```
 OOO    V   V  EEEEE  RRRR   V   V   III   EEEEE  W   W
O   O   V   V  E      R   R  V   V    I    E      W   W
O   O   V   V  E      R   R  V   V    I    E      W   W
O   O   V   V  EEEE   RRRR   V   V    I    EEEE   W   W
O   O   V   V  E      R R    V   V    I    E      W W W
O   O    V V   E      R  R    V V     I    E      WW WW
 OOO      V    EEEEE  R   R    V     III   EEEEE  W   W



        1
       11
        1
        1
        1
  ..    1
  ..   111



*START* Job OVERVI Req #165 for EGB    Date 28-Sep-82 22:05:35 Monitor: Rational
File RM:<RPE.DOC>OVERVIEW..1, created: 11-Mar-82 14:25:17
        printed: 28-Sep-82 22:05:35
Job parameters: Request created:28-Sep-82 22:05:00   Page limit:18   Forms:NORMA
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:
```

Apse Overview

This Apse overview describes the Aspe design on four different levels of abstraction:

Manager's view

A manager is interested in those aspects of the Apse that potentially influence his buy decision. Important issues are sophistication of the environment, the effect on productivity, job satisfaction, switchover costs.

User's view

The user needs a simple, understandable model of the structure and operation of the system. The model should be capable of explaining and guiding the users routine operations.

Advanced programmer's view

An advanced user will have a more detailed model of the apse. In particular (s)he will be knowledgeable about backup and crash recovery and know about integration of new tools.

System programmer's view

The system programmer will have an intimate knowledge of the working of the Apse to allow sophisticated modifications to the system.

Manager's view

The R1000 APSE is a state of the art program development system designed to mitigate the software crisis, reduce software life cycle costs, and improve software/system reliability.

The environment consistent uses Ada on all levels of interaction. This use of Ada concepts throughout the design results in an APSE that is understandable and can be explained in terms of Ada semantics. Only very few additional concepts are required.

Apse provides a mechanism for tight but expandable integration of programming tools and knowledge about the use of these tools. This knowledge allows automatic application of tools and automatic enforcement of programming standards.

The human interface of the Apse is designed to support a wide spectrum of users ranging from novice to experts.

User's view

Context_of_a_user

A user in the system may be a real person or a project. Each user has an ID, unique across the whole system. For each user there is a default package in the system (the user package). User packages are associated with accounting, password, and resource constraints. There may be user packages without an associated user. A user may access several user packages if the appropriate password is provided.

The whole R1000 programming system is one big package in which user packages, project packages, utility packages are nested. Ada visibility rules provide the basic mechanism for protection. The position of the user package within the system determines the which objects are accessible to the user.

A user logs onto the system by specifying his name (known to the system) together with the password of his default user package. The user will get "access" to his user package. A user may acquire and release access to additional user packages.

Having access to a user package means that the system editor provides a window into this and all enclosed packages. The user may inspect and modify all objects within the user package as detailed below. The user has read only access to all entities visible inside the user package.

Operations

All user data are stored as values of variables inside the user package. All user programs are subunits of the user package; they are positioned in the appropriate environment.

Subsequently we distinguish the "program text" from a "program instance". (Note that program text is not stored in text but rather in tree form.) A program instance refers to the collection of all stacks associated with a package and task. The user may perform the following operations on program text and instances.

1) A semantically consistent (in itself and with its context) program text may be elaborated; i.e. a corresponding program instance will be created in the appropriate context and is accessible to other users.
2) A program instance which is not statically named can be deleted, i.e. it is reduced to program text.
3) Program text without associated program instance may be changed arbitraryly (syntax directed editor enforces correct syntactic structure).
4) Ada statements and expressions can be typed by the user; they are executed/evaluated in the innermost enclosing program instance.

A user program may call the debugger. A user may "accept" a debug request of on of his programs. Accepting such a request will have the same effect as accessing the scope in which the debug call originated. All normal editing functions are available for debugging purposes.

System_backup

The entities of program and data saved on backup storage are Ada program units and collections. For these entities the system provides the operations "open" and "close". An entity is opened automatically as soon as a component of this entity is changed. Whenever a page of an open entity is written onto disk it will not overwrite the old copy (successive writes of the same page will overwrite each other).

A sysop "close" is provided to close an open entity. The close operation will suspend execution of the entity (temporaryly); it will write all changed pages associated with the entity on disk; after successful write, the old versions of the all new pages are deleted. The close operation leaves an entity in a consistent state.

If a system failure occurs a saved entity may be restarted/recovered from the consistent (old) data on disk.

Tool_integration

Tools area Ada programs; they have to conform to the organization of the data to which they apply. A tool-building tools will allow to add new tools and knowledge to the system.

Invocation of tools can be effected in various ways. Tools may simply be called explicitely by the user. More importantly, tools may be invoked automatically based on some knowledge the system has on the use of this tool.

Knowledge about tools is embedded with the data manipulated by tools. Application of a tools in a particular situation can be enforced or suggested; this may be controlled on different levels: system-wide, within a project, for an individual user, for an individual data object. For example, a text document may be defined to be "spelling corrected". The object will know that for each altered part the spelling correction tool has to be invoked.

System programmer's view

Representation_of_program_text

The diana tree representing all programs on the R1000 is broken up into small pieces which are stored (and recovered) separately. The program tree for a compilation unit is stored with a task of type unit. Each instance of task type unit will store the tree for a unit internally in whatever form it wants. Tools that want to operated on the stored unit will make an entry call and obtain a private copy of a diana tree. After completion the tool can update the stored version with another entry call.

The task unit incorporates all knowledge about tools that have to be applied automatically to altered units. E.g. task unit will call the compiler whenever necessary (similarly version control tools etc).

If unit calls a compiler (or any other tool if required) it will pass
- a diana representation of the unit and
- a diana representation of all visible declarations (the context).

the latter is a "readonly" version of diana, i.e. we are guaranteed that the compiler (tool) cannot change the context.

Making the context read only allows us to share context among all unit tasks. The diana form of the context is strictly redundant information which can be reconstructed by the unit tasks at any time. The system will age the diana representation of the context and delete infrequently used parts of it.

Unit will abort certain running tools (compiler) if the stored program unit is being updated.

Representation_of_program_instances

The representation of running programs is defined by the architecture. For the systems programmer this representation is accessible via the predefined system package "program_interface". This package provides the following operations for a control stack (program entity).

- elaborate and add a new declaration on this stack
- execute a statement on this stack
- evaluate an expression on this stack (need to pass back the result)
- extend the imports list of this package
- determine the stack name of the dynamic predecessor
- modification of code segments

System_Information

System programmers have access to miscellaneous system information. A "User data base" provides information about valid user names, their associated (default) user packages, passwords for each user package as well as resource limitation for each package and usage of resources.