Monday, 26 June 2006
The Vietnam of Computer Science

(Two years ago, at Microsoft's TechEd in San Diego, I was involved in a conversation at an after-conference event with Harry Pierson and Clemens Vasters, and as is typical when the three of us get together, architectural topics were at the forefront of our discussions. An crowd gathered around us, and it turned into an impromptu birds-of-a-feather session. The subject of object/relational mapping technologies came up, and it was there and then that I first coined the phrase, "Object/relational mapping is the Vietnam of Computer Science". In the intervening time, I've received numerous requests to flesh out the discussion behind that statement, and given Microsoft's recent announcement regarding "entity support" in ADO.NET 3.0 and the acceptance of the Java Persistence API as a replacement for both EJB Entity Beans and JDO, it seemed time to do exactly that.)

No armed conflict in US history haunts the American military more than Vietnam. So many divergent elements coalesced to create the most decisive turning point in modern American history that it defies any layman's attempt to tease them apart. And yet, the story of Vietnam is fundamentally a simple one: The United States began a military project with simple yet unclear and conflicting goals, and quickly became enmeshed in a quagmire that not only brought down two governments (one legally, one through force of arms), but also deeply scarred American military doctrine for the next four decades (at least).

Although it may seem trite to say it, **Object/Relational Mapping is the Vietnam of Computer Science**. It represents a quagmire which starts well, gets more complicated as time passes, and before long entraps its users in a commitment that has no clear demarcation point, no clear win conditions, and no clear exit strategy.

History

PBS has a good synopsis of the war, but for those who are more interested in Computer Science than Political/Military History, the short version goes like this:

South Indochina, now known as Vietnam, Thailand, Laos and Cambodia, has a long history of struggle for autonomy. Before French colonial rule (which began in the mid-1800s), South Indochina wrestled for regional independence from China. During World War Two, the Japanese conquered the area, only to be later "liberated" by the Allies, leading France to resume their colonial rule (as did the British in their colonial territories elsewhere in Asia and India). Following WWII, however, the people of South Indochina, having thrown off one oppressor, extended their anti-occupation efforts to fight the French instead of the Japanese, and in 1954 the French capitulated, signing the Geneva Peace Accords to formally grant Vietnam its independence. Unfortunately, global pressures perverted the efforts somewhat, and instead of a lasting peace agreement a temporary solution was created, dividing the nation at the 17th parallel, creating two nations where formerly no such division existed. Elections were to be held in 1956 to reunify the country, but the US feared that too much power would be given to the Communist Party of Vietnam through these elections, and instead backed a counter-Communist state south of the 17th parallel and formed a series of multilateral agreements around it, such as SEATO. The new nation of South Vietnam was born, and its first (dubiously) elected leader was Ngo Dinh Diem, a staunchly anti-Communist who almost immediately declared his country under Communist attack. The Eisenhower Administration remained supportive of the Diem government, but Diem's loyalty with the people was almost nonexistent from the beginning.

By the time the US Democratic Party's John F Kennedy came to the White House, things were coming to a head in South Vietnam. Kennedy sent a team to Vietnam to research the conditions there and help formulate his strategy on the issue. In what's now known as the "December 1961 White Paper", an argument for an increase in military, technical and economic aid was presented, along with large-scale American "advisers" to help stabilize the Diem government and eliminate the National Liberation Front, dubbed the Viet Cong by the US. What's not as widely known, however, is that a number of Kennedy's advisers argued against that buildup, calling Vietnam a "dead-end alley".

Faced with two diametrically opposite paths, Kennedy, as was typical for his administration, chose a middle path: instead of either a massive commitment or a complete withdrawal, Kennedy instead chose to seek a limited settlement, sending aid but not large numbers of troops, a path that was almost doomed from the beginning. Through a series of strategic blunders, including the forced relocation of rural villagers (known as the Strategic Hamlet Program), Diem's support was so deeply eroded that Kennedy hesitatingly and haltingly supported a coup, during which Diem was killed. Three weeks later, Kennedy was also assassinated, throwing the domestic US political scene into turmoil as well. Ironically, the conflict began by Kennedy would in fact later be associated most closely with his replacement.

Johnson's War

At the time of the Kennedy assassination, Vietnam had 16,000 American advisers in place, most of whom weren't involved in daily combat operations. Kennedy's Vice President and new replacement, however, Lyndon Baines Johnson, was not convinced that this path was leading to success, and came to believe that more aggressive action was needed. Seizing on a dubious incident in which Vietnamese patrol boats attacked American destroyers[1] in the Gulf of Tonkin, Johnson used pro-war sentiment in Congress to pass a resolution that gave him powers to conduct military action without an explicit declaration of war. To put it simply, Johnson wanted to fight this war "in cold blood": "This meant that America would go to war in Vietnam with the precision of a surgeon with little noticeable impact on domestic culture. A limited war called for limited mobilization of resources, material and human, and caused little disruption in everyday life in America." (source) In essence, it would be a war whose only impact would be felt by the Vietnamese--American life and society would go on without any notice of the events in Vietnam, thus leaving Johnson to pursue his first great love, his "Great Society", a domestic agenda designed to fix many of US society's ills, such as poverty[2]. History, of course, knows better, and--perhaps cruelly--calls the Vietnam conflict "Johnson's War".

Initially, it must be noted that Vietnam-as-disaster is a more recent perception; Americans polled as late as 1967 were convinced that the war was a good thing, that Communism needed to be stopped and that Vietnam, should it fall, would be the first of a series of nations to succumb to Communist subversion. This "Domino Theory" was a common refrain for American politics in the latter half of the 20th century. Concerns of this sort plagued American foreign policy ever since the Communists successfully or nearly-successfully subverted several European governments during hte latter half of the 1940's, and then China in the 50's. (It must be noted that Eisenhower and John Foster Dulles, formulators of the theory, never included Vietnam in their ring of dominos that must be preserved, and in fact Eisenhower was surprisingly apathetic about Vietnam during some of his meetings with Kennedy during the White House transition.)

In 1968, however, the Vietnam experience turned significantly, as the North Vietnamese and Viet Cong launched the Tet Offensive, a campaign that put to lie all of the reassurances of the American government that it was winning the war in Vietnam. Ironically, as had been the case for much of the war, the NVA/VC forces lost a substantial number of troops, far more than their American opponents, yet the Tet Offensive is widely considered by historians to be the breaking point of American will in the war. Following that, popular opinion turned on Johnson, and in a dramatic news conference, he announced that he would not seek re-election. Furthermore, he announced that he would seek a negotiated settlement with the Vietnamese.

Nixon's Promise

Unfortunately, American negotiating position was seriously weakened by the very protests that had brought the Americans to the negotiating table in the first place; NVA/VC leadership recognized that the NVA/VC forces, despite staggering military losses that nearly broke them (several times), could simply continue to do as they were doing, and wring concessions from the Americans without offering any in return. Running on a platform that consisted mostly of a promise to "Get America out of Vietnam", Johnson's successor, Republican Richard Nixon, tried several tactics to bring pressure to the NVA/VC forces to bargain, including increased air-combat presence (such as the Christmas bombings and Operation Menu) and regular violations of nearby Laos and Cambodia, pursuing the line of supplies from North Vietnam to cells in South Vietnam. Nothing worked, however, and in 1973 Nixon's administration signed the Paris Peace Agreement, ending American involvement in that conflict. Two years later, South Vietnam had been overrun, and on April 30, 1975, Communist forces captured Saigon, the capital of Vietnam, forcing the evacuation of the American embassy and the most memorable image of the war, that of streams of fleeing people seeking space on the Huey helicopter perched on the roof of the embassy.

War's End

The Second South Indochina War was over, America had experienced its most profound defeat ever in its history, and Vietnam became synonymous with "quagmire". Its impact on American culture was immeasurable, as it taught an entire generation of Americans to fear and mistrust their government, it taught American leaders to fear any amount of US military casualties, and brought the phrase "clear exit strategy" directly into the American political lexicon. Not until Ronald Reagan used the American military to "liberate" the small island nation of Grenada would American military intervention be considered a possible tool of diplomacy by American presidents, and even then only with great sensitivity to domestic concern, as Bill Clinton would find out during his peacekeeping missions to Somalia and Kosovo. In quantifiable terms, too, Vietnam's effects clearly fell short of Johnson's goal of a war in "cold blood". Final tally: 3 million Americans served in the war, 150,000 seriously wounded, 58,000 dead, and over 1,000 MIA, not to mention nearly a million NVA/Viet Cong troop casualties, 250,000 South Vietnamese casualties, and hundreds of thousands--if not millions, as some historians advocated--of civilian casualties.

Lessons of Vietnam

Vietnam presents an interesting problem to the student of military and political history--exactly what went wrong, when, and where? Obviously, the US government's unwillingness to admit its failures during the war makes for an easy scapegoat, but no government in the history of modern society has ever been entirely truthful with its population about its fortunes of war; one such example includes (but is not limited to) the same US government's careful censorship of activities during World War Two, fifty years earlier, known in American history as "the last 'good' war". It's also tempting to point to the lack of a military objective as the crucial failing point of Vietnam, but other non-military objectives have been successfully executed by the US and other governments without the kind of colossal failure accompanying Vietnam's story. Moreover, it's important to note that the US did, in fact, have a clear objective in what it wanted out of the conflict in South Indochina: to stop the fall of the South Vietnam government, and, barring that, the cessation of the "spread" of Communism. Was it the reluctance of the US government to unleash the military to its fullest capabilities, as General William Westmoreland always claimed? Certainly the failure in Vietnam was not a military one; the casualty figures make it clear that the US, by any other measure, was clearly winning.

So what were the principal failures in Vietnam? And, more importantly, what does all this have to do with O/R Mapping?

Vietnam and O/R mapping

In the case of Vietnam, the United States political and military apparatus was faced with a deadly form of the Law of Diminishing Returns. In the case of automated Object/Relational Mapping, it's the same concern--that early successes yield a commitment to use O/R-M in places where success becomes more elusive, and over time, isn't a success at all due to the overhead of time and energy required to support it through all possible use-cases. In essence, the biggest lesson of Vietnam--for any group, political or otherwise--is to know when to "cut bait and run", as fishermen say. Too often, as was the case in Vietnam, it is easy to justify further investment in a particular course of action by suggestion that abandoning that course somehow invalidates all the work--or, in Vietnam's case, the lives of American soldiers--that have already been paid. Phrases like "We've gone this far, surely we can see this thing through" and "To back out now is to throw away everything we've sacrificed up until this point" become commonplace. At least during the later, deeply bitter years of the second half of Vietnam, questions of patriotism came into question: if you didn't support the war, you were clearly a traitor, a Communist, obviously "unAmerican", disrespectful of all American veterans of any war fought on any soil for whatever reason, and you probably kicked your dog to boot. (It didn't help the protestors' cause that they blamed the soldiers for the war, holding them accountable--sometimes personally--for the decisions made by military and political leaders, most of whom neither the soldiers nor the protestors had ever met.)

Recognizing that all analogies fail eventually, and that the subject of Vietnam is deeper than this essay can examine, there are still lessons to be learned here in an entirely different arena. One of the key lessons of Vietnam was the danger of what's colloquially called "the Slippery Slope": that a given course of action might yield some early success, yet further investment into that action yields decreasingly commensurate results and increasibly dangerous obstacles whose only solution appears to be greater and greater commitment of resources and/or action. Some have called this "the Drug Trap", after the way pharmaceuticals (legal or illegal) can have diminished effect after prolonged use, requiring upped dosage in order to yield the same results. Others call this "the Last Mile Problem": that as one nears the end of a problem, it becomes increasingly difficult in cost terms (both monetary and abstract) to find a 100% complete solution. All are basically speaking of the same thing--the difficulty of finding an answer that allows our hero to "finish off" the problem in question, completely and satisfactorily.

We begin the analysis of Object/Relational Mapping--and its relationship to the Second South Indochina War--by examining the reasons for it in the first place. What drives developers away from using traditional relational tools to access a relational database, and to prefer instead tools such as O/R-M's?

The Object-Relational Impedence Mismatch

To say that objects and relational data sets are somehow constructed differently is typically not a surprise to any developer who's ever used both; except in extremely simplistic situations, it becomes fairly obvious to recognize that the way in which a relational data store is designed is subtly--and yet profoundly--different than how an object system is designed.

Object systems are typically characterized by four basic components: *identity*, *state*, *behavior* and *encapsulation*. Identity is an implicit concept in most O-O languages, in that a given object has a unique identity that is distinct from its state (the value of its internal fields)--two objects with the same state are still separate and distinct objects, despite being bit-for-bit mirrors of one another. This is the "identity vs. equivalence" discussion that occurs in languages like C++, C# or Java, where developers must distinguish between "a == b" and "a.equals(b)". The behavior of an object is fairly easy to see, a collection of operations clients can invoke to manipulate, examine, or interact with objects in some fashion. (This is what distinguishes objects from passive data structures in a procedural language like C.) Encapsulation is a key detail, preventing outside parties from manipulating internal object details, thus providing evolutionary capabilities to the object's interface to clients.[3]. From this we can derive more interesting concepts, such as *type*, a formal declaration of object state and behavior, *association*, allowing types to reference one another through a lightweight reference rather than complete by-value ownership (sometimes called composition), *inheritance*, the ability to relate one type to another such that the relating type incorporates all of the related type's state and behavior as part of its own, and *polymorphism*, the ability to substitute an object in where a different type is expected.

Relational systems describe a form of knowledge storage and retrieval based on predicate logic and truth statements. In essence, each row within a table is a declaration about a fact in the world, and SQL allows for operator-efficient data retrieval of those facts using predicate logic to create inferences from those facts. [Date04] and [Fussell] define the relational model as characterized by *relation*, *attribute*, *tuple*, *relation value* and *relation variable*. A *relation* is, at its heart, a truth predicate about the world, a statement of facts (*attributes*) that provide meaning to the predicate. For example, we may define the relation "PERSON" as {SSN, Name, City}, which states that "there exists a PERSON with a Social Security Number SSN who lives in City and is called Name". Note that in a relation, attribute ordering is entirely unspecified. A *tuple* is a truth statement within the context of a relation, a set of attribute values that match the required set of attributes in the relation, such as "{PERSON SSN='123-45-6789' Name='Catherine Kennedy' City='Seattle'}". Note that two tuples are considered identical if their relation and attribute values are also identical. A *relation value*, then, is a combination of a relation and a set of tuples that match that relation, and a *relation variable* is, like most variables, a placeholder for a given relation, but can change value over time. Thus, a relation variable People can be written to hold the relation {PERSON}, and consist of the relation value

```
{ {PERSON SSN='123-45-6789' Name='Catherine Kennedy' City='Seattle'},
  {PERSON SSN='321-54-9876' Name='Charlotte Neward' City='Redmond'},
  {PERSON SSN='213-45-6978' Name='Cathi Gero' City='Redmond'} }
```

These are commonly referred to as tables (relation variable), rows (tuples), columns (attributes), and a collection of relation variables as a database. These basic element types can be combined against one another using a set of operators (described in some detail in Chapter 7 of [Date04]): restrict, project, product, join, divide, union, intersection and difference, and these form the basis of the format and approach to SQL, the universally-acceptance language for interacting with a relational system from operator consoles or programming languages. The use of these operators allow for the creation of *derived relation values*, relations that are calculated from other relation values in the database--for example, we can create a relation value that demonstrates the number of people living in individual cities by making use of the project and restrict operators across the People relation variable defined above.

Already, it's fairly clear to see that there are distinct differences between how the relational world and object world view the "proper" design of a system, and more will become apparent as time progresses. It's important to note, however, that so long as programmers prefer to use object-oriented programming languages to access relational data stores, there will always be some kind of object-relational mapping taking place--the two models are simply too different to bridge silently. (Arguably, the same is true of object-oriented and procedural programming, but that's another argument for another day.) O/R mappings can take place in a variety of forms, the easiest of which to recognize is the automated O/R mapping tool, such as TopLink, Hibernate / NHibernate, or $g(Gentle.NET). Another form of mapping is the hand-coded one, in which programmers use relational-oriented tools, such as JDBC or ADO.NET, to access relational data and extract it into a form more pleasing to object-minded developers "by hand". A third is to simply accept the shape of the relational data as "the" model from which to operate, and slave the objects around it to this approach; this is also known in the patterns lexicon as Table Data Gateway [PEAA, 144] or Row Data Gateway [PEAA 152]; many data-access layers in both Java and .NET use this approach and combine it with code-generation to simplify the development of that layer. Sometimes we build objects around the relational/table model, put some additional behavior around it, and call it Active Record [PEAA, 160].

In truth, this basic approach--to slave one model into the terms and approach of the other--has been the traditional answer to the impedance mismatch, effectively "solving" the problem by ignoring one half of it. Unfortunately, most development efforts, like the Kennedy Administration, aren't willing to see this through to its logical conclusion with a wholesale commitment to one approach over the other. For example, while most development teams would be happy to adopt an "objects-only" approach, doing so at the storage level implies the use of an Object Oriented DataBase Management System (OODBMS), a topic that frequently has no traction within upper management or the corporate data management team. The opposite approach--a "relational-only" approach--is almost nonsensical to consider, given the technology of the day at the time this was written[4].

Given that it's impossible, then, to "unleash the objects to their fullest capabilities", as General Westmoreland might call it, we're left with some kind of hybrid object-to-relational mapping approach, preferably one that's automated as much as possible, so that developers can focus on their Domain Model, rather than on the details of the object-to-table(s) mapping. And here, unfortunately, is where the potential quagmire begins.

The Object-to-Table Mapping Problem

One of the first and most easily-recognizable problems in using objects as a front-end to a relational data store is that of how to map classes to tables. At first, it seems a fairly straightforward exercise--tables map to types, columns to fields. Even the field types appear to line up directly against the relational column types, at least to a fairly isomorphic degree: VARCHARs to Strings, INTEGERs to ints, and so on. So it makes sense that for any given class defined in the system, a corresponding table--likely to be of the same or closely related name--is defined to go with it. Or, perhaps, if the object code is being written to an already existing schema, then the class maps to the table.

But as time progresses, it's only natural that a well-trained object-oriented developer will seek to leverage inheritance in the object system, and seek ways to do the same in the relational model. Unfortunately, the relational model does not support any sort of polymorphism or IS-A kind of relation, and so developers eventually find themselves adopting one of three possible options to map inheritance into the relational world: table-per-class, table-per-concrete-class, or table-per-class-family. Each of them carries potentially significant drawbacks.

The table-per-class approach is perhaps the most easily understood, for it seeks to minimize the "distance" between the object model and the relational model; each class in the inheritance hierarchy gets its own relational table, and objects of derived types are stitched together from relational JOINs across the various inheritance-based tables. So, for example, if an object model has the base class Person, with Student derived from Person and GraduateState derived from Student, then there will be three tables required to hold this model, PERSON, STUDENT, and GRADUATESTUDENT, each holding the fields corresponding to the class of the same name. Relating these tables together, however, requires each to have an independent primary key (one whose value is not actually stored in the object entity) so that each derived class can have a foreign key relation to its superclass's table. The reason for this is clear: a GraduateStudent object, by virtue of its IS-A relationship to Student and Person, is a collection of all three sets of state, and the distinction between the classes is largely removed by the time an object of this type is created--in both Java and .NET, for example, the object itself is a chunk of memory that holds the instance fields defined in all of its classes and superclasses, along with a pointer to the table of methods defined by that same hierarchy. This means that when querying for a particular instance at the relational level, at least three JOINs must be made in order to bring all of the object's state into the object program's working memory.

Actually, it gets worse than that--if the object hierarchy continues to grow, say to include Professor, Staff, Undergrad (inherits from Student), and a whole hierarchy of AdjunctEmployees (inheriting from Staff), and the program wants to find all Persons whose last name is Smith, then JOINs must be done for every derived class in the system, since the semantics of "find all Persons" means that the query must seek data on the PERSON table, but then do an expensive set of JOINs to bring in the rest of the data from across the rest of the database, pulling in the PROFESSOR table to fetch the rest of the data, not to mention the UNDERGRAD, ADJUCTEMPLOYEE, STAFF, and other tables. Considering that JOINs are among the most expensive expressions in RDBMS queries, this is clearly not something to undertake lightly.

As a result, developers typically adopt one of the other two approaches, more complex in outlook but more efficient when dealing with relational storage: they either create a table per concrete (most-derived) class, preferring to adopt denormalization and its costs, or else they create a single table for the entire hierarchy, often in either case creating a discriminator column to indicate to which class each row in the table belongs. (Various hybrids of these schemes are also possible, but typically don't create results that are significantly different from these two.) Unfortunately, the denormalization costs are often significant for a large volume of data, and/or the table(s) will contain significant amounts of empty columns, which will need NULLability constraints on all columns, eliminating the powerful integrity constraints offered by an RDBMS.

Inheritance mapping isn't the end of it; associations between objects, the typical 1:n or m:n cardinality associations so commonly used in both SQL and/or UML, are handled entirely differently: in object systems, association is unidirectional, from the associator to the associatee (meaning the associated object(s) have no idea they are in fact associated unless an explicit bidirectional association is established), whereas in relational systems the association is actually reversed, from the associatee to the associator (via foreign key columns). This turns out to be surprisingly important, as it means that for m:n associations, a third table must be used to store the actual relationship between associator and associatee, and even for the simpler 1:n relationships the associator has no inherent knowledge of the relations to which it associates--discovering that data requires a JOIN against any or all associated tables at some point. (When to actually retrieve that data is a subject of some debate--see the Loading Paradox, below).

The Schema-Ownership Conflict

Discussions of inheritance-to-table and association mapping schemes also reveals a basic flaw: At heart, many object-relational mapping tools assume that the schema is something that can be defined according to schemes that help optimize the O/R-M's queries against the relational data. But this belies a basic problem, that often the database schema itself is not under the direct control of developers, but instead is owned by another group within the company, typically the database administration (DBA) group. To whom does responsibility for designing the database--and deciding when schema changes are permissible--belong?

In many cases, developers begin a new project with a "clean slate", an empty relational database whose schema is theirs to define as they see fit. But, soon after the project has shipped (sometimes even earlier than that, due to political and/or "turf war" issues), it becomes apparent that the developers' ownership of the schema is temporary at best--various departments begin clamoring for reports against the database, DBAs are held accountable to the performance of the database thereby giving them cause to call for "refactoring" and denormalization of the data, and other development teams may start inquiring about how they might make use of the data stored therein. Before too long, the schema must be "frozen", thereby potentially creating a barrier to object model refactoring (see The Coupling Concern, below). In addition, these other teams will expect to see a relational model defined in relational terms, not one which supports an entirely orthogonal form of persistence--for example, the "discriminator" column from the Inheritance-to-Table Mapping Problem will represent difficulties, and arguably be all but unusable, to relational report generators such as Crystal Reports. Unless developers are willing to write all reports (and their UIs, and their printing code, and their ad-hoc capabilities...) by hand, this is usually going to be an unacceptable state of affairs.

(To be fair, this is not so much a technical problem as it is a political problem, but it still represents a serious problem regardless of its source--or solution. And as such, it still represents an impediment to an object/relational mapping solution.)

The Dual-Schema Problem

A related issue to the question of schema ownership is that in an O/R-M solution, the metadata to the system is held fundamentally in two different places: once in the database schema, and once in the object model (another schema, if you will, expressed in Java or C# instead of DDL). Updates or refactorings to one will likely require similar updates or refactorings to the other. Refactoring code to match database schema changes is widely considered to be the easier of the two--refactoring the database frequently requires some kind of migration and/or adaptation of data already within the database, where code has no such requirement. (Objects, at least in this discussion, are ephemeral in-memory instances that will disappear once the process holding them terminates. If the objects are stored in some kind of object form that can persist across process execution--such as serialized object instances stored to disk--then refactoring objects becomes equally problematic.)

More importantly, while it's not uncommon for code to be deployed specifically to a single application, frequently database instances are used by more than one application, and it's frequently unacceptable to business to trigger a company-wide refactoring of code simply because a refactoring on one application requires a similar database-driven refactoring. As a result, as the system grows over time, there will be increasing pressure on the developers to "tie off" the object model from the database schema, such that schema changes won't require similar object model refactorings, and vice versa. In some cases, where the O/R-M doesn't permit such disconnection, an entirely private database instance may have to be deployed, with the exact schema the O/R-M-based solution was built against, creating yet another silo of data in an IT environment where pressure is building to *reduce* such silos.

Entity Identity Issues

As if these problems weren't enough, we then walk into another problem, that of *identity* of objects and relations. As noted above, object systems use an implicit sense of identity, typically based on the object's location in memory (the ubiquitous *this* pointer); alternatively, this is sometimes referred to as an *OID* (Object IDentifier), usually in systems which don't directly expose memory locations, such as the object database (where an in-memory pointer is pretty useless as an identifier outside of the database process). In a relational model, however, identity is implicit in the state itself--two rows with the exact same state are typically considered a relational data corruption, as the same fact asserted twice is redundant and counterproductive. To be fair, we should be a bit more explicit here; a relational system can, in fact, permit duplicate tuples (as described above), but this is often explicitly disallowed by explicit relational constraints, such as PRIMARY KEY constraints. In those situations where duplicate values are allowed, there is no way for a relational system to determine which of the two duplicate rows are being retrieved--there is no implicit sense of identity to the relation except that offered by its attributes. The same is not true of object systems, where two objects that contain precisely identical bit patterns in two different locations of memory are in fact separate objects. (This is the reason for the distinction between "==" and ".equals()" in Java or

C#.) The implication here is simple: if the two systems are going to agree on the sense of identity, the relational system must offer some kind of unique identity concept (usually an auto-incrementing integer column) to match that of the notion of object identity.

This causes some serious concerns regarding automated O/R systems, because the sense of identity is entirely different--if two separate user sessions interact with the same relation in storage, the relational database system's concurrency systems kick in and ensure some form of concurrent access, typically via the transactional metaphor (ACID). If an O/R system retrieves a relation out of storage (essentially forming a "view" over the data), we now have a second source of data identity, one in the database (protected by the aforementioned transactional scheme), and one in the in-memory object representation of that data, which has no consistent transactional support aside from that built into the language (such as the monitors concept in Java and .NET) or libraries (such as System.Transactions in .NET 2.0), either of which can be--and unfortuantely frequently are--easily ignored by developers. Managing isolation and concurrency is not an easy problem to solve, and unfortunately the languages and platforms commonly available to developers aren't yet as consistent or flexible as the database transaction metaphor.

What complicates this problem further is that many O/R systems introduce significant caching support into the O/R layer (usually in an attempt to improve performance and avoid round-trips to the database), and this in turn presents some problems, particularly if the caching system is not a write-through cache: when does the actual "flush" to the database take place, and what does this say about transactional integrity if the application code believes the write to have occurred when in fact it hasn't? This problem in turn only compounds when the O/R system runs in multiple processes in front of the database engine, commonly found in clustered or farmed application server scenarios. Now the data identity is spread across *n+1* locations, *n* being the number of application server nodes, and *1* being the database itself. Each node must somehow signal its intent to do an update to the other nodes in order to obtain some kind of concurrency construct to prevent simultaneous access (by another instance of the same session, or by an instance of a different session accessing the same data), which takes time, killing performance. Even in the case of a read-only cache, updates to the data store must somehow be signaled to the caches running in the application server nodes, requiring server-to-client communication originating from the database; support for this is not well-understood or documented in the current crop of modern relational databases.

The Data Retrieval Mechansim Concern

So once the entity is stored within the database, how exactly do we retrieve it? In all honesty, a purely object-oriented approach would make use of object approaches for retrieval, ideally using constructor-style syntax identifying the object(s) desired, but unfortunately constructor syntax isn't generic enough to allow for something that flexible; in particular, it lacks the ability to initialize a collection of objects, and queries frequently need to return a collection, rather than just a single entity. (Multiple trips to the database to fetch entities individually is generally considered too wasteful, in both latency and bandwidth, to consider credibly as an alternative--see the Load-Time Paradox, below, for more.) As a result, we typically end up with one of Query-By-Example (QBE), Query-By-API (QBA), or Query-By-Language (QBL) approaches.

A QBE approach states that you fill out an object template of the type of object you're looking for, with fields in the object set to a particular value to use as part of the query-filtration process. So, for example, if you're querying the Person object/table for people with the last name of Smith, you set up the query like so:

```
Person p = new Person(); // assumes all fields are set to null by default
p.LastName = "Smith";
ObjectCollection oc = QueryExecutor.execute(p);
```

The problem with the QBE approach is obvious: while it's perfectly sufficient for simple queries, it's not nearly expressive enough to support the more complex style of query that frequently we need to execute--"find all Persons named Smith or Cromwell" and "find all Persons NOT named Smith" are two examples. While it's not impossible to build QBE approaches that handle this (and more complex scenarios), it definitely complicates the API significantly. More importantly, it also forces the domain objects into an uncomfortable position--they *must* support nullable fields/properties, which may be a violation of the domain rules the object would otherwise seek to support--a Person without a name isn't a very useful object, in many scenarios, yet this is exactly what a QBE approach will demand of domain objects stored within it. (Practitioners of QBE will often argue that it's not unreasonable for an object's implementation to take this into account, but again this is neither easy nor frequently done.)

As a result, usually the second step is to have the object system support a "Query-By-API" approach, in which queries are constructed by query objects, usually something of the form:

```
Query q = new Query();
q.From("PERSON").Where(
  new EqualsCriteria("PERSON.LAST_NAME", "Smith"));
ObjectCollection oc = QueryExecutor.execute(q);
```

Here, the query is not based on an empty "template" of the object to be retrieved, but off of a set of "query objects" that are used together to define a Command-style object for executing against the database. Multiple criteria are connected using some kind of binomial construct, usually "And" and "Or" objects, each of which contain unique Criteria objects to test against. Additional filtration/manipulation objects can be tagged onto the end, usually by appending calls such as "OrderBy(*field-name*)" or "GroupBy(*field-name*)". In some cases, these method calls are actually objects constructed by the programmer and strung together explicitly.

Developers quickly note that the above approach is (generally) much more verbose than the traditional SQL approach, and certain styles of queries (particularly the more unconventional joins, such as outer joins) are much more difficult--if not impossible--to represent in the QBA approach.

On top of this, we have a more subtle problem, that of the reliance on developers' dicipline: both the table name ("PERSON") and the column name in the criteria ("PERSON.LAST_NAME") are standard strings, taken as-is and fed to the system at runtime with no sort of validity-checking until then. This presents a classic problem in programming, that of the "fat-finger" error, where a developer doesn't actually query the "PERSON" table, but the "PRESON" table instead. While a quick unit-test against a live database instance will reveal the error during unit-testing, this presumes two facts--that the developers are religious about adopting unit-testing, and that the unit-tests are run against database instances. While the former is slowly becoming more of a guarantee as more and more developers become "test-infected" (borrowing Gamma's and Beck's choice of terminology), the latter is still entirely open to discussion and interpretation, owing to the fact that setting-up and tearing-down the database instance appropriately for unit tests is still difficult to do in a database. (While there are a variety of ways to circumvent this problem, few of them seem to be in use.)

We're also faced with the basic problem that greater awareness of the logical--or physical--data representation is required on the part of the developer--instead of simply focusing on how the objects are related to one another (through simple associations such as arrays or collection instances), the developer must now have greater awareness of the form in which the objects are stored, leaving the system somewhat vulnerable to database schema changes. This is sometimes obviated by a hybrid approach between the two, whereby the system will take responsibility for interpreting the associations, leaving the developer to write something like this:

```
Query q = new Query();
Field lastNameFieldFromPerson = Person.class.getDeclaredField("lastName");
q.From(Person.class).Where(new EqualsCriteria(lastNameFieldFromPerson, "Smith"));
ObjectCollection oc = QueryExecutor.execute(q);
```

Which solves part of the schema-awareness problem and the "fat-fingering" problem but still leaves the developer vulnerable to the concerns over verbosity and still doesn't address the complexity of putting together a more complex query, such as a multi-table (or multi-class, if you will) query joined on several criteria in a variety of ways.

So, then, the next task is to create a "Query-By-Language" approach, in which a new language, similar to SQL but "better" somehow, is written to support the kind of complex and powerful queries normally supported by SQL; OQL and HQL are two examples of this. The problem here is that frequently these languages are a subset of SQL and thus don't offer the full power of SQL. More importantly, the O/R layer has now lost an important "selling point", that of the "objects and only objects" mantra that begat it in the first place; using a SQL-like language is almost just like using SQL itself, so how can it be more "objectish"? While developers may not need to be aware of the physical schema of the data model (the query language interpreter/executor can do the mapping discussed earlier), developers will need to be aware of how object associations and properties are represented within the language, and the subset of the object's capabilities within the query language--for example, is it possible to write something like this?

```
SELECT Person p1, Person p2
FROM Person
WHERE p1.getSpouse() == null
  AND p2.getSpouse() == null
  AND p1.isThisAnAcceptableSpouse(p2)
  AND p2.isThisAnAcceptableSpouse(p1);
```

In other words, scan through the database and find all single people who find each other acceptable. While the "isThisAnAcceptableSpouse" method is clearly a method that belongs on the Person class (each Person instance may have its own criteria by which to judge the acceptability of another single--are they blonde, brunette, or redhead, are they making more than $100,000 a year, and so on), it's not clear if executing this method is possible in the query language, nor is it clear if it should be. Even for the most trivial implementations, a serious performance hit will be likely, particularly if the O/R layer must turn the relational column data into objects in order to execute the query. In addition, we have no guarantees that the developer wrote this method to be at all efficient, and no ways to enforce any sort of performance-aware implementation.

(Critics will argue that this is a workable problem, proposing two possible solutions. One is to encode the preference data in a separate table and make that part of the query; this will result in a hideously complicated query that will take several pages in length and likely require a SQL expert to untangle later when new preferential criteria want to be added. The other is to encode this "acceptability" implementation in a stored procedure within the database, which now removes code entirely from the object model and leaves us without an "object"-based solution whatsoever--acceptable, but only if you accept the premise that not all implementation can rest inside the object model itself, which rejects the "objects and nothing but objects" premise with which many O/R advocates open their arguments.)

The Partial-Object Problem and the Load-Time Paradox

It has long been known that network traversal, such as that done when making a traditional SQL request, takes a significant amount of time to process. (Rough benchmarks have placed this value at anywhere from three to five orders of magnitude, compared against a simple method call on either the Java or .NET platform[5]; roughly analogous, if it takes you twenty minutes to drive to work in the morning, and we call that the time required to execute a local method call, four orders of magnitude to that is roughly the time it takes to travel to Pluto, or just shy of fourteen years, one way.) This cost is clearly non-trivial, so as a result, developers look for ways to minimize this cost by optimizing the number of round trips and data retrieved.

In SQL, this optimization is achieved by carefully structuring the SQL request, making sure to retrieve only the columns and/or tables desired, rather than entire tables or sets of tables. For example, when constructing a traditional drill-down user interface, the developer presents a summary display of all the records from which the user can select one, and once selected, the developer then displays the complete set of data for that particular record. Given that we wish to do a drill-down of the Persons relational type described earlier, for example, the two queries to do so would be, in order (assuming the first one is selected):

```
SELECT id, first_name, last_name FROM person;
SELECT * FROM person WHERE id = 1;
```

In particular, take notice that only the data desired at each stage of the process is retrieved--in the first query, the necessary summary information and identifier (for the subsequent query, in case first and last name wouldn't be sufficient to identify the person directly), and in the second, the remainder of the data to display. In fact, most SQL experts will eschew the "*" wildcard column syntax, preferring instead to name each column in the query, both for performance and maintenance reasons--performance, since the database will better optimize the query, and maintenance, because there will be less chance of unnecessary columns being returned as DBAs or developers evolve and/or refactor the database table(s) involved. This notion of being able to return a part of a table (though still in relational form, which is important for reasons of closure, described above) is fundamental to the ability to optimize these queries this way--most queries will, in fact, only require a portion of the complete relation.

This presents a problem for most, if not all, object/relational mapping layers: the goal of any O/R is to enable the developer to see "nothing but objects", and yet the O/R layer cannot tell, from one request to another, how the objects returned by the query will be used. For example, it is entirely feasible that most developers will want to write something along the lines of:

```
Person[] all = QueryManager.execute(...);
Person selected = DisplayPersonsForSelection(all);
DisplayPersonData(selected);
```

Meaning, in other words, that once the Person to be displayed has been chosen from the array of Persons, no further retrieval action is necessary--after all, you have your object, what more should be necessary?

The problem here is that the data to be displayed in the first Display...() call is *not* the complete Person, but a subset of that data; here we face our first problem, in that an object-oriented system like C# or Java cannot return just "parts" of an object--an object is an object, and if the Person object consists of 12 fields, then all 12 fields will be present in every Person returned. This means that the system faces one of three uncomfortable choices: one, require that Person objects must be able to accomodate "nullable" fields, regardless of the domain restrictions against that; two, return the Person completely filled out with all the data comprising a Person object; or three, provide some kind of on-demand load that will obtain those fields if and when the developer accesses those fields, even indirectly, perhaps through a method call.

(Note that some object-based languages, such as ECMAScript, view objects differently than class-based languages, such as Java or C# or C++, and as a result, it is entirely possible to return objects which contain varying numbers of fields. That said, however, few languages possess such an approach, not even everybody's favorite dynamic-language poster child, Ruby, and until such languages become widespread, such discussion remains outside the realm of this essay.)

For most O/R layers, this means that objects and/or fields of objects must be retrieved in a lazy-loaded manner, obtaining the field data on demand, because retrieving all of the fields of all of the Person objects/relations would "clearly" be a huge waste of bandwidth for this particular scenario. Typically, the object's entire set of fields will be retrieved when any field not-yet-returned is accessed. (This approach is preferred to a field-by-field approach because there's less chance of the "N+1 query problem", in which retrieving all the data from an object requires 1 query to retrieve the primary key + N queries to retrieve each field from the table as necessary. This minimizes the bandwidth consumed to retrieve data--no unaccessed field will have its data retrieved--but clearly fails to minimize network round trips.)

Unfortunately, fields within the object are only part of the problem--the other problem we face is that objects are frequently associated with other objects, in various cardinalities (one-to-one, one-to-many, many-to-one, many-to-many), and an O/R mapping has to make some up-front decisions about when to retrieve these associated objects, and despite the best efforts of the O/R-M's developers, there will always be common use-cases where the decision made will be exactly the wrong thing to do. Most O/R-M's offer some kind of developer-driven decision-making support, usually some kind of configuration or mapping file, to identify exactly what kind of retrieval policy will be, but this setting is global to the class, and as such can't be changed on a situational basis.

Summary

Given, then, that objects-to-relational mapping is a necessity in a modern enterprise system, how can anyone proclaim it a quagmire from which there is no escape? Again, Vietnam serves as a useful analogy here--while the situation in South Indochina required a response from the Americans, there were a variety of responses available to the Kennedy and Johson Administrations, including the same kind of response that the recent fall of Suharto in Malaysia generated from the US, which is to say, none at all. (Remember, Eisenhower and Dulles didn't consider South Indochina to be a part of the Domino Theory in the first place; they were far more concerned about Japan and Europe.)

Several possible solutions present themselves to the O/R-M problem, some requiring some kind of "global" action by the community as a whole, some more approachable to development teams "in the trenches":

1. **Abandonment.** Developers simply give up on objects entirely, and return to a programming model that doesn't create the object/relational impedance mismatch. While distasteful, in certain scenarios an object-oriented approach creates more overhead than it saves, and the ROI simply isn't there to justify the cost of creating a rich domain model. ([Fowler] talks about this to some depth.) This eliminates the problem quite neatly, because if there are no objects, there is no impedance mismatch.
2. **Wholehearted acceptance.** Developers simply give up on relational storage entirely, and use a storage model that fits the way their languages of choice look at the world. Object-storage systems, such as the db4o project, solve the problem neatly by storing objects directly to disk, eliminating many (but not all) of the aforementioned issues; there is no "second schema", for example, because the only schema used is that of the object definitions themselves. While many DBAs will faint dead away at the thought, in an increasingly service-oriented world, which eschews the idea of direct data access but instead requires all access go through the service gateway thus encapsulating the storage mechanism away from prying eyes, it becomes entirely feasible to imagine developers storing data in a form that's much easier for them to use, rather than DBAs.
3. **Manual mapping.** Developers simply accept that it's not such a hard problem to solve manually after all, and write straight relational-access code to return relations to the language, access the tuples, and populate objects as necessary. In many cases, this code might even be automatically generated by a tool examining database metadata, eliminating some of the principal criticism of this approach (that being, "It's too much code to write and maintain").
4. **Acceptance of O/R-M limitations.** Developers simply accept that there is no way to efficiently and easily close the loop on the O/R mismatch, and use an O/R-M to solve 80% (or 50% or 95%, or whatever percentage seems appropriate) of the problem and make use of SQL and relational-based access (such as "raw" JDBC or ADO.NET) to carry them past those areas where an O/R-M would create problems. Doing so carries its own fair share of risks, however, as developers using an O/R-M must be aware of any caching the O/R-M solution does within it, because the "raw" relational access will clearly not be able to take advantage of that caching layer.
5. **Integration of relational concepts into the languages.** Developers simply accept that this is a problem that should be solved by the language, not by a library or framework. For the last decade or more, the emphasis on solutions to the O/R problem have focused on trying to bring objects closer to the database, so that developers can focus exclusively on programming in a single paradigm (that paradigm being, of course, objects). Over the last several years, however, interest in "scripting" languages with far stronger set and list support, like Ruby, has sparked the idea that perhaps another solution is appropriate: bring relational concepts (which, at heart, are set-based) into mainstream programming languages, making it easier to bridge the gap between "sets" and "objects". Work in this space has thus far been limited, constrained mostly to research projects and/or "fringe" languages, but several interesting efforts are gaining visibility within the community, such as functional/object hybrid languages like Scala or F#, as well as direct integration into traditional O-O languages, such as the LINQ project from Microsoft for C# and Visual Basic. One such effort that failed, unfortunately, was the SQL/J strategy; even there, the approach was limited, not seeking to incorporate sets into Java, but simply allow for embedded SQL calls to be preprocessed and translated into JDBC code by a translator.
6. **Integration of relational concepts into frameworks.** Developers simply accept that this problem is solvable, but only with a change of perspective. Instead of relying on language or library designers to solve this problem, developers take a different view of "objects" that is more relational in nature, building domain frameworks that are more directly built around relational constructs. For example, instead of creating a Person class that holds its instance data directly in fields inside the object, developers create a Person class that holds its instance data in a RowSet (Java) or DataSet (C#) instance, which can be assembled with other RowSets/DataSets into an easy-to-ship block of data for update against the database, or unpacked from the database into the individual objects.

Note that this list is not presented in any particular order; while some are more attractive to others, which are "better" is a value judgment that every developer and development team must make for themselves.

Just as it's conceivable that the US could have achieved some measure of "success" in Vietnam had it kept to a clear strategy and understood a more clear relationship between commitment and results (ROI, if you will), it's conceivable that the object/relational problem can be "won" through careful and judicious application of a strategy that is celarly aware of its own limitations. Developers must be willing to take the "wins" where they can get them, and not fall into the trap of the Slippery Slope by looking to create solutions that increasingly cost more and yield less. Unfortunately, as the history of the Vietnam War shows, even an awareness of the dangers of the Slippery Slope is often not enough to avoid getting bogged down in a quagmire. Worse, it is a quagmire that is simply too attractive to pass up, a Siren song that continues to draw development teams from all sizes of corporations (including those at Microsoft, IBM, Oracle, and Sun, to name a few) against the rocks, with spectacular results. Lash yourself to the mast if you wish to hear the song, but let the sailors row.

Endnotes

[1] Later analysis by the principals involved--including then-Secretary of Defense Robert McNamara--concluded that half of the attack never even took place.

[2] It is perhaps the greatest irony of the war, that the man Fate selected to lead during America's largest foreign entanglement was a leader whose principal focus was entirely aimed within his own shores. Had circumstances not conspired otherwise, the hippies chanting "Hey, hey LBJ, how many boys did you kill today" outside the Oval Office could very well have been Johnson's staunchest supporters.

[3] Ironically, encapsulation, for purposes of maintenance simplicity, turns out to be a major motivation for almost all of the major innovations in Linguistic Computer Science--procedural, functional, object, aspect, even relational technologies ([Date02]) and other languages all cite "encapsulation" as major driving factors.

[4] We could, perhaps, consider stored procedure languages like T-SQL or PL/SQL to be "relational" programming languages, but even then, it's extremely difficult to build a UI in PL/SQL.

[5] In this case, I was measuring Java RMI method calls against local method calls. Similar results are pretty easily obtainable for SQL-based data access by measuring out-of-process calls against in-process calls using a database product that supports both, such as Cloudscape/Derby or HSQL (Hypersonic SQL).

References

[Fussell]: *Foundations of Object Relational Mapping*, by Mark L. Fussell, v0.2 (mlf-970703)

[Fowler] *Patterns of Enterprise Application Architecture*, by Martin Fowler

[Date04]: *Introduction to Database Systems*, 8th Edition, by Chris Date.

[Neward04]: *Effective Enterprise Java*