Tool Support for Object-Oriented Development: Specification and Design

Steven Wartik

Computer Science Report No. RM–87–02
September 9, 1987

Abstract

Object-oriented development is emerging as an important software design technique. This document specifies a set of tools intended to assist in the architectural design phase of a project being developed via object-oriented design. The tools help assure that a project is structured well, in that it conforms to the principles of object-oriented development: the focus in the decomposition is on objects, not procedural abstractions. Furthermore, the tools help in developing reusable software packages.

To help separate fundamental concepts from implementation details, the tools are first discussed in terms of an idealized world, independent of both operating systems and implementation languages. Later, they are mapped onto a Unix-like file system, with the assumption that the software is being developed using the Ada programming language. An implementation strategy for doing so is discussed.

This work was supported in part by NSF grant DCR–8602674.

Tool Support for Object-Oriented Development: Specification and Design

September 9, 1987

1. INTRODUCTION

This document specifies a set of tools that support the architectural design of software in an object-oriented manner. The purpose of the tools is to provide a particular view of software that reflects the principles underlying object-oriented development [2]. They are intended to be useful additions to a software development environment, supplementing existing design aids by helping to structure the construction of systems. The goal of this document is to provide a reasonably complete description of the tools and the entities they manipulate, along with information on their implementation.

For much of this document, we assume the environment hosting the tools contains an integrated database, *àla* PCTE [4]. Environments with integrated databases are far more powerful than ones with traditional hierarchical file systems. In the former, sophisticated structures and data relationships can be represented and described simply. In the latter, anything beyond a hierarchy requires an implementation on top of the existing capabilities. Since the tools create and manipulate structures that are often non-hierarchical, we can greatly simplify the discussion by assuming the database-oriented view. A previous paper described conventional hierarchical file structures to be used during object-oriented development in a more traditional environment [8]. A section at the end of this paper discusses the implications of moving these tools onto such a system.

As regards the database model, we choose to allow features that currently are supported only in research operating systems, not commercial ones. Few commercial database management systems allow strings of arbitrary length, for instance, yet this feature is clearly advantageous in representing such entities as documentation or program source. (Again, this assumption is not strictly necessary but simplifies the discussion.) We choose an entity-attribute-relationship model of the data [3], since we often want to refer to entities and their attributes; moreover, the terminology of the EAR model is more powerful than those of the three great models.

A primary consequence of using the database model is that the notion of a "file" is not relevant. One does not issue a command such as "edit the file containing the specification part of package P," but rather, "edit the specification part of package P." Users refer directly to the entities, not to the files containing them. For this reason, the tool descriptions refer to entities and not to files. Indeed, even the database concept is irrelevant. Conceptually, the entities exist independent of any secondary storage medium, if one recognizes their permanence.

No requirements are to be placed on how the tools are invoked. The following are candidates (not claimed to be a complete list of possibilities):

— Command-level tools invoked on request to build software structures.

— Operations performed automatically on text as it is entered by a user.

— Subroutine library-level tools that could be utilized by other tools, as part of the process of automating software development.

— Graphics tools mapping between a user-constructed display (that is, the user links together modules on a two-dimensional display) and a database.

A determining factor should be how easily they integrate into the host environment. This might imply that two (or all) of the above styles should be available.

It is important to understand the intent of these tools. They are *not* an attempt to simulate the capabilities of a structure editor, with features to prevent syntactic and semantic mistakes when programming. Instead, they exist to aid in the architectural design of software, while integrating to some degree an architectural model (namely, one provided via an EAR-oriented information management system) with the syntax of a programming language. For this reason, the tools manipulate only a small set of language entities. Moreover, due to language nuances, they sometimes (perhaps confusingly) manipulate those entities only in certain contexts. For instance, a procedure that is a programming unit is treated differently from a procedure nested within another procedure. The latter is usually unimportant in architectural design derived through object-oriented techniques. It represents an implementation detail and not part of the overall architecture.

The presentation is usually independent of a particular programming language, since object-oriented development is an architectural design technique that is not tied to implementation languages (although it is certainly true that some languages are more suited to expressing its concepts than others). However, discussions of modularity will sometimes require terminology that is not standard across languages. At these times we will use the terminology of the Ada† programming language [1].

Much of the document is aimed at organizational issues: configuration management of the system, with consequent automatic notification of developers when a change is made. The OVERSEE configuration management system [7], and its organization capabilities, are tacitly assumed to be available for the use of the tools.

This document is organized as follows. In the next section, we define the entities that are pertinent to our system, and the relationships between them. We then present the functional specifications of the tool set. Finally, we discuss possible implementation techniques.

2. ENTITIES, ATTRIBUTES AND RELATIONSHIPS

There are many ways to view the entities these tools manipulate. One extreme stems from "pure" object-oriented development. In this view, the entities are objects, procedural abstractions and task types. They correspond to the entities that compose a software structure derived through object-oriented design. Such entities are ultimately mapped to the terminology of a programming language: packages, procedures and tasks in Ada, for instance. Thus one would create and manipulate objects, which the tools would map to packages. At the other extreme, entities may be viewed in terms of their realization in a programming language. In this view (in Ada), one would create packages, realizing that they correspond to objects.

Both approaches have merits and detractions. The first is better in that designers work independent of a particular language as they build the software architecture, expressing themselves purely in object-oriented design notation. The second shows the mapping between the implementation language and the entity model more clearly; since this mapping is used during detailed design, implementation and modification, the information it contains is of no small consequence. However, the second approach has certain problems. In Ada, for instance, an object is always implemented as a package but a package does not always define an object. Working solely in terms of language constructs would therefore not represent the abstract concepts of object-oriented development properly.

It is important to use techniques that eliminate errors as early as possible. For this reason, we adopt something closer to the first approach than the second, but a little less strict. To recapture the advantages of the second approach, we introduce tools that help developers map between architectural design and implementation. The next section will discuss them in more detail.

2.1 Entities

Broadly speaking, the entity classes are libraries, directories, objects, procedures and task types. A library contains a set of zero or more objects, procedures and task types. Some entities within the library have global visibility, and may be freely used (via a 'with' clause in Ada) by other entities in or outside the library. Other entities are hidden through the software structure.

Entity classes are further divided to identify an entity's role. Objects may be "unit objects," "hidden objects" or "interface objects." All of these objects are stored in libraries; a unit object is visible to other entities in the library, and to entities outside the library. In other words, it is a module that is independent of others, and hence may be re-used by other entities at will (via a 'with' clause in Ada). A hidden object has less visibility within its library, and no visibility outside its library. It is derived from a unit entity or another hidden entity, being used in

† Ada is a registered trademark of the United States Government, Ada Joint Program Office.

the implementation of that entity. Therefore it is at a lower level of abstraction in the software structure, and is visible only at that level. Often, there is no distinction between a hidden object and a unit object, except that the former are derived from that latter during the design. If so then the hidden objects may be made into unit objects; this is an operation called *reuse*, since it makes the object globally visible and hence reusable. The abstract levels of the software structure established by the derivation still exist, but they become virtual rather than physical (see the example in the section on implementation). Note that an ideal project would contain only unit objects, implying that all objects in the project are reusable.

Finally, an interface object is an object that is visible through the specification part of another object. an interface object therefore exists with respect to some other object.

Similarly, procedures are classified according to their purpose as "unit procedures," "hidden procedures" or "interface procedures." Unit and hidden procedures are analogous to unit and hidden objects: a unit procedure has global visibility in its library, whereas a hidden procedure's visibility is limited to the level in the software structure where it exists. Interface procedures are visible through an object; that is, they are its interface routines. Like interface objects, interface procedures always exist with respect to some object. Unit and hidden procedures, on the other hand, are procedural abstractions in the system.

Objects, procedures and task types all have both specification and implementation entities. These serve the same purposes as they do in Ada. Moreover, in object-oriented design, entities are first specified and then implemented, and the entities at lower levels of abstraction are derived from the implementation, not the specification. The separation therefore more accurately represents the software structure than an approach that treats the specification and implementation as attributes of the same entity.

The "directory" entity helps organize the software for viewing and editing. It is not strictly necessary—relationships can define the same structure as can a directory. However, it helps in establishing a proper context for entities. A directory for a procedure entity contains both the specification and implementation. A directory for an object contains not only the object's specification and implementation, but its interface (interface procedures) as well. It defines, then, a set of objects that are currently being inspected. Unlike their counterparts in a file system, directories are not subordinate to other directories except as defined through the relationships between entities. (However, a file system implementation could map these directories onto its own in an obvious way.) Directories also have the property that one per library is designated as "current" for each developer using the library. The current directory defines the set of entities being viewed.

2.2 Entity Attributes

All entities have a name attribute that helps identify the entity. (The unique identification is as in Ada.) Their other attributes vary depending on kind. In general, one may assume the attributes that exist store enough textual information to construct the appropriate entity. Procedure specification entities, for example, have an interface attribute (the parameter list and, for a function, data type returned) that is a text string. (Certain tools may want to view this as a set of identifier and data type entities, but, for the tools in this document, a text string is simpler and equally effective.) Procedure implementation entities have a code attribute, which is a text string containing the code that constitutes the implementation. Note that this could be pseudo-code or Ada, depending on the stage of the design; we make no requirements on its form.

Entities have other attributes that depend on the entity class. Each entity has a specification attribute. The text attribute, which is the textual representation of the entity in the given implementation language, is an important one. There is no requirement that an entity be stored in a text file. Instead, it could be stored as a set of attributes and relationships. For instance, an object entity has the attributes interface and implementation. The interface attribute consists of a set of operations and types. This information can be derived from scanning a text file, but equally well the text file could be derived from scanning it. However, the limitations of available compilers mandate the need to be able to produce a textual representation of an entity. In this document, therefore, we shall feel free to refer to entities as both text and as a set of attributes and relationships, depending on the needs of a given tool.

2.3 Relationships

Figures 1–3 depict the relationships between the entities in the standard E-R diagram notation, focusing on derivation and reuse.† Each diagram shows, for one entity type (including both its specification and implementation)

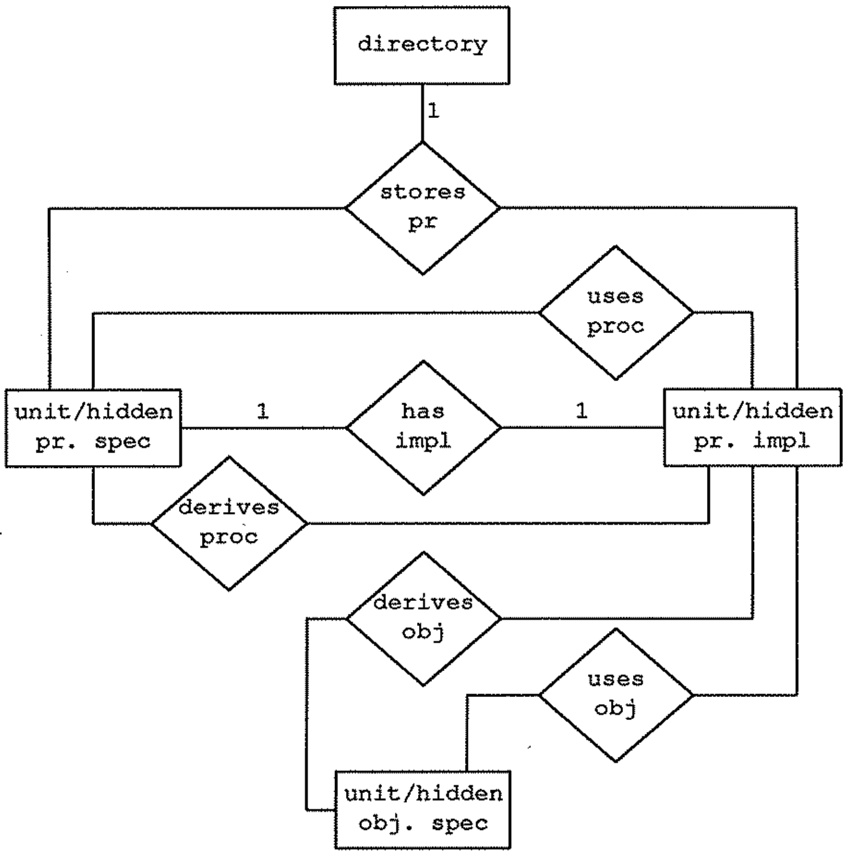

Figure 1: Entity-Relationship Diagram of Procedure Structure

how the entity is related to other enties. The focus is on what is derived from the entity, and what entities the entity "re-uses". Unit and hidden entities are grouped together, because they can and often do serve the same purpose. Except as noted, the relationships are many-to-many. They are as follows:

`has impl`	A specification has exactly one corresponding implementation part. (Actually, an Ada package need not have a body if it contains only constants; such a package is not an object, however.)
`uses ...`	Certain entities use others in their implementation. For instance, a procedural abstraction may require other procedural abstractions or objects provided through a package. Specifications may also 'use' entities: a generic list package requires an object as the element type of the list.
`derives ...`	In object-oriented development, an entity is sometimes implemented by defining other entities that are to be used in its implementation. The result is a new, underlying architecture. The original entity *derives* the entities in this architecture. Note that the implementation, not the specification, derives entities.
`shows`	An object's specification shows certain other entities (procedures and objects) to the outside world.

† For brevity, tasks are not included. Their relationships, however, are analogous to those shown: Procedure and object implementations can reuse task specifications, task specifications can use objects, and task implementations can use and derive other specification entities.

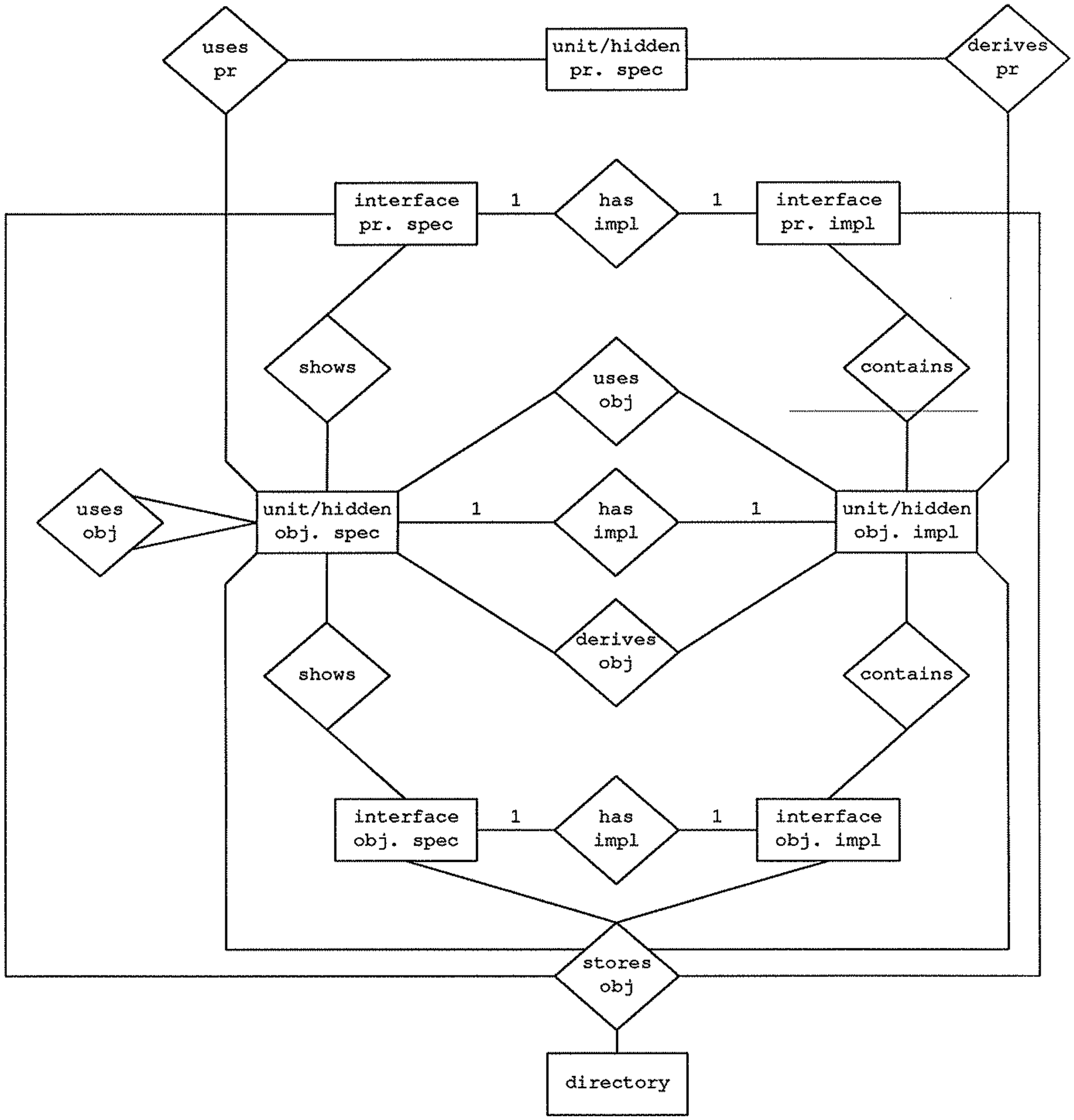

Figure 2: Entity-Relationship Diagram of Object Structure

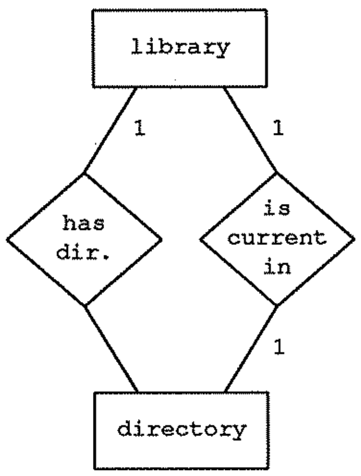

Figure 3: Entity-Relationship Diagram of Library Structure

`contains`	If the specification of object *O* shows a specification for entity *E*, then *O*'s implementation contains *E*'s implementation.
`stores ...`	A directory object contains a set of entities of a given type (object, procedural abstraction or task).
`has dir`	A library has a given set of one or more directory entities. A directory exists in exactly one library.
`is current in`	Exactly one directory per developer is current in a library (the picture does not show developers).

The next section explains these relationships in more detail, in terms of the proposed tools.

3. FUNCTIONAL SPECIFICATIONS

There are three broad categories of tools:

1. Those that create or modify an entity.
2. Those that are used for navigating through a system.
3. Those that query attributes of a system.

Abstractly, these tools take program units as input, and produce as output those program units mapped onto some database structure that reflects the software structure. This, however, is a purely functional view and is not appropriate as a means to develop software. Consider, for example, the following command for creating a package:

```
create package P
     with respect to E,
     with interface I and implementation J,
     requiring R
```

where:

- E is some existing entity;
- I is a set of procedure specifications;
- J is a set of procedure bodies;
- R is a set of entities that are required to compile the package (procedures, packages, etc.)

While this command is a logical view of what is needed to create a package, it cannot be given as a whole. People develop software piecemeal, but the above command requires knowing all entities needed. This is contrary to top-down development principles. Some mechanism to include stubs is needed. Therefore, the tools defined in this section work only on portions of entities.

Table 1 summarizes the proposed tools. The following sections will give detailed descriptions of each.

3.1 Entity Creation

This command category is used to create any entity that corresponds to some module in the software structure. Commands in this category do not create the entire entity. Instead, they create a framework—directory entities and fragments of programming language text—that may be expanded and edited through the other tools. Each module has an interface and an implementation, and only the barest essentials of each are initially created.

In Ada, modules are the program units: packages, procedures and tasks. The creation tools are intended to create these entities, which are most important to the architectural design. Note that the aim is to create independent, reusable program units, instead of modules physically nested within other modules. The tools discourage (or rather, do not provide for) such entities, since they are seldom reusable and tend to depend on the entities in which they are nested. Physical nesting as a means to show entity relationships is useful in a language such as Pascal, whose scope rules mandate that hierarchy be implemented via nesting. In a language with modularity constructs, nesting to show hierarchy is not only unnecessary, it is poor coding practice [5]. The tools therefore assist only in creation of independent modules, with relationships that define software structure. Hence, creation preserves the information about the software architecture, and does so in a way that makes all modules either unit or hidden entities. This should heighten reuse possibilities.

Ada compilation semantics require the recompilation of all units that depend on an entity when the entity's body is recompiled. Changing an entity's body should not require recompiling entities that depend on its specification, however. For this reason, the tools maintain the specification and implementation in a manner that allows a compiler to recompile the body without recompiling the specification. At times, however, the specification and implementation must be treated together. The tools therefore merge the two when appropriate. Specific instances are given below.

3.1.1 create object

The general form of this operator is:

```
create <type> object name
```

where <type> is one of 'unit', 'interface', or 'hidden'. The effect is to establish a context for building an object: a new directory entity is created, as is the specification entity for the object. The directory entity is named for the object. The specification entity, when created, contains no information; it is only a stub. In Ada the specification would be as follows:

```
package name is
end name;
```

Note that creation involves creation only of the specification portion of an object (and all other entities), as is standard in architectural design. Bodies are created through the 'implement' command, described below. The entities that comprise the specification are also filled in through other commands described below.

The three types of objects each identify a different kind of visibility within a library. A 'unit' object is a program unit, meaning that it is supposed to be visible at the top level of development. A 'interface' object is a subpackage of an existing object, with its specification given in the object's specification. A 'hidden' object is used only by the implementation of an object. Note that a hidden object is also a program unit; it differs from a unit object in that it is placed at a subordinate level in the software structure.

Figure 4 shows how Ada entities correspond to the object types. If we assume lib_p is a unit entity, then it must be directly visible in the library. It contains the package specs_p—such a construct is often useful to group related entities of a package—which is visible to any unit that references lib_p. The implementation of lib_p uses the package needed_p. If needed_p already exists in the library, then since it is visible to lib_p it must be a unit object. If it is defined during the design of lib_p, then it is not immediately visible and hence is a hidden entity.

Interface and hidden objects exist with respect to some other object. Therefore they are created with respect to a given object, determined by the current entity. Hence a current entity must be established to use these variants of the operation. A hidden object may also be created with respect to a procedure or task. An interface object or unit object cannot. This is logical because interface objects are supposed to be visible through some other object, and procedures and tasks do not make objects visible. In Ada it follows by definition, since the language enforces the

	Tool Name	Brief Description
Creation and Modification Tools	`create` *`vis_spec`* `object`	Create an object entity.
	`create` *`vis_spec`* `procedure`	Create a procedure entity.
	`create` *`vis_spec`* `task`	Create a task entity.
	`use`	Use an existing entity.
	`implement`	Create the implementation part of an entity.
	`edit`	Make a change to an entity that does not affect its interface.
	`modify`	Make a change to an entity's interface.
	`reuse`	Make a hidden entity into a unit one.
	`rename`	Rename an entity.
	`delete`	Delete an entity.
Navigation Tools	`set current entity`	Search for a given entity, changing to its context if found.
	`up`	Move up one level in the software structure.
Query Tools	`display interface`	Display an entity's interface.
	`display implementation`	Display a package's hidden components.
	`display usage`	List all objects that use a given object.
	`display structure`	Display a project's structure, showing its architecture.

(*`vis_spec`* is one of '`unit`', '`hidden`' or '`interface`')

Table 1: Summary of Tools

```
package lib_p is              -- lib_p is a unit entity.
    package specs_p is        -- specs_p is a visible entity.
        ...
    end specs_p;
    ...
end lib_p;

with needed_p;                -- needed_p could be a unit or hidden entity.
package body lib_p is
    ...
end lib_p;
```

Figure 4: Examples of Object Types

above rule by not allowing anything other than parameters and entry names to be visible.

The context in which the object is created establishes the names that may be legally included in a 'with' clause. While all units may theoretically be referenced, doing so would violate the software structure. Therefore only entities at the appropriate abstraction level may be included. To permit otherwise would violate the principle of design hiding that is fundamental to object-oriented development.

Creating an interface object modifies its enclosing object's specification. The corresponding specification entity must therefore change during the creation of the entity. This applies to interface procedures as well (see below).

3.1.2 create procedure

The general form of this operator is:

```
create <type> procedure name
```

The type is as above. For units, a new directory context is created, named for the procedure. Here 'hidden' is reserved for procedural abstractions, not for implementations of interface routines. Creating an interface procedure when the current entity refers to a package adds an interface routine to the package. Again, an interface or unit procedure cannot be created with respect to a procedure.

The effect of creating a hidden procedure is to establish a new procedure in a context that is not visible from the original context's interface. This provides proper module structure, and separates data from procedural abstraction.

Note that creating a procedural abstraction may involve surrounding it in a package. This could occur for procedures that have an exit status whose value is most logically represented as an enumeration. In Ada, data type definitions must be within a unit, so that enumeration and procedure must be "packaged" together. For example, one might create:

```
package perform_action_p is
    type exit_status is (okay, good, not_so_good);
    procedure perform_action(in_p: in integer; status: out exit_status);
end name_p;
```

to properly associate the `exit_status` type with the `perform_action` procedure.

3.1.3 create task

The general form of this operator is:

```
create <type> task name
```

The effect is to create a task type. A new directory context is created for it, named for the task. Because tasks have a specification and a body, this operation creates a specification entity for the task; like objects, the specification entity is initially empty but may be modified.

In Ada, tasks cannot be compilation units. Instead, they are most often declared within packages and procedures. In object-oriented development, where tasks are seen as potentially reusable objects, task types should be declared in packages, not in procedures (although of course task objects can be declared and activated in procedures). Therefore this operator creates something like the following for the specification:

```
package name_p is
    task type name is
    end name;
end name_p;
```

3.1.4 creation of other entities

Other entities—data types, constants, and variables—are, or should not be, global in object-oriented design. Hence they are not created by operators given here. In general they should appear in package specifications or implementations. However, future versions of this document may want to consider operations such as "add new data type to package."

3.2 Implementation

This operation has the form:

```
implement [ name ]
```

A specification of the named entity must already have been created in the current context. The operation creates a framework for the implementation of the named entity, consisting as before only of stubs. The developer may fill in these stubs to complete the entity's implementation.

For a unit or hidden procedure, the operation creates an implementation entity for the body, which the user may then complete. If the entity is an interface procedure, then the enclosing object must already have been implemented. In either case, the implementation entity created for the procedure is simply a procedure with no internal declarations and a null action:

```
procedure name ( parameters ) is
begin
end name ;
```

In Ada, an interface procedure's body exists in a package's body. Therefore a designer must implement a package (that is, run the 'implement' tool with the package's name as an argument) prior to implementing any interface procedures within it. Each implementation of an interface procedure might involve a change to the package body; a suggested means to accomplish this is described below. (that is, the 'implement' operation must have be run for the package) In Ada, one can keep the two entities distinct through the `separate` construct, and it is recommended that the tools adopt this approach.

If the entity is an object, implementation involves modifying its interface and creating its body. The initial body is a set of stubs for each interface procedure entity in its specification. If the entity is an interface object, the interface and implementation of the enclosing object must be modified as well. As with interface procedures, the enclosing entity must already be implemented. To prevent having to modify the package body each time an interface procedure is implemented, the tools should make use of Ada's 'separate' facility. That is, implementing a package body would produce the following:

```
package body name is
    procedure VP1 is separate;
    procedure VP2 is separate;
end name;
```

Implementing the interface procedures VP1 and VP2 would produce the following stubs:

```
separate(name)
procedure VP1 is
begin
end VP1;

separate(name)
procedure VP2 is
begin
end VP2;
```

3.3 Changing an Entity

Normally, a designer is forbidden by the tools from making certain changes. The tools should not allow a designer implementing an object to change the object's interface, since implementation of an object is a separate operation. However, there are obviously circumstances where it is desirable to make such a change. In order to preserve the software structure, the tools carefully control such situations.

There are many ways an entity may be modified. The tools categorize modifications based on their potential effect on other entities, as follows:

1. The change could be to an entity's body, made in such a way as to have no effect on other entities. For example, an algorithm might be changed for performance reasons. (Furthermore, changes to an entity's body affect other entities—if an algorithm's semantics change, for instance.)

2. The change could be to an entity's specification. For example, a parameter could be added to a procedure. Such a change necessitates changes to all entities that call the procedure, since its calling sequence has been altered. (Note, however, that adding an interface procedure to an object does *not* affect other entities—disambiguating problems aside.)

3. The change could be to an entity's relationship to other entities. For example, a hidden object might be converted into a unit object after its general utility is discovered, or after a design flaw is revealed that necessitates a change to the software structure. Such a change does not affect other entities, but has an effect on project's structure and organization.

This is a broad class of changes and we do not wish to group so many types of modification under a single command. Therefore we propose the following terminology. 'Edit' is a word that is commonly used to describe changes one considers to be made in isolation of other entities; therefore we use it to describe changes of type 1. 'Modify' refers to type 2 changes, and 'reuse' refers to type 3 changes.

3.3.1 Edit

The definition of editing implies that it may only be applied to an entity's body. Any body may be edited so long as the change does not introduce a conflict with the specification. For example, it would be illegal when editing an Ada package's body to change a procedure's interface in the body.

An 'Edit' change assures that all entities using the changed entity will compile without change. It does guarantee that the new system will behave identically, since the change can alter semantics. In fact, the 'edit' operation is intended to effect bug fixes and to enhance existing code. It should not be taken as a guarantee of similar functionality. It and the other two types of changes are intended more to help effect configuration management: the type of modification determines the scope of changes in the rest of the system. A useful side effect of the tool that performs editing might be to notify all developers of objects that use the newly-edited entity. This should probably be handled through mechanisms of a configuration management environment such as OVERSEE [7].

3.3.2 Modification

A 'modify'-type change requires certain changes to other entities. The change may be straightforward, such as adding an extra parameter to a procedure, or it may be more extensive, necessitating changes to algorithms or data declarations. In almost all cases, the change required will always be an 'edit', not a 'modify'. This is true because the changed entity is usually referenced in an implementation, not a specification part. For example, adding a parameter to a procedure requires editing the implementations of all procedures that call it, but not changing their specifications. Some exceptions to this rule do arise. Consider modifying a data type D in a package specification P. If P is used by some other package Q, the specification of Q may want to use D for procedure parameters, variable declarations, *etc*. A change to D would therefore affect Q's specification.

Considering the nature of modification, entities that use a modified entity will probably not compile. As with editing changes, therefore, the tools should automatically report the scope of the change to all potentially affected developers. The analysis is more complex than with editing, since changes are more widespread. Also, whereas an 'edit' may require another entity to be edited, 'modify' changes may require another entity to be either edited or changed, as described above. The tools must thus be able to categorize and report the nature of the change. In fact, tool support to automatically effect some types of changes (*e.g.*, renaming) should be available.

3.3.2.1 modify package

The above discussion implies that certain objects may not simply be edited; they must be changed through particular commands. Such is the case with packages. The following types of transformations may legally be applied to a package:

1. Interface procedures and objects may be added to or deleted from its specification. (However, the entities themselves may not be changed; to change an interface procedure's parameter schema, for instance, one must modify that procedure.)

2. Information such as data types and variables may be added, deleted or changed.

3. Data types and variables (but again, not procedure or package bodies) may be altered in its implementation. Note that these changes require only changes to entities that are part of the package.

3.3.2.2 modify procedure

Changes to the specification and changes to the body are treated separately. The former are 'modify' changes, whereas the latter are 'edit' changes. To modify a procedure's specification is to make more or less arbitrary changes to anything except its name.

3.3.3 Reuse

To reuse an entity is to convert it from a hidden entity into a unit entity. This has no effect on whether a program will compile. However, it obviously has drastic impact on a system's software structure. The impact is apparent in terms of the directory arrangement, but *not* in terms of the virtual software structure. Hence the navigation commands described below should not be affected; however, the library containing the entity obtains a new globally visible unit.

Not all entities are reusable. To be reusable an entity must not be tied to a hidden entity (or levels of abstraction are violated). The operation should be able to detect this.

3.4 Deletion

One can delete an entity from the software structure via the command:

```
delete entity-name
```

Clearly this command not without danger. If an entity *D* is deleted, and another entity *B* is used by *D* but no others, then to delete *D* would leave *B* 'dangling.' In such a situation, *B* can be deleted too or made global, at the user's option.

The effect also differs depending on whether the entity being deleted is global or not. A global entity is removed from the software structure, but is not physically deleted. A hidden entity is both removed and deleted. The gross implications of this should require user confirmation.

3.5 Renaming

Entities are renamed via the command:

```
rename [ entity_name ] to new_name
```

If the first entity name is omitted, the current entity is used. The entity's name is changed in all necessary contexts. This involves changes to:

1. Any text files containing the entity's name (both specification and implementation files).
2. Any directory structures storing the entity.

In addition, entities that use the newly-named entity must be changed.

3.6 Navigation

3.6.1 set current entity

This operation sets the current entity being edited (or simply viewed). Its form is:

```
set current entity to entity
```

The effect is to search through the list of known objects and establish a proper context in which to view or edit the given entity. The entity's specification and (direct) implementation will be visible; also, pointers to the entities it uses will be visible. Nothing else should be visible. The entity must already exist.

3.6.2 up

This operation is similar in intent to the Unix command 'cd ..', which sets the current directory one level higher. Its form is simply:

```
up [ to-where ]
```

It moves the current directory up one level, which should be equivalent to moving up one level of abstraction in the architecture. However, the operation is somewhat more complex than the aforementioned Unix command: Unix directories are always trees, but software structures are not. If up is invoked with no argument in such a situation, it reports an error and does not change the current entity. If it is given an argument, however, it disambiguates by changing to the entity named by the argument (which must be a parent of the current entity).

Like the Unix file system, a root directory entity exists. It is not possible to move up from there; if the 'up' command is invoked at that level, it has no effect.

3.7 Queries

The queries provide useful information about a project's architecture. They work relative to the current entity, although they can also take entity arguments that specify objects in other areas.

3.7.1 Display Interface

This command has the form:

```
display interface [ of name ]
```

It displays the interface of the given entity (the current entity, if none is given). The exact nature of the display depends, of course, on whether the entity in question is an object, procedural abstraction or task.

3.7.2 Display Implementation

This command has the form:

```
display implementation [ of name ]
```

Each package has certain entities that constitute its implementation. This command provides a summary of those entities. The only ones given are those directly involved in the implementation of the package. Specifically:

1. Those packages on the 'with' list that are not global.
2. Those procedures and packages contained within the package body, but not in the specification.

3.7.3 Display Usage

This command has the form:

```
display usage [ of name ]
```

Each entity is used by other entities in the software structure. (In Ada, the set of entities that "use" entity E are those that have the line "with E" preceding their declaration.) This command lists those entities for the named entity. If no name is given, a current entity must be set, and the command will list the entities that use the current entity.

3.7.4 Display Reusability

This command has the form:

```
display reusability [ of name ]
```

It gives information on how potentially 'reusable' an entity might be. Reusability is determined by whether the object can be made global without changes to the software architecture. If so, the object is considered reusable; if not, then it is considered not reusable.

3.7.5 Display Reuse Needs

This command has the form:

```
display reuse needs [ of name ]
```

For an object determined not reusable (by the 'display reusability' query), this command states what entities prevent it from being re-used. (If the entity is reusable, then the command produces no output.)

3.7.6 Display Structure

This command has the form:

```
display structure [ of name ]
```

It displays the structure of the named entity. It could present the display in several forms:

— As a partially-ordered set. It might produce lines such as:

```
addcbs : rcs_file_p generate_change_list
generate_change_list : rcs_file_p unix_env
```

to indicate that the entity 'addcbs' depends on the entities 'rcs_file_p' and 'generate_change_list', and that 'generate_change_list' depends on the entities 'rcs_file_p' and 'unix_env'.

— As a graph. This would be along the lines of what can be done using the Unix System V graphics filters, in particular 'dtoc'.

— As input to the 'pic' tool. This choice differs from the previous one in that it is intended to produce something suitable for inclusion in documents.

A partially-ordered set is perhaps the most flexible approach, since it can be used, among other things, as input to a postprocessor that produces a graphical display. Simply producing a graph is obviously the least flexible. However, if a partially-ordered set is produced, extra work is required to write postprocessors.

This command is normally intended to work for an entire system. It may be applied to subsystems as well. Note that it re-orders the global/hidden package notions. That is, if `A` uses global packages `B` and `C`, then displaying the project structure should still produce some kind of hierarchy with `A` on top.

4. IMPLEMENTATION SUGGESTIONS

These operations are to be implemented first on a computer running the 4.2 BSD version of the Unix[†] operating system. As noted elsewhere [8], certain representations described in this are impractical under a standard hierarchical arrangement, due to the lack of circularity. Symbolic links can resolve this problem to a certain degree, but care is required, and anyway some Unix utilities do not fully understand symbolic links.

Therefore, not all relationships can be expressed accurately under Unix. Moreover, even for a purely hierarchical file system, a straight representation of entities (using files and directories) tends to produce trees of great breadth and depth, since directories and files introduce a level of indirection during entity reference. The initial version of the tools should therefore be kept simple by implementing only the structures as described in [8].

A suggested implementation strategy is as follows. A 'library' is a set of directories, all organized under one central directory. Each of these directories constitutes an entity with global visibility. In it is stored information in the format proposed in [8]. Directories are created when the 'create' command is used.

If a module builder decides to use an existing global entity, then no relationship will be explicitly shown in the file structures (note that it is expressed by `with` clauses). However, if an entity is re-used, and hence made global, then the fact that it originally was part of some software structure will be shown through a symbolic link to the entity's directory. (This should help in preserving information about a project's evolution.)

As an example, consider Figure 5. It shows the directory structures for a library that contains three entities: a procedure, a package and a task. Each is global. The procedure's implementation is shown in some detail. Assume also that the procedure uses package 'pkg'. By the above rule, we can infer that the package was constructed independently from the procedure.

Now suppose that hidden package 'p' is to be made global. The structure changes to that shown in Figure 6, because global entities all exist at the same level within a library. The dotted line shows the symbolic link. It allows developers to easily trace the system's original architecture. While the information still can be derived, the use of the symbolic links is intended to help developers easily follow through the architecture, and to navigate through the directory structure. In particular, 'up' would have rather different semantics were the symbolic link not present; with the link, a developer can easily examine package `p` from within the hidden directory, and then revert back to the hidden directory.

The commands share certain information. In the Unix implementation, for example, two entities are sometimes stored in the same Unix directory. Which one is "current" must be determined in some manner. Either of the following methods should be acceptable:

1. Use a file in the directory whose name begins with a period, so that it is not immediately visible through the 'ls' command.
2. Set an environment variable to the value, using aliases and the style of the SPMS commands [6].

The second approach requires some initial effort on the user's part, but is less clumsy and, in the event of a crash, does not leave extraneous files. Either approach should do for the first implementation.

The **set current entity** command could be implemented as in SPMS, but the number of directories is potentially large. To keep its performance acceptably high, it is recommended that a database of symbolic links be created and maintained by the creation commands. They will be placed in a directory at the same level as the main library, perhaps named for the library but prefixed with a period. The command can then search this directory to see where an entity is located. In fact, this also provides a simple means to quickly check existence. Since symbolic links consume little space and can be created quickly, this scheme should add little overhead.

A command-driven interface will probably exist. An abbreviation scheme is recommended. Ledgard's method of using the first letter of each word in a command [5] is a good candidate. However, it is not immediately suitable. The commands in Table 1 would be ambiguous under this system. Either the commands should be renamed, or

[†] Unix is a registered trademark of AT&T Bell Laboratories.

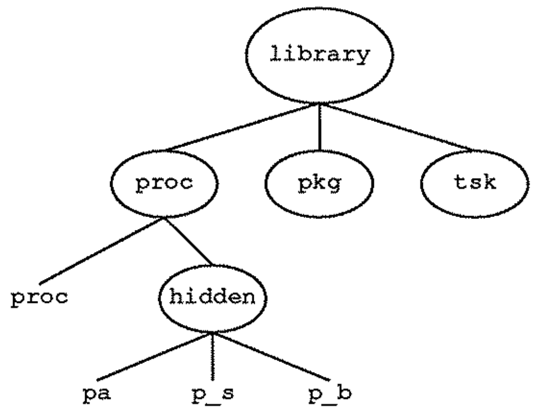

Figure 5: Initial Library Configuration

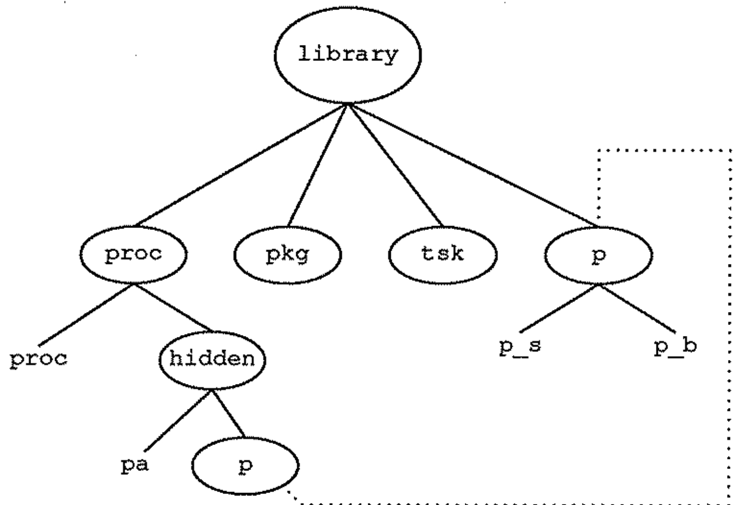

Figure 6: Library Configuration after Re-Using Package p

another abbreviation scheme must be used.

REFERENCES

[1] *Ada Language Reference Manual*, ANSI/MIL-STD-1815A, American National Standards Institute, Inc., 1983.

[2] G. Booch, "Object-Oriented Development," *IEEE Trans. on Software Eng. SE-12*, 2 (Feb. 1986), pp. 211–221.

[3] P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Trans. Database Systems 1*, 1 (Jan. 1975), pp. 9–36.

[4] F. Gallo, R. Minot and I. Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System," *SIGPLAN Notices 22*, 1 (Jan. 1987), pp. 12–15.

[5] H. Ledgard, *Professional Programming*, Addison Wesley, Reading, MA, 1987.

[6] P. Nicklin, *The SPMS Software Project Management System*, Unix Programmer's Manual (4.2 Berkeley Software Distribution), Berkeley, CA, Aug. 1983.

[7] S. Wartik, "Rapidly Evolving Software and the OVERSEE Environment," *SIGPLAN Notices 22*, 1 (Jan. 1987).

[8] S. Wartik, "File System Structures for Object-Oriented Design," *Proc. Fifth National Conf. on Ada Technology*, Washington, D.C., Mar. 1987, pp. 411–419.