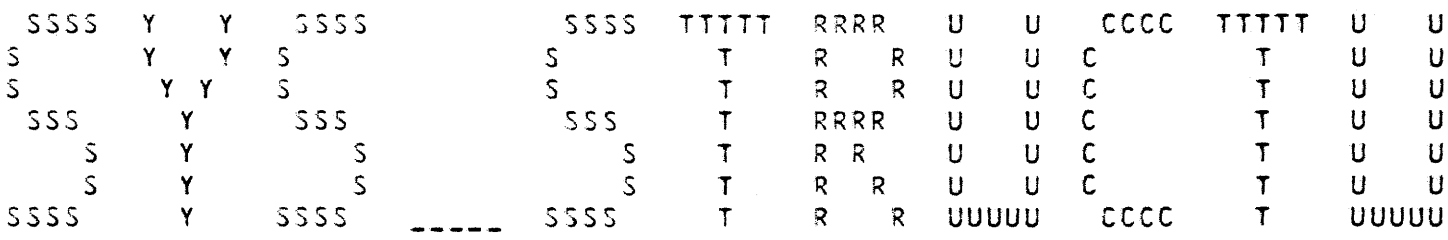

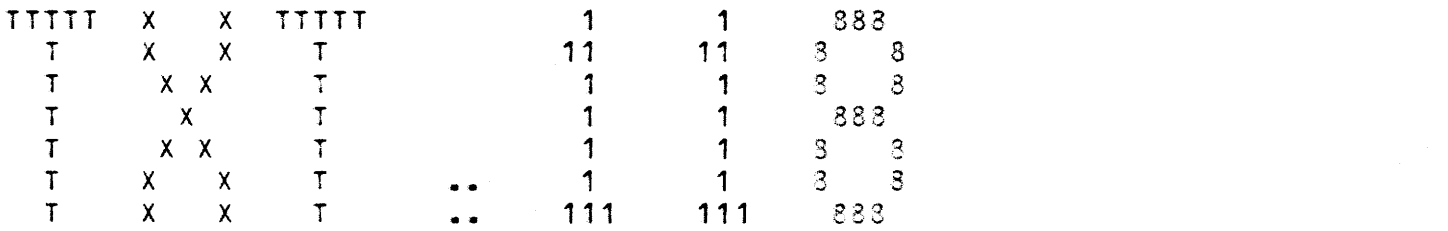

START Job SYS_ST Req #156 for EGB Date 28-Sep-82 22:06:06 Monitor: Rational
File RM:<RPE.DOC>SYS_STRUCTURE.TXT.118, created: 14-Sep-82 22:25:51
printed: 28-Sep-82 22:06:07
Job parameters: Request created:28-Sep-82 22:03:20 Page limit:36 Forms:NORMAL
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:

1. INTRODUCTION

1.1 Purpose

This note outlines the design structure and major components of the Rational Programming Environment.

The initial release of the Rational Programming Environment does not attempt to provide an exhaustive set of functions. Rather, it provides basic functions and a foundation for extension. Thus we are concerned with establishing the structuring principles, basic mechanisms and information representation techniques which can be used throughout the system. In addition, we are interested in decomposing the system into work units and defining interfaces to facilitate near-term development efforts.

1.2 Design Goals (optional reading)

The primary goal is to provide an efficient and reliable implementation of the functions defined for the Rational Programming Environment. Functional requirements and user interface considerations which determine the design are documented elsewhere (RM, etc.).

Given the scope of the environment, a major design goal has been to structure the design around a set of basic ideas which allow one to reason about the behavior of the system. The best software engineering practices are applied to manage the complexity of developing, testing and integrating the Rational Programming Environment and to assure a reliable and robust product.

The long expected life of components of the environment and the fact that we are providing a new and immature technology imply that maintainability, modifiability and testability are a major design goals. Major changes can be expected throughout the product life cycle and the system structure must accommodate incremental changes gracefully.

Maximizing technical and marketing flexibility requires that the system be as modular as possible. It may be desirable to bundle software in different configurations, move portions of the software to distributed system components (intelligent terminals, etc.) and reconfigure specific installations rapidly.

Extensibility is a major design goal driven both by marketing and technical considerations. The product life cycle is viewed as beginning with an initial core environment which is then continually extended to provide increasing function and improved user productivity.

1.3 System Structure

This section briefly outlines the overall structure of the system. Later sections will provide rationale and additional detail. Figure 1 provides a simplified view of the static structure of the system. The system consists of a set of extended abstract types, a set of common system facilities, and an editor subsystem.

Each extended abstract type provides facilities for the construction, manipulation, access and storage of a class of objects or set of related classes of objects. The extended abstract type embodies considerable knowledge about the kinds of objects it manages and the tools available for manipulating those objects.

The common facilities provide mechanisms for easily and uniformly implementing certain operations which are found in all of the extended abstract types. These facilities make it easier to construct extended abstract types and encourage a consistent approach to providing version control, access control, etc. It may turn out that (as the design evolves) many of these common system facilities will end up as (lower-level) extended abstract types.

The editor subsystem provides a core editor, a generic object editor, a generic command object editor (which is very similar to the generic object editor), an image data structure for core editor to object editor communication, and the session command object editor. The core editor provides the direct user interface and is responsible for terminal and window management, keyboard mapping, basic text editing operations, detail control, and overall session management. Each extended abstract type instantiates the object editor package for structured manipulation of objects of the type. Thus an object editor is defined in terms of exaclty one extended abstract type; although, an extended abstract type may itself involve the composition of several types. The object editor in turn may instantiate a command object editor for executing extended commands on objects of the type. The session command object editor instantianted in the core editor provides an environment for executing session-level commands.

Figure 2 illustrates a slightly different view of the system, which reflects some of the dynamic structure of the system. We see that at runtime the system consists of the set of extended abstract types and a number of sessions concurrently manipulating the system state represented by the various extended abstract types. Each session consists of a single core editor task, one or more session command object editor tasks, and a number of object editor tasks (at most one per type). Each object editor may in turn have spawned a number of command object editors (at most one per window of that type), each of which may have spawned another command object editor (and so on recursively for command object editors).

While the system structure encourages extension by adding extended abstract types and object editors, there are a number of basic types and editors which will be included in the initial product. These are illustrated in Figure 3, which can be considered a refinement of Figure 1. There are seven extended abstract types in Figure 3 (R1000, user, text, mail, diana, elaborated context and Ada data structure), each providing an object and command editor for the corresponding type.

1.4 Scope

Section 2 describes the structure and operation of the editing subsystem. Section 3 discusses the basic set of extended abstract types and the object and command editors which go along with each type. Section 4 discusses a number of system facilities which are shared by the various extended abstract types.

2. EDITING SUBSYSTEM

2.1 Visible Structure

The visible (static) structure of the editor can be seen in Figure 4 (<rpe.doc>editor.ada) which shows the Ada interface between the editor and the rest of the system. This interface is designed to encapsulate all core editor to object editor protocols and specify the operations which an extended abstract type must provide when instantiating an object editor.

2.1.1. Image. The image package in the editor interface describes all of the data structures used for communication between the object editors and the core editor. Basically, image.pointer provides a displayable representation for structured objects of any type. This representation is used for passing input from the core editor to the object editor, and for passing display information from the object editor to the core editor. Image.Data consists of two parts a template and a list of object_references which provide the actual parameters for the template. The template is basically a two dimensional text structure with stubs indicating where parameters are to be inserted. The actual parameters are either displayable images (represented by and Image.Pointer) or object ids (Image.Object_Id). Object ids provide a handle for associating display information with underlying objects. The protocol for naming objects is discussed below.

2.1.2 Core Editor. While the core editor is a large program, implementing all of the direct user interface, it presents a relatively simple interface to the rest of the system. The Core_Editor package exports only two types, name and port. Creating a core editor returns a Core_Editor.Name which is the handle for the editor instance. The only operations on names (besides the create operation) are the delete operation (to delete a core editor) and the open operation which returns a Core_Editor.Port. A port is the communication link between an object editor and the core editor. Given a port, there are operations for an object editor to get a command, read back changes made by the user, and update the display. All of these are with respect to an object id. The object ids are assigned by the object editor. The object editor is responsible for maintaining the mapping between between object ids and the underlying objects. The core editor mainains the mapping between object ids and display related data structures.

2.1.3 Object Editors.

The object editor generic is instantiated once per extended abstract type. The generic implements (and encapsulates) a specific protocol for object editor interaction with the core editor. The get_command, readback and replace operations on command_editor ports are used only by the body of the object editor generic (and the command object editor generic), so the core editor "knows" that all of its users follow the expected discipline. For example, after doing a get command and getting a readback_object command, the object editor will read back the object and then replace it.

The generic also specifies (in the list of generic formals) the set of tools which must be provided by the extended abstract type to support structured editing. Besides the object type (and an associated list type and operations), the extended abstract type must provide tools for unparsing, parsing, editing and accessing objects. In addition, the extended abstract type may provide a short circuit command table, which identifies extended commands that can be invoked directly (rather than being evaluated by tools in the elaborated context abstract type, as discussed below). The short circuit command information is also used for command completion.

2.2 Internal Structure

In addition to bodies for the core editor, object editor generic, and the image package, the body of the editor (fig 5 -- <rpe.doc>editor_body.ada) contains a command object editor generic and an instantiation

2.3 Runtime Organization

2.3 Core Editor Design

2.4 Object Editor Design

2.5 Command Object Editor Design

3. EXTENDED ABSTRACT TYPES

Concepts

The notion of an extended abstract type is introduced to provide a framework for organizing system objects and for reasoning about the behavior of system components. Figure 4 provides a pictoral representation of an extended abstract type. An extended abstract type consists of one or more related abstract types, an extensible set of tools built upon the abstract types, and storage facilities (objects of an extended abstract types are by nature long-lived).

The abstract types that make up an extended abstract type are fairly conventional. An abstract type is generally designed such that the set of operations satisfy some completeness criteria (that is, the operations are sufficient to perform all interesting manipulations of objects of the abstract type), while also reflecting the relevant abstract properties of the type being defined. The set of operations is generally designed to provide a consistent level of abstraction, rather than mixing primitive operations and very high level operations. While necessarily minimal, the set of operations is usually kept reasonably small so that the abstraction can be easily maintain and understood.

A tool is any package or subprogram built on top of a set of abstract types to provide some higher level function. For example, the diana extended abstract data type includes the parser, simplifier, pretty printer, semantic analyzer, data flow analyzers, code generators, and many other tools. Tools may also be much smaller and simpler; for instance, small subroutines which evaluate some simple predicate on a diana tree.

While the tools associated with an extended abstract type are primarily concerned with the types contained in the extended abstract type, the tools may use types from other extended abstract types. For example, there may be a version of the pretty printer as part of the diana extended abstract type which uses a text type from the text extended abstract type. Thus extended abstract types can reference each other, in the same way that a pair of simple abstract types may contain mutual references.

There are certain operations that must be provided for every long-lived system entity. The extended abstract type paradigm provides a template, specifying the minimal interfaces and defining conventions which allow consistent interaction between different system components. The standard interface includes tools for editing and for controlling (and synchronizing) access to objects. Figure 5 outlines the interface to the prototypical extended abstract type.

R1000 extended abstract type

The interface between the environment and the machine is provided by the R1000 extended abstract type, which encapsulates the architecture and operating system. The R1000 extended abstract type provides interfaces for code generation, elaboration, resource management, debugging, and system performance monitoring and control.

Diana extended abstract type

The Diana extended abstract type plays a central role in the Rational Programming Environment. All objects in the system a have a source representation in Ada, and Diana provides the internal representation for all Ada. The Diana extended abstract type consists of tne Diana abstract type, facilities for efficient storage and manipulation of Diana, protocols for synchronizing access to portions of the Diana tree, mechanisms for automatically invoking tools when changes are made to the Diana tree, and tools for parsing, pretty printing, semanticizing, and modifying diana trees. The compiler itself is part of the Diana extended abstract type.

Elaborated context extended abstract type

The elaborated context extended abstract type is built upon the diana and R1000 extended abstract types. This type provides the facilities for actually manipulating the runtime representation, including all elaborated objects. Using diana and the R1000, the elaborated context extended abstract type provides facilities for installing and withdrawing objects, executing Ada in context, and performing the "eval" function.

Data Structure extended abstract type

The data structure extended abstract type is largely a restricted form of the elaborated context extended abstract type. It provides symbolic (diana) access to the runtime representation of data structures. It provides tools and editor support for creating, manipulating and displaying Ada data structures.

User extended abstract type

The user extended abstract type manages basic user account information including passwords and any other user authentication information.

Text extended abstract type

The text extended abstract type provides operations for structured text editing and tools for manipulation and storage of text. (RM 7.1)

Mail extended abstract type

The mail extended abstract type is built upon the text extended abstract type and provides facilities for creating, sending and recieving mail. (RM 7.3)

4. COMMON SYSTEM FACILITIES

General

Help

Access Control

Automatic Tool Invocation

Version Control