Issues in the Design of Object-Oriented Database Programming Languages

Toby Bloom
MIT Laboratory for Computer Science

Stanley B. Zdonik
Brown University
Department of Computer Science

Abstract

We see a trend toward extending object-oriented languages in the direction of databases, and, at the same time, toward extending database systems with object-oriented ideas. On the surface, these two activities seem to be moving in a consistent direction. However, at a deeper level, we see difficulties that may inhibit their ending up at the same point. We feel that many of these difficulties are a result of the underlying assumptions that are inherent in the fields of programming language and database systems research. Many of these assumptions are historical and contribute to a set of cultural biases that often prevent the two communities from interacting as effectively as possible.

The purpose of this paper is to try to uncover some of the cultural presuppositions that have inhibited development of a fully integrated database programming language. We have identified database and language features that seem to be difficult to reconcile. We try to uncover the basic problems in these two areas that these features were intended to solve. In order to resolve these problems, we attempt to distinguish fundamental differences from historical artifacts.

1. Introduction

The database and the programming language communities seem to be moving toward each other in terms of the problems that they are addressing. Database systems have been attempting to increase their power by associating more and more functionality with the data. Object-oriented databases, in particular, differ from their more conventional relational counterparts in that they incorporate notions of type, data abstraction, and inheritance [19,35]. These ideas have been studied extensively in the programming language domain for some time. Conversely, many modern programming language research efforts [2,18,22] are attempting to add notions of persistence and sharing, two ideas that are fundamental to databases. A result of these activities would be a database programming language in which there is a single model of data for both persistent and non-persistent data.

The fields of programming language design and database system design are both quite broad. Different styles of system have developed in response to different application needs. For example, languages like FORTRAN are suited to scientific and numerical applications while various dialects of LISP were designed to meet the needs of symbolic computation. Similarly, CODASYL database systems have addressed Cobol-compliant business applications while relational databases and their query languages were developed as decision support tools. The comments in this paper relate to merging two very specific subparts of these communities. We are concerned with languages that are designed to support software engineering and programming-in-the-large (e.g., CLU) and databases that are concerned with complex, non-standard applications such as software environments (e.g., ENCORE).

A result of these activities could be a *database programming language* in which there is only a single type system for both persistent and non-persistent data. There are several benefits of a database programming language. First, the programming process is simplified if the programmer does not have to be aware of two distinct systems, each with its own language. Second, we believe that there are opportunities for increased performance if a single system manages storage. That system can apply optimizations that the separate systems approach might not have available.

This work was supported in part by DARPA under Contract No. N00014-83-K-0146 and DARPA Order No. 4786, in part by NSF under Contract No. DCR-8605597, in part by ONR under contract No. N00014-86-K-0621 and in part by IBM research contract 559716.

There has been much previous work analyzing problems of integrating databases and programming languages [8,30,36]. [5] has suggested that the fundamental problem in these systems has been in making two very different type systems coexist. Until now there has been little agreement among the communities about what a suitable unified type model might be. The object-oriented data model seems to provide many of the characteristics needed by both databases and programming languages. For this reason, there seems to be great promise in marrying the two technologies.

Although the object-oriented model seems to solve the type problems described by [5], there are still differences that exist between object-oriented databases and object-oriented programming languages that need to be explored. In this paper, we begin by describing some of the basic assumptions that have influenced the direction of the two fields. On the database side these include persistence and fine-grained sharing, while on the programming language side they include reliable software construction and programming-in-the-large. We then discuss a number of areas in which these assumptions lead to differing and sometimes conflicting approaches.

2. Culture

In this section we will briefly sketch some of the assumptions that seem to permeate the world of databases and the world of programming languages. Many of these assumptions predate the development of object-oriented systems, yet we will argue that these assumptions are responsible for many of the problems that exist in creating database/programming language hybrids. These assumptions make perfect sense when each of the areas is examined in isolation. When one is trying to put the two areas together, however, one must be aware of the assumptions because it is often the clash between the two points of view that leads to difficulties.

2.1 Database Design Principles

Database system technology has primarily grown up in an environment of commercial data processing. Initially, databases were designed to address primarily accounting and bookkeeping applications in large organizations. The fundamental requirements of these applications are persistence and sharing of data. By persistence we mean that all objects remain in existence beyond the duration of the application program. By sharing in this persistent environment, we mean that independent, possibly concurrent applications may use the same data. All of the assumptions in this section stem either directly or indirectly from these two requirements.

Because independent applications needed access to the same shared data and because databases were not prepared to include programs, the basic architecture of database systems placed the data under the control of a separate process called the database management system (DBMS). In this environment, **data was separate from function.** Applications ran in a process of their own and accessed the data through a standard data manipulation language that expressed none of the semantics of the applications. This separation has profound implications on the functionality required of the database. As we shall see, these requirements include declarative constraint mechanisms, query languages, and transaction processing. It also leads to assumptions about the way in which a database is used, such as the notion that the data is under control of a centralized administrator.

A database system should be designed with the needs of all present and future applications of the organization carefully balanced. It is the job of the database administrator (DBA) to perform this task of **centralized development** The DBA is responsible for designing the logical views of the data as well as the underlying physical structures upon which these views are implemented.

The choices that are made at both the logical and the physical levels must be made very carefully. One application may need to have an employee file sorted by employee number, while another may need to have the same file sorted by job category. These conflicting requirements are mediated by the DBA. The DBA uses additional information about the relative priorities of these applications to make these decisions.

It is important to notice that decisions like these are made somewhat independently from the application programs that will make use of the data. The cultural assumption is that the database is designed first. *Programs are written after the database has been designed,* and they must abide by the decisions of the DBA.

This is in sharp contrast to the programming language view of the relationship between data and programs. Programming languages lack the notion of external control of the data. All data is considered to be internal to some module of a

program. An application program is free to set up its environment in any way that optimizes its own requirements. The program comes first; the data is viewed as part of the program.

A database is intended to stand as a model of the application environment. There has been a movement within the database field to incorporate more and more **semantics** into the data model so that the database can more closely reflect an application. Constraints and automatic side-effects are examples of proposed database mechanisms that address this need. Some data models [14] go a long way in this direction by including, for example, inverse attribute declarations, derived attributes, and attribute-value derived subclasses. We will see that some of these mechanisms are difficult to integrate with programming language facilities.

Object-oriented databases extend data semantics by adding type extensibility. This is accomplished by including notions of data abstraction and type-specific operations, thus bringing databases closer to their programming language counterparts. The conflict between constraint-based semantics and type-based semantics remains.

Databases have traditionally addressed data-intensive business applications. Experience with these applications has shown that efficient handling of **large amounts of data** on slow secondary storage media is extremely important. The need for flexible control over storage management choices has led to a technology that is absent from programming languages. This technology includes explicit data clustering capability, indexes on collections (i.e. relations), and sophisticated query optimization techniques.

The types of applications for which databases were initially built can be characterized by large volumes of uniformly-structured data items, without much internal structure. As a result, the successful data models developed were similarly **uniform and simple** in structure. They tended to be record-oriented, where the contents of a field was a single atomic piece of data. This simple model allowed for development of such features as ad-hoc query languages, query optimization, constraint checking, and control over underlying storage management. As we will discuss in section 3, these features do not easily carry over to more complex models.

2.2 Language Design Principles

Programming languages have historically focussed on **processing**, rather than on data. The concern has been with improving the tools available to programmers to build complex systems involving large amounts of code, and relatively small amounts of data (compared with databases). On the other hand, languages have been incorporating support for more complex structuring of data, including graphical structures and complex inter-relationships, for some time. The emphasis has clearly been on **local data** however. The language facilities provided for handling large amounts of data on secondary storage have been minimal. They are usually limited to file-level access leaving the programmer with the responsibility of managing the organization of these complex data objects in the files. Sharing of data among programs has in general been accomplished through sharing of these files, with none of the support provided by databases.

Object-oriented programming has shifted the focus considerably closer to that of databases, by emphasizing the organization of software around the data objects, rather than around flow of control. Note however, that there is still a noticeable difference between the emphasis in object-oriented programming languages and that in databases, even object-oriented databases. While the languages now focus on the data objects, the important aspect of objects is what *behavior* the objects exhibit: what operations they provide or what messages they accept. In databases, the emphasis remains on applications that require large amounts of shared, persistent data. Behavior includes the operations but also extends to other kinds of declarative semantics.

Overall however, we see that object-oriented languages and object-oriented databases are natural complements of one another: one emphasizing processing, complex structuring and local data; the other focussing on a more declarative approach, shared data outside the domain of applications, and support for very large amounts of data. In the remainder of this section, we discuss the perspective, and set of language design principles, which language designers bring to a joint language/database design venture.

We believe that a clean fusion of languages and databases would greatly simplify the programming of certain applications. In particular, we are interested in large, complex applications that require shared, persistent data in a production setting. A schematic editor for electronic CAD is one such application. In a production programming environment, we are dealing with application development divided among a group of programmers. The life-cycle for these applications is usually quite long, and many modifications will be

made during its lifetime. Detailed specifications of the software exist and must be met by the implementation.

There is no single set of language design principles adhered to in all current language designs. Different languages focus on different aspects of the programming process, or on different application domains and hence consider different issues to be of primary importance. In this discussion, we focus on supporting large software projects in which the resulting software is to be used in a production environment. Because we are interested in producing an integrating database programming language, we look here at the principles by which language designers evaluate whether a construct should be added to a language, and what combination of constructs is sufficient. We do not claim this list to be exhaustive; we present those guidelines we have found to be at issue in our discussions of database programming languages.

The object-oriented approach provides for modular construction and independent interfaces so important for programming in the large. **Unambiguous specifications** of these interfaces is an important goal in the design of languages for large-scale production programming. Bringing the level of programming closer to that of the application domain by building application-specific types lowers the complexity the programmer must handle by making the translation from application requirements to code easier.

Reliability is enhanced because only operations in the type module can access the object representation, and so it is easier to guarantee that data isn't corrupted accidently. Abstraction also helps to ensure that local modifications can be made without unexpected side-effects in other parts of the application. In building production software, it is important to understand all of the interactions among different parts of the application, so that unexpected side-effects don't arise.

When evaluating whether a given construct should be included in a language, several criteria are used. One is the utility of the mechanism. A language should incorporate a **few, general constructs**, rather than many special-purpose ones. In general, there is no need for two different ways to do the same thing. Mechanisms should be added to the language only if they provide expressive power sufficiently greater than that which can be achieved in their absence.

Another consideration is the **interaction with other constructs** in the language. Different mechanisms should not interact in complex or unexpected ways. It should be possible to describe the meaning of one construct independently of others. If including a construct changes the meaning of other mechanisms when used together, the complexity of the interactions must be carefully considered. Finally, language constructs are often evaluated in terms of how easy they are to use, or how prone they are to misuse or error.

These principles, among others, come into play when we consider merging common programming language and database features into a unified database programming language. Later in this paper, we discuss several standard database features in terms of their interaction with standard language features. We use these criteria to analyze the conflicts that might stand in the way of integrating the two worlds.

3. Comparison

While the database and programming language fields agree on high-level goals such as correctness and efficiency, differing requirements have led to different cultural assumptions. As a result of such differences in philosophy, mechanisms developed within the two areas that do not interact well. This section presents the major areas in which database and programming language assumptions lead to conflicts.

3.1 Triggers and Constraints

One aspect of database technology that needs to be reexamined is the role of declaratively specified knowledge about the application in the form of *constraints or triggers*. This knowledge is specified with the data and will be monitored by the database system at all times. A trigger can be defined as a predicate and a body. The meaning of a trigger is intuitively whenever the predicate becomes true, execute the body. A constraint is intended to describe a predicate that must always be maintained. One could think of a constraint as a trigger whose predicate is the negation of the predicate that is to be maintained and whose body raises an exception. In summary, we have:

Trigger: Predicate and body
Constraint: Trigger with raise exception as body

Constraints might be expressed in terms of predicate calculus, or they might be captured in

some specialized part of the DDL (data definition language).

As an example, consider an employee database, with employee, and department types. With a constraint mechanism, we might choose to express the fact that no employee can make more than his/her manager.

```
[For all e in employee]
        (e.salary < e.dept.manager.salary)
```

A constraint like this causes a check every time the employee salary is changed, an employee is added, a manager is changed, a manger salary is changed, or an employee is moved to a new department. The system can determine when checks should be performed, so that it is not necessary at commit of every transaction.

In a language without a constraint mechanism, every operation that might violate a constraint must explicitly check that constraint. In languages that provide exception handling, the possibility of violating a constraint appears in the interface of the operations as well. For example:

```
raise_salary =
        operation(e:employee,
                  new_salary:integer)
        signals (too_high)
                if new_salary >
                        e.manager.salary
                then signal too_high
                else e.salary := new_salary
        end
```

An example of a trigger that is supported by some semantic data models is that of an inverse relationship. Here we might declare that employees are related to departments by a *works-in* relationship and departments are related to employees by an *employs* relationship. We might declare that these two relationships must always be connected in the obvious way.

```
Type employee
        works-in: department

Type department
        employs (inverse works-in):
                        set of employees
```

As a result, changing the *works-in* relationship for an employee e to be department d would automatically update the *employs* relationship for d to include the employee e.

Triggers and constraints are important tools in database design. They ensure that the many applications running on the database cannot violate the integrity of the data. The mechanisms are declarative and associated with the data, not the processing. This structure allows the database to maintain integrity constraints across various parts of the schema even though various applications use views that hide some of those related parts.

The trigger mechanism also performs some of the job that applications programmers would otherwise have to assume. For example, when the value of an employee's *works-in* relationship is changed to be shoe-department, the database system automatically inserts that employee in the set of employees that is the value of the *employs* relationship for the shoe-department object.

The notion of associating constraints with the data, and declaring them only once is also appealing to language designers. Hence, the fundamental notion of constraints is not at odds with language design goals.

The trigger mechanisms seem less useful, in that programmers will have to be aware of what relationships are maintained automatically and which are not, so as to write correct implementations of the operations. Performing the update automatically thus does not free the programmer from having to pay attention to the relationships among the data objects. Not only could updates be missed if the programmer assumes a trigger exists when it does not, but also, there might be situations in which the programmer explicitly performs an action that is then performed again by a trigger. While this can be assumed to be an error in the trigger's predicate, it is unclear how prone to such errors a trigger mechanism would be as part of a complete programming language.

Constraint mechanisms might also interact with exception handling in some languages. In databases, typically, if a constraint is violated, then the transaction responsible is aborted. In languages that provide support for exception handling, the constraint violation would typically signal an exception, in addition to, or instead of aborting the transaction. (While the transaction cannot commit in that state, raising an exception gives the caller a chance to correct the problem and continue.) Our requirements for well-defined module interfaces indicate that if a given operation might signal an exception, that information should be specified in the interface, so that callers can be prepared to handle the expected exceptional conditions. This requirement conflicts with the requirement that

constraints be specified in only one place. In the example above, the information about the given constraint must appear in the interfaces to operations raise_salary, change_dept_manager, change_emp_dept, hire_emp, and probably others. Language designers consider it important from a methodological viewpoint that programmers be aware of which operations might violate which constraints, and of what the requirements are for arguments to those operations. A similar conflict arises with specifying the side-effects that might result from triggers being activated as a result of invoking an operation. We are therefore left with a question of how a constraint mechanism, and possibly a trigger mechanism, can be designed to be part of a database programming language in a way that preserves the declarative nature of the mechanism without conflicting with requirements for operation specification.

Finally, we have to examine how these mechanisms will function in an object-oriented environment. It is essential to the practicality of the declarative mechanism that static analysis can be used to determine when the predicate of a trigger or constraint need be checked. It is infeasible to check every trigger before commit of every transaction, or (worse yet) before any activity might see an inconsistent state that should have been detected by a trigger. In an object-oriented environment in which any operation may be user-defined, and the system cannot automatically determine which operations can see the effects of which others, or what the side-effects of a given operation might be, is it possible to sufficiently narrow the points at which trigger predicates must be checked? This issue is not one of conflicting language and database principles; it is an issue of how to extend a relational database mechanism to an object-oriented environment.

Thus, we have identified two problems here that must be addressed in designing a database programming language: one is an implementation issue, minimizing trigger and constraint checking, and the other a design issue, handling the interaction between exception handling mechanisms and constraint mechanisms.

3.2 Query Optimization

When dealing with large amounts of data, it becomes imperative to be able to search the database as efficiently as possible. The relational database community depends heavily upon automatic query optimization for this purpose, and there is concern about any model that does not support such optimization.

In moving toward an object-oriented model, some of the properties upon which query optimization depends have changed. In particular, in relational databases, query optimization depends upon the algebraic properties of the relational operations, as well as on the physical structure of relations. For example, it is known that *join* and *select* commute. The uniform structure across all data also allows heuristics such as selecting first on fields for which indexes exist, thus narrowing the set over which a join must be performed.

In an object-oriented system, there is no standard set of operations across all data, since operations are defined on a per-type basis. The definition of "=" can be different for different types. Furthermore, there is no way to determine in general whether any two operations commute. Hence, the standard query optimization techniques are not useful.

On the other hand, it is difficult to compare efficiency in the two models. Undoubtedly, in the object-oriented model, there will be cases in which direct links between two kinds of objects exist, where previously a join would have been necessary. The more sophisticated structures possible in the new model may thus make some optimizations unnecessary. It is also possible to design type-specific optimizations and implement them as a part of the operation code. This requires more programmer time, but can take advantage of type-specific information like commutativity and other semantic knowledge. Finally, we do not know the utility of standard compiler optimization techniques in the database programming language world.

Overall, it is likely that more work will be needed to find appropriate optimization techniques for this environment. However, we do not yet understand the requirements in this domain.

3.3 Persistent Data

Persistent data is a fundamental property of database systems. Persistence is used by databases in a number of ways. Data is kept external to applications, in non-volatile storage, so as to divorce the data's lifetime from the lifetime of any application. The data can be shared among applications concurrently, or sequentially. Applications are thought of as short-lived; any state needed for longer than the lifetime of an application process is stored in the database.

The notion of persistent data is relatively new to languages. Historically, languages have

concentrated on supporting manipulation of data local to a process, and often provide loopholes for maintaining data for longer than a process lifetime. These loopholes usually consist of file access operations. Data can be moved into files as "uninterpreted bags of bits"; the application is responsible for translating from the file format back into the form used for local data. Overall, languages have provided little support for long-lived data.

The lack of support for persistence has been recognized as a shortcoming of language designs, and we are now seeing languages that address the problem. The model of persistence in various languages does not always directly match the database model. PS/Algol [3] supports persistence primarily for "process continuity". The aim seems to be to allow the user to terminate a session and continue later from where the application left off. In some sense, the goal is to support long-lived processes. The fine-grained concurrency control needed for sharing data among concurrent applications is lacking, at least in the early versions of which these authors are aware. It is therefore assumed that the goals differ some from those of traditional databases. The language allows for declaring data to be persistent, and the system handles the transfer between volatile and non-volatile storage, so the application program is freed from having to translate between file format and local format.

Argus is another language that has addressed the issue of persistence. Argus still treats data as local to applications. However, applications are now seen as long-lived, and hence the data that constitutes the application's state must be long-lived. Of course, stable storage and transaction facilities are needed to support the appearance of application processes that exist forever, in spite of crashes and various other failures. The difference between this view and the database view is in whether the static data or the active application process is the persistent entity. The database view could be described as a centralized, persistent database manager process whose state is the entire database, and hence is not inherently inconsistent with this language view. In practice, differences arise because a single database manager has little semantic knowledge about the data, whereas when data is controlled by the application level, more semantic information is available. One example of this difference is that in Argus, it can be determined when a persistent object is no longer accessible from the application and can be removed from stable storage or garbage collected. When the data management is separated from the use of the data, the system cannot make such deductions. Overall, the approach of considering data to be local to a persistent process is consistent with an object-oriented approach. Since the operations on objects should be associated with the objects, and all access to an object should be through invocation of its operations, objects and their operations can be encapsulated in an active entity which mediates access to the objects and defines the object's lifetimes.

We thus have seen different uses for persistent data, and somewhat different structuring of the control over persistent data. It must be decided which model is best suited to object-oriented database languages.

3.4 Large Amounts of Data

The language approach in which data objects are treated as local to a persistent process was not designed with the goal of handling large amounts of data. There are implementation problems that arise in supporting large databases with this structure, and some of those problems are not as yet solved. The problems we describe here are primarily those that have arisen in the first Argus implementation, since that is the one case with which the authors have detailed knowledge of implementation. These problems are very similar to those encountered by database designers a decade ago, when they started addressing the issues of very large databases. They discovered that to get efficient access they had to circumvent the operating system and handle their own disk accesses. Persistent languages are finding similar problems with virtual memory.

In a database system, the primary copy of the data is in stable secondary storage. Updates are maintained in a log, and when recovering from a crash, the primary copy is returned to a consistent state by traversing the log and undoing updates from incomplete transactions then redoing transactions that have been committed, but whose results had not yet been written to the primary copy.

The Argus implementation, since it considers the data to be process state, keeps the primary copy of the data in virtual memory. Stable copies are kept for use in case of a crash. The major problem with this structure is the efficiency of recovery after a node crash. When the primary copy is in virtual memory, and the system considers virtual memory to be volatile in a crash, then the entire state must be reconstructed in virtual memory from some combination of checkpointed state and log

information. For large states, this process can be very slow. Realize however, that most of the data in virtual memory will actually be intact on a disk (in swap space) after a crash. Work is currently underway on how to recover virtual memory following crashes [16]. The problem of very large virtual memories still exists. Boundaries in real systems must be dealt with. Argus currently runs up against limits in the size of swap space that limits how much data can be stored at a single node. There are also limitations in the ability to cluster data on physical storage so as to improve access time.

There are a number of alternatives for solving these problems, but investigation is still underway. Until the issues are resolved, a virtual memory model for large amounts of data is unsuitable. For a database programming language, we need to understand better how solutions from the database world might be applied.

3.5 Classes

The notion of *class* has a different meaning and different usage in the database and language communities. Language designers tend to use the word class to mean data type. A class is the definition of the behavior and structure of a set of objects. In the database view, class is used to refer to the set of objects themselves rather than, or in addition to, such a definition. Some database languages, like Galileo, include both types and classes, so as to distinguish between the collection of objects and their definition. Merging the two notions of class leads to difficulties, which we discuss in this section.

In the database view, classes are used as a *primary* means of grouping objects and searching for objects [14]. In some sense, classes take the place of relations. There is an assumption that there must be system-supported mechanisms for grouping objects into collections automatically. It is considered an undue burden on the programmer to have to create and manage such collections. It is important that the system automatically insert objects into appropriate collections at their creation. Furthermore, it is thought to constrain ad-hoc querying if the only way to access objects is through collections anticipated by the type manager or other applications. Automatically supported class/collections enable applications to reach any existing objects of interest, without acquiring explicit references to those objects. Assuming objects are grouped into these classes (i.e., collections) automatically also allows some query optimization techniques to be carried over from relational databases. In particular, the type definer (akin to the database administrator) can determine the best access paths. For example, the type designer can declare that the class should be stored as a B-tree with appropriate keys.

Programming languages have not typically thought of classes as providing access to all of the objects of the class. From a language design viewpoint, several questions arise about use of such a mechanism, whether as part of the existing class-as-type, or as a parallel construct. We examine three issues here:

(1) Is such a mechanism needed; do the benefits justify the added complexity?
(2) Does class-as-collection interfere with the use and meaning of class-as-type?
(3) What is the interaction with other language facilities; in particular, can languages support automatic storage management (with garbage collection) if class/collections are supported?

Providing a collection associated with each type, without a built-in mechanism requires explicit action in every type definition. Every creation operation (in Smalltalk, the *new* operation), would have to explicitly insert the new object into the collection. And, initially, the collection would have to be explicitly created as well. A type definer could of course choose not to provide such a collection.

From the database perspective, relieving the type designer from this chore is more than just a convenience; it is a kind of integrity constraint. It ensures that a class is always present for all types, and this class will be used by ad hoc queries. From the programming language perspective, it seems like a good idea to let the type definer decide whether objects of the type should be accessible in this manner.

Does providing access to all objects of a type via such a class mechanism interact with the use of types as definitions? There are two potential problems here. One is a methodological consideration. Providing these special collections encourages programmers to depend on these standard groupings instead of deciding how best to organize the objects used in given applications.

The second issue is one of abstraction and the scope of type definitions. Language designers tend to think of type definitions as global in scope. There is little need to hide the definition of a type as long as the scope of given objects of a type can be controlled. Globally available type definitions enhance reusability of code and increase the

benefits of modularization. However, if all of the objects of a type are accessible through the type (or an automatically associated class object), then we might inadvertantly allow access to objects that are serving as representations (i.e., analogous to the instance variables). This would allow users to break data abstraction.

For example, suppose we define a *hash-table* type. Any application that requires a hash table should be able to use that definition. However, there may be no meaningful relationship between hash tables in different application domains. Why should it be possible to locate all hash tables through a *hash-table* class object? In addition to not always making sense, it provides a protection problem in that one application can get to the underlying representation of another application's data. If *hash-tables* are being used to represent *ordered-sets*, it is possible to access a *hash-table* that should only be visible through the *ordered-set* interface. It violates abstraction in that there now exists a path to the representation of an object that does not go through the abstract layer.

From the database perspective, the proper way to protect objects from such unauthorized access and solve the problem of not seeing objects that make sense to a particular application is through a view mechanism. To solve the problem of violating abstraction, we need some kind of scoping mechanism that only allows appropriate class members to be visible. For example, inside the *ordered-set* class definition, one sees hash-tables that are representations of *ordered-sets*. Outside the class only the hash-tables that were created as direct instances would be visible. More work is needed to determine how a view mechanism can be used to guarantee levels of abstraction in access through classes.

It also seems clear that part of the conflict in goals here stems from different historical assumptions about the types involved. In the database world, the types are usually domain-specific and higher-level, like type person, or type ship, and abstraction issues don't arise in these "top-level" types. It is only recently with the emergence of object-oriented databases that the notions of abstract and concrete types have appeared as a part of the database culture.

The final language design concern is an implementation issue, but if unresolved, will be reflected in language semantics. Experience has shown that automatic storage management, and in particular automatic storage reclamation are crucial for managing software complexity, enhancing software reliability and reducing programming time. Without automatic storage management, dangling pointers cause hard-to-find bugs, and programmers waste large amounts of time figuring out the best storage management strategies for every program. When every object is always accessible through its class, garbage collection is impossible, because objects never become inaccessible. Various strategies to overcome this problem have been proposed, but to the authors' knowledge none avoid anomalous behavior and retain the benefits of the class/collections.

One solution is to remove the class pointers as roots for the garbage collector - then if no other references to an object exist, the object can be reclaimed. Unfortunately, this also eliminates the ability to depend on the class for access to the objects, since an object might disappear directly after it is created. If you have to keep duplicate pointers, then the utility of the class mechanism drops substantially. Also, with such a scheme, it becomes impossible for an application to ensure that an object is made inaccessible.

From the database viewpoint, explicit deletion is an acceptable alternative; from the language viewpoint, it is not. Possibly, this difference is due to the fact that in relational databases, the only access to an object (i.e. record) was through the relation, and hence dangling references did not exist, and it could always be determined if a referenced object no longer existed. Such is not the case in programming languages. (While perhaps not entirely infeasible, maintaining information on deleted objects presents formidable implementation issues.)

Hence, trying to merge class-as-type with class-as-collection provides another example in which related features from the two worlds are difficult to combine because fundamental principles are violated by the combination, if not by each feature in isolation.

4. Conclusions

This paper describes some of the cultural assumptions that stand in the way of a seamless integration of databases and programming languages. Such a hybrid must retain the advantages of both styles of system. An understanding of both cultures is essential if the issues raised by the cultural clashes are to be resolved.

We have tried to raise the issues that need to be considered. We have described these issues and

some of the conflicts that arise from them. This paper is intended as a springboard for further discussion. As such, we have tried to crystalize the arguments on both sides.

It appears that none of these problems is insurmountable, however, it is our feeling that more research is needed to determine the right design decisions that properly balance requirements from both communities.

5. Acknowledgements

The authors wish to thank Bruce Bullis, Denise Ecklund, Earl Ecklund, Barbara Liskov, David Maier, Patrick O'Brien, Craig Schaffert, Bill Weihl, Peter Wegner, and David Maier's Friday morning database discussion group at OGC for their helpful comments and suggestions.

6. References

[1] Antonio Albano and Luca Cardelli and Renzo Orsini, Galileo: A Strongly-Typed, Interactive Conceptual Language, ACM Transactions on Database Systems, 10(2):230-260, June, 1985.

[2] M. P. Atkinson and P. Bailey and W. P. Cockshott and K. J. Chisholm and R. Morrison, Progress with Persistent Programming, Databases - Role and Structure: An Advanced Course. Cambridge University Press, Cambridge, England, 1984, pages 245-310.

[3] Malcom Atkinson, et.al, PS-Algol: an Algol with a Persistent Heap, Sigplan Notices :24-30, July, 1982.

[4] M.P.Atkinson, et.al, An Approach to Persistent Programming, The Computer Journal 26(4):360-365, 1983.

[5] M.P.Atkinson and O.P. Buneman. Database Programming Language Design, Technical Report, University of Glasgow, October, 1985.

[6] Baroody, Jr., Anthony James, The Evaluation of Abstract Data Types as an Implementation Tool for Database Management Systems, PhD thesis, University of Wisconsin - Madison, 1978.

[7] Alan Borning THINGLAB: A Constraint-Oriented Simulation Laboratory, SSL-79-3. Xerox PARC, July 1979.

[8] Peter Buneman, Can We Reconcile Programming Languages and Databases?, Databases - Role and Structure: An Advanced Course. Cambridge University Press, Cambridge, England, 1984, pages 225-243.

[9] Luca Cardelli, Amber, Technical Memorandum TM 11271-840924-10, AT&T Bell Laboratories, 1984.

[10] Ole-Johan Dahl and B. Myhraug and K. Nygaard, The Simula 67 Common Base Language, Technical Report S-22, Norwegian Computing Center, Oslo, Norway, 1970.

[11] L. Peter Deutsch, Constraints: A Uniform Model for Data and Control. SIGPLAN Notices 16(1):118-120, January, 1981 also Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling, June, 1980.

[12] Peter Deutsch. Efficient implementation of the Smalltalk-80 System, in Conference Record of the 11th ACM Symposium on Principles of Programming Languages, pages 290-296. ACM, Jan., 1984.

[13] Adele Goldberg and David Robson. Smalltalk-80, The Language and its Implementation, Addison-Wesley, Reading, Mass., 1983.

[14] Michael Hammer and Dennis McLeod. Database Description with SDM: A Semantic Database Model. TODS , Sept., 1981.

[15] D. H. H. Ingalls. Design Principles Behind Smalltalk. BYTE 6(8):286-298, August, 1981.

[16] Elliot K. Kolodner, Recovery Using Virtual Memory, S.M. Thesis, MIT, June 1987.

[17] Barbara Liskov and Alan Snyder and Russell Atkinson and Craig Schaffert. Abstraction Mechanisms in CLU. Communications of the ACM 20(8):564-576, August, 1977.

[18] Barbara Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler and W. Weihl, Argus Reference Manual, Programming Methodology Group Memo 54, MIT-LCS, March, 1987.

[19] David Maier et al, Development of an Object-Oriented DBMS, in OOPSLA86, pages 472-482. ACM, Oct., 1986.

[20] David C. J. Matthews, Poly Manual. SIGPLAN Notices 20(9):52-76, September, 1985.

[21] James Morris, Jr., Types are not sets. in Conference Record of ACM Symposium on Principles of Programming Languages, Boston, Mass., pages 120-124. ACM, October, 1973.

[22] Patrick O'Brien, Bruce Bullis, and Craig Schaffert, Persistent and Shared Objects in Trellis/Owl, Technical Report DEC-TR-440, Digital Equipment Corp., July, 1986.

[23] Brian Oki, B. Liskov, and R. Scheifler. Reliable Object Storage to Support Atomic Actions, In Proceedings of the Tenth Synposium on Operating Systems Principles. Nov., 1985.

[24] Paolo Paolini, Abstract Data Types and Data Bases, PhD thesis, University of California, Los Angeles, 1981.

[25] Lawrence A. Rowe, Issues in the Design of Database Programming Languages, SIGPLAN Notices 16(1):29-35, January, 1981.and Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling, June, 1980.

[26] Lawrence A. Rowe and Kurt A. Shoens. Data Abstractions, Views and Updates in RIGEL, In Proceedings of the ACM- SIGMOD Conference on Management of Data, Boston, Mass., pages 71-81, May, 1979.

[27] Craig Schaffert, et al, An Introduction to Trellis/Owl, In OOPSLA '86, Conference Proceedings. SIGPLAN, Sept, 1986.

[28] Joachim W. Schmidt and Manuel Mall. Pascal/R Report, Technical Report IFI-B-66/80, Fachbereich Informatik, Universitat, Hamburg, January, 1980.

[29] Joachim W. Schmidt and Manuel Mall. Abstraction Mechanisms for Database Programming, SIGPLAN Notices, 18(6):83-93, June, 1983. Proceedings of the SIGPLAN '83 and Symposium on Programming Language Issues in Software Systems, San Francisco, CA.

[30] Mary Shaw and William A. Wulf. Toward Relaxing Assumptions in Languages and Their Implementations, SIGPLAN Notices 15(3):45-61, March, 1980.

[31] Alan Snyder, Encapsulation and Inheritance in Object-Oriented Programming Languages. SIGPLAN Notices 21(11):38-45, November, 1986, OOPSLA '86 Conference Proceedings, N. Meyrowitz (editor), Sept 1986, Portland, Oregon.

[32] Guy Lewis Steele, The Definition and Implementation of a Computer Programming Language Based on Constraints. MIT Artificial Intelligence Laboratory, TR-595, August, 1980.

[33] Norihisa Suzuki and Minora Terada. Creating Efficient Systems For Object-Oriented Languages.In Conference Record of the 11th ACM Symposium on Principles of Programming Languages, pages 290-296. ACM, Jan., 1984.

[34] Anthony I. Wasserman and David D. Sheretz and Martin L. Kersten and Reid P. van de Reit and Mark D. Dippe. Revised Report on the Programming Language PLAIN, SIGPLAN Notices 16(5):59-80, May, 1981.

[35] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In Proceedings of the 19th Annual Hawaiian Conference on Systems Science. Jan., 1986.

[36] Stephen N. Zilles, Types, Algebras and Modeling.SIGPLAN Notices 16(1):207-209, January, 1981. Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling, June, 1980.