Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

334

K.R. Dittrich (Ed.)

Advances in Object-Oriented Database Systems

2nd International Workshop on Object-Oriented Database Systems
Bad Münster am Stein-Ebernburg, FRG
September 1988
Proceedings

Multiple Inheritance and Genericity for the Integration of a Database Management System in an Object-Oriented Approach

Andrew U. Frank
Computer Science & Surveying Engineering
University of Maine
Orono, ME 04469
FRANK@MECAN1.bitnet

Abstract
Experience in designing and writing application programs using an object-oriented method reveals problems in connecting application programs to Database Management System (DBMS) services. This is due to the difference between the declarative description of data assumed in a DBMS and the behavioral, encapsulated format in an object-oriented language. To overcome this problem, the integration of DBMS services with an object-oriented language must be improved. A method is proposed to achieve seamless integration of DBMS functionality with application code using inheritance. The language must provide 1) **multiple inheritance**, allowing the objects to be stored in the database to inherit the necessary methods, and 2) **genericity**, in order to document what operations each object to be stored in the DBMS must export for use by the DBMS (eg. access operations). Most current object-oriented languages do not provide both features and thus make it difficult to add a DBMS package.
Using multiple inheritance and generic classes, a DBMS package can be written in the same language as the application program, and can be added to an object-oriented programming system. The method is also suitable for integrating other packages than DBMS (geometric data handling, graphics, user interface etc.) in an object-oriented environment.

1. Introduction

We are working on designing and programming Spatial Information Systems (also known as Geographic Information Systems) using an object-oriented software engineering method. The software engineering problems are very similar to the ones encountered in the design of CAD/CAM systems, VLSI, etc. In general, the object-oriented methods of software engineering are beneficial for this class of applications, but the connection of object-oriented application code with the database management system causes a break in the point of view. Since the Pingree Park Conference [SIGMOD 81], attention to the logical integration of DBMS functionality with programming languages and AI has been studied. On a broader base, constructing an object-oriented DBMS requires to bridge the two different concepts of 'global database schema declaration' and the object-oriented modularization.
Object-oriented application code encapsulates module internals (especially the data structure) and makes visible only a behavioral (procedural) interface. The DBMS on the other hand expects a declarative description of the data. The use of layers of modules transforming between the DBMS and object-oriented application modules has been proposed [Härder 1988]. It is in our experience a viable, but not straight forward solution to bridge the conceptual gap. The code in these layers is difficult to write, because it requires understanding of both the DBMS and the application world. At the same time, it is quite schematic. Using such application layers restricts changes in the remainder of the application code.
We propose to use inheritance for the integration of DBMS functionality with the application modules. Using this single language feature, typical for object-oriented programming, overcomes the mismatch between DBMS and object-oriented application programs: We will describe first, how inheritance is used to provide the DBMS operations to the object. Then we will discuss implementation concepts, showing the need for genericity. Objects, which are to be

stored in the database (subsequently called DB-objects), inherit the DBMS operations from a superclass 'generic DB-object'. Additionally, each DB-object class must export some operations for the use of the 'generic DB-object' (e.g. operations to access components of the objects). We will then point out, that a language which provides multiple inheritance and supports generic classes, allows writing a DBMS in the same language as the application and adding it as a set of modules to the application. We will conclude with stating that the same concepts and language features can be used to integrate other generic support packages (eg. graphics and user interface support).

The integration of a DBMS as application code is not a trivial problem. Solutions reported use object-based, but not object-oriented languages [Wegner 1987] and a relational data model with a global schema on the level of the application. This contradicts the object-oriented view of data encapsulated in modules. Our own efforts to integrate a DBMS in an object-oriented programming method (based on a Pascal extension) was stifled by problems of understanding how to combine separate object definitions with generic database functions. In this paper, we will argue that these problems can be overcome by using a language that provides multiple inheritance and genericity (eg. EIFEL [Meyer 1986]). The methods reported here are a refined view of the current implementation of the DBMS 'PANDA'.

In order to avoid confusion, the proposal in this paper is not based on a specific data model. The method can be applied with any data model that is compatible with object-oriented programming. If the DBMS code itself is written in an object-oriented language using the proposed mechanism, it will help us to understand the architecture, semantic, design and implementation issues of DBMS in general. This will lead to the DBMS kernel and toolbox [Härder 1985], where specific implementations for DBMS subsystems can be selected to suit the application demand best.

2. Providing DBMS Services to the Application Programmer

An object-oriented language demands the specification of each object type with all pertinent operations in a single module. The only way of accessing data is to use operations defined on the objects. Thus, every class of DB-objects must provide the necessary database operations. The following example out of a geometry application based on a node-edge-face-structure shows, how the class provides also database operations :

```
module node
create (name, x, y) return id
find (id) return node
findByname (name) return node
...
getName (node) return name
getX (node) return coordinate
...
distance (node, node) return real
...
```

Three differences appear in comparing the proposed with the classical approach:

- The data description is encapsulated in a module and not centralized in a DBMS schema.
- Access from the DBMS to the data is through defined interfaces and not directly from the DBMS code to the storage representation.
- Database operations are defined for each object separately and not generally for all DB-objects (and become therefore often simpler, because complex rules for parameter

types and value encoding are not necessary).

In object-oriented languages, the definition of object types can be arranged in a generalization hierarchy, such that properties (eg. operations) of the super classes apply also to all the sub classes[1]. It is thus sufficient to define a 'generic DB-object' with all necessary DBMS operations and have all the classes that should be included in the DBMS inherit its operations.

The module 'generic DB-object' must offer operations to create/store a new object, delete an object and depending on the data model used, find objects based on values using indices, aggregating objects to complex objects, etc. Other DBMS functionality must be provided, for example it must be possible to group changes in transactions, etc.

If 'generic DB-object' is declared with all the DBMS operation, a specification of the form "type node subtype of generic-DB-object" can provide the objects of type node with the required behavior. Application programmers may want to encapsulate this node object in an additional layer and not export the 'raw' database operations. This way even very complex consistency checks can be defined and enforced (without need for an additional formalism beyond the programming language).

To be generally useful, objects like 'node' must be able to inherit the behavior from other super classes than 'generic DB-object' and therefore, multiple inheritance is necessary (in our application, 'node' is also subtype of 'geometric objects', etc). In a language supporting only single inheritance (i.e. in most languages easily available, eg. C++, inheritance is probably used for other aspects of the design and not available to integrate DBMS functions (nor for the same matter, display managers or other packages). Thus multiple inheritance is crucial for this method of integration.

3. Implementation Strategy

Can database operations simply be inherited? In this section we lay out a strategy and list the necessary features of an object-oriented language necessary for its support. The following outline is based on the current structure of 'PANDA', an object-oriented DBMS that we use to build geographic information systems. It does not describe the current structure, which is obscured by, among other things, limitations in the programming language [Egenhofer 1988], but reflects the understanding we gained from building it.
In order to write the code for the 'generalized DB-object', from which DBMS behavior is inherited, we found a need for genericity combined with inheritance. In the next subsection we explain what we understand by genericity (for a more extensive discussion see [Meyer 1986] or [Olthoff 1986], [Cardelli 1985]). The second subsection shows its use for the definition of the 'generic DB-object'.

3.1 Genericity

Generic constructs are very similar to inheritance, but not exactly equivalent. They are, in our opinion, a slightly different formulation of the same concept in two different 'traditions' (specification languages and object-oriented languages). Inheritance in object-oriented languages should be extended to include genericity.

Genericity is theoretically based on the notion of algebraic systems or abstract algebras [Zilles 1984]. For example, a semi group is defined as a structure <S, *>, consisting of the elements in

1 It has been pointed out [Schaffert 1986], that a type hierarchy can be an implementation hierarchy (eg. Smalltalk-80) or a hierarchy of visible behavior (a specification hierarchy, eg. Trellis/Owl [Schaffert 1986]) - we assume here a specification hierarchy.

the set S with the associative operation '*'. From this definition, different semi groups can be constructed, for example the semi group of positive integers with addition. It is only necessary that the set of objects bound to S has an associative operation which can be bound to '*'. The following example out of [Sutor 1987] shows such a definition of a semi group:

```
semiGroup () : category == set with
    "*" : ($,$) -> $
    associative ("*")
```

Similarly, we can construct sorted lists as algebraic structures over elements which must include an order relation (with the properties reflexive, asymmetric and transitive). Specification languages have modelled this concept by including generic type constructors, allowing the description of abstract types, which are instantiated by bounding the formal type and operation variables to actual types and operations (CLU], ADA , ScratchPad [Sutor 1987], etc). This documents that a new type can only be constructed if the necessary operations are provided. It would be desirable that the language could also test, that the operations have the necessary properties. This is still a topic of ongoing research in programming and specification languages.

Inheritance is very similar to genericity: similarly to the subclasses inheriting all operations from their super class, the instantiations in genericity provide the same operations as their generic type. Inheritance in most object-oriented languages does not allow for type variables, nor formal operation variables required in the subtype, but the same effect is often achieved by using of method selection in the superclass based on 'self', and using the same operation names in all subclasses; however, this latter possibility is more difficult to understand and the class interface does not document the need for operations in the subclasses.

3.2 DBMS Functionality provided as a Generic Class

Providing DBMS functionality by defining the class 'generic DB-object' depends on a number of operations defined for each subclass because a number of DBMS operations need access to certain object components or depend otherwise on values of the objects.

As a simplified example, consider the following definition of a sorted list:

```
   "sortedList of element (with lessOrEqual (a,b: element):boolean)"
   operations "initialize", "insert", "find", "getNext" ...
```

where element is a type variable and the 'with' clause documents, that the implementation depends on an operation 'lessOrEqual' defined for elements. It is possible to write generic code that implements the operations without specific knowledge about the elements of the list, except that a 'lessOrEqual' operation is provided by the element. The following example shows a possible instantiation of the generic sorted list above:

```
   "sortedList of Name (with lessOrEqual mapped to name.before)"
```

where name.before is an operation that compares two names. Note that this respects encapsulation by using the module interface of 'names' instead of directly accessing the data in 'names'.

We have used the described method extensively for PANDA (without using a language that supports the above shown syntax) to write the functions of 'generic DB-object' for a data model similar to entity relationship [Chen1976] or the molecule concept [Batory 1984] [Härder 1987]. The following operations are typically necessary for each object class:

- comparison of two objects of the same type ('lessOrEqual' and 'equal')
- intersect object-geometry with a given rectangle to maintain a spatial access structure (Field Tree [Frank 1983], similar to Grid File [Nievergelt 1984] or EXHASH [Tamminen 1982])

- computation of an integer for values used for a hashing based index
- conversion to a string for each field (used for a basic output function)

Every DB-object has to provide the necessary operations which are mapped to the formal operations used in the generic definitions of the DBMS operations. We have found that most of these provided operations are very simple to write. Indeed, the code is very regular and can be produced automatically from a higher level description of the objects. In some cases, the programmer needs to write special code, different from the one automatically produced. For example this is the case, when the operation uses a value which is not a component of the object, but must be derived from the components.

4. Integration of DBMS with Object-Oriented Programming System

Following the proposal made, an object-oriented DBMS can be written as an application level package. This is an attractive alternative to the systems that include DBMS functionality within an object-oriented language [Penney 1987]. Most software engineering arguments speak for the former solution:

- The language is not burdened by DBMS functionality, and if DBMS support is not necessary for an application, it need not be included.
- The DBMS is integrated using standard language features. No new constructs need to be learned by the application programmer, and a mismatch between language and DBMS concepts can be avoided.
- The DBMS package can be selected and tailored specifically for an application class without requiring changes in the language.

The latter argument is important for the non-standard applications we work on (engineering databases, spatial information systems, cad/cam, etc.). They pose specific requirements for a DBMS: Depending on the application area, a DBMS must provide special functionality (eg. access based on spatial location), and its implementation must be adapted to yield the performance necessary.

5. Conclusions

We have shown how integration between an object-oriented DBMS and an application program can be achieved using inheritance. A language with multiple inheritance and genericity allows to define the data to be stored as modules which inherit DBMS functionality:

- Multiple inheritance is necessary, in order to allow objects to inherit the DBMS operations as part of their behavior (and still inherit other traits).
- Generic class descriptions with formal operation parameters to document the operations each subclass of the generic DBMS object must provide in order for the DB operations to work.

Data to be stored is defined as subclasses of the 'generic DB-object' that inherit DBMS functions. These subclasses must export a set of operations to be used by the DBMS. The major advantage of the proposed method is that the interface between the DBMS code and the objects is clearly documented: The DBMS code shows which operations each DB-object must provide. These operations are the only interfaces between DBMS and object internals, thus respecting object-oriented philosophy. No DBMS specific features are necessary in the language and the DBMS can be written as an application level program, without dependency between object-oriented language and DBMS.
We believe, that multiple inheritance and genericity will be generally useful for building large software systems, especially information systems. Similar to DBMS, other functions can also be factored out and one would like to use other pre fabricated packages for presentation graphics and window management, for geometric data processing etc. [Smith 1986] [Sandberg 1986].

References

[Batory 1984] D.S. Batory and A.P. Buchmann. Molecular Objects, Abstract Data Types, and Data Models: A Framework. In: 10th VLDB conference, Singapore, 1984.

[Cardelli 1985] L. Cardelli and P. Wegener. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, 17(4), April 1985.

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

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

[Härder 1985] T. Härder and A. Reuter. Architecture of Database Systems for Non-Standard Applications (in German). In: A. Blaser and P. Pistor, editors, Database Systems in Office, Engineering, and Scientific Environment, Springer Verlag, New York (NY), 1985.

[Härder 1987] T. Härder, K. Meyer-Wegener, B. Mitschang, A. Sikeler. PRIMA - a DBMS Prototype Supporting Engineering Applications. In: 13th VLDB conference, Brighton (England), 1987.

[Härder 1988] T. Härder, B. Mitschang, H. Schöning. Query Processing for Complex Objects. submitted for publication, 1988.

[Meyer 1986] B. Meyer. Genericity versus Inheritance. In: OOPSLA '86, Portland (Oregon), 1986.

[Nievergelt 1984] J. Nievergelt et al. The GRID FILE: An Adaptable, Symmetric Multi-Key File Structure. ACM Transactions on Databases, 9(1), 1984.

[Olthoff 1986] W.G. Olthoff. Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience. In: OOPSLA '86, Portland (Oregon), 1986.

[Penney 1987] D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In: OOPSLA '87, Orlando (Florida), 1987.

[Sandberg 1986] D. Sandberg. An Alternative to Subclassing. In: OOPSLA '86, Portland (Oregon), 1986.

[Schaffert 1986] C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt. An Intorduction to Trellis/Owl. In: OOPSLA '86, Portland (Oregon), 1986.

[SIGMOD 81] Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling. Pingree Park (Colorado, June 1980), SIGMOD Record, 11(2), February 1981.

[Smith 1986] R.G. Smith, R. Dinitz and P. Barth. Impulse-86: A Substrate for Object-Oriented Interface Design. In: OOPSLA '86, Portland (Oregon), 1986.

[Sutor 1987] R.S. Sutor and R.D. Jenks. The Type Inference and Coercion Facilities in the Scratchpad II Interpreter. In: SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, St. Paul (Minnesota), 1987.

[Tamminen 1982] M. Tamminen. Efficient Spatial Access to a Data Base. In: ACM-SIGMOD, Orlando (FL), 1982.

[Wegner 1987] P. Wegner. Dimensions of Object-Based Language Design. In: OOPSLA '87, Orlando (Florida), 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.