Data modeling for software development

by R. W. Matthews
W. C. McGee

One of the motivations for the use of a facility such as the Repository Manager™ in an information processing system is to centralize the information needed for the development of software. What this information is and how it is interrelated is defined in the underlying data model. This paper discusses the kinds of information required for software development and offers some suggestions on how the data model should be organized and implemented.

A facility such as the Repository Manager™[1] allows an installation to create an organized, shared collection of information about an enterprise's information systems. One of the motivations for such a facility is to centralize the information needed for the development of the enterprise's software. This information is generated by a variety of tools. At the same time, it is accessed by a variety of tools, not only while the software is being developed but also throughout its life as it undergoes correction, change, and maintenance.

The information generated in the development of software can be thought of as a software development database, the clients of which are the various software development tools. The problem of designing such a collection of information is not unlike that of designing any database. In particular, provision will be made for sharing information among tools, so that data stored by one tool are accessible on an equal basis to any other tool. Equally important, when information is elicited from the user, it must be done only once.

In order for tools to share data, the data must have an agreed-to format or model. We refer to this model as a *data model*. This model allows the tool set to evolve over time without the constant renegotiation of intertool interfaces. The model should also be reasonably stable, so that existing tools are not adversely affected by model extensions. Additions to the model will be frequent but generally not disruptive. Changes should be infrequent, and deletions should hardly ever occur.

In this paper we present a technique for the design of a data model suitable for software development. We begin by briefly reviewing the types of information generated by software development activity, and then consider the manner in which the data can be organized. We conclude by suggesting some criteria by which the success of a data model can be measured.

Data requirements for software development

Before a data model suitable for software development can be designed, it is necessary to understand the information that is generated by software development activity. This can be approached by first

identifying the major stages in the software development cycle and then considering the data produced at each stage.

According to Reference 2, the software development cycle has the following major stages:

- Requirements
- Analysis and design
- Produce
- Build and test
- Production and maintenance

Table 1 indicates the major kinds of data produced in the various stages of the software development cycle. For example, in the analysis and design stage, software specifications are written to state the purpose of the software, the functions that it is to perform, the external interfaces to the software, and the environment(s) in which it is to operate. These specifications are usually written in natural language and are then used to design the various components of the software: databases, panels, reports, programs, and so forth. Designs may be expressed again in natural language, or they may be given in a formal language that can be processed by a design tool to yield data required at the next stage of the development cycle.

As can be seen from the table, the data generated in software development have a range of formats and sizes. Specifications tend to be unstructured collections of text, whereas source programs have more restricted formats. Data definitions tend to be compact, whereas programs and listings tend to be bulky.

An important characteristic of these data is that they tend to be highly interrelated. That is, data produced in one stage are (or may be expected to be) related to other data in that stage or different stages. For example, source modules are related to the program specification that they implement; object modules are related to the source modules from which they are produced by a compiler, and so forth. When these relationships are properly recorded, the impact of changes in one aspect of the software on other aspects (e.g., the effect of a change to a record format on programs that use the record) can be readily determined.

In addition to data on the software itself, it is useful to maintain data about the hardware/software environment in which the software was developed and in which it will execute. When this is done, the impact of changes in the environment on the software (e.g., the substitution of one terminal type for another) can be readily determined.

Table 1 Data generated in the software development cycle

Requirements	business models business objectives/problems
Analysis and design	specifications database design panel/report design program design
Produce	source code object code database definitions panel definitions development libraries listings
Build and test	test plans integration plans test cases test results test libraries
Production and maintenance	operating procedure production libraries problem reports problem fixes change management records

Representation of software development data

The first step in the development of a data model for software development is to identify and formally define the types of data involved. For this purpose, it is useful to use the concepts of *entity* and *relationship*. An entity is any identifiable thing or event that can be characterized in terms of a set of attributes and their associated values. For example, a source module is an entity with such attributes as programming language, statement count, or reentrant. A relationship is an association of two or more entities which may have attributes of its own. For example, the association of an object module and the source module from which it was compiled is a relationship whose attributes include date of compilation.

Entities fall into *types*, which are based primarily on the particular attributes that they possess. For example, source modules tend to have the same or similar attributes, and therefore they can be considered entities of a single type. Relationships can be similarly typed, based on their attributes and on the types of the entities that they associate.

A collection of entity types and relationship types constitutes an *entity-relationship data model.* The model is used by various software development tools in creating and maintaining entity and relationship instances, which represent software development information. The model can be extended over time by adding types to support new kinds of software information.

An approach must be found that divides the entity types into understandable and manageable units.

The second step in developing a data model for software development consists of determining how entities and relationships of each type will be implemented. It is in this step that the spectrum of data forms characteristic of software development can be best accommodated. In particular, entities that are compact and must be accessed from many other entities can be implemented in an entity-relationship (ER) facility, such as that provided by the IBM Repository Manager/MVS™ (RM).

Entities that are bulky and can be organized into records that are accessed sequentially can be represented as flat files, and surrogates for such entities can be stored in an ER facility to document their existence and their relationships with other entities.

Model organization

When facing the task of identifying the various types of entities to be controlled and maintained, it quickly becomes apparent that an approach must be found that segments or divides the set of entity types into understandable and manageable units. Each of these units can then be analyzed for completeness and its role within the overall data model. Each unit can also be evaluated for any dependencies or associations with entity types in other units.

Entity-type categories. One such approach involves segmenting the set of entity types into distinct categories, according to their relationships with other entity types and to the overall model. By this approach, we group those entity types that deal with software development from a conceptual standpoint. These might include the following entity types: business process, business goal, business entity, and business attribute. Together they serve to define the requirements for business applications.

Another group might be composed of entity types to be used to describe a software program or system from a logic or design point of view. Such entity types as data views and data elements describe the data design for applications and databases that meet the requirements identified in the conceptual model.

A third grouping in this example represents the physical implementation of the software program. Here, the entity types deal with the descriptions of various implementations of the logical application description. The data structures described in the logical definition are, in this case, represented as table definitions or segment definitions. The data view may be represented by a panel definition or a report format definition entity.

There are other entity types, however, that apply to more than one of these categories. Organization unit and location are examples of this type of entity, which we call *universal entity types.* Table 2 lists the three entity type categories just discussed and includes some examples of entity types and entity instances that fit within each category.

Example of relationships. The glue that joins these categories of entity types into a single view of data is supplied by *relationship types* (e.g., use, produce), which identify interdependencies and associations between individual entity types regardless of their category. Using the levels and entity types depicted in Table 2, relationship types can be identified between entity types within a single category as well as between entity types in different categories. For example, a PAYROLL process uses information about an EMPLOYEE (e.g., rate, hours worked, etc.) as input to EXECUTE_PAYROLL to produce (MONTH_END) a paycheck. This is an instance of a relationship that associates a process with the data it either produces or uses.

Figure 1 illustrates some sample relationship types that associate entity types within a single category

Table 2 Entity categories with sample object types and instances

Category	Entity Types	Sample Entity Instances
Conceptual	business entity	EMPLOYEE
	business relationship	EXECUTE_PAYROLL
	business process	PAYROLL
	business event	MONTH_END, PROJECT_COMPLETION
	business goal	
	organization unit	
	location	
Logical	user view	CHECK
	data view	EMPLOYEE_PAY_VOUCHER
	data element	ACCOUNT_NUMBER
Physical	source code	PRTCHK00
	segment definition	EMPSEG01
	table definition	EMP_TABLE
	panel definition	EMPPANIA

(business entity to business process) and across category boundaries (data view to table definition).

Conceptually, there is no implication that the payroll process is a computer-based application. The payroll process could be performed by monks laboring in silence and seclusion, using quills and ink under candlelight. At this conceptual level, the entities often refer to tangible things, events, and ideas.

At the logical level, those conceptual definitions are dealt with in data processing terms. Again, however, the definitions have no specific implementation characteristics. Here we see that a user view (e.g., CHECK) produces information described in a data view (e.g., EMPLOYEE_PAY_VOUCHER); the data view comprises many data elements (e.g., ACCOUNT_NUMBER).

The physical implementation of the payroll program unit is documented at the physical level. Here is recorded the fact that the payroll program reads records from the employee table and prints an employee pay voucher.

Other relationships that can be defined here show that the payroll program is a single and unique implementation of the payroll function unit. The payroll function unit is in turn a logic design for all or part of the payroll process, as defined at the conceptual level. Similar relationships tie the employee information to a data design and an implementation of that design.

Different types of descriptions

A second method of sectioning the entity types uses the type of descriptive information they contain relative to the software product being developed. Reference 3 suggests that the minimal set of description types address the what, how, where, when, why, and who aspects of the product.

The application of the entity-type categories and description-type approaches simultaneously allow the partitioning of the entity types into very concise units. The grid or template into which these units fit is then very much in line with the framework described by Zachman in Reference 3. Figure 2 shows this correlation. The scope and detailed description perspectives that appear in the Zachman framework are shown here, but they are not applicable to the previous section on model organization. By building on the example shown in Table 2 and placing the entity types in appropriate cells in the grid, the resulting model (without relationships) is shown in Figure 3.

Use of modeled data

We have discussed various aspects of modeling data for software development. In this section, we consider briefly the uses which can be made of the data. Some of the functions we describe, such as program analysis, are now experimentally state-of-the-art, but their development should be stimulated by the use of a common data model.

Figure 1 Sample entity types and interrelationships

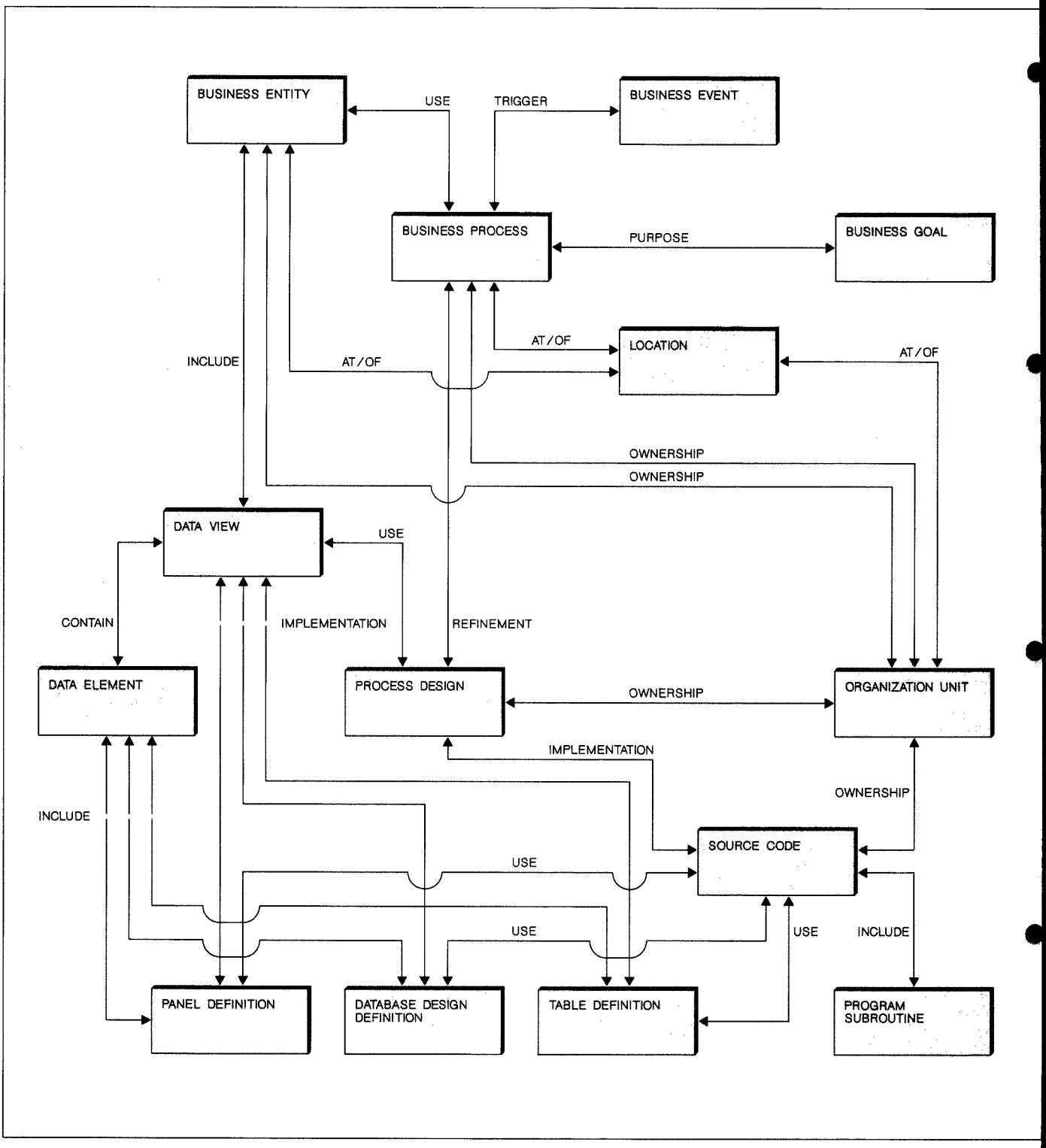


Figure 2 Comparison with the Zachman framework

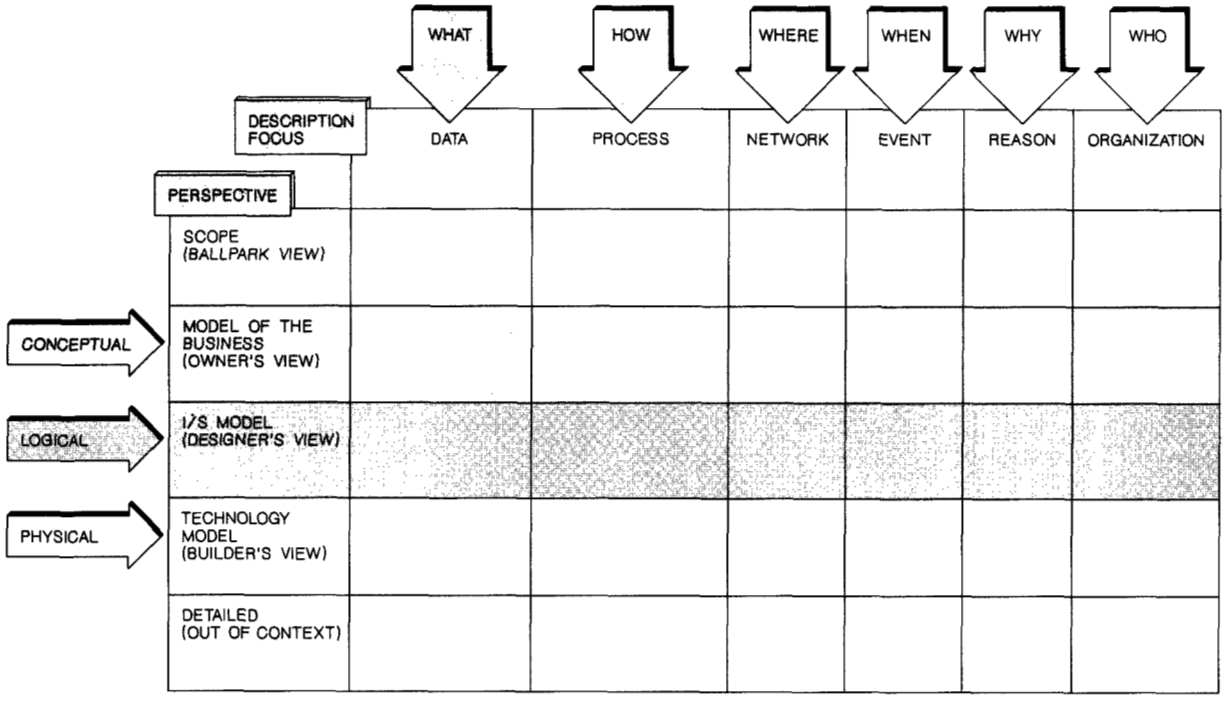
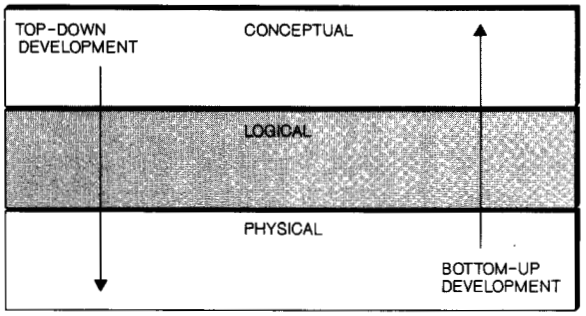


Figure 3 Entering object types in the framework

PERSPECTIVE / DESCRIPTION FOCUS	DATA	PROCESS	NETWORK	EVENT	REASON	ORGANIZATION
SCOPE (BALLPARK VIEW)						
CONCEPTUAL — MODEL OF THE BUSINESS (OWNER'S VIEW)	-BUSINESS ENTITY -BUSINESS RELATIONSHIP	-BUSINESS PROCESS -BUSINESS RULE	-LOCATION	-BUSINESS EVENT	-BUSINESS GOAL	-ORGANIZATION UNIT
LOGICAL — I/S MODEL (DESIGNER'S VIEW)	-DATA VIEW -DATA ELEMENT	-PROCESS DESIGN -LOGIC RULE	-LOCATION	-EVENT		-ORGANIZATION UNIT
PHYSICAL — TECHNOLOGY MODEL (BUILDER'S VIEW)	-DBD DEFINITION -TABLE DEFINITION -PANEL DEFINITION	-SOURCE CODE -LINK PLAN	-LOCATION			-ORGANIZATION UNIT
DETAILED (OUT OF CONTEXT)	-TABLE DDL -DBDGEN JCL	-PROGRAM -SUBROUTINE	-NETWORK MAP			

Figure 4 Traditional top-down flow and bottom-up flow



Traditionally, the process of developing software begins with requirements, continues with analysis and design, produce, build and test, and ends with production and on-going maintenance. Following that scheme and applying it to the model organization grid previously illustrated, it is apparent that there is a kind of flow through the model. During the requirements phase, those entity types identified in the conceptual level of abstraction are populated and manipulated. In the early stages of the design phase, those same entity types supply the basic information to shape the design. The logical entity types are populated with information from the conceptual entities. Depending on the robustness of the logical entity types and the information they contain, some or all of the information needed for coding can be found in those entities. A tool that can obtain the design information from the logical entities can in turn populate the physical entities and perhaps generate the bulk of the code needed for the software program. Under these conditions, the user is required to enter information just once. Translation and transformation of information from one level to the next is the job of software tools.

Also note that there is no requirement that the user begin with entering requirements information at the conceptual level. The user may start at the logical or physical levels. The level of detail is, of course, different at each level, but there is no requirement that the definitions be completed at one level before moving on to the next. The model structure fully supports an iterative style of modeling. Therefore, facts or concepts identified at one level can be iteratively applied to other levels.

We have been discussing here what is often referred to as a top-down approach, that is, populating the repository model from the conceptual level downward. Another method for populating the repository model is the bottom-up method, which uses, for example, the analysis of existing programs to discover their structure, data usage, logic design, and operational characteristics. This method normally begins with populating the physical entities, and from them (with additional input) populates the logical and then the conceptual objects. Figure 4 shows how these two methods vary.

Concluding remarks

The success of a data model for software development can be measured in terms of the following affirmative criteria:

- The model is easy to understand and use.
- A prospective software development tool developer can easily find the entity types and relationship types required.
- The entity and relationship types have been implemented appropriately.
- The model meets the performance objectives of the development tool.
- The model is robust.
- The model can be extended by the tool developer in a nondisruptive way.

Acknowledgment

We wish to acknowledge the benefits we have received from the pioneering work of John Zachman in information systems architecture.

Cited references

1. J. M. Sagawa, "Repository Manager Technology," *IBM Systems Journal* **29,** No. 2, 209–227 (1990, this issue).
2. V. J. Mercurio, B. F. Meyers, A. M. Nisbet, and G. Radin, "AD/Cycle Strategy and Architecture," *IBM Systems Journal* **29,** No. 2, 170–188 (1990, this issue).
3. J. A. Zachman, "A Framework for Information Systems Architecture," *IBM Systems Journal* **26,** No. 3, 276–292 (1987).

Robert W. Matthews *IBM Programming Systems, Santa Teresa Laboratory, P.O. Box 49023, San Jose, California 95161-9023.* Mr. Matthews is currently an advisory programmer in repository product development at the IBM Santa Teresa Laboratory. He joined IBM in 1974 as a junior programmer in the IBM development laboratory, Kingston, New York, where he specialized in supporting the software development and library processes for the Kingston Programming Center. In 1976, he assumed similar re-

sponsibilities at the Santa Teresa Laboratory. Since 1986, he has worked on different aspects of the use of a repository in software development. Mr. Matthews holds a degree in mathematics from New Mexico Institute of Mining and Technology.

William C. McGee *IBM Programming Systems, Santa Teresa Laboratory, P.O. Box 49023, San Jose, California 95161-9023.* Mr. McGee is currently a senior programmer in repository product development at the IBM Santa Teresa Laboratory. He joined IBM in 1964 as a staff member of the Palo Alto Scientific Center, where he specialized in physics applications, computer graphics, and database systems. In 1969, he received an Outstanding Contribution Award for work on Data Base/Data Communication (DB/DC) requirements and strategy. In 1970 he joined the DB/DC development group in Palo Alto, where he was manager of the DB/DC architecture department. Other assignments in DB/DC development have included performance evaluation, distributed data requirements and planning, and data dictionary planning and development. From 1951 to 1959, Mr. McGee was with the General Electric Hanford Atomic Products Operation in Richland, Washington, as manager of the numerical analysis unit and as a reactor data specialist. From 1959 to 1964, he was head of systems programming and research at Ramo Wooldridge Corporation in Canoga Park, California. Mr. McGee received the A.B. degree in physics from the University of California at Berkeley in 1949, and the M.A. degree in physics from Columbia University, in 1951. He is a member of the Association for Computing Machinery.

Reprint Order No. G321-5395.