

ACCELERATING YOUR OBJECT-ORIENTED DEVELOPMENT

Thad Scheer and Theresa Smith

A Product of the Gaithersburg Software Technology and Architecture Center

Lockheed-Martin Mission Systems - Gaithersburg, MD

1999

An Objectivity, Inc. White Paper

www.objectivity.com

Abstract

This paper is an overview of the issues that arise from implementing object persistence with a relational database. The basis for this paper is our recent experience with Object-Oriented projects that used relational database technology. Those experiences have shown us that the cost of mixing the two paradigms is very high and can seriously detract from the benefits of Object-Oriented development. There is no question that object-to-relational solutions can be made to work, but at what price? This paper describes approaches to building object-to-relational bridges and highlights problems and compromises that typically arise. This is not a guide to implementing an object-to-relational bridge; instead, its purpose is to explain some fundamental issues and provide information that would be useful in making a database technology selection. This paper is most pertinent in the context of programming languages that have little or no native support for object persistence, such as C++.

Introduction/Background

Our intent with this paper is to show how object-to-relational solutions often deny projects many of the benefits of Object-Oriented development, such as flexibility, reuse, and simplicity. This denial is a natural consequence of object persistence in languages such as C++ and is difficult to overcome. Even with the proliferation of in-house object-to-relational solutions, it has seldom been our experience that object-to-relational translation is robust enough to adequately preserve the spirit of Object-Oriented development. This is not to say that these solutions do not work, that is if "work" is defined as "able to move data between an Object-Oriented application and a relational database". However, if "work" is defined more strongly to include assurances regarding the integrity of object associations, high performance, and preservation of the principle of implementation hiding these solutions typically fail. It should be noted that implementation hiding is a central goal in Object-Oriented software development, a principle that is difficult to preserve in an application that uses object-to-relational translation.

The two most common pitfalls of object-to-relational translation are 1) the bloat of programming not germane to the software's purpose, and 2) the introduction of explicit and implicit software dependencies due to translating complex relationships. Solutions that are naive to the special challenges of object persistence will aggravate this further, but the problems are mostly intrinsic and are difficult to avoid even under the most accommodating circumstances. In building object-to-relational translation software we are challenged to implement object persistence without violating the independence or cohesiveness of objects. The difficulties associated with doing this are exacerbated by a relational database's need to use foreign keys as a means of implementing relationships. Commercial tools offer a modicum of relief but many projects choose to build their own object-to-relational solution rather than buy one. Moreover, many architects will still choose building over buying, often due to having underestimated the complexity of this problem. Such projects tend to suffer with primitive and insufficient language bindings, a major source of unwanted software rigidity caused by object-to-relational solutions.

This paper concerns the use of a relational database to save and reconstitute objects from an Object-Oriented application. There are two ways to view this. The first is true persistent objects, where whole objects and their relationships must be persistent. Think of these objects as "sleeping" when they are in a database. The other is to assume that data are of primary importance, and objects are merely a necessary side effect of computation. This will be the case if a database schema is a controlled interface or is shared by several different software clients. Typically in these situations the data schema is a separately controlled design and the values in database tables have uses beyond the context of sleeping objects. In either of these cases a major challenge is to implement Object-Oriented software without allowing an object-to-relational solution to force undesirable coupling between software components at the application level of architecture. The issue cannot be sidestepped by assuming a database need only see objects as mere data; data that can be loaded into C-style structures and persisted by some kind of "access module" or "access class". Trying to think of objects as merely data is not a solution, mostly because there are no good methods of extracting the data in a way that is transparent to the objects. At least when a project admits to having persistent objects, a commercial tool can be used and will alleviate some of the problems inherent with object-to-relational translation. Moreover, the semantics of an object association makes attempts to handle objects as "strictly data" impractical.

An important difference between the Object paradigm and the Relational paradigm is that objects like to hide their data attributes, and relational databases like to expose theirs.[2] The hiding of data attributes in O-O constitutes information/implementation hiding, an important quality of software design. Relational databases, however, depend on

having open access to data attributes, such access is required to support ad hoc cross-table queries. This and other semantic differences between O-O and Relational are cumulatively referred to as impedance mismatch. The way we use the term, impedance mismatch refers to a semantic difference between object structure and relational structure. It is our experience that the project level risks from impedance mismatch are usually underestimated. Many of the problems outlined in this paper result from the need to overcome impedance mismatch, and the difficulties in doing so.

Object Persistence is not a Service

Programmers usually access a database through a function call library included with the database package. It is typical for these libraries to differ between database brands. To insulate an application from a particular database product and allow migration, it is often wise to insulate from the database library with a service layer. Application code uses a service layer for all database services, and the service layer bridges to the product specific library calls.

It is common to extend this concept by wrapping the normally flat database services with objects. This gives a database an Object-Oriented look and feel to a programmer. Instead of calling free functions, programmers use objects that represent important database abstractions, such as table, row, and query.

Persistent services do not constitute object persistence, even if implemented as objects. Object persistence is about saving and reconstituting application objects, not about the services that do so. It is not sufficient to address object persistence by wrapping a database interface with an Object-Oriented facade. Moreover, implementing object persistence via functional services can violate the privacy of objects by requiring data to be moved between object attributes and persistence services. As a rule, no software outside an object's own class should use the private attributes of the object. Unfortunately, this is usually unavoidable when persistence is implemented as a service, where it is common to use internal object attributes, or to replicate attributes, in database aware logic. Acts as these violate the principles of information hiding and contradict the goals of Object-Orientation. The result is limited opportunities to reuse and modify objects.

Unfortunately many software architects remain steadfast in considering persistence an infrastructure service. To further the false notion that persistence is a service, there are many commercial and public domain class libraries that masquerade as object-to-relational but are only service layers. These products provide Object-Oriented access to relational databases (examples are RogueWave's DBTools.h++ and Microsoft's DAO). Such frameworks provide an object wrapper to a relational database, essentially making the programmer's interface to the database more Object-Oriented. These frameworks abstract what a relational database is and how it works, but leave it to developers or commercial products like Persistence™ or Secant® to implement object persistence.

To use a non-object database to store objects requires translation. The software that performs the translation will need tentacles into the application that make it difficult to implement as a service. Translation software must be able to access the attributes and logical structure of objects, both of which should be private/hidden details. There are few clean ways of implementing a translation layer without introducing intrusive dependencies that remove some of the practical incentives behind Object-Orientation.

The Role of Components in Software Design Flexibility

An attractive benefit of Object-Oriented development is design flexibility and scalability. This refers to the raw ability to change a design without significant re-engineering. Unfortunately, even with Object-Oriented development these qualities are difficult to achieve. The best efforts to achieve flexibility and scalability can be derailed by a multitude of subtle forces.

The extent to which software is flexible or scaleable is largely determined by its internal structure. The internals of software nearly always consist of components. Components help to make a design more flexible by keeping the inner workings separated. A good component is self-contained and can be changed without impacting other components. A good component is also independent and can be used without much external support, a practical prerequisite for reuse. Components that are reasonably self-contained and independent are said to be encapsulated.

When internal parts of a component are not self-contained the software using the component becomes more complicated, less flexible, and more difficult to change. This may not be serious in isolated cases, but if encapsulation is consistently compromised [by an approach to object persistence] the damage can debilitate a software design and reduce its capacity to support future change.

Our experience has been that object-to-relational bridges have side effects that violate encapsulation and limit the independence of components. The side effects are so subtle that software engineers and database experts alike are normally unaware of the exact nature of the consequences until it is too late

Objects Have Dependence But Tables Do Not

Software dependencies can be either a driving force in design or a consequence of things gone awry. Software has two kinds of dependencies, logical and physical. It is not important here to distinguish between the two; it is only necessary to understand the general concepts. For readers familiar with procedural software development, dependencies are not new,

but be aware that dependencies are a much more important issue in an Object-Oriented design.

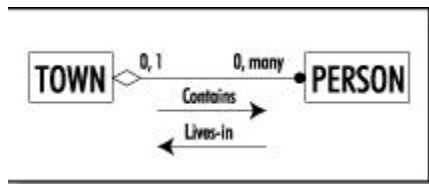
Dependencies arise from using components. Even small-scale applications rely on components for internal modularization and reuse. For example, a program that uses a library function to output text to a console, will have a dependency on the component hosting that function. This kind of dependency is common in C programs where low-level functions for input/output, math, and communications are packaged in libraries (components).



A program depends on its components and libraries

Object-Oriented designs use classes to achieve modularization (classes are the components). A class is a cross between a software module and an entity (from entity-relationship modeling). Surprisingly, few in our industry understand that good static object modeling involves balancing the rules of entity-relationship modeling with software module interaction. Object-Oriented development combines module centric software design with data centric design (the degree to which they are combined differs from one methodology to the next).

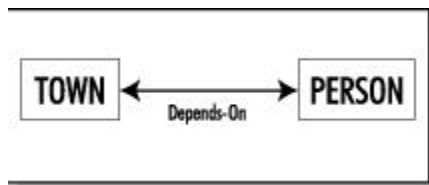
To illustrate dependencies between classes, consider a TOWN containing zero or more PERSON(s). If this is true for all towns, then we might build a model like the one shown below.



TOWN contains 0 or more PERSON(s)

From a database or entity-relationship point of view this is a simple relationship to implement. It is even simple to keep the reciprocal semantics of the real world; i.e. if a town is holding persons, a person may also be in a town. With a reference to a person, one can find the corresponding town; with a reference to a town, one can find all of the people in it. This seems like a simple model that should not present any implementation challenge. Unfortunately, this kind of situation causes major problems in object-oriented software. Dependencies!

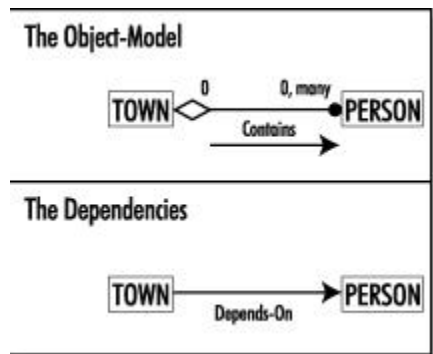
In database engineering, TOWN and PERSON are merely logical entities. In Object-Oriented design, TOWN and PERSON are still logical entities, but they are also software components. To preserve a bidirectional relationship, such as between TOWN and PERSON, from the data model into Object-Oriented design would cause a cyclic software dependency between the two components (classes).



TOWN and PERSON depend on each other, a Cyclic-dependency

A cyclic dependency means that components make use of each other. In this case neither TOWN nor PERSON can be used independently. Dependency considerations are a major difference between object modeling and entity-relationship modeling. Having cyclic dependencies between software components can cause serious problems. The most obvious is the limitation of each component's potential for reuse. In this example, because PERSON depends on TOWN, it cannot be used independently. It becomes quite impossible to reuse PERSON as an independent software component if there

are bidirectional (cyclic) dependencies to TOWN. Therefore, as part of Object-Oriented design, software engineers work diligently to eliminate one direction of each bidirectional relationship. After considering dependencies, the situation might be modeled as shown below:



TOWN depends on PERSON, but PERSON is completely independent of TOWN

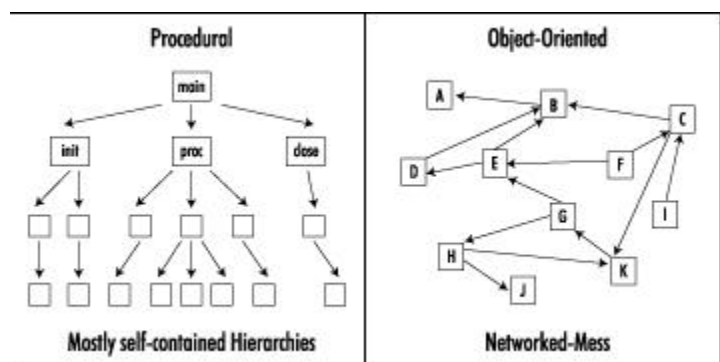
The dependency between TOWN and PERSON now only goes one direction. The consequence is that a PERSON does not know it lives in a town, and has no way to find out which TOWN it might be in. The benefit is that the class PERSON is now independent of TOWN and can be freely reused. In this case, a sacrifice to entity-relationship semantics is a necessary activity for achieving a good Object-Oriented design. This is an example of balancing a class' role as both an entity and a software module.

While it is not always possible, or even desirable, to eliminate bidirectional dependencies, it is a general goal to keep them to a minimum. Software engineers deliberately craft each relationship to avoid unnecessary dependencies, especially reciprocal ones. Database engineers, however, typically expect every relationship to be bidirectional by default. It is unnatural for an engineer accustomed to traditional data modeling to view relationships as having direction or for a relationship to create dependencies. Object-to-relational impedance mismatch often begins when classical assumptions from relational database design are not throttled with concerns for software dependency problems.

Dependencies are worse for O-O

Dependency problems can be more significant for Object-Oriented software than for procedural designs. One difference with O-O is that classes are much more fine-grained than procedural function libraries. Classes typically contain only 5 to 15 functions, compared with 25 to 50 in a procedural library.[11, 13] Furthermore in Object-Oriented design, classes are used throughout application code, not just for lower-level utilities. For these reasons, Object-Oriented designs have many more component modules than procedural applications with the same functionality.

Procedural designs typically have fewer dependency problems because of the natural hierarchies that form. Object-Oriented designs can evolve into vast networks, because there is no natural hierarchy imposed. The combination of dense modularization and a network style organization make O-O particularly vulnerable to dependencies.



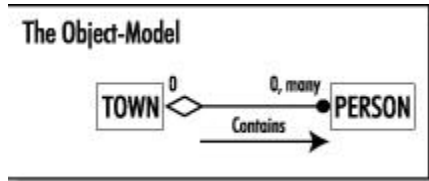
Another less obvious type of software dependency comes from data-fields. Even simple data-fields can cause implicit dependencies. For example, a class PERSON might have a data-field to store town-location as a way to remember how to get home. This is an obvious violation of data normalization, since town-location is clearly an attribute of TOWN. A database engineer could easily predict the maintenance and consistency consequences of this design choice, but it may not be obvious that this would also cause an implicit software dependency. Embedding an attribute into PERSON that

naturally belongs to another class reduces the independence of PERSON. Dependencies arise anytime a class has foreign attributes because programming logic inevitably becomes dependent on those attributes. The problems of implicit dependencies are relevant for anyone considering incorporating foreign keys into objects.

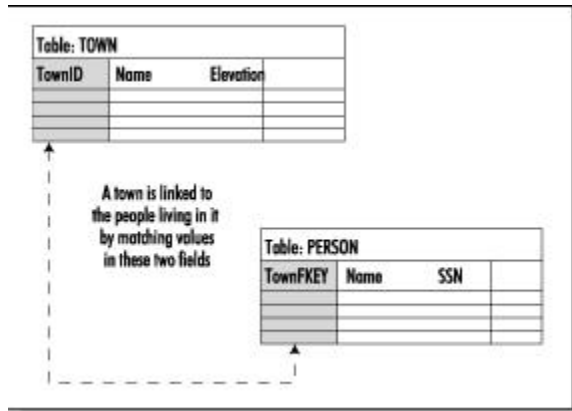
Foreign Keys and Account References

Note: an alternative to foreign keys is to use relational join tables instead. Join tables are much less disruptive to software structure, but are disliked by database engineers because of poor runtime performance. Join tables are used rarely; therefore our discussions largely assume that foreign keys are embedded in tables.

Foreign keys are used in a relational database to create relationships. Borrowing the example from earlier that had a TOWN containing zero or more PERSON(s), we see the object model below:



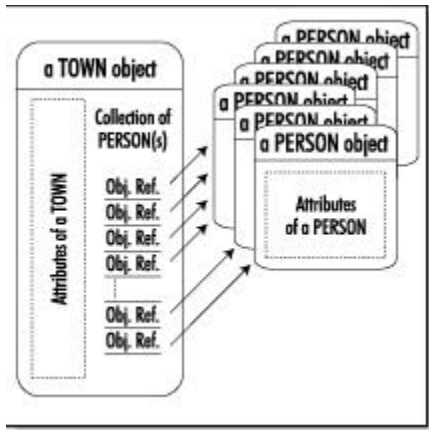
A possible relational database mapping for this would be the following two tables:



Possible database schema for TOWN to PERSON relationship

This is only one of several ways to map this relationship, but it is a common approach. Each PERSON in the database has a town-foreign-key (TownFKEY), which holds the identity of a TOWN, and is used to relate a specific PERSON and TOWN to each other. Knowing the identity of a TOWN makes it simple to find all of the PERSON(s) living in it. A relational query can be used to scan the PERSON table and search for PERSON(s) with a TownFKEY matching the TOWN in question. This kind of search is consistent with the intended use of a relational database.

An Object-Oriented application would implement the same relationship much differently. Object-Oriented applications do not use keys and foreign keys to maintain relationships. Objects use **collections**, which hold **references** (normally memory addresses) to other objects. In this example, TOWN would have a collection of object references to PERSON(s).



In Object-Oriented software, a TOWN has a collection of references to PERSON(s)

Looking at the two classes as software components (rather than logical entities), the collection causes TOWN to have a dependency on PERSON (TOWN specifically contains and makes use of PERSON(s)). However, a PERSON has no reference to its corresponding TOWN, thus keeping the class PERSON independent. PERSON can therefore be reused as a software component in other situations not involving towns.

Knowledge of relationships is reversed: For one-to-many relationships (like the example), database convention is to incorporate foreign keys in a table representing the side of the many. An object model of the same relationship puts object references on the **side of the one**. The reversal is important because software must interchange data between object references and foreign key columns. If object references are in each TOWN object, but the foreign keys are in a PERSON table, this forces the software that persists PERSON to know about TOWN, thus creating a cyclic-dependency. It is common among object-to-relational bridge solutions to embed foreign keys into classes. While this may simplify programming, it only serves to make the cyclic dependencies more direct.

Schema Mapping and Semantic Translation Often Mistaken as the Important Issues

A preoccupation of many who lecture about object-to-relational is the compatibility between object models and relational database structure.[3,4,14] Arguments presented at conferences and in the literature often adjudge the semantic differences. Our position is not to refute those solutions, as it is not our view that static model compatibility is a significant issue. It is widely recognized that the logical structures used in object modeling can be mapped to relational database tables.[3,14] We refer to this process here as **Schema-mapping**. Cookie-cutter schema-mappings are widely available even for relationships like class inheritance (a principal O-O relationship) where there is no explicit relational support.[3] However, even with the availability of coarse mappings between static O-O structures and relational tables, some of the mappings require significant non-trivial translation- such as the reversal of relationship knowledge.

Moreover, the substance of a good object-to-relational bridge encompasses more than just static semantic translation. Behavioral rules must also be followed. A relational database query typically uses data from single fields taken from one or more tables. In contrast, an Object-Oriented application will access whole objects via a technique called traversal. "Traversing to an object" means to find objects by following object references or by exploiting static relationships (associations or aggregation).

Because objects are accessed through references, an object is always used as a whole, never as its single fields. A good object-to-relational bridge manipulates the relational database in ways that mimic object-traversal. There are several approaches to this, some of which are SQL based. The SQL approaches, and many others, are well documented in the literature [3,14], but fail to address the sophisticated programming needed to convert query sets into objects or to store/update objects by generating dynamic SQL.

The notion that most of the challenges have been met is often encouraged in the literature. This stems from the fact that we understand structural and behavioral issues. Unfortunately, understanding an issue is not the same as having a solution. The programming for the translation logic that converts data-sets into objects, and visa-versa, has a subtle but voracious appetite for information that spans class boundaries. The information is often private to objects (or should be), and must be combined with information from other objects. Combining internal properties destroys the independence of classes and makes change and reuse more difficult.

Obtaining Foreign Keys from Objects

In the earlier example we considered it acceptable for TOWN to be dependent on PERSON. As discussed, giving

PERSON a reference to TOWN would reciprocate the relationship, making an unacceptable cyclic dependency that would diminish the independence of PERSON. Therefore, for dependency reasons, TOWN would contain references to PERSON(s), but a PERSON would not contain any reference to its corresponding TOWN, and hence no foreign key data. **Dependencies make it generally unacceptable to include foreign keys in objects.**

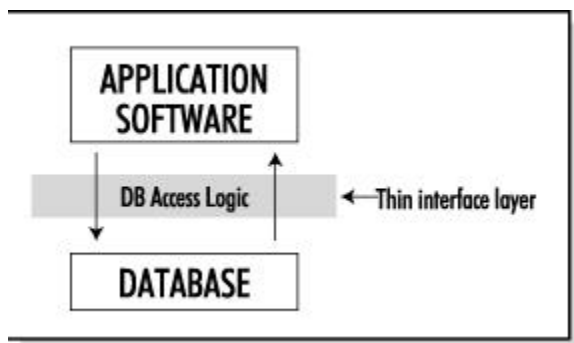
When persisting objects in a relational database, a layer of programming prepares records for the database by copying object attributes; we call this the **DB-Logic**. Given that a foreign key cannot be embedded in an object without introducing new dependencies, and that an object's corresponding database table requires a foreign key value, the DB-Logic must somehow obtain the correct foreign key without creating a software dependency. This can be accomplished by having an external entity pass a foreign key to the DB-Logic, or by endowing the DB-Logic with connections that allow it to obtain the foreign key on its own. It can be very difficult to do either of these without creating some new dependency or making the resulting structure very rigid against change.

It is important to understand that obtaining a foreign key is a major complication in object-to-relational persistence. The concept of a foreign key is contradictory to the way Object-Oriented dependency structures naturally develop. It is very difficult to develop and package DB-Logic such that it has access to the (private) attributes of an object as well as access to foreign key values while following rules of good software encapsulation and hence good O-O.

The Boundary Between an Application and the Database Interface Logic

It should be clear from the earlier discussion that a class, in O-O, plays two major roles. Conceptually, classes are logical data entities, but concretely they are software components. As a result, every class is subordinate to the rules and constraints of both data modeling and software component modularization.

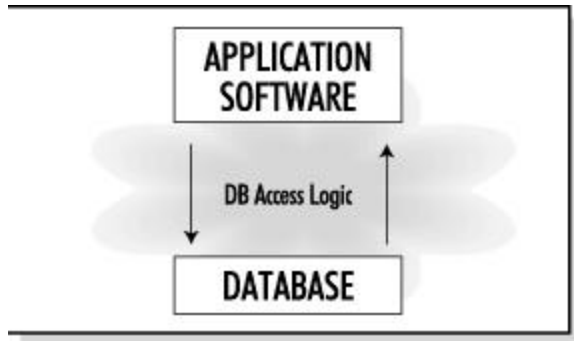
This is all particularly important to the layer of software between the application objects and the database. A typical model is shown below:



Application software & database separated by a thin layer of programming

In this picture, application software and database are shown to be independent, separated by a small layer of programming. This is a desirable model because it suggests that many applications can use a common database and encourages reusable database logic.

The problem with this model is that it is not very realistic. An Object-Oriented application will have its own schema (the object schema) and a database will have its own schema. The database access logic must be aware of both in order to translate between them. For instance, the DB-Logic cannot perform database queries, inserts, or deletes without being aware of the database schema. Likewise, the DB-Logic cannot traverse object relationships, nor access attribute values without being aware of the object schema. For the translation between object and relational to be correct, the DB-Logic must always be synchronized to changes in either schema, making it vulnerable to several sources of change. The actual model more closely resembles the diagram below:



DB-Logic for object persistence is difficult to confine to an isolated layer

This illustrates how persistence is not a service. The difficulties relate to the limited ways DB-Logic can gain awareness of an object schema. DB-Logic must be able to traverse an object schema to get information needed to populate foreign keys in the database. In most Object-Oriented programming languages the object schema is distributed, not centralized, with bits and pieces encapsulated in every class (e.g. each class encapsulates its own relationships and components). Software engineers do this intentionally to insulate a class from changes to its neighbor's implementation. The distribution makes it difficult to separate the DB-Logic as a separate service layer because it needs to have intimate access to several classes.

Structural information is difficult to obtain because it is hidden (private). Developers have devised several ways for DBLogic to access structural information. (1) Structural information could be made public rather than private. This invites other code writers to use those structures, which by rights should be private, making internal changes difficult. (2) Structural information could be replicated in parallel DB-Logic. This creates a large number of diffuse DB-Logic components that are difficult to modify as a unit, as would be needed for performance tuning. (3) Each class could play host to its own DB-Logic, thus allowing access to the private attributes of the class. This has the drawback of bringing unwanted dependencies to the host class. The unwanted dependencies can result from various database mapping needs, including foreign keys.

There are few good places to put DB-Logic. DB-Logic, because of dependency considerations, cannot be packaged with a class and still access foreign key information. DB-Logic cannot be packaged outside a class and still have access to the class' internal attributes and relationships; at least not without having the class expose those attributes in its interface. The challenge is therefore: **(1) without violating the independence of a class, how to efficiently map relationships encapsulated within a class to a database table the class is unaware of; and (2) how to keep the specifics of data mapping separated from the thing being mapped.**

There are technological solutions to some of these issues. For example, some programming languages are reflective [16], meaning they offer runtime access to object schema metadata. A more common approach, used by commercial object persistence products, is automated code generators and source code analyzers. Code generators use compile-time knowledge of both the object and database schema to modify source code and transparently add any necessary dependencies and DB-Logic without impacting design flexibility.

Important Change-Cases

As a rule, changes will be required during the life of a software design. Designers accommodate change by taking measures to ensure that anticipated changes happen easily and without widespread disruption. When a likely source of change is known in advance it is called a change-case. Some sources of change are generic and, as such, should be anticipated by nearly every design. We will present four change-cases that are important for any Object-Oriented design to anticipate. Failure to accommodate these four changes is a likely indication that a software structure is inflexible and brittle. Later we will use these change cases to evaluate the various object-to-relational approaches.

- **CC#1: Add, Remove or Modify attributes:** New attributes and changes to attributes are a constant source of design change. In Object-Oriented design, most data are encapsulated as private attributes in classes to insulate other components from changes to those attributes. Whenever an attribute is de-normalized (appears in multiple classes), the impact of a change to that attribute is magnified. Any general practice of attribute de-normalization can make this change-case difficult to accommodate and will result in rigid and inflexible software.
- **CC#2: Reuse a class (add a new client class):** Classes are constantly reused in Object-Oriented design. This practice is not only common but encouraged. In general, nothing should discourage class reuse. Without class reuse a significant amount of software has to be redundantly developed

and maintained. Remember, an additional association to a class, even when the class is unaware of the association, constitutes reuse.

- **CC#3: Change the internal implementation of a class:** Designers must be able to change the internal workings of a class without impacting components outside the class. This includes making internal adjustments by adding, removing and changing subordinate classes. To support this change-case, the internal workings of a class must be kept hidden so nothing outside the class can grow dependent on internal mechanisms. Design integrity can be seriously compromised if internal changes pathologically result in external consequences.
- **CC#4: Add sub-classes via inheritance:** A key feature of Object-Oriented programming is an object's indifference to the type of a related object when a type hierarchy is involved. The mechanics of Object-Oriented development allow designers to add new sub-types without impacting any clients using the base type. This is a crucial part of Object-Oriented design: persistence mechanisms that tamper with a designer's ability to do this will seriously limit the flexibility of a design.

Schema Integrity and Demand Loading

It is important to understand that some relationships are mandatory ("unconditional" in Shlaer/Mellor).[15] Mandatory relationships require object(s) at one end of a relationship to always be accessible. For example, if a TOWN is composed of PERSON(s), and the composition relationship is mandatory, then all instances of TOWN must have at least one PERSON. The cardinality constraint on a relationship determines if it is mandatory or not. If the cardinality constraint is (0, many) the relationship is not mandatory because zero objects are allowed. If the cardinality constraint is (1, many) the relationship is mandatory because there must be at least one object. [Note that the original example using TOWN did not have a mandatory relationship.] Whether a relationship is mandatory often depends on business rules in the problem domain. These constraints should be followed and must never be unilaterally altered by a programmer. **It is very important that an object's mandatory relationships be traversable at the instant the object is instantiated, before its first external use.**

The simplest way to make a persistent relationship traversable is to load the objects at both ends. In practice this would mean that every time a composite object is loaded from a database, all of its parts would also be loaded. With this approach, if TOWN and PERSON were joined by a mandatory relationship, even cursory use of TOWN would require that its PERSON(s) be loaded. This would insure the integrity of the object schema at runtime. Software correctness can be encouraged by disallowing any situations in which a software error might occur due to a client's inability to know what internal parts of an object have, or have not been loaded.

To always load every part of a composite object, especially for cursory use, is incredibly wasteful of CPU and memory resources. Often when software developers are presented with this situation, they choose to err against protecting the runtime integrity of an object schema in favor of performance. A compromise of integrity, such as this, should be unacceptable but is often allowed until consequences surface downstream. When the consequences finally surface (and they do), the resulting system instability is often blamed on other causes (sometimes O-O in general). Very little attention in the literature or the general community has been given to the important issue of runtime object schema integrity, the cause of many hard to correct software errors.

A more practical answer is to use demand loading, which postpones the reconstitution of mandatory associations from the database until those associations are actually traversed. With demand loading, cursory use of an object does not need to carry the overhead of loading its subordinate parts. Intelligent implementations of demand loading will also utilize bulk data exchanges to make sensible and efficient use of database queries and network transfers.

Demand loading preserves the integrity of an object schema, maintains efficient performance, and efficient use of memory. Unfortunately, demand loading is not always an easy feature to incorporate in an object-to-relational solution and is nearly impossible to add after the fact. Be warned that most approaches to object-to-relational persistence resist attempts to incorporate demand loading.

A typical problem with demand loading is the difficulty accomplishing it without embedding database-aware object references into application classes. To support demand loading, object references must be smart enough to determine the load state of dependent objects and then be able to load them when necessary. Static associations between classes are usually implemented as class attributes. Therefore any "smart" object references would infect the application class with their dependencies. This can limit the independence of application classes for reuse. Another example why object persistence is more than a service!

A "smart" object reference, capable of demand loading, typically has database dependencies. This is necessary to make database connections, to monitor the load state of objects, and to load objects when necessary. Often these require database connection names and other initialization parameters that can limit the host class from being reused in a different situation, or with another database.

A good demand loading solution allows for easily tunable behavior. Situations will arise where demand loading must be disabled due to performance inconsistencies; this should be supported. Also, if demand loading has been disabled in one subsystem, it should still work in others. All-or-nothing solutions are typically worse than no solution.

Even carefully controlled partial loads are bad

Every mandatory association must be traversable when a persistent object is instantiated; this is needed to ensure integrity of the software. But if only one function on a composite object is being called, and that function does not require any of the object's subordinate parts, why waste the effort to load them? If load on demand is not available, why not accept a performance gain by partially loading the object in that case?

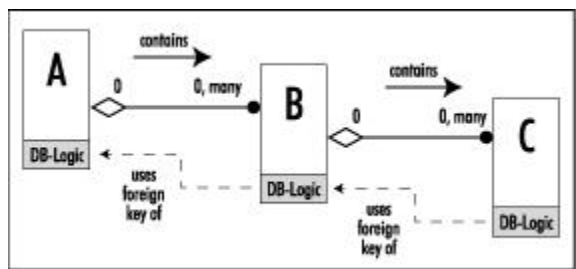
Reviewing a fundamental tenant of Object-Oriented software, implementation hiding; developers should be free to change a class's internal implementation without breaking any external clients. The developer of a client class should not even know about the aggregate parts of another class, much less make assumptions about whether a given function makes use of those parts. To prevent developers from making too many assumptions, classes hide their internals to keep clients from relying on them and inadvertently limiting future freedom to make internal changes.

Section 2 Object-to-Relational Approaches

There are many structure types that must be mapped by an object-to-relational bridge. The following discussion uses "composition" in all of the examples. This choice was made because it is the simplest of the static structures used in Object-Oriented design.

There are three approaches presented here, they have been nicknamed embedded, scaffolding, and manager. Each of these uses a different philosophy for incorporating the DB-Logic into an Object-Oriented design. These approaches have been abstracted slightly from what one might see on an actual project, but are representative of the available options.

The "Embedded" Approach: Because a class contains its own structural information, many see it as an obvious place to encapsulate Database-Logic. One popular approach is to add a thin layer of database logic to each persistent class. The database logic is responsible for reading and writing instances of the class. A logical parent or external client passes in foreign key information.



Simple Embedded Approach to Persistent Objects

Persisting objects to a database with this technique would have an instance of 'A' tell an instance of 'B' to write itself to the database. The instance of 'A' passes its own key to 'B', whereupon 'B' would tell each of its 'C' objects to write itself. With this approach, each class is responsible for preparing its own database records using foreign keys passed in from the logical parent. Restoring objects from the database is done via operations on the class that execute a database query and instantiate objects in memory.

For the initial development, the simplest way to implement this approach is to read and write each object collection individually. For example, to load an instance of 'A', static code in class 'A' would reconstitute the appropriate collection of 'B' objects by invoking DB-Logic on class 'B'. Each 'B', in turn, would obtain its collection of 'C' objects by invoking DB-Logic on 'C'. If there are multiple instances of 'B', the result is multiple requests to load 'C' objects, one request for every instance of 'B'. It should be obvious that this implementation is extremely slow because of its inefficient use of the database, but it is often the favored choice because it is simple and direct. Note that "load on demand" can easily be implemented using this approach.

A more robust implementation of this approach is to have DB-Logic in each class perform bulk reads and writes. Performing a bulk data exchange for reading is difficult because instances must be loaded by class. The idea is to load all C's at once, then all B's at once. The challenge is to load the right C's and right B's, getting the correct objects into each collection and limiting queries to only those objects that are related indirectly to the root object (a particular 'A' in this case). This means that the key for 'A' is needed to formulate a query to retrieve only relevant C's. Having DB-Logic in 'C'

use a key from 'A' is a dependency which limits the independence of 'C' for reuse outside the scope of 'A'. Eliminating that dependency under this approach while maintaining performance turns out to be tricky (read as: "we have never seen it"). As our experience has taught us, vigilance over good encapsulation along with thoughtful concern towards runtime performance are opposing forces in many object-to-relational solutions.

Necessities of the Embedded Approach:

1. Domain logic & database technology are tightly coupled
2. Classes have code that manipulates foreign keys
3. Foreign keys cause a software dependency from logical child to logical parent
4. Mixes database schema specific knowledge with application logic
5. Requires relational JOIN queries
6. The DB-Logic is highly prolific and diffuse

Consequences of the Embedded Approach:

- Class reuse is limited because each class is tightly coupled to the database
- Class reuse is limited because of cyclic parent-child-parent dependencies
- A class cannot be reused in a different hierarchical situation without creating a separate version
- DB-Logic is scattered such that there is no single place to change or fine-tune it
- The database schema cannot be modified without breaking application classes
- A large amount of complex DB-Logic to maintain
- DB-Logic is duplicated once for every persistent class

Impact to Change-Cases

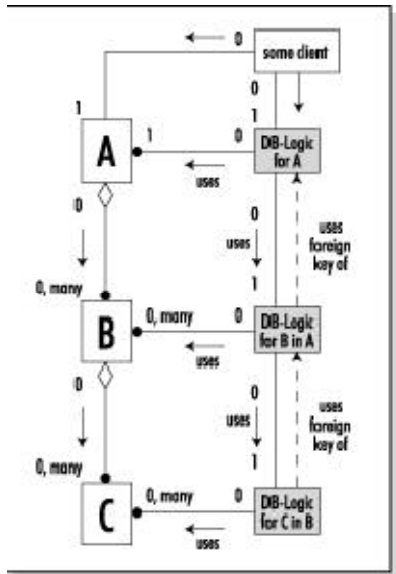
CC#2 Violated - Reuse a class (add a new client)

This approach virtually prohibits the reuse of any persistent class. Reusing a class would require another version of the class with new DB-Logic. This means that classes cannot be used as fully self-contained, fully unit tested drop-in components. Reuse, a major benefit of Object-Oriented, cannot be obtained with this approach.

Notice there is no architectural layering or separation of domains with this approach. The DB-Logic and the application logic are completely mixed together. This approach makes it difficult to make application-wide adjustments to the DBLogic (often necessary when tuning for performance).

Variations of this approach use inheritance to separate application logic from database logic, but cause class proliferation and add complexity in the long run. The general technique is to put application code in a base class and hang DB-Logic below in a derived class. Application code can then be reused sans the dependencies of the DB-Logic. This works well if the application code itself does not have any inheritance relationships. If inheritance is used at the level of the application code, then the approach quickly disintegrates. The approach only works if the application classes are at the bottom of an inheritance hierarchy. In this situation, using inheritance as a standard way to separate database services and application code is of limited or no value.

The "Scaffolding" Approach: The major problem with the embedded approach is the tight coupling of application logic to database logic. The scaffolding approach isolates application code, making it largely independent from the DB-Logic. The DB-Logic replicates the hierarchical structures of the application classes in a parallel hierarchy. A DB-Logic class encapsulates the capability to persist and restore its application class counterpart, even if the application class is composed of many support classes.



Scaffolding replicates the schema structure in a parallel hierarchy

An important fact of the approach is that there needs to be a DB-Logic component built for each use of an individual application class. Because relationships are usually encapsulated in classes as private details, the only way to get structural information is for developers to replicate the application's structural relationships in the DB-Logic. This means a different DB-Logic component must be built every time a class is reused. Also, relationships between DB-Logic components must exactly reflect relationships between application classes.

The advantage of the approach is that most application classes are isolated from the DB-Logic. Since application classes have no dependencies on the DB-Logic, designers may freely reuse them.

Necessities of the Scaffolding Approach:

1. Relationships & schema are not encapsulated
2. DB-Logic must be able to read/write private class attributes ('friendship' does not work here)
3. Requires relational JOIN queries
4. The DB-Logic is highly prolific and diffuse (doubles the number of classes)

Consequences of the Scaffolding Approach:

- Poor performance & large memory footprint from lack of demand loading
- Cannot support demand loading without putting database aware object references into application classes and adding unwanted dependencies to those classes
- Clients may become dependent on private attributes exposed to allow DB-Logic access
- A large amount of complex DB-Logic to maintain
- DB-Logic is implemented once for every persistent class
- No one place to change or fine-tune database access logic

Impact to Change-Cases:

CC#2 Violated- Reuse a class (add a new client)

Reuse is not prohibited, but can be discouraged by this approach. Each time a class is reused, another DB-Logic class is created that has dependencies to the class' internal composition. Once a class has been reused several times, making an internal change requires developers to find all places where the class is used and propagate the change to the impacted DB-Logic. If components are reused across a large system, distributed over a family of systems, or over multiple customers, this can be difficult.

CC#1 Violated - Add, Remove, or Modify attributes

The approach makes it difficult to modify internal attributes. For external DB-Logic to be able to persist or reconstitute the state of an object, there must be operations in the public interface that expose the private attributes. Class "friendship" is of no help because the application class is forbidden (by the approach) from being aware of the DB-Logic. If the class' private attributes are exposed, it is inevitable that they will be exploited, thus making it very difficult to make internal adjustments.

While this approach does separate application logic from DB-Logic, not all application classes can be independent of database logic. As can be seen from the diagram, classes A, B and C are all independent of database logic, but the class labeled "some client" is related to database logic. If the client is an application/domain class, the dependency problems have only been shifted to the client, but not eliminated.

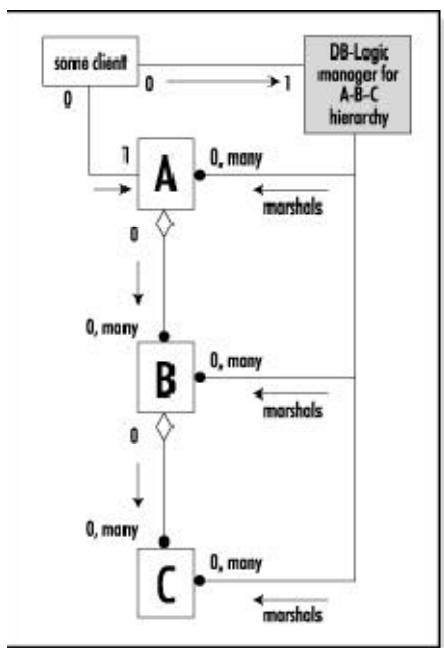
Classes, in the scaffolding approach, no longer fully encapsulate their own relationships. Every hierarchical relationship between application classes must be mirrored in the DB-Logic. This reduces the plasticity of the design. Every structural change affects both the application component and the corresponding DB-Logic. Fortunately, the impact is not terribly severe because the DB-Logic is never reused, making the impact relatively obvious to developers.

Demand loading is very awkward, if not impossible, with this approach. Demand loading requires object references capable of determining the load or cache state of an object. Object references that are smart enough to accomplish this typically have database-specific dependencies and require initialization parameters such as database connection names. Embedding database-aware object references in an application class effectively ties a class to the database in the same way as embedding database logic into a class. Keeping with the spirit of the approach (to separate application classes from database logic) demand loading is prohibited. This means that resulting designs will either have poor performance, or programmers will have to compromise the runtime integrity of the object schema.

A major business-case against this approach is the voluminous amount of code required to implement it. The parallel hierarchy uses a lot of software, which takes time to develop and becomes part of the software's maintenance problem. Some implementations offer a modicum of reuse in the database logic, but because so many things are dependent on both the specifics of a class and the specifics of database table definitions there is a lot of code replication. This approach can be prohibitively expensive, sometimes doubling the effort needed to write software, or worse.

The Manager Approach: Consolidating database logic for a complete structure into a single manager can solve some of the problems with the scaffolding approach. A disadvantage of scaffolding is the fragility and complexity of having parallel hierarchies combined with a massively increased development effort. Having a single database manager may not eliminate the dependencies, but it does help reduce complexity and development effort.

On close inspection, each DB-Logic component in the scaffolding approach must be coded specifically for each use of an application class, so little or no harm is done by consolidating a class structure's DB-Logic into a single component.



A single database manager houses persistence logic for a whole hierarchy

This approach has all of the benefits and most of the same limitations as scaffolding, but it is simpler to comprehend. The application classes are still isolated from the DB-Logic, and the DB-Logic must still be endowed with knowledge of the structural relationships. One significant advantage is that there are fewer DB-Logic components, which makes it easier to reuse database code and make adjustments for tuning or optimization. The DB-Logic itself is much more complicated in this approach because it encapsulates too much, but the reward is a simplification of the overall design.

Because a database manager hoards knowledge about several classes and potentially many tables, it is vulnerable and brittle. Any internal change to a component in the application hierarchy can impact the manager. This can be serious because the internal complexity of the manager is high and the impact of a component change may not be immediately obvious to a software developer making a change.

Necessities of the Manager Approach:

1. Relationships & schema not encapsulated in classes (but they are in a managers)
2. DB-Logic must be able to read/write class attributes (cannot be C++ style 'friends')
3. Requires relational JOIN queries
4. The DB-Logic is prolific (lots of complexity in each manager)

Consequences of the Manager Approach:

- Demand loading is very difficult, causing slow performance in situations that require it
- Clients may become dependent on private attributes exposed to allow DB-Logic access
- Changes to application classes can have a nebulous impact across DB managers
- A large quantity of complex DB-Logic to maintain
- DB-Logic is implemented once for every persistent class (albeit encapsulated in a manager)

Impact to Change-Cases:

CC#2 Violated- Reuse a class (add a new client)

Again, reuse is not prohibited, but can be discouraged by this approach. The more a class is reused, the more managers that will have dependencies on the class' internal composition. Once a class has been reused several times, making an internal change requires finding all the places the class was used and propagating the change to the impacted managers.

CC#1 Violated - Add, Remove, or Modify attributes:

The approach makes it difficult to modify internal attributes. For the external DB-Logic to be able to persist or reconstitute the state of an object there must be operations in the public interface that expose the class' private attributes. Class "friendship" is of no help because an application class is forbidden (by the approach) from being aware of any DB-Logic. If the private attributes of a class are exposed, it is inevitable that they will be exploited, thus making it very difficult to make internal adjustments.

Experiences From a Real Project:

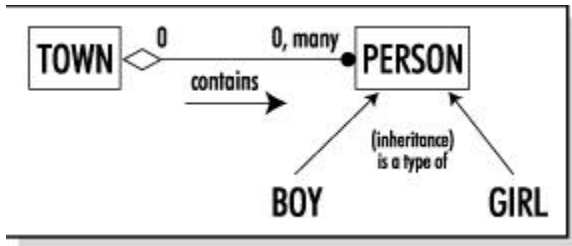
Most real-life implementations of object-to-relational bridges are hybrids of the three approaches presented above. A recent project used a combination of manager, scaffolding and embedded. That project attempted to generate much of the scaffolding using an in-house code generator and delegated construction of the managers and embedded logic to individual product development teams. For this project the result was an enormous bloat in additional software, where code dedicated to persistence represented roughly 70% of the total amount of software on the project. While some of the code was machine-generated, limitations of the code generator inhibited regeneration, making maintenance of the scaffolding structures manual. The result enormously increased the total software under configuration control. Compile and runtime performance was also badly affected.

Changes to persistent attributes or static object schema resulted in developers having to manually propagate changes through database scaffolding and manager components. This situation created a very inflexible architecture, where a single change-case typically impacted several complex components. Furthermore, tuning the DB-Logic in the scaffolding structure proved to be difficult because the code generator replicated and distributed the database logic that was later manually maintained. Updated releases of the code generator caused churn. New translation policies or bug fixes to code generator had to be applied manually to software already in existence.

Problems Supporting Class-Inheritance

Inheritance is a very important relationship in Object-Oriented development. Inheritance is similar to composition in that both have hierarchical structure. The difference between inheritance and composition is usage. Composition is used to hide internal details (some call it assembly/sub-assembly), while inheritance is used to hide class type.

Most of the object-to-relational literature focuses on inheritance as sub-typing, such as with generalization-specialization. While this is accurate, there is more to inheritance than simply the expression of sub-types. Sub-types are a familiar database-engineering concept with many reference implementations.[3,9] However, inheritance in Object-Orientation is different because of the type substitution that is possible among derived types (better known as "polymorphism" in O-O circles). This twist makes inheritance very difficult to implement in an object-to-relational solution.



TOWN contains PERSON(s), unaware if each instance is a PERSON, a BOY, or GIRL

This diagram shows that a TOWN may be associated with many PERSON(s). Two other classes, BOY and GIRL, are both types of PERSON (e.g. they are derived from PERSON). Both BOY and GIRL share the attributes of PERSON (read as: a BOY is a PERSON, and a GIRL is a PERSON). The class BOY is composed of PERSON plus anything unique about BOY, and class GIRL is composed of PERSON plus anything unique about GIRL. The fact that derived classes are essentially aggregates is interesting but the challenges associated with persistent inheritance come from elsewhere. The relationship between TOWN and PERSON is more significant because TOWN is completely transparent to the actual type of PERSON. Any object that is a PERSON, or is a type derived from PERSON, is allowed in this relationship. Because BOY and GIRL are types of PERSON, instances of either can be legally substituted in the relationship with TOWN. Because of this, TOWN objects believe they contain PERSON(s), but may actually contain BOY(s) and/or GIRL(s). When a TOWN uses services of PERSON, it will not discriminate the derived types. Every PERSON is treated equally by TOWN, without regard to its actual type. This is called type substitution, and it is a way to extend software without changing components that have already been developed and tested. By using derived type substitution where appropriate, a base class (PERSON in this case) can represent class types that may not even exist at the time of the original development. Inheritance relationships to new class types are a way to effect changes without impacting the rest of the software.

It is difficult to reconstitute this relationship from a database and preserve TOWN's ambivalence to types derived from PERSON. A query capable of finding every potentially involved object would have to span the various tables mapped to classes PERSON, BOY, GIRL and any other derived type that gets added later. Such a query would be needed to reconstitute TOWN's collection of PERSON objects. If the DB-Logic associated with TOWN has to be modified every time a new PERSON type is added, then the practical usefulness of inheritance has been lost.

If the inheritance hierarchy is completely normalized in the database, then reconstituting the relationship from TOWN to PERSON requires queries over at least two tables, BOY and GIRL, and possibly over a PERSON table as well. It is left to software to retrieve the correct data from the database and to create objects of the appropriate type. That software must know which tables to query (BOY, GIRL, PERSON) and must create object instances of the correct type. However, new sub-types of PERSON should impact no application classes, including TOWN.

Therefore, if DB-Logic is packaged with TOWN (the client), as would be done with the embedded approach, then to query the right tables and instantiate the correct objects, the class would require modification anytime a sub-class to PERSON is added. This would be a serious setback to software dependency management and design flexibility.

Under the scaffolding approach, application classes would be insulated from any direct consequence of adding subtypes, but the DB-Logic would not be. This is because the DB-Logic in the scaffolding must be able to instantiate the new class and possibly query over an additional table. For a maintenance developer, adding a sub-class to PERSON would require finding the persistent clients to PERSON (and the clients of its base classes) and locate the DB-Logic for those clients. This activity removes much of the advantage to inheritance in the first place. The same problems would be experienced if manager were used.

To briefly address schema mapping, which seems to obsess many pundits of object-to-relational techniques for mapping inheritance [3,4,14], there are three basic techniques for structurally mapping inheritance hierarchies to relational database tables: horizontal, vertical and typed. It is important to realize that regardless of the mapping approach, the result has the same impact on software. Each mapping technique requires modifications to the software every time a new derived class type is introduced. In essence, the software must be aware of each new derived class type, a consequence that diametrically opposes the principle of type substitution.

At present we have not seen any reasonable way, sans automated code generators, to handle inheritance in an object-to-relational bridge. The dependency problems that arise, no matter where the DB-Logic is packaged, are chronic for inheritance. A code generator is in a unique position to add necessary dependencies transparently and provide a degree of insulation to clients from changes in an inheritance hierarchy. This is one of the major areas where any manual object-to-relational solution compromises the practical usefulness of Object-Orientation.

Pre-Processors/Code Generators

Our experience has been that most projects choose to develop their own object-to-relational solution rather than buy one

off-the-shelf. Normally, this results in developers writing all of the persistence/database logic themselves, with only small amounts of reuse code and poor consistency across a software product. Occasionally projects choose to build a code generator to alleviate some of the monotonous work, but building a code generator in concert with developing an application can be an ill-fated idea.

The general concept of using code generators for persistence, whether purchased or homemade, has both advantages and disadvantages. The advantage is that automated code generation is the only solution that actually works to preserve the benefits of Object-Oriented software design and still provide reasonable database access performance. Because code generators automatically add the database access logic, developers are shielded from software dependencies that might otherwise be created had the same code been hand generated. The disadvantage is that code generators seldom work as required, and can lead to a much worse situation than would have occurred without one.

Code generators for object persistence must be able to parse both the object schema and the database schema to generate appropriate mapping logic. A good code generator generally has the capability to reverse-engineer code and database schema, or can interface with CASE tools. The most reliable method is for a code generator to employ code and database analyzers that reverse-engineer structural relationships from actual, as-built, products. Once the structural schema is known for both the database and the Object-Oriented application, a code generator can add appropriate database logic to accomplish an efficient mapping. The generated software typically includes logic to swap collections for foreign keys, manage object identity, accomplish demand loading, handle integrity locks, and provide object-caching services. When components are changed or reorganized, either in software or in the database, the code is simply regenerated.

For complex persistent relationships like inheritance, code generators are the only solution that works. When an association is to the base of an inheritance hierarchy, the database logic must be aware of all of the sub-types. If that same code was developed manually, the client of the relationship would become dependent on all sub-types due to the persistence logic, ruining the incentive to use inheritance as a tool for insulating against future change. Code generators can eliminate this problem. We have been unable to locate any other reasonable way to map inheritance relationships to a relational database without significant sacrifice.

Unfortunately, code generators seldom work very well. Some commercial code generators are tolerable, but in-house code generators are often disastrous. The development pressures of a project seldom allow enough time and resources to mature tools that are essentially used only for development. Code generators are very complicated and take much more time to design, test, and mature than is usually allowed in a development schedule.

Code generators, by their nature, tend to not capitalize on typical methods of Object-Oriented reuse across generated components. Most reuse in generated code is via code replication. If generated code ever has to be maintained by humans, the cost can be great. The cost of manually applying a change to software that has been infected with generated code can be very high. With a good code generator this is never a problem because developers simply regenerate the code when a change occurs. However, poor code generators often put developers in a situation where there is no alternative to hand modification of generated code. This eliminates the possibility of using the code generator again in the future, because the manual changes would be stepped over and lost by subsequent regeneration. These situations contribute to adding voluminous and often cryptic generated code to the software under configuration control, and thus entered into the normal maintenance pool.

The only technique we have found to implement an object-to-relational bridge without significant compromises is automated code generation. However, projects should not develop their own code generators unless they plan on marketing them as an actual product. We strongly suggest that any project requiring an object-to-relational bridge buy the technology from a commercial vendor. We recognize that the commercial products have many significant warts as well, but in the long run the commercial products will give an O-O project a better chance at preserving the quality of software architecture and design flexibility. The most important thing in any code generator is the ability to regenerate code at later times. Any code generator that encourages developers to alter the generated product, or even exposes the generated product to developers, should be avoided.

The Ubiquitous Relational Join

We have so far focused on the differences between the various approaches to object-to-relational mapping. One aspect that is common to all object-to-relational mapping schemes is the relational join operation. This is particularly true with more complex relationships such as composition and inheritance. By their very nature, Object-Oriented designs typically have a large number of these types of structures. Mapping these structures to a relational database is simple, but normally requires multiple relational join operations to reconstitute the objects. Relational joins are typically very slow and, unfortunately, they are ubiquitous in object-to-relational mapping approaches (including automated code generation). Although the purpose of a relational database is to relate data, there is no explicit support for logical relationships in a relational database. Two tables cannot be linked via a formal relationship. The relationships between tables are implemented solely by the use of keys and foreign key fields. To retrieve complex data, stored across tables, a query must join records based on the keys and foreign key values, a highly computational activity. The JOIN is therefore an intrinsic and ubiquitous part of accessing hierarchical data. Relational JOIN operations are a built-in and low-level feature

of any relational database, but are still very slow in comparison to queries over single tables.

Object-to-relational bridges must rely on a relational JOIN to reconstitute hierarchical structures like composition and inheritance. Especially in the case of composition, these structures are a very common and essential part of most object models. Our experience has been that composition and inheritance are by far the most used relationships in Object-Oriented design. Therefore most object persistence involves significant hierarchical structure, and will require frequent use of relational JOINS. It does not matter if the relational JOIN is programmed as static SQL, dynamic SQL, or through a database's proprietary API, the result is still expensive computation. We dwell on this because it makes relational databases intolerably slow as devices for implementing object persistence. Furthermore, the slowness is an intrinsic part of the object-to-relational problem and cannot be easily optimized away.

The one way to eliminate the overhead of a relational JOIN in object-to-relational mapping is to avoid foreign keys altogether. By taking hierarchical structures, like composite objects, and squeezing them into a single table, JOINS can be avoided. This approach, however, has some serious limitations. In fact, this type of approach directly opposes some of the innate strengths of relational database technology. In particular, most of the data integrity guards including entity integrity and referential integrity are no longer possible with this type of approach. There is no way of knowing if the data in the database has been updated correctly, or if key components have been lost along the way. Specifically, this approach makes it necessary to use null fields, making it impossible for the database to insure integrity and validate field values. This makes maintenance and change difficult, bypasses integrity, and creates very large super-tables that are difficult to read and expensive to search. This approach is usually not used because database developers dislike the side effects. (It should be noted, that if this sounds familiar, the lack of insured integrity is what happens to an application when object schema integrity cannot be guaranteed.)

Another major consideration is deciding how to implement the SQL, including the JOIN queries. In particular, this may be done using either static SQL or dynamic SQL. One thing is clear; when building a generic framework, only dynamic SQL will work.[3] In any manual (i.e. not code-generated) object-to-relational solution it should be carefully considered where the programming for the JOIN (usually SQL) goes. Our experience with static SQL has been bad, where in one case an executable with 50 classes required eight separate routines for the embedded SQL, totaling 400 configuration controlled C++ functions, each containing some very brittle SQL. In addition, static SQL will not work for a general object-to-relational framework, it requires a lot of hand tuning and use-specific coding.

In summary, the relational JOIN generally cannot be avoided in an object-to-relational solution. Relational database performance is a serious issue because of the ubiquitous nature of hierarchical structures in most Object-Oriented designs.

Object-Relational Technology, A Solution?

Object-Relational database technology, an extension to relational database technology, enhances the core functionality of a relational database by incorporating "objects" and other new capabilities. Some of the extensions Object-Relational offers are: Extended Data Types (EDTs), complex objects, and support for inheritance [9,17]. Basically, EDTs allow complex types to be defined from built-in database types, and permit subroutines to be added into a type. Complex objects (not to be confused with complex types) are collections which themselves may be defined as either an EDT or a database built-in type.

Since the problems related to schema mapping are often emphasized in the popular literature, it is understandable that many individuals conclude that an Object-Relational database should solve the majority of object-to-relational problems. It can be reasoned that by using EDTs, a class can be mapped directly to a corresponding database EDT type. By effectively combining EDTs with complex objects, it is possible to drastically reduce the impedance mismatch between the Object-Oriented software and the database. However, it is important to note that poor language bindings are still a major issue, and many of the relational database vendors are having major difficulties supporting inheritance relationships, and worse yet, polymorphism. While Oracle 8 has incorporated most Object-Relational features, it still does not support inheritance relationships, nor does it have good language bindings. The illustrated database has some abstruse problems with respect to polymorphism.

The other new capability common to most Object-Relational databases is support for Object Identifiers (OIDs). OIDs are unique system-generated identifiers for objects. This is a nice feature for the software as it relieves application programmers from generating their own unique identifiers. Unfortunately, just as in the case of EDTs, and complex objects, OIDs fail to address the core issues of instantiating objects from data-sets and formulating SQL from data extracted across a network of objects. Even with OIDs, foreign key problems are not eliminated. It is still the case that foreign keys exist; they just hold OID values instead of user defined keys. It is also true that OID-based foreign keys are still on the opposite side of the relationship than object references in software. It is this reversal of relationship semantics that leads to the same conflicts between system performance and basic software engineering principles. This relates back to the fact that impedance mismatch between programming language bindings and database is in no way solved by Object-Relational databases.

While "object" extensions to relational database technology do reduce the overall impedance mismatch between Object-

Oriented software and the database, they contribute little in solving the significant problems in building an object-to relational bridge. The lack of good language binding is probably the most significant reason why Object-Relational databases fail to deliver true object persistence. They reduce the semantic mismatch between Objects and Relational, but do not go far enough to remove the dependencies that limit software flexibility.

What About Object-Oriented Databases?

Throughout this paper we have avoided any discussion of Object-Oriented databases. We wanted the paper to focus on the intrinsic problems with object-to-relational. However, it is worth noting that Object-Oriented databases address most of the problems with object persistence. There are several varieties of Object-Oriented Databases (OODB) available from commercial vendors. Each variety has distinguishing qualities that make choosing one very situational. In general, most commercial OODB products offer good language bindings and typically make use of code-generator technology. Object-Oriented databases also eliminate the problems associated with foreign keys. Most OODBs are based on object references, not foreign key relationships. Because of a more compatible information model, and better language bindings, the impedance mismatch between an Object-Oriented application and the database is low with an OODB.

Given the state-of-the-art, the best way to implement object persistence, without sacrificing software-engineering principles, is by using a good commercial OODB. In most cases they allow a designer to take full advantage of the benefits of inheritance in addition to relationship integrity checks, object caching, demand loading and good performance. Unfortunately, it is sometimes difficult to get clean insulation of application code from database specific functions, a major detractor for many commercial OODB products. NOTE: Not all OODBs are equal in insulating the program code from the database. Some are much more intrusive than we would like!

Some OODB systems limit the accessibility of data from non-Object-Oriented external users. Other OODB systems are friendlier in this regard and offer SQL interfaces, through which O-O unaware clients can access data as if it were in a relational database.

One major benefit to an OODB is performance. OODBs typically perform much better than a relational database wrapped in an object-to-relational solution. Most Object-Oriented structures are hierarchical, a complication for relational databases because of the join operations that become necessary. OODB systems are intended to traverse networks of objects and are efficient at the kind of operations that are typical in an Object-Oriented application.

Conclusion

There is no law of physics prohibiting object-to-relational. It can work and is implemented with some success on a regular basis. The problem with object-to-relational is the lack of a really clean way to implement it. Commercial tools, in spite of their faults, provide the cleanest approach because they hide intrusive software dependencies.

There are many practical variations to the three approaches outlined in this paper (embedded, scaffolding, and manager). These approaches can be combined in countless ways, each with qualities and shortcomings not mentioned here. However, even after considering the innumerable methods of combining and interpreting these approaches, the basic problems remain. There are forces pulling for high performance, strong encapsulation, logical cohesion, modularity, and relational normalization. In managing one or two of these forces, others fall aside. The contradictions are inherent to the problem and cannot be avoided.

Object-to-relational solutions will always need to reverse relationship semantics between the software and the database. The truth is there are serious mismatches between objects and tables. These mismatches need to be resolved either by the software or by the database. Placing this responsibility on the database typically causes performance to suffer. Placing this responsibility in the software seriously degrades the quality of the design with respect to flexibility, reuse, and scalability.

Inadequate language bindings are mostly a consequence of languages like C++. Until better languages become common, or commercial object-to-relational solutions are more accessible, this is not likely to change. In all but the simplest software systems, performance is a major issue. To address performance, a software engineer often must violate Object-Oriented and software-engineering principles such as encapsulation, cohesion, separation of concerns, and minimal interfaces when faced with object-to-relational persistence. If the infrastructure to support object to-relational persistence is not stable, robust and fast, then it will be circumvented as developers find ways around their immediate problems.

The benefits of Object-Oriented software development, including software flexibility and reusability, do not just happen. They require adherence to basic principles of software engineering, principles that are encouraged by Object-Orientation. Although using a relational database in an Object-Oriented software application does not appear to directly violate any sacred principles, the indirect consequences obstruct the advantages of Object-Orientation. Any serious proposal to use a relational database to support object persistence should be backed by a robust code generator that has been specifically designed to reduce impedance mismatch. Remember that object persistence is not a service, not an API, and not an architectural layer. Object persistence permeates a design and intrudes into application code. If these issues are not handled before application development starts, be prepared to sacrifice much of the practicality behind Object-Orientation.