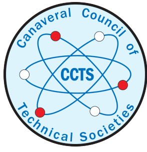


Apr 1st, 8:00 AM

Rule-Based Configuration Control Mechanisms

Brian G. Sayrs
Lockheed Software Technology Center, 96-01,30E 2100 East St. Elmo Road, Austin, Texas 78744 512/ 448-9751

Joseph C. Ross
Lockheed Software Technology Center, 96-01,30E 2100 East St. Elmo Road, Austin, Texas 78744 512/ 448-9751

Follow this and additional works at: https://commons.erau.edu/space-congress-proceedings

Scholarly Commons Citation

Sayrs, Brian G. and Ross, Joseph C., "Rule-Based Configuration Control Mechanisms" (1988). *The Space Congress® Proceedings*. 1.
https://commons.erau.edu/space-congress-proceedings/proceedings-1988-25th/session-9/1

This Event is brought to you for free and open access by the Conferences at Scholarly Commons. It has been accepted for inclusion in The Space Congress® Proceedings by an authorized administrator of Scholarly Commons. For more information, please contact commons@erau.edu.

EMBRY-RIDDLE Aeronautical University
SCHOLARLY COMMONS

Rule-Based Configuration Control Mechanisms

By

Brian G. Sayrs and Joseph C. Ross

Lockheed Software Technology Center
2100 East St. Elmo Road, 96-01, 30E
Austin, Texas 78744
512/448-9751

Rule-Based Configuration Control Mechanisms

Brian G. Sayrs and Joseph C. Ross

Lockheed Software Technology Center, 96-01, 30E
2100 East St. Elmo Road, Austin, Texas 78744
512/448-9751

Abstract

This paper explores the use of rule-based techniques to manage reusable software libraries. In particular, we examine the properties of partially instantiated Ada generic packages and present an object-based view of a particular collection of reusable Ada generic packages. We argue that because types are the primary mechanism for structuring programs in Ada, our ability to organize and manage large Ada software systems is commensurate with the software development environment's support for organizing and managing types. We have assembled a testbed environment for Evolutionary Software Associates' Workshop object management software. The testbed enables us to evaluate the Workshop system and demonstrate the feasibility of the evolutionary approach to the development of large Ada systems. The evolutionary approach to software engineering seeks to integrate tools that support software development with tools that support software maintenance. Initially the Workshop is being used in conjunction with a LISP-based development environment, but it is, in principle, language and platform independent. We are currently experimenting with rules and class definitions for structuring information about the products and processes in software design and development. We are designing and implementing control mechanisms that can be automatically activated when the developers engage in certain events. An inference mechanism determines which rules can fire and in some cases will cause transformations to occur automatically. The developers interact with the environment through a Software Spreadsheet™ (Clemm 1987) which actively indicates the status of software objects.

The software development concepts embodied in the Workshop represent an emerging and promising software development methodology. In this paper we discuss applying these concepts to the development of reusable libraries of Ada software components.

Key Words and Phrases: Ada, partially instantiated generic packages, Workshop, Software Spreadsheet, rule-based, reusable software libraries.

The views and opinions expressed in this paper are those of the authors and should not be construed to be those of the Lockheed Software Technology Center, Evolutionary Software Associates, or the Department of Defense.

Software Spreadsheet™ is a trademark of Evolutionary Software Associates.
Ada ® is a registered trademark of the U.S. Government, Ada Joint Program Office.

Rule-Based Configuration Control Mechanisms

Brian G. Sayrs and Joseph C. Ross

Lockheed Software Technology Center, 96-01, 30E
2100 East St. Elmo Road, Austin, Texas 78744-1016
512/448-9751

1.0 INTRODUCTION

Software *malleability* is one of the key reasons for allocating system requirements to software as opposed to hardware. As a result, software engineers are continually challenged to develop and maintain software that reflects changes in requirements, design, specifications, and operational conditions. Software *stability* is required because software is a product that is managed, tested, accepted, delivered, and controlled. The opposing goals of malleability and stability in a software system provide the basis for software engineering decisions that trade off aspects of one for aspects of the other. Software malleability is realized through the processes of program development and maintenance. These activities involve creating new code and making changes to existing code and are supported by tools such as editors, debuggers, and execution environments. Software stability is realized through configuration control mechanisms that enforce software engineering standards and practices. Conventional software development environments provide tools that independently support configuration control and program development. This paper describes a new approach that integrates support for creating, changing, and controlling software at the tool level.

Our near-term goal is to demonstrate the feasibility of applying knowledge-based techniques to help manage multiple versions of Ada software components in a software development environment. The utility of reusable software libraries would be significantly enhanced by tools that help specialize components for particular applications and tools that help manage multiple versions of components. Automated techniques for component specialization is an active area of computer science research and beyond the scope of this paper. General Ada software libraries are commercially available (Booch 1987) and specialized component libraries are currently under development at Lockheed and elsewhere. Current software environments deal primarily with code artifacts. Since library components are rarely useful without some alteration and customization, future environments that incorporate reusable software libraries should also provide tools for capturing the rationale and techniques for specializing components. The development of knowledge-based techniques for managing the products and processes in a software engineering environment provides a basis for research and development of reusable software libraries.

The Ada programming language (Ichbiah 1980) was intended to provide language-level support for software engineering practices that ease the burden of controlling software changes. It was recognized early on that there would be new requirements for Ada programming support environments (Buxton 1980), but much more attention has been given to specific tools, such as compilers and editors. Research on Ada compilation led to the definition of Diana, a Descriptive Intermediate Attributed Notation for Ada (Goos 1981), which provides a notation for representing Ada programs as data and is especially suitable for passing data (Ada programs) through the various phases of compilation. Diana is written in a more general data description language, the Interface Description Language (IDL) (Nestor 1981), developed at Carnegie-Mellon University. The SoftLab project at the University of North Carolina at Chapel Hill (UNC) is using IDL as the basis

---

The views and opinions expressed in this paper are those of the authors and should not be construed to be those of the Lockheed Software Technology Center, Evolutionary Software Associates, or the Department of Defense.

Software Spreadsheet™ is a trademark of Evolutionary Software Associates.
Ada ® is a registered trademark of the U.S. Government, Ada Joint Program Office.

for integrating tools in a software engineering environment (Snodgrass 1986). We are currently working with a copy of the IDL translator, obtained from UNC, and are considering IDL as a representation language for interfaces between reusable library components.

In the following section we describe a new architecture for software development environments that integrates a tool for managing the stability of a software system with tools that support program development. We then describe an approach to building reusable software libraries with Ada, and demonstrate how this approach can be supported by the concepts and techniques embodied in this new architecture.

2.0 WORKSHOP OVERVIEW

The Workshop represents a new approach to software development environment architectures (Balzer 1986). The primary goal of the Workshop system is to provide a tighter coupling between tools that assist in the development, maintenance, and evolution of large software systems. It deviates from conventional environment architectures by providing an intermediate model for software artifacts that insulates the global view, which maintains the stability of a software system as a whole, from local views that take advantage of the malleability of particular software objects. This intermediate model is referred to as a Software Spreadsheet and is analogous to conventional spreadsheets in that it maintains relationships among user defined objects. The Spreadsheet model provides the nexus in the software development environment architecture that connects tools that support (local) changes to software objects with tools that support (global) control of those changes (fig. 1).

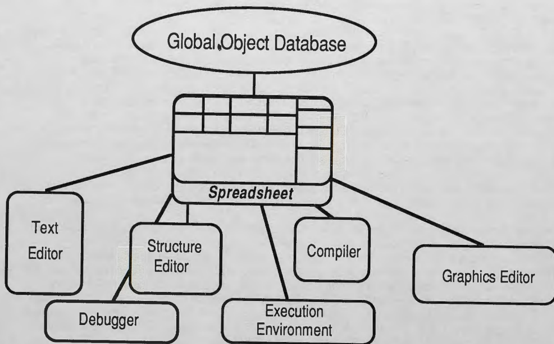


Fig. 1. The Spreadsheet model

Individual software objects are stored on a global data base and made visible to programmers through their Spreadsheets. In this approach, the conventional file system is replaced by a global object data base. All objects are defined as belonging to a particular *class*. A class is a collection of attributes that provides a template for creating and modifying objects. The Spreadsheet provides an

object editor that views objects as collections of value/attribute pairs. The Spreadsheet also provides a Spreadsheet Manager that manages all of the objects in a particular Spreadsheet. The Manager also monitors the relationships the objects in the Spreadsheet are required to satisfy. Object relationships are implemented with rules that coordinate and automate the effects of changes made by the programmer. An underlying inference mechanism is used to "fire" rules and assists in propagating the effects of changes. The power of the Spreadsheet model for software development depends on its ability to be interfaced with existing program development tools, such as compilers, editors, and debuggers.

2.1 Workshop Built-In Facilities

The Workshop provides built-in facilities for modeling software components as objects. These facilities enable the global object data base to coordinate the actions of multiple Spreadsheets (users) and provide data base access for individual users. The Workshop also provides extension facilities that enable the Spreadsheet model to be customized for particular classes of software objects. In the example below, we show an example of augmenting the Spreadsheet model to take advantage of the syntax and semantics of a subset of Ada. We expand upon this idea by showing how the model can be extended further to take advantage of specific project development guidelines and constraints in the application domain. The Software Spreadsheet language is extended by creating new classes for the top level defining forms of a particular programming language. In the example below, we define classes for generic packages. The example demonstrates the clarity of an object-based view of the relationships between partially instantiated generic packages. The Spreadsheet language also allows rules to be defined for automating, coordinating, and validating changes made by the user. We shows how rules can be defined that check the consistency of constraints in the Ada packages against requirements, development standards and practices.

The rest of this paper concerns the definition of a software spreadsheet model for a subset of the Ada programming language and an example of the application of that model to a reusable library of partially instantiated generic packages.

3.0 A SOFTWARE SPREADSHEET MODEL FOR RESUABLE LIBRARIES

A project is currently underway at the Lockheed Software Technology Center (LSTC) for developing a research prototype of a binding between Ada application programs and a commercially available relational data base system which uses the Structured Query Language (SQL). This binding is being implemented by constructing a mapping from the programming language constructs of Ada to the relational model objects manipulated by SQL. A significant idea behind this approach is that the constraints imposed by the relational model and SQL should be mapped into Ada constraints and enforced by the Ada compiler, when possible. Some development guidelines, as we show below, are more suitably represented with a higher level structuring mechanism, such as the Software Spreadsheet.

The actual Ada software being developed consists of a set of reusable generic packages that capture the structure of the relational model and SQL. This model can then be instantiated for a specific application by a reusable library administrator to permit manipulation and inspection of the relational data base by Ada application programs.

3.1 Software Design and Development Criteria for the Ada/SQL Binding

The software developed for this project and the development process itself were influenced by many criteria, among which the following are some of the most significant :

1. The syntax of Ada procedures and datatypes which are provided for use as a data base access language, should be as similar to that of ANSI standard SQL as possible, so as to be intuitively usable by someone familiar with SQL. Furthermore the power and flexibility of the resulting Ada data base language should not be less than that of SQL.

2. The data base access language should be syntactically correct Ada, so as not to require any preprocessing, and so as to permit an Ada compiler and Ada-specific editor to provide early detection of syntactic errors in data base language statements.

3. Semantic constraints on the validity of SQL queries should be mapped to Ada constraints as much as possible, so that an Ada compiler and Ada-specific editor can assist in early detection of semantic errors in data base language statements.

4. Dependencies on particular operating systems or data base systems should be isolated to the lowest level packages, so that the bulk of the interface, and the Ada data base access language are portable to any system with a validated Ada compiler.

5. Specifications for all Ada packages should be developed first, so that the interfaces between them can be agreed upon and determined to be semantically correct by the Ada compiler. The bodies of the packages should only be developed after these interfaces have been established.

6. Documentation should be included as a part of each separate Ada unit. This documentation should describe what function the unit performs, how it performs this function, and the rationale behind any significant implementation decisions.

3.2 Extending the Spreadsheet Model for Ada Packages and Generic Objects

An Ada package is a collection of Ada objects that define an abstract data type and can exist as an independent compilation unit. The commonly-accepted abstract data type methodology is a model of programming that separates the implementation of operations from the specification of operations. A package has a declarative *specification* part, which controls access to any resources being managed by the package and a *body* part, which contains objects which are not visible outside the package. Encapsulation is the process of hiding information and was first suggested by Parnas as a technique for demonstrating the effectiveness of modular decomposition (Parnàs 1971). Thus, packages provide a way to encapsulate the details of the way in which a resource is implemented, while providing an abstract view of the resource to the external environment. While Ada enables a clear separation of specifications and implementations, the rationale for a particular implementation and the relationship between requirements, specifications and implementations is not easily represented within the language itself. The spreadsheet language provides constructs for defining relationships between classes of Ada objects. The spreadsheet model is supported by a mechanism for checking the relationships and propagating changes initiated by programmers. Object classes can be defined for all the top level defining forms in the Ada programming language. For instance, the general structure of an Ada package is as follows:

```
package Package_Name is
   ...declarations describing
      exportable view of resources ...
end Package_Name;

package body Package_Name is
   ...actual implementation of resources,
      hidden from the external environment.
end Package_Name;
```

This structure can be described in the spreadsheet language by defining a class for packages and listing the attributes shared by objects in that class (fig. 2).

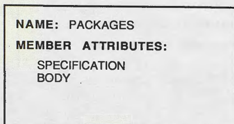


Fig. 2. Definition of a class for packages

Attributes can be declared to be either *required* or *optional*, so that when objects are created, values for particular attributes may or may not be required. As an example, from the standpoint of an Ada compiler, bodies are optional attributes of packages. However, Ada compilers require the specification for a package to be defined before compiling the body. The spreadsheet language also provides constructs for defining rules that can assist in coordinating compilation priorities (fig. 3).

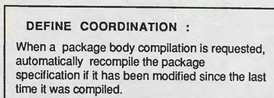


Fig. 3. Definition of rules to assist in coordinating compilation priorities

3.2.1 Generic Classes

The Ada construct for generic packages is a mechanism for specifying package templates. Since they are just templates, they are similar to type definitions. Executable packages are created from generic types by declaring instantiations of the generic templates. Generic packages are thus very useful for defining *classes* of objects, and are necessary in a strongly typed language such as Ada, to define general data structures whose semantics don't depend on the type of the components involved.

The syntax for specifying a generic package is exactly like that for a normal package, except for the addition of a *generic* part which contains generic parameters that can be referenced inside the package. Values for these parameters are provided when instances of the package are declared. These parameters are used to provide information which is necessary to an executable instance of the package, but is not relevant to the abstraction represented by the generic object. The structure of a generic package is as follows:

```
generic
   ...generic parameters...
package Generic_Package_Name is
   ...generic package specification ...
end Package_Name;

package body Generic_Package_Name is
   ...generic package body...
end Generic_Package_Name;
```

The corresponding class for generic packages can be defined as shown in figure 4.

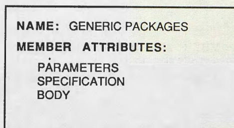


Fig. 4. Definition of a class for generic packages

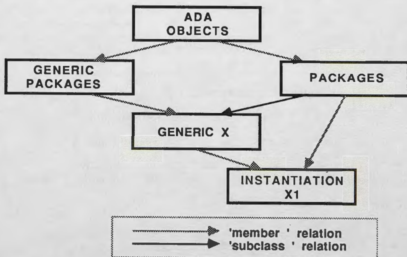


Fig. 5. Set of relations for a subest of the Ada programming language

Member and subclass relations can be defined, maintained and checked by the Workshop during program development. Figure 5 shows a set of relations for a subset of the Ada programming language. *Subclass* relations are represented as a black arrows, while *member* relations are gray ones. Thus, both Generic Packages and Packages are defined to be members of a class called Ada Objects, which would also have many other members corresponding to other Ada constructs. A particular generic package such as Generic X is a template which can potentially be instantiated in many different ways to create many different packages. This idea is captured in the diagram by showing Generic X to be a class which is a member of Generic Packages, but a subclass of Packages. This subclass is the set of possible instantiations of Generic X, and any particular instantiation, such as Instantiation X, is a member of both Generic X and Packages.

3.3 A Domain-Specific Extension of the Spreadsheet Model

Classes for particular generic packages can be defined that contain application-specific attributes in addition to the attributes inherited by all generic packages. This enables more elaborate structuring of software objects and the creation and maintenance of relationships among objects that go beyond the syntax and semantics of Ada. In the example that follows, we present a brief overview of the relational model upon which the requirements for Ada/SQL binding project are based. We then demonstrate how the development criteria for the Ada/SQL binding project outlined above can be represented and used to provide mechanical assistance during the course of software development, maintenance and evolution.

3.3.1 The Relational Model

A data base system can be viewed at different levels. At the lowest level is the *physical data base*, in which the actual data values are stored on a physical storage device such as a disk. However, the data is typically manipulated using a more abstract representational model of the conceptual *data base*. SQL is a language in which data is viewed according to the *relational model.*

Data base languages, including SQL, consist of a *data definition language* used to define the structure of the conceptual data base, and a *data manipulation language* used to insert, modify and access data in the data base. Thus, the data definition language is used to create a template or *schema* for the data base, while the data manipulation language is used to create, access and modify an instance of it.

In the relational model a schema consists of a collection of domains of values and a collection of relations which group the domains in various ways. A relation is therefore a subset of the Cartesian product of a list of domains, where a domain is a set of values of a certain type. The members of a relation are called *tuples*, which are groupings of values which represent valid instances of the relation. In data base terminology, relations are often called *tables*, in which the subsets of each domain involved in the relation are *columns*, and each row is a tuple. We aggregate the generic packages used to implement schema, tables and columns into a higher level module, called Relational Model Generics (fig. 6). This enables us define rules that apply to all the members of this class. For a particular class of packages, such as Generic Table, we can define a subclass of packages (fig. 7).

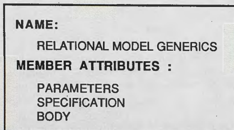


Fig. 6. Relational Model Generics module

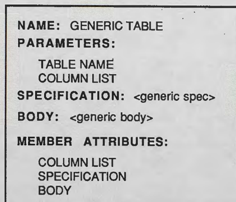


Fig. 7. Generic Table subclass of packages

The class Generic Table of packages is also a member of the class of Relational Model Generics. All classes are members of the system defined class "class" and inherit *name, member attributes,* and other special attributes for printing, graphic display, and so forth. A particular member of the class *generic table* is a package that implements a specific type of table. For instance, a table that

contains information about employees, such as their name, address and phone number, would be an object in the Workshop and appear in the spreadsheet window as shown in figure 8.

```
NAME: EMPLOYEE TABLE
COLUMN LIST:
    NAME
    STREET
    CITY
    STATE
    PHONE
SPECIFICATION: <package spec>
BODY: <pakcage body>
```

Fig. 8. Employee Table

3.3.2 Using Generic Packages for the Ada/SQL Interface

Generic packages are well suited for defining the classes of objects used by the relational model, since they represent general structures which do not depend on the types of components involved.

A schema is collection of tables, regardless of what relations the tables represent, while a table is a collection of columns, regardless of the type of the values in a column. Similarly, a column is a set of values of a certain type, and its "columness" does not depend on the type of values it contains.

However, a particular schema is made up of a particular collection of tables, and different tables might not make sense in that context. Similarly, a particular table represents a particular relationship, and makes sense only between certain sets of values. Furthermore, a column represents a particular domain of values, and only values of that type make sense for that particular column.

Therefore, in the Ada/SQL project, the concept of a relational schema is represented as a generic package. This generic package has as generic parameters, a name for the schema and a list of the tables which the schema contains. Similarly, the concept of a table is a generic package with generic parameters for the name of the table and a list of the columns which a table contains. The concept of a column, on the other hand, is represented as a generic package instantiated with the column name, and various attributes of the domain of the elements of the column, such as the element type.

The declaration of the generic column package is contained within the declaration of the generic table package, which is contained within the declaration of the generic schema package. This nested structures permits the Ada compiler to enforce the constraint that the name of a table within a schema is one of the list of table names used to instantiate the schema. Similarly, the name of a column of a table must be one of the list of column names which were supplied when that particular table instance was declared. This nesting also preserves the relational model semantics that says a table is part of a schema and a column is part of a table. Figures 9–12 we show the package specifications for the relational model generics.

Figure 9 shows the nested, fully generic definitions. To create the corresponding packages, the generics must be instantiated in the following order: (1) schema, (2) table, (3) column.

Figure 10 shows a particular instantiation of Schema. The Company_Schema package declaration results from the following Tables type declaration and Schema generic instantiation:

```
type Tables is (Employee, Finance, Insurance);
package Company_Schema is new Schema (Tables);
```

```
generic
   type Tables is (<>);
package Schema is

   generic
      Table_Name : Tables;
      type Columns is (<>);
   package Table is

      generic
         Column_Name : Columns;
         Column_Type : Sql_Types;
         Column_Width : Integer;
      package Column is

      end Column;

      Undefined_Column,
      Multiply_Defined_Column : exception;

   end Table;

   procedure Build_Schema (Ada_View_File_Name : String);

   Undefined_Table,
   Multiply_Defined_Table : exception;

end Schema;
```

Fig. 9. Fully generic schema

```
package Company_Schema is

   type Tables is (Personnel, Finance, Insurance);

   generic
      Table_Name : Tables;
      type Columns is (<>);
   package Table is

      generic
         Column_Name : Columns;
         Column_Type : Sql_Types;
         Column_Width : Integer;
      package Column is

      end Column;

      Undefined_Column,
      Multiply_Defined_Column : exception;

   end Table;

   procedure Build_Schema (Ada_View_File_Name : String);

   Undefined_Table,
   Multiply_Defined_Table : exception;

end Company_Schema;
```

Fig. 10. Partially instantiated schema, Company_Schema package

Figure 11 shows the Company_Schema package declaration that results from instantiating the Table component of Company_Schema as an Employee_Table:

```
type Columns is (Name, Street, City, State, Phone);
package Employee_Table is new Company_Schema.Table(Employee, Columns);
```

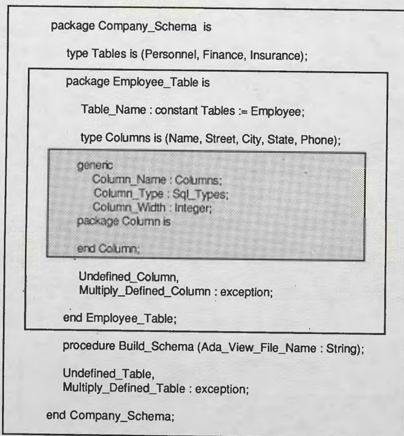


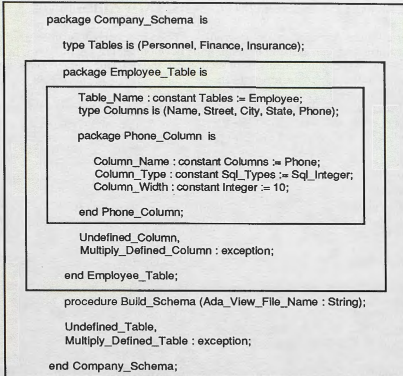
Fig. 11. Partially instantiated schema and table package

Figure 12 shows one complete thread of instantiation for a Company_Schema generic.

To fully instantiate a Company_Schema, all of the Tables and all of the Columns for all of the Tables would have to be instantiated.

Figures 9–12 indicate the code expansion that results from the use of generics but do not provide a clear view of the relationships among the various software objects that are created. This example is certainly not trivial, but one could imagine much more complicated applications. As Large software systems continue to grow and evolve, they become increasingly difficult to manage and work on. The object-based view, shown below, provides a conceptual structure for the Schema, Table, and Column packages.

Figure 13 shows the semantic network representation of the relationships among the software objects as defined with the software spreadsheet language in the Workshop.



```
package Company_Schema  is

    type Tables is (Personnel, Finance, Insurance);

    package Employee_Table is

        Table_Name : constant Tables := Employee;
        type Columns is (Name, Street, City, State, Phone);

        package Phone_Column  is

            Column_Name : constant Columns := Phone;
            Column_Type : constant Sql_Types := Sql_Integer;
            Column_Width : constant Integer := 10;

        end Phone_Column;

        Undefined_Column,
        Multiply_Defined_Column : exception;

    end Employee_Table;

    procedure Build_Schema (Ada_View_File_Name : String);

    Undefined_Table,
    Multiply_Defined_Table : exception;

end Company_Schema;
```

Fig. 12. A single thread of instantiation

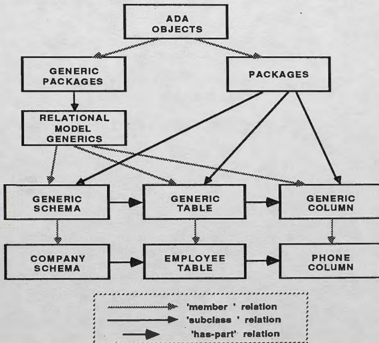


Fig. 13. Object-based view of software structure

4.0 RULES FOR IMPLEMENTING CONSISTENCY GUIDELINES

The criteria for the Ada application program described in section 3.1 can be used to define rules for the Workshop which can assist in managing the development of the program. In this section we will examine some of the requirements from section 3.1 and describe some classes, attributes and rules which can be defined to create an application-specific Ada spreadsheet model.

The first criterion deals with the need for similarity between the syntax of the Ada procedures to be developed and the ANSI-standard SQL syntax. This correspondence can be represented by defining the following classes:

1. SQL_Statements: The set of SQL statement types,whose attributes include Name, Ada_Procedure, and Parameter_List.

2. Ada_SQL_Procedures: The set of Ada procedures that together form an SQL-like interface for an application program and which have attributes for Name, SQL_Statement, and Parameter_List.

3. Ada_Keywords: The set of keywords of the Ada language.

The following constraints can then be specified and checked by the spreadsheet during the course of program development:

1. The value of the Ada_Procedure attribute of SQL_Statements must be specified and must be a member Ada_SQL_Procedures.

2. The value of the SQL_Statement attribute of Ada_SQL_Procedures must be specified and must be a member of SQL_Statements.

3. For every X in SQL_Statements, the value of SQL_Statement of Ada_Procedures of X must be X.

4. For every X in SQL_Statements, the value of Name of X must be the same as the value of Name of Ada_Procedure of X unless Name of X is a member Ada_Keywords.

5. For every X in SQL_Statements, the length of Parameter_List of Ada_Procedure of X must be the same as the length of Parameter_List of X.

Additional constraints can then be specified concerning the ordering and names for parameters, among other things.

5.0 SUMMARY

This example demonstrates that the object-based view of a reusable library of partially instantiated generic Ada packages can be supported by the Software Spreadsheet model. The Spreadsheet language provides constructs for explicitly describing relationships among software objects and the Spreadsheet manager can coordinate and propagate actions initiated by individual programmers. We have also shown how explicit relationships between requirements, specifications, and implementations for Ada packages can be checked with consistency rules. Since this checking is done in the background, the program developer is relieved from some of the detailed bookkeeping chores associated with software development standards and practices.

This work represents an integration of concepts and techniques used to manage software in a fileless environment with an approach to building reusable libraries of partially instantiated generic Ada packages. We thank Bob Balzer and Geoff Clemm of Evolutionary Software Associates and Steve Sherman for their suggestions and encouragement. We also thank Jim Allison, who

originated the nested generic approach to constructing interfaces between Ada and relational data bases, and Susan Mouton, Barbara Wills, Adam Linehan, and Allison Westbrook, who are developing the reusable library for the Ada/SQL project and are building a working prototype.

REFERENCES

Allison, J. L. 1987. *A prototype binding of ANSI-standard SQL to Ada.* Technical proposal in the area of data base management systems to the Department of Defense, Software Technology for Adaptable, Reliable Systems (STARS), Foundation Ada Software for Research in Virtual Interfaces program. Lockheed Software Technology Center, Austin, Texas.

Balzer, R. M. 1986. Living in the next generation operating system. In *Proceedings of the 10th World Computer Congress* (IFIP Congress '86), Dublin, Ireland.

Booch, G. 1987. *Software components with Ada: Structures, tools, and subsystems.* Menlo Park, California: Benjamin/Cummings.

Buxton, J. N. 1980. *Requirements for Ada programming support environments: Stoneman.* DOD report, Office of the Undersecretary of Defense, Pentagon, Washington, D.C.

Clemm, G. M. (technical contact). 1987. *The Software Spreadsheet: A tool for managing the development process.* Technical report, Evolutionary Software Associates, Los Angeles.

Goos, G., and W. A. Wulf, eds. 1981. *Diana reference manual.* CMU-CS-81-101, Computer Science Department, Carnegie-Mellon University, Pittsburgh.

Ichbiah, J. D., B. Krieg-Brueckner, B. A. Wichmann, H. F. Ledgard, J. C. Heliard, J. R. Abrial, J. G. P. Barnes, M. Woodger, O. Roubine, P. L. Hilfinger, and R. Firth. 1980. *Reference manual for the Ada programming language.* Minneapolis: Honeywell, Inc., and Cii-Honeywell Bull.

Nestor, J. R., W. A. Wulf, and D. A. Lamb. 1981. *IDL: Interface Description Language.* CMU-CS-81-139, Carnegie-Mellon University, Computer Science Department, Pittsburgh.

Snodgrass, R. and K. Shannon. 1986. *Supporting flexible and efficient tool integration.* SoftLab Document 25, Computer Science Department, University of North Carolina, Chapel Hill.