Design of a WSRC Repository with an End-User Emphasis

Technical Report for
Westinghouse Savannah River Company (WSRC)
Savannah River Site (SRS)
Aiken, SC 29808

January 14, 1994

John A. Miller
Walter D. Potter
Krys J. Kochut
Sunderraman Krishnan
Bonnie Edwards
Wensheng Zhang
Jayesh Sahasi

Department of Computer Science & AI Programs
415 Graduate Studies Research Center
University of Georgia
Athens, GA 30602-7404

Prepared for the U.S. Department of Energy Under Contract No. DE-AC09-89SR18035

Table of Contents

Executive Summary

The information systems of the 1990's are evolving toward highly multifaceted, yet integrated systems. Database Management Systems (DBMS's) will provide the primary facilities for storing corporate information. In terms of content and functionality, there will be multiple operational databases as well as several special purpose databases. Forward looking organizations such as WSRC, will also have information warehouses. In terms of technology, there will be a collection of DBMS's from multiple vendors based on multiple underlying data models (e.g., Network (ADABAS), Relational (Oracle), and Object-Oriented (ObjectStore)). Furthermore, there will be centralized, distributed, and, before too long, federated databases. Moreover, this is not the whole story, much of the corporate information will be stored in data files. The organization of these is dependent on the Operating System (OS) (of which again there is a multitude, e.g., MVS, AIX, VMS, UNIX, OS/2, MS-DOS, Windows, and Windows NT). Actually, this is only half the story; equally, if not more, important is how to enter, manipulate, display, report, and analyze the corporate information. There exists almost a bewildering number of packages, languages, and tools to either directly provide these capabilities or allow their rapid development by trained application developers. Over time, in-house development of applications will add substantially to the problem of keeping track of everything. As if to make matters worse, at the same time the software side is exploding in complexity and variety, a fundamental shift is also occurring on the hardware side. Corporations are down-sizing (or right-sizing) from their multi-million dollar mainframes to networks of relatively inexpensive Personal Computers and Workstations. Client-Server Architectures will become the standard approach to distributing advanced services (both informational and computational) to the desktop. Still, connectivity to existing mainframes will need to be provided for most large organizations for several years to come.

If this is the situation facing MIS Directors or Chief Information Officers (CIO's), and as mentioned before the information systems of the 1990's are to be more integrated, more user-friendly, and set up to make the ever growing number of end-users (often numbering in the thousands) more self-reliant, then it does not sound very pleasant. Fortunately, technology, which created these problems in the first place, also has some solutions. Some examples are Standards for Graphical User Interfaces, Information Warehouses, Sophisticated Data Dictionaries, and possibly CASE Tools. Unfortunately, none of these provides a comprehensive solution to the problem of integration. However, a very promising emerging technology does. Repository technology's very purpose is to make sense and order out of the complex mess discussed in the previous paragraph. A repository is like a database, but with many value-added capabilities. Its purpose is to keep track of corporate information resources or assets (e.g., databases, data files, and warehouses), and the means by which these resources are used (e.g., applications and productivity tools). Repository technology is capable of organizing the development side of things as well.

One may think of a repository as a logical storehouse of information. This does not mean that it duplicates the role of corporate databases or warehouses. A repository stores meta-data rather than actual data. Meta-data are data about data. This simply means, for example, that rather than storing the employee data (thousands of records), it would store information about the storage of employee data (e.g., there is a table called `Employee` in the database called `Personnel_DB`, which is managed by the DB2 DBMS running on the IBM Mainframe; the table consists of the following columns..., enforces the following constraints..., may be accessed by the following users..., and may be manipulated by the following applications...;). User (end-users, developer, etc.) will consult the repository when they want to find something (a data file, a database table, an application, or even a version of a software module). A repository can take on an even more active role, and actually control the operation of the aggregate information system, controlling how (and who/when) data may be accessed, how applications may be run, or even how versions of software modules may be checked-out for modification.

This report will go into depth about these and related issues. It will focus on the needs of WSRC and, in so doing, will enable the scope of the repository to be reduced somewhat. For example, one goal is to minimize the impact on present application development practices; consequentially, WSRC may not wish to provide version control as a repository service. Another goal is to maximize the self-reliance of end-users, so the repository should be very user-friendly and provide clear descriptions of the elements stored in the repository. The ability to browse the repository and to perform keyword searches should allow end-users to find what they are looking for. Once found, the descriptions within the repository will get them started. The repository will also guide them to more detailed information (e.g., in on-line help, in on-line documentation, or even in off-line printed manuals).

1. Introduction

The middle 1990's appears to be the time when many enterprises will be looking to introduce repositories as a focal point for their information systems. Some organizations are looking for a more complete and comprehensive replacement for their integrated data dictionaries, while others look toward repositories as the basis for CASE tool integration. The focus of this report will be the former, although mention of the latter will be made from time to time. One of the main purposes of such a repository would be to keep track of corporate data processing assets (e.g., databases, files, and software). This will enable end-users to use the repository to search for an application that meets their needs as well as figure out how they can access and run the application. The repository should be able to address the basic what, where, and how questions typical of end-users. (See Section 6.2.1 for a definition of end-user. Briefly, an end-user is a user who is interested in using software, but not developing it.) As WSRC's data processing environment becomes more multi-faceted (with client-server as well as mainframe-based capabilities), it will become increasingly more important for the thousands of end-users to have access to a repository to meet their needs. In addition, the repository will also be important and beneficial to WSRC application developers. (A detailed discussion of this facet of the repository is given in Appendix A).

International Business Machines (IBM) and Digital Equipment Corporation (DEC) have been major players in this arena. (There are also several independent software companies that are now selling repository products; see Appendix D.) They both IBM and DEC see repositories as providing a logically centralized storehouse of corporate meta-data as well as supplying the core for their integrated software development architectures/environments. IBM's repository is called Repository Manager (RM) and runs on top of DB2 under MVS. Newer products are expected out in 1994 which run on top of an object-oriented DBMS under AIX/6000 and OS/2. DEC's repository is called Common Data Dictionary/Repository (CDD/Repository) and runs on top of Rdb under VMS and OSF. Both companies have spent several years developing large, complex information models that indicate the types of information that their repositories should contain. Since some of this is specific to an enterprise, they as well as the majority of repository vendors see extensibility as essential. Extending a core information model is necessary, but it may not be so easy, since for example IBM's current information model contains somewhere around 330 entity types, 1250 relationship types, 1570 attribute types, 4000 policies, and 430 constraints.

The basic objective of repositories is to keep track of all the data-processing assets of an enterprise. Actual data, inputs to and outputs from programs, data files, and databases, would not be stored in the repository, but everything else could be. Hence, the huge size of IBM's information model. Primarily, the repository is to store data about data, or meta-data. Rather than storing database instance information such as John Doe's employee record, typing information indicating the structure and integrity (in the form of constraints) of Employee records is maintained. In addition to the structure of stored data, it is also important to know how data are originated, how they are transformed, and in which forms they can be displayed. In other words, tracking the flow of information is quite useful. This indicates that it is necessary to store information about users, applications, and screens/windows. In addition, since users (directly or indirectly through a tool or application) will be accessing and manipulating data, their authorizations and privileges to access/manipulate data as well as run applications could be stored in the repository.

In summary, from one point of view repositories provide the glue for CASE tool integration, while from the other they are the next-generation replacement for integrated data dictionaries. To service both needs requires a system of substantial complexity. If an enterprise wishes to lean more heavily in the direction of one of the viewpoints, then some simplification is possible. One such simplification would be to consider the form of a repository at an enterprise that does not rely on CASE technology in any systematic way. One disadvantage is that the automatic population of the repository via the tools used throughout the software life-cycle (planning, analysis, design, implementation, testing, installation, and maintenance) would be reduced. The enterprise would need to rely more on alternative means of population such as source code scanners, import/export facilities (e.g., to/from DBMS dictionaries, and 4GL library meta-data), and as a last resort, manual data entry.

Repository technology is complex; fully populating a repository takes a long time, and the introduction of this technology will impact a large fraction of the user community. Consequently, one might naturally ask why this technology is necessary. The Information Technology (IT) environment of the the mid to late 1990's will be multifaceted (multiple machine types, multiple vendors, multiple operating systems, multiple DBMS's, multiple application development tools, purchased software from all sorts of sources, as well as substantial in-house development). Centralized, client-server, and fully distributed architectures will also co-exist. This explosion in variety and complexity has the potential to lead to chaos. The hope of repository technology is that not only can it advert this, but that it can make IT more accessible, more understandable, and easier to use than it is today, as well as also enhancing application development productivity.

Perhaps an analogy would be helpful. From the users' point of view, they could think of a repository as their guide to a huge library (an information library). As with real libraries some of the information is there to help library patrons access other information. Examples include card catalogs, reference materials, citation indexes, abstracts, etc.). These are often consulted before the actual information of interest is found and read. Without these resources large libraries would be nearly inaccessible. After browsing the library, the users could access/use anything in the library that they are authorized to use (e.g., read a manual, access a database table, or run an application). Eventually, users should come to view the repository as an indispensable source of vital information - their roadmap that helps them perform their daily tasks. Some libraries also provide librarians who play an active role in assisting patrons in finding information. They can essentially check-out the information for the patron. Along the lines of this analogy (or metaphor), Figure 1 depicts how a repository could be used to assist users in finding data processing assets (applications to run, or files or databases to access). The figure is divided into two parts: Figure 1.a depicts a repository that is basically passive. It tells users what they needs to know, but the rest is up to them. Figure 1.b depicts a repository that plays a more active role by checking assets out for users. (A more detailed discussion of active versus passive, as well as the gradations between them, is presented in Section 4).

This report details how repository technology can be utilized by the Westinghouse Savannah River Company (WSRC). As such, it provides an overview of issues concerning repositories. It discusses the advantages/disadvantages of utilizing repository technology, the intended scope of the repository, and desirable characteristics that a repository product should have. It presents a design of a repository that emphasizes end-users. This emphasis is particularly appropriate for WSRC since (i) it is not a commercial software development company (where the development side is paramount), and (ii) it has a very large and varied end-user community exhibiting increasingly greater demands for information at their fingertips. Once a repository is designed and built (or purchased and installed), there are still several important and complex issues to be dealt with. The principal issues are the following: techniques to populate a repository, user interfaces, repository services, repository administration, and impact on the user community. Furthermore, the design of the repository is typically not a one shot deal; it will likely need to be extended as experience is gained in using the repository. These issues are covered in detail in this report. Supplementary technical information will also be provided by two additional documents. Further information on detailed aspects of building and populating a repository may be found in the thesis by (Krishnan, 1994), while further information on repository services may be found in the thesis by (Edwards, 1994). These graduate students are developing the *WSRC Repository Prototype*. This prototype will serve as a vehicle for evaluating the complexity and utility of various aspects of repository technology. It will also facilitate the refinement of techniques and approaches to populating, administering, and using repositories. It will serve as a testbed for such experimentation both at the University of Georgia and at WSRC. This prototype is being implemented using the Oracle RDBMS, Oracle Pro*C, and Oracle CASE tools.

Figure 1. Overview

2. Scope of the Repository

As discussed above, many vendors are designing repositories to have a comprehensive scope. Such a wide net will catch much of the important corporate meta-data, automatically. Therefore, this seems to be the ideal. It is still an unanswered question as to whether this ideal is feasible for most enterprises. The complexity and impact on current development practices may be too costly. Therefore, one sensible approach is to divide up the problem and determine which pieces are most suitable to which organizations. We are proposing a three dimensional decomposition.

1. **CASE vs. Non-CASE Oriented**. Organizations which do not systematically apply CASE (Computer-Aided Software Engineering) technology, would be primarily interested in keeping track of information resources (files, databases, and in some cases expert systems/knowledge-bases) and application programs. Naturally, information resources and application programs do not exist in a vacuum, so that if we wish to know where they are and how to use them, then we also need to keep track of users, machines, systems and tools. In addition, if models do exist, an organization may wish to keep them in the repository. They are not only useful in the development process, but can benefit end-users as well. For example, diagrammatic representations of the information resource can assist users in finding the particular data that they are interested in. Consequently, organizations not oriented toward CASE may still be interested in facets of CASE technology. It boils down to a cost-benefit analysis. Is the extra complexity of storing model information justified by the additional benefits provided to users? Those organizations who currently use or are planning to CASE should refer to Appendix A, for a more complete discussion of the costs and benefits of using this technology in conjunction with repository technology.

2. **Data vs. Process**. If the purpose of a repository is simply to keep track of information resources, then it should more properly be called an integrated data dictionary. However, without adding inordinate complexity, applications as well as associated information can also be tracked. This dichotomy is revisited if we decompose it along the previous dimension (CASE vs. non-CASE). With respect to non-CASE, most organizations could benefit from a systematic, logically centralized tracking of information resources and applications. However, with respect to CASE, it seems that there are some reasonable options: Some organizations may wish to store both data models (e.g., Entity-Relationship Models) and process models (e.g., Data-Flow Models); some may choose to store just data models; while others may not want to store any kind of model. An argument for the second option is that data modeling technology is easier to use and easier to demonstrate its benefit, while the need for process modeling technology may not be that great when development is primarily done with Fourth Generation Languages, 4GL's, (e.g., Software AG's NATURAL 2 or PowerSoft's PowerBuilder).

3. **Macro vs. Micro**. It is clear that a repository should provide a macroscopic view of information resources and application programs. The question really concerns how much detail to include. This is particularly relevant on the application program side. An application is composed of modules which are themselves composed of sub-modules (e.g., procedures, functions, classes, types, etc.). One can even drill-down to the level of statements, fields, constants, and variables if necessary. The more an organization is interested in a disciplined approach to software development - one that maximizes the reuse of software components - the more detail it would like to see. However, if end-users are the central concern, unnecessary complexity can be discarded. One place to cut it that seems reasonable is at the module level. By providing information about the applications and their constituent modules, end-users will be able to find the applications they want, and some of the more sophisticated end-users will even be able to rebuild applications from their source modules (this may be necessitated by a system upgrade). In addition, developers will have a guidepost to find detailed source code and documentation to help them achieve the goal of code reuse. (An interesting aside concerning this very issue, has to do with performance. IBM wishes to maintain all sorts of detailed information in their repository. A considerable amount of database research suggests that storing complex objects which have components consisting of subcomponents in a relational database will lead to poor performance. IBM has found this to be the case, and is in the process of implementing its repository on top of an object-oriented DBMS.)

A suitable starting point for WSRC would be to plan on using repository technology in a largely non-CASE oriented environment, where both data and process aspects are important, and with a macroscopic viewpoint taking precedence. Any development or procurement should not totally limit the scope to this, since repository technology is a long range solution, and WSRC will likely wish to expand its scope over time.

3. Defining the Term Repository

A precise and clean definition of the term repository may not be possible, given its comprehensive nature and due to the fact that different organizations have different needs and emphases. For those not oriented toward CASE technology, it may be helpful to view a repository as a next generation integrated data dictionary. Along these lines, let us characterize data dictionaries, integrated data dictionaries, and finally, repositories.

3.1. Data Dictionaries

In a database, information about schemata (plural of schema) is stored as meta-data in a data dictionary (Date, 1990; Elmasri and Navathe, 1989; Wertz, 1989). Meta-data are "data about data" and, thus, indicate what kinds of actual data are available. For example, in a relational database with a data dictionary there may be tables indicating what data tables, attributes, index structures and authorized users are associated with a particular database. An entry for say an Age attribute might include the name of the attribute, a human readable description, a data type specification (useful to both humans and programs), and integrity constraints (e.g., in the range [18, 65]). Thus, data dictionaries describe the information that is contained in *databases*.

3.2. Integrated Data Dictionaries

Information Systems (IS) shops are interested in broadening the notion of a data dictionary in several ways. Two of these ways lead to the notion of an integrated data dictionary. (A possible source of confusion is that some authors use this term to mean a data dictionary integrated with the DBMS; we are using the term to indicate that the integrated data dictionary gathers information that would otherwise only exist in separate data dictionaries.)

1. **Multiple Databases**. It is common for a single DBMS to support multiple databases. In such cases, meta-data about distinct databases (e.g., payroll, inventory, customer transactions) need to be available to the IS users, designers, implementors, and administrators. This is relatively easy to deal with, so ordinary data dictionaries (e.g., the data dictionary that comes with the Oracle RDBMS) can store this information. Still, it can be useful to obtain a higher level integration. For example, in addition to storing information about the tables and columns for database x and for database y, one could also store information about any commonalities that exist between. Maybe both x and y store salary information. Making such associations is not always easy because of the existence of homonyms (one name used for two different things) and synonyms (different names for the same thing). Enforcement of enterprise-wide naming conventions and efficient name approval schemes can help reduce these problems.

2. **Multiple DBMS's**. The situation becomes more complex if multiple DBMS's (often sold by different vendors) are used. There exist many sources of differences: different query

languages, different ways of representing structures and constraints, and even different data types. Minor differences can be handled by standard translation tables, while more major ones may require a data administrator to specify complex mappings. Standards such as SQL2 and NDL help reduce these complexity, but even if a vendor implements a standard, they usually extend it in a non-standard way.

3. **Multiple Underlying Data Models**. In modern large organizations, even greater diversity is to be expected. Many of the DBMS's will be based on different underlying data models (e.g., relational, network, hierarchical, and object-oriented). For example, WSRC makes extensive use of the Network Data Model upon which Software AG's ADABAS is based, and also makes use of the Relational Data Model upon which Oracle, Sybase, and Rdb are based. Furthermore, there are expectations that Object-Oriented Databases will be utilized in the not too distant future. Techniques to deal with this diversity can be complex. Many of the problems are analogous to those addressed by the literature on Federated Databases (see Appendix B).

4. **Other Information Resources**. These include other types of information resources, such as *data files*, *warehouses*, and possibly even *knowledge-bases*. The meta-data maintained about data files can be quite varied. For example, some systems such as DEC's RMS allow a record structure to be imposed on data files, so that the meta-data recorded for such files would not be unlike the meta-data for tables in relational databases (Hobbs and England, 1993). Much of the information maintained by today's organizations is not held in traditional databases, so keeping track of such external data sources is important.

3.3. Repositories

If the information resource is simply to be described, and not tracked, controlled, nor integrity checked beyond that done upon database insertion, then an integrated data dictionary may suffice. If, however, these additional capabilities are desirable, or if an organization wishes to understand the environment in which the information resource exists, then further broadening is called for. A determination of what additional types of information are to be included in a repository is very much related to the discussion in the previous section on the intended scope of the repository. The desirable types of additional information include:

1. **Applications**. IS shops maintain information resources and provide the capability to store, retrieve, and manipulate them. These capabilities are provided by software running on top of hardware. This software consists of systems (e.g., Operating Systems (OS's), Database Management Systems (DBMS's)), productivity tools (e.g., Spreadsheets), and application programs. Although end-users can utilize any of these, their most common means of accessing information resources is with applications. Because applications (typically developed in-house) tend to proliferate, it can be very beneficial to organize information about them in a repository. The amount and degree of detail to be maintained is again

dependent on the intended use of the repository. If the level of detail is cut at the module level, then the repository would store both definitional information and documentation concerning application programs, and their constituent modules, as well as information on how to access the source code (instructions, as well as host, path, and file names).

2. **Application Environment**. If information about applications is to be stored, then surely some associated information should be also. The types of associated information will again depend on the orientation of the organization. Most organizations will be interested in keeping track of software that has been developed or is undergoing revision and/or maintenance. Important issues here concern the management of versions. Questions to be addressed include: How versions of modules are brought together to make a new version of an application; what tools are used to produce the new version (e.g. compiler, linker, make-tool); and what is the status of the new version (e.g., test, alpha, beta, production). Additional associated information includes what the application is run on (network, machine, required system software) as well as who is authorized to use it and what types of usage they are allowed (e.g., Create, Read, Update, and/or Delete (CRUD)). Finally, all of the basic components (e.g., Information Resources, Applications, Users, etc.) should be thoroughly cross-referenced. Some organizations may also be interested in keeping track of analysis and design information. The primary addition here would include both data and process models. These models are typically centered around a diagram (or set of interrelated diagrams). Popular types include: Entity-Relationship Diagrams (ERD's), Bachman Diagrams, Object Modeling Technique (OMT), Data Flow Diagrams (DFD's), Function Hierarchy Diagrams, Action Diagrams, Structure Charts, and Petri Nets. At present ERD's and DFD's tend to be the most popular, but object-oriented models such as OMT are catching on.

We will close this section by giving a couple of quotes of others who have defined or characterized the word repository, the first by (Flaatten et al., 1992) and the second by (Tannenbaum, 1993).

> "Data dictionaries were initially created in the early 1970's to centralize and coordinate all these descriptions of data. They were thus the ancestor of CASE tools, and they are still at the core of the installation functionality (program generation, coding, testing) of most serious CASE tools today. Indeed, the emerging notion of *repository*, the facility that stores and retrieves descriptions of all aspects of a system, design as well as code, data as well as procedures, is directly based on the concept of data dictionary."

> "MIS development organizations are revisiting the repository as a bridge to new technology areas. Organizations beginning to down-size must grapple with the relationship "new" applications have to each other, to mainframe-based legacy applications,

and to multiple development tools in use. They also face an installed diversity of execution platforms, operating systems and deployed software. Previously, developers viewed the repository as just a computer-aided software engineering (Case) tool integrator or corporate "data dictionary". Now, however, MIS is turning to repositories for their overall integration capabilities and accessibility to both the application development environment and end-user production world."

4. Desirable Characteristics

In her article (Tannenbaum, 1993) lists three desirable characteristics that a general-purpose (or "true") repository should have.

1. **Extensibility**. The capabilities of a repository are largely determined by the schema that is used for its underlying database. The schema will indicate what entities are stored and what relationships exist between entities. The data model for such a schema is often called a meta model since the repository stores meta-data. Therefore, the design of the meta model is very key to the usefulness of the repository. Some organizations may find a vendor supplied meta model that fits their needs satisfactorily, but it would be better if an organization could extend the capabilities of the repository by adding, changing or deleting components in the meta model. However, (Matthews and McGee, 1990) caution that the meta model should be reasonably stable, allowing additions to occur frequently, changes to occur infrequently and deletions to hardly ever occur. Closely related to this issue is the ability to *customize* the repository to better meet the needs of the organization (e.g., a GUI-based navigator/browser can be customized to better suit the users' preferences).

2. **Vendor Independence**. In the emerging client-server environment, systems, tools and applications will be purchased from a variety of vendors as well as developed in-house. With an open architecture and well-defined (eventually standardized) interfaces the repository should be able to work with a variety of products. Purely proprietary solutions are likely to run into problems.

3. **Versioning**. Large IS shops will need to deal with versions of software: models, systems, tools, databases, and applications. Keeping track of which versions of *x* work with what versions of *y* can be difficult, but necessary nonetheless. In particular, multiple versions of applications are commonplace. Furthermore, an application is made up of modules, the code for which is stored in an source file. To fix bugs, improve performance, enhance visual appearances, or add functionality, developers will be modifying these source files. This often results in multiple versions of a module. It is not necessarily the case that a end-user would want to run the version of the application built from the latest versions of the component modules. Some of these modules may have not been tested or used long enough to satisfy certain end-users, or possibly, data format requirements have changed in the new version and the end-user does not want to change over just yet. For large IS shops, some means of organizing versions can be very helpful (more about this in Section 8.2).

Some additional characteristics that can be used in evaluating repositories are given below. There may not be a definitive choice for each issue, rather choices should be made to met the present and evolving needs of the organization.

4. **Active vs. Passive**. If the repository serves purely as a source of descriptive information, it is said to be passive. If the repository controls the use of data and applications, it is said to

be active. This control can be exerted over end-users (who is permitted to run/access what), developers (who is permitted to modify source and when may they do so, i.e., check-out software), and software itself (what applications are permitted to access what data tables). (In lieu of uniform repository-based control of access, security features of most Operating Systems (OS's) can be used to control access to files, and DBMS's can be used to control the access to database tables/views. Unfortunately, OS-based control of access to files is usually not as flexible as one would like.) In addition to this type of control, repository information can affect processing, e.g., the existence of an index structure can influence query processing. Ideally, a repository product should allow an organization to adjust its level of activity (control) over time to best meet its present needs. When a repository is first installed, an organization might prefer it to be passive. As users become familiar with it and have had adequate training, the level of control can be increased. Finally, DEC divides this type of classification further into three types of dictionaries/repositories (Hobbs and England, 1993).

4.a. *Passive*. "A passive dictionary (/repository) stores the definitions inside the dictionary. It can be accessed by other users, but there is no tightly coupled integration to the development environment."

4.b. *Active*. "An active dictionary (/repository) is tightly integrated with the development environment. It includes all the functionality seen in a passive dictionary, but also forces software tools to store data definitions within it. This enables such tools as compilers to read the definitions from the dictionary at compilation time and then inform it that they are using a particular piece of information it holds."

4.c. *Dynamic*. "A dynamic dictionary (/repository) is one from which the application development environment obtains the current definition at run time rather than at compilation time. Whenever changes are made to the dictionary, they are then automatically transferred to the development environment. Alternatively, a messaging system will advise the development environment of any such changes. The dynamic dictionary sounds like the perfect solution in a development environment. Unfortunately, however, the performance overhead in obtaining definitions at run time makes it less common."

5. **Legacy Systems vs. Client-Server Systems**. Some repositories are designed for a mainframe environment, while others focus on Local-Area Networks consisting of a multitude of Personal Computers and Workstations. And, of course, others work in both worlds. There are several ways to look at this issue. In a purely mainframe environment with centralized control, the level of chaos does not match that of a large multi-platform, multi-vendor client-server environment. Therefore, one might argue that the need for a repository is greater in a client-server environment. In addition, in this environment the opportunities are greater since much of the application development is yet to be done. Finally, repositories in this environment are more likely to be used, since they will have user-friendly GUI

interfaces. Organizations dependent upon legacy systems, and particularly those wishing to integrate these two environments should look for repositories that work in both worlds. So far, only a few repository vendors provide full functionality in both environments (see Appendix D).

6. **Physical Distribution**. Logically, the repository should appear to be an integrated whole, but physically it may be either distributed or centrally located. The tradeoffs are the usual ones of improved performance and reliability (with distribution) vs. less complexity (without distribution). As client-server architectures become more important and mature systems software becomes available on them, physical distribution would provide the following advantage: elimination of the central node as a bottleneck and as a single point of failure that disables the entire repository (and therefore important IS activities). The availability of high performance, robust distributed DBMS's upon which the repository technology could be built, makes this a viable option (concurrency control and recovery would be handled by the underlying distributed DBMS).

7. **In-House vs. Purchased Software**. Applications developed in-house should be able to register a substantial amount of useful meta-information into the repository. For example, if useful comments are entered into the comment boxes provided by PowerBuilder, then PowerBuilder itself will be able to supply much of the needed information via its export/reporting capabilities. In the WSRC Repository Prototype, we are extracting PowerBuilder's meta-data and inserting it into the prototype's Oracle repository database. (See Appendix F for an overview of PowerBuilder's meta-information.) The situation is more difficult for purchased software. Some options are to enter some meta-data manually, attempt to adapt an extract program to extract the purchased software's own meta-data, or (when they become available) try to purchase software that has repository-ready interfaces. This final option is not likely to occur widely until a definitive repository standard has emerged.

8. **Type of Modeling Technique**. Information models of repositories are of two basic types: Those based on the *ER (Entity-Relationship)* approach, and those based on the *OO (Object-Oriented)* approach. An ER approach is popular because of its wide use in designing relational databases. Typically, an extended ER approach would be preferred. The OO approach is gaining in popularity because of the emergence of OODBMS's and the fact that they are superior for designing systems composed of complex objects (as is the case for comprehensive repositories). In addition, both 3GL-based (e.g., C++) and 4GL-based (e.g., PowerBuilder) development are beginning to follow the object-oriented paradigm. Even the standards organizations are moving in this direction.

5. A Multi-Level Framework

At this point we will go into a more detailed analysis of what constitutes a repository. The organization of a repository is best understood as a multilevel framework. We will motivate this by considering the general concept first. A database is a large integrated collection of information which is organized according to type. Consequently, it is a two level concept, with instances (actual data) forming the lower level and type information expressed in the schema forming the upper level. Modern databases view this type information as important enough to store data about these types in a separate small database, known as a data dictionary. Since a data dictionary is itself a database, it is again a two level concept. Therefore, modern databases actually consist of two databases: a database storing the application data, and a database storing data about this application data (in other words meta-data). These notions can be clarified with a diagram consisting of three levels.

```
                          =============
Dictionary Schema         | meta model |
                          -------------
  Dictionary Data         | meta-data  |  Application Schema
                          -------------
                          |    data    |  Application Data
                          =============
```

The top level (meta model) simply defines a type structure, but contains no data. Therefore, the meta model specifies the type structure for the meta-data. Relevant types include `Table`, `Column`, and `Constraint`. The instances that fill up these types would be stored at the level below. Examples of `Table`s include `Employee`, `Department`, and `Customer`. Again, these instances at the meta-data level represent types for the level below. Instances for the `Employee` type might include `employee A`, `employee B`, and `employee C`.

Multi-level frameworks, such as the one above, exhibit an interesting regularity. Each of the middle levels viewed from below represents type information, while viewed from above represents instances. The exceptions are the top-most and bottom-most levels. At the data level, there is no view from below, so that the type viewpoint does not exist. Similarly, at the meta model level, there is no view from above, so that no instance (or data) viewpoint exists.

The above discussion may lead one to ask how the meta model is itself structured. The answer is that it is fixed and hard-coded. The vendor decides what kind of meta-data is needed for its product and supplies this. However, to meet the special demands of some of its customers, some vendors allow their data dictionary to be extended. A natural way to do this is to let the structuring information for the meta-data (i.e., the meta model) be defined in yet another database, which is conceptually one level higher. Consequently, one ends up with a four-level framework, as shown below.

```
                     ==================
                     | meta meta model |  Definition Schema
                     ------------------
 Dictionary Schema   |   meta model    |  Definition Data
                     ------------------
   Dictionary Data   |    meta-data    |  Application Schema
                     ------------------
                     |      data       |  Application Data
                     ==================
```

This new fourth level provides for extensibility and interoperability. Since this upper level database is relatively small, its structure can be fully specified in standards, and even if there are multiple standards, translations between these standards will be facilitated. An example of data at this fourth level might be the data that fills-in a type called `Object-Type` which could consist of instances such as `Table, Column,` and `Constraint`. If an enterprise wishes to include information on application programs, they can extend the dictionary by adding an object-type called `Application`.

In summary, we now have three databases, an Application Database, a Dictionary Database, and a Dictionary Definition Database, with each database consisting of a lower level (instances) and an upper level (types). We can think of these two levels as forming level-pairs, namely, Definition, Dictionary, and Application. Hence, we have four levels and three level-pairs. This highly useful framework is the basis for several International Standards (Thompson, 1992):

1. **IRDS** (Information Resource Dictionary System). This standardization effort was sponsored by ANSI (American National Standards Institute). A standard was completed in 1988 (Law, 1988), but there are several proposals under review which would extend the standard. A draft of ANSI IRDS V2 (Hodges, 1992; Goldfine, 1993) which includes the ATIS (A Tool Integration Standard) specifications developed by DEC and Atherton Technologies has recently been produced (Hobbs and England, 1993). Requirement #4 (of 259) for IDRS2 states that "the IRDS shall be object-oriented, with multiple inheritance and complex objects" (Goldfine, 1993).

3. **PCTE** (Portable Common Tool Environment). This standardization effort was sponsored by ECMA (European Computer Manufacturer's Association). Standards documents

(Simon, 1993) were produced in 1986 (PCTE V1.4), 1988 (PCTE V1.5 and PCTE+), and 1991 (a joint report by both ECMA and NIST). Extensions are ongoing today, e.g., a language-specific binding to C++ is currently under development. Recently, IBM has announced support for PCTE within their proposed AIX repository (Thompson, 1993c).

3. **CDIF** (Case Data Interchange Format). This standardization effort was initiated by a group of CASE vendors and was sponsored by EIA (Electronics Industry Association). The standard specifies a common format for import/export files. The parts of this standard that concern repositories should be complete by 1994.

Within these standards, the names for the levels vary, but the concepts remain the same. The various names are shown below.

	IRDS	PCTE	CDIF
meta meta model	Fundamental	Meta Schema	Meta Meta
meta model	IRD Definition	SDS	Meta
meta-data	IRD	Object	Model
data	Application	(undefined)	Application

Note, IRD stands for Information Resource Dictionary, while SDS stands for Schema Definition Set.

6. Modeling Technique

The question arises of how to precisely design/define these three databases. We need not bother with the Application Database, since this data is handled by conventional means. (Typically, an enterprise will define several applications schemata to meet its own particular needs.) Consequently, we are not directly interested in an enterprise's data, but rather the enterprise's meta-data. Thus, we need only design/define the Dictionary Database and the Definition Database. (The scope of the Dictionary Database determines whether we are dealing with a dictionary, an integrated dictionary, or a repository). As far as we can tell, there are four approaches to designing/defining the Dictionary Database (whose schema is referred to as the meta model) and the Definition Database (whose schema is referred to as the meta meta model). These approaches are:

1. **Extended Entity-Relationship Model**. This is a reasonable choice, since it is the most widely used approach to designing databases. Its design can be straightforwardly mapped to several implementation oriented models (relational, network, or hierarchical models). A disadvantage is that the Entity-Relationship (ER) Model (Chen, 1976) needs to be extended to be useful in complex settings (e.g., with Entity Supertypes, Entity Aggregations, more general means of expressing constraints, etc.). Such extensions have been added. The problem is there is no consensus as yet to the specific form of these extensions. The relevant standards organizations have used this approach to design and/or specify their standards, e.g., IRDS, PCTE, and CDIF. In addition, several repository vendors such as IBM and DEC have used this approach.

2. **Object Model**. As with object-oriented programming, Object-Oriented (or simply just Object) Modeling is gaining in popularity. IBM adds to an extended ER Model the notion of object type. Object types are similar to entity types, except that they model things that are more abstract than data and which include behavioral specifications (in the form of methods) as an essential component (Sagawa, 1990). IBM's Repository Manager (RM) contains the following types: Entity Types, Entity Aggregations, Relationship Types, and Object Types. Both Entity and Object Types provide a sub (super) typing mechanism. Entity Aggregations are groups of closely related Entity and Relationship Types. They provide a convenient means to provide an uncluttered global picture. Details can be obtained by exploding an aggregate into its components. Recently, IBM has used C++ class definitions to define their models, and DEC has used object modeling based on ATIS (DEC, 1991). The main advantages of an object modeling approach are that it provides superior forms of aggregation compared to even extended ER modeling, and allows behavior to be encapsulated within the objects (in the form of methods). Furthermore, if a repository is meant to be comprehensive, it must store complex objects that consist of sub-objects made up of sub-object, etc., so the performance with an RDBMS may be inadequate. Consequently, if an OODBMS is to be used, it is only natural to use an object modeling

technique. An OODBMS is likely to be preferred if an organization wishes to fully support Application Development (AD) with the repository. There are two reasons for this: (i) AD tends to create a large number of highly complex objects, and (ii) many AD languages (e.g., PowerBuilder and C++) are object-oriented so that an object-oriented Application Program Interface (API) to the repository is the most natural way to go. For organizations emphasizing information management and end-users, an RDBMS might suffice, but even the standards organizations are moving to the OO approach.

3. **ISO SQL**. Since all the concepts in relational databases can be expressed using Structured Query Language (SQL), it is a reasonable candidate. It was used by IRDS as a means of specifying the standard. An advantage of SQL and more recently SQL2 is that the languages have a rigorous standard. While SQL2 is preferable to SQL, it still has some disadvantages, principally, that it does not support subtyping or many-to-many relationships. The former problem will be rectified with SQL3 which should become a standard in 1995.
4. **Prolog**. Prolog can be used to define and manipulate ER or relational data, and is very good for specifying constraints and business rules. KnowledgeWare uses this approach for their repository which supports their CASE tools.

6.1. Meta Meta Model

The basic information model for an enterprise is the meta model. It specifies the types of meta-data to be stored (e.g., on information resources, applications, and users). In the next subsection we will develop this model. In order to precisely define the components of the meta model, we must first define the level above it, since the types specified in the meta model become data at meta meta level and are stored in the types defined at this higher level. The types specified at the meta meta level are fixed by the vendor/designer; extensibility is for the level below. The fixed types at this topmost level are usually motivated by ER and/or OO modeling concepts. We shall do the same here. Although many believe that an OO approach is superior, we will use an extended ER approach for the following reasons: we have access to Oracle's CASE*Designer which uses the ER approach; we are developing the WSRC Repository Prototype database using Oracle, Version 6/7 (an OO design ideally needs an OODBMS or an extended RDBMS (e.g., Oracle, Version 8)); and so long as excessive detail can be left out of the repository, an RDBMS may suffice. Still, we felt that by itself, the facilities provided by Oracle's CASE*Designer were lacking, so we added a few extensions (see below).

LEVEL 0: (ER Diagram for the Meta Meta Model)

Object_Model
Object_Type
    Entity_Type
    Relationship_Type
    Arc
    Entity_Aggregate
    Virtual_Rel_Type
Attribute
Diagram

The actual ER diagram is shown in Figure 2. The entity types in order of importance for the meta meta ER diagram are defined below.

1. **Object_Model**. Entity Type. An Object_Model consists of a collection of interrelated Object_Types, and is typically displayed in one or more Diagrams.

2. **Object_Type**. Entity Supertype. An Object Type is anything that can be displayed in a diagram.

2.a *Entity_Type*. Entity Type. An Entity Type is used to model a class of things. An entity is an instance of an Entity Type and must be uniquely identifiable. A special case of an Entity Type is an Entity Supertype. These represent Entity Types that can be subtyped. In diagrams, Entity Types are denoted by boxes. Entity Supertypes will have boxes contained within them.

2.b *Relationship_Type*. Entity Type. A Relationship Type is used to model a class of associations. A relationship is an instance of a Relationship Type and represents a particular association between two or more entities. The arity of the relationship indicates how many entities participate in the relationship. Since ternary (or higher arity relationships) are hard to understand, we restrict ourselves to binary relationships. Each relationship will have a source and destination entity. A Relationship Type can be further classified in terms of how it constrains the mapping between source and destination entities. The mapping can be many-to-many, many-to-one, one-to-many, or one-to-one; and participation in a relationship by entities can be either optional or mandatory. In diagrams, Relationship Types are denoted by lines connecting two boxes together. Many-sides are indicated by crow's feet, while optional participation is indicated by making part of the line dashed. *Extension*: We allow Relationship Types to have their own attributes, since the alternative is to elevate the Relationship Type into a new Entity and introduce two new Relationship Types. Repository designs are too complex for this luxury.

Figure 2. Meta Meta Model

If a design tool (e.g., Oracle's CASE*Designer) does not include this capability, it can be accomplished by inserting the missing attributes after the relational schema has been generated.

2.c *Arc*. Entity Type. An Arc is used to indicate that an entity can participate in one of several Relationship Types, but not more than one (mutual exclusion).

2.d *Entity_Aggregate*. Entity Type. *Extension*: Since large ER Diagrams with more than 10 to 15 Entity Types are very difficult to display, let alone understand, it is important when designing complex models such as those required for repositories to decompose the problem. A good technique to accomplish this is to use Entity Aggregations to form multilevel ER Diagrams. The highest level (LEVEL 0) would provide a macroscopic view, and would typically include several Entity Aggregates. Each Entity Aggregate would hide its details, which are that it consists of several Entity Types, Relationship Types, and possibly additional Entity Aggregations. The details of an Entity Aggregate are exposed by a lower level (e.g., LEVEL 1) ER Diagram. This more detailed ER Diagram must include a primary Entity Type which serves as a surrogate for the Entity Aggregate. The Entity Aggregate provides a way of hierarchically organizing a set of ER Diagrams. For the purpose of generating, for example RDBMS schemata, the hierarchical structure would need to be flattened into a large single level ER model (this need not be displayed, but needs to be either stored or reproduced from the dictionary/repository).

2.e *Virtual_Rel_Type*. Entity Type. *Extension*: Virtual Relationship Types are similar to ordinary Relationship Types except for one notable difference: At least one of the attached Object_Types is an Entity Aggregate rather than an Entity Type. When a multilevel ER Diagram is flattened by exploding the Entity Aggregate, each Virtual Relationship Type is replaced by an actual Relationship Type by connecting the end(s) to the appropriate Entity Type within the Aggregate. By default a single Virtual Relationship Type will be mapped to single ordinary Relationship Type by simply connecting it to the surrogate for the Entity Aggregate. This can be refined manually by a designer specifying replacement or supplementary Relationship Types. Once specified, the connecting Entity Types can be recorded as part of the model information so that the flattening operation can be performed automatically.

3. **Attribute**. Entity Type. An Attribute is a descriptive property whose value describes an entity (or as in our extension a relationship).

4. **Diagram**. Entity Type. A Diagram is used to display interconnected Object Types. An Object Type can appear in more than one diagram. The coordinates indicating where to display the Object Type are stored as attributes of the Relationship Type associating Diagram with Object Type.

6.2. Meta Model

The meta model is often referred to as a *Repository Information Model* (RIM). Designing the RIM correctly is very important. It determines the type of corporate meta-data that is to be maintained in the repository as well as the structure and interrelationships of its components. Much of the current thinking is that repository vendors should supply a core information model that the client organization can then extend (possibly with the help of vendor consultants) to meet their particular needs. An additional benefit of such an information model is that it can provide a unified high-level conceptual view of the enterprise's information resources and applications. This can be a boon to end-users, developers, and project managers alike, since without an overall map it is quite possible to get lost in all the information. (Please note that this does not mean that all of the Application level schemata need to be united in one grand model; the integration is at a higher level of abstraction.) This issue has been stressed by several authors (Hoffnagle and Beragi, 1985; Zachman, 1987; March and Kim, 1992). They claim that without a common high-level model to describe the existing corporate information resource, fragmentation and confusion may ensue in any attempts at integration. This enterprise level view of the corporate Information System (IS) will enhance the understanding of what is currently available in the IS and how various resources are related. It can also be used as a guide for new development to help assure that it will fit in with the existing system.

The following is an extended ER Design of an RIM. The ER design is divided into several pieces. The overall design is given by LEVEL 0, while more detailed pieces are given by several LEVEL 1 designs. As indicated by the LEVEL 0 design, such a repository would keep track of Data Files, Databases, Applications, Systems, Machines, and Networks as well as Users. Relationships would be used to cross reference these entities.

With such a repository users could (among other things) do the following:

1. browse for the application they are interested in running;
2. see if they are authorized to run the application;
3. if it is licensed software, determine if there are any copies available for check-out;
4. determine its minimal execution requirements (e.g., MS-Windows, 12 MB of memory, etc.);
5. be informed of the basics on how to run it (along with information on how to learn more about the application by consulting on-line help, running tutorials or reading manuals); and
6. find out where the application is currently available.

Similar questions could be answered regarding information resources such as databases. The repository would also keep track of information flows (what is the source of a data item, who accesses it, what applications use it, etc.). Many organizations would also be interested in keeping track of hardware (machines and networks). In particular, it seems that in a client-server environment such information would be important to users who need to know where applications/data are located (and possibly how to make the connection).

6.2.1. ER Model for the Overall Repository

LEVEL 0: (Macro Perspective of the Overall Repository)

- Information_Resource
    - File
    - _Database
    - Warehouse
- _Application
- User
    - End_User
    - Application_Developer
    - Database_Designer
    - IS_Manager
- Business_Policy
    - Business_Goal
    - Business_Rule
- Model
    - Data_Model
    - Process_Model
- Tool
    - Development_Tool
    - Productivity_Tool
- System
- Machine
- Network
- Business_Unit

The actual ER diagram is shown in Figure 3. Some of these entity types are actuality entity aggregations. These are shown in the ER diagrams with an AE in the lower right hand corner. In addition, many of these entity types have subtypes and some of these subtypes have their own subtypes as well. To keep the Macro ER diagram from becoming too cluttered, only first level subtyping will be shown. Any further subtyping will be deferred to the LEVEL 1 ER diagrams. Let us now describe more fully the meaning and purpose of the entity types listed above. We shall do so in order of importance.

Figure 3. Macro Model

1. **Information_Resource**. Entity Supertype. This is the most important part of an information system. It is where the data is actually stored. Users often store their data in files. If the contents of a file (e.g., a data file or a source code file) are of some general interest, then descriptive information should be stored in the repository so that others can use/access it. Data that is meant to be widely shared should be integrated and organized into a database. It is essential to store descriptive information about the organization's databases in the repository. Finally, as the data in operational databases begins to age, some organizations have chosen to migrate it to an information warehouse (which may store the original or a summarized form of it). The subtypes of Information Resource are as follows:

1.a. *File*. Entity Supertype. The repository can be used to keep track of important files by registering them in the repository. An organization may be interested in keeping track of important data files, documents, source files, object files, object library files, and executable files. To fully identify a file via its location it is often necessary to give a hostname and full pathname.

1.b. *_Database*. Entity Aggregation. Information about databases with more than one user should definitely be maintained in the repository.

1.c. *Warehouse*. Entity Type. Rather than just putting old data into archival storage (e.g., on tape), a newer strategy is to keep them or at least summarized versions of them on-line and easily accessible. The data is warehoused in case it is needed at some later time. Maintaining historical data is also important for tracking trends. Although an (Information) Warehouse should really be an entity aggregation, we are modeling it as an entity type and viewing it as a black-box, since a Warehouse will likely have its own repository.

2. **_Application**. Entity Aggregation. It is important to keep track of applications since these are the primary means by which data are stored, accessed, and manipulated.

3. **User**. Entity Supertype. The purpose of an information system is to service the user community. The repository needs to keep track of what kinds of users exist and what authorizations and responsibilities users have. As far as authorizations are concerned, users may be authorized to run certain applications (see User Authorization under Repository Services). Access to corporate databases and the repository itself can be carefully controlled through the use of Authorized Roles (see the next sub-section). In terms of responsibilities, there are several types: what systems analyst is responsible for an application, what developer/programmer is responsible for a module, what DBA is responsible for a database, what database designer is responsible for a table, and what end-user (or end-user group) is responsible for the data entered into particular table (or column). This last item is related to the idea of tracking information flows, and in particular determining the source of information. This can be important if erroneous data is found, since it can be tracked to a responsible party. A corrective action might involve a change in how such data is entered. One other issue is the question of responsible party versus authorized user. The responsible

party could be a group leader who is in charge of several authorized data entry clerks. Some organizations refer to a person with such responsibilities as a custodian. (See Modification Tracking and Usage Analysis under Repository Services for further discussions of these issues.) Finally, users can be classified in several ways. These multiple classifications can be modeled in several ways: by subtyping, by participation in a relationship, and by including indicator attributes. We propose to do all three: include an indicator attribute within the User entity type, namely Security_Level (public, sensitive, or classified), have a relationship type connecting User with the Business Unit they work in, and divide the User entity type into the subtypes shown below. More information about users and the roles they play can be found in Appendix C.

3.a. *End_Users*. Entity Supertype. An End-User can be defined as a computer user whose purpose is to use computer technology to accomplish some business function or task. Although some may participate in high-level system designs they are not inclined nor trained to carry out detailed design, much less implementation. There can be several types of end-users: top management -- might occasionally use an Executive Information System (EIS), middle management -- would study and analyze the outputs of applications looking for trends or unusual behavior, business professionals -- accountants, financial analysts, sales personnel, business analysts, etc. would frequently access corporate information, engineers -- would run certain applications frequently (but would be less interested in corporate information), and clerks -- would be responsible for doing much of the data entry.

3.b. *Application_Developer*. Entity Supertype. These are users who are responsible for developing/programming applications or their component modules. They can be further sub-divided into Systems Analysts, Applications Programmers, and possibly Systems Programmers.

3.c. *Database_Designer*. Entity Supertype. These are the users who are responsible for designing databases. A subclass of this contains users who have the added responsibility of maintaining databases, namely Database Administrators (DBA's).

3.d. *IS_Manager*. Entity Type. This is a relatively small group of individuals who are responsible for the overall function of the IS shop. This group may include a Chief Information Officer (CIO), Application Development (AD) Managers, Project Managers, Data Administrators, and a Repository Administrator.

4. **Business_Policy**. Entity Supertype. Important products of business planning and IS analysis activities are business policies. These can codified in semi-formal English and stored in the repository.

4.a. *Business_Goal*. Business Goals are simply succinct descriptions of what an organization wishes to accomplish. Some examples are:

Increase the revenue in department x by 5%.
Begin mainstream use of the Information Warehouse by date y.

4.b. *Business_Rule*. Business Rules can be expressed in semi-formal English using the following template "ON event IF condition DO action". Some of these business rules will be concrete enough to be implemented as rules (constraints or triggers) that are fired when the repository database (or other database) is updated. The rest of the business rules could be enforced by other means (e.g., use of a language like REXX as IBM does, or use of an expert system, or use of checking facilities built into specific tools as KnowledgeWare does with their CASE tools). An example of a Business Rule is:

ON Update
IF x is an Engineer AND x.YearsExperience $\geq$ 5 AND x.salary < 40000
THEN set x.salary = 40000

This is an example of a rule that is enforced when an information resource is updated. It is like a constraint, except that a specific corrective action is given. Rules that are triggered at query time could be used to derive values, e.g.,

ON Read
IF x.Age is null
THEN return ComputeYears (DATE, x.BirthDate)

5. **Model**. Entity Supertype. Models serve a dual role. They are useful when designing applications or databases, and once an application or database is developed they provide useful high-level documentation. Data models (such as ERD's) are easy to develop and can be straightforwardly translated (either by hand or by tool) into database DDL statements. Although not so straightforward, process models (e.g., DFD's) attempt to do the same thing for applications. An ER diagram providing more details about the Model entity type is shown in Figure 4.

5.a. *Process_Model*. Entity Supertype. Process Models are used to design and document an application.

5.b. *Data_Model*. Entity Supertype. Data Models are used to design and document a database. A subtype of Data_Model is _ER_Model (see Figures 4 and 8).

6. **Tool**. Entity Supertype. Although hard to precisely define, tools help users perform some general task. There are two basic kinds: Development Tools and Productivity Tools.

6.a. *Development_Tool*. Entity Supertype. Development Tools assist in producing other programs/databases (e.g., a compiler, a design tool, a testing tool, a debugger);

6.b. *Productivity_Tool*. Entity Type. Productivity Tools help users perform general tasks such as word processing or spreadsheet analysis.

Figure 4. Model Model

7. **System**. Entity Type. Applications and databases require certain system software to exist in order to function. For example, if a user wishes to run a certain application on his/her machine, the user needs to make sure that it has the right (including version number) system software (e.g., Operating System (OS), Windowing System, and/or Database Management System (DBMS)) installed. Simple mistakes such as having the wrong version installed can lead to days of troubleshooting. If it is desired to place more emphasis on system software, then this entity type should be converted to a supertype whose subtypes could include Operating_System (e.g., MS-DOS, OS/2, AIX, MVS, VMS, UNIX, Macintosh), Windowing_System (e.g., MS-Windows, Presentation Manager, Motif, Macintosh), and DBMS (e.g., Oracle, Sybase, DB2, IMS, ADABAS). Since a repository will touch all aspects of an information system, deciding what not to include (or where to cut the level of detail) is essential to achieving a usable repository.
8. **Machine**. Entity Type. What was said of system software is also true of hardware. Software will only run on certain hardware, and in addition, may require a certain amount of memory, disk space, etc. From the point of view of system administration and maintenance, this entity type would have relatively more importance and could be decomposed according to machine category (Mainframe (e.g., IBM-3090, ES-9000), Workstation (e.g., RS-6000, Sun), or Personal Computer (e.g., PC Clone, Macintosh)) as well as component parts.
9. **Network**. Entity Type. Since the main purpose of the repository is to help users understand what information is available and how to access/use this information, some rudimentary information on machine and network connectivity will be helpful. For example, if a user discovers that a desired application is on machine x, the user would like to know how to make a connection to (i) run the application remotely, or (ii) remotely copy the application to his/her machine. Note that a comprehensive active repository could insulate the user from this distribution information by checking-out the application for the user.
10. **Business_Unit**. Entity Type. An important aspect in assessing the benefit of databases/applications is to determine what business units they service.

A summary report called *Entities and Their Descriptions* concisely describes all of the entity types for the WSRC Repository Prototype (see Appendix G). The relationships that entities participate in are as important as the information stored about the entities themselves. The relationship types are shown in the ER Diagram and are elaborated upon in the *Entity Model Information* report produced by Oracle's CASE Dictionary (see Appendix G). Finally, the attributes defined for each entity type are listed in another report (called *Entities and Their Attributes*) again produced by Oracle's CASE Dictionary (again see Appendix G). Note, the relational tables produced by the CASE Generator will contain additional attributes (or columns) for foreign keys and subtype indicators.

6.2.2. ER Model for the Database Entity Aggregate

The first expansion (from LEVEL 0 to LEVEL 1) will take the Database entity aggregate from the Macro ER diagram and expand it into its own ER Diagram. This design was patterned after the SQL2 standard for data dictionaries and the Oracle, Version 7 data dictionary. Issues of mainly a physical nature were ignored to simplify the design. The entity types for this diagram are listed below.

LEVEL 1: (for _Database -- explode its structure)

- Database
- Database Object
  - Table
  - View
  - Synonym
  - Index
  - Sequence
  - Cluster
  - Column
- Rule
  - Constraint
  - Trigger
- Privilege
  - System_Privilege
  - Object_Privilege
- Role
- Key_Column_Usage
  - Unique
  - Foreign
- Domain
- Schema
- Table_Space

The actual ER diagram is shown in Figure 5. The entity types in order of importance for the Database ER diagram are as follows:

1. **Database**. (Primary) Entity Type. A collection of database objects that taken together are useful for potentially broad application areas (e.g., a payroll database).

Figure 5. Database Model

2. **Database_Object**. Entity Supertype. A database object is a fundamental structure in a database. A database object is something created and stored in a database. Users are granted privileges to access/use database objects. Tables, Views, Synonyms, Indexes, Sequences, Clusters, and Columns are all types of database objects (Koch and Muller, 1993). Each database object has a name, a system-generated identifier, an owner, a creation time, and a time of last modification. Each of the subtypes inherit these attributes.

2.a. *Table*. Entity Type. The data themselves are stored in (base) Tables. These two-dimensional tables are made up of rows and columns.

2.b. *View*. Entity Type. Views (or virtual tables) allow the information in tables to be logically reorganized to met the needs of specific user groups.

2.c. *Synonym*. Entity Type. Synonyms are simply alternate names for database objects.

2.d. *Index*. Entity Type. Indexes provide a means for improving the performance of operations which access (based on attributes values) certain rows of tables.

2.e. *Sequence*. Entity Type. Sequences provide a means for generating values for system-defined keys.

2.f. *Cluster*. Entity Type. Clusters provide a means of storing together data from multiple tables, and can be used to enhance the performance of queries involving joins.

2.g. *Column*. Entity Type. Columns go together to form tables. Even though a column is a part of a table, columns are considered to be database objects, since for example privileges to access them can be assigned to users.

3. **Rule**. Entity Supertype. The SQL2 standard supports many types of constraints. Oracle, Version 7 and the upcoming SQL3 standard also support triggers. Quoting from (Spencer, 1993),

> "Declarative integrity constraint is a capability that helps to enforce simple business rules, such as ‘don’t pay nonexistent employees’. Prior to Oracle 7, users had to code these rules in the client application. The productivity benefit of declaring simple business rules, rather than having to code them, is immense."

3.a. *Constraint*. Entity Supertype. A constraint can be used to help validate data that is entered into the database, which helps maintain the integrity of the information. There are several types of constraints: Domain_Constraints, Referential_Integrity constraints, Table_Constraints, and general Assertions. These can be used to restrict attribute values to a given range, make sure corresponding values already exist in the database, or even check to make sure that an employee’s salary increase does not cause his compensation to exceed that of his supervisor. Other types of constraints are enforced when a column is declared to be a primary key (uniqueness and entity integrity (not null) constraints).

3.b. *Trigger*. Entity Type. A trigger has all the checking power of general constraints, and in addition, allows actions to be specified. Such actions could, for example, automatically insert data into a second table when a user attempts to do an insertion. Without this secondary insertion, the integrity of the database could be compromised. They can also be used to produce derived values.

4. **Privilege**. Entity Supertype. A privilege authorizes users to perform certain database operations.

4.a. *System_Privilege*. Entity Type. A system privilege authorizes users to CREATE, ALTER, and/or DROP database objects. This gives users the capability to define and fundamentally change the definitional structure of a database (i.e., use the Data Definition Language (DDL)).

4.b. *Object_Privilege*. Entity Type. An object privilege authorizes users to read (SELECT) and/or modify (INSERT, DELETE, UPDATE) existing database objects (typically tables and views). Read-only access is provided by the Query Language (proper) (QL), while the Data Manipulation Language (DML) can be used for modifications.

5. **Authorized_Role**. Entity Type. Rather than assigning a set of privileges to a particular user, it is more convenient to assign the set to a role and then assign roles to users. An ER diagram indicating how the Authorized Role entity type relates to other entity types is shown in Figure 6.

6. **Key_Column_Usage**. Entity Supertype. Access, relationships, and semantics are mainly manifest via certain columns (attributes).

6.a. *Unique*. Entity Supertype. A (unique) key is a column (or sometimes a combination of columns) that can be used to uniquely identify a row (or record). Since it is possible for a table to have more than one key, one is chosen as the primary key.

6.b. *Foreign*. Entity Type. A foreign key is a column (or in rare cases a combination of columns) that references a (unique) key (usually the primary key) in another table.

7. **Domain**. Entity Type. A domain is a set of values defined by the user. They can be used in place of general data types when defining columns.

8. **Schema**. Entity Type. A schema is simply a set of (typically interrelated) database object definitions. All objects in the schema must be owned by one user.

9. **Tablespace**. Entity Type. Most database objects require space for their storage. They are stored together in tablespaces. Although this is primarily an issue at the physical level (and thus of less concern to the repository), this information is included since related databases may be stored in the same tablespace. Thus, this information can provide a hint to the overall organization of databases within a DBMS.

Figure 6. Authorization Model

6.2.3. ER Model for the Application Entity Aggregate

The second expansion (from LEVEL 0 to LEVEL 1) will take the Application entity aggregate from the Macro ER diagram and expand it into its own ER Diagram. The design presented here was motivated by two guiding principles: (i) the design should be general enough to work for both 4GL's (e.g., PowerBuilder) and 3GL's (e.g., C/C++), and (ii) the design should have a macroscopic (modules) rather that microscopic (statements and variables) orientation. The entity types for this diagram are listed below:

LEVEL 1: (for _Application -- explode its structure)

- Application
  - Base_Application
  - Sub_Application
- Application_Keyword
- Application_Version
- Module
- Module_Version
- Data_Gateway
- Screen_Window
- Selector
- Application_Group

The actual ER diagram is shown in Figure 7. The entity types in order of importance for the Application ER diagram are:

1. **Application**. (Primary) Entity Supertype. Typically end-users will access information resources by running an application (program). Therefore, there will be a large community of users who will need to know about applications: what applications are available, where they are, and how to run them. It is also important for users to have a variety of ways to look-up and find an application that they are interested in. Consequently, there should be several attributes that facilitate look-up and assist the users in running the application (details could be found in manuals or on-line help).

Figure 7. Application Model

Application_Name (less than 30 chars)
e.g., Purchase_Orders
Short_Description (should fit on one line)
e.g., This application is used to request purchase orders
Long_Description (concise paragraph)
Application Category
e.g., Purchasing
Invocation_String
Description_of_Parameters
Description_of_Inputs
Description_of_Outputs

Finally, much of the information about applications will be in the form of relationship types to the following entity types:

Process_Model	(e.g., a DFD)
Data_Gateway	(e.g., gateway to a database view)
System	(e.g., requires >= DOS 5.0/Windows 3.0)
Machine	(e.g., requires >= 386 PC)
Document_File	(e.g., User's Manual)
User	
Authorized_Role	
Business_Unit	
Business_Policy	
Tool	
Application_Keyword	
Application_Version	
Screen_Window	
Selector	
Application_Group	

There are two important relationship types connecting Application to User: (i) a relationship type indicating who the responsible programmer is, and (ii) a relationship type indicating who is authorized to run particular applications.

1.a. *Base_Application*. Entity Type. Typically, when an application is run, a base screen (or window) appears. This screen will provide the user with options on what to do next (either perform an action or more likely select a sub-application).

1.b. *Sub_Application*. Entity Type. Designing applications to have sub-applications makes it easier for users to navigate to the specific action they want to perform.

2. **Application_Keyword**. Entity Type. This Entity Type is used to associate applications with keywords to facilitate flexible lookups. A similar facility might also be useful for other components of an information system (e.g., database tables).

3. **Application_Version**. Entity Type. To allow improvements (some of them experimental) to applications to be carried out without adversely effecting current operations, it is quite useful to allow different versions of the same application to exist simultaneously. It is to the version, rather than the application, that the details concerning the constituent modules (actually module versions) are attached. Some important attributes are the following:

   Version_Number
   Status (e.g., Test, Production)
   Creation_Date
   Last_Modification_Date

   The status of the application indicates whether it is, for example, a test or production version. The repository should also provide a means for promoting versions to a higher status (e.g., from Test to Production). As with applications, their versions also carry much of their information in the form of relationships, e.g., to Module_Version, to Executable_File, and to Application.

4. **Module**. Entity Type. A module is a component of an application. Actually, a module can be shared by many applications (i.e., there are shared and dedicated modules). A module may be entirely devoted to implementation or it may also contain externally visible definitions. In the latter case, changes to the module may have a large impact. In typical cases, a module may consist of an object class definition, method implementations, type/structure definitions, procedure/function definitions, procedure/function implementations, and screen/window specifications. (Examples of modules in PowerBuilder include Windows, Menus, DataWindows, Functions, and User Objects). As discussed before, if an organization wishes to put more emphasis on development, it might chose to model this entity type (Module) as an entity aggregate and explode its structure (see Appendix A).

5. **Module_Version**. Entity Type. As with applications, there can be simultaneous versions of modules. A specific module version is kept in a Source_File, and is attributable to a particular programmer/developer. (Note, using some source code control systems the source is not stored in its entirety, but can be reproduced on demand). The date of creation and date of last modification are recorded. The modifications are intended to be minor, since if it is a major modification, then it should produce a new version. These minor modifications can be documented solely within the Source_File. Major modifications that result in the creation of a new version will be documented in the repository itself. Thus, a complete history of major modifications can be produced from the repository (see Modification Tracking under Repository Services). To handle potential future legal situations, the meta-data about

module versions should be maintained even if a version is no longer in use. However, because of the relatively larger size of the actual source files, they may be relegated to archival storage (e.g., on tape). Minor modifications can also be tracked by examining relevant source files. Determination of what constitutes a major versus minor modification can be codified as a Business_Policy. (We have associated the compiler options with the module version. The main reason for doing this is to accommodate, for example, optimized versus non-optimized versions. In such cases, the same source file can be used to create different versions.)

6. **Data_Gateway**. Entity Type. This describes the mapping from the application (e.g., a PowerBuilder DataWindow) to relevant parts of a database (e.g., Oracle Tables and Views).
7. **Screen_Window**. Entity Type. Typically, each new sub-application selected will be in a new window, screen, or panel.
8. **Selector**. Entity Type. Selectors (menus or guided prompts) allows users to navigate down (from base-application to sub-application, etc.) to the appropriate level.
9. **Application Group**. Entity Type. An Application_Group is simply a set of applications that are interrelated enough to share modules. (In PowerBuilder, all applications in a group use the same pibble file.) This introduces a natural grouping of applications. Another form of grouping which is implementation independent is based on the Application_Category attribute (under item 1.). This allows applications to be grouped according to usage. Application in one category would tend to appeal to a common group of users.

6.2.4. ER Model for the ER_Model Entity Aggregate

This expansion (from LEVEL 0 to LEVEL 1) will take the ER_Model entity aggregate from the Macro ER diagram and expand it into its own ER Diagram. The model should be designed to match the capabilities of a group of related design tools. Ideally, it should follow a standard such as PCTE, so that modeling tools supplied by multiple vendors can interface with the repository. This is one area where the extensibility of the repository is particularly important. Since there is such a large variety of modeling techniques available (and there are likely to be many new ones), extensibility gives organizations the flexibility they need. For the WSRC Repository Prototype, however, to keep things simple and concrete, we will pattern our design on Oracle’s CASE*Designer. In actual practice, the model should be more general; specific capabilities matching a particular tool should be provided through specialized tables and/or views. The entity types correspond to those in the meta meta model with the extensions removed. (Although duplicating the capabilities of the meta meta model would be a logical choice, we explicitly chose otherwise to emphasize that they are in fact distinct; this also reduces the chances of becoming "level lost".) The entity types for this diagram are listed below:

LEVEL 1: (for _ER_Model -- explode its structure)

ER_Model
ER_Object_Type
    ER_Entity_Type
    ER_Relationship_Type
    ER_Arc
ER_Attribute
ER_Diagram

The actual ER diagram is shown in Figure 8.

Figure 8. ER Model Model

7. Techniques to Populate a Repository

Today's complex information systems are produced using a variety of systems/tools (DBMS's, 4GL's, GUI Builders, CASE Tools). The amount of meta-information needed to provide useful descriptions of the information resources and applications developed with these tools can be quite large. Fortunately, many of these tools maintain their own meta-data, so that a repository can acquire most of its information by extracting it from tool/system specific meta-data sources (e.g., data dictionaries, directories, CASE repositories/encyclopedias, library managers).

There are three basic approaches to storage organization:

1. **Fully Materialized Repository**. In this case all data accessible via the repository would actually be stored in the repository. Information would be extracted, scanned or directly entered. All of this information would be stored in a large unified (although potentially physically distributed) database. A couple of disadvantages to this approach are that (i) meta-data is now stored redundantly and (ii) as a by-product of redundancy, problems of consistency and timeliness arise. The use of triggers to automatically update the repository, for example, when a database table or view is created/altered, could reduce or eliminate the latter problem.
2. **Non-Materialized Repository**. This is the opposite extreme of the former case. The repository would store no information on its own, rather it would have several extract and transform modules that could take meta-data out of a system/tool specific source, transform them to a standard form, and combine them with meta-data from other sources, and finally present them as part of a query response. A couple of disadvantages of this approach are that (i) the sum total of meta-data from system/tool specific sources may be inadequate and must be supplemented, and (ii) some of the transforms may be too complex to yield an acceptable response time for queries and browsing actions.
3. **Partially Materialized Repository**. This approach can combine the best aspects of the previous two cases at the cost of increased complexity. The ultimate decision of which of the three to choose depends upon the needs and the set of systems/tools that a particular IS shop has.

In the WSRC Repository Prototype implementation, the meta-data are principally stored in Oracle tables. Consequently, the Oracle and Oracle CASE meta-data are easily extracted. Additional meta-data are extracted, scanned and directly entered. Although approach two saves space, it can become complex given the potentially large number of meta-data sources. Therefore, we have chosen the third approach, weighted toward being fully materialized. Most of the information would actually be stored in the repository, but some information would just be referenced in the repository, and actually stored somewhere else (source code files are a good example). Note that even if items are stored outside of the repository, it is still possible to exercise some control over them (e.g., the owner of certain files could be designated to be the repository).

There are five principal ways to populate a repository:

1. **Integrated Tools**. Some development tools (e.g., CASE tools) are designed to directly interface with repositories. They store and retrieve their models, schemata, and data definitions in a common repository. Some examples are: Cohesion tools interfacing with CDD/Repository, and KnowledgeWare ADW and IEW interfacing with R&O Rachade's Repository. In such cases, as databases or applications are designed, their meta-data are automatically recorded in the repository. In addition, the repository may apply consistency checking (of the new information itself and for conflicts with existing repository information).

2. **Extract**. With current technology, this is probably the preferred technique. It involves extracting and transforming data from other meta-data sources. The complexity of this technique depends on how close the structure of the system/tool specific meta-data matches the structure of the repository and whether or not compatible export facilities are provided with the systems/tools. For example, assuming a repository built on top of the Oracle RDBMS, it would be straightforward to extract meta-data from Oracle or DB2 (relational data model and SQL), a bit more challenging from Postgres (extended relational data model) or from ADABAS (network data model), and a bit more difficult yet from PowerBuilder or NATURAL 2. However, the existence of export utilities (e.g., PREDICT's export tool) can offload some of the task from the repository. In the WSRC Repository Prototype, we extract meta-data from Oracle, Oracle CASE, and PowerBuilder.

3. **Scan**. In some cases it may be necessary to obtain meta-data from sources that are not inherently of this character. For example, typing information and useful commentary are often found in C++ header source files. If an IS shop significantly utilizes 3GL programming, stylistic standards for header files should be enforced not only to make sure the necessary meta-data exist, but also to make these files easier to parse (both code and comments). Documentation files are an additional source of such information. Several repository vendors supply scanners for a variety of common business languages (e.g., COBOL, C/C++, Pascal, PL/I, Ada). Some vendors even supply parser-generators, so that an organization can create a new scanner by supplying the parser-generator with the grammar of the language they wish to scan. This technology can be extended to facilitate *reverse engineering* to create useful high-level information such as data or process models from detailed schemata or source code (McClure, 1992).

4. **Direct Manual Entry**. It may be the case that the meta-data existing in an organization (which can be collected into a coherent whole via the three techniques mentioned above) is inadequate. It will thus need to be supplemented. One way to do this is to enforce a policy requiring IS personnel to enter specific types of comments into the fields provided with most systems/tools (e.g., Oracle and PowerBuilder provide these). The necessary

information will then show up in the system/tool specific meta-data so the repository can extract it. If this is still found to be inadequate, then commentary can be directly entered into the repository. For example, general organizational information such as business rules and goals might need to be entered directly into the repository.

5. **Resource Tracking Utility**. An interesting new development is resource tracking software. These utilities automatically scan a network to keep track of hardware -- including components such as keyboards, memories, boards, and disk drives -- and application and system software. One such utility, Apsylog, Inc.'s PC Galaxy collects this information and stores it in an SQL database (Willett, 1993).

More information on populating repositories may be found in the thesis by (Krishnan, 1994).

8. Use of a Repository

Having discussed what kinds of information are stored in a repository and how to populate the repository with (meta-) data, it is important next to consider how a repository will be used. We will decompose this discussion into three issues: (i) the form and function of user interfaces to the repository, (ii) the basic services that a repository suitable for WSRC should provide, and (iii) the advanced services that can provide value-added capabilities, at a cost. The WSRC Repository Prototype will provide at least a partial implementation of all of the basic repository services. Closely related to these issues are those on the management and administration of the repository (see the next section).

8.1. User Interfaces

A repository should include several interfaces. For users it should provide for query-based access, presumably SQL or some extension of it. An object-oriented query language is another viable possibility. The repository should also provide an easier to use GUI-based interface. In addition, it will need to provide interfaces to software, both to extract meta-data from software and to provide information to programs.

Let us consider an example in which an end-user is attempting to find a suitable application for the task at hand. Suppose the end-user wants to find an application dealing with purchase orders. The submission of an SQL query like the following would provide the answer.

```
SELECT Application_Name, Short_Description
FROM Applications
WHERE Application_ID IN
   (SELECT Application_ID
    FROM Application_Keywords
    WHERE Keyword = ’Orders’);
```

Note that Oracle’s CASE*Generator adds an ’s’ onto the end of Entity Types when producing relational Tables (e.g., Application --> Applications). This query will retrieve the names and short descriptions of all applications (e.g., coded in PowerBuilder or NATURAL 2) that have ’Orders’ as one of their keywords. Options need to be provided to make this keyword lookup case insensitive. Closely related to the facility for keyword lookup are the facilities provided by (i) a glossary (table defining standard terminology), (ii) a thesaurus (table associating common terms), and (iii) a table of standard abbreviations.

Inquiries such as the one shown above would be even easier, if a Form or PowerBuilder Application were provided to allow users to either enter a few keywords in a dialog box or select from a scrollable list. In addition, navigational browsing facilities should be provided, so users can determine what is related to what. For example, a user could browse through applications in some Application Category, and upon finding a interesting application, select it to show its attributes, or follow its relationships to discover what it is associated with (e.g., what tables it accesses). It would be nice if this navigation could be guided by an ER or OO model diagram. Also, an AI tool that takes into consideration the preferences of the user to better tailor and simplify the browsing would be quite useful.

8.2. Repository Services

In addition to using a repository as a general source of information, via browsing and querying as discussed above, it should also provide certain key services.

1. **Impact Analysis**. When a change to an information resource or application is contemplated, its impact should be assessed. This is most likely to occur when one needs to change a base table or some module making up an application. Note that there are standard mechanisms to minimize the impact of changes. The changes to database base tables can be minimized if all applications and users reference views rather than base tables (taking advantage of logical data independence). Still, some changes to base tables will force changes to views, and therefore the changes will effect both applications and users. Modifications to application modules for the purpose of performance enhancement or debugging need not have major impacts, since they need not change any definitions of data structures, procedures or classes. Changes that do effect definitions, need to be carefully considered. For example, if a module is changed, should this effect current application versions, or result in a new version of the entire application. In any case the affected user community should be notified (if not consulted beforehand). Whether or not a module contains definitions that are visible from outside the module is indicated by an attribute of the module entity type.

2. **Change Notification**. When changes are made that affect the user community, notifications should be sent. Notification messages can be attached to applications, tools, or put in users’ electronic mailboxes. If a change notification message is attached to an application, when a user runs the application, the message would be displayed informing him/her of the change (e.g., a bug was discovered and fixed).

3. **Modification Tracking**. Change histories should be maintained for both information resources and applications. Let us consider how this might apply to a typical database table. From a definitional point of view (DDL statements), we would have a complete history of who (and when) CREATEd and who (and when) ALTERed the definition of a table. Furthermore, if the table has been DROPed from an operational database, we can record where it is backed up on tape, or how it is mapped into the information warehouse. From a data instance point of view (DML statements), we could maintain a complete history of all INSERTs, DELETEs, and UPDATEs to the table. Many organizations may find that

keeping track of changes to data instances (which is data not meta-data) is too costly, and may prefer to use a utility that extracts this information from a DBMS's Log on demand. Once data has migrated into the warehouse, modification tracking can be thought of as maintaining the geneology of data, since summarized data can come from multiple sources (in other words have more than one parent). Because of the lower frequency of changes in the warehouse, instance tracking would be less costly.

4. **Configuration Management**. It is not feasible for a user to have complete knowledge of all files and module versions that compose an application. Configuration management tools help remedy this problem by providing an abstracted view of building applications to the user in that only the application name and version number are required to build an application. Developers will store within the repository a hierarchical system model for a given application version resembling a makefile dependency graph. Therefore, the repository provides a way to create links between different configurations. It is through cross-references between source files, include files, header files, libraries, etc. and their relative last update times that automatic builds are possible. Another useful automation is the resolution of files residing in different directories or even on different host machines. This approach to configuration management is referred to as the composition model approach.

5. **Version Management**. The primary problems to be addressed here are: How to collect the appropriate module versions together and relate them to an application version? How can this version mapping be automated or simplified for the user? How can this facility support both Configuration Management and Modification Tracking? Version Management can extend into the development process; however, we have broken this off as a separate advanced repository service called Version Control (see below).

6. **User Authorization**. The question of authorization decomposes into two sub-issues: The first issue concerns access to the repository contents, while the second more complex issue concerns access to resources referenced by the repository.

6.a. Access to the repository contents could be patterned after the authorization mechanisms provided by Oracle, Version 7, since they are very flexible and powerful. Users are assigned roles which indicate what privileges they have. Privileges are divided into two categories: system privileges and object privileges. System privileges specify the definitional (DDL) capabilities that users have (i.e., to CREATE, ALTER, and DROP database objects), while object privileges specify the operations that users can perform on databases objects (e.g., INSERT, DELETE, UPDATE, and SELECT). Authorization to access repository contents could be applied to applications as well as users. One way to do this would be to assign a default role to an application. A more flexible alternative would to define a maximal role for the application and intersect its privileges with the privileges of the user currently running the application. If the maximal role was unlimited, then the application would simply inherit the role of the user.

6.b. Authorization to access resources referenced by the repository such as database tables or application programs is complicated by the fact that authorization mechanisms are built into database management systems and operating systems. The facilities provided by DBMS's are typically quite good and should suffice without being supplemented. Still, it would be convenient for managers to be able to go to a central facility to determine what a particular user is allowed to access/do. Therefore, it may be useful to extract such information from the DBMSs' data dictionaries/catalogs, even though the enforcement would be left up the DBMSs'. Access to applications can be controlled (to some extent) by operating systems and networking software. In many cases this control is ad hoc (if not inadequate), so some organizations may wish to institute greater control, by requiring that applications be checked-in/checked-out of the repository. Unfortunately, complete enforcement of this in a large client-server environment would be quite a challenge, so they may wish to restrict this control to just sensitive and classified applications.

7. **Audit Support**. The data stored in an information system must be recorded, processed, and reported properly. In order to ensure the integrity of the system, auditing standards and controls are designed to perform verification and validation of data, monitor proper access to stored data, and deal with access violations. Internal auditors can use the repository during the systems development phase to ensure that the standards and conventions agreed upon are adhered to by system designers. Some examples of such uses are naming conventions and standards, authorized users, and authorization levels. Most DBMS products offer a logging facility to record transactions to their databases. Including this logged information into the repository makes the job of auditing across several DBMS's easier. (Because of the size of the log and the fact that it is constantly updated, rather than being in the repository proper, it would be more efficient to simply reference the log from the repository and use utilities to extract desired information.) This ensures an "audit trail" so that if there is a need to re-create an action or file, there is enough information to do so. Another method to examine processing is to process test data and examine the test results to ensure that they match the expected results.

8. **Subscription Service**. Reports circulated on a monthly (or weekly) basis are produced from information in operational databases and the warehouse. The repository can record who gets what kinds of reports and when. This information can be recorded as business policies (ON event DO action). DBMS's such as Oracle, Version 7 can automate this by implementing the business policy as a trigger. This facility would be similar to that provided by DEC's Information Catalog.

8.3. Advanced Repository Services

The services listed here are possible considerations for future inclusion. At the present time their costs probably outweigh their benefits. Some will require repository-readiness to be built into software supplied by vendors.

1. **Usage Analysis**. To determine the value of information and applications it is useful to know who uses them and how often they are used. Such information can be maintained in relationships that cross-reference users with the applications they use, applications with the information resources they use, and users with the information resources they use. A frequency attribute on each of these relationships can record the number of times each such interaction occurs. As an alternative every individual interaction could be recorded, but it seems that the summary information should suffice. Obtaining detailed usage information can be quite difficult, particularly for database tables. This would require that every database access (e.g., using Brio's DataPrism) either shoot through the repository or produce a side-effect transaction to update the repository. On the other hand, usage statistics for applications could be maintained if users were required to check-out/check-in copies of applications from the repository. The same could be said for productivity tools like DataPrism.
2. **Type Management/Transformation**. Although more for developers, type management/transformation can be helpful in two respects. The general idea comes into play when a data type or structure definition needs to be used by several developers and in different languages. For example, if an Oracle table needs to be accessed in Pro*C or in PowerBuilder's script language, the repository can assist in this process. By automatically translating and inserting the appropriate code, development can be sped up and made more error-free. However, to provide complete consistency checking on types would require that the Module entity type be decomposed into its components (see Appendix A).
3. **Version Control**. This repository service is again for developers. To control a team-oriented approach to application development, version control is essential. Developers can check-out module versions or even sub-components of modules. If a version is checked-out in update mode, then typically it is locked so that no one else can modify it until it is checked back in. Alternatively, complex merge algorithms can be applied. On the other hand, if a version is not checked-out for update, then any authorized developer should be able to check-out a version in read-only mode. Because of its impact on current development practices, an organization may prefer to simply use a source-code control system on an as needed basis.

More information on repository services may be found in the thesis by (Edwards, 1994).

9. Repository Administration

The person responsible for managing, controlling, and maintaining the repository will need to have had experience in both data administration and management of application development. The repository administrator could be a new full-time position or additional duties added to a data administrator. At times the workload will be intense, so that large chunks of time must be allocated to the repository administration duties.

The repository administrator must *enforce policies* regarding security, privacy, and protection. They must also enforce policies and procedures for designing and building databases and applications, such as entering necessary comments when a developer registers his/her program with the repository. (One way to enforce this is to tell developers that they will be evaluated by examining what they have registered in the repository. If the prerequisite information is not entered, his/her program will not be registered).

Some of the other important specific duties of the repository administrator are enumerated below.

1. **Managing the Meta Model**. The repository administrator will be responsible for maintaining the repository meta model (or RIM). Any extensions or changes to the RIM must be done with care since their effects can be far reaching. Because RIM's are typically large and complex, extensions should be infrequent and well planned. It may be desirable at times to utilize a consultant from the repository vendor to assist in this process.

2. **Maintaining Completeness and Consistency**. The repository administrator will be responsible for making sure new entities (purchased software, applications developed, databases designed) are entered into the repository. Ultimately, the administrator will be responsible for the overall completeness and consistency of the repository. Automated and manual validation checks should be performed periodically. This supplements the built-in validation that occurs when items are entered into the repository. Manual validation could be done in a sampling (or spot check) fashion.

3. **Registering Tools**. Adding a new application to the repository can be done in a variety of ways. However, the addition of some new tools (e.g., a design tool) will require the expertise of the repository administrator. Often this will require a new view (a virtual repository table) be created particularly for that tool (or tool class). In some cases it may even require an extension to the base tables.

4. **Enforcing Naming Standards**. The repository administrator will be responsible for making sure naming conventions and standards are followed. The administrator must also be involved in setting these conventions and standards.

5. **Training of Users**. The repository administrator will be responsible for organizing and scheduling the training of repository users (see below under User Impact Assessment).

6. **Optimizing Effectiveness**. The repository administrator will be responsible for monitoring the success of the repository. The administrator will need to determine to what degree the repository is being used, and if it is not, determine why it is not being used (e.g., is it too hard to use, are the productivity benefits for a particular user group too small, or is additional training called for). Furthermore, if the performance of the repository is insufficient, the administrator is responsible for taking measures to tune its performance.
7. **Defining the Role of Repository**. The repository administrator should define the the role of the repository within the organization: who can use it; who should it; who must it. The role of the repository is likely to increase over time: its scope is likely to widen, its level of detail is likely to increase, and its degree of control over IS activities is likely to increase. The repository administrator needs to establish both short-term and long-term goals and objectives for the evolving role of the repository within the organization.
8. **Authorizing Users**. The repository administrator will be responsible for authorizing access to the repository. The degree of access should be dependent on the needs of the user as well as his/her training.
9. **Distributing Components Among Servers**. If an organization chooses a distributed repository, then the repository administrator must determine where to place parts of the repository. Efforts should be made to maximize the availability and reliability of the repository.
10. **Controlling Concurrent Access**. The repository administrator will be responsible for setting the policy on concurrent access to the shared repository. Conventional database transaction management (e.g., two-phase locking) will suffice in many cases. However, if elements are checked out for update for an extended period of time, then concurrency control techniques suitable for long-duration transactions (e.g., version control) need to be utilized.
11. **Overseeing Back-up and Recovery**. The repository administrator will be responsible for scheduling back-ups of the repository. The underlying database technology can be used to facilitate this. These aspects of repository management are particularly important because of the vital nature of the information contained in the repository.

9.1. User Impact Assessment

The impact on the user community should be gradual. Negative factors can be reduced by appropriate training, support, policies, and incentives. These measure should be instituted to maximize the ultimate effectiveness of the repository. A repository seen in its entirety can be quite complex. However, if groups of users can be trained on a need to know basis, this complexity need not be prohibitive. End-users will not be directly modifying repository contents (although their actions may produce automatic updates). The training of such users would be concerned with how to effectively navigate the repository structures to find information of interest to them. Some of the more sophisticated end-users may also need to be trained in how to execute certain repository services. Designers and developers would need additional training since they would have a more direct effect on repository contents. The names (for applications or database objects) they choose and the comments they provide must be done according to corporate policies. Minimal or misleading commentary can greatly diminish the utility of the repository.

10. Summary

Repository technology can provide organizations significant benefits. Because of the cost and comprehensive nature of this technology, any procurement of repository technology should be done very carefully. Two types of procurement are reasonable choices: (i) The easiest choice would to purchase a complete solution from a repository vendor. (ii) An alternative is to purchase a partial solution from a database vendor who has supplemented its DBMS with tools to support integration and interoperability. This alternative would be cheaper, but would require in-house development to provide full functionality. (Complete in-house development would be very difficult.)

Repository technology will eventually affect the way most users interact with their systems. As an organization migrates the repository to a more active role, the repository will increasingly impose itself upon the user community. Therefore, it is important that this technology be phased in over time, with ongoing feedback from the user community as well as top management. In the early stages the primary efforts will revolve around populating the repository and carrying out pilot studies with different types of user groups.

Some of the expected benefits of introducing repository technology are as follows: (i) information about corporate data processing assets would be accessible to the user community; (ii) duplications and inconsistencies could be reduced; (iii) end-users would have the information that they need to make them more independent and productive; (iv) designers and developers would have a guidepost to current applications, their component modules, and relevant design models, all of which would help in software maintenance efforts as well as new development; and finally, (v) IS manager/administrators will also find this information useful.

11. References

Augusto Argento, Chiara Bonferini, Fabrizio Demantte and Serena Manca, 1993. *ECMA PCTE, CORBA and ATIS: the Easy Way to O-O ECMA PCTE Services*, Software Development Technologies, DEC, Italy, 1993.

J.M. Artim, J.M. Hary and F.J. Spickhoff, 1990. "User Interface Services in AD/Cycle," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 236-249.

Elias M. Awad and Malcolm H. Gotterer, 1992. *Database Management*, Boyd & Fraser Publishing Company, Danvers Massachusetts, 1992.

Carlo Batini, Steffano Ceri and Shamkant B. Navathe, 1992. *Conceptual Database Design: An Entity-Relationship Approach*, The Benjamin/Cummings Publishing Company, Redwood City, California, 1992.

Carlo Batini, Giuseppe Di Battista and Giuseppe Santucci, 1993. "Structuring Primitives for a Dictionary of Entity Relationship Data Schemas," *IEEE Transactions on Software Engineering*, Vol 19, No 4, April 1993, pp. 344-365.

Ken S. Brathwaite, 1988. *Analysis, Design, and Implementation of Data Dictionaries*, McGraw-Hill Book Company, New York, NY, 1988.

Chris Brown, 1993. "The Role of a Repository in the Adoption of Object-Oriented Techniques & Databases," *CASE Trends*, July 1993, pp. 45-60.

BrownStone Solutions, Inc., 1993. *Client/Server Repository Strategy*, BrownStone White Paper, 1993.

Peter P. Chen, 1976. "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol 1, No 1, March 1976, pp. 9-36.

Minder Chen and Ronald J. Norman, 1992. "A Framework for Integrated CASE," *IEEE Software*, March 1992, pp. 18-22.

G. Chroust, H. Goldman and O. Gschwandtner, 1990. "The Role of Work Management in Application Development," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 189-208.

David Clarke, Keith Matthews and John Pratt, 1992. "The Engineering Database," *ICL Technical Journal*, May 1992, pp. 39-53.

Chris J. Date, 1990. *An Introduction to Database Systems*, Vol. 1, Fifth Edition, Addison-Wesley Publishing Company, Reading Massachusetts, 1990.

DEC, 1991. *COHESION Environment for CASE: A Single Environment for All Your Development Needs*, Digital Press, 1991.

M.E. Dewell, 1990. "Cross System Product Application Generator," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 265-273.

Bonnie M. Edwards, 1994. *Implementing and Evaluating Common Repository Services*, Masters Thesis, March 1994.

Ramez Elmasri and Shamkant B, Navathe, 1989. *Fundamental of Database Systems*, The Benjamin/Cummings Publishing Company, Redwood City, California, 1989.

Eric Felton and Eric Soutter, 1992. "ICL's ICASE Products", *ICL Technical Journal*, May 1992, pp. 23-38.

Per O. Flaatten, Donald J. McCubbrey, P. Declan O'Riordan and Keith Burgess, 1992. *Foundations of Business Systems*, Second Edition, Anderson Consulting, The Dryden Press, Orlando, FL, 1992.

Alan Goldfine, 1993. *Information Resource Dictionary System Requirements Specification Standing Document*, NIST, February 1993.

Elke Gronert, 1993. "German Repository Takes on U.S. Market," *Software Magazine*, May 1993, pp. 111-113.

Gupta, 1993. *SQLWindows*, Gupta Corporation Sales Literature, 1993.

Mary Hanna, 1993. "Can CASE Bridge to Object World?" *Software Magazine*, July 1993.

William Harrison, Harold Ossher and Mansour Kavianpour, 1992. *Object Oriented Tool Integration Services (OOTIS): OOTIS Tool Integration Model - IBM AIX-CASE Proposal*, IBM, June 1992.

William Harrison, Harold Ossher and Mansour Kavianpour, 1992. "Integrating Coarse-Grained and Fine-Grained Tool Integration," *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE'92)*, July 1992, pp. 23-35.

K.P. Hein, 1990. "DevelopMate: A New Paradigm for Information System Enabling," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 250-264.

D.M. Hembry, 1990. "Knowledge-Based Systems in the AD/Cycle Environment," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 274-286.

Lilian Hobbs and Ken England, 1993. *Digital's CDD/Repository*, Digital Press, 1993.

Robert E. Hodges, 1992. *Repository Context <Information Resource Dictionary System (IRDS)> Reference Model Technical Report*, NIST, September 1992.

G.F. Hoffnagle and W.E. Beragi, 1985. "Automating the Software Development Process," *IBM Systems Journal*, Vol 24, No 2, 1985.

Simon Holloway (Ed.), 1989. *The Future of Data Dictionaries*, The British Computer Society Database Specialists Group, Gower Publishing Company, Brookfield, Vermont, 1989.

Cheng Hsu, M'hamed Bouziane, Laurie Rattner and Lester Yee, 1991. "Information Resources Management in Heterogeneous, Distributed Environments: A Metadatabase Approach," *IEEE Transactions of Software Engineering*, Vol 17, No 6, June 1991, pp. 604-625.

Judith S. Hurwitz, 1993. "The Changing Development Paradigm: The Movement to Client/Server Computing," Hurwitz Consulting Group Report on CDE, 1993.

IBM, 1991. *Repository Manager/MVS Release 2: Guide for Users and Administrators*, IBM, March 1991.

IBM, 1991. *Repository Manager/MVS Release 2: Repository Modeling Guide*, IBM, March 1991.

William H. Inmon, 1991a. *Database Machines and Decision Support Systems: Third Wave Processing*, QED Information Sciences, Inc., Wellesley Massachusetts, 1991.

William H. Inmon and Susan Osterfelt, 1991b. *Understanding Data Pattern Processing: The Key to Competitive Advantage*, QED Technical Publishing Group, Boston Massachusetts, 1991.

William H. Inmon, 1992. *Building the Data Warehouse*, QED Technical Publishing Group, Boston Massachusetts, 1992.

Michael H. Kay, 1992. "The Architecture of an Open Dictionary," *ICL Technical Journal,* May 1992, pp. 85-107.

Warren Keuffel, 1993. "Paradigms in Configuration Management," *Software Development*, Sept 1993.

George Koch and Robert Muller, 1993. *ORACLE7: The Complete Reference*, Osborne McGraw-Hill, Berkeley, CA, 1993.

Krys J. Kochut, John A. Miller and Walter D. Potter, 1991a. "Design of a CLOS Version of Active KDL: A Knowledge/Data Base System Capable of Query Driven Simulation," *Proceedings of the 1991 AI and Simulation Conference*, New Orleans, Louisiana, April 1991, pp. 139-145.

Krys J. Kochut, John A. Miller, Walter D. Potter and Anthony D. Wright, 1991b. "h-KDL: A Historically Extended Functional Object-Oriented Database System," *Proceedings of the Tools '91 International Conference,* Santa Barbara, California, July 1991, pp. 73-86.

Sunderraman Krishnan, 1994. *Building and Populating a Prototype Repository*, Masters Thesis, March 1994.

Margaret Henderson Law, 1988. *Guide to Information Resource Dictionary System Applications: General Concepts and Strategic Systems Planning*, NBS Special Publication 500-152, 1988. (NBS is now NIST)

Salvatore T. March and Young-Gul Kim, 1992. "Information Resource Management: Integrating the Pieces," *Database,* Vol. 23, No. 3, 1992, pp 27-38.

R.W. Matthews and W.C. McGee, 1990. "Data Modeling for Software Development," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 228-235.

Edie McClure, 1992. *The Three R's of Software Automation: Re-Engineering, Repositories, and Reuse,*

V.J. Mercurio, B.F. Meyers, A.M. Nisbet and G. Radin, 1990. "AD/Cycle Strategy and Architecture," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 170-188.

John A. Miller, Walter D. Potter, Krys J. Kochut, Ali A. Keskin and Ender Ucar, 1991a. "The Active KDL Object-Oriented Database System and Its Application to Simulation Support," *Journal of Object-Oriented Programming,* Special Issue on Databases, Vol 4, No 4, July-August 1991, pp. 30-45.

John A. Miller, Walter D. Potter and Krys J. Kochut, 1992b. "Knowledge, Data, and Models: Taking an Objective Orientation on Integrating these Three," *IEEE Potentials* Vol 11, No 4, December 1992, pp. 13-17.

Huw Oliver, 1992. "A Comparison of ISO-IRDS and PCTE," Hewlett-Packard Technical Report, 1992.

Oracle, 1992a. *Guide to Oracle Products and Services*, 1992.

Oracle, 1992b. *CASE Technical Overview*, 1992.

E. Sanders Partee, 1991. "The Road to Repository," *Data Base Management*, Vol 1, No 3, April 1991.

David Plotkin, 1992. "Selecting a Repository," *Database Programming & Design*, Vol 5, No 4, April 1992.

Walter D. Potter, Robert P. Trueblood and Carol M. Eastman, 1989. "Hyper-Semantic Data Modeling," *Data & Knowledge Engineering*, Vol 4, 1989, pp. 69-90.

Walter D. Potter, Terry Anthony Byrd, John A. Miller and Krys J. Kochut, 1992. "Extending Decision Support Systems: The Integration of Data, Knowledge, and Model Management," *Annals of Operations Research,* Vol 38, 1992, pp. 501-527.

Walter D. Potter, Krys J. Kochut, John A. Miller, Veena P. Gandham and Ramakrishna V. Polamraju, 1993a. "The Evolution of the Knowledge/Data Model", *International Journal of Expert Systems: Research and Applications*, Special Issue on AI and Databases, Vol 6, No 1, 1993, pp. 39-81.

Walter D. Potter, John A. Miller and Krys J. Kochut, 1993b. "A Hyper-Semantic Approach to Intelligent Information Systems," *Integrated Computer-Aided Engineering*, (to appear, 1993).

J.M. Sagawa, 1990. "Repository Manager Technology," *IBM Systems Journal*, Vol 29, No 3, 1990, pp. 209-227.

Amit P. Sheth and James A. Larson, 1990. "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys,* Vol 22, No 3, Sept 1990, pp. 183-236.

Alan R. Simon, 1993. *The Integrated CASE Tools Handbook*, Van Nostrand Reinhold, NY, 1993.

Software AG, 1989. *PREDICT CASE: Concepts and Facilities*, Software AG, 1989.

John F. Sowa and John A. Zachman, 1992. "Extending and Formalizing the Framework for Information Systems Architecture," *IBM Systems Journal*, Vol 31, No 3, 1992, pp. 590-616.

Ian Sommerville, 1992. *Software Engineering*, Fourth Edition, Addison-Wesley Publishing Company, Reading Massachusetts, 1992.

Gordon Spencer, 1993. "Making a Stronger Server: Oracle Claims its Implementations Do More than Simply Catch up to Competitors," *SunWorld*, October, 1993, pp. 43-46.

Adrienne Tannenbaum, 1992a. "Implementing a Corporate Repository," *Data Base Management*, Feb 1992.

Adrienne Tannenbaum, 1992b. "LAN-Based Repositories," *CASE Trends*, Nov 1992.

Adrienne Tannenbaum, 1993. "U.S. Market Seeing Repository Rebirth," *Software Magazine*, June 1993, pp 43-61.

Toby J. Teorey, Guangping Wei, Deborah L. Bolton and John A. Koenig, 1989. "ER Model Clustering as an Aid for User Communication in Database Design," *Communications of the ACM*, Vol 32, No 8, August 1989, pp. 975-987.

Ian Thomas, 1989. "PCTE Interfaces: Supporting Tools in Software-Engineering Environments," *IEEE Software*, November 1989, pp. 15-23.

A. Kenneth Thompson, 1992. "CASE Data Integration: The Emerging International Standards," *ICL Technical Journal*, May 1992, pp. 54-66.

Graham Thompson, 1993a. "Repository-Driven Client/Server Development: Part I," *Data Management Review*, April 1993.

Graham Thompson, 1993b. "Repository-Driven Client/Server Development: Part II," *Data Management Review*, June 1993.

Graham Thompson, 1993c. "The Need for CASE Integration," R&O, Inc. Technical Report, April 1993.

Graham Thompson, 1993d. "Empowering Your Organization To Manage Its Information Assets," R&O, Inc. Technical Report, October 1993.

Charles J. Wertz, 1989. *The Data Dictionary: Concepts and Uses*, Second Edition, QED Information Sciences, Inc., Wellesley Massachusetts, 1989.

Shawn Willett, 1993. "Galaxy Automatically Tracks Network Resources and Updates a Database," *ComputerWorld*, 1993.

WSRC, 1992. *ISE System Development Methodology Manual (U)*, WSRC Technical Report, February 1992.

WSRC, 1993. *Applications Design Architecture (U)*, WSRC Technical Report, September 1993.

John A. Zachman, 1987. "A Framework for Information Systems Architecture," *IBM Systems Journal*, Vol 26, No 3, 1987, pp. 276-292.

Appendix

A. Repositories from an Application Development Perspective

Much of the design presented in the body of this report, will be quite useful to developers, particularly the information on applications and their modules. In this section, we will explore what additional features could be added to further benefit application developers. In terms of the Repository Information Model (RIM), there are two avenues that could be pursued. The first involves drilling down on the application side at least one more level. Rather than stopping at the module level, a module should be treated as an Entity Aggregate and an ER diagram revealing its sub-structure should be produced. The second involves introducing CASE technology.

A.1. Components of Modules

A second level expansion (LEVEL 1 to LEVEL 2) will take the assumed Module entity aggregate and expand its own ER Diagram (see Figure 9). A Module can contain both Definitional and Implementation components; however, it is good programming style to separate these these into different modules. As mentioned before the changes to definitions are likely to have significant impacts, while changes to implementations will not.

- Module
    - Definition
    - Implementation
- Constant
- Type
    - Record_Type
    - Array_Type
    - Class
    - Primitive_Type
- Component
    - Field
    - Method
- Procedure
    - Function

Literal constants should not be used in programs, but rather Constant Definitions should be used. If at some point the constant needs to changed, only one line of source needs to be modified.

Figure 9: ER Diagram for Module

Data Type Definitions specify how data will be organized in application programs. They are used to specify the structure of records, arrays, or variables of primitive (built-in) type. Records are made up of their constituent fields. In a systems programming environment, additional types would include pointers, unions, and enumeration types. In addition, object-oriented languages allow object-types (or classes) to be defined. If a developer is accessing records stored in a database or produced by another application, the developer need not worry about defining the data structure, since the repository can automatically generate it for him/her (assuming a language supported by the repository is used).

Class Definitions are used to define the data type for objects. Objects integrate both structural (data elements or fields) and behavioral (methods) aspects. Object-oriented programming is based on the following four ideas: (i) Encapsulation - the data elements and methods are encapsulated within the class and often the data elements are made private so they can only be accessed via methods. (ii) Message Passing - a program becomes a collection of objects which conceptually communicate by send sending messages to each other (e.g., `stack1.push(5)`, is a message to the object called `stack1` to `push` the value `5`). (iii) Inheritance - new derived classes can be built rapidly from existing classes by using inheritance (the new class inherits the facilities of the existing class, and can then modify or add to these facilities); under single inheritance a class can inherit from only one parent class, while this restriction does not apply under multiple inheritance; a collection of classes related by inheritance will form an inheritance hierarchy or lattice. (iv) Polymorphism - when a message such as `push` is sent out, a method to respond to the message needs to be found; the particular method, though, is dependent on the type of the object to which the message was sent, which may not be known until run time.

Procedure Definitions should tell users (callers) of the procedure all they need to know to use the procedure in their program. It should include the procedure name, its parameters, and its return value, as well as comments to describe its purpose and functionality. (Note, a function is a special-case of procedure, one that returns a value. Some definitions also require that a function have no side-effects. Finally, in C/C++ a procedure is a special-case of function, a function returning void.)

Long and detailed code is the province of the implementation modules. While the procedure and method definitions are given in definition modules, their implementations or bodies are coded in the implementation modules.

A.2. CASE Technology

For organization committed to the application of CASE technology, repositories provide the foundation for CASE tool integration. This technology assists the developers during the entire process of software development from requirements analysis, to design, to implementation, to testing, to installation, and finally, to maintenance. All (or part) of the software development lifecycle can be supported by the repository. The repository should maintain and control requirements documents, diagrams for analysis and design, (e.g., ERD's and DFD's), software testing information, as well as information on applications and their sub-components. To maximize an organization's flexibility we can divide this into two subcomponents: CASE for Database Design and CASE for Application Development.

CASE for Database Design:
- Requirements Documents
- Data Models
  - ER Diagrams
- Test Queries and Transactions

CASE for Applications Design:
- Requirements Documents
- Design Documents
  - DFD Diagrams
- Test Programs

One of the advantages of coupling CASE and repository technology, is that a greater percentage of the repository can be automatically populated. Furthermore, tool interfaces to the repository can provide additional constraint and consistency checking. For example, when a model is checked into the repository, it can be thoroughly checked and rejected if it does not meet certain criteria. Additional population techniques would typically also be required. For example, scanners or manual entry might be needed if preexisting or purchased software is to be included in the repository. However, those in the CASE camp might argue that heavy reliance on a pot-pourri approach to populating the repository might leave it in an incomplete and inconsistent state.

Several CASE vendors are working out standards that will make it so their tools will interoperate. Repositories will be the place to hold the models produced by multiple vendors. As projects are completed and applications developed, the tools will automatically (with consistency checking) be updating the repository. Consequently, when applications are ready for production use, vital information is available on-line. This information can be used by end-users, application developers, and project managers. The end-users can use it to search for an application that meets their needs as well as figure out how they can access and run the application. The repository should be able to address the basic what, where, and how questions typical of end-users. Developers will find the repository to be a vast storehouse containing most of the information of interest to them, e.g., their models, application programs, modules, submodules, etc. Project managers may also find this information valuable, allowing them to schedule and track application development as well as monitor the usage of applications. Both IBM and DEC have integrated development environments (Application Development Platform (ADP) in the case of

IBM, and Cohesion in the case of DEC) built using repository technology as their centerpiece.

Further insight into the issues involved when using CASE technology can be gained by decomposing CASE into its major subdivisions: UpperCASE (the front-end the lifecycle) and LowerCASE (the back-end of the lifecycle).

1. **UpperCASE vs. LowerCASE**. This relates to which phases of the software life-cycle an enterprise is interested in. The major subdivisions here are referred to as UpperCASE and LowerCASE. UpperCASE refers to the front-end part of the software life-cycle (up to and including design), while LowerCASE refers to the back-end (implementation through maintenance). Many CASE vendors as you might naturally expect are primarily concerned with UpperCASE. They want to be able to import/export their models (data models and process models) to/from a repository. To achieve multivendor interoperability they have developed ongoing standards that their tools should conform to (e.g., Portable Common Tool Environment (PCTE)), as well as standard formats for import/export files (Case Data Interchange Format (CDIF)).

Before leaving this section, we should comment on another approach to decomposing a repository. If planning, analysis and design are points of focus in an organization, it may wish to adapt Zachman's Information System Architecture (ISA) to its particular approach to application development and use this as the basis for decomposing a repository. Zachman decomposes application development along two dimensions: one corresponding to the lifecycle (planning, analysis, conceptual design, detailed design, and implementation), and the other corresponding to the w-questions (*w*hat - data, ho*w* - function, *w*here - network, *w*ho - people, *w*hen - time, and *w*hy - motivation). His decomposition produces 30 cells each containing their own form of information such as Narrative Text, Entity Relationship Diagrams, Process Flow Diagrams, Data Flow Diagrams, Structure Charts, Network Logistics Diagrams, Organizational Charts, etc. See (Zachman, 1987; Sowa and Zachman, 1992) for details.

B. Integrated Data Dictionaries: Handling Diversity

An integrated data dictionary must resolve differences between dictionaries supplied by different vendors. It must also be very careful about data that is stored by multiple DBMS's (e.g., personnel data). If the DBMS's all roughly adhere to the SQL standards, then resolution of differences will not be unduly difficult. Often it may be necessary to integrate non-SQL or even non-relational (e.g., Hierarchical, Network, or Object-Oriented) DBMS's. The research literature on federated databases is helpful in dealing with these complexities (Sheth, 1990). One way of organizing such a data dictionary is to figure out what portions of the vendor supplied data dictionaries are of interest to users, while ignoring portions required just by the DBMS itself. The relevant data could be extracted from a vendor dictionary, mapped to the global model, and stored in the integrated dictionary. (The more active the integrated data dictionary or repository is to be, the greater the need to extract more of the information from the DBMS.)

One way to provide a common view of databases built using different technology (Hierarchical, Network, Relational, or Object-Oriented) is to reverse engineer their schemata to an extended ER Model. The information can be stored at this abstract level in the repository. (Of course the schema information appropriate for particular technologies would need to be stored also.) A rough equivalence between the various database technologies is shown in the following table.

ER Model	Relational	Network	Hierarchical	Object-Oriented
Entity Type	Table	Record Type	Record Type	Class
Ent. Supertype				Superclass
Entity	Row	Record	Record	Object
m-1 Rel. Type	Foreign Key	Set Type	PCR-Type	Set-Valued Attr.
m-m Rel. Type	Table	Record Type		Set-Valued Attr.
Attribute	Column	Data Item	Field	Attribute
	Constraint			Constraint
				Method

Note, Rel stands for Relationship, m-1 stands for many-to-one, Attr stands for Attribute, and PCR stands for Parent-Child Relationship.

In Federated Databases (Sheth, 1990), local schemata are mapped to a Common Data Model (CDM). This data model must be semantically rich enough to provide equivalent representations for the constructs provided by the DBMS's used by an enterprise. Typically, the CDM is based on an extended ER or OO approach. At this point, the individual CDM schemata need to be unified using schema integration techniques. As a final step the unified CDM schema may (or may not) be mapped into a form the users' prefer, e.g., a relational schema. In a federated database, users may then submit queries based on this federated schema. In a repository, it would simply provide an integrated presentation of corporate databases; queries would still need to be decomposed on a DBMS by DBMS basis.

C. Users and their Roles

C.1. End-Users.

An End-User can be defined as the person or group of people that are the primary beneficiaries of the data maintained within an information system, and that have the responsibility for insuring the accuracy of the data. The best way to see the decomposition of the End-User Supertype is to show an example of the types of roles that end users may play during their business activity. In an inventory control scenario, there would be shipping/receiving clerks, sales personnel, buyers, strategic managers, and possibly an inventory business analyst.

The shipping/receiving clerks would be considered the end-users that are on the front line in the inventory process. They have direct responsibility for manipulating the data maintained regarding the stock levels of parts in inventory. They notate quantities removed from inventory, quantities added to inventory, and discrepancies. They are also responsible for ensuring correct alignment of the information system with the actual parts warehouse contents.

Sales personnel are dependent on accurate stocking information as well, in order to perform their day to day function. Essentially, their position in sales requires them to access inventory information on a routine basis and rely on the accuracy of the information.

Buyers or purchasing agents have a close relationship with the inventory data on a day to day basis just as sales personnel do. Their function, among others, is to replenish items in inventory, analyze and adjust reorder points, and plan for heavier or lighter demands on inventory items. In addition, they are responsible for preparing usage summaries for upper level management that contain trend information, pricing information, and opportunity cost information.

The upper level management personnel, sometimes called strategic managers, use the trend analysis information provided by the users with more direct contact with the data and the items they represent. These managers represent the major decision makers that influence the direction the inventory process takes with respect to the operations of the rest of the business (such as manufacturing). Their actions are based on abstracted information derived from the actual data values.

The inventory business analyst acts as the liaison between the materials management division and the management information systems division. Typically, the analyst knows the functions of all inventory personnel and their relationship with the inventory data (manipulators, accessors, analyzers, summarizers, decision makers). During information system development, the analyst provides input to the development team in order to insure accurate representation of the user environment. Goals for any new information system are determined and validation methods are established by the inventory analyst. In addition, the analyst provides support to ensure proper functioning of existing (legacy) systems, and authorizes the modification and acceptance of system revisions.

C.2. Application Developers

Typically, Application Developers are people that deal with the design, implementation, and maintenance of information systems. These systems support the end-users in their application. Application Developers do not depend on the actual data within the information system for their daily activity. However, they must have some awareness of the importance and use of the data so that their efforts as developers actually benefit the end-users. They are responsible for the programs and databases that manipulate and store the end-users' data.

On the application development side, several roles exist. These include the MIS Director (i.e., the executive that manages the enterprise application development effort), the systems analyst, the programmer, the resource coordinator, and the database administrator. Note that this list may not be complete for every development shop, rather it is intended as an example of the types of roles played in the enterprise. A case in point is the inclusion of the database administrator within this group. Some scenarios may call for the database personnel to be in a separate Users subtype and be decomposed into various roles such as database designer, administrator, query analyst, and user trainer, etc.

D. Current Examples of Repository Products

Several companies are currently marketing general-purpose repositories. Their products are briefly described below (companies with the more desirable products are starred).

1. * BrownStone Solutions
   New York, NY
   Data Dictionary Solution (DDS), CA/S, Cross Case Enabler

   Characteristics: Works for mainframe and client-server environments

2. * Digital Equipment Corporation
   Maynard, MA
   CDD/Repository (previous versions were know as CDD/Plus)

   Characteristics: Currently runs under OpenVMS/Rdb.

3. DSIMS Corp.
   Waxahachie, TX
   DSIMS Data Dictionary

   Characteristics: More like an integrated data dictionary; works with IMS and DB2 under MVS.

4. * InfoSpan
   Minneapolis, MN
   IRMS*/RM, IRMS*/VIM, IRMS*/BNM, IRMS*/RAM, CaseSpan, Information Asset

Manager (IAM)

Characteristics: Industry's first implementation of the ANSI/FIPS IRDS standard; runs on PC's and workstations.

5. * Manager Software Products, Inc.
   Lexington, MA
   Method Manager

   Characteristics: Runs under a variety of platforms and works with a variety of databases.

6. Mark V Systems
   Encino, CA
   East

   Characteristics: Not comprehensive.

7. * Reltech Products, Inc.
   Arlington, VA
   DB Excel

   Characteristics: DB2/MVS Based Server, IBM oriented (follows AD/Cycle (Platform) specs). Also run under OS/2 and can interface with a variety of DBMS products.

8. * R&O, Inc.
   Lexington, MA
   Rochade

   Characteristics: Client-Server, LAN-based server, CASE, Multi-Platform

9. Software Solutions, Inc.
   Alexandria, VA
   PC Dictionary

   Characteristics: Not comprehensive.

10. * IBM
    Armonk, NY
    Repository Manager/MVS (RM/MVS)

    Characteristics: The repository provides the foundation for IBM's AD/Cycle (more recently referred to as AD/Platform). There is speculation that new products that run under OS/2 and AIX 6000 may be announced in 1994.

In addition, there are several repositories that form the core of a vendor supplied development environment. Examples include:

1. KnowledgeWare, Inc.
   Atlanta, GA
   Information Engineering Workbench (IEW), Analysis/Design Workbench (ADW)
2. Texas Instruments
   Dallas, TX
   Information Engineering Facility (IEF)
3. Bachman Information Systems, Inc.
   Bachman Re-Engineered Product Set
4. Manager Software Products, Inc.
   Data Manager
5. Intersolv, Inc.
   Excelerator
6. Software AG
   PREDICT, PREDICT CASE
7. Oracle Corporation
   Redwood Shores, CA
   Oracle CASE, Cooperative Development Environment (CDE)
8. Gupta Corporation
   Menlo Park, CA
   TeamWindows

This last entry is noteworthy because of its very low price.

E. Definitions

E.1. Models

To build complex pieces of software, it is advantageous to build models of them first. A model is a higher-level simplified representation of the software to be produced. It is often graphically based.

1. Data Model -- focuses on data, primarily files and databases. Typically extended Entity Relationship Diagrams (ERD's) are used.
2. Process Model -- focuses on processes and their constituent modules and functions. Typically Data Flow Diagrams (DFD's) are used.
3. Object Model -- encapsulates data and process components into an object which is a representation of something in the real-world that has a state (information content) and exhibits well-defined behavior. There are no definitive standards here.

E.2. Software

The descriptions given below represent the central intent of the category. Of course grey areas exist, so that for a particular piece of software it may be hard to categorize it.

1. Application -- a program which performs some business function (payroll, order entry, inventory).
2. Tool -- (a) Development Tools -- a program that assists in producing other programs (e.g., a compiler, a design tool, a testing tool, a debugger); (b) Productivity Tools -- a program that helps users perform general tasks such as word processing or spreadsheet analysis.
3. System -- a set of programs that provide enhanced services to users (developers as well as end-users), e.g., OS's, DBMS's.

F. PowerBuilder's Meta-Information

Meta-information is contained in files called pibble files (".pbl" files). Each pibble file contains a group of object instances: applications, windows, data-windows, menus, functions, user objects, structures, etc., and each kind of object can have more than one instance of the same kind. For example, you can have two or more applications that are in one pibble file, which can also contain more than one menu or data-window, etc.

Each object instance is called an entry in a pibble file. In fact, these instances are OO object instances and can be operated upon through their corresponding painters. So the meta-information in a pibble file can be said to be the sum of all meta-information concerned with each entry. Meta-information for each entry can be obtained through the library painter. When you are in the library painter, you may choose one of the entries in a pibble file and export this entry to a text file. This text file can also be up-loaded into other pibble files.

The meta-information is similar to meta-information that you might expect to collect from a well-documented C++ program. The contents of entry exports are such things as global variable declarations, local variable declaration, shared variables, predefined objects (window, menu, etc.) structures and their associated operations (create, destroy, and events which are characteristic of window-based application, as well as event handling routines). The key information that can be extracted from a data-window is the select clause. A data-window is a buffer between your application and a database, and the two operations are update (write the data in the data window back to a database) and retrieve (read data from a database into the data-window).

G. WSRC Repository Prototype Design Reports