Foundations of
Object Relational Mapping
v0.2 [mlf-970703]
Mark L. Fussell

1220 N. Fair Oaks Ave, #1314

Sunnyvale, CA 94089

408.734-9068

Mark.Fussell@ChiMu.com

www.chimu.com

Table of Contents

Overview

This document describes general concepts needed for object-relational mapping. It serves as an introduction to the issues involved in doing general object-relational mapping and provides a foundation for understanding frameworks that supports that mapping.

The document begins by describing object modeling and relational modeling. It then covers how to integrate objects into the relational model. This is followed by how to implement that integration with current database systems. Next, the issues for client-server objects are covered independently from the whole mapping problem. This leads to the conclusion, which finishes the preparation for analyzing specific object-relational approaches by reviewing the topics and describing object-relational approaches.

Introduction

Object-relational mapping is the process of transforming between object and relational modeling approaches and between the systems that support these approaches. Doing a good job at object-relational mapping requires a solid understanding of object modeling and relational modeling, how they are similar, and how they are different. Ideally we should have a single integrated model that described both approaches. This would ensure we understand and explicitly document both concepts and their relationships. This document will present what we believe to be the only correct integration of the two worlds that is suitable for implementation on a relational database.

Difficulties occur when we have to deal with the real systems implementing object and relational models. These systems have implementations that are deficient or inconsistent with the theoretical approaches. Relational databases have been deficient for multiple decades in correctly implementing the core concepts of relational theory. On the other hand, object modeling is not standardized, so each programming environment implements its own variation. Because of these deficiencies, object-relational mapping is more complicated than it needs to be.

Fortunately, object modeling and relational modeling have such different concerns that they are actually extremely compatible. Relational theory is concerned with knowledge and object techniques are primarily concerned with behavior. Mapping between the two models requires deciding how the two worlds can refer to each other. We will first describe the two worlds in more detail and then show how they can be integrated.

    July 23, 1997

Object Modeling

Object modeling describes systems as built out of objects: programming abstractions that have identity, behavior, and state. Objects are an abstraction beyond abstract data types (ADTs), where data and variables are merged into a single unifying concept. As such the object modeling includes many other concepts: abstraction, similarity, encapsulation, inheritance, modularity, and so on. See references for Object-Oriented design ([Booch 95], [Rumbaug+BPEL 91], [Kilov+R 94]) for more information on the concepts of object modeling.

For this paper, we will be concerned with the basic object concepts of identity, behavior, state, and encapsulation.   We will also need some higher level concepts of type, association, class, and inheritance. Each of these will be defined below.

Basic Concepts

Identity

Objects have **identity**, which distinguishes them from all other objects. This is the crucial step for describing how objects are different from ADTs. When an object is created it is distinguishable from all other objects whether its state (or "value") happens to be equal.

State

Because an object can be distinguished independently of its "value", it has a **state**: the current value associated with this identity. Objects can have a single state throughout their life (which would make them degenerately like ADTs) or can go through many state transitions. Because objects are encapsulated, the state is an abstraction and is only visible by examining the behavior of the object.

Behavior

Objects provide an abstraction that clients can interact with. The **behavior** of an object is the collection of operations an object provides (its **interface**), the responses these operations give to the caller, and the changes the operations cause to the object (and other objects in the system). All interactions with an object must be through its interface and all knowledge about an object is from its behavior (returned values or side effects) to the interface interaction.

Encapsulation

Encapsulation provides an abstraction and prevents external parties from seeing the implementation details of that abstraction. For objects, clients can interact with the public behavior of the object (and by so doing identify the state of an object) but they can not see how the behavior and the state are implemented.

Higher-level Concepts

Type

A **type** is the specification of an interface that objects will support. An object implements a type if it provides the interface described by the type. All objects of the same type can be interacted with through the same interface. An object can implement multiple types at the same time.

Associations

Types can be **associated** with other types, which specifies that the objects of one type can be **linked** to objects of the other type. Having a link provides the ability to traverse from one object to the other objects involved in the link.

Class

One approach to implementing objects is to have a **class**, which defines the implementation for multiple objects. A class defines what types the objects will implement, how to perform the behavior required for the interface and how to remember state information. Then each object will only need to remember its individual state. Although using classes is by far the most common object approach, it is not the only approach (using prototypes is another approach). It is really peripheral to the core concepts of object-oriented modeling.

Inheritance

Inheritance can apply to types or to classes. When applied to types, inheritance specifies that if an object is of 'Type B' where Type B inherits from 'Type A' than that object can be used just like an object of 'Type A'. Type B is said to **conform to** Type A and all objects that are 'Type B's are also 'Type A's.

When applied to Classes, inheritance specifies that a class uses the implementation of another class with possible overriding modification. This frequently implies type inheritance but that may not always be the case.

Summary

Object models are different from other modeling techniques because they have merged the concept of variables and abstract data types into an abstract variable type: an object. Objects have identity, state, and behavior; Object models are built out of systems of these objects. To make object modeling easier, there are concepts of type, inheritance, association, and possibly class. Although object modeling is only a small step from data type oriented programming, it produces a significantly different feel and structure for programs. Object modeling's focus on identity and behavior is completely different from the relational model's focus on information.

Relational Modeling

Relational modeling describes information as predicate logic and truth statements. We give the relational database a logical model and tell it truth axioms about the world. From this information model a relational database can remember and return the original information as well as prove "new" (derived) truths. We must also have a way to be sure we understand what we are telling the database and what the database is telling us: we must be able to translate between the human knowledge and the database model. All of this is accomplished in the relational model through well-defined terms like relation, tuple, domain, and database.

Relational Terminology

The following describes the concepts to the relational model. Most of these are identical to the definitions given in [Date 95], but I draw a finer distinction between certain terms (specifically relation and relation value).

Relation

A **relation** is a truth predicate. It defines what attributes are involved in the predicate and what the meaning of the predicate is. Frequently the meaning of the relation is not represented explicitly, but this is a very significant source for human error in using the database system. An example of a relation is:

> ***Person:*** *{SSN#, Name, City} There exists a person with social security number SSN#, who has the name Name, and lives in a city named City.*

Attribute

An **attribute** identifies a name that participates in the relation and specifies the domain from which values of the attribute must come. In the above relation, *Name* is an attribute defined over the STRING domain. The above relation should explicitly identify the domains for each attribute:

> ***Person:*** *{SSN# :* SSN*, Name :* STRING*, City :* CITYNAME*} There exists a person with social security number SSN#, who has the name Name, and lives in a city named City.*

Domain

A **domain** is simply a data type. It specifies a data abstraction: the possible values for the data and the operations available on the data. For example, a String can have zero or more characters in it, and has operations for comparing strings, concatenating string, and creating strings.

Tuple

A **tuple** is a truth statement in the context of a relation. A tuple has attribute values which match the required attributes in the relation and that state the condition that is known to be true. An example of a tuple is:

> <**Person** *SSN#* = "**123-45-6789**" *Name* = "**Art Larsson**" *City* = "**San Francisco**">

Tuples are values and two tuples are identical if their relation and attribute values are equal. The ordering of attribute values is immaterial.

Attribute Value

An **attribute value** is the value for an attribute in a particular tuple. An attribute value must come from the domain that the attribute specifies.

Relation Value

A **relation value** is composed of a relation (the heading) and a set of tuples (the body). All the tuples must have the same relation as the heading and, because they are in a set, the tuples are unordered and have no duplicates. A relation value could be shown as a set of tuples:

{ <**Person** *SSN#* = "**123-45-6789**" *Name* = "**Art Larsson**" *City* = "**San Francisco**">,
<**Person** *SSN#* = "**231-45-6789**" *Name* = "**Lino Buchanan**" *City* = "**Philadelphia**">,
<**Person** *SSN#* = "**321-45-6789**" *Name* = "**Diego Jablonski**" *City* = "**Chicago**"> }

It is more common and concise to show a relation value as a table.

: Person

SSN#	Name	City
123-45-6789	Art Larrson	San Francisco
231-45-6789	Lino Buchanan	Philadelphia
321-45-6789	Diego Jablonski	Chicago

In this representation, the heading is specified by the ": Person", the attributes of the heading are ordered (for presentation only) by the column headings, and the rows define what tuples exist.

All ordering within the table is artificial and meaningless. The following is a different presentation of the identical relation value.

: Person

City	SSN#	Name
Philadelphia	231-45-6789	Lino Buchanan
San Francisco	123-45-6789	Art Larrson
Chicago	321-45-6789	Diego Jablonski

Relation Variable

A **relation variable** holds onto a single relation value at any point in time, but can change value at any point in time. Relation variables are typed to a particular relation, so they will always hold relation values that have a heading with that relation. A relation variable would look like:

People : Person

This shows the variable name "People" and the variable relation type "Person".

Using the tabular structure from above, we can show a relation variable and its current value.

People : Person

City	SSN#	Name
Philadelphia	231-45-6789	Lino Buchanan
San Francisco	123-45-6789	Art Larrson
Chicago	321-45-6789	Diego Jablonski

Database

A **database** is a collection of relation variables. It describes the complete state of an information model, can change state (by changing the relation variables), and can answer questions about its particular state.

Base Relation Values

A **base relation value** has been explicitly told to the database at some point in time. The above People relation could be such a base relation, in which case we explicitly told the database that Lino Buchanan and Art Larrson are people.

Derived Relation Values

Derived relation values are calculated from other relation values known to the database. For our example data, the number of people located in each city is:

: PersonCount

City	Count
Philadelphia	1
San Francisco	1
Chicago	1

This can be derived from the information in the "People" relation variable.

Derived relation values are most commonly the result of relational expressions and queries. They are also frequently permanently remembered (and recalculated) through **views**: derived relation variables.

Coupling between relations, variables, and values

Relations, variables, and values are more inter-linked than they may first appear to be. Because a relation includes a meaning, a relation variable must have the same meaning as the relation. So defining a variable "HappyPeople : Person" does not make sense because the predicate for 'Person' does not describe happiness. What we are really saying is we have a new relation 'HappyPerson' which is almost identical to person but has a slightly extended meaning:

> **HappyPerson:** *{SSN#, Name, City} There exists a person with social security number SSN#, who has the name Name, and lives in a city named City, and that person is happy.*

or using a more concise definition

> **HappyPerson** extends **Person:** *And that person is happy.*

New anonymous relations are produced for all derived values (except the identity transformation). This can cause confusions because the anonymous relations are not well defined and different users of the database may have different interpretations.

Common Database Terminology

The previous section defined the correct relational model terminology, but databases have had a long history with different terms that are now overloaded to apply in the relational context. These common terms are not as precise as the relational terms, but the approximate equivalencies between these terms are given in the following table.

Common	Relational
table	relation variable
row	tuple
column	attribute
column value	attribute value
database	database

Summary

Relational modeling works in terms of predicates, truth axioms, and derivable truth statements. Relations define the possible truth statements, tuples describe the current known truths, relation values collect truth statements together, relation variables remember values, and relation expressions can derive new values.

The relational model is very different from the object model. In many ways the difference is helpful because in most areas object and relational modeling are orthogonal: their concerns are completely different. The next chapter introduces how to integrate the two approaches.

Objects integrated into the Relational Model

Object modeling describes a system through objects that have identity, behavior, and encapsulated state. Relational models describe a system by information. How can a relational model support object modeling? Relational modeling seems to have no way of representing any of the object modeling properties nicely. Tuples have neither identity nor encapsulation. Tuple attribute values are encapsulated but are pure values, so they have neither identity nor state. This is what is frequently called an impedance-mismatch between object approaches and relational databases.

Fortunately the impedence mismatch is not really there. Predicate logic is quite good at describing the state of the world (or a model of the world), so relational databases must be quite good at describing the state of an object model. To see how easily they can model an object model's state, we will first expand the relational model slightly.

Objects as Tuple Attribute Values

What if we allowed Objects (with identity and state) to be tuple attribute values?

Instead of simply having a social security number (or even a complex value like a graphic image or a rectangle) as a tuple's attribute value, we can have a 'Person' as a tuple's attribute value. This would allow us to have a predicate "This person is known to be the parent of this person" instead of saying, "The person with this SSN is known to be the parent of the person with this SSN". We don't care about a person's SSN (which might change) or any of the other attributes of a person: we have direct representations of the people themselves.

__Parent__	__Child__
Person#1	Person#2
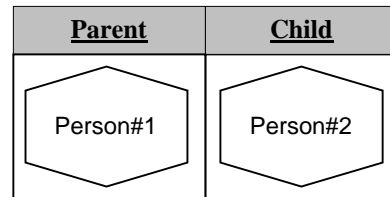
This merger provides much flexibility. Anywhere we used to have a primitive or abstract data type we can now have an object. For this to fit with the relational model, we need to enhance Domains to be able to take their "values" from a pool of existing objects or to be able to create a new object when asked.

Redundancy and Normalization

Now that we have integrated the relational model and the object model, we have the problem of "which do we ask?" Do I ask 'Person#1' for its child (object approach) or do I ask the Parenthood relation for the children of 'Person#1' (the relational approach)? Presumably I can ask both questions, but then how do we make sure changes made to one or the other place are synchronized? We clearly have a redundancy (denormalization) problem between objects with attributes and the tuples in relations.

This problem is especially obvious for the basic attribute tables (or Entity table in ER modeling) where every row lists the attributes of an object. Do we ask the object to change its attribute or do we change a row in the table?

Object	SSN	First_Name	Last_Name	Employer
Person#1	123-45-6789	Lino	Buchanan	Company#3
Person#2	234-56-7890	Art	Larrson	Company#9

Because this is a relational model, the relational features should take precedence: We only want to extend the relational model well enough so we can easily represent objects in it. Because the relational model has a complete approach for changing the state of the database we should not add a second one. So, we do not change the state of an object through the methods of an object, but must instead modify the appropriate relation variables (by adding, removing, or replacing/changing tuples) to cause the desired changes.

Object Attributes as Views

Should we allow an object to answer a question about its attributes? That would be convenient. Instead of having to look up in the Parenthood table for parents and the Person table for basic attributes we could instead ask Person#1 for it #firstName and its #parents. From the relational perspective, the person object would provide a centralized view on all the tables that can refer to a person object (i.e. all those tables which have attributes with Domains that can contain a person).

Assuming the notation <Person#1> represents that actual person object, then the query

```
'SELECT Person.SSN FROM Person WHERE Person.Object = <Person#1>'
```

is equivalent to

```
'SELECT <Person#1>.SSN'
```

This query assumes two things: first, it is obvious what relation variable (table) controls the SSN attribute for Person; and second, it is obvious which attribute in the relation variable we are starting from (in this case the "Object" attribute). In general these assumptions are unlikely to be true, so we probably would need to explicitly define how the attributes of an object are a view on the appropriate relation variables. Something like:

```
CREATE DOMAIN PERSON CLASS {
    ssn AS SELECT SSN FROM Person WHERE Object = THIS
    firstName AS SELECT First_Name FROM Person WHERE Object = THIS
    parents AS SELECT Parent FROM Parenthood WHERE Child = THIS
    children AS SELECT Child FROM Parenthood WHERE Parent = THIS
}
```

Where we added the ability to declare a domain which will have objects with identity as its values (a CLASS), the ability to declare attributes of that class as views on the database, and the new keyword THIS to refer to whichever object we are currently dealing with. Under the covers these views could be significantly optimized which would give the same performance advantage for traversals as an object database.

```
SELECT children.firstName
FROM Person.Object as Parent, Parent.children as children
WHERE Parent.firstName = 'ART' "
```

Behavior

To add object behavior requires the ability to specify method implementations for any given object. Something like:

```
CREATE DOMAIN PERSON CLASS {
    setName(newName : String) AS UPDATE ... WHERE Object = THIS
```

```
    }
```
Discussing behavioral additions is beyond the scope of this document. So far we have all the capabilities required for an information modeling and storage system, and additional behavior could be easily added to objects, tables, or the database as a whole. Effectively these are the three "types" of objects that could have behavior within our object-relational system.

Inheritance

Type based inheritance can be used as part of integrity constraints. When we specify that a domain is of a particular type, we are allow only objects that implement that type or any conformant subtype to be values in that domain. This permits similar flexibility and integrity as for a type based programming language.

For example, if we have the two classes:
```
CREATE DOMAIN PERSON CLASS {...}
CREATE DOMAIN EMPLOYEE CLASS EXTENDS PERSON {...}
```
An object from either class then can be the value of an attribute of type PERSON. We know that the object will at least support the PERSON interface although some may additionally support the EMPLOYEE interface.

Class based inheritance can be used to ease the creation of classes by having subclasses inherit the attributes and behavior of the superclass.

What about inheriting among tables? Inheriting among tables is not inheritance at all; it is just multi-relation compression and management. Tables are related by having common attributes (columns and domains) and by having common predicates. Table "inheritance" (compression) does not affect whether the tables are related or not, it is just a simple way to implement possibly related tables. In this sense, it is similar to class inheritance, but there is no reason to overload the term "inheritance".
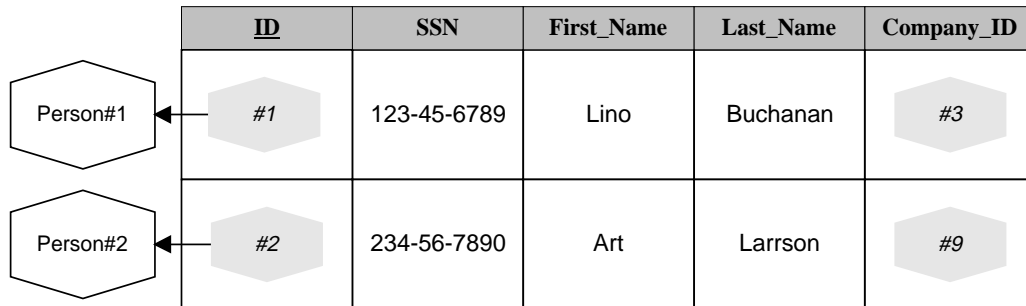
Summary

This chapter has presented a correct integration of objects into the relational model. These objects have identity, which is the foundation for the true integration of object modeling. Objects can also have attributes and behavior, but these properties must be in terms of the truth statements and predicates that are the foundation of the relational model. Attributes are read from relational expressions and state modifying behavior must alter the state of relation variables in the database. The two models are merged and enables us to have the convenience of object notation with the expressive power of predicate logic.

Using Current Relational Databases

The previous section assumed we could modify the relational model to add features we need for object-relational mapping. In the long run this may be possible but currently it is not. Current relational systems do not support objects with identity as the values of tuple attributes. Most relational systems do not support anything more than basic data types as tuple attributes. Have we returned to the impedance mismatch? No, we already have an approach that is a good integration of Relational and Object models. We only need to implement it with current technology.

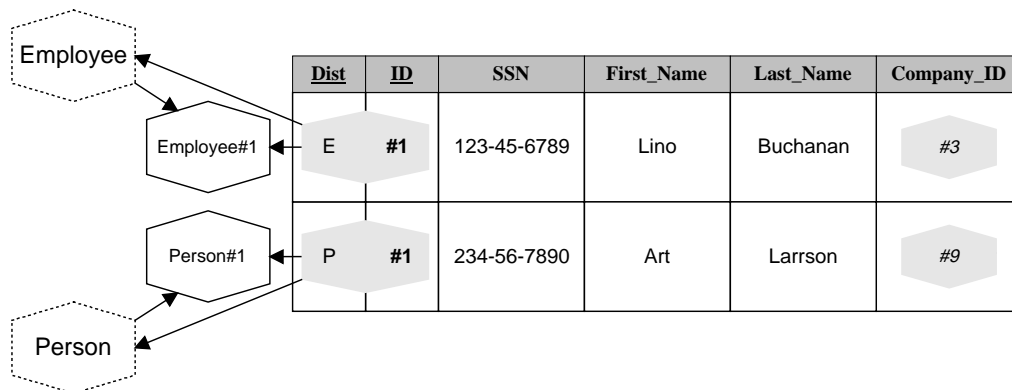## IdentityKeys: Relinquishing Objects for ObjectShadows

We can not directly store objects with identity in current relational databases. Instead we will just store an IdentityKey as an **ObjectShadow**: the existence of an IdentityKey indicates that an Object exists but that the database can not directly represent the actual object. We only have the shadow of the object, which is less convenient to work with but contains as much information.

	ID	SSN	First_Name	Last_Name	Company_ID
Person#1	#1	123-45-6789	Lino	Buchanan	#3
Person#2	#2	234-56-7890	Art	Larrson	#9

Distinguishers: Identifying an Object's Type

In the above example the type of the ObjectShadow is obvious. It is always a person for the ID column and it is always a company for the Company_ID column. What if we allow the type of a domain to vary over multiple classes? How do we identify what type of object that shadow is for? We need some manner to first identify the object's class and then use the object's IdentityKey to identify the object within that class. We need a more knowledgeable shadow.

We can add this knowledge through a distinguisher that identifies the object shadow's class. We first find the class through the distinguisher and then find the object of that class through the identityKey.

	Dist	ID	SSN	First_Name	Last_Name	Company_ID
Employee#1	E	#1	123-45-6789	Lino	Buchanan	#3
Person#1	P	#1	234-56-7890	Art	Larrson	#9

Summary

Working with current relational technology makes implementing objects in the database much more cumbersome, but the model is still valid and possible. Instead of having real objects we have object shadows which indicates the existence of a particular object. An object shadow is simply an IdentityKey when the objects are all from a single type. An object shadows is an IdentityKeys plus a distinguisher when the objects could be from one of multiple types. In all cases the ObjectShadow must be sufficient to uniquely identify the single true object that should be in its place. If we have this, we have objects.

Client-Server Object Issues

Object-relational mapping intrinsically brings in client-server issues because a relational server is separated from the client application. The client-server issues are nonetheless independent of relational mapping. This conveniently divides the problems into more manageable pieces. This chapter will discuss client-server issues solely in terms of objects. The next chapter can then focus on the specifics of relational mapping.

The major issues when dealing with client-server objects is to be able to manage the identity and state of objects on each of the client and the server, and then handle the relationships between the two systems' objects. This is different from the relational approach where everything is just a value. For that approach, the client is only getting a simple snapshot of the server state and then must state explicitly how the server state should change. The object model tries to provide a more transparent interface for the client, but this actually causes a more complex model and a more sophisticated framework.
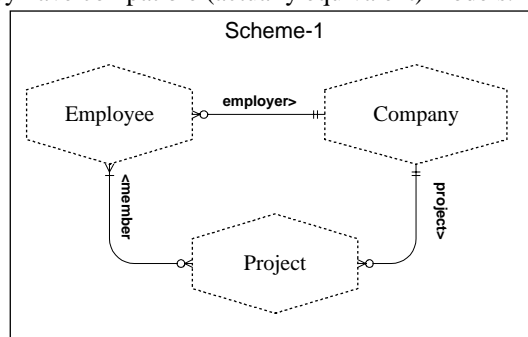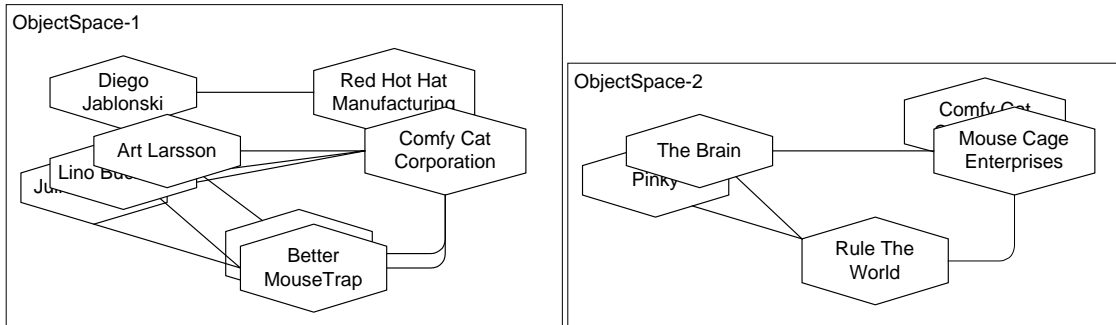
ObjectSpaces

To help describe these independent states we will define the term ObjectSpace. An ObjectSpace is a closed collection of objects based on a single (possibly very large) scheme. An ObjectSpace is self-contained and isolated from other ObjectSpaces: two ObjectSpaces can be based on the same scheme but they have no references to each other and changes to either ObjectSpaces's state will not impact the other.
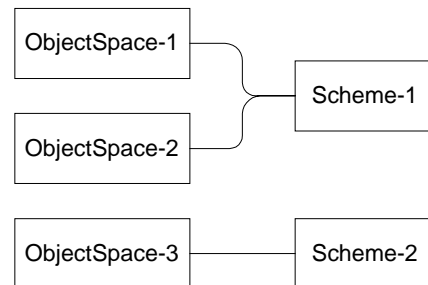
ObjectSpace Example

Scheme-1 shows a scheme for a simple business model. ObjectSpace-1 shows one complete state of a business model: it represents the model of interest to Purrfect Analysis. ObjectSpace-2 shows a second, independent, but also complete state of the same scheme. ObjectSpace-1 and ObjectSpace-2 are completely independent even though they have compatible (actually equivalent) models.

Client-Server ObjectSpaces

ObjectSpaces are, by default, unrelated. A simple application could "load" all the objects from one ObjectSpace, modify them, and then save them back (similar to working with a document). At the same time a different application could work with a different ObjectSpace. Neither application would have an impact on the other.

Client-Server applications can not work this way. The basic model for client-server applications is a single central server with multiple clients that can connect to that server. The ObjectSpaces for these different applications are not independent, but are instead inter-linked by the state of the server's ObjectSpace.

True ObjectSpace is on the Server

For client-server applications, the "true" ObjectSpace is located on the Server. This is where the one true state of the business model is kept for everyone to see and modify. If each client were to connect to the server's ObjectSpace (locking out all other clients for that time period) and to make changes, we would again return to the simple, single ObjectSpace model. Unfortunately this would prevent any type of concurrency among different users of the application. Instead each client will also have its own ObjectSpace that is replicated from the servers true ObjectSpace.
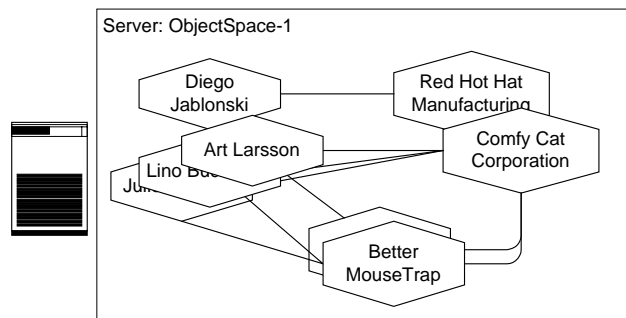

Proxies and Replicates

A **Proxy** is an object that stands in for another object (the RealSubject) and manages the client interaction with the RealSubject. A proxy pretends to be the RealSubjectto make life easy on the client: a client does not need to consider all the issues involved with talking to the RealSubjectwhich may be on a different machine or could change in different contexts. A **Replicate** is a Proxy which holds local state and performs local operations which are later propagated to the RealSubject. This is as opposed to a **Forwarder**, which holds no local state and which immediately propagates all operations to the RealSubject.

Proxies must be able to find and pretend to be their RealSubject. This causes a Proxy to have two identities: their local identity and their "real" identity. The RealSubject must have some type of **IdentityKey**, a value that uniquely identitifies the RealSubject for its proxies. Each proxy can then hold onto the RealSubject's IdentityKey for later use in finding and interacting with the RealSubject.

Replicated ObjectSpaces are on the Clients

Each client has its own ObjectSpace but this is only a temporary working-copy of the server's true ObjectSpace. Each object in a client's ObjectSpace is a replicate of a server object, and the whole client ObjectSpace forms a partial or complete replicate of the server's ObjectSpace.
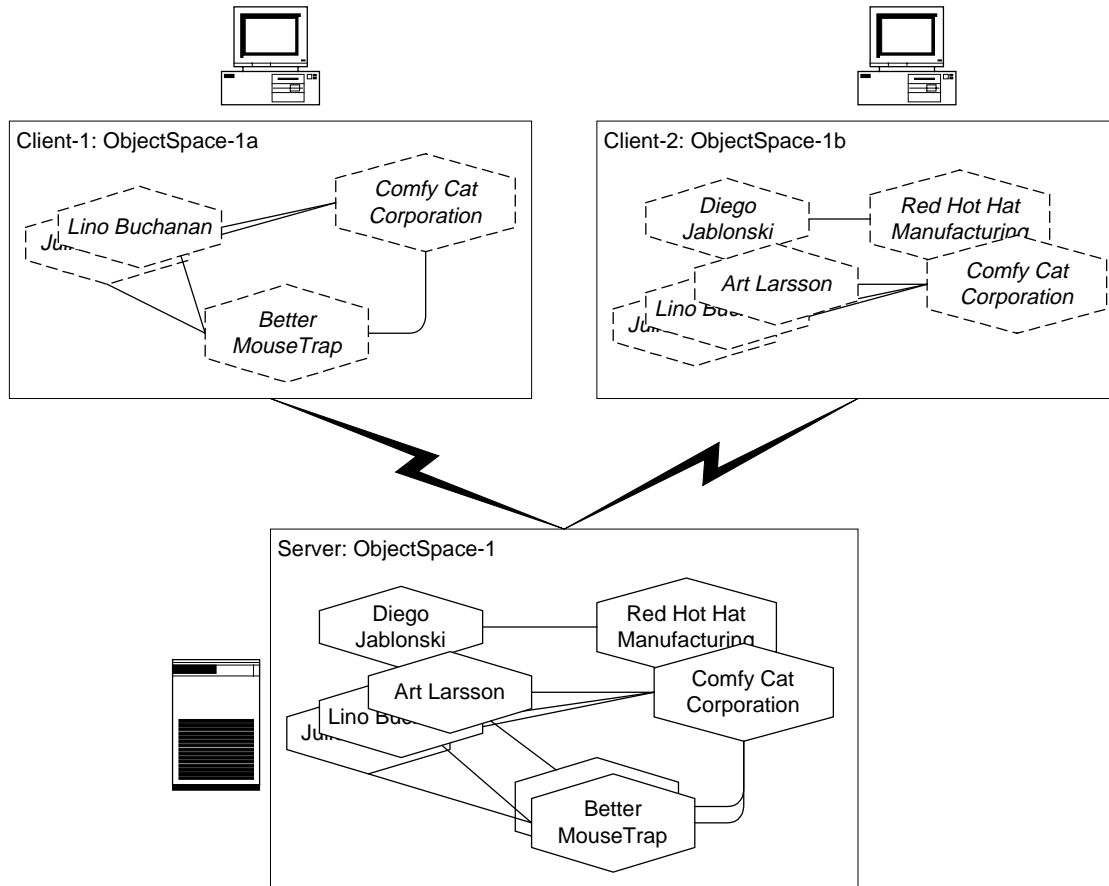
Concurrency and Conflicts

If each client deals with a non-intersecting subset of the servers ObjectSpace then we can have easy and "perfect" concurrency: clients can cause changes to their ObjectSpace replicates and propagate these to the server without worrying about a conflict with another client. For our example, changes to "Diego Jablonski" will not conflict with changes to "Lino Buchanan".

For most applications it is very unlikely that clients will always using non-intersecting subsets. For the cases where the ObjectSpaces on clients overlap, there must be some type of concurrency control between the clients and the server. Concurrency controls can vary on granularity, visibility, pessimism, functional dependency, and many other axes. There are too many variations of concurrency control to be discussed here, but I will include a couple examples.

Approach-1

A client can "check-out" a collection of objects from the server and no other client can see these objects until they are checked back in. For our example, client-1 can check-out Comfy Cat and all its employees and projects. Then client-2 could work with "Red Hot Hat" but not do anything with Comfy Cat. This automatically causes each client to have non-intersecting subsets.

Approach-2

A client can either read "check-out" or write "check-out" objects from the server. Two clients can check-out the same object as long as they are not both trying to write to it. Alternatively we can prevent dirty-reads as well: a client can only check out an object if there are no write-locks on it and can only write-lock an object if no other client has checked it out.

Approach-3

A client can replicate any object from the server but will only be able to write changes back to the server if the server object has not changed since the client produced the replicate. This is the standard optimistic locking.

Summary

Because relational mapping intrinsically involves a client-server system, we need to be able to handle the issues with that system. Most of the issues have nothing to do with relational mapping but are instead involved with having multiple ObjectSpaces between a Server and its Client applications. We need to recognize that the client objects are Replicates of the server objects, that they must keep track of the IdentityKey of their server object, and that there are many issues and approaches for handling concurrency between the multiple clients.

Conclusion

We now have all the major pieces needed to discuss and analyze object-relational mapping. We have discussed object modeling and relational modeling. We integrated the two models into a single object-extended relational model and described how to implement that model on current relational database systems. Finally we discussed the issues with client-server object models.

Not all object-relational mapping products will be able to support the complete integrated model and all the client-server capabilities. Some will choose a partial approach or simply not consider objects at all. The best approach for mapping is up to each project and product vendor. It will depend on how much of the integrated model they require, whether they can implement all the needed functionality, and the time & cost considerations. The following table identifies and describes levels of object-relational mapping sophistication starting from simply using the relational model and up to implementing a full object mapping as described in this document. At the end is a short description of the functional capabilities of an active object server.

1	**Pure Relational**	Use relational database model in the client and the server. The client UI is organized around rows and tables and uses the standard relational operations to tables and rows.
2	**Light Object Mapping**	Have some use of objects on the client and try to isolate most of the application code from the specifics of SQL. Convert rows from the database into basic objects on the client. Encapsulate hard-coded SQL calls within certain classes/methods so most of the application does not have to be coupled to them.
3	**Medium Object Mapping**	Primarily use objects on the client and write the entire application behavior in terms of these objects. Application may have to manage transactions of these objects. Flexibly convert rows from the database into the appropriate type of objects and maintain identity for these objects. Manage associations between objects of variable types. Allow retrieval queries to be specified in object model terms and then converted to the needed SQL calls. Also allow simple identity based retrieval queries to be answered locally (without the database hit).
4	**Full Object Mapping**	Completely use objects in the application code. Similar to previous level, but allow very general class data storage formats (for inheritance and composition), manage replicate vs. server state differences, allow queries to be completely executed locally when possible, combine multiple simple object queries into a single, larger database queries.
5	**Object Server**	Server itself also uses objects to organize both the information and behavior, so retrievals and updates can be completely specified in terms of the object model and then executed on either the server or client as appropriate (for processing power vs. data movement).

Choosing a technique to use from among these levels will impact performance, cost, scalability, application behavior, development time, and maintenance. No technique is the best at all the criteria. More sophisticated techniques are not appropriate for all applications and many applications work just fine with a purely relational model. Other applications will grow to a size and complexity that is easier to manage using object models, and will have to decide which technique and products will best serve their needs.

Standard Definitions

This section collects definitions of terms needed for general Object-Relational mapping.

Object	An identifiable, encapsulated entity that is interacted with by sending messages. Objects have behavior, state, and identity (but see ValueObject for a variation).
Type	Specifies the public behavior and a conceptual grouping for objects that are members of the Type
Protocol	Specifies a collection of methods that together provide a higher level interface to an object. An object can **be** a Type, an object can **support** a Protocol, and a Type can **specify support for** a Protocol.
Class	Describes the types and the implementation for a set of objects.
Factory	An object that can create other objects.
Identity	The ability to tell an object apart from another object independently of whether their type and state is equal.
Immutable	Can not be changed after being created.
ValueObject	An object that does not have identity independent of its value. A ValueObject is immutable and should be considered identical to anything that it is equal to. Primitive data types in Smalltalk (most numbers, Symbols) are ValueObjects. Java Strings are very close to ValueObjects except they are not guaranteed to be identical for the same value (they would be if they did an automatic "intern()"). Java primitive types are not Objects.
Attribute	A public property of an object that shows the state of the object. Frequently there is a minimal collection of attributes that uniquely determine the state of the object.
BasicAttribute	An attribute that takes its value from ValueObjects. This is as opposed to associations which connect two or more objects with identity.
Association	A defined relationship between two objects with identity.
Instance Variable	A private implementation to remember part of an object's state.
ObjectShadow	The information needed to see that an object exists without any true representation of the real object. Relational databases could be considered to work with ObjectShadows: they record the information about an object but never have a real object to interact with.
Proxy	An object that stands in for another object (the RealObject) and manages the client interaction with the RealObject.
Forwarder	A proxy which immediately forwards messages, possibly over process and machine boundaries, to the RealSubject.
Replicate	A proxy which holds local state and performs local operations which are later propagated to the RealSubject

Stub	A proxy which acts as a placeholder for the RealObject and must become another type of proxy (for example, forwarder or replicate) when interacted with by a client.
RealIdentity	The identity of the RealObject that a proxy represents instead of the proxy's independent identity. For proxies we are rarely interested in their own identity, we just want to know the identity of the RealObject on the server.
IdentityKey	A value that defines the RealIdentity of a Proxy.
Binding	Associating a client object to a database object, which turns the client object into a Proxy
Builder	Builds up another object that can later be extracted
Writer	Writes information directly to another object (usually another writer or a Stream)
Reader	Reads information from another object (another Reader or a Stream)
Stream	Able to sequentially retrieve or store information

.

References

Booch 94 Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994.

Brown+W Kyle Brown and Bruce G. Whitenack. "Crossing Chasms: A Pattern Language for Object-RDBMS Integration". http://www.ksccary.com/ORDBJrnl.htm

Cattell+ 96 R.G.G. Cattell, Editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, San Francisco, 1996.

Codd 90 E.F. Codd. *The Relational Model for Database Management, Version 2*. Addison-Wesley, Reading, MA, 1990

Date 95 C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, MA, 1995.

Date 95b C.J. Date. *Relational Database Writings 1991- 1994*. Addison-Wesley, Reading, MA, 1995.

Firesmith+E 95 Donald Firesmith, Edward Eykholt. *Dictionary of Object Technology: The Definitive Desk Reference*. SIGS Books, Inc., New York, NY, 1995.

Kilov+R 94 Haim Kilov and James Ross. *Information Modeling: An Object-Oriented Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

Rumbaugh+BPEL 91 James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

Stonebraker+M 96 Michael Stonebraker with Dorothy Moore. *Object-Relational DBMSs, The Next Great Wave*. Morgan Kauffman, San Francisco, CA, 1996.