IN-61-CR
23656

SOFTWARE REUSE ENVIRONMENT USER'S GUIDE

Contract Number NAS5-27684
Task 117

Prepared For:

NASA Goddard Space Flight Center
Data Systems Technology Division
Greenbelt, MD

POC: Walt Truszkowski

March 30, 1988

COMPUTER TECHNOLOGY ASSOCIATES, INC.
14900 Sweitzer Lane, Suite 201
Laurel, MD 20707

(NASA-CR-196885) SOFTWARE REUSE ENVIRONMENT USER'S GUIDE (Computer Technology Associates) 44 p

N95-10827

Unclas

G3/61 0023656

TABLE OF CONTENTS

TABLE OF CONTENTS
(CONTINUED)

LIST OF FIGURES

LIST OF ACRONYMS

ADT	Abstract Data Type
CTA	Computer Technology Associates, Inc.
CTRL	Control key
DBMS	Database Management System
DFD	Data Flow Diagram
DFE	Data Flow Editor
DSD	Data Structure Diagram
EDFD	Entity Data Flow Diagram
ERD	Entity Relationship Diagram
ERDDFD	Entity Relationship Diagram/Entity Data Flow Diagram Consistency Checker
ERE	Entity Relationship Editor
ESC	Escape key
FORTRAN	Formula Translator Programming Language
IDE	Interactive Development Environments, Inc.
OD	Object Diagram
ODGENER	Object Diagram Generator
OOD	Object-Oriented Design
OOS	Object-Oriented Specification
PCT	PICture diagram editor
TTY	Teletype terminal

1 INTRODUCTION

This document describes the services provided by the prototype Software Reuse Environment, which was developed by CTA for NASA Goddard Space Flight Center, Code 520. This is one of three guides delivered by CTA as part of the environment. The other two guides are:

o Software Generation and Installation Guide

o SEMANTX: Defining the Schema

The *Software Generation and Installation Guide* describes the software source modules that make up the Reuse Environment, with instructions on how to generate and install an executable system from the source code.

SEMANTX: Defining the Schema describes how a reuse database is created. Actually this guide is more general than the reuse database, as it describes how to generate a SEMANTX™ database. SEMANTX is an off-the-shelf tool that we have used to implement the reuse database. It is a product of Semantyk Systems, Inc.

The Software Reuse Environment is built upon SEMANTX as well as on the IDE Structured Analysis Integrated Environment™. (IDE is Interactive Development Environments, Inc.) SEMANTX itself is built on top of the Unify™ Database Management System. To use the Software Reuse Environment you should have the User's Manuals for SEMANTX, for Unify, and for the IDE software. CTA has provided all of these with the environment.

You should also have the User's Manual for the Object-Oriented Design Rule Checker. The Object-Oriented Design Rule Checker is a product of CTA. It is intended to facilitate the design process, and can be used effectively in conjunction with the tools of the Software Reuse Environment. The executable program and the manual are being provided by CTA for use at NASA Goddard. While strictly speaking it is a separate tool, it will appear to the user as an integrated part of the environment.

The overall structure of the Software Reuse Environment is shown in Figure 1-1.

Organization of this Guide

This guide consists of a high-level description of the environment (Section 2) followed by more detailed discussions of specific features.

Section 2 presents the tools that are available in the environment. As you will see, all the tools are invoked from a single unifying window by means of mouse clicks and, in some cases, additional parameters for which you are prompted. The emphasis in Section 2 is on what tools are available, how they are invoked, and the typical way in which you would use them.

In Section 3 we describe the Object-Oriented Specification (OOS) methodology, which we recommend as a way of describing software requirements in this environment. The Software Reuse Environment is intended for reuse of products *throughout* the development lifecycle, from specification through coding and testing. By using OOS, the

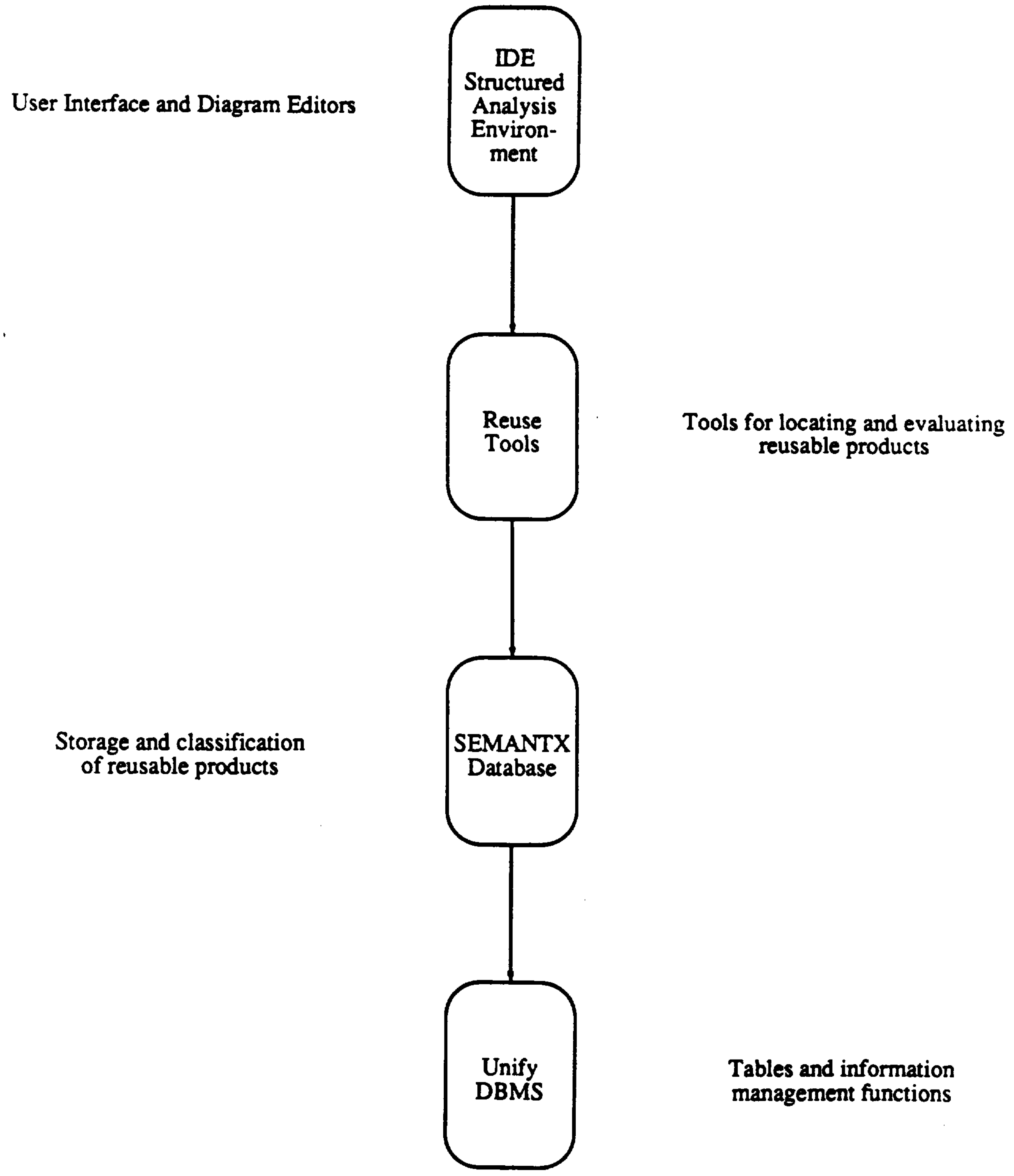
User Interface and Diagram Editors
IDE Structured Analysis Environment
Reuse Tools
Tools for locating and evaluating reusable products
Storage and classification of reusable products
SEMANTX Database
Unify DBMS
Tables and information management functions

Figure 1-1: Overall Structure of the Reuse Environment

analyst can specify requirements in a manner that facilitates future reuse. Because of the form that an Object-Oriented Specification takes, developers can easily understand what is available in the Reuse Database.

In Section 4 we explain how to use the tools that support OOS. This topic is covered in more detail than it was in Section 2. The key activities described in this section are *checking the specification* for consistency and completeness, and *annotating the specification* to automatically generate a design.

In Section 5 we describe the Design Quality Assessor. This tool can help you ensure adequate standards for software placed in the Reuse Database.

Section 6 covers the Reuse Database itself. We describe what *types of products* are stored in the database, the different *relationships* that can hold between products, and the means available for *classifying* products. To enter either products or classification keywords into the Reuse Database we use the SEMANTX tool. This procedure is described in the SEMANTX User's Guide, and is not covered in this document.

Finally, in Section 7, we describe how you can search for reusable products directly from the IDE diagram editors. Through this feature, reuse becomes an integrated part of the specification and design process. The basic sequence of activities is to *annotate the diagram objects* with keywords, *initiate a search* for candidate reusable products, *view the attributes* of the candidate products, *navigate the database* for possibly better choices, and *select one or more products* for reuse.

The Software Reuse Environment is a prototype, and both the software as well as this User's Guide are intended to evolve. We welcome comments and suggestions on how the environment and/or the documentation can be improved.

2 THE MAIN SCREEN AND THE TOOLS PROVIDED

The tools of the Reuse Environment are invoked from a main screen called the IDEtool screen. To bring this screen up, enter the following command at your keyboard:

```
IDEtool
```

The use of IDEtool is described in detail in the IDE documentation. Here, we will touch on the main points, focusing on the features that are unique to the Reuse Environment.

If you have entered the command shown above, you will see that the IDEtool screen consists of 5 panels, layed out vertically one on top of the other. In the following subsections we explain the function of each panel.

2.1 Top Panel: Diagram Names and the Execute Button

The top panel serves two main purposes:

o To enter a diagram file name, with appropriate directory qualifications

o To invoke tools by means of the Execute button

A key point to remember about the Execute button is that *nothing happens until you click it!* You must do this each time you execute a command.

You can also turn "Help Mode" on and off using the Help Mode button. In Help Mode, clicking any other field on the screen (or hitting the RETURN key in a text field) causes a Help window for that field to be displayed. You stay in Help Mode until clicking the Help Mode button again. While in Help Mode you *cannot* execute commands or even set up commands. (Setting up commands is discussed in the next few Subsections.)

2.2 Second Panel: Command Groups

The second panel lets you select a "command group." The principal groups are those associated with the different types of diagrams, i.e., there is a group of commands for Data Flow Diagrams (click the icon labeled DFE---Data Flow Editor), for Entity-Relationship Diagrams (click ERE), and for Picture diagrams (click PCT). There are other command groups that do not apply directly to a type of diagram. These are represented without icons, on the right-hand side of the panel. For our purposes, the most important one is labeled "Reuse Database."

2.3 Third Panel: Commands

Notice that when you change your selection of a command group, the contents of the third panel change. The third panel shows you the commands that are available in the currently selected command group. Let us summarize the commands that are most relevant to the Reuse Environment:

Under DFE we have the following commands:

o Edit Entity Data Flow Diagram

o Check Object-Oriented Specification

o Generate Object Diagrams

''Edit Entity Data Flow Diagram'' differs from the command just preceding it, ''Edit Data Flow Diagram,'' by giving you access to the Reuse Database. (This is discussed in Section 7.) Data Flow Diagrams (DFDs) are just what you are familiar with from Structured Analysis. Entity Data Flow Diagrams (EDFDs) are a variation intended to be used in Object-Oriented Specification (OOS).

''Check Object-Oriented Specification'' invokes an OOS consistency checker. ''Generate Object Diagrams'' is a command you can use to create a first approximation to a design, automatically, on the basis of your OOS. These tools are described in Sections 4.1 and 4.2, respectively.

In the ERE group we have the following commands:

o Edit Entity Relationship Diagram

o Check Object-Oriented Specification

''Check Object-Oriented Specification'' is, therefore, available under both DFE and ERE. This is because an OOS consists of both Entity Relationship Diagrams (ERDs) and EDFDs. The checker exists to ensure that the two types of diagrams are consistent with each other. Consult Section 4 for more on this.

Picture (abbreviated as PCT) is the general-purpose diagram editor. Using it you can create pretty much any kind of engineering diagram you wish. It provides a wide variety of icons and arc types, and does not enforce any syntactic rules. In the Reuse Environment we use Picture to draw Object Diagrams. Under PCT we have the following commands:

o Edit Object Diagram

o Check Object-Oriented Design

o Assess Object-Oriented Design

''Edit Object Diagram'' is similar to ''Edit Diagram,'' which immediately precedes it in this group. The difference is that ''Edit Object Diagram'' provides some additional options, such as decomposing procedures and state objects (Section 4.3) and searching for reusable objects (Section 7).

''Check Object-Oriented Design'' causes the OOD Rule Checker to be invoked. The rule checker can help you achieve a complete and consistent set of Object Diagrams. ''Assess Object-Oriented Design'' activates the Design Quality Assessor. This tool is described in Section 5.

Finally, under Reuse Database (on the right-hand side of the second panel) we have these commands:

o Navigate Products

o Navigate Keywords

o List Product Arctypes

o List Keyword Arctypes

o Show Detail

o Post-Search Options

The Navigation and List Arctype commands are used to examine the contents of the Reuse Database. The Show Detail command lets you look at the attributes of a product in the Reuse Database. These attributes describe the product in some detail, and point you to additional information. See Section 6 for more on these commands.

The Post-Search Options command is intended for use *only* after activating a search of the Reuse Database from one of the diagram editors. If you invoke the Post-Search Options at any other time they will simply not do anything (feel free to try, though). These options allow you to examine the results of a search, and to select one or more products for reuse in a specification or design. The correct use of the Post-Search Options is described in Section 7.

2.4 Fourth Panel: Command Arguments

As you select different commands, the contents of the fourth panel may change. In this panel you are prompted for any arguments required by the command you just selected.

Be sure to hit the RETURN key after entering a text argument. And remember, *nothing* happens until you click Execute in the top panel.

2.5 Fifth (Bottom) Panel: Command Input and Output

The bottom panel acts as a TTY screen for the command you invoke. It works just as if you had typed the command in reponse to an Operating System prompt.

Any screen output of the command will appear in this bottom panel. In addition, if the command prompts you for more input, you should enter that input in this bottom panel. Be sure to move the mouse cursor into the panel before typing.

You should acquire the habit of checking this panel each time you execute a command. Look for any status, error, or other messages that may be issued by the program you have invoked.

3 OBJECT-ORIENTED SOFTWARE SPECIFICATION

This section describes a method of analyzing requirements for object-oriented software. The method is intended to flow smoothly into the Object Diagram design method, and from there into programming with Ada. Our method is intended to serve as an alternative to Structured Analysis when the use of an Object-Oriented Design method is foreseen.

We assume that the analyst who is using this method has available a textual statement of requirements for a software system. Ideally, the analyst would already have distilled the requirements statement into a database of discreet, trackable requirements. This is not necessary for the method we describe here, but it is highly advisable. **The steps described below are a method for understanding and articulating the implications of the original requirements (whether in text or database form).**

Structured Analysis is the specification method most widely used in Code 500. Structured Analysis is a method of articulating *functional* requirements. The transition from a Structured Analysis specification to Object Diagrams can be awkward. The criteria for grouping functions in Object-Oriented Design are quite different from those used during Structured Analysis. In Structured Analysis, each process bubble represents a transformation of inputs to outputs, and decomposition consists of asking what steps are involved in this transformation. The constituent steps, however, may operate on entirely different data abstractions. If this is the case, then from an object-oriented viewpoint the sub-operations belong in different objects, with a chain of messages effecting the sequence of operations.

The Object-Oriented Specification method involves creating a hierarchy of Entity Data Flow Diagrams (EDFDs) within the context of an Entity-Relationship model. Each active entity is represented as a process in an EDFD. Each passive entity is represented as a data flow. Lower level EDFDs decompose each active entity either into subentities or into functions performed by the entity.

The specification method consists of seven steps:

1. Identify key problem-domain entities.

2. Distinguish between active and passive entities.

3. Establish data flow between active entities.

4. Decompose entities (or functions) into sub-entities and/or functions.

5. Check for new entities.

6. Group functions under new entities.

7. Assign new entities to an appropriate domain.

Steps one through three are, in principle, performed once. We may, however, backtrack at any time to reconsider these steps. Steps 4 through 7 should be iterated until the desired level of detail is reached.

3.1 Identify Key Problem-Domain Entities

Identification of the "right" problem-domain entities is crucial to the development of a stable design. There is no hard and fast way to define what is and what is not a problem-domain entity. Part of the analysis process is to assess the consequences of including (or not including) potential problem-domain entities. Ask the following questions:

o Is every known functional requirement met by one of the entities?

o Is the internal state of the system adequately represented by the states of the entities?

Chapter 1 of Jackson's book *System Development* [2] discusses these issues in detail.

A heuristic procedure for identifying the problem-domain entities is suggested by Grady Booch in the first edition of his book, *Software Engineering with Ada* [3]. The procedure is to create a list of the key nouns and noun phrases from the original (textual) statement of requirements. Such a list may give you a first approximation of the problem-domain entities. You should take this heuristic with a grain of salt, however. Booch even omitted the procedure from the second edition of his book. In a typical requirements specification there will be many, many nouns that are completely irrelevant to the problem-domain entities. You must therefore be highly selective when examining the specification for key nouns.

A more realistic variation on this approach would be to locate the nouns and noun phrases in the *requirements database* (if there is one). A requirements database distills the original textual statement into a trackable set of requirements, in a form that is less verbose than the original statement. In performing such distillation, engineering judgment and selection are applied to the original statement. The chances of your identifying significant nouns from the requirements database are therefore greater than if you were to use the original requirements statement as your source.

A still more manageable approach draws on techniques from Structured Analysis. Process names in Structured Analysis are typically of the form "verb_object," e.g., *monitor_status, control_payload, validate_commands*. The *nouns* in such names refer to problem-domain entities. As a heuristic procedure, you may want to draw a Context Diagram and a level-0 Data Flow Diagram in order to determine the problem-domain entities by this "verb_noun" method. These diagrams will not be part of the Object-Oriented Specification. They serve only as guides to help you in identifying entities.

Draw an Entity-Relationship (ERD) diagram to record the problem-domain entities and their inter-relationships. The IDE Entity-Relationship Editor allows you to draw such diagrams interactively. You may also want to create an Entity Dictionary, which is analogous to the Data Dictionary in Structured Analysis. For more information on this approach, refer to the paper by Stark and Seidewitz, "Towards a General Object-Oriented Ada Lifecycle" [4].

3.2 Distinguish between Active and Passive Entities

We define active and passive entities as follows:

An **active entity** is an entity whose functions you want to consider in the specification phase. A **passive entity** is one whose functions you do *not* want to consider until the design phase.

This is a little different from the definition you might expect. Intuitively you might expect us to define an active entity as any entity that performs functions on other entities. A passive entity would be any entity that has functions performed *on it*. This is not far from the definition we use, but it is not exactly the same. **The main point to remember is that active entities will be represented by process bubbles in the Entity Data Flow Diagrams, while passive entities will be represented as data flows and/or data stores.**

3.3 Establish Data Flow between Active Entities

In this step we draw a top-level Entity Data Flow Diagram (EDFD). EDFDs are similar to the Data Flow Diagrams with which you are familiar from Structured Analysis, but the interpretation of the diagrams is different.

In an EDFD, the process bubbles represent either *entities* or *functions*. A function is simply a transformation of inputs to outputs, as in Structured Analysis. An entity may perform one or more functions, i.e., it may transform inputs to outputs, but it also has an internal *state* which evolves, in time, as a result of inputs received.

Create a level-0 process for each top-level active entity in the Entity-Relationship model. Passive entities should appear as data flows or data stores. The ERD does not explicitly distinguish between top-level and lower-level entities, so if you have not already made this distinction you will have to do so now. Typically, your ERD may show relationships with names like ''contains'' or ''includes.'' This is an indication that that ''contained'' or ''included'' entity should appear in some lower-level EDFD, rather than at level 0.

3.4 Decompose Entities (or Functions) into Sub-entities and/or Functions

Steps 4 through 6 form the heart of the specification method. When we perform Step 4 the first time, we are decomposing a top-level EDFD which consists of active entities and data flows. Some of the data flows are passive entities. We now decompose each active entity in this diagram into *subentities and/or functions*. The subentities are entities out of which the larger entity is *composed*. The functions (if there are any) are *performed by* the entity being decomposed. **You should differentiate between entities and functions by placing an asterisk in front of the name of an entity.**

In subsequent passes through this step you will be decomposing both entities *and* functions. You can decompose an entity into subentities and/or functions. Functions, however, may be decomposed only into subfunctions.

In identifying functions, consider what the entity *does*. Ask again (as you did in Step 1) whether every known functional requirement of the system can be met by one or more of the functions identified. A heuristic approach, similar to that described in Step 1, may be used to identify functions. In this case, instead of locating the nouns in the requirements specification, you can locate the verbs and verb phrases. Some of these verbs will refer to functions that the system is supposed to perform. We emphasize, once again, that judgment is required in selecting the significant verbs from the original requirements specification. Typically, such judgment and selection are exercised when creating a *requirements database* from the original textual specification.

If, in Step 1, you identified the subjects and/or objects of these verbs as problem-domain entities, you should now associate the functions with the corresponding problem-domain entities (subject or object of the verb).

3.5 Check for New Entities

At each stage of decomposition, you should STOP and ask whether any new entities are implied by the functions you have introduced. We recommend using the ''verb_object'' heuristic procedure, which was described in Section 3.1. Put all function names into the form ''verb_object,'' and examine these names for ''objects'' not previously identified. Data stores and data flows should also be considered as potentially representing passive entities. Ask yourself whether the data store or data flow is significant enough to include in the Entity-Relationship model.

If you discover new entities that should be included in the specification, you then have to decide where to place them. You may, for example, replace a function (introduced in Step 4) by a new entity. The function might then be placed one level down, below the new entity. We do not recommend any specific guidelines for placing the new entities, except to note that you may have to reorganize your diagrams at this point. **It is in this step and the next that you should consider such issues as strength and coupling, visibility, and the quality of entity abstractions.**

3.6 Group Functions under New Entities

If you have introduced new entities in Step 5, you must now identify all the functions performed by (or on) the new entities. This is how you ensure a faithful representation of the entities of the system. This feature, which we call *functional completeness*, is characteristic of the object-oriented approach. It provides object-oriented specifications (and subsequent designs) with stability under evolving functional requirements. By including all functions performed by an entity, whether they are immediately required or not, you increase the chances of being able to respond to future functional enhancements with minimal respecification and redesign. You also increase the probability that the components developed will be reusable in other systems.

3.7 Assign New Entities to Appropiate Domains

This last step is intended to help you manage the complexity of proliferating entities. We recommend that you assign each new entity (introduced through Step 5) to some *domain*. The domain may be new. If your Entity-Relationship model is becoming complex, you may want to represent each domain in a separate ERD. This will prevent the original ERD (which represents the application or problem domain) from growing too large to visualize. The *set* of ERDs, taken together, provides a comprehensible view of the entities of the system.

You may find it helpful to consider different *levels* of entity domains. For example, the following list describes a hierarchy of *types* of domains. This is a kind of ''virtual-machine'' hierarchy, in so far as higher-level domains *use* the domains at lower levels:

1. Application Domains
2. Technology Domains
3. Computer Science Domains
4. Execution Models
5. Execution Domains

During the specification phase you will probably be concerned only with an application domain and, possibly, some technology domains. Some examples of technology domains are Communications, Database Management, Signal Processing, and Graphics. As you approach the design phase you may begin to address classes of algorithms and implementation approaches. When this happens, you may want to identify the Computer Science Domains that will provide the conceptual repertory for these decisions.

4 OBJECT-ORIENTED SPECIFICATION TOOLS

The Object-Oriented Specification tool set consists of five tools:

o Entity-Relationship Editor (**ere**)

o Data Flow Editor (**dfe**)

o Entity-Relationship Diagram/Entity Data Flow Diagram consistency checker (**erddfd**)

o Object Diagram Generator (**odgener**)

o Object Diagram Editor (**picture**)

o Object Diagram Subprogram and Data Structure Edit Feature (**picture** option)

The **ere**, **dfe**, and **picture** programs are part of the IDE Structured Analysis Integrated Environment, and are documented in the IDE User's Manual. CTA has developed the programs **erddfd** and **odgener**, as well as the Subprogram and Data Structure Edit Feature.

4.1 The Diagram Editors

As you read in Section 3, an Object-Oriented Specification consists of one or more Entity-Relationship Diagrams (ERDs) and a hierarchy of Entity Data Flow Diagrams (EDFDs). *To edit an ERD, do the following:*

1. Select the ERE icon in the second panel of the IDEtool screen

2. Enter the name of the diagram you want to edit in the first panel

3. Select ''Edit Entity Relationship Diagram'' in the third panel

4. Click the ''Execute'' button in the first panel

Alternatively, you can enter

ere *filename*

to invoke the editor directly from the Operating System prompt, without IDEtool.

To edit an EDFD, do the following:

1. Select the DFE icon in the second panel of the IDEtool screen

2. Enter the name of the diagram you want to edit in the first panel

3. Select ''Edit Entity Data Flow Diagram'' in the third panel

4. Click the ''Execute'' button in the first panel

You may also invoke this editor directly, by entering

dfe *filename*

but if you do this you will not be able to access the Reuse Database from the editor (see Section 7). The same will be true if you select ''Edit Data Flow Diagram'' instead of ''Edit *Entity* Data Flow Diagram'' in the IDEtool third panel.

4.2 Identifying Entities in an EDFD

According to the OOS method, described in Section 3, we need to be able identify the entities in an Entity Data Flow Diagram (EDFD). We do this by means of a naming convention. The name of each entity (whether a process or a data flow) is prefixed with an asterisk (*).

For example, an entity with the name *spacecraft* in the Entity-Relationship Diagram (ERD) would appear in some EDFD (probably the level-0 EDFD) as a process with the name **spacecraft*. In the EDFD below **spacecraft* there might appear subentities, such as **platform* and **sensors*. Both *platform* and *sensors* would also appear as entities in an ERD. The EDFD below **platform* might contain functions such as *monitor_sensor_status* and *control_envelop.*

Passive entities are represented as data flows. For example, if an entity named *sensor_data* appears in an ERD, then there should be some data flow with the name **sensor_data* in some EDFD.

4.3 Checking the Entity Data Flow Diagrams

Before checking the mutual consistency of the ERDs and EDFDs, you should probably check the consistency of the EDFDs themselves. To check a single diagram do the following:

1. Select the DFE icon in the second panel of the IDEtool screen

2. Enter in the first panel the name of the EDFD that you want to check

3. Select ''Check Data Flow Diagram'' in the third panel

4. Click the ''Execute'' button in the first panel

To check an entire EDFD hierarchy use the same procedure but select ''Check Decomposition'' instead of ''Check Data Flow Diagram.''

The DFD rule checker is part of the IDE tool set, and is documented in the IDE User Manual.

4.4 Checking an Object-Oriented Specification

The consistency checker developed by CTA focuses on the relationship between the ERD and the EDFDs. *While the OOS method described in Section 3 advocates* *drawing multiple ERDs (one for each pertinent domain), the OOS rule checker currently supports only a single ERD.* The name of the ERD and the name of the level-0 EDFD shold be identical except for their extensions, i.e.,

filename.ere and *filename.dfe*

The consistency checker applies the following rules to the ERD and the EDFD hierarchy:

o Each entity in the ERD must appear in some EDFD

o Each entity in an EDFD must appear in the ERD

o If two entities are related in the ERD, then they must be related in the EDFDs as follows:

- If both entities are active, i.e., they are represented as EDFD processes, then either one is a subprocess of the other, or there is a dataflow between them.

- If one entity is a process and the other is a dataflow, then the dataflow must either be contained in the process, or be an input or an output of the process.

You can invoke the OOS checker as follows:

1. Select either the ERE or DFE icon (second panel)

2. Enter in the first panel the name of the ERD *without* the extension ".ere" (this should be the same as the name of the level-0 EDFD without *its* extension)

3. Select the command "Check Object-Oriented Specification" (third panel)

4. Click the "Execute" button (first panel)

You can also invoke the OOS checker directly from the Operating System prompt, without IDEtool, by using the following command line:

erddfd *erd_name dfd_name*

In this case *erd_name* and *dfd_name* may be different, and the file name extensions (".ere" and ".dfe") may be included or omitted.

4.5 Generating Object Diagrams Automatically

The specification phase ends when entities and functions have been decomposed to an appropriate level. The Object Diagram Generator may then be used to create, automatically, an initial set of Object Diagrams based on the specification. These Object Diagrams constitute a first cut at a design that meets the specification. See reference [5] for an introduction to the Object Diagram methodology.

Before applying the Object Diagram Generator to a hierarchy of EDFDs, you must annotate the EDFDs with certain design information as follows:

- Use a pound sign (#) to identify independent threads.

  For each function that runs as an independent thread, place a pound sign in front of the function's label. For example:

  #monitor_sensor_status

  Do *not* annotate entities with the pound sign. Remember, entities are those processes whose labels begin with an asterisk.

- Use an ampersand (&) to identify reusable or replaceable functions.

  A function is reusable, for example, if it occurs under several entities. If two functions appearing under different entities appear to be similar, consider whether they might be realized by the same (reusable) function. You may also wish to consider whether a function could be reused in other systems.

  A function is potentially replaceable if 1) the function depends heavily on the computer environment or on other information that is prone to change, or 2) the implementation of the function is likely to change, with the interface to the function staying the same.

  Place the ampersand in front of the function's label. For example:

  &create_sfdu

  Do *not* annotate entities with an ampersand.

- Use a question mark ("?") to indicate that the direction of control flow is opposite to the direction of a data flow. For example,

  ?sensor_data

  indicates that the recipient of *sensor_data* initiates the data transfer, not the sender. If the question mark is omitted, the default assumption is that control flow and data flow are in the same direction.

- Use an asterisk to identify instances of Abstract Data Types (ADTs).

  Place the asterisk at the beginning of the data flow or data store's label.

You may now execute the Object Diagram Generator. An Object Diagram will be created for the level-0 EDFD, and for each EDFD whose parent process is an entity (i.e., annotated with an asterisk). The EDFD annotations are used to determine how the processes, data stores, and data flows will be translated into Object Diagram items. The translation rules are as follows:

- Entities (identified by an asterisk) become proper objects.

- Data stores become state objects.

o Concurrent functions (identified by the pound sign) become tasks.

o Nonconcurrent functions (i.e., no asterisk and no pound sign) become procedures.

Two objects will be connected by a control flow if their corresponding processes are connected by data flows. If the data flow and control flow are in the same direction, then the data flow becomes an input parameter to the called object; otherwise, the dataflow becomes an output parameter of the called object.

Each ADT becomes a proper object in the level-0 Object Diagram. Each object that accesses the ADT will be connected to the ADT by an arrow. (For the Object Diagram Generator, an object "accesses" the ADT if the original process, from which the object was created, is either a source or destination of the ADT dataflow.) These arrows may be "off-edge." If so, additional arrows are generated to connect the objects' ancestors to the ADT.

The "provides" clause of each object is created on the basis of the object's input and output parameters. A procedure object provides one item, namely itself, with the appropriate input and output parameters. A state object provides itself. A task provides an entry for each subfunction that communicates to outside of the task. A proper object provides those procedures, entries, and states that are required to accept/generate the object's input/output parameters.

The "uses" clause of each object is generated on the basis of 1) the arrows emanating from the object, and 2) the "provides" clauses of the objects at the other end. The items used are determined by the parameters passed between the two objects.

Reusable functions (identified by the ampersand) are all moved into a single level-0 object---a "library package"--- to ensure that they are potential visible to any object that needs them. An arrow is drawn to the service package from each object that, prior to the move, was an ancestor of a reusable function. For each such arrow, the "uses" clause of the former ancestor is updated accordingly.

The Object Diagrams generated automatically are only a rough draft of a design. Exception handling must still be added. Components may have to be moved to improve the design quality or to better address the requirements. The entries of tasks may have to be changed: the generator can only guess at this. The eventual design may not correspond to the decomposition of the EDFDs. The Quality Assessor, described in Section 5, can provide advice on refining the design.

To invoke the Object Diagram Generator, do the following:

1. Select the DFE icon (second panel)

2. Enter in the first panel the name of the level-0 EDFD

3. Select the command "Generate Object Diagrams" (third panel)

4. Click the "Execute" button (first panel)

You can also invoke the Object Diagram Generator with the following command line:

odgener *filename*

where *filename* identifies the level-0 EDFD, with or without the ".dfe" extension. The name of the top-level Object Diagram will be the name as that of the level-0 EDFD, but with the extension ".pct" instead of ".dfe." The name of a sub-Object Diagram is the name of the object it decomposes, followed by the ".pct" extension.

4.6 Subprogram and Data Structure Edit Feature

The Object Diagram Generator maps EDFD entities (identified by an asterisk) to proper objects. Functions that are immediately below an entity are mapped to procedure or actor objects, i.e., to the bottom of the Object Diagram hierarchy. Functions at lower levels of the OOS are not mapped into the Object Diagrams at all. These functions are not part of the object-oriented view of the system. They are subfunctions of the procedures and actors provided by the objects.

The Subprogram Edit Feature provides a way to view and edit the subfunctions of Object Diagram procedures. Each procedure can be "pushed" or "exploded" into a DFD that represents its functional decomposition. If such a DFD was part of the specification, it is retrieved by the Subprogram Edit Feature. If not, a new DFD is created.

The Subprogram Edit Feature enables you to view (and edit) the subprogram hierarchy beneath an Object Diagram procedure. The Data Structure Edit Feature provides a similar capability for state objects. You can view and edit the data structures of which each state object is composed.

Both features are accessed through the **options** item in the PICture menu. This menu is displayed by holding down the right mouse button with the mouse cursor inside the Object Diagram. The **options** item is selected by moving the mouse cursor to the item, and then sliding the mouse cursor to the right (this is the 'walking menu' feature of PICture). An **options** menu then appears. The **options** menu contains an item called **push**; when selected, this menu item invokes the Subprogram and Data Structure Edit Features. The **push** item is selected by moving the mouse cursor to the item, and then releasing the right mouse button.

Do not confuse the **push** item in the **options** menu with the **push** item on the PICture command panel. They serve analogous purposes, but they invoke different editors. The PICture (i.e., standard) **push** cause PICture to be invoked on a lower-level diagram. The **push** in the **options** menu causes either the Data Flow Editor or Data Structure Editor to be invoked.

When you select **push** from the **options** menu, you will be prompted to select a symbol from the Object Diagram. You should select either a procedure symbol (i.e., a rectangle) or a state symbol (parallel lines). The symbol is selected by clicking the left mouse button with the mouse cursor positioned within the symbol.

Depending on whether the selected symbol is a procedure or a state, either the Data Flow Editor (for procedures) or the Data Structure Editor (for states) will be invoked. The name of the diagram (DFD or DSD) is obtained by appending '.dfe' or '.dse' to the label of the selected symbol. For example, if you select a procedure labeled *verify_constraints*, the Data Flow Editor will be invoked on a diagram called *verify_constraints.dfe*. If a state object labeled *parameter_database* is selected, the Data Structure Editor will be invoked on a diagram called *parameter_database.dse*.

5 ASSESSING OBJECT-ORIENTED DESIGNS

This section describes a tool for evaluating object-oriented designs. The tool assumes a design expressed as a hierarchy of *Object Diagrams* [5]. The need for an automated quality assessor stems from the following reasons. First, it is often difficult to decide what constitutes a "good" object. As designers we need testable criteria for deciding how to group procedures and data. Secondly, applying such criteria manually to the design of a large system is prone to error. There are too many interfaces to be coordinated.

Our approach to quality assessment is to flag undesirable design constructs. For convenience, we distinguish between three levels of severity:

- o Questionable: The design construct *may* be appropriate (even necessary) within the context of your design---but you should adopt it only after you have considered its drawbacks and evaluated the trade-offs.
- o Undesirable: The design construct should be changed. It is potentially harmful to the reliability, maintainability, or reusability of the software, and there is almost always a more effective way to accomplish the design goal.
- o Hazardous: The design construct is undesirable and introduces a high risk to reliability.

The constructs flagged by the quality assessor fall into two categories:

1. Weak abstractions
2. Inadequate information hiding

Abstraction quality addresses the question of what is a good object. *Information hiding* refers to the protection of data from improper use.

5.1 What is a Good Object?

During the specification phase we thought of objects as entities that occurred naturally in some domain, e.g., the application domain, a technology domain, or a computer domain. This view is equally valid for software design, but at this more detailed level the following more precise definition is often useful:

An object is a data structure together with procedures that act on the data, such that the procedures are tightly coupled to the data and are not tightly coupled to anything else.

To build complex objects (i.e., objects within objects), simply substitute "previously defined object" for "data."

With this definition of "object," the question of "good" objects reduces to defining "tight coupling." It is simple to define tight coupling between procedures and data: a procedure is *coupled* to the data it accesses; *tight* coupling occurs when the access is direct, as opposed to access *through* some other procedure. For example, in

Figure 5-1, Procedure_A and Procedure_B are both coupled to the Data_Structure_C, but only Procedure_B is *tightly* coupled to Data_Structure_C.

It is not as easy to define tight coupling between procedures, but there are some clear-cut cases. Two objects, for example, are coupled if one uses the other. The coupling is *tight* if the two objects use *each other,* as in Figure 5-2, or, more generally, if there is a "uses" cycle containing them both. The coupling is even tighter if two procedures within the two objects call each other, i.e., are mutually recursive (Figure 5-3). In such cases there is clearly something wrong with the design. This is an example of how the technical definition of "object" can lead to automated quality assessment criteria.

5.2 Methods of Information Hiding in Ada

We single out information hiding as an assessment criterion because designers traditionally do not hide enough information. The legacy of languages less structured than Ada---starting with FORTRAN, in which all shared variables are globally visible---has fostered a style of design and programming in which visibility is the default. Thoughts of future requirements may, for example, lead us to suspect that *some day* we will need to access a certain variable or procedure, even though there is no such need today; so we make it visible to all of the software. Object-oriented design is a more systematic approach to accommodating future needs. In OOD we structure the software so that *potentially* needed information is placed in the external interface of an object; the information is not *actually* visible to another unit of software until the unit explicitly declares such a requirement---in Ada, by means of the "with" statement.

Information hiding in Ada refers primarily to the distinction between the external interface and the internal content of an object. The external interface, which consists of *callable procedures*, *type declarations*, and *visible variables*, is all that is known to users of the object. In Ada the external interface is defined by means of a *package specification*. Procedures and variables that are defined within the object (i.e., the *package body*) but not within the external interface (the package specification) are inaccessible from outside the object.

Data encapsulation is a form of information hiding in which variables are shielded from improper access. All access to the variables occurs through a set of procedures within the encapsulating object. In Ada this is typically accomplished through *private types*. The variable itself may be declared and referenced outside the object, but the *data type* of the variable is known only by a "meaningless" name. The variable's subfields, therefore, cannot be altered or even examined in detail except by calling one of the privileged procedures.

Generics are another means of hiding information. When the encapsulating package is defined to be generic, even the "privileged" procedures---those within the package---are ignorant of the data type on which they are operating. They must operate in a uniform manner that works correctly regardless of the data type. This is one way to prevent improper use of the internal structure of data.

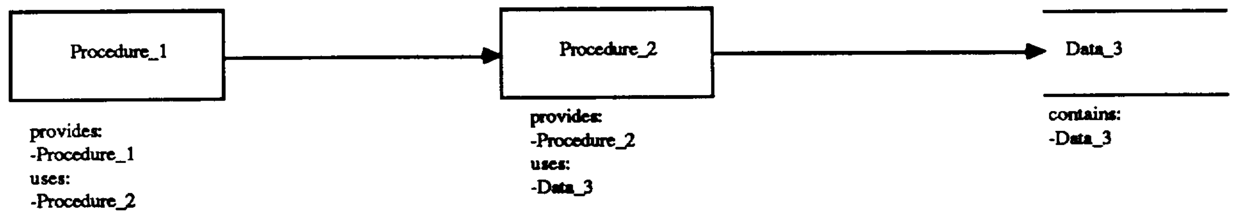

Figure 5-1: Coupling to Data May be Direct or Indirect

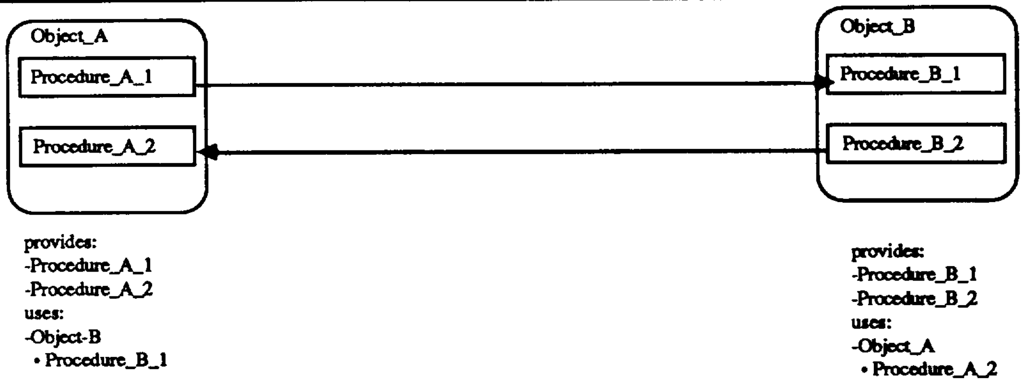

Figure 5-2: Mutual Use Indicates Tight Coupling of Objects

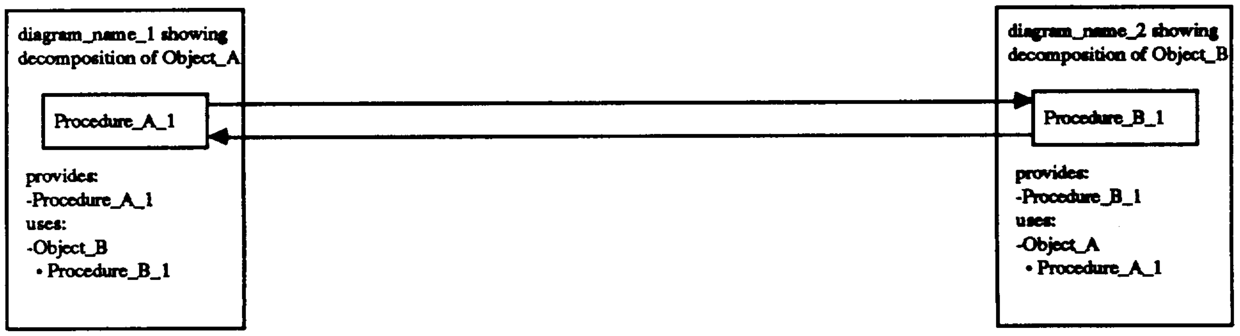

Figure 5-3: Mutual Recursion Over Object Boundaries Indicates Undesirable Coupling

5.3 The Quality Checks

From the criteria discussed in Section 5.1 and 5.2 we have extracted a handful of quality checks. This section describes the specific constructs searched for, and diagnostic messages issued.

5.3.1 Unencapsulated Data

Data structures that occur in a top-level Object Diagram are by definition not encapsulated. When the Quality Assessor encounters such a structure, as illustrated in Figure 5-4, it issues the following message:

```
Diagram: top_level_diagram_name
State object "Data_C" has no parent
and can be accessed by ALL other objects.
Severity level: Undesirable.
```

5.3.2 Data Accessed from Outside the Containing Object

Figure 5-5 illustrates another case of insufficient data encapsulation. The Quality Assessor's response is as follows:

```
Diagram: diagram_name
State object: "Data_B_1"
Accessed by procedure "Procedure_A_1"
Accessed by procedure "Procedure_B_1"
Severity level: Questionable.
```

5.3.3 Data Updated by More than One Procedure

Access control issues can arise even when data structures are encapsulated. For example, should more than one procedure be able to update the data? The quality assessor will flag as questionable any such instances, as in Figure 5-6:

```
Diagram: diagram_name
State object: "Data_3"
Altered by procedure "Procedure_1"
Altered by procedure "Procedure_2"
Severity level: Questionable.
```

5.3.4 Concurrent Updates Possible

In Figure 5-7, two program units that update the same data are capable of running concurrently. Such cases are flagged as hazardous.

```
Diagram: diagram_name
State object: "Data_3"
Altered by task "Task_1"
Altered by procedure "Procedure 2"
Severity level: Hazardous.
```

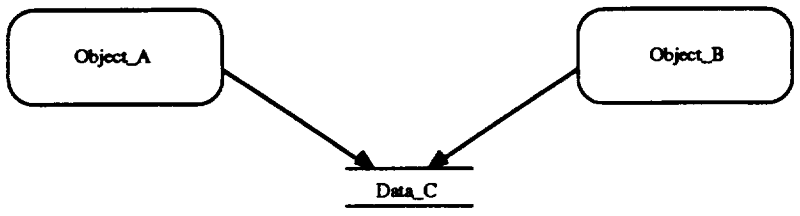

Figure 5-4: Data Structures in a Top-Level Object Diagram are Potentially Visible to any Other Object

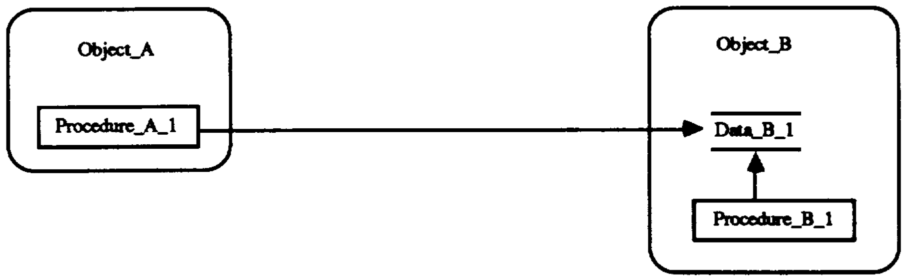

Figure 5-5: Data May be Insufficiently Shielded from Procedures Outside the Containing Object

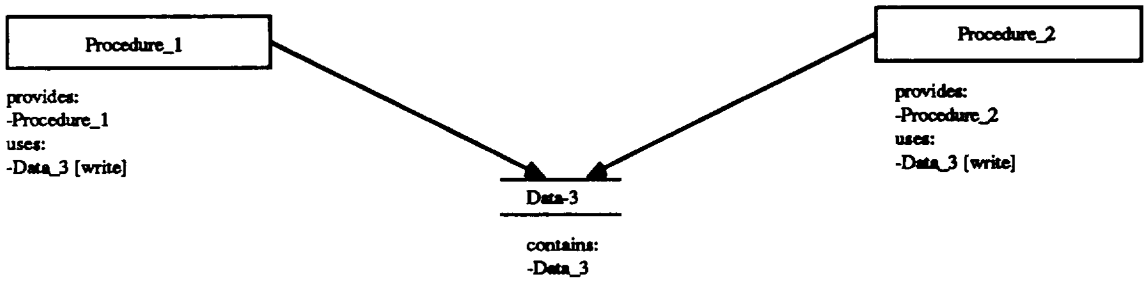

Figure 5-6: The Designer is Notified if a Data Structure can be Updated by More than One Procedure

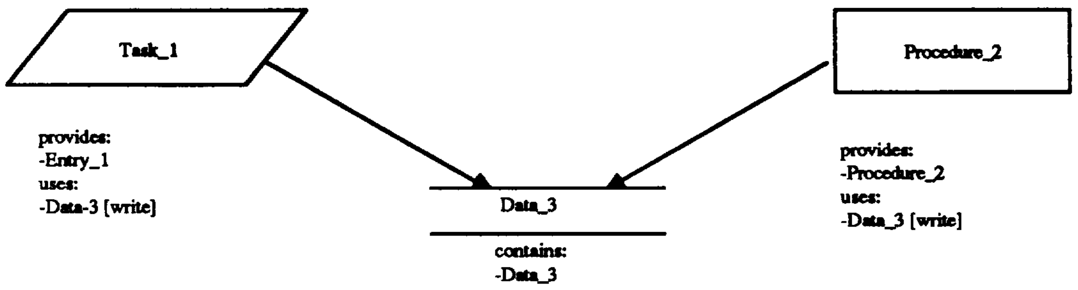

Figure 5-7: Potentially Concurrent Updates are Flagged as Hazardous

5.3.5 Mutually Recursive Procedures or Tasks in Different Objects

As we noted above, the construct illustrated in Figure 5-3 (above) suggests that the objects have not been well thought out. The Quality Assessor issues the following diagnostic:

```
Procedure "Procedure_A_1" within diagram "diagram_name 1" and
procedure "Procedure_B_1" within diagram "diagram_name 2"
use each other.
Severity level: Undesirable
```

5.3.6 Virtual Machine Levels Crossed

A questionable form of coupling occurs when one object uses another object which is more than one *virtual machine level down* from the first object. An object is assigned a virtual machine level according to 1) the set of objects it uses, and 2) the set of objects that use it. For example, if two objects use each other then they are at the same virtual machine level. If one object uses another object, but there is no "uses" chain in the opposite direction, then the "using" object is assigned to a higher virtual machine level than the "used" object. In the purest virtual machine hierarchy, objects at each level would use only objects at the next lowest level. Exceptions to such an arrangement are flagged by the assessor as questionable. For example, the assessor will flag as questionable a configuration of three objects A, B, and C in which A uses B and both A and B use C (Figure 5-8). If C happens to be a library unit (i.e., a highly reusable package) then this usage is probably acceptable. In any case---library unit or not---the designer should make a conscious decision:

```
Diagram: diagram_name
Object: "Object_A" on vm level 3 uses
object: "Object_C" on vm level 1.
Severity level: Questionable.
```

5.4 Invoking the Assessor

The invoke the assessor select the PCT icon in the second panel of the IDEtool screen. Enter the name of the top-level Object Diagram of the design you want to assess. Select *Assess Object-Oriented Design* in the third panel, and then click *Execute* in the top panel. The output from the Assessor will appear in the bottom panel.

Before invoking the Assessor you should first make sure your Object Diagrams are consistent. The Assessor behaves unpredictably with an inconsistent set of Object Diagrams. To check the consistency, proceed as above but instead of *Assess* select *Check Object-Oriented Design* in the third panel. The output from the checker will appear in the bottom panel.

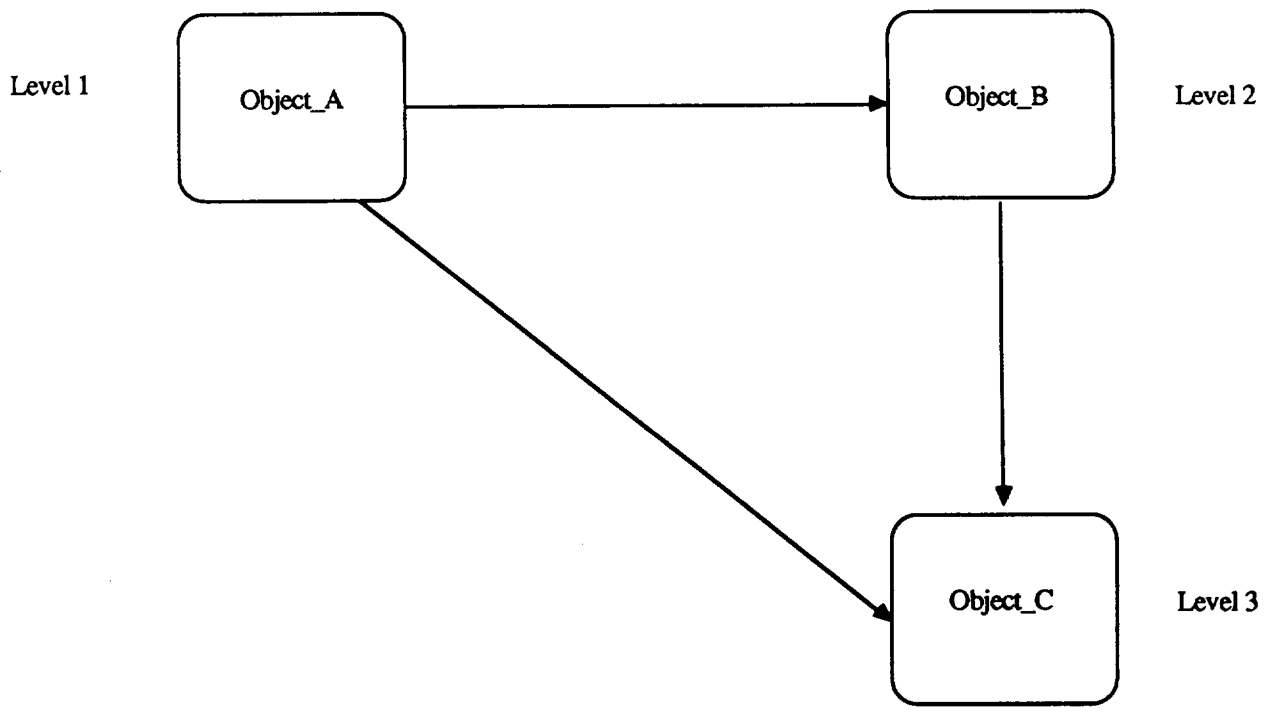

Figure 5-8: The Designer May Want to Reconsider When Virtual Machine Levels are Crossed

6 THE REUSE DATABASE

In this section we discuss the structure and contents of the Reuse Database. Understanding this structure will assist you in performing the following activities:

o Analyzing an application domain to identify typical systems, components, and variations---the framework for reuse

o Developing reusable products, or refining products to make them reusable

o Classifying and storing reusable products

The database contains *descriptions* of reusable products. These products may be entire systems or subsystems, or individual components. Reuse is not limited to the source code or executable modules. Documentation, specifications, test plans, test results, etc. are all *aspects* of the software product and can all be reused. By ''reuse'' we do not necessarily mean integrating a product ''as is'' into your system. Reuse may simply consist of modeling your work on a previously existing product.

The Reuse Database consists of two semantic networks:

o *obnet:* the object network

o *keynet:* the keyword network

A *semantic network* consists of *nodes* and *arcs* (or, if you prefer, *entities* and *relationships*). Different types of arcs represent different kinds of relationships between nodes. Two nodes may be connected by several arcs---each of a different type---indicating that the two nodes are related to each other in several different ways.

Obnet nodes represent reusable products. *Keynet nodes* are just keywords. The keywords are used to classify and describe the reusable products. The two networks are linked by means of *associations* between keywords and obnet nodes. The keywords and associations together provide a *faceted classification* scheme, in which there are several ''dimensions'' or ''facets'' by which a reusable product may be classified.

6.1 Types of Reusable Products

Reusable products (i.e., obnet nodes) may be of different *types*. The Reuse Database currently supports the following product types:

o System

o Object

o Task

o Method

o Data Structure

o Exception

Reuse of an entire *system* is a reasonable possibility during the earlier phases of development, e.g., during requirements analysis. Typically this form of reuse involves modeling a new system on an existing system. We consider this a form of reuse whether or not you choose to incorporate some components of the old system directly into the new one.

Objects are the basic element of Object-Oriented Specification and Object-Oriented Design. We sometimes refer to objects as ''entities.'' (We tend to use ''entity'' during the specification phase and ''object'' during the design phase.) Do not confuse ''objects'' with obnet nodes! An object is represented in Ada as a package; in the database it is represented as *one type* of obnet node, but there are other types of obnet node as well (i.e., system, task, method, etc.)

The term *method* is taken from Smalltalk. For our purposes---since we are primarily concerned with Ada software---a method is either a *procedure*, a *function*, or a *task entry*. Similarly, *tasks*, *data structures*, and *exceptions* are to be understood in the sense of Ada. The reason we include exceptions is so that you can fully represent the information contained in an Object Diagram by means of nodes and arcs in the obnet.

6.2 Relationships Between Reusable Products

The following arc types are available in the obnet:

o System *contains* Object

o Object *contains* Object

o Object *contains* Task

o Object *provides* Method

o Object *provides* Data Structure

o Object *uses* Method

o Object *uses* Data Structure

o Object *handles* Exception

o Task *provides* Method

o Task *uses* Method

o Task *uses* Data Structure

o Task *handles* exception

o Method *raises* Exception

o Method *handles* Exception

o Method *uses* Method

o Method *uses* Data Structure

These arc types are all based on the Object Diagram methodology [5]. Figure 6-1 presents the same information in the form of an Entity-Relationship Diagram.

6.3 Attributes of Reusable Products

Each reusable product has a set of *attributes*, which depends on the type of product. The following attributes are currently defined in the Reuse Database:

o System attributes:

- Application category
- Target system
- Project
- Version
- Programming language
- Reference

o Object attributes:

- Application category
- Target system
- Project
- Version
- Programming language
- Size (bounded, undounded, or limited)
- Access (protected, sequential, guarded, controlled, multiple, or multi-guarded)
- Concurrency (operation concurrence or object concurrence)
- Type
- Iterator (yes or no)
- Managed (yes or no)
- Priority (yes or no)

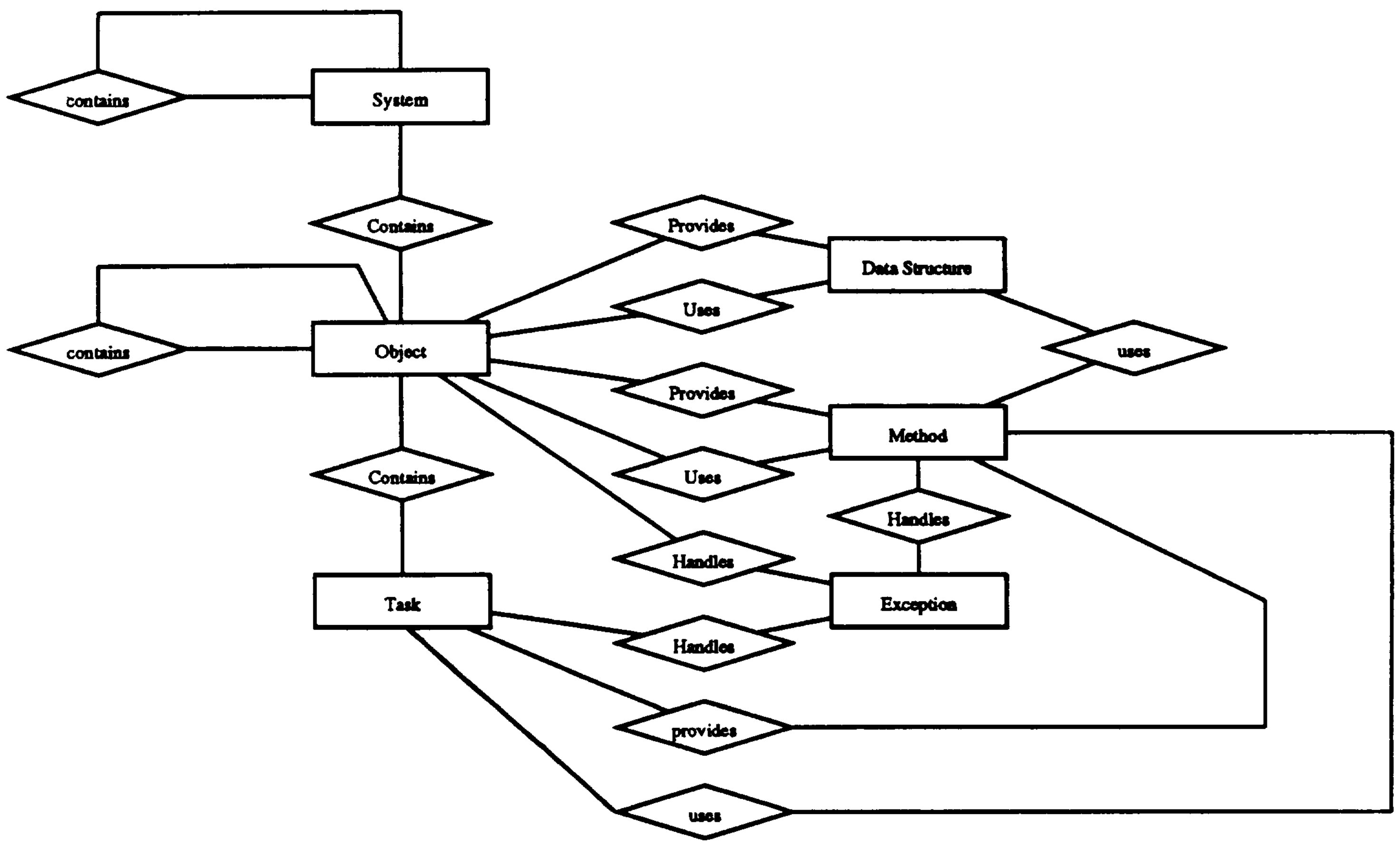

Figure 6-1: Product and Relationship Types in the Reuse Database

- Balking (yes or no)
- Reference

o Method attributes:

- Application category
- Target system
- Project
- Version
- Programming language
- Concurrency (operation concurrence or object concurrence)
- Reference

o Data Structure attributes:

- Application category
- Target system
- Project
- Version
- Programming language
- Reference

The attributes *Size, Access, Concurrency, Iterator, Managed, Priority,* and *Balking* are taken from the taxonomy of Ada components developed by Grady Booch and refined by Edward V. Berard.

The *Reference* attribute is intended to serve as a pointer to additional documentation about the product. For example, it may contain the name of a system and directory containing the product's source code and related documents.

You can display the attributes of a product in the Reuse Database as follows. First select *Reuse Database* in the second panel of the IDE main screen by positioning the mouse cursor in this selection and then clicking the left-hand mouse button. Then select *Show Detail* in the third panel (again with the left-hand mouse button). Notice that you are now prompted for two arguments in the fourth panel. One argument is the *name* of the product whose attributes you wish to see. Type this name in where the caret indicates, then hit RETURN. The other other argument is the *type* of the product, and here you simply select the appropriate type by clicking the mouse on your selection. Then move the mouse cursor up to the top panel and click *Execute*. The attributes of the product will be displayed in the bottom panel.

6.4 Navigating the Product and Keyword Networks

Navigation is a means of browsing the contents of the Reuse Database. We call it ''navigation'' because you can move from product to product, or from keyword to keyword, by traversing any of the relationships (''arcs'') that are represented in the database. By navigating the database you can get a sense of what reusable products are available, and of the categories into which they are classified (by means of keywords).

Another reason for navigating is as a follow-up to an automated search for reusable products. The automated search process is described in Section 7. The automated search may turn up products that are *almost*, but not exactly, what you are looking for. By examining other products that are ''similar'' or closely related to the located products, you may find something that better meets your requirements. For example, you may not have used the best keywords to describe the product you are looking for; or you may interpret the keywords differently from the person who created the database. By navigating through related keywords you may find a more accurate description of the product you want.

The navigation commands work as follows. You first select a product or keyword to start navigating from, and a type of relationship (an ''arc type'') to navigate over. You can use the commands *List Products Arctypes* and *List Keyword Arctypes* (in the third panel of the IDEtool screen) to see what arc types are available to be navigated over.

When you have selected a starting product or keyword, and an arc type, and you then execute the appropriate *Navigate* command (in the third panel of the IDEtool screen) the environment will display the *neighborhood* of the selected item (product or keyword) with respect to this arc type. The display is in graphical form. The neighborhood consists of all ''parents'' and all ''children'' of the selected item. A parent is an item that has an arc (i.e., a relationship) *from* itself *to* the selected item. A child is an item that has an arc *from* the selected item *to* itself. The neighborhood of an item, in short, consists of the item's closest relatives.

We use PICture, IDE's general-purpose diagram editing tool, to display the neighborhoods. The first time you select *Navigate Products* or *Navigate Keywords* within an IDEtool session, PICture will automatically be invoked to display the neighborhood. When you are finished navigating, if you plan to stay in the IDEtool session, *do not quit* PICture. Instead, just *close the window* in which it is running: that way, PICture will continue running but it will not clutter up your screen. The next time you select *Navigate Products* or *Navigate Keywords*, the following message will appear in the bottom panel of the IDEtool window:

Activate PICture to view neighborhood.

To view the neighborhood, simply open the window in which PICture is running; then click the left-hand mouse button on the *Load* command in the top panel of PICture. This will cause the new neighborhood display to be loaded.

The advantage of leaving PICture running in the background, as just described, is that you do not have to wait for PICture to be re-executed every time you want to navigate. Executing PICture involves loading the program into memory, which takes up considerable time.

Navigation consists of moving from neighborhood to neighborhood in order to visualize the contents of the database. There are two ways you can move from neighborhood to neighborhood:

o Stepwise

o Jumping

In stepwise navigation, you select any item in the current neighborhood, and display *its* neighborhood. In other words, a parent or child of the central item in the current neighborhood becomes the central item of the new neighborhood. The previous central item will be one of the parents or children in the new neighborhood. *To move stepwise to a new neighborhood, do the following:*

1. Pop up the editor menu by depressing the right-hand mouse button

2. Move the mouse downward until the *Options* command is highlighted

3. Slide the mouse to the right until the *Options* menu pops up

4. Slide the mouse down until the option *Neighborhood* is highlighted

5. Release the right-hand mouse button

You will be prompted with the following message:

Please select node for new neighborhood.

in the upper left-hand corner of the PICture window. In response to this prompt, click the left-hand mouse button on the item that you want to make into the new central item. You will see the message

Executing nbrhd

in the upper left-hand corner. When this message gets replaced by the message

nbrhd completed

click the left-hand mouse button on the *Load* command in the top PICture panel. The new neighborhood will then be displayed. ***Remember, you must click* Load *to display the new neighborhood.*** This is a consequence of our using PICture as the mechanism for displaying the neighborhoods (it is not really the intended use of PICture).

To jump to an entirely different neighborhood, which may have no overlap with the current neighborhood, simply replace the name of any item in the current neighborhood with the name of the new item, to whose neighborhood you wish to jump. Do this by placing the mouse cursor inside the item whose name you are replacing (it can be any circle-node in the diagram), using the DELETE key to erase the current name, and then entering the new name. Hit the ESC key to terminate the new name. When you have finished entering the new name, click the left-hand mouse button on *Store* in the top PICture panel. ***Clicking* Store *is essential for the new name to be "recorded."*** Then execute the *Neighborhood* option, as described above, selecting the node that you have just changed to be the center of the new neighborhood.

At any point during the navigation of reusable products you can **display the attributes** of the products in the current neighborhood. The attributes are recorded in the text annotations of the nodes in the diagram. To view the annotation, depress the right-hand mouse button to pop up the PICture menu, and slide the mouse down until *Text* is highlighted. Then slide the mouse to the left until a second menu pops up. Then slide the mouse down until *Show Text* is highlighted. Now let up the right-hand mouse button. A message in the upper left-hand corner of the PICture window will prompt you to select the node whose text you wish to see. You should click the left-hand mouse button on the node representing the reusable product whose attributes you want to view. The attributes will then pop up next to the node. To hide the attributes again, simply click the left-hand mouse button again on the same node.

The displays you see during navigation are of neighborhoods with respect to a specific arc type. The arc type is identified in the middle of the display, next to the central node. **At any time during navigation, you can change the arc type to another arc type.** To do this, simply edit the arc type label, i.e., move the mouse cursor to the label until a gray arrow appears pointing to the label; use the DELETE key to erase the characters of the old arc type label, and then enter the new arc type. Use the ESC key to terminate the new arc type label. Then click the left-hand mouse button on *Store* in the top PICture panel. ***You must click* Store *to have the new arc type recorded. Otherwise it will not take effect.***

After you hit *Store* the new arc type is recorded, but the display remains the same until you request a new neighborhood. Subsequent neighborhoods will display parents and children with respect to the new arc type. If you want to keep the central node the same, but display its neighborhood with respect to the new arc type, just select the *Neighborhood* option as described above; then click on the central node as the item whose neighborhood you want to see.

7 LOCATING REUSABLE PRODUCTS

This section describes how to search for reusable products. In our scenario you do not access the Reuse Database directly (of course you can if you want, in which case you should consult the SEMANTX User Guide). Instead, you initiate automated searches from a diagram edit session. You can initiate a search from any of the following diagrams:

o Entity-Relationship Diagrams (ERDs)

o Entity Data Flow Diagrams (EDFDs)

o Object Diagrams (ODs)

The steps of the search process are 1) annotate the diagram nodes with keywords, 2) select a search command, 3) select an item to search for, 4) examine the results of the search and make corresponding reuse decisions, and 5) return to the diagram edit session.

7.1 Annotating Nodes with Keywords

The nodes within the diagrams can be annotated with textual descriptors. To create or edit the descriptor of a node, press down the right-hand mouse button with the mouse cursor positioned outside of all nodes. This causes the command menu to pop up. While still holding down the right-hand button, mouse the mouse cursor down to the command labeled *Text*. Now (still holding down the righ-hand button) move the mouse directly to the right until another menu appears next to (and a little below) the menu containing *Text*. Move the mouse down to the line that says *Edit text* in this second menu. Now let up the right-hand button, move the mouse into the node whose descriptor you wish to create or edit, and click the *left*-hand button. Notice that a small colon (:) appears just outside the node you have selected. This colon is the prompt to enter text into the descriptor. Move the mouse to the prompt, and enter the text with which you want to annotate the node.

You can also annotate the diagram as a whole by editing the diagram's text descriptor field. This is done in the same way that you edit a node's text descriptor, with the following exception: After you 1) select *text* from the edit menu, 2) slide the mouse to the right, and 3) select *Edit text* from the secondary menu, you should 4) click the left-hand mouse button with the cursor *outside* of all nodes (instead of clicking it on a node). This will initiate an edit of the diagram's text descriptor.

You can bring up a full-screen editor to facilitate moving around the text and making changes. To do this, type in CTRL-e. You will observe the message *Executing Pipetool* in the upper right-hand corner of the diagram window. In a few seconds a new window will pop up with the full-screen editor (on UNIX systems this is typically the *vi* editor, although you can change this). When you have finished editing the descriptor, save it using the ordinary editor command and then exit the editor (for example, in *vi* you would type in *:wq* followed by RETURN). Do *not* destroy the window in which the editor appeared (the IDE editors become flaky if you do this); instead, just bring the diagram window to the front again by clicking the left-hand mouse button on its border. If you like, you can *close* the window that contained the editor.

The automated search for reusable products is based on keywords. You place these keywords in the descriptor of a diagram node. The keywords are the means by which you describe (or specify requirements for) the reusable product for which you are looking.

The form in which you specify keywords is as follows:

```
Keywords:
- item1 => keyword1
- item2 => keyword2
- item3 => keyword3
etc.
```

where spaces and tabs may be placed anywhere between words. We call this a *keywords clause*, and one or more such clauses can appear anywhere in the node descriptor. Each *item* should be the name of either the diagram node itself or of an item (i.e., a procedure, function, task entry, or data structure) that is provided by (or contained in) the node. An item may have several keywords, in which case you simply enter multiple lines with the same item name. You may leave out "*item* =>" in which case

```
- keyword
```

is assumed to be a keyword describing the node itself. In fact, the *item* is only relevant to Entity-Relationship and Object Diagrams, not to Entity Data Flow Diagrams. In an ERD or an OD, the syntax for provided (and contained) items is as follows:

```
Provides:
- method1
- method2
etc.
```

and

```
Contains:
- data1
- data2
etc.
```

Like the keywords clauses, any number of "provides" and "contains" clauses can appear anywhere in a node descriptor in an Entity-Relationship or Object Diagram.

NOTE: Currently there is an annoying inconsistency between the diagram-based tools and the schema of the Reuse Database. The diagram-based tools, including the search commands, distinguish between "provides" (which is used for procedures and task entries) and "contains" (which is used for data, i.e., states). This notation is now obsolete, and we *should* use "provides" for all such items, i.e., procedures, task entries, and data. That is how the Reuse Database schema is defined. ("Contains" in the Reuse Database refers to sub-systems or sub-objects, not to data elements.) Until the diagram-based tools are updated, however, you should use "contains" for data items in the diagrams, and "provides" for procedures and task entries.

7.2 Selecting a Search Command

You can now point to the node and, through a simple mouse click, tell the environment to "find me one of these." Do this as follows. Press down the right-hand mouse button so that the diagram editor menu appears. While keeping the right-hand button down, move the move down the menu until the item called **Options** is highlighted. Now move the mouse to the right. A second menu---the menu of options---will pop up next to, and a little below, the first menu. Move the mouse down this second menu until one of the Search4 ("Search for") options is highlighted. The specific Search4 option you should choose depends on the type of product you are looking for. The following Search4 options are available:

From Entity Relationship Diagrams:

o Search4system

o Search4entity

o Search4method (a "method" is a procedure or function provided by an entity)

From Entity Data Flow Diagrams:

o Search4system

o Search4entity

o Search4function

From Object Diagrams:

o Search4system

o Search4class (i.e., a class of objects)

o Search4object

o Search4method

o Search4data

7.3 Selecting an Item to Search For

When you have highlighted the appropriate Search4 option, release the right-hand mouse button. In all cases except Search4system, the following prompt will appear in the upper left-hand corner of the edit window:

Please select node for Search4*type*

where *type* depends on the specific Search4 option you selected. In response to this prompt, select the node that represents the product you are looking for.

If you selected Search4system you will *not* be prompted to select a diagram node. Instead, the search will be based on the keyword annotation of the diagram as a whole.

If you selected Search4method or Search4data, specifying the diagram node is not sufficient to tell the environment which method or data structure to search for. An entity in an ERD can provide several methods; an object in an OD can provide several methods and contain several data structures. In these cases, a new window will pop up with a display of the provided methods (or the contained data structures), and you will be prompted to enter the name of the method or data structure to search for.

7.4 Examining the Results and Making Reuse Decisions

After you have initiated the search, you must transfer from the *diagram window* to the *IDE main window* (click the left-hand mouse button on the border of the IDE main window). Now just wait for the following message to appear in the bottom panel of the IDE main screen:

Okay to invoke post-search options.

This message means that the search is completed. You are then ready to examine the results of the search and to follow up the search based on the suitability of the products that were located. Move the mouse to the second panel and click the left-hand button on *Reuse Database*. Then move to the third panel and click on *Post-Search Options*. You now have the following options:

o List Located Products

o List Keywords Used

o Reuse Product

o Add Keyword

o End Search

List Located Products shows you the results of the search. A list of all products having one or more of the keywords you specified is displayed in the bottom panel. To get more information about any of these products, you can display its attributes---how to do this was described at the end of Section 6.3. Displaying the attributes of a product will take you out of *Post-Search Options*, but you can return to *Post-Search Options* at any time (in fact, you *must* return to *Post-Search Options* in order to end the search and return to your diagram edit session).

List Keywords Used displays a list of the keywords that were used in the search. This may be helpful especially if the diagram node annotation contained keywords for many different items: only some of these keywords will apply to the item for you are looking for.

When you select *Reuse Product* under *Post-Search Options*, you will be prompted for the name of the product that you want to reuse. This can be one of the products located through the search, but it does not have to be. *There is no automated verification that the product you specify actually resides in the Reuse Database.* The effect of this option is to cause a ''Reuses'' clause to be placed in the descriptor of the node from which the search was initiated (or, in the case of Search4system, in the diagram's descriptor). In other words, this is a way of automatically recording the decision to reuse a certain product. You may reuse as many products as you wish. The syntax of the Reuses clause is similar to that of the Keywords clause, i.e.,

```
Reuses:
- item1 => product1
- item2 => product2
etc.
```

The *item* in this case will be the same item you selected when initiating the search.

Add Keyword operates in a manner similar to *Reuse Product* except that, in this case, a new Keywords clause will be added to the node descriptor (or diagram descriptor).

7.5 Returning to the Diagram Edit Session

To end the search process you must select the *End Search* option. This signals the diagram edit session that the search (and post_search) is over and that editing can resume. If you added any keywords during Post-Search, you may want to initiate another search for the same item, using the additional keywords as a more finely tuned description of the desired product.

REFERENCES

[1] Bailin, S.C. and Moore, J.M. *An Operational Concept of Software Reuse.* Computer Technology Associates, Inc., Document Number 330-3015068-87-02a. June, 1987.

[2] Jackson, M. *System Development.* Englewood Cliffs: Prentice Hall, 1983.

[3] Booch, G. *Software Engineering with Ada,* second edition. Addison-Wesley, 1987.

[4] Stark, M. and Seidewitz, E. ''Towards a General Object-Oriented Ada Lifecycle,'' *Proceedings of the Joint Fourth Washington Area Ada Symposium/Fifth National Conference on Ada Technology.* March 1987.

[5] Seidewitz, E. and Stark, M. *General Object-Oriented Software Development.* NASA GSFC Software Engineering Laboratory SEL-86-002. August 1986.