Egenhofer, M., and A. U. Frank. "Panda: An Extensible DBMS Supporting Object-Oriented Software Techniques." Paper presented at the GI/SI-Fachtagung, Zürich, March 1989 1989.

Panda: An Extensible DMBS Supporting Object-Oriented Software Techniques*

Max J. Egenhofer
Andrew U. Frank
National Center for Geographic Information and Analysis
and
Department of Surveying Engineering
University of Maine
Orono, ME 04469, USA
MAX@MECAN1.bitnet
FRANK@MECAN1.bitnet

ABSTRACT. The PANDA database management system was designed for non-standard applications which deal with spatial data. It supports an object-oriented program design with modularization, encapsulation, and reusability, and can be easily embedded into complex applications, such as spatial information systems or cartographic expert systems. It is presented how complex objects and their operations are defined. A layered structure on top of the programmer's interface provides object operations which include potentially complex consistency constraints.

1. Introduction

The interface of a non-standard DBMS must fit well into the methods used for the implementation of the applications; therefore, it is important that database methods and techniques fit well within a software engineering environment. Software engineering techniques, such as the concept of abstract data types [Guttag 1977] [Parnas 1978] [Zilles 1984] and abstract object types [Sernades 1987], in which each module encapsulates an object type with all its pertinent operations, are commonly used for applications; however, database management systems do not sufficiently support them. Their interfaces to programming languages, like Embedded SQL, have demonstrated to be incompatible with sophisticated software engineering concepts and cumbersome [Christensen 1987].

This short version of a longer report [Egenhofer 1988b] describes the software engineering techniques used in PANDA [Frank 1982], an object-oriented database tailored for applications with spatial data. PANDA is an acronym for Pascal Network Database Management System. The system was originally designed as a network for DBMS with object-oriented concepts and enhancements for storage and access of spatial data and evolved over the years to an object-oriented design. PANDA consists of about 40,000 lines of program code and has been running several years, primarily

* This work was partially funded by a grant from NSF under No. IST-8609123 and equipment grants from Digital Equipment Corporation.

used for research and teaching in undergraduate and graduate courses. Its development started at the Swiss Federal Institute for Technology, Zurich, and has been continued at the University of Maine. Code was transferred between such vastly different hardware and operating systems as DEC-10 (under TOPS-10), IBM 370 (under VM/CMS), and VAX/MicroVAX (under VMS).

PANDA's data model is based upon the basic concepts of abstraction: classification, generalization, and aggregation [Brodie 1984]. The term *object* is used for a single occurrence (instantiation) of data; *type, class,* or *abstract data type,* refer to sorts of objects; *superclass* describes the grouping of several classes in an is_a relation (generalization) [Dahl 1966]; and *subclass* is the specialization of a superclass. *Inheritance* defines a superclass in terms of one or more other classes, propagating the properties of the superclass to all subclasses transitively.

Panda's DB kernel manages storage and retrieval of data from persistent storage devices, buffers pages and records to improve performance, provides facilities for transaction management, incorporates structures for logical access (hashing, B*-trees, Field Tree for spatial access [Frank 1983]), and provides generic structures for aggregation and generalization. The kernel is built in a layered and modular structure so that it can be easily extended by additional parts, e.g., a multi-user facility. The extensible part of PANDA is the definition of the application-specific classes based on the definition of value types which can be arbitrarily complex acyclic data types.

The paper starts with a brief discussion of PANDA's software environment. Section 3 focuses on PANDA's object model. The four phases during the definition of complex classes are presented. In section 4 the implementation of object operations in layered structure on top of the generic programmer's interface is presented. The paper concludes that better object-oriented programming languages can help to implement object-oriented databases closer to the designed models.

2. Software Engineering Aspects

An abstract data type (ADT) is a mathematical structure which fully defines the behavior (semantics and operations) of objects. Abstract data types are specified as an algebra describing what sorts of objects (types) are dealt with, and what kinds of operations they are subject to. A set of axioms determines the effects of the operations. Abstract data types can be combined in layers, where higher –level abstract data types are first described independently (specifications), and it is then shown how this behavior can be achieved using other, hierarchically lower abstract data types (abstract implementation [Olthoff 1985] [Frank 1986]).

The software engineering environment of PANDA supports *modularization, encapsulation, reusability,* and *transportability*. Programmers are motivated to write object-oriented code as the implementation of ADTs in a standardized way such that other programmers can easily read and correct it. Modularization is achieved by using a Pascal precompiler [Egenhofer 1988a], which provides types and routines from one module to another, with type checking across modules. The implementation of operations is encapsulated into the module. A highly modular programming style requires that the modules are managed in a controlled library system providing the user information about existing ADTs, their operations, and their specifications. The compiler is embedded into a library management system with hierarchical directory structure where several versions of modules can be kept in parallel. Transportability of

code among different Pascal compilers and various hardware is guaranteed by the precompiler, because only the precompiler itself, not the particular code, must be adapted to fit specific compiler features.

3. Object Definition

An object-oriented data model gives rise to the formation of arbitrarily complex classes, the majority of which is composed of a (limited) number of lower structured parts, here called *components*. The same components occur repeatedly as part of various classes, and it is more economical to define and reuse components, instead of defining redundantly a multitude of very similar classes. Though this may resemble a traditional entity-attribute type of concept, it is only the object-oriented implementation of several classes with the same components which are defined only once and reused in every class they are part of. The decomposition of objects into components releases the application designer from redundant definitions of specific object operations for the structures. Considerable overhead is removed by defining the operations for the value types and applying them for classes. Users are not aware of this decomposition because they see only objects and operations upon them.

PANDA's object definition consisting of *value, classes,* and a generalized *objectType* adopts the modular concept and fits well into the layered architecture. An application model is defined in standardized operations that are easily integrated into the kernel. The operations are implemented as short Pascal routines which are executed during PANDA's startup. This approach releases the database administrator from recompiling the entire DBMS code; instead, only the initialization modules for each class and an interface module must be compiled, and (shareable) images linked. A systematic implementation with strict naming conventions allows the generation of code for these modules. Currently, code is generated partially from definitions in a text file; a more user friendly generation immediately from a graphical schema design is being investigated. The four phases of the object definition are distinguished.

3.1. Phase 1: Declaration of User Terms

The kernel was originally initiated with templates for the names of the classes, components, links, and paths. These templates must be replaced with the particular names used in the schema design of the application.

3.2. Phase 2: Definition of Value Types

A value type is the Pascal implementation of an abstract data type in a single programming unit, consisting of a type definition and a set of pertinent operations, and serves as class component. The type definition can be any simple data type, such as integer, string, real, etc., or any structured type, such as array or record, except types for dynamically allocated variables, such as pointers[1]. Value types can be built upon other value types by using their definitions and methods. Due to restrictions of current compilers, only acyclic combinations of value types are permitted.

[1] This limitation is with respect to the intended persistency of objects.

Each value type must provide a set of fundamental operations necessary for indexing and hashing structures of objects. For example, hashing requires a function that calculates a hash value, and an operation that compares two values for equality. The implementation of these operations depends upon the structure and the semantics of the class. Unlike traditional databases that support only a limited, hard coded set of types, PANDA is extensible. This implies that the operations for the supporting structures are not predefined and must be provided by the application designer.

3.3. Phase 3: Composition of Classes

The properties of a class are determined by its own components and the components transitively inherited from its superclasses. Details about the implementation of a class are hidden, such that modules outside of the class definition are not aware of the decomposition into object components. Class components are implemented as Pascal records. Inheritance is simulated with the help of variant records, which is necessary due to the lacking support of inheritance in Pascal.

These type definitions are not yet sufficient for the object definition. Programming languages of the FORTRAN/Algol type separate compilation and execution of code into two phases which provide different levels of information. During compilation, variables and types are named, and types may be composed from other types. During executing, the names and the relation between a variable and its type is not accessible by the user code. Likewise, it is not possible to find out whether a type is part of another structured type.

In order to provide the compile time knowledge also during execution, the relations between each class must be defined explicitly as programming code. PANDA has the database administrator write these standardized routine which will be executed by the kernel during startup, initializing the user's model. For each class, the composition of the type and the corresponding ini-routine are combined in a separate module.

3.4. Phase 4: Definition of Database Objects

All classes are specializations of the most general superclass *objectType* from which all pertinent DB-operations are inherited. The objectType is implemented as a Pascal record with varying parts for each specific class. It combines all specific classes into a single, compatible type to which common (system) components, such as tuple identifiers and aggregate pointers, are added.

4. A Layered Structure of Object Operations

The object-oriented approach requires the definition of complex objects *and* their pertinent operations. While the knowledge about the composition of complex classes is essential to the database kernel, the object operations are the application programmer's tools to manipulate objects. The programmer's interface of the DBMS kernel is a collection of object-oriented manipulation and retrieval operations which can be called from the application programs. These operations are defined for the generalized objectType, and are compatible with any class. Conceptually, all database operations, such as *store*, *delete*, and *update*, are inherited to each class of the schema. Their

implementation on top of the programmer's interface is a layered structure of model-specific object operations. This is the location to implement consistency checks. A layered structure of object operations has been developed which is generally applicable for any object-oriented application. Based upon the fundamental object operations, more complex operations can be defined. By restricting the operations to a single task, the code for the routines stays small and correctness can be verified more easily.

The structure of these operations is the same for every application: First, the operations are defined to make s specific object, and to assign values to and access them from an object. Then, unary object operations for storing, modifying, deleting, and accessing individual objects are defined based upon the generic DB-operations offered in the programmer's interface. Another layer treats all binary operations manipulating aggregates. These operations are exploited in the next layer to form complex object operations, including complex consistency constraints.

4.1. Level 1: Make, Get, and Put Operations

The first layer is a collection of modules with the basic operations to manipulate the individual description of a single object: creating an instance (*make*), assigning values to (*put*), and extracting values from an instance (*get*). These operations hide the implementation of the classes from the user, preventing uncontrolled access. Since the properties of a superclass are propagated to all subclasses, *get* and *put* operations are compatible with objects of the subclasses as well.

For each class a separate module contains these operations. Their implementation is trivial, and the code can be generated with the knowledge of the object description. Simple consistency constraints, such as checking whether a value lies within a range, can be added to the put operations.

4.2. Level 2: Unary Object Operation

The second layer covers all database operations to store, update, and access a single object which are inherited from the common superclass *objectType*. The implementation of these operations is straightforward because the generic object operations are part of the programmer's interface and can be immediately applied to each object class. Depending on the definition of the class, different access methods are supported, such as access with a key value and spatial access. For each class, a separate module with the specific object operations is implemented.

4.3. Level 3: Binary Object Operations

The third layer comprises aggregate operations that always involve two objects. Standard operations are the addition of a part to an aggregate, the removal of a part from an aggregate, cancellation of an entire aggregate, and the access of parts of an aggregate. The operations which establish links among objects require that the corresponding objects have been loaded into the database before. Reversely, remove and cancel dissolve the links without deleting the previously linked objects from the database. For each aggregation, a separate module with the aggregate operations is implemented.

Three types of access operations for aggregates are distinguished: (1) iterating over all aggregate components, similar to a FOR EACH loop in CLU [Liskov 1981], (2)

getting a specific aggregate component with a certain value, and (3) getting the composite object part of an aggregate. The second access method is only efficiently supported if a sorted access path was defined.

4.4. Level 4+: Complex Object Operations

The fourth and later layers combine operations of the lower levels to form more complex operations. The level structure is open and can be extended according to the complexity of the application. Entire applications have been written in this highly structured form. The advantage of the object layers is that very complex operations can be implemented by combining other object operations. Following the rule that no object operation may use other operations of higher level, a well-structured application package can be designed.

5. Conclusion

The close relation between the implementation of object-oriented databases and object-oriented software techniques has been explained. For an object-oriented database it is of vital interest to tie into the software environment of the application. PANDA's object-oriented programmer's interface facilitates applications on top which conform with an object-oriented design.

With the growing complexity of the application the layered structure of complex object operations can be extended beyond the four basic layers introduced. The embedding of consistency constraints into these object operations is a natural and object-oriented way, starting with very general operations, and constraining them more and more.

Conventional programming languages do not easily support the implementation of object-oriented databases and often, methods must be simulated to match the model. Using a language that supports multiple inheritance, clearer designs and more condensed implementations become possible.

References

[Brodie 1984] M.L. Brodie. On the Development of Data Models. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York (NY), 1984.

[Christensen 1987] A. Christensen and T.U. Zahle. A Comparison of Self-Contained and Embedded Database Lagnguages. In P. Stocker and W. Kent, editors, Proceedings 13th VLDB Conference, Brighton, England, September 1987.

[Dahl 1966] O.-J. Dahl and K. Nygaard. SIMULA—An Algol-based Simulation Language. Communications of the ACM, 9(9), September 1966.

[Egenhofer 1988a] M. Egenhofer and A. Frank. A Precompiler For Modular, Transportable Pascal, SIGPLAN Notices, 23(3), March 1988.

[Egenhofer 1988b] M. Egenhofer and A. Frank. Object-Oriented Software Techniques in PANDA. Technical Report 96, Surveying Engineering Program, University of Maine, Orono (ME), December 1988.

[Frank 1982] A. Frank. PANDA—A Pascal Network Database System. In: G.W. Gorsline, editor, Proceedings of the fifth Symposium on Small Systems, Colorado Springs (CO), 1982.

[Frank 1983] A. Frank. Problems of Realizing LIS: Storage Methods for Space Related Data: The Field Tree. Technical Report 71, Institute for Geodesy and Photogrammetry, Swiss Federal Institute of Technology (EHT), Zurich, Switzerland, 1983.

[Frank 1986] A. Frank and W. Kuhn. Cell Graph: A Provable Correct Method for the Storage of Geometry. In: D. Marble, editor, Second International Symposium on Spatial Data Handling. Seattle (WA), 1986.

[Guttag 1977] J. Guttag. Abstract Data Types and the Development of Data Structures. Communications of the ACM, June 1977.

[Liskov 1981] B. Liskov et al. CLU Reference. Lecture Notes in Computer Science, Springer Verlag, New York (NY), 1981.

[Olthoff 1985] W. Olthoff. An Overview on ModPascal. SIGPLAN Notices, 20(10), October 1985.

[Parnas 1978] D.L. Parnas and J.E. Share. Language Facilities for Supporting the Use of Data Abstraction in the Development of Software Systems. Technical Report, Naval Research Laboratory, Washington (DC), 1978.

[Sernades 1987] A. Sernades et al. Object-Oriented Specification on Databases: An Algebraic Approach. In: P. Stocker and W. Kent, editors, Proceedings 13th VLDB Conference, Brighton England, September 1987.

[Zilles 1984] S.N. Zilles. Types, Algebras, and Modelling. In: M.L. Brodie et al., editors, On Conceptual Modelling, Springer Verlag, New York (NY), 1984.