```
 III    N    N  TTTTT  EEEEE  RRRR    FFFFF    AAA     CCCC  EEEEE
  I     N    N    T    E      R   R   F       A   A   C      E
  I     NN   N    T    E      R   R   F       A   A   C      E
  I     N N  N    T    EEEE   RRRR    FFFF    A   A   C      EEEE
  I     N   NN    T    E      R R     F       AAAAA   C      E
  I     N    N    T    E      R  R    F       A   A   C      E
 III    N    N    T    EEEEE  R   R   F       A   A    CCCC  EEEEE



L       PPPP    TTTTT          1
L       P   P     T           11
L       P   P     T            1
L       PPPP      T            1
L       P         T            1
L       P         T      ..    1
LLLLL   P         T      ..  111
```

START Job INTERF Req #92 for EGB Date 23-Jul-82 14:45:55 Monitor: Rational
File RM:<RPE.DOC>INTERFACE.LPT.1, created: 11-Jul-82 17:51:18
printed: 23-Jul-82 14:45:55
Job parameters: Request created:23-Jul-82 14:45:53 Page limit:126 Forms:NORMA
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:A

Rational Machines Incorporated

A Rational Interactive Ada Development and Nurturing Environment

User Interface
DRAFT 8

Ariadne. In Greek mythology, she gave Theseus thread to guide him out of the labyrinth.

Table of Contents

Chapter 1
Goals and Objectives

The goal of the environment is to provide a set of facilities that enhances the productivity of users of the R1000 system. The following characteristics have been identified as important:

- Simple

  * Easy to learn and use
  * Easy to do simple things; possible to do complex things

- Consistent

  * Basic principles apply uniformly
  * Compatible with rules and operations of Ada
  * Predictable in terms of a relatively simple model

- Friendly

  * Small amount of typing to get things done
  * Assistance with syntax and semantics
  * On-line documentation and help
  * User retains control interaction sequence
  * Hard to accidently lose data

- Powerful

  * All of Ada available directly to the user
  * System functions directly visible and available
  * No distinction between system and user functions

In general, the user interface is a state-of-the-art facility with respect to ease of use, functionality, and user friendliness. It strives to meet all of these goals, resolving any conflicts so as to favor the operations commonly performed during software development tasks.

Chapter 2
Concepts and Paradigm

Ariadne is an object-oriented environment polarized by Ada. Information is organized into a number of Ada objects: program units, types, variables, etc. Each object has a number of attributes, including a type and name. The type of the object determines what functions can be applied to the object, following the rules of Ada where appropriate. The user manipulates the environment by applying operations (procedures) to objects, and by creating and deleting objects. Users apply operations to objects to display and change their contents, display and change their properties, and perform related alterations of an object's state.

Users can structure their objects into a hierarchy based on containment

directly analagous to the way Ada programs are organized by using directory packages. Directory packages have the static characteristics of elaborated Ada packages. Each directory package can contain type declarations, procedures, variables or tasks -- all of the objects that can be contained in any Ada package.

spec/body? exceptions? other packages? private part?

2.1. Ada Orientation

One of the basic tenets of Ariadne is the usefulness of Ada as a form of communication between the user and the system. Ada's role in this communication is multi-faceted:

1. Ada is the system command language.

   User commands are expressed in Ada. Syntactic/semantic completion may be used to assist in reducing the user memory and typing requirements, but the command that is actually executed will have an Ada form and Ada semantics.

2. Ada defines system semantics.

   Since all of the objects in the system are defined according to Ada rules, command syntax is pre-defined and command parameters can be deduced from Ada definition of the command.

3. Ada provides the system structure.

   The view the user sees of the system is an Ada package nested inside other Ada packages that provide basic system functions.

4. Ada tasks provide traditional operating system facilities.

   A task provides the natural correspondent of a conventional file -- a variable-length collection of (typed) data values accessible through a set of well-defined operations. Similarly, most built-in operating system facilities can be structured as tasks, softening the distinction between system- and user-defined facilities.

2.2. Extensions to Ada

Ada was designed as a programming language, and though most of the constructs map well into environment constructs, some of the compiler-oriented constraints cannot be enforced within the Ariadne uses of those constructs. That is, Ada rules have been extended in exactly those places where there exists no direct Ada equivalency:

- Dynamic modification of directory packages.

- Execution of user commands.

These problems derive from inserting the user into the middle of the edit-compile-execute cycle, a fundamental advantage of an integrated environment.

2.2.1. Directory Packages

Standard Ada packages do not change form after they are instantiated. Ada semantics, quite appropriately, include no notion of dynamic changes to programs during execution. For directory purposes, however, it must be possible to add new objects to the package without losing the state of the pre-existing nested objects (as would be done if the entire package were re-compiled).

This limitation is overcome by a process called re-elaboration. Re-elaboration allows dynamic changes to the package instantiation to be effected as if the package had originally been compiled in its altered form. Although normal Ada packages do not share the ability to be re-elaborated with their directory counterparts, directory packages retain all Ada-related capabilities -- user procedures can call directory procedures, modify directory variables and use directory types.

2.2.2. Commands

Ada defines a semantics for statement execution. This semantics does not recognize the possibility of compiling and executing an Ada statement from within an Ada program. As a result, user commands, even though they take the form of Ada statements, do not fit.

This can be solved by introducing a "magic" procedure, Eval, that takes a string representing a valid Ada statement and executes it. This function, or a similar one, has to exist somewhere if Ada is to be the Ariadne command language. Confirming its existence merely reinforces the user's view of the system as a set of Ada objects. Ada statements cannot execute in a vacuum -- a context for this execution must be defined.

Eval causes commands to be executed within a task. A package can only support a single thread of execution at any given time. Having commands execute as if they were inserted directly into an existing package transmits this limitation to user interactions, placing unacceptable restrictions on both inter-user and intra-user concurrency.

The context of the eval-generated task is easiest to explain if it corresponds to the end of the body of the current directory package. Ease of access will greatly encourage the user to include all objects in the visible part of his package and it is possible to provide sufficient facilities that the user can still have a reasonable degree of protection. The majority of these decisions are isolated in the eval procedure and establishing its context, making it possible to experiment before choosing.

There are also facilities to aid in the entry and processing of commands. These include syntactic and semantic completion of command names, object names, and other parameters. Facilities for the easy re-execution of

previous commands or commands similar to previous commands are provided. All of these features derive from the editor-based organization of the Ariadne interface.

2.3. Editor-Based User Interaction

Users can manipulate objects by issuing specific commands that request a change, or by making the change directly by editing the object. The two processes are closely intertwined, since commands are entered for execution using the facilities of the editor.

The editor allows the user to display information about an object. This information describes the object's name, state, contents, and/or other attributes of the object. The specific form of this display is called an object view. The specific contents of the display is called an object image.

The basic editor paradigm is that the user changes the object image using editor operations and, periodically, the actual object is made consistent with its edited image. This may, depending on the circumstances, be done at the request of the user or automatically by the system. In any case, the user can think of the process as editing the object itself.

In the case of Ada, the underlying object is the Diana representation and the image a pretty-printed program "source". For the debugger, the underlying object is an executing program instance (some of whose characteristics are interpreted by reference to the Diana tree). It is possible for either the image to change (due to user action) or for the underlying object to change (due to program execution).

Changes to the image are subject to syntactic and semantic constraints before being reflected in the underlying object.

2.3.1. Selection and Action

To make changes to the environment, the user must specify the object to be affected, and the action to be performed on it. This may result in a change to the object itself, to the image of the object, to the view of the image of the object, or to the editor's state.

The general method for changing something is to first point at it with the editor's cursor and then to specify the action to be performed. The object to be affected is called the selected object and the process of the user specifying the object is called selection. The action is then specified by the user by pressing an appropriate key sequence.

2.3.2. Structure of Object Images

Images displayed on the screen are structured so that the user can deal with an object image in units appropriate for that object.

An Ada program, for example, consists of a tree of objects consisting at the lowest level of identifiers, keywords, operators, etc., and at higher levels of expressions and statements, etc.

This nested object structure is reflected in the corresponding image in terms of object cursors, positions in the image that include the entire extent of a chosen object, not just a point in its text. An object cursor represents a region in the image to which the user can apply operations. More general regions can be created, but the object-oriented region retains an important role in the use of the system because of the knowledge built into the editor that facilitates object selection, movement and manipulation.

A uniform mechanism, prompting, is used to show places in the image where additional information is required to complete the object. In addition, information about the next permissible entry at any point can be obtained through syntactic or semantic completion facilities.

2.3.3. Display Control

It is the rule, rather than the exception, that top-level objects are too large to be entirely displayed at any one time. A typical response to this problem is windowing: clipping the image at the edge of the display region. This works because proximity is a good indicator of interest. Detail control makes it possible to select a level of abstraction (based on object nesting) for viewing. Explicit ellision facilities make it possible to directly control the viewing level for specific objects. The ability to follow logical links between items, e.g. show the declaration of an object, makes it possible to go beyond even hierarchical contiguity in showing the user what he needs to see.

2.3.4. Undo

A general Undo facility will be provided to allow the user to back out of actions that he has taken to change objects. The granularity of this mechanism and the relation between image and object undoing operations is not clear.

2.3.5. User program interaction

Programs that require user input read it from an abstract stream that is generated either by the user using the editor, or by another program (under control of the user).

When the user wishes to type input for a program that requests it, the input is entered into a log file that is read as the program executes. This file is editable using the normal editor operations. The user program reads this data as if it were typed sequentially.

Thus, the editor is always present and available to the user for handling entry of user-supplied data.

2.4. Predefined Objects

The R1000 system provides objects of a number of types and operations that apply to them as part of the initial system. Users can create new object types and new operations at any time and add them to the system. These types and operations are then available.

2.5. Predefined Types

The basic types defined by the Ada language are built into the environment. These include scalar types (integer, enumeration), structured types (arrays, records), and types themselves. Program components such as packages and tasks are also included.

In addition, the R1000 environment provides several other types. Included are file, documents of various kinds, and Ada program-related types. The Ada-related types include package source and runnable versions. The standard object-image correspondance supplied makes it possible to edit any object in the system. This implies the ability to obtain the text image for display, printing, etc. In addition, it is possible to perform each of the following:

- Create/delete/rename an object
- Move/copy an object to a new location
- Display/alter access rights for an object
- Change password or other access control mechanism
- Grant capability to user identity

Specific characteristics of predefined types.

Type	Operations
string	Execute as a command
type	Find instances Show declaration Aggregate template
package	Kill instantiation Semantic queries Cross reference Control of pretty-printing format Syntax/semantic search Control of optimization within compiler Control user views

task	Show state/resource usage of running tasks
	Suspend/resume/abort a task
	Kill module instantiation
	Edit task priority
	Edit task resource limits
	Show program state & context
window	Set currently active
	Move
	Set size
	List ~~directory of~~ windows
	Change visibility
	Create/edit key-command bindings
	Enter/edit a keyboard macro
	Execute a keyboard macro
Session	Attach to existing session
	Create/suspend/destroy new session
	List sessions
	Change access rights of current session
mail	Send mail
	Show incoming mail
	Display and edit aliases
	Display end edit distribution lists
	Display messages
	Delete message
	Forward message
	Reply to message
user	Force logoff
	Show global accounting information
	Set resource limits
any object with version control	Show revision history
	Make a new version
	Merge versions
	Make parallel changes in several versions
	Add history information (comment)
	Install revision
	Show differences between versions

Chapter 3
Package Control

3.1. Introduction

Packages are separated into two groups: those that are elaborated once and remain static for their entire lifetime and those that are elaborated and de-elaborated (possibly) several times. The former are referred to as "directory packages"; the later are characteristic of packages that are part of normal Ada programs.

A hierarchical nesting of packages accomplishes the same purpose as directory trees do in conventional systems. There are, however, a number of crucial differences, both in how the user views these packages and on the ramifications for how the system seems to work.

3.2. User View

At the beginning of a session, the user starts out in his "home package", looking at a highly ellided display of its contents. Although the home package has both a visible part and a body, screen space limitations make it desirable to limit redundancy.

The following is an ellided, but otherwise source-oriented view of a typical home directory on TOPS-20.

```
package Jim is
    Keys    : Text_File;
    Meeting : Document;
    package Apse is ... end Apse;
    package Red  is ... end Red;
end Jim;


package body Jim is
    Editor_Options  : Editor.Options;
    Mail_Box        : Mail.User;
    package Misc is ... end Misc;
    package body Apse is ... end Apse;
    package body Red  is ... end Red;
    package body Misc is ... end Misc;
begin ...
end Jim;
```

Neither the body nor the visible part is a complete representation of the user "directory". The body is necessary to hold procedure bodies, but cannot hold variables that need to be seen by others. On the other hand, there is a large amount of redundancy in the two parts that the user need not see every time he views the package.

A more useful view of this package, for directory purposes, might be:

```
package Jim is
    Keys    : Text_File;
    Meeting : Document;
    package Apse is ... end Apse;
    package Red  is ... end Red; ...
package body Jim is
    Editor_Options  : Editor.Options;
    Mail_Box        : Mail.Use;
    package Misc is ... end Misc; ...
end Jim;
```

← I don't think this is a good idea!

The information content is the same as for the original. To maintain Ada ordering on the package may require a large number of ellipsis marks at the ends of lines. Visibility and elaboration order characteristics are preserved, though the placement of the bodies is ignored.

A potential problem with this arrangement is that the source distance between the two portions of this display can be very large. According to strict Ada rules (see 3.7), the visible parts of all users would probably precede all of the corresponding bodies, considerably separating the two parts of the display. As long as Ariadne provides an easy way to juxtapose the two pieces, the user seems unlikely to bridle at (or even notice) the difference between appearance and "reality".

Should Ariadne create the spec (body) when the body (spec) is created?

3.3. Selective Visibility

Each Ada package is represented in the system as a Diana tree and presented to the user as an Ada source program. What appears in the source representation of the package determines its visible objects and their characteristics. By providing the ability to specify different views to different users, the owner of the package can control the access other users have to the package and its contents. How?

The visible part/body paradigm provides a reasonable mechanism for controlling access in programs, but is not sufficient for all environment purposes. The goal is to extend this facility to provide more flexibility without introducing new Ada constructs or compromising the security of those that already exist.

This is accomplished by providing the user facilities for specifying how a package should be presented to a particular user. Specifically, this visibility can be changed in two ways:

1. Object visibility. Remove object(s) from the visibility of other object(s).

2. Available operations. Do not allow operation(s) to be used on a particular object.

Among the reasons for using multiple views is that they allow each of the

How do I show him please?

users who can see the package to see it as if it were a complete and correct Ada object. The elaborated package instance is created to conform to the owner's view. Nothing in the view-generation process can grant access that would not be available to the package owner under Ada rules. Specifically:

1. Objects in the body for the owner cannot be promoted to the visible part of other views.

2. User views can add constraints to owner objects, but cannot remove them.

3. Each subordinate view must be a legal Ada package in its own right.

Such limitations do not extend the set of legal programs, so any program developed under the more restrictive regime can also be constructed without the restraints.

3.4. Views

Using this method, the example package from above might look like:

```
package Jim is
    Keys    : Read_Only_Text_File;
    Meeting : constant Document;
    package Apse is ... end Apse;
end Jim;
```

This is a consistent Ada view of the package, although more limited than the owner's:

1. Package Red is no longer visible, so cannot be referenced.

2. Keys is now a Read_Only_Text_File, derived from Text_File without modification operations. An alternative form would be to retain the same type, Text_File, and limit the visible operations.

3. Meeting is now a constant and cannot be changed. Making the object constant is logically an operation limitation, but Ada doesn't allow explicit assignment procedures.

4. No package body is visible because this user is not authorized to see the body of the package (see 3.6).

This is a form of compile-time access control. Access to an object from a program is established at compile-time (re-elaboration time for directory packages). Revocation is possible using the re-compilation rules, i.e. if

How do I create these, how do they look from outside...

How do I tell who can see what?

the object is made non-visible, packages that depend on it must be re-compiled. Once an elaborated program has access to an object, that access cannot be revoked until the program becomes de-elaborated.

The architecture also provides run-time support for these access constraints, though there is some debate about how to provide operation limitation in the general case.

Access to an object can be granted selectively to specific directory packages. If access to an object is granted to a directory package, then all objects within that package have the same access rights to the object as granted to the directory.

3.4.1. Implementation

The logical place to implement multiple views is in Diana. This allows Diana to enforce the visibility for the compiler and other tools. On the other hand, it introduces the possibility of doing access control checking on a large number of objects during each compilation. A possible compromise would be to only allow access control at the package level, but this removes a fair amount of flexibility.

The user needs a natural interface with which to specify how others will view the package without confusion. Possible methods (not necessarily mutually exclusive):

1. Edit the context for each group of users.
2. Place pragmas to indicate visibility.
3. Specify defaults for particular users for new objects.

3.5. Dynamic Access Control

For any object with a proceduralized access pattern, it is possible to implement access control by means of a programmed checks of the user against some sort of access list. The implementor can enforce opening before use (checking the initial access and handing out a value to be used in future communication) or a can check with each access. Checking with each access makes it impossible to pass the right to perform operations onto another user's task, where an open object descriptor can be passed as a parameter.

Dynamic checking is somewhat more expensive than static checking, but provides the owner with an easier way of changing access rights. Unfortunately, dynamic access controls don't display as naturally in the user's view of the system unless files are treated as special objects known to the system. This form of dynamic checking is exactly analogous to that provided by conventional systems.

So how do we display?

3.6. Modification Control

Some mechanism must be exist to regulate the modification of package and view contents and the ability to execute programs within the a particular context.

Conceptually, access to a package can be controlled in exactly the same way access to any other object can be: reduce the apparent rights of other users to the operations on the object. In the case of packages, the type is Diana-derived and includes operations to view the object, view its body, re-elaborate it, etc.

Another possibility is to use a dynamic access control mechanism similar to that used for files. The static method would probably work as well, but the user's view is unchanged by the choice and there is less complexity introduced to support the types of operations that are anticipated. Further, environment interactions with the source-views of different packages are not frequent enough to justify concern for efficiency.

3.7. Linear Time-Order Elaboration

Ada visibility rules are based on the notion of linear elaboration. The concept of linearity used is one that is appropriate to a compiler, i.e. appearance order in the program text. Ariadne will need to use a time-order rule of linear elaboration. That is, any item that is included in a directory package must already be elaborated. Since directory packages go through many generations of re-elaboration, this makes it possible to create mutually referential visible parts for directory packages. This is desirable for the directory functions, removing the need to order users, but not desirable for static packages that are being used as library units in an implementation. There may need to be an explicit pragma that relaxes the elaboration rules to prevent mutual references in library packages.

Chapter 4
Roget

The RMI Object Generating and Editing Tool is the editor-based front-end for Ariadne. As such, it fulfills the combined purposes of the system editor and virtual terminal, appearing to the user to be the single point of interaction with all parts of the system. Roget is the focus for providing the user with the same interface throughout his terminal session, somewhat independent of the characteristics of the particular system facilities he uses.

4.1. Basic Characteristics

Roget is essentially a multi-window screen editor that has been inserted between the user and the rest of Ariadne. If done properly, most users will be unable to make the distinction between the editor interface and the

functions it provides access to. A key to this transparency is the ability to deal with text representations as if they were the underlying object, not merely a print representation of it.

4.1.1. Objects and Presentation

Each object has some representation in the system that is convenient for its implementor, e.g. Diana for Ada programs. Each object also has a user-readable representation that is used for communication between the user and the system. This representation is called the image of the object. For each object, there is exactly one image. To provide flexibility in formatting the image for a particular purpose, an image may have more than one view.

Views are presentations of an image in a form tailored to the user's specific needs.

4.2. Cursors

One of the problems with discussing cursors is that there are really a large number of position-specifiers that are called "cursors". This is generally confusing, but particularly when discussing the distinctions between the various types. To minimize confusion, the following terms will be used to specify which cursor is being discussed.

1. Physical cursor. The blinking block/underline that the user actually sees on the screen; of interest only when the screen is not being redrawn, i.e. when it is resting on a particular screen position.

2. Point cursor. A logical position between two characters in the human-readable image of an object. The physical cursor appears over the character to the right of the point cursor.

3. Input cursor. The point cursor where new user input will be placed.

4. Output cursor. The point cursor where a new output item will be placed.

5. Viewing cursor. The point cursor representing the current focus of user interest; often identical to the position represented by the input and/or output cursors.

6. Object cursor. A region surrounding a point cursor corresponding to a logical object. Since objects are nested, more than one object cursor is possible for most point cursors.

7. Selected region. A region of the human-readable image used by the user for text manipulation. May correspond to an object cursor, but need not.

4.2.1. Cursor Location

Consider the problem of specifying where the physical cursor should be after the user enters a change of some sort. Leaving aside cases in which there is an active effort to move the cursor to the next reasonable location, there remains the problem of how to re-place the physical cursor where the user expects it. Remember that the user can have entered arbitrarily complicated program changes prior to hitting <ENTER>. The case against a few of the obvious candidates:

1. Absolute line-column position. New lines could have been introduced and/or indentation may have changed.

2. Specific line, relative column from line indentation. Breaking up the current line can cause one input line to be part of several output lines.

3. Position within an object. Works best if finest-grain objects are small and do not overlap more than one line, but the object may have disappeared in the change.

The last alternative seems the most promising, assuming that the situation in which the object disappears can be handled. Independent of implementation, the user must be able to reasonably predict where the cursor will end up. Otherwise, he will be forced to wait and see what happened before he can make his next keystroke decision. In some applications, it is expected that the ability to locate the cursor may change the semantics of the operations performed. For instance, command completion can depend on the current cursor position.

4.2.2. Aggregate Positions

Objects are structured hierarchically by each object editor to provide natural groupings based on the semantics of the object type. For each point cursor, there is a smallest containing object. Object cursor operations allow the scope of an object cursor to be expanded (contracted) to each level of containing (contained) object. There are also operations to move to the next (previous) object at the same level.

In addition to providing movement, object cursors provide a quick way of selecting sections of the image that correspond to natural units for the object being edited. The user is able to take advantage of this to move, copy, etc. logically connected blocks of text. In addition to object cursors, it is also possible for the user to directly select a region between two points or to extend an object cursor by appending other objects or character positions.

4.3. Display Characteristics

Although the representation that the user sees consists of characters that are meant to convey the contents of objects, not all of the characters that

appear in the image directly correspond to themselves in the underlying object. In particular, two kinds of meta-text appear in images, prompts and ellipsis.

An ellipsis, commonly displayed as "...", is a visual indication that more information is contained in the image (and the object) than is currently displayed to the user. This serves to allow the user more flexible management of available screen space than merely selecting rectangular subareas of the detailed image.

Ellipsis is used in two related contexts:

1. Global detail filtering and initial view selection, where the system chooses (with user assistance) a particular presentation of the object that is expected to provide the best compromise between detail and available space.

2. Specific user control over detail. Presented with a particular view, the user can tailor it by expanding an ellipsis that appears on the screen or by requesting that an existing object be ellided, increasing the range of objects visible.

Ellided areas are not directly editable. It makes no sense to change the number of dots in the ellipsis or insert text in the middle. Deletion of the entire ellipsis, though possible quite drastic, is well defined, however.

4.3.1. Prompting

A prompt is a visual reminder of an incomplete object. The prompt occupies the position where an entry is needed to complete the underlying object and takes the form of a (somehow) highlighted reminder as to the nature of the required entry.

Prompts automatically disappear when anything is typed over them (or deleted from them), the assumption being that the new entry will replace the prompt. For any other text-related operations, each character of the prompt is just like any other on the screen, and the prompt is a complete object.

4.3.2. Following Object Nesting

Due to the nested nature of objects, it is possible to move down the directory package tree by expanding the detail of each contained object. At some point, each new level should become a distinct window so that the operation becomes one of selecting a new window to handle the indicated object in more detail. This leads to the problem of creating and managing windows.

4.4. Window Management

Describe how windows are created, managed, and destroyed.

Will want a way to find out what windows there are, how they were created and what their "status" is.

Will want to have a status display that makes it clear when exceptional events occur, e.g. mail, program waiting for input, etc. Possibilities are blinking notices and rotating banners.

4.5. Object Completion

Describe the mechanism for the various forms of creating complete objects from incomplete user entries. Similar mechanisms should be available for commands, keywords, data names, etc.

4.6. Undo

Describe the object/text undo facilities.

Chapter 5
Ada as a Command Language

So we're going to use Ada as the command language, how do we make it work.

There is the problem of declaration that is solved by nested directory packages from which is inherited objects and operations.

There is the notion of Eval to take what the user types and convert it into Ada tasks.

5.1. Command Interpretation

Explain how Eval works.

5.2. Naming and Access Control

Is this different from the access control chapter, other than how do you make the commonly-desired names more accessible to the user.

5.3. Context Definition

How does the user define the context in which execution takes place.

How does the user determine what sort of context (source, compiled, code-

generated, instantiated, running) in which he is currently editing. Does he have an independent context for each of these types of source situations.

Chapter 6
Interactive Program Input/Output

Programs need one or more places to write output. Some of that output may be "intended" for the user and some of it may be the result of the computation to be saved. Users are relatively used to the notion of an interactive session in which the user input is interspersed in the output. In program terms, this usually manifests itself as calls (such as TTY_IO) that assume a source of input and a destination for output.

Each program is invoked from a program segment that the user enters (or otherwise selects) and asks to have executed. Managing the process of accepting user input and presenting user output is important to the user view of the system.

6.1. Structure of User Interactions

Each user command interaction potentially involves many different entities (independent of specific command characteristics):

1. Standard output.
2. Error output.
3. Standard input.
4. Command input: the text of the command itself..
5. Interaction script: the time-interleaved record of the above.

Each of these entities is managed as a text file, while retaining the appearance of normal interactive terminal operation where desirable. In the normal case, the user command window reports the current state of the interaction script. Thus, the Ada statements forming the command sequence appear, followed by the input and program output, just as in a traditional terminal session.

6.2. Standard Input

The appearance of interaction in a file-oriented input system is accomplished by providing two different control cursors into the file, one for program input and one for user editing. In the "normal" interactive case, the user cursor is at the end of the file and the program cursor is not far behind. The most immediately apparent advantage being that the user has the facilities of the editor with which to modify the input. Optional line (or other granularity) interlocks will be provided to make sure that the user is pleased with what has been typed before the program reads it.

One characteristic of this approach is the assumption that user input will be subject to interpretation for the detection of editing command sequences. This places some constraints on underlying application programs, but these are normally only exercised by editor-paradigm programs that we would expect to use the provided facilities. It is also true that the user can avail himself of the key-mapping facilities to obtain the effect of direct access to user keystrokes, including the usurpation of system control function keys.

The input system shares the characteristic of traditional systems that typeahead is consumed by successive input requests, so the user can type Ada statements while the statements in the previous command are still running, but without the Ada completion facilities, etc. Alternatively, the user can request a new command window and enter commands in it completely asynchronous of already running tasks. In this circumstance, the complete Ada program development assistance is available.

When a task "runs out" of input data, it will simply wait for input until the user provides input to that window. If multiple user tasks try to read from the same window, the input task will serialize them, but the interleaving of input characters is likely to be as unsatisfactory as it is on other systems. When data is requested from the terminal, the user can be notified of the need for input, and potentially its type, by the standard highlighted prompting mechanism.

6.3. Standard Output

Normal program output is written to the standard output file and to the interaction script. Normally, the interaction script is mapped onto an active window, but it is possible to focus directly on the output and not have the interaction script visible (though this has implications for the ability to provide input). In fact, the user can direct that any file be similarly followed on the screen or not be visible -- the standard options are in no way special.

Since user programs do not write to the terminal, but to a managed window, random text should never appear in the middle of a window (except for terminal glitches, etc.).

The default output characteristics will make it easy to segregate output from independently initiated tasks. Multiple user tasks addressing the same window will have their output serialized by the output task, but beyond that the user must still handle the problem of unsynchronized output.

6.4. Script Management

Immediate feedback of typed characters is very useful in reassuring the user that he has typed what he thinks he has. Seeing the characters in the context in which they are actually accepted is also useful in confirming the appropriateness of what was typed. To accomplish both of these

objectives, the interaction script is created by appending user input and inserting program output. As the input is consumed, the output cursor moves past, accomplishing the appearance of inserting the input data where it is used.

Similar provision can be made for characters that are not to be echoed. To a first approximation, this can be handled by setting status information in the input package. Unless the echo characteristic is altered before a character entered, it will initially (not) appear when it shouldn't (should). The editing paradigm is particularly useful here, since it is possible to have the appearance correct by the time a particular section of the display stabilizes; for example, a password entered before echoing is turned off will go ahead and print, but will be erased as it is accepted.

6.5. Controlling Interaction Characteristics

Because input, output, etc. are all files, they can be redirected by substituting user filenames.

Output can be stopped when the available window space is full for ^Q/^S-like functionality.

Would like to have a visual signal as to the progress of program sequential input relative to what has been entered. As a minimum, there must be a way of determining when the available input has been exhausted (read them bits too much and they get tired).

→ a way to throw away output & scan

Chapter 7 through?
Help

The purpose of a help system is to provide the user with whatever information will make it possible for him to continue to be productive. Levy's Law of the Lost Layman [Byte, August 1983] gives the following five types of questions the user is likely to have for Ariadne.

1. What does X [do to Y]?
2. How do I do X?
3. What does X mean?
4. What state am I in?
5. What exactly did I do wrong?

how ?

A number of problems face any system that attempts to answer these questions.

1. How does the user ask for help?
2. How does the system detect which form of help the user is requesting?
3. How does the user control the amount of help and its presentation?
4. How is the help facility made available througout the system

without requiring each command to be planned around the help facility?
5. How is help text provided, encouraged and/or kept up-to-date?

It would be nice if there were something here that provided criteria for judging facilities.

Before proceeding to the hard part, designing a help interface, we will start by taking cheap shots at what has been done.

7.1. Available Systems

There are a large number of systems that provide some sort of help to the user. I will draw examples primarily from Tops-20 and Unix because they are widely known and typical of what is available. More comprehensive facilities (for instance, Info in Tops-20 Emacs) are available, but are more difficult to characterise in short form.

7.1.1. What does X [do to Y]?

This facility is provided by Tops-20 in the form of the Help command and in Unix by the "print manual entry" command. Both system simply print a pre-prepared description of the command at whatever level of detail the author thought appropriate. Tops-20 favors a brief description, one the order of a single screen, that references the complete documentation. The Unix manual entry is the complete documentation.

Neither system provides much in the way of detail control or guidance in how to find out what is wanted without reading a large quantity of documentation. Note that this is the natural form of help from the implementor's point of view, since the question and answer are organized along the same lines as the program. This should not lead to the conclusion that this level of help is uniformly well-done.

7.1.2. How do I do X?

This question is addressed by apropos commands. The user thinks of a word or a set of words that seems to him to be related to what he wants to do. Typically, the system looks in a particular place for the word and reports what it finds. Step 1 is then used to find out how to use X. Users frequently prove inadequate to the task of choosing words that the system understands and are sufficiently specific to keep the list of possibilities short enough to read. This seems to be related to the tendency to search for keywords only in command titles (greatly reducing the number of terms related to a particular command) and the lack of feedback mechanisms as to the words the user chooses when searching for something.

7.1.3. What does X mean?

Not well addressed by most systems. The apropos command may be of some use in explaining terms or messages, but is usually oriented toward commands

rather than status. It would seem reasonable to require that all major terms in system messages have some help associated with them.

7.1.4. What state am I in?

Probably the most heartfelt of the questions asked by the lost user (frequently to a person since the system can't help). A trivial version of this is embodied in the ? facility of Tops-20. The ? doesn't tell the user much about his global state, but does provide considerable information about what options are available if he wants to continue entering the current command. Note that ? seldom gives any information about what the various options do, just what the various arguments can be. This is a weak form of what is available using strong typing -- most of the information provided is the parameter type and its possible values if it is an enumeration type.

7.1.5. What about that statement isn't legal?

This isn't all that much of a problem on conventional systems and it still isn't handled very well. The typical response is to print error messages and assume that the user will understand them or have a manual to read. Given the admitted complexity of Ada semantics, it is imperative that we have good ways of explicating errors and providing help understanding the underlying concepts. Help with understanding how Ada works is crucial to the user getting help with most of the rest of the system.

7.2. Integrating Help into Ariadne

add a prec to each function?

A number of the Ariadne features directly support the user in ways approximating a help feature.

1. Syntactic/semantic completion provides most of the ? facility.

2. Show the definition of X answers some of the questions about X and its role.

3. Editor-supported detail control provides a uniform mechanism for varying the detail depending on the particular purpose.

4. The ability to move from one window to another without loss of state makes it possible to ask questions without disturbing the entry that prompted the question.

Together, these features are the beginning of a Tops-20-level help facility, but more is needed.

7.2.1. Wishlist

To be useful, help information should ideally have the following characteristics:

1. Convenience.

   To the extent that help is easily obtained and reliable, the user will be encouraged to use it. To the extent that it is hard to use and of inconsistent quality, the user will only try to use it if truly lost.

   We should be able to bind help to a single key and maintain that link at all times. We should probably choose a sequence on a function key instead of a single character.

2. Self-explanatory.

   There is nothing worse than a help system that is entered easily, hard to understand and hard to leave.

   The conventions of our help system should be the same as those for the rest of the system, which will make it easier for the user to use help once he masters the basic editor interactions. On the other hand, it is not possible to assume knowledge when the user is asking for help.

3. Ability to extract context.

   Whatever information about the user's state and probable needs the help system can use to tailor the initial advice is likely to be well received by the user. For instance, a common time to ask for help is when an error message is incomprehensible. The help system should be able to detect this and answer accordingly.

4. Appropriate level of detail under user control.

   Users of widely variant ability and experience will end up asking for help from the same source. The novice should be able to find out what is needed for simple uses quickly and the the expert should be able to find out the details required for his particular purpose without having to read a primer on the particular topic.

   This can be assisted by structured presentation and detail control, but care must still be exercised in writing and organizing the text. It would be useful if the user could set default, and/or the system could remember some of the salient characteristics of previous encounters. On the other hand, the goal is to help the user, not do research in AI.

5. Currency.

   Comments in programs quite frequently get out of date because the program will run even if the comments are wrong. Any documentation that seems to be separated from the program is even more likely to get out of date.

The apparent path from the defining occurrence to the help text must be made short. It will also be important to capture as much information as possible about what questions the user asks and doesn't get answers to along with what entry he finally settled on. It would be useful to know the difference between typos and misunderstandings, and the difference between help sessions that provide the desired assistance and those that end with the user punching out the terminal.

HELP - proc owner will ever object auto update?
↳ topic so like stuff