the library and view/play their content before assigning it to the problem.

Questions should be designed to be simple with informative responses for invalid answers. The sequence of questions should start from simple to more involved ones. One problem could be handled with a number of questions that are presented using one instruction. A linear sequence approach of questions is acceptable. Students may select to start from any instruction and continue from there. This structure allows the formation of questions to be independent of the sequence they are presented in. It also allows the sharing of certain instructions between problems.

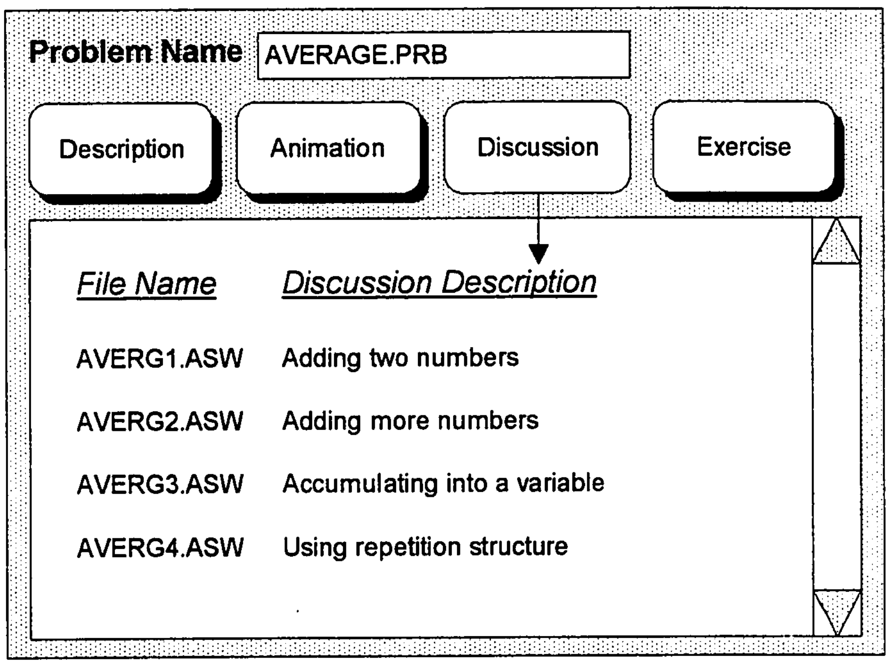
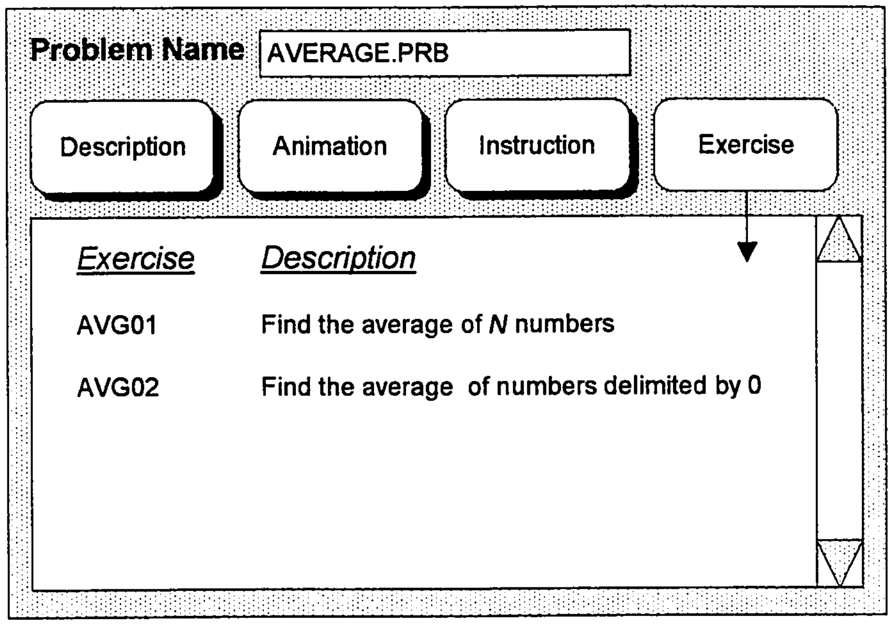


Figure 44. Discussion Component Interface

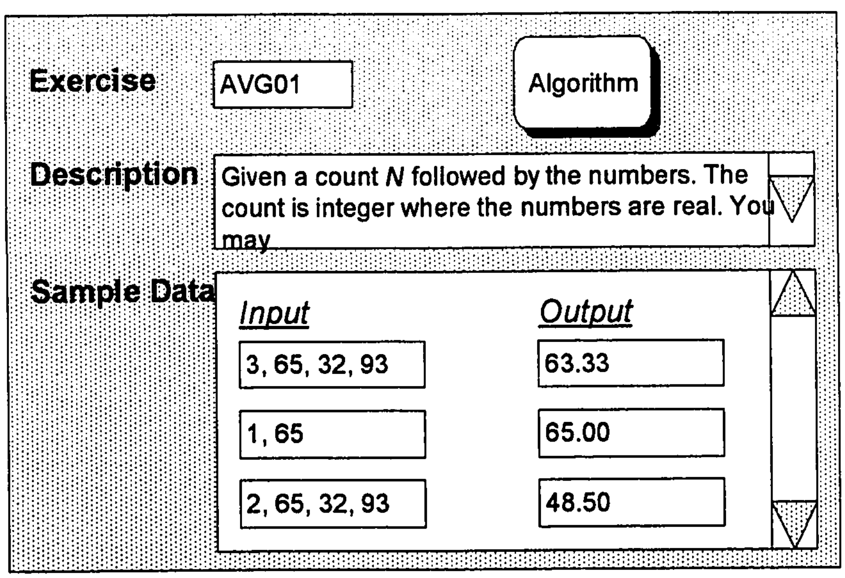
Exercise Component

Each problem can be stated in different forms, but still contains the heart of the problem. Exercise component contains a list of programming exercises derived from the associated problem. The student should be able to solve those exercises after reviewing the previously discussed components. The designer specifies the exercise name, textual description, and sample input/output data. Figure 45(a) shows a list of assigned exercises given to average calculation problem. Exercise names and their first line of descriptions are listed on the worksheet. To add a new exercise, you simply insert a record and give its name and description. Double-clicking the exercise record opens up the exercise definition panel, as in Figure 45(b), for the designer to provide sample input/output data. An exercise can have more than one sample input data. For each input data, the designer should specify the expected results of a developed program. Each sample data can accommodate multiple records, what is displayed is just the first record. To enlarge an element and work with it, double-click it to be enlarged.

When a student has reviewed the problem from all directions, he can then tackle available exercises. The student can read the description of the exercise and analyze its sample input data and output requirements. The interface is equipped with a facility to access the student's associated algorithm or program solution. The exercise name must be used as the name of the constructed algorithm and code. This naming scheme facilitate quick reference to constructed solutions for a problem's exercise. Clicking on *algorithm* icon initiates the algorithm construction module. If this is the first attempt to construct the exercise algorithm, an initialized algorithm is opened using the exercise name. If there is an algorithm in existence, it is opened for further maintenance.



a) List of exercises



b) Exercise review

Figure 45: Exercise Component Interfac

4-2-2 Algorithm Construction

This module is considered to be the heart of our work. It provides the student with an interface to construct an manipulate algorithms. Figure 46 shows the algorithm construction interface. The interface consists of an algorithm construction area, a set of dragable icons representing commands and structures, and menu commands. The algorithm is created by placing statement icons onto the flow-line. The order of icons and the content you give them creates your algorithm. Standard menu commands such as File and Edit are used to open/close an algorithm and copy/paste its elements. The student may name the algorithm any name, but to solve a given exercise, the student must name the algorithm as the exercise's name. This will facilitate quick reference for the student to access a developed solution and it will also facilitate the evaluation process of students developed solutions for a given exercise.

Construction Area

Construction area consists of a flow-line that represent the sequence of executed statements. The user can select a statement and paste it on the flow-line to build the algorithm. Block icons have their own flow-line. When an icon is placed on the flow-line, the user is prompted to give its attributes. Double-clicking on an icon opens its definition panel.

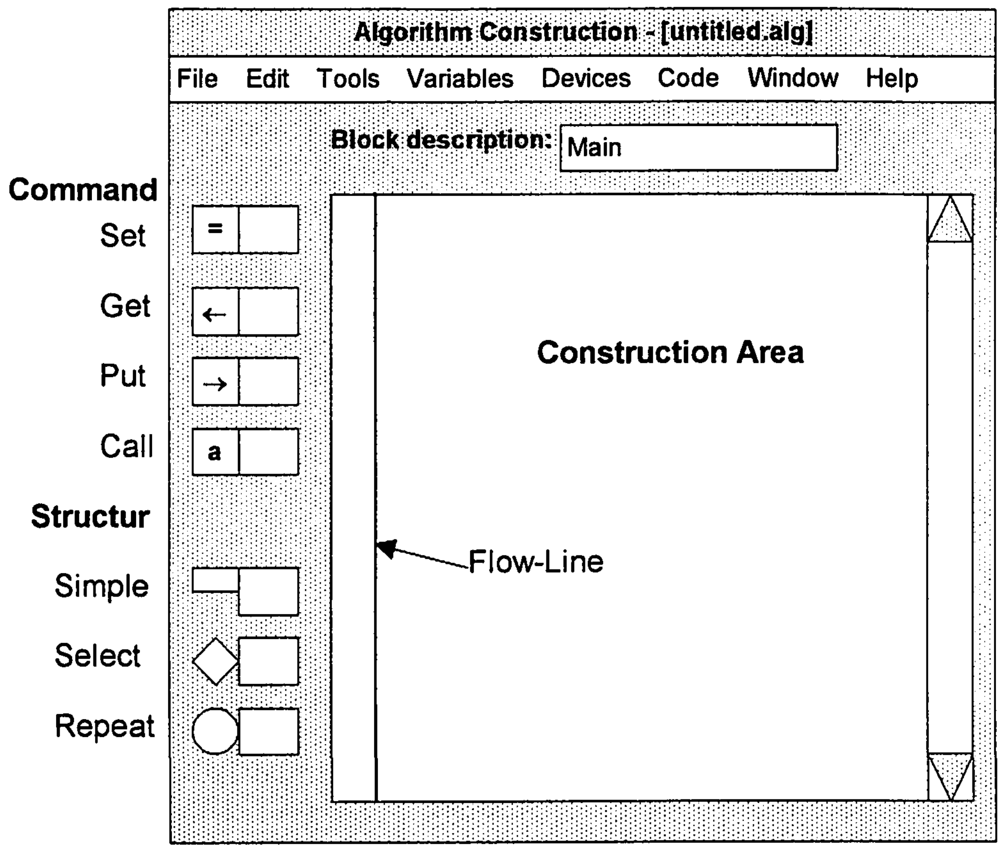


Figure 46: Algorithm Construction Interface

Construction Icons

The user is provided with a set of construction icons that can be used to build an algorithm. These icons represent CAPE's algorithm statements. We have carefully designed the icons to represent their action by choosing symbols that are consistent with standard representation of programming constructs, such as a diamond shape for conditional block and a circular shape for repeated block, in most cases. To build an algorithm, the user simply pastes these icons on a flow-line and the system attaches it to the flow-line. Edit menu commands can be applied to selected icon(s) such as copy/paste. Each structure statement has its own flow-line that can accommodate other icons.

Command Icon

A command icon is used to represent basic algorithmic statement. Pasting the icon on a flow-line prompts the user for the specification of the statement attributes, such as variable name(s) or expression(s). The name of the icon is indicated next to it along with required attribute parameters. These parameters are free form and length. Multiple variables may be specified in some commands, such as Get or Call, but not in Set. Figure 47 shows the first two statements of average calculation algorithm. To modify a command parameter, click on it and type.

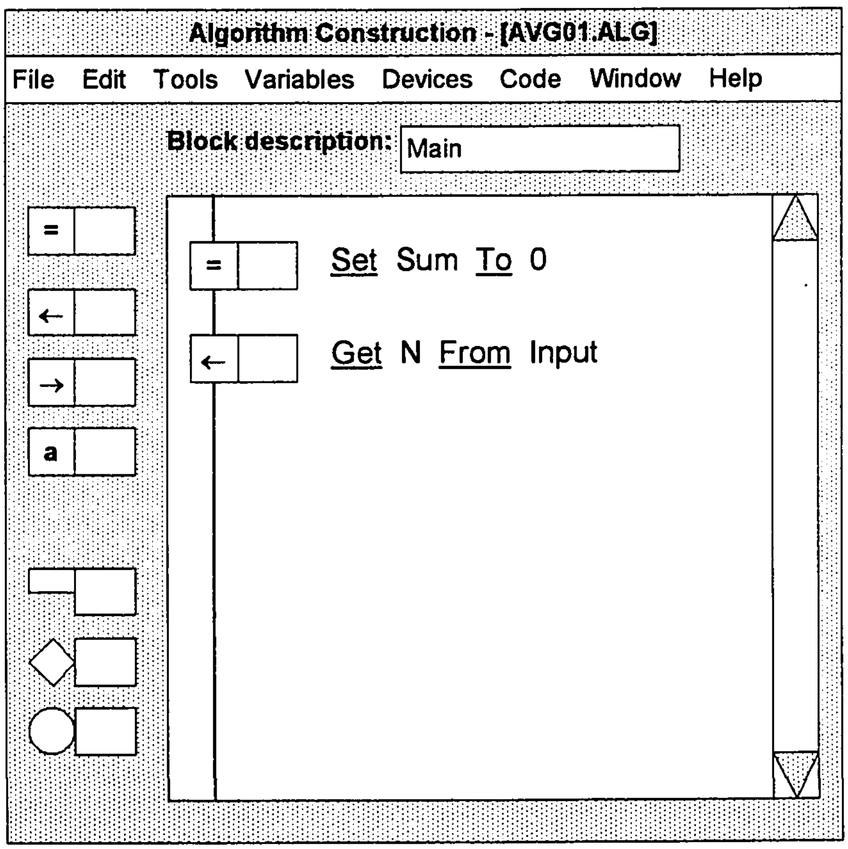


Figure 47: Command Statement Interface

Repeat Structure

Repeat structure contains a block of statements that are executed repeatedly. The block consists of one flow-line with repetition control attributes. Figure 48 shows the display of repeat structure window. The window is displayed after pasting the icon ready to accept its block statements. Repetition control consists of two parts, condition and index. They are used to control the number of iterations the block has to execute. Repetition control can either have a condition specified or index and its parameters, but not both. The condition part specifies the logical expression(s) that has to be true for the block to execute repeatedly, indicating WHILE loop construct. The index part is used to repeat the block execution for an integer index varying from an initial value to a maximum limit. The block is executed for every increment of the index, indicating a FOR loop construct.

For our average calculation problem, the repetition is performed using an index varying from 1 to *N*, which was read by the second statement. The two statements were pasted after the initialized block was displayed. Once the statement parameters and statements have been specified, the window could be closed to view the algorithm's main block. The repeat statement icon is represented along with its repetition control parameters on the flow-line.

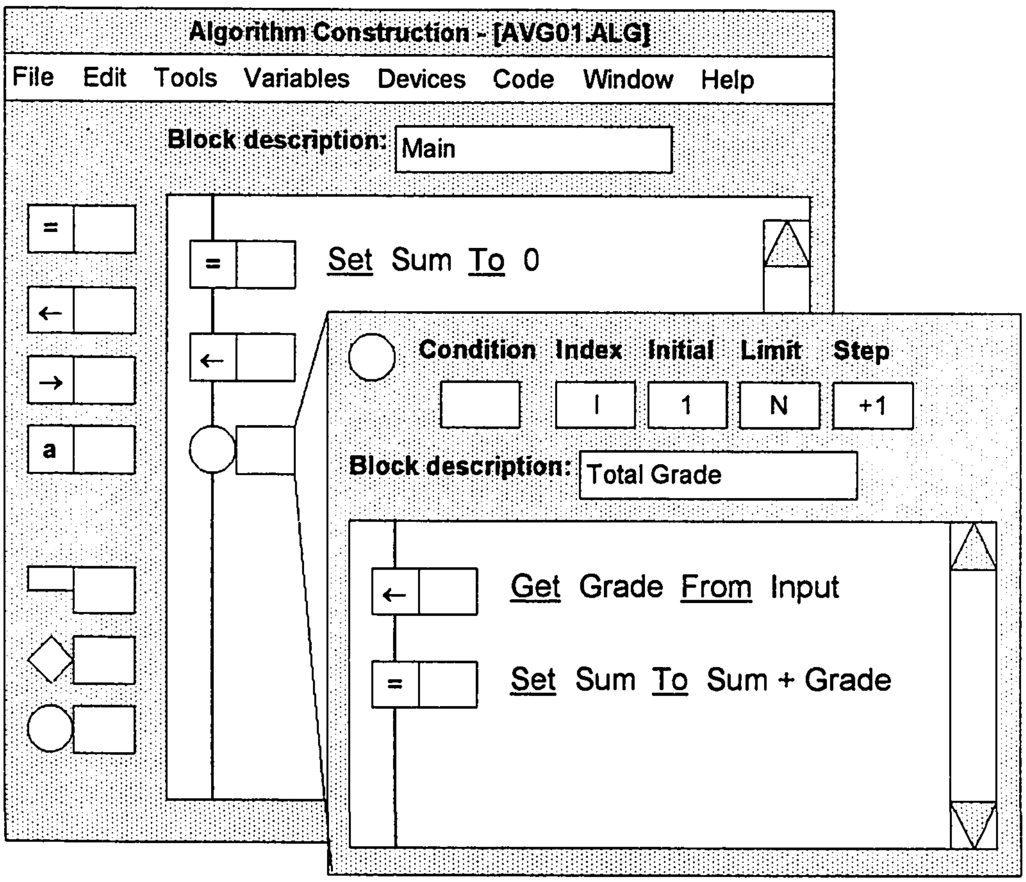


Figure 48: Repeat Statement Interfac

Select Structure

Select statement contains a set of cases that are executed conditionally. Each case has its own block of instructions. Pasting a select statement icon opens up the first case condition and its associated block. To view or specify the next condition in line, you may scroll by clicking the right arrow for next condition and left arrow for previous condition. The displayed block of statements on the flow line corresponds to the selected case. Figure 49 shows a select statement with two conditions, the first one when the average is greater than 90 and the second when the grade is greater than 70. There is no else condition used in the example, but to include one you can simply scroll for an empty condition and specify its statements.

Menu Commands

File and Edit menu are standard commands to open or edit algorithms. Tools menu consist of simple, select, or repeat functions which are used to group a selected set of statements under the indicated construct to support *bottom-up* design. Variables menu allows the maintenance of derived local and device variables as well as specifying input and output variables when the algorithm is used as a function or procedure. Maintenance normally includes the specification of the variables type and size. Devices menu also allows the declaration of derived devices. Code menu allows the viewing of derived code based on the constructed algorithm using any of the available languages. Figure 50 shows the content of menu commands.

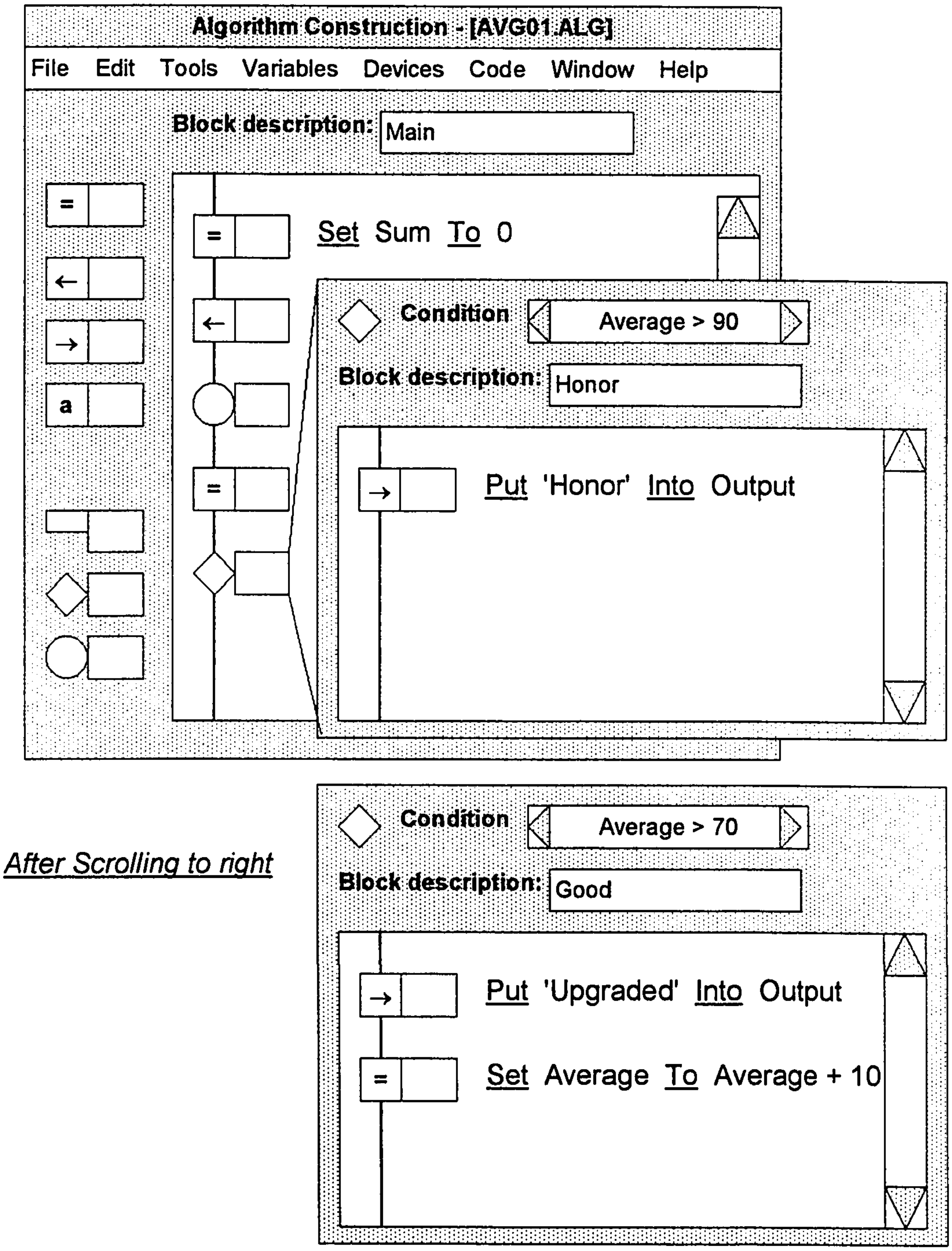


Figure 49: Select Statement Interface

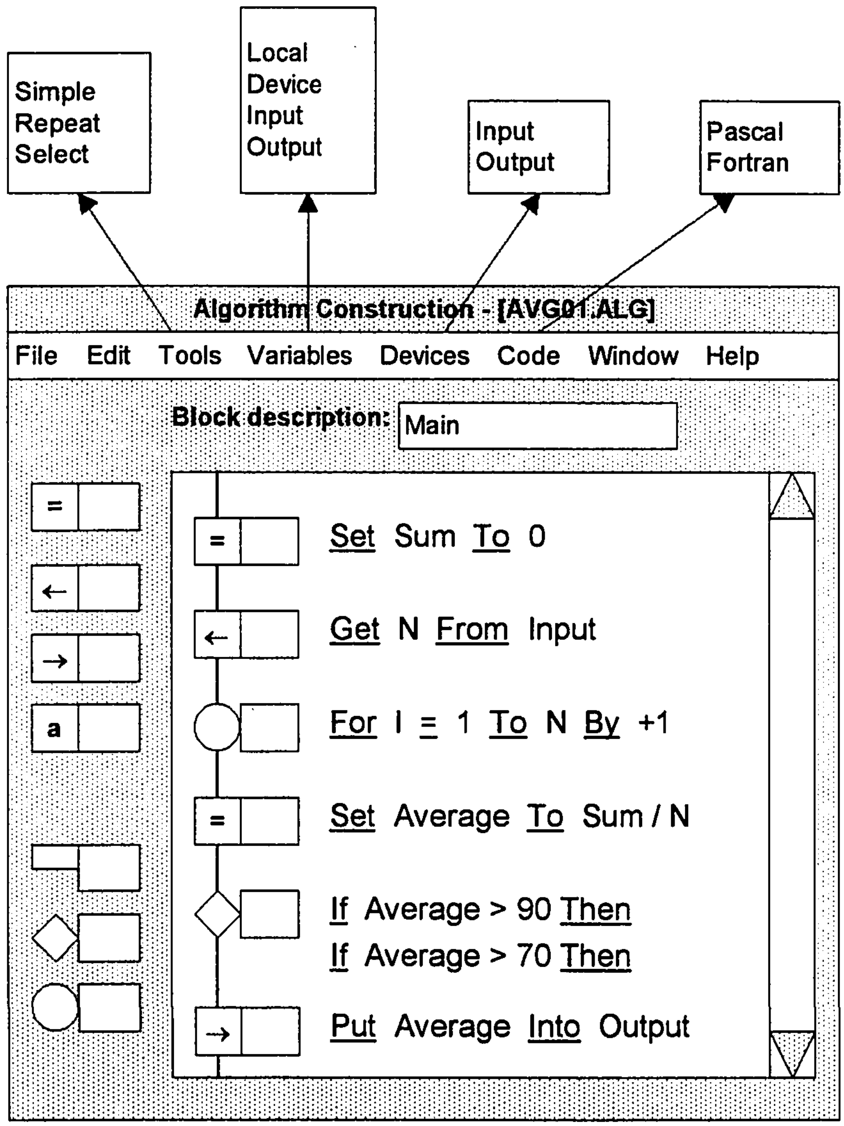


Figure 50: Algorithm's Menu Commands

CHAPTER 5

CONCLUSION

The variety of procedural programming languages widely used in practice stresses the importance of building computing environments aimed at helping students acquire the capability to solve programming problems independently of a specific language. Moreover, if such environments are endowed with the capability to produce code automatically in different languages, students are motivated to study algorithmic problems operationally, and are provided with a tool for rapid prototyping in various programming languages. On this basis we have designed the system described here to be used in the teaching/learning of programming with university students at an introductory level. For such students, understanding the role of algorithms and obtaining the knowledge on implementation techniques are issues in programming education and practice.

5-1 Comparison With Other Models

METKIT CAI system integrates concepts of hypertext with those of computer aided instruction to provide novice users with guidance through knowledge based on didactic strategies. CAPE's problem definition module is designed to provide users with guidance in understanding programming problems. The description component is made up of pedagogical steps, whose successive unrolling will constitute the covering of the concept.

METKIT basically provides facilities to update and browse through conceptual knowledge. CAPE provides similar mechanism to understand programming problems and also recognizes knowledge available in commercial coding environments, such as help and tutorial.

INTELLITUTOR attempts to work as a human programming tutor to guide a student in writing a Pascal program, detects logical errors, and makes advises to fix them. The environment permits easy coding using the information of a manual and syntax graph. CAPE does not provide a coding environment, instead it generates code in different languages based on algorithmic specification. INTELLITUTOR comprehends programs by means of semantic pattern matching method where CAPE does not. We didn't intend to have CAPE as a tutor since tutoring is given by instructors in a lecture session. INTELLITUTOR is designed to support coding environments where CAPE supports algorithmic construction to generate an output code, which then can be imported into any coding environment.

CAE model allows students to interact with the system via the problem solving environment, called the virtual world. The student attempts to solve the problem by interacting with the objects to obtain information about the problem in order to solve the problem. CAPE observes the idea of presenting a problem via a virtual world in its problem definition module. The difference is that CAPE does not analyze what the user has reviewed since it describes the problem and presents its solution. The challenge is to solve the problem's exercises. CAE is capable of interpreting student's problem solving attempts and provide a diagnosis of his strategy to the instructor. CAPE, on the other hand, provides the instructor with a mechanism to evaluate student solution by applying the problem's sample data. For our purposes, a program is "correct" if there exist some set of test data for which the program will yield the correct answer. CAPE allows the instructor to identify students with incorrect solutions to further review their constructed algorithm and/or specified code.

5-2 Contribution

Our research in this area has led us to the development of a model that intersects a number of subject areas. Algorithm construction module is our major contribution to programming environments which provides educational capabilities to help students learn programming. It motivates students to study the algorithmic problems operationally and provides a mechanism for rapid prototyping in various programming languages. CAPE could be highlighted in the following points:

- Supports program development process
- Produces code automatically in different procedural languages
- Its structure allows the incorporation of other procedural languages
- An algorithm could be constructed using a simple set of seven statements
- Open for authoring systems to produce lessons for problem definition module
- Open for coding environments to edit, compile, and debug programs
- Aids instructors to explain differences between languages syntax
- Defines an abstract method to explain programming concepts

5-3 Future Directions

A prototype of the model is being developed to accommodate KFUPM laboratory sessions. Information and Computer Science department offers an introductory course to programming using FORTRAN for over 500 students every semester. Laboratories are equipped with personal computers running MS Windows. A group of senior students are implementing the model using Visual Basic 4.0 and MS Access 2.0.

The following is a list of suggested future enhancements to be incorporated into CAPE:

- Determine the type of a variable from specified expression
- Define methods for C language and any other selected language
- Detect two dimensional arrays
- Enrich data structures with pointers

Another direction for future research may lead to Automatic Programming since CAPE aids in constructing algorithms and then generates high level language code. The specified expressions in CAPE are currently mapped as they were specified. An expression analyzer capable of generating the expression in desired language syntax would help in achieving a complete code that requires no further modification.

REFERENCES

1- M. Mullerburg and D. Meyerhoff, "The METKIT CAI system," *Information and Software Technology*, Vo 35 No 2, Feb 1993, pp 120-126.

2- H. Ueno, "Integrated intelligent programming environment for learning programming," *IEICE Transactions on Information and Systems*, vol E77D no 1, Jan 1994, pp 68-79.

3- P. Forcheri and M. Molfino, "Software tools for the learning of programming: A proposal," *Computers & Education*, vol 23 no 4, Dec 1994, pp 269-276.

4- J. Du Plessis, J. Van Biljon, C. Tolmie, and T. Wollinger, "A model for intelligent computer-aided education systems," *Computers & Education*, vol 24 no 2, Feb 1995, pp 89-106.

5- S. Booth, "A study of learning to program from an experiential perspective," *Computers in Human Behavior*, vol 9 no 2-3, Sum-Fall 1993, pp185-202.

6- M. Sammons, "Students assess computer-aided classroom presentations," *T.H.E. Journal*, vol 22 no 10, May 1995, pp 66-69.

7- S. Maddock, "A personal CAL workbook," *Computers & Graphics*, vol 18 no 3, May-Jun 1994, pp 299-304.

8- P. Navrat and V. Rozinajova, "Making programming knowledge explicit," *Computers & Education*, vol 21 no 4, Nov 1993, pp 281-299.

9- M. Reek, "A top-down approach to teaching programming," *SIGCSE Bulletin*, vol 27 no 1, Mar 1995, pp 6-9.

10- R. Schank, "Active learning through Multimedia," *IEEE Multimedia*, Spring 1994, pp 69-78.

11- R. Price, *Computer-Aided Instruction: A Guide for Authors*, Brooks/Cole Publishing Company, Pacific Grove, California, 1991.

12- ICS Department, *ICS 101 Supplementary Material*, King Fahd University of Petroleum and Minerals, Dhahran, 1993.

13- B. Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall International, Hertfordshire, England, 1990.

14- M. Hodges and R. Sasnett, *Multimedia Computing: Case Studies from MIT Project Athena*, Addison-Wesley, 1993.

15- C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, New York, 1987.

16- N. Shu, *Visual Programming*, Van Nostrand Reinhold, New York, 1988.

17- J. Lambert, "Restructuring the introductory computer science course for engineers," *ACM SIGCSE*, 3/1995, pp 204-208.

18- E. Smith and G. Westhoff, "The Taliesin project: Multidisciplinary education and multimedia," *Computers in Human Behavior*, vol 9 no 2-3, Sum-Fall 1993, pp185-202.

19- C. Wolfe, "Using Authorware Professional for developing courseware," *Behavior Research Methods, Instruments & Computers*, vol 24 no 2, May 1992, pp 273-276.

20- Task force of the pre-College Comittee of the education board of the ACM, "ACM model high school computer science curriculum," *Communications of the ACM*, vol 36 no 5, May 1993, pp 87-90.

21- G. Schneider, S. Weingart, and D. Perlman, *An introduction to programming and problem solving with Pascal*, John Wiley & Sons, New York, 1982.

22- J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented modeling and design*, Prentice Hall, New Jersey, 1991.