# Connecting Business Objects to Relational Databases

**Joseph W. Yoder**

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
j-yoder@uiuc.edu

**Ralph E. Johnson**

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
johnson@cs.uiuc.edu

**Quince D. Wilson**

McDonnell Douglas
Technical Services Co
Springfield, IL 62704
72773.2663@compuserve.com

## Abstract

These patterns describe how to implement business objects so that they can be mapped to non object-oriented databases. There is an impedance mismatch between these technologies since objects consist of both data and behavior while a relational database consists of tables and relations between them. Although it is impossible to completely eliminate this impedance mismatch, you can minimize it by following the proper patterns. The proper patterns hide persistence from the developer so that effort can be spent on understanding the domain rather than in making objects persistent.

## Introduction

Developers of object-oriented systems that use a relational database usually spend a lot of time making objects persistent. This is because of a fundamental impedance mismatch[1] between the two paradigms; objects consist of both data and behavior and often have inheritance while relational databases consist of tables, relations and basic predicate calculus functions to return desired values.

One way to avoid the impedance mismatch between objects and relations is to use an object-oriented database. However, systems often need to store objects in a relational database. Sometimes a system needs relational calculus or the maturity of a relational database. Other times the corporate policy is to use a relational database rather than an object-oriented database. Whatever the reason, a system that stores objects in a relational database needs to provide a design that reduces the impedance mismatch.

This paper describes only a part of a pattern language for mapping objects to relations, but it describes the patterns that we thought were not adequately described elsewhere. An overview of the entire pattern language can be found in [Keller 98-2]. Some of the patterns that are described well are the patterns for designing the relational database and optimizing it [Brown 96] [Keller 97-1, 97-2, 98-1]. The *Serializer* pattern [Riehle et al. 1998] describes how to serialize objects so they can be stored and retrieved from different backends, such as flat files, relational databases, and RPC buffers.

We have used or studied several persistent object systems (GemStone [GemStone 96], TopLink [TopLink 97-1, 97-2], and ObjectLens [OS 95]). In addition, we implemented a simple persistence framework for the Illinois Department of Public Health (IDPH) in VisualAge [VA 98] for Smalltalk [Yoder & Wilson 98]. The patterns described here are in all these systems. The commercial systems usually take the patterns to more extreme measures than our framework does. We would have preferred to buy a persistence framework, but our budget could not afford the ones that were available. The applications that needed to use the persistence framework were simple, involving only a few dozen tables. Each application managed medical information about a patient. The applications share demographic information such as patient name and address and information about hospitals and doctors. Each application then covers different areas like immunizations, or blood screening for patients. Although an application could manage a lot of information about a patient, it would only examine one patient at a time. The examples will show you how these patterns work together to solve the problem of persisting the `Name` and `Address` objects needed for the applications being developed by the IDPH.

These patterns flow together and work hand-in-hand to solve the problem of the impedance mismatch described above. A *Persistence Layer* isolates the developer from the details of implementing persistence and protects the application developer from changes. The *Persistence Layer* is a special case of building a layer to protect yourself from changes in the application or in the database. One way to implement a *Persistence Layer* is with a `PersistentObject`. Another way is with a Broker [BMRSS 96].

Reading and writing objects to the database requires basic create, read, update, and delete operations. Although each object could have its own interface for accessing the database, your system will be easier to use and maintain if you provide a common set of operations to the *Persistence Layer* that all of your objects can use. Regardless of the way you implement the *Persistence Layer*, it will need to support *CRUD* (create, read, update, and delete) operations.

---

[1] The formula for impedance is: Z = square root of L/C. Even though the dictionary definition does not apply, the implementation definition does. When you have an impedance mismatch as defined above (for example 300 to 75 ohms) you have circuits that work on either size of the mismatch, but they cannot pass along the information. In order to correct the situation (in the electronics world) you place a circuit (transformer) which will pass the signals across basically translating the signals.

When you are connecting objects to databases you have the same situation. Both circuits (objects & database) work but they do not pass signals (information) back and forth between them. To fix this problem, you build a circuit (the patterns being reviewed) to place between them to pass the signals along.

*CRUD* operations eventually use SQL code to access the database. It is important to have some sort of *SQL Code Description* to build the actual SQL calls to the database.

When the values are being brought in from the database or stored back to the database, the system must perform *Attribute Mapping* to map between values from columns in the database and values being stored in an object's attributes. Part of the impedance mismatch between relational and object systems is that they have different types of data. Mapping attributes between objects and databases requires *Type Conversion* to convert the types of values between the two technologies.

It is important to save objects back to the database when their attributes have changed. Therefore, any system persisting objects should keep track of which objects have changed with some sort of *Change Manager*. This lets the system keep track of which objects have changed so that they are sure to be saved when needed. The *Change Manager* can also help minimize database access by only creating transactions to save objects that have changed.

Since every object is unique in any object-oriented system, it is important to create unique identifiers for new objects with an *OID Manager*. It is also important to support transactions to ensure that changes to an object are atomic and can be rolled back through a *Transaction Manager*. Any system accessing a RDMBS will need to provide connections to the desired databases through some sort of *Connection Manager*. It is also beneficial to handle the mappings of database table and column names through a *Table Manager*.

The pattern catalog in Table 1 outlines the patterns discussed in this paper. It lists each pattern's name along with a short description of what the pattern solves. These patterns collaborate to map domain objects to a relational database.

| **Pattern Name** | **Description** |
|---|---|
| *Persistence Layer* | Provide a layer for mapping your objects to the RDBMS or other database. |
| *CRUD* | All persistent object need, at a minimum, create, read, update, and delete operations. |
| *SQL Code Description* | Defines the actual SQL code that takes the values from the RDBMS or other database and retrieves them for the object's use and vice-versa. It is used to generate the SQL for performing the *CRUD* operations. |
| *Attribute Mapping Methods* | Maps the values between the database values and attributes. This pattern also handles complex object mappings. Populates the object(s) with the row values. |
| *Type Conversion* | Works with *Attribute Mapping Methods* to translates values from the database to the appropriate object types and vice-versa. Insures data integrity. |
| *Change Manager* | Keeps track of when an object's values have been changed for maintaining consistency with the database. It determines the need to write the values to a database table or not. |
| *OID Manager* | Generates Unique Keys for the Object Ids during an insert. |
| *Transaction Manager* | Provides a mechanism to handle transactions while saving objects. |
| *Connection Manager* | Gets and maintains a connection to the database. |
| *Table Manager* | Manages the mappings from an object to its database table(s) and column(s). |

**Table 1 - Pattern Catalog**

# Persistence Layer

## *Also Known As:*

Relational Database Access Layer

## *Motivation:*

If you build a large object-oriented business system that stores objects in a relational database, you can spend a lot of time dealing with the problems of making your objects persistent. If you aren't careful, every programmer working on the system has to know SQL and the code can become tied to the database. It can be a lot of work to convert your system from using Microsoft Access to using DB2, or even adding a few variables to an object. You need to separate your domain knowledge from knowledge of how objects are stored in a database to protect developers from these types of changes.

## *Problem:*

How do you save objects in a non object-oriented storage mechanism such as a relational database? Developers should not have to know the exact implementation.

## *Forces:*

- Writing SQL code is easy for programmers who are familiar with databases.
- Designing a good persistence mechanism takes time and is not directly related to providing features to users.
- Database access is expensive and often needs to be optimized.
- Developer should be able to solve the domain problem of an application without worrying about how to save the values to and from the database.
- Having a common interface with template methods makes for better code reuse.
- Using a single interface will force all classes to the lowest common denominator.
- The type of persistent storage may change over the life of an application.
- The domain model will probably change frequently over the life of an application.

## *Solution:*

*Provide a Persistence Layer that can populate objects from a data storage source and can save their data back to the data storage source.* This layer should hide the developer from the details of storing objects. This is really a special case of building a *Layer* [BMRSS 96] to protect yourself from changes. All persistent objects use the standard interface of the *Persistence Layer*. If the data storage mechanism changes, only the *Persistence Layer* needs to be changed. For example, the corporate direction may be to start using Oracle rather than DB2 and then switch in midstream.

The system needs to know how to store each object and to load it. Sometimes an object is stored in multiple databases over multiple media. An object that is part of a more complex object needs to keep track of which object it is a part; this is called the owning object. This owning object idea makes it easier to write complex queries. Therefore it is important for the *Persistence Layer* to provide a means to uniquely identify each object and its parent. This unique identifier and parent identifier are useful if you implement a *Proxy* [GHJV 95] pattern as a place holder for component objects.

To summarize using the pattern names, this *Persistence Layer* provides the necessary methods to provide the *CRUD* operations by building up the *SQL Code*, providing the *Attribute Mapping Methods*, *Type Converting* the objects data values, accessing the *Table Manager*, providing access to a *Transaction Manager*, and connecting to the database through the *Connection Manager*. The *Persistence Layer* will also help provide proper *Change Management* and will collaborate with the *OID Manager* to provide unique objects identifiers.

There are many ways to implement a *Persistence Layer*. Here are a few of them.

1. Use an *Object Layer* [Keller 98]. Subclass each domain object from an abstract `PersistentObject`. Each object would then inherit how to perform the necessary *CRUD* operations. This is the choice the example uses and the primary benefit is that it is easy to implement. Although it requires a little code in each domain class that is database specific, this code is segregated, and is easy to find and change. It allows optimization when necessary, though a heavily optimized system can be hard to understand.

2. Use a *Broker* that can read or write domain objects to or from the database. The broker must know the format of each domain object, and generates the SQL to read or write it. This choice separates the database code from the domain object class. It is the solution that scales best, though it requires a lot of infrastructure.

3. Compose each domain object from a set of data objects that have a one-to-one mapping to the database tables. Thus, whenever a domain object changes, it changes the corresponding data object that is then saved whenever the domain object is saved. For example, a `Patient's` values could map to the `Name` and `Address` database tables. The `Patient` would have data objects mapping to the `Name` and `Address` database tables. ObjectShare's ObjectLens for VisualWorks builds up database objects this way. The *Persistence Layer* is managed through these data objects. This alternative is simple to implement and easy to understand, though it can be slow and developers are forced to make a one-to-one mapping to database tables.

The primary decision should be made based upon the forces of the needed flexibility, scalability, and maintainability.

## *Example Implementation:*

There has been a lot of work done towards describing the details of building *Brokers* [BMRSS 96] correctly. Because of this and because we have more experience with implementing a *Layered Object*, our examples will focus around this implementation of the pattern. The other patterns described do work in the Broker implementation and we will mention briefly how this can be done in our discussions of the other patterns. However, all of our example code will describe the implementation as it relates to the `PersistentObject`.

Figure 1 is a UML class diagram for an implementation of *Persistent Layer* which map domain objects to a relational database. Note that in this example, domain objects that need to be persisted are subclasses of the `PersistentObject`. The `PersistentObject` provides for the interface to the *Persistence Layer*. The `PersistentObject` interacts with the *Table Manager*, which will provide the physical table name for the *SQL Code*. During the *SQL Code* generation, the `PersistentObject` interacts with the *Connection Manger* to provide the necessary database connection. If needed, the *OID Manager* is queried for a new unique identifier. Thus, the `PersistentObject` is the central hub for any information the domain object requires but does not contain in the instance. The `PersistentObject` provides the standard interface to the *Persistence Layer*. Once the *SQL Code* has been prepared with the cooperation of the other patterns, the SQL statement is "fired" by the Database Components. In IBM's VisualAge for Smalltalk these are the `AbtDBM*` applications.

The attributes for the `PersistentObject` are as follows:

- `objectIdentifier` – This is a unique identifier for the object and can be the database key.
- `isChanged` – Identifies whether or not the object "`isDirty`" which tells the *Persistence Layer* to write the change to the database.
- `isPersisted` – Identifies whether or not the object has ever been written to the database.
- `owningObject` – Identifies the parent object (in a complex object) and is used as a foreign key in the database. Note that the foreign key is stored with the object not the parent object.

The public methods for the `PersistentObject` are as follows:

- `save` – Writes the object data to the database. It will update or insert rows as necessary.
- `delete` – Deletes an object data from the database.

- `load:` - Returns a single instance of a class with data from the database.
- `loadAll` – Returns a collection of instances of a class with all the data from the database for that particular class. This is useful for retrieving data for selection lists.
- `loadAllLike:` - Returns a collection of instances of a class will select data from the database.

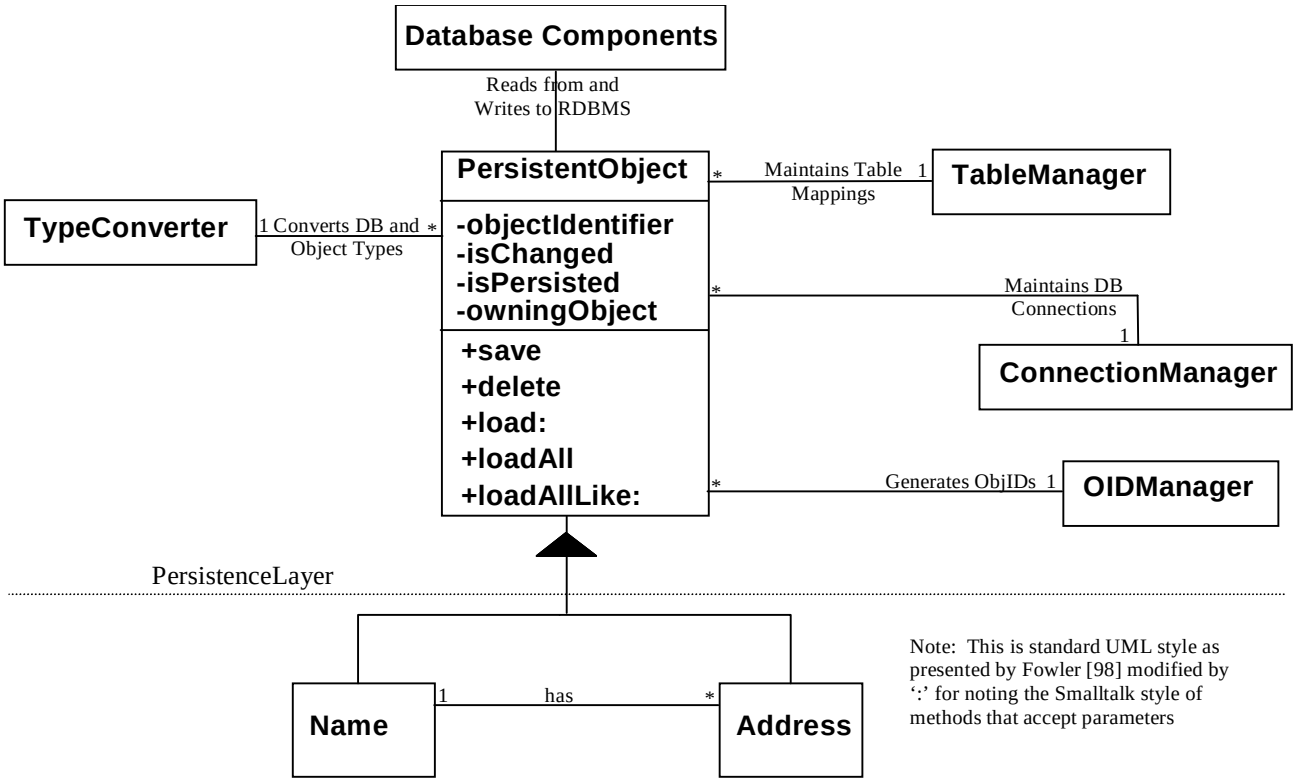


**Figure 1 – Persistence Class Diagram**

Records can be read in three ways:

- a single row (`PersistentObject>>load:`),
- all records (`PersistentObject>>loadAll`),
- or all that match a specific criteria (`PersistentObject>>loadAllLike:`).

The specific criteria is provided by creating a new instance of the object to load with the attributes set needed to find a match. This functions with the `PersistentObject>>load:` and `PersistentObject>>loadAllLike:` methods. The `PersistentObject>>loadAll` method is useful for retrieving reference data from tables where you want to populate a selection list or drop down list.

The following is example code from the `PersistentObject` described above. These are the public interface methods with transaction management (described later in this paper). The `read:` and the `saveAsTransaction` methods are described in more detail in the *CRUD* pattern.

***Protocol for Public Interface*** `PersistentObject (instance)`

**`load`**

```
"Answer a single instance of a subclasse of PersistentObjects
that matchs self.”

| oc |
oc := self loadAllLike.
^oc isEmpty ifTrue: [nil] ifFalse: [oc first]
```

**loadAllLike**

```
"Answer a collection of subclasses of PersistentObjects that
match self. The selectionClause method in the Domain object is
called to prepare the WHERE clause for the read method in the
PersistentObject"

^self class read: ( self selectionClause )
```

**save**

```
“Saves self to the database wrapped in a transaction.”

self class beginTransaction.
self saveAsTransaction.
self class endTransaction.
```

**delete**

```
“Deletes self from the database wrapped in a transaction.”

self class beginTransaction.
self deleteAsTransaction.
self class endTransaction.
```

***Protocol for Public Interface*** PersistentObject (class)

**loadAll**

```
"Answer a collection of ALL my instances from the database."

^self read: nil.
```

The following is example code from the Name class. Names have Addresses and therefore it will have a component that could need to be saved. This method is overwritten for any domain objects that have persistent components that need to be saved.

***Protocol for Private Interface*** Name (instance)

**saveComponentIfDirty**

```
"Verifies existence of the address object and also verifies a
proxy pattern is not holding the position.  The address owning
object is set to the current object and then the address object
is saved as part of the current transaction."

(self address isNil or: [self address isKindOf:
                                          PPLAbstractProxy])
      ifTrue: [^nil].
self address owningObject: self objectIdentifier.
self address saveAsTransaction
```

## *Consequences:*

✓ Another important benefit of isolating the application developer from the details of how an object saves itself is that it makes it easier to implement domain objects. Thus, it is less work to evolve the

domain model. By encapsulating the functionality of the object persistence mechanism, a developer will effectively be hidden from the details of saving the object.

✓ Database technologies can change without affecting your application code.

✓ Changing the way an object is stored in a database can be easy since we have isolated where the changes need to be made.

✓ User needs to only call the save method to persist the object. The developer does not need to detect whether or not the record already exists in the database.

✗ A *Persistent Layer* can make it complicated and sometimes difficult to do operations that might be easy to write in *SQL Code*.

✗ Optimizations can be difficult in a *Persistent Layer*. A programmer should run benchmarks for different approaches seeing what works the best for their implementations.

## *Related or Interacting Patterns:*

- *Relational Database Access Layers* [Keller 97-2] is a similar pattern in that it describes a layer in which your objects that need to be persisted communicates with.
- Layered Architecture [Shaw 96] describes the patterns necessary to layer architecture which isolates the development from changes.
- *Layered Architecture for Information Systems* [Fowler 97-1] discusses implementation details that can be applied when developing layered systems.
- Layers [BMRSS 96] describe details that need to be considered during the architecture and design of a layered system.

## *Known Uses:*

- The GemStone OODBMS uses a *Persistence Layer* to hide the fact that a value is a persistent object. Proxies use the Persistent Layer to fetch values when they are actually needed. In this case, the storage system is not a RDBMS.
- The Caterpillar/NCSA Financial Model Framework [Yoder 97] uses a *Persistence Layer* which all of the values are saved through a *Query Object* [Brant & Yoder 96]. In this case, the applications are not storing arbitrary domain objects in the database, but only retrieve transactions. However, the Persistence Layer is still used to hide details of the database and the RDBMS technology.
- ObjectShare's VisualWorks Smalltalk ObjectLens [OS 95] uses a *Persistent Layer* to map data objects to and from database tables.
- VisualAge for Smalltalk also uses a *Persistence Layer* with their `AbtDbm*` applications. VisualAge provides GUI builders with graphical connections to provide a persistence mechanism.
- The Illinois Department of Public Health's TOTS and NewBorn Screening projects use an implementation very similar to the one presented in these examples.
- TopLink, MicroDoc, Sparky, and Object Extender [MicroDoc 98, Sparky 98, OE 98] each provide a *Persistence Layer* for mapping objects to relational databases.
- The PLoP registration system implements a *Persistent Layer* for saving Java objects to the PostGress database. [ JOE PUT THE REF HERE].

# CRUD

## *Also Known As:*

Create, Read, Update, & Delete
Basic Persistence Operations

## *Motivation:*

Suppose you have a Patient class with components of class Name and Address. When you read a Patient, you must also read a Name and Address. Writing out a Patient will probably cause you to write a Name and Address objects to the database. Should they all have the same interface for reading and writing? Maybe some object requires a different interface to the database? Can we give them all the same interface? If so, what should it be?

Any persisted object needs operations to read from and write to the database. Values being persisted may also be for newly created objects. Sometimes, objects may need to be deleted from the persistence storage. Therefore, anytime an object needs to be persistent, it is important to provide, at a minimum, create, read, update, and delete operations.

## *Problem:*

What minimal operations are needed for a persistent object?

## *Forces:*

- All objects that are saved to a database need to have a mechanism for loading themselves and saving themselves.
- Having the code for reading and writing in one place makes it easier to evolve and maintain objects.
- It is easier to implement a set of nested classes if each of them has only to implement the same small interface.

## *Solution:*

*Provide the basic CRUD (create, read, update, and delete) operations for persistent objects.* Other operations that may be needed are `loadAllLike:` or `loadAll`. The important point is to provide at least enough to instantiate objects from a database and store newly created or changed objects.

If all domain objects have a common `PersistentObject` superclass, then this class can define the *CRUD* operations and all domain objects can inherit them. Subclasses can override them if necessary to increase performance.

If the *Persistence Layer* is implemented using *Brokers*, then the *Brokers* implement the *CRUD* operations. In either case, the *Persistence Layer* must generate the SQL code to read or write the domain object. Thus, each domain objects must make available a description of the SQL code necessary for accessing the database available for the *CRUD* operations. Thus *CRUD* works closely with a *SQL Code Description* for insuring that the operations are sufficient for persisting domain objects.

## *Example Implementation:*

The `PersistentObject` described above provides the standard interface to the basic set of operations for mapping the objects to the database; save, load, etc. These methods are inherited from the `PersistentObject` which access the *CRUD* operations. Some of these *CRUD* methods may need to be overwritten by the domain object. The `executeSql:` method is provided by the `AbtDBM*` database components in which it takes any SQL statement and returns back values from the database. The `updateRowSql` and the `insertRowSql` are described in the *SQL Code Description* pattern below.

### ***Protocol for CRUD*** PersistentObject (class)

This method takes a WHERE clause as an agent, and returns a collectin of objects that correspond to the rows that matches the WHERE clause.

**read: aSearchString**

```
	"Returns a collection of instances populated from the database."
	| aCollection |
	aCollection := OrderedCollection new.
	(self resultSet: aSearchString)
		do: [:aRow | aCollection add: (self new initialize: aRow)].
	^aCollection
```

### ***Protocol for Persistence Layer*** PersistentObject (instance)

These methods save or delete objects from the database. These methods make the decisions of what kind of SQL statements (insert, update, or delete) based on the object's values. Once the decisions have been made the SQL statement is then fired to the database.

**saveAsTransaction**

```
	"Save self to the database."

	self isPersisted ifTrue: [self update] ifFalse: [self create].
	self makeClean
```

**update**

```
	"Updates aggregate classes then updates self to the database"

	self saveComponentIfDirty.
	self basicUpdate
```

**create**

```
	"Inserts aggregate classes then inserts self to the database."

	self saveComponentIfDirty.
	self basicCreate
```

**basicCreate**

```
	"Fires the insert SQL statement to the database"
	self class executeSql: self insertRowSql.
	isPersisted := true
```

**basicUpdate**

```
	"Fires the update SQL statement to the database."
	(self isKindOf: AbstractProxy) ifTrue: [^nil].
	isChanged ifTrue: [self class executeSql: self updateRowSql]
```

**deleteAsTransaction**

```
	"Delete self from the database.."

	self isPersisted ifTrue: [self basicDelete].
	^nil
```

```
basicDelete
      "Fires the delete SQL statement to the database."
      self class
      executeSql:('DELETE FROM ',self class table,' WHERE ID_OBJ=',
                              (self objectIdentifier printString)).
```

### *Consequences:*

✓ Once your object model and your data model have been analyzed, the results can be implemented in *CRUD* providing a performance-optimized solution, thus isolating the developer from having to worry about performance details. ***** If your object model and your datamodel are analyzed, you can implement your *CRUD* operations that will provide for optimal performance for that database, hiding the details from the application developer.

✓ Flexibility to retrieve data based on how many rows or what kind of data (dynamic, static, or somewhere in-between) is needed.

✓ Simple implementation to SAVE the data to, or back to, the database. The application developer does not have to determine whether to insert or update the object.

✗ *CRUD* can cause sub-optimal performance if the object model and data model have not been properly analyzed. This will make the job of the developer more difficult if they have to compensate another way.

### *Related or Interacting Patterns:*

- *Transaction Manager* handles the transactions for these operations.
- *CRUD* interacts with *SQL Code* for generating the necessary database calls.

### *Known Uses:*

- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- ObjectShare's VisualWorks Smalltalk ObjectLens [OS 95] uses a *CRUD* in order define how to manipulate simple data objects. VisualAge Smalltalk [VA 98] also uses a *CRUD* with their `AbtDbm*` applications.

# SQL Code Description

## *Also Known As:*

Definition of code to query, update, insert, and delete
Object Query Language (OQL) description
Common Query Language (CQL) description
Structured Query Language (SQL) description

## *Motivation:*

Somewhere the SQL Code that reads, updates, inserts, and deletes values from the database has to be written for keeping your object values consistent with their persistent storage. Once again look at a Patient class with components of class Name and Address. The SQL code needs to be stored for reading and writing values for the Patient. You must also store the SQL for Name and Address. On one hand you can hard code the SQL for reading and writing to the database. You could store the values in a common place. A schema map could also be developed for storing the representation of the object mappings to the database and dynamically be generated at runtime.

## *Problem:*

Where do you store the actual descriptions for generating the SQL statements necessary to perform the *CRUD* operations?

## *Forces:*

- When accessing a relational database, SQL Code for the database access has to be supplied from somewhere.
- The amount of SQL Code grows as the domain model grows.
- Writing efficient SQL Code requires you to understand a lot about the data model and database.
- The domain model will probably change frequently over the life of an application.
- SQL Code could be put wherever database access is needed.
- Duplication of similar SQL Code can create maintenance problems.
- Generating SQL Code from metadata can hide details of how an object accesses the framework from the developer but can have performance and maintenance tradeoffs.

## *Solution:*

*Provide a place where the developer describes the SQL Code for maintaining the consistency between his object and the persistent storage.* Minimally, business objects need to know how to perform *CRUD* operations (create, read, update, and delete). Somewhere, the necessary SQL Code to perform these *CRUD* operations needs to be defined.

It is important to maintain the consistency between the values from objects and the persistent storage. It is also important to provide a means where by an embattled programmer is less likely to forget to update a SQL statement when a domain object is modified.

This pattern can be implemented many ways, but the key idea is that the SQL statement is encapsulated and easy to associate with the persistent object. Thus, each object has the necessary SQL Code available to interact with the database. With the SQL Code closely associated with the domain object, it is less likely that an embattled programmer will forget to update a SQL statement when a domain object is modified.

One way to do this is to actually write the complete SQL Code necessary for each of the operations. Then let the *Persistence Layer* read the SQL Code from the domain object, create the database connection, and make the database call. Both Brokers and Object can generate the SQL code from the domain object.

Another way to do this is to provide an Object Query Language (OQL) that is a description of the necessary operations needed for the *CRUD* operations. The OQL is then interpreted for building up the necessary calls to the database.

Another alternative is to use *Metadata* [Foote & Yoder 1998] to describe the *CRUD* operations. A *CRUD* operation interprets metadata to build the appropriate SQL. Most commercial frameworks use this approach. It would build up this structure from some sort of *Schema Map* [Foote & Yoder 1998]. Most of these commercial frameworks provide a visual language for building and manipulating these queries. Using *Metadata* and *Schema Maps* can become complicated to implement and maintain. They can also generate non-optimal queries. However, they can make it easy for the programmer to describe the mappings between the domain object and the database; specifically when a visual language has been developed to assist with the process.

If you are implementing on a larger scale where you have thousands of lines of SQL code and the dynamic SQL isn't fast enough, you would want to replace or modify the SQL example to call stored procedures, pre-compiled SQL, implement a push-pull technology, or cache. This is called the *Optimizing Query* pattern [Keller 97-2].

### *Example Implementation:*

When you are using a database, you will have to produce SQL statements for retrieving, inserting, updating, and deleting records. These statements in their simplest form:

```
SELECT * FROM table_name.
INSERT INTO table_name ( column_name ) VALUES ( values ).
UPDATE table_name SET column_name = xyz WHERE key_value.
DELETE FROM table_name.
```

These seem simple enough, but how do you get the values from an object into the statement.

You can use a stream to place the "fixed" portions of the statement and the attributes from the object onto or you can define the columns you want to retrieve.

```
aStream nextPutAll: 'SELECT';
      nextPutAll:  column_names ;
      nextPutAll:  'FROM ';
      nextPutAll:  table_name.
```

or:

```
aStream nextPutAll: 'INSERT INTO ';
      nextPutAll: table_name;
      nextPutAll: (column_names);
      nextPutAll: 'VALUES ';
      nextPutAll: (values).
```

Below are examples from the `Name` class instance methods. The `Name` class is a domain object in our example. Names and addresses are common in most, if not all, administrative type applications. This is a simple example to demonstrate the *Persistence Layer* with a very small complex object. As stated above the SQL statements do not have to be hard coded. If your database and your object model are very similar then they could be generated from a schema.

### ***Protocol for SQLCODE*** (instance)

These methods provide the *Persistence Layer* with the actual SQL statement that is to be sent to the database. The SQL statement is built on a stream and then passed to the *Persistence Layer* for execution. This example shows the use of the *Type Converter* and *Table Manager* patterns described later in this document.

These methods build the actual SQL statement that is going to be sent to the database. A write stream is used instead of concatenation for performance reasons.

**insertRowSql**

```
"Returns a sql insert statement with values from the object."

| aStream |
aStream := WriteStream on:(String new).
aStream nextPutAll: 'INSERT INTO ';
      nextPutAll: self class table;
      nextPutAll:' (ID_OBJ,
            ID_OBJ_OWN,
            NAM_FST,
            NAM_LST,
            NAM_MID,
            EML_ADR,
            ORG_NAM,
            NUM_PHO)
            VALUES (';
      nextPutAll: (self typeConverter prepForSql:
            (objectIdentifier:= (self getKeyValue)));
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:self
            owningObject);
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:(self first
            asUppercase));
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:(self last
            asUppercase));
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:(self middle
            asUppercase));
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:(self email));
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:(self
            organization));
      nextPut: $,;
      nextPutAll: (self typeConverter prepForSql:(self phone));
      nextPutAll: ')'.
^aStream contents.
```

**updateRowSql**

```
"Returns the update sql statement with values from the object.”

| aStream |
aStream := WriteStream on:(String new).
aStream nextPutAll: 'UPDATE ';
        nextPutAll: self class table;
        nextPutAll: ' SET NAM_FST=';
        nextPutAll: (self typeConverter prepForSql:(self first
                  asUppercase));
        nextPutAll: ', NAM_LST=';
        nextPutAll: (self typeConverter prepForSql:(self last
                  asUppercase));
        nextPutAll: ', NAM_MID=';
        nextPutAll: (self typeConverter prepForSql:(self middle
                  asUppercase));
```

```
            nextPutAll: ', EML_ADR=';
            nextPutAll: (self typeConverter prepForSql:(self email));
            nextPutAll: ', ORG_NAM=';
            nextPutAll: (self typeConverter prepForSql:(self
                      organization));
            nextPutAll: ', NUM_PHO=';
            nextPutAll: (self typeConverter prepForSql:(self phone));
            nextPutAll: ' WHERE ID_OBJ=';
            nextPutAll: (self typeConverter prepForSql:self
                      objectIdentifier).
      ^aStream contents.
```

This method provides the `where` condition for SQL `select` statement. Each class has this method to determine what can be used in the `where` clause of the `select` statement.

**selectionClause**

```
      "Answer a string representation of a where clause."

      | aStream app|
      aStream:= WriteStream on:(String new).
      ( self objectIdentifier isNil )
      ifFalse: [ aStream nextPutAll: 'ID_OBJ=';
                 nextPutAll: (self class typeConverter prepForSql:
                                self objectIdentifier).
                 ^aStream contents ].
      ( self owningObject isNil )
      ifFalse: [ aStream nextPutAll: 'ID_OBJ_OWN= ';
           nextPutAll: (self typeConverter prepForSql: self owningObject)].
      ^aStream contents.
```

### ***Protocol for SQLCODE*** (class)

These methods provide table name for the above methods and define which columns will be returned from the database with the where clause produced above.

**table**

```
      "Returns the table name from the Table Manager."

      ^TableManager getTable: 'EXAMPLE'
```

**buildSqlStatement: aString**

```
       "Returns the read SQL statement for the object"


      | aStream |
      aStream := WriteStream on:(String new).
      aStream nextPutAll: 'SELECT
                              ID_OBJ,
                              ID_OBJ_OWN,
                              NAM_FST,
                              NAM_LST,
                              NAM_MID,
                              EML_ADR,
                              ORG_NAM,
                              NUM_PHO FROM ';
            nextPutAll: self table.
```

```
((aString isNil) or:[ aString trimBlanks isEmpty])
      ifFalse:[aStream setToEnd;
            nextPutAll: ' WHERE ';
                  nextPutAll: aString].
^aStream contents.
```

### ***Protocol for SQL Code*** PersistentObject (instance)

Each object is required to provide the *SQL Code Desciption* to the *Persistence Layer*. If the object does not require part of the SQL code then it should return "`shouldNotImplement`".

**`insertRowSql`**
```
      ^self subclassResponsibility
```

**`selectionClause`**
```
      ^self subclassResponsibility
```

**`updateRowSql`**
```
      ^self subclassResponsibility
```

### *Consequences:*

✓ Flexibility, only the required set is returned. Also the SQL statement could be replaced with calls for various forms of stored or compiles queries.

✓ The performance of the SQL statement is easily determined with DB tools.

### *Related or Interacting Patterns:*

- *SQL Code Description* uses an *Interpreter* [GHJV 95] for producing the language for the database.
- *SQL Code Description* uses the *Builder* [GHJV 95] pattern to provide the same process for different objects.
- *SQL Code Description* could use *Metadata* [Foote & Yoder 98] to produce the SQL statement.
- *SQL Code Description* needs to know the *Schema* [Foote & Yoder 98] in order to produce the statement correctly.
- *SQL Code Description* is the code generated for the *CRUD* operations in the *Persistence Layer*.
- *SQL Code Description* is generated with values from the *Attribue Mapping Methods*.

### *Known Uses:*

- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- ObjectShare's VisualWorks Smalltalk [OS 95] uses a *SQL Code Description* in order to define how to perform *CRUD* operations for simple data objects. VisualAge Smalltalk also uses a *SQL Code Desciption* with their `AbtDbm*` applications.
- In the GemStone GemConnect [GemConn 98], *SQL Code Description* is used for getting values to and from a relational database into objects.

## Attribute Mapping Methods

### *Also Known As:*

Mapping DB Values to Objects
Mapping Objects to DB Values

### *Motivation:*

When a record is brought in from the database, each value returned from a database column must be mapped to an attribute or a set of attributes in the object. Similarly, when saving values back to the database, an object's attributes must some how be mapped to database column(s). Consider the Patient example. A `Patient` could have the patient's name and sex associated with it. This could be read in from a patient table in the database. Also, `Patient's` could have an address associated with them. These values might have foreign keys that reference other objects such as `Address`. Thus when reading in a `Patient` object, the attribute mapping would have to map the column holding the name and sex values with the `Patient's name` and `sex` attributes respectively. Also, an `Address` object needs to be created and mapped to the `Patient's address` attribute.

### *Problem:*

Where and how does the developer describe the mappings between database values and attributes?

### *Forces:*

- Objects store values in attribute variables while databases store their values in columns.
- There is not always a one-to-one mapping from an object's attributes to the columns in a database table.
- Some objects require their values from multiple databases, multiple database tables and/or are complex objects.
- Non object-oriented databases do not always represent hierarchies or object types very well.

### *Solution:*

*For every domain object that needs to be persistent, write a method that maps the database values to the object's attributes and write a method that maps the values from the object back to the database.* The *Persistent Layer* will use the first method to take the returned values from the database and store them in the appropriate object attribute. Similarly, when the `PersistentObject` is being saved, the *Persistent Layer* will use the second method for taking values from the object and putting them out to the database. The `PersistentObject` maps data while it is generating the SQL code.

These methods take a returned row from the database and fill in the appropriate object attributes. Sometimes a few columns may map to a single attribute. These methods must also take an object's attribute values and map them to the appropriate database columns during a database write routine. Usually a single attribute maps to one or more database column but sometimes multiple attributes may map to a single database column. In some cases, attributes may have to be populated from different databases on different platforms. In this case, the aggregate class would load from the other database and return the object to be assigned to the current attribute.

There are usually at least two sets of *Attribute Mapping Methods*. One set is used for reading values from the database and the other set is used for writing values back to the database. While values are being mapped to and from the database, *Attribute Mapping Methods* need to provide calls to the *Type Conversions*.

*Metadata* could be used either with or without a *Schema* to define the attribute mappings. In this case, an *Interpreter* would be used to generate the attribute mappings. A visual language could also be provided for

describing the mappings. This solution is usually harder to implement and maintain. However, once implemented, it could be much easier for a programmer to map an object's attributes.

When an attribute is being mapped to another domain object, *Proxies* are often used for lazy initialization. Take the `Patient` example mentioned above. Since you may seldom need the address information for a patient, a *proxy* could be initialized for the `address` attribute. Then, whenever the patient's address needs to be accessed, the patient's address information can be read in from the database and an `Address` object can be created. In this case, the `address` attribute would also be updated to point to the newly created `Address` object.

### *Example Implementation:*

Once the data has been retrieved from the database it needs to be moved from the returned row into the objects attributes. The returned row (in VisualAge) is a dictionary so it can be accessed by:

```
aRow at: keyValue.
```

Next, you would assign the value to the attribute:

```
attribute := (aRow at: keyValue).
```

Or, if the attribute is to contain an instance of another class:

```
attribute := ((Class new) owningObject:
                                objectIdentifier; youself) load.
```

This will load an instance of the `Address` class from the database and assign it to the attribute. If the database containing the address information was in a different database or on a different platform, the `Address` class would load from the alternate database.

Below are the *Attribute Mapping Methods* from the `Name` class that shows how to map from the database row to the object and apply *Type Conversion* in the process. The *SQL Code Description* pattern shows how to map from the object back to the database.

### ***Protocol for Map Attributes*** (instance)

These methods accept `aRow` (a single record) from the `PersistentObject>>read:` method. Each row from the table is passed to a new instance (initialized by the `PersistentObject`) and each element from the row has *Type Conversion* applied as necessary before assignment to the attribute takes place. In the case of a complex object, the attribute is assigned the value returned from sending a `PersistentObject>>load:` (`PersistentObject>>loadAllLike:` if a collection is needed) to the attribute class type with an instance for the *SQL Code Description* to provide the conditioned SQL statement. For example, in the `Name` class, you would send the `loadAllLike:` message to the `Address` class with the `objectIdentifier` as a parameter. This would load all `Addresses` that have the `owningObject` identitifier the same as the `Name` class.

**`initialize: aRow (Name class)`**

```
"Initializes an instance from the database row."

objectIdentifier := self typeConverter convertToNumber:
       (aRow at: 'ID_OBJ').
owningObject := aRow at: 'ID_OBJ_OWN'.
isPersisted := true.
first := self typeConverter convertToUpperString: (aRow at:
      'NAM_FST').
middle := self typeConverter convertToUpperString: (aRow
      at: 'NAM_MID').
last := self typeConverter convertToUpperString: (aRow at:
      'NAM_LST').
email := self typeConverter convertToString: (aRow at:
      'EML_ADR').
```

```
organization := self typeConverter convertToUpperString:
      (aRow at: 'ORG_NAM').
phone := self typeConverter convertToNumber: (aRow at:
      'NUM_PHO').
address := ((Address new)
      owningObject: objectIdentifier;yourself) load
```

### *Consequences:*

✓ There is a single point of change when the DB structures changes.

✓ The attribute name and the table column name do not have to be the same.

✓ Developers new to the project can easily cross match table column names to attribute names.

✗ This attribute mapping method needs to be maintained as the database and domain object evolves.

### *Related or Interacting Patterns:*

- *Type Conversion* needs to be done while mapping values to and from the database.
- When a call to the *Persistence Layer* is made to read or write values to the database, SQL code will be generated which will need to have *Attribute Mapping Methods*.
- *Metadata* could be used with a *Schema* for defining the attribute mappings. This would probably use an *Interpreter* to generate the actual mappings.

### *Known Uses:*

- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- ParcPlace's VisualWorks Smalltalk [OS 95] uses *Attribute Mapping Methods* in order define how to perform attributes map to and from data objects. VisualAge Smalltalk [VA 98] also uses *Attribute Mapping Methods* with their `AbtDbm*` applications.
- The `ResultSet` class in JDBC is a row and methods such as `getInt(), getString(), getDate()` are those work as attribute mappings and type conversion for getters. For mapping back to the database in JDBC, the `PreparedStatement` class builds up the SQL statement with '?" as placeholders for parameters. These are set with `setInt(), setString(), setDate()` and so forth.
- GemStone GemConnect maps attributes from database columns to object attributes.

# Type Conversion

## *Also Known As:*

Data conversion
Type translation

## *Motivation:*

The values in a database do not always map directly into object types. For example, a boolean value may be saved as a "T" or "F" in a database. In the `Patient` example, the sex could be stored as an attribute in which there is a class called `Sex`, which has certain behavior associated with male instances and different behavior with female instances. The database might store the values as "M" and "F". When the values are read in from the database, the "M" would need to be converted to a male instance of the `Sex` class and the "F" would need to be converted to a female instance of the `Sex` class. *Type Conversion* allows for object values to be converted to and from database values.

## *Problem:*

How do we take objects that may not have a database type and allow for them to map to a database type and vice-versa?

## *Forces:*

- Values represented in a database may not map to the desired object type.
- There is an impedance mismatch from object types to database types.
- Values used in an application may be edited and stored in a database from many other applications.
- Object attributes and database column values could all be stored as strings or numbers, thus minimizing the impedance mismatch.

## *Solution:*

*Have all values convert their respective types through a Type Conversion object. This object knows how to handle nils and other mappings of objects to and from database values.* When objects are persisted from large multi-application database the data formats can vary. This pattern ensures the data retrieved from the database is appropriate for the object.

It is important that we ensure the data read from the database will work with our object. It is also important that the data written to the database will comply with the database's rules and maintain data integrity.

Each object attribute is passed through the appropriate *Type Converter* to apply the data rules necessary for proper application or DBM use. Applying the data rules at the domain level is much more efficient that at the interface level. This is usually because all domain objects will follow a common set of *Type Conversions*.

When considering that some legacy databases do not understand the concept of nil (NULL), they could actually save a blank string for a "No Data" condition. When the value is read back it would result in an empty string which could mean "Blank Data" to the application. It is dependent upon the database and the application to what rules are defined.

This *Type Conversion* can be done directly in the mapping code. The data types will be converted to appropriate object types or vice-versa. Quite often, a common set of conversions can be abstracted out. For example, a `Boolean` object might map to 'T' or 'F' in the database, `Timestamps` could map to `Strings`, and `NULLS` might map to an empty string. When you have these types of common conversions, then an appropriate routine can be called to preprocess the types for conversion. These conversion routines could become part of a *Strategy* [GHJV 95].

Implementations can vary dependent on your needs. The example that follows places all the conversion methods in a single object. This keeps the methods localized to one area and allows the conversion object to be dynamically switched, if necessary. Thus, you could have a *Strategy* that applies different conversion algorithms for different databases. If the converters are not needed by the rest of the application this is a "cleaner" approach.

Another option would be to extend each base class affected or used and place the methods there. Each object would know how to convert itself into a format necessary for the database. If you were using multiple databases then each of these methods would have to accommodate the differences in the formats.

Yet another approach would be to put all of your conversion routines in `PersistentObject` thus isolating the place where you would have to change the code if you need to map to a new database or the conversion evolves. These methods would then be available to any object that inherits from `PersistentObject.` A similar situation occurs, as in the previous option, when you have multiple databases.

All of the above mentioned solutions will work regardless of the persistence mechanism chosen.

### *Example Implementation:*

When you are populating the attributes from a database where you have the database row values in a dictionary you could simply use:

```
attribute := (aRow at: key).
```

This will accomplish the task, however, if the database value is not guaranteed to meet the object's need then your code might fail during use of the attribute. When you apply Type Conversion, such as:

```
attribute := self typeConverter convertToUpperString:
                                                (aRow at: key).
```

The attribute value will be what the application needs. This will first retrieve the class responsible for conversion and then pass the database value to the method to ensure the attribute is populated with an upper case string.

A similar scenario occurs when preparing the object's attribute values to be stored in a database. You could simply place the attribute value on a stream.

```
nextPutAll: (attribute) printString.
```

This also accomplished the task, however, if (self first) happens to be an object that does not understand `printString` then the code will fail. Should the database require NULL for nil or an empty string for nil then every statement would a conditional statement. When you apply Type Conversion, such as:

```
nextPutAll: (self typeConverter prepForSql: (attribute
                                                asUppercase)).
```

The attribute will be converted to the proper format. As above, the class responsible for conversion will be retrieved and the attribute will be converted.

*Type Conversion* methods are accessed through the `PersistentObject` as the values are either being prepared for being saved or when the returned database rows are being mapped to an object's attributes. The `initialize:` and `insertRowSql`: methods (in each domain object) show examples of Type Conversion.

### ***Protocol for Type Conversion*** `PersistentObject (instance)`

This method determines which class is responsible for converting (translating) type between table types and object types. This method could be expanded to determine, at run time, which database is in use or which class to use for the conversions.

**typeConverter**

```
"Returns the class responsible for Type Conversion”

^TypeConverter
```

***Protocol for Type Conversion*** TypeConverter (class)

These methods provide the `PersistentObject` with consistently formatted data values. When converting from the object to the database (`TypeConverter>>prepForSql:`), the type can be determined and formatted based on what the database requires. When converting values from the database to an object, the type in the database may not be the type the object/application wants. In the *Mapping Attributes Methods*, each attribute is *Type Converted* to ensure the data values are correct. In some cases a default value is provided for when the database does not contain data. When several applications use the same database for diverse implementations, the data rules may not be able to be enforced at the database level.

**convertToBooleanFalse: aString**

```
"Returns a boolean from a string character.  False is the default
 value."

^'t' = aString asString trimBlanks asLowercase
```

**convertToString: aString**

```
"Returns a string from a db string or character.  Default returns
a new string."

^aString asString trimBlanks
```

**convertToNumber: aNumber**

```
"Returns a number from a db number.  Default returns zero."

^aNumber isNumber ifTrue: [aNumber asInteger] ifFalse: [0]
```

This method converts an object into the correct database format to be placed on a stream. The object is tested for what type it is and the appropriate format is returned to be placed on the stream for the SQL Code. This provides the Persistence Layer a common format for data types. The default date format in this example is assumed to be (Locale current lcTime dFmt: '%m/%d/%Y'). If you do not want to use a full date format the date formating should be changed to match your database. Note: These formating types are supported for IBM DB2 UDB V5.0.

**prepForSql: anObject**

```
“Returns anObject in an appropriate format for a stream.”

anObject isNil ifTrue: [^'NULL'].
anObject isString
      ifTrue:
            [anObject isEmpty
                  ifTrue: [^'NULL']
                  ifFalse: [^anObject trimBlanks printString]].
anObject isNumber ifTrue: [^anObject printString].
anObject abtCanBeDate ifTrue: [^anObject printString
                                                printString].
anObject abtCanBeBoolean
      ifTrue: [anObject ifTrue: [^'T' printString]
                        ifFalse: [^'F' printString]].
```

```
anObject abtCanBeTime
      ifTrue:
            [^self databaseConnection databaseMgr
                  sQLStringForTime: anObject].
(anObject isKindOf: PPLPersistentObject)
      ifTrue:
            [anObject objectIdentifier isNil
                  ifTrue: [^'NULL']
                  ifFalse: [^anObject objectIdentifier
                                                printString]]
```

### *Consequences:*

✓ Can help insure data consistency.

✓ Object types can vary regardless of the database types.

✓ Default values can be assigned for empty values from the database.

✓ Prevents "Undefined object does not understand" errors.

✓ Provides increased RMA for the applications. (Reliability, Maintainability, and Accessibility).

✗ It can be time consuming to convert types, especially when reading in many values from a database.

### *Related or Interacting Patterns:*

- *Strategy* [GHJV 95] could be used to implement this pattern.
- *Attribute Mapping Methods* calls *Type Conversion* while the SQL code is being built.
- *SQL Code Description* might embed calls for *Type Conversion.*

### *Known Uses:*

- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- ObjectShare's VisualWorks Smalltalk [OS 95] uses *Type Conversion* in convert to and from database and object types. VisualAge for Smalltalk also uses *Type Conversion* with their `AbtDbm*` applications.
- GemStone GemConnect converts between database types and object types.
- TopLink also provides for *Type Conversion.*

# Change Manager

## *Also Known As:*

HasChanged
IsDirty
Laundry List
Object Manager

## *Motivation:*

The usual way to use a patient management system is to call up the record for the patient and add a record for the latest visit. But sometimes the patient's address has changed. The system should only write back the patient's address if it has actually changed.

One way to implement this is to provide a separate screen for changing the address with a separate update button that causes the address to be written back. This is awkward and requires a lot of code maintenance. It would be better to let the user of the patient management system edit the address when necessary and for the system to only write back changes to the address object and other values of a patient when the user decides.

In general, a *Persistence Layer* should keep track of all `PersistentObjects` that have changed state, and should make sure that they are all written back to the database.

## *Problem:*

How to tell that an object has changed and needs to be saved to the database? It is important to prevent unnecessary access to the database and to ensure that the user knows when some values has changed and not saved before they exit the program?

## *Forces:*

- Most objects that are read never get modified.
- It is a waste of time to save unchanged objects.
- Developers often forget to save an object when it changes. Users are worse.
- If a write to the database is made every time an object changes, then another write or rollback would be required for cancellation requests by the user.
- Complex objects might only have one of their components changed, thus not requiring a complete write of all values to the database.
- Writing back objects that have not changed can make it hard to audit who actually last changed the object.

## *Solution:*

*Set up a Change Manager that keeps track of any* `PersistentObject` *that changes one of its persistent attributes.* This *Change Manager* will be used whenever any requests to saving objects is needed.

One way to do this is by inheriting from a `PersistentObject`, which has a dirty bit that gets set whenever one of the attributes that maps to the database changes. This dirty bit is usually an instance variable with a boolean value which indicates when an objects values have changed. When the boolean value is set, the `PersistentObject` will save the new values to the database when a request to save it is invoked. If the boolean value is not set, the `PersistentObject` will bypass the write to the database.

*Laundry List* is a pattern that can also be used for persisting data. This works by storing the changes in a laundry basket. Then you can control what you want to do with them. Another thing it to keep track of which attributes have changed and then only save back the dirty attributes. Thus if one application changes

a few of the columns of the tables and another application changes the others, you can be sure to only update what you are changing. It can also help with concurrent applications changing the table.

Dependent upon the class hierarchy the implementation can vary. One solution is to modify the setter methods to either set the flag whenever an object's values are changed (as in the example) or by adding the object to the laundry list of changed objects. This is a very simple but effective solution. Another alternative would be to use something similar to method wrappers [Brant, Foote, Johnson, & Roberts 1998] for doing the same during any set of the method. Yet another alternative would be to initialize persistent attributes as a dependency mechanism. Once an attribute value changes, either set the dirty bit or add the object to the laundry list and then remove the dependency; once dirty, always dirty. Another way to implement this could be by using *Metadata* that describes all persistent attributes. Whenever the state of any object changes it can use the *Metadata* to decide if the object has become dirty or not. Method wrappers could use this *Metadata* to provide a similar service.

Data access is generally very expensive and should be used sparingly. By being able to identify that an object does or does not need to be written to the database can increase performance drastically. In addition to performance considerations, the user interface can benefit by being able to prompt the user to save before exiting. This ability adds to the user accepting your application by knowing that if they forget to save the changed information, the system will prompt them to save it. The user can come to the conclusion that the application is useless very quickly if they have to reenter the same data more than once.

Another feature that the *Change Manager* can provide is that of remembering original state or changed state of an object. This could be implemented by using a *Memento* [GHJV 95]. If the user of your system needs to be able to revert back to original state, the *Change Manager* could keep around the original values so that an undo call could be invoked. Also, you might want to provide your system with the ability to have multiple undos.

If you have an object with one of its attributes as an ordered collection of other objects, (in the example the `Name` attribute address could be an `OrderedCollection` of `Addresses`) and you remove one of the instances of `Address`, how does the database get notified to delete the row? One way is to set key values of the instance to `nil`. Then the domain object provides an `isValid` method that will then delete the row. This works but the application programmer has to handle all the `nil` instances with specialized code. A better approach is to use a *Deletion Manager* that the application programmer will set the instance to when the user presses the delete button (or other mechanism). Because each object knows how to delete itself, the instance is removed from the collection and from the database. The *Deletion Manager* is a *Singleton* [GHJV 95] that holds on to the deleted instances until a save or cancel is issued by the user.

### *Example Implementation:*

The example will show how an accessor method for the first name attribute in the `Name` class sets the dirty bit whenever the attributes changes its value. The accessor is the default generated by VisualAge with the addition of the `makeDirty` call, which sets the inherited attribute of `isChanged` to be true.

```
first: aString

      "Save the value of first."

      self makeDirty.
      first := aString.
      self signalEvent: #first
            with: aString.
```

### *Protocol for Change Manager* `PersistentObject (instance)`

These methods provide the *Persistence Layer* with the ability to change the dirty flag. This saves the *Persistence Layer* from having to write data to the database that has not changed. It also provides the GUI programmer a way to test the object in order to provide the user a message to save their data or not.

**makeDirty**

```
"Indicates an object needs to be saved to the db.  This method
can be called from the 'setter' method as in the example above."

isChanged:=true.
```

**makeClean**

```
"Indicates that the object does not need to be saved to the db or
 that no changes have affected the  object"

isChanged:=false.
```

### *Consequences:*

✓ The user will more readily accept the application.

✓ Better performance by not writing object that has not changed back to the database.

✓ When merging data between databases the flag can be set so the record will be inserted to the new database as necessary.

### *Related or Interacting Patterns:*

- *State* [GHJV 95] is an alternative to using boolean flags.
- *Mementos* could be used for undo support by the *Change Manager.*
- *Launcry Lists* can keep track of all changed objects.
- A *Deletion Manager* could be used for assisting with the deletion of parts of a complex object.
- *ObjectManager* [Keller 98-2] is a very similar pattern.

### *Known Uses:*

- Operating Systems uses the dirty bit for virtual memory.
- Most DBMS's use a dirty bit with cache [Keller 98-1].
- Caches are common to using dirty bits.
- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- In the GemStone OODBMS a *Change Manager* is used for keeping track of when an object has changed thus knowing when the object needs to be saved back to the server.
- ObjectShare's VisualWorks Smalltalk [OS 95] uses a *Change Manager* in order to designate when a persistent object has changed its values. VisualAge Smalltalk also uses a *Change Manager* with their `AbtDbm*` applications. VisualAge provides GUI builders with graphical connections to provide a persistence mechanism.

## OID Manager

### *Also Known As:*

Unique Key Generator

### *Motivation:*

Whenever an object is persisted, it is important to note the uniqueness of an object. All objects are unique in an object-oriented system so it is important to give an object a unique Object Identifier usually called OID. The *OID Manager* insures that a unique key is generated for all objects that are being stored in the database.

## *Problem:*

How do we insure that each object gets stored uniquely in a database regardless if it shares similar state with another object or not?

## *Forces:*

- Changing keys and duplicates are something that you do not want to do in a DB. Database administrators consider this very bad.
- Adding ids is sometimes artificial and usually requires an extra column in the table.
- Creating a unique key with distributed databases.
- Create a unique key to positively identify a record for *SQL Code*.

## *Solution:*

*Provide an OID Manager that creates unique keys for all objects that need to be stored in the database. Insure that all newly created objects that need to be persisted get a unique key*. When a new object that needs to be persisted is to be written to the database a unique identifier is generated. The generation process needs to be quick and it needs to insure uniqueness.

A solution is to generate a random number. Once the number is generated it must check to see if it is already used. When running across multiple databases that might or might not be available locally, the time required increases. When merging data from multiple databases the keys have even a greater chance of being duplicate.

Another solution is to store the last used number in a local table and concatenate it with a key that is local and unique. This requires each write to read and write from/to the key table.

A better solution (the one used in the example) is a modification of the previous method, which alleviates the need to write each key used to the table ([Ambler 97]). When a key is needed a singleton instance is asked for a key. If the singleton instance does not have the number (i.e. first write) a table is read to get a block of numbers. When the number is returned it is immediately incremented by a specific amount (application specific) and written back to the table for the next time or another user to access. The number is returned to the calling object and incremented by 1 and stored in memory until the next time a key is needed. When the block is used up the table is read again the process repeats.

A key generating *Strategy* [GHJV 95] could be used for creating these keys. One database or site might use one algorithm while another could use another algorithm. The important thing is that the key is unique across a table and preferably across the whole database or databases.

Some databases have ways to generate unique keys. Need to make sure that if you have multiple servers, there is no clash with the generation algorithm. You could also use a TCP/IP address and/or the hard-drive serial number appended with some other number to insure uniqueness for an object identifier. Also some databases have a way to generate sequence of numbers which could be used for OIDs. No matter what algorithm you use, it is important that it is thread-safe.

***Relate this to the Unique Key patterns in Kellers and Browns pattern language.

## *Example:*

The *OID Manager* provides the *Persistence Layer* with a unique identifier for the domain object, which can be used as the database key. The *OID Manager* maintains a table that has a number in it which the *OID Manager* increments and writes back when it does not have a valid number to return to the *Persistence Layer*. This example retrieves a block of numbers from the table when it does not have a valid number and then maintains it until the number reaches the number written back to the table. The size of the block in this example is 10 (see the increment method below).

### ***Protocol for Accessors*** OIDManager (instance)

This method returns the value for the size of the block of numbers to be retrieved from the database when a new block of numbers is needed. This can be set by the applications at startup or it is not it will be set to 10.

**increment**

```
"Return the value of increment."

(increment isNil)
      ifTrue:[increment:=10].
^increment
```

### ***Protocol for Key Generation*** OIDManager (instance)

This is the core method for this pattern. A column value is read from a table which all users accessing the database use. When a value is read it is then immediately incremented by the value store in the increment attribute. This value defaults to 10. The new number is then written back to the table for the next use or user. The value is initially read is stored in attribute and manipulated as necessary to provide a unique number. The number is added to site (or database) key to create a unique 14 or 15 digit number.

**readKey**

```
"Generate a unique key value for use in storing  an object in the
database."

| newKey aQuerySpec aResultSet aMaxKey prep |
PersistenceObject beginTransaction.
aQuerySpec := AbtQuerySpec new
                  statement: ' SELECT  NUM_SEQ  FROM SEQUENCE '.
aResultSet := PersistenceObject databaseConnection
                  resultTableFromQuerySpec: aQuerySpec.
aMaxKey := aResultSet first at: 'NUM_SEQ'.
newKey := aMaxKey + self increment.
PersistenceObject
      executeSql: 'UPDATE SEQUENCE SET NUM_SEQ = ' , newKey
                    printString.
PersistenceObject endTransaction.
prep := self class siteKey * self keySize.
self lowKey: prep + aMaxKey.
self currentKey: prep + aMaxKey.
self highKey: prep + (newKey - 1).
^nil
```

### ***Protocol for Key Retriever*** OIDManager (instance)

This is the method to retrieve a key. The attributes are checked to see if a number needs to be read by the above method or just increment the attribute of this singleton instance by one and return the value.

**getKey**

```
"This is the method to retrieve a key."

self currentKey = 0 ifTrue: [self readKey].
self currentKey = self highKey
      ifTrue:
            [self readKey.
            ^self currentKey].
self currentKey: self currentKey + 1.
^self currentKey
```

### *Consequences:*

✓ Wasted numbers if incremented too much when written back to the table.

✓ Lost time if not incremented enough when written back to the table.

✓ Unique single column keys improve performance and make the coding the SQL much simpler and easily abstracted.

### *Related Patterns:*

- An *OID Manager* is a *Singleton* so that all persistent objects use a common place to generate their keys.
- A *Strategy* could be used for generating the keys.

### *Known Uses:*

- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- All OODBMS use a key generation algorithm for creating unique keys when they save objects.
- The PLoP Registration uses the Microsoft Access sequence database command for generating unique keys.
- In DCOM, each interface is known at runtime by its Interface Identifier (IID). IID is a DCOM-generated globally unique identifier (GUID) for interfaces. GUIDs are 128 bit IDs generated by the DCOM API function CoCreateGuid. This API call works according to an algorithm specified by OSF DCE. It uses the current date and time, network card ID and a high-frequency counter. Basically, you use a tool (e.g., Microsoft Developer Studio) to obtain these GUIDs and then put them in your DCOM IDL.
- CORBA 2.0 has federated interface repositories--repositories that operate across multiple ORBs. To avoid name clashes, repositories assign unique IDs (Repository IDs in CORBA parlance) to global interfaces and operations. A Repository ID is a string consisting of a 3-level name hierarchy. CORBA 2.0 defines 2 formats:

    i) IDL names with unique prefixes. The components are: the string IDL, a list of identifiers separated by `/' and a major and minor version. Components are separated by `:' and the first identifier is a unique prefix--Java uses the same idea. For example:

        IDL:JoeYoder/Foo/Bar/:1.0

    JoeYoder is the unique prefix
    Foo is the module
    Bar is the interface

    ii) DCE Universal Unique Identifiers (UUIDs). This uses the UUID generator provided by DCE--like for DCOM. This time the components are separated by `:'. The first component is the string DCE, the second is the UUID and the third is a version number (no minor version). For example:

        DCE:100ab200-0123-4567-89ab:1

# Transaction Manager

## *Also Known As:*

Unit of Work
Transaction Object

## *Motivation:*

When saving objects, it is important to allow for objects to be stored in such a way that if some values are not stored properly, previously stored objects can be rolled back. In the patient example, you not only want to store the attributes of the `Patient` object, but you also want to store any changes to the `address` objects that it contains. If trying to save any one of the `Patient's Address` objects fails, you want to rollback any other writes to the database that was made for saving the patient information. A *Transaction Manager* provides support for beginning transactions, committing transactions, and rolling back transactions.

## *Problem:*

How do you write a group of objects to the database in such a way that if any of the writes fail, none of the objects are written.

## *Forces:*

- Objects are usually related in such a manner that if one object doesn't save correctly, then you do not want it's related object to save.
- There is an impedance mismatch of knowing which messages are transactions and which are not.
- Every write to the database could be setup as an individual transaction.
- Complex objects could build up a complicated SQL statement that takes care of transactions as a single unit of work.

## *Solution:*

*Build a Transaction Manager that works similar to other transactions managers. This manager allows for the beginning of transactions, the ending of transactions, the committing of transactions, and the rollback of transactions.* The transaction manager usually maps to the RDBMS's *Transaction Manager* if it provides one. All modern day databases have them built in so it is best to use what is provided.

It might be preferable to cache values for performance reasons. When this is the case, it is important for part of the *Transaction Manager* to include a way to commit and rollback cached values. If the data store that you are mapping to doesn't include a *Transaction Manager* then it is necessary for original state of objects to be saved at the beginnings of transactions so that if a rollback is requested, the original values can be written out to the data store.

## *Example Implementation:*

Without transaction management you could use something like:

```
anObject save.
```

This would commit each SQL statement as it is sent to the database. If there was any kind of failure in the middle of saving `anObject` you may end up with half `anObject` in the database, especially if `anObject` is a complex object.

When you apply transaction management you basically wrap up all the SQL statements built for `anObject` and tell the database to either save the bundle or forget the whole thing. This could look something like:

```
beginTransaction
anObject save
anotherObject save
endTransaction
```

This would ensure that both save commands executed successfully prior to committing the changes to the database. When your object are complex object where each object knows how to save its aggregate objects you want to make sure have a complete object written to the database.

The following code is from excerpts from the `PersistentObject`. These show the wrapper effect for implementing transaction management. The code (`self class beginTranaction`) tells the database to start and stop a transaction will be dependant on which database you are using and what language you are developing in. You may need to extend transactions to handle your specific implementation. Also, if the database you are saving to (maybe a flat file) doesn't support transactions, you are going to have to develop a complete *Transaction Manager* to support the needs of your users. This example wraps all saves and deletes to an object with a `beginTransaction` and an `endTransaction`. The class methods make the call to the database connection class. This is where VisualAge has provided the support for committing and rolling back transactions to its supported databases.

***Protocol for Public Interface*** `PersistentObject (instance)`

**save**

```
self class beginTransaction.
self saveAsTransaction.
self class endTransaction.
```

**delete**

```
self class beginTransaction.
self deleteAsTransaction.
self class endTransaction.
```

***Protocol for Public Interface*** `PersistentObject (class)`

**beginTransaction**

```
self databaseConnection beginUnitOfWorkIfError:
                        [ self databaseConnection rollbackUnitOfWork ]
```

**endTransaction**

```
self databaseConnection commitUnitOfWork
```

**rollBackTransaction**

```
self databaseConnection rollbackUnitOfWork
```

### *Consequences:*

✓ Complex objects are save completely or not at all.

✓ Referential integrity is maintained no half objects.

✗ All other applications writing to the database have to use the *Transaction Manager*. You may have to add checks to see if anyone else has changed the database values that you are saving.

✗ Unless there is support built in from the database, writing a complete transaction manager is hard.

### *Related or Interacting Patterns:*

- *Transaction Manager* is very similar to a *Transaction Object* [Keller 98-2]. *Transaction Objects* encapsulate transactions by building objects out of them. These *Transaction Objects* provide a operations for begin, commit, and rollback.

### *Known Uses:*

- Most all database type applications provide a similar function or definition for a unit of work.
- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- VisualWorks provides a transaction manager in their ObjectLens.
- VisualAge also provides support for transactions through the `ABTDbm*` classes.
- GemStone GemConnect provides transaction management for database objects.

# Connection Manager

## *Also Known As:*

Current Connection
Database Session

## *Motivation:*

Whenever a persisted object is being read in or written out to the database, a connection to the database must be made. The information typically needed during a database transaction is the name of the database, the name of the user, and the user's password. This information could be queried from the user each time access is made. This would make for a frustrated user and most users will not know the database name or alias. You could also capture the username and password on startup and build up the connection information and pass it around throughout the code. This makes for a lot of extra information being passed around when it is not needed. The best way is to build a *Connection Manager* that stores this global information and is used whenever access to the database is needed.

## *Problem:*

How does the persistent manager keep track of the database to connect to and what user is currently connected?

## *Forces:*

- All database connections need to have access to the where the connection is being made somewhere.
- Referencing global variables can keep code clean and straightforward.
- Maintaining the current database session rather than creating a new one for every transaction is more efficient and can make debugging easier.
- Providing a single place to maintain necessary connection information and operations can make code easier to understand and maintain.

## *Solution:*

*Create a Connection Manager object, which holds all of the values that need to be used for the database connection.* The common values are usually the database session, the current user logged into the system, and any other global information used for auditing, transactions, and the like.

When an application needs to keep one copy of some information around, it could use the *Singleton* pattern [GHJV 95] or a single active instance as described in the Design Patterns Smalltalk Companion [MORE HERE 98] and a *Session* [Yoder & Barcalow 97]. The *Singleton* is usually stored in a single global location, such as a class variable or as a class itself. Unfortunately, this pattern sometimes breaks down when an application is multi-threaded, multi-user, or distributed. In some situations a true *Singleton* may be needed. In other situations, each thread or each distributed process can be viewed as an independent application, each needing its own private *Singleton*. But when the applications share a common global environment, the single global location cannot be shared. A mechanism is needed to allow multiple *Singletons*, one for each application. Therefore, the *Connection Manager* is the *Singleton* that manages any and all database *Sessions*.

The *Connection Manager* is what the *Persistence Layer* interacts with to get the currently needed database connection. The *Connection Manager* also interacts with the *Table Manager* to get any needed database table names for any connection that is being made. Thus, once again, a *Strategy* [GHJV 95] could be used for choosing the appropriate database and table mappings. Imagine you had the possibilities to save your values out to five databases and you chose the one with the least activity for speed. The values could all synch up with a primary database in a batch process. This distributed database system could be accessed through a *Strategy* thus insuring users a way to get good performance.

### *Example Implementation:*

The *Connection Manager* establishes the connection to the database based on information from the *Table Manager.* When the *Persistence Layer* requests a connection the *Connection Manager* uses the Database Components to provide an interface to the database via ODBC, CLI, or what ever else is available. Should the connection be unavailable at the time, or drop in the middle, the *Connection Manager* has provided the error block to execute and the Error Collector records the error and recovers if possible.

In a nut shell, when a domain object says:

```
self databaseConnection
```

the connection is returned from the *Connection Manager.* The type of connection and which connection (if multiple connections are used) is determined by the *Connection Manager*.

In our example, either a local or remote database is chosen. We have revealed how you would interact with the ABTDbm* classes from VisualAge to get the needed connection for the local database.

***Protocol for Public Interface*** ConnectionManager (class)

```
databaseConnection
      "Checks the local variable to see which connection to make or
       test for and returns the connection."

      ^self getLocal
            ifTrue: [self localConnection]
            ifFalse: [self remoteConnection]

localConnection
      "Answer a connection to the PPLPersistentObject.  The alias name
       is checked to see if a connection is already open, if not one is
       created with error blocks to trap errors that occur during
       connection and while the connection is open."

      | connection |
      (connection :=
         AbtDbmSystem activeDatabaseConnectionWithAlias: 'Example') isNil
            ifTrue:
            [connection :=
               ((AbtDatabaseConnectionSpec
                        forDbmClass: #AbtOdbcDatabaseManager
                        databaseName: 'Example')
                        promptEnabled: false;
                        connectUsingAlias: 'Example'
                        logonSpec: (AbtDatabaseLogonSpec
                                    id: ''
                                    password: ''
                                    server: '')
                     ifError: [:error | PPLErrorCollector dbConnectionError])
               autoCommit: false;
            yourself].
```

```
"If the environment is development no error block is used so the
 debugger will appear.  During runtime the error block is set to
 invoke the error collector.  This is for the manager and will
 trap all dbm type errors.  The error block above is specific to
 the connection."
      connection databaseMgr
      errorBlock: (System startUpClass isRuntime
      ifTrue: [[:error | PPLErrorCollector dbmError: error]]
      ifFalse: [nil]).
^connection
```

### *Consequences:*

✓ The *Connecion Manager* provides the connection information for all persisted objects.

✓ This makes for a single place of change when the connection information changes.

✓ This provides for a way to support multiple connections which can be used for an object to load from the database or when you want to distribute your database load.

### *Related or Interacting Patterns:*

- *Connection Manager* is an alternative to a *Singleton* [GHJV 95] in a multi-threaded, multi-user, or distributed environment.
- *Connection Manager* is similar to a *Session* in that the *Connection Manager* maintains the *Session* to the database and the needed global information related to the database connection.
- *Connection Manager* can use a *Strategy* for choosing the current connection.

### *Known Uses:*

- For VisualWorks, the Lens framework for Oracle and GemBuilder for GemStone have `OracleSession` and `GbsSession` classes respectively. Each keeps information such as the transaction state and the database connection. The *Connection Manager* is then referenced by any object within the same database context.
- The Caterpillar/NCSA Financial Model Framework has a `FMState` class [Yoder 97], which serves as a *Session* and maintains the *Connection Manager*.
- VisualAge has `AbtDbmSystem`, which maintains track of the active connection. *Connection Manager* makes calls to the VA classes that in turn control the physical connections.
- Illinois Department of Public Health TOTS and NewBorn Screening projects.

# Table Manager

## *Also Known As:*

Data Dictionary
Table Mappings

## *Motivation:*

The persistent storage that objects map to may evolve over time, or there may be multiple stores for objects. A *Table Manager* describes the mappings of databases to tables and columns, thus keeping the details away from the developer. This allows for changes to the database naming schema to not affect the application developer.

## *Problem:*

How does an object know what table and columns name(s) to use especially when multiple tables are needed to save the object? This magnitude of the problem increases when multiple databases are used.

## *Forces:*

- Mapping objects to relational databases ultimately requires mapping values to database tables.
- Developers want to be able to quickly change the table names as an application grows and changes.
- Mapping objects from identical tables or views with different names at different locations.
- Ability to switch from one database to another easily.
- Hard coding table names and column names where SQL code is described is very straightforward.

## *Solution:*

*Provide a place to retrieve the necessary table and column name(s) needed for objects to persist themselves.* When an object is being stored, have it look up its table names in a *Table Manager*. By having the string that defines the name in only one-place, modifications and testing the changes becomes very quick and efficient.

This pattern uses a *Singleton* [GHJV 95] to return to the domain object the table or column name(s) it needs. The instance contains dictionaries to store the names and when the object sends a message requesting the name, the instance methods will check with the *Connection Manager* to decide which dictionary to address. This provides the ability for the object to send the same message regardless of which database it must access.

When retrieving data from multiple sources, this pattern provides the ability to dynamically change which database or table is accessed. This flexibility alleviates the necessity to write the same *SQL Code Description* over and over for the same tables in multiple databases and still accesses it sequentially.

The *Table Manager* in a sense holds on to *Metadata* for the table mappings. This *Metadata* is used whenever a table name is needed. It could be used for dynamically generating database queries. Basically, a *Schema* describing the mappings is used whenever access to the database is needed.

## *Example Implementation:*

The *Table Manager* stores the table names in dictionaries to provide to the *SQL Code* when requested. When the *Table Manager* is initialized the dictionaries are initialized with the table names. Then according to the *Connection Manager* the appropriate dictionary is accessed and the table name is returned.

When a domain object says:

```
self class table,
```

the `TableManager` will return the name of the table for the class.

### *Consequences:*

✓ One advantage for using *Table Manager* is that when a table name changes there is only one place to make the change in the code.

✓ Another advantage, multiple databases can be accessed with the same table or view having a different name.

✗ Writing a table manager involves creating a way to interpret the table and column mappings isn't always easy to develop or maintain.

### *Related or Interacting Patterns:*

- The *Persistence Layer* and the *Connection Manager* call the *Table Manager* whenever database access is needed.
- The *SQL Code Description will use the Table Manager* in order to isolate the developer from changes to the database.
- The *Table Manager* is a *Singleton*.
- The *Table Manager* is really just a way of using *Metadata* to describe the *Schema* of the database.
- The *Table Manager* could use an *Interpreter* for complicated mappings or for dynamic systems using *Metadata*.

### *Known Uses:*

- Illinois Department of Public Health TOTS and NewBorn Screening projects.
- Data Dictionaries are used in many database systems.
- VisualWorks ObjectLens uses a *Spec* [Foote & Yoder 98] in conjunction with a *Schema* to provide the table mappings.

## Putting It All Together

Now that you have seen all of the patterns, you might be asking, "how do I fit it all together?" All of these patterns collaborate together to provide a mechanism for mapping persistent objects to a database. Figure 2 shows how the patterns interact with one another. The *Persistence Layer* provides the standard interface for the *CRUD* (create, read, update, and delete) operations needed to persist domain objects. The *Persistence Layer* builds the calls to the database by using the *SQL Code Description* provided by the domain objects. During the generation of the SQL code, the *Persistence Layer* interacts with the *Table Manager* to get the correct database table and column names. When values are being returned from the database or written back to the database, attributes values must be mapped to database column names and vice-versa. This is done with *Attribute Mapping Methods*. *Attribute Mapping Methods* do some *Type Conversions* while the SQL code is being generated. *Attribute Mappings* and *Type Conversions* can also happen while the *Persistence Layer* instantiates a new object. The *Persistence Layer* saves objects to the database through the *Connection Manager* only when an object's value has changed. This change is managed by the *Change Manager*. The *Connection Manager* could interact with the *Table Manager* to decide on which database to use. The *Persistence Layer* provides access to a *Transaction Manager* whenever transaction processing is needed.

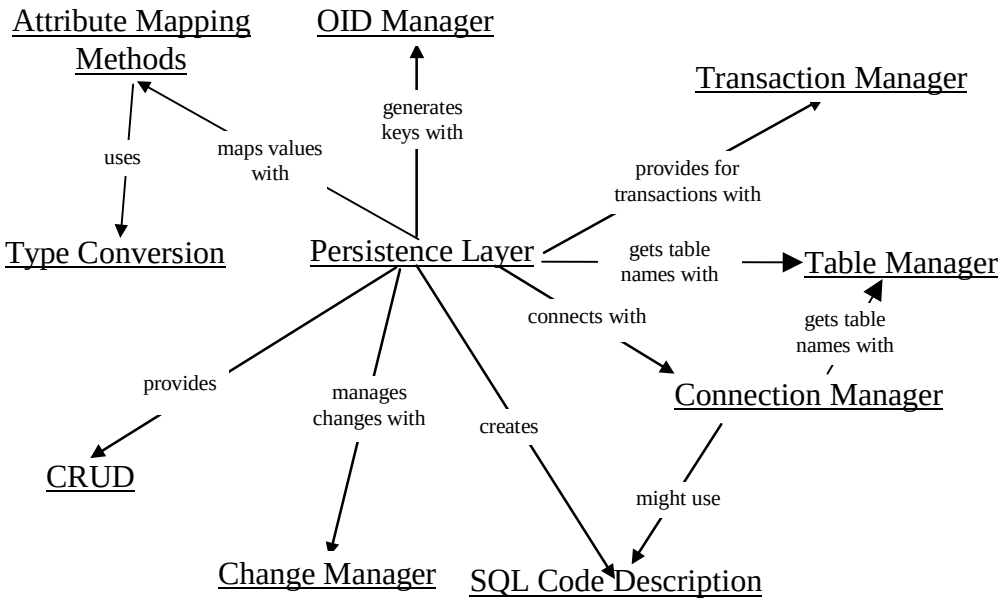


**Figure 2 - Pattern Interaction Diagram**

A class scenario graph (as seen in Figure 3) shows how the objects we used for our example code interact while loading values in from the database. A `Name` class can be asked to `load`. This message will be forwarded to its superclass, which is `PersistentObject`. This in return will get a connection from the `ConnectionManager`. The `PersistentObject` will then generate the SQL. In order to do this it will have to ask `Name` for the SQL and get any table name mappings from the `TableManager`. Once the SQL has been generated, a call to the `Database Component` will be made to get the resulting set of values. Then, for each returned row from the database, a new `Name` object will be created and for each attribute for that row, it will be assigned to the specific object attribute. During the mapping of the

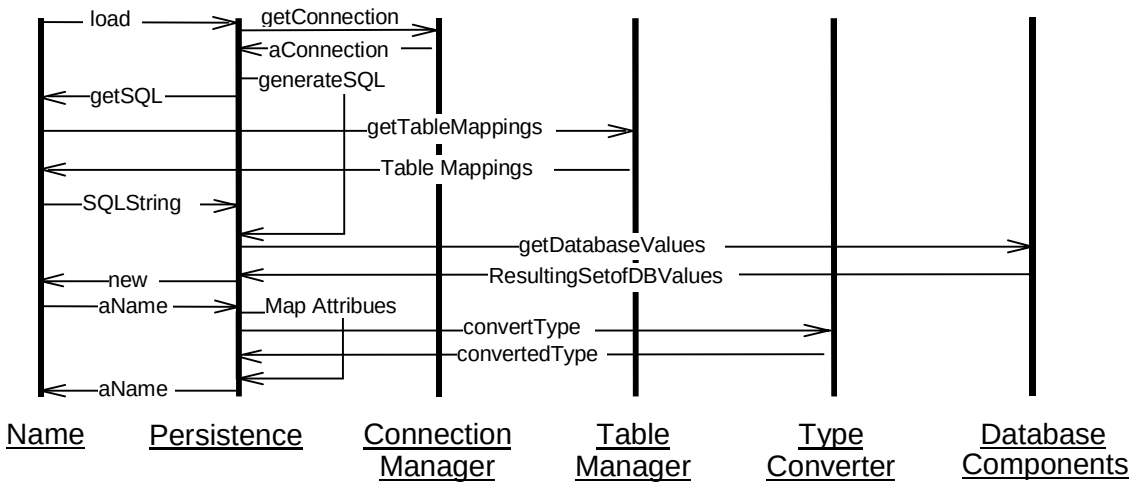


**Figure 3 - Class Interaction Diagram**

database value to the object attribute, the database types will be converted to the corresponding object type. Once this is complete, `PersistentObject` will return a collection of all `Names` that were created.

## Acknowledgments

We are grateful to the members of Professor Johnson's Patterns seminar: John Brant, Ian Chai, Brian Foote, Peter Hatch, Lewis Muir, Dragos Manolescu, and Aiji Nabika and Don Roberts; designers of the infrastructure projects at Caterpillar: Rhett Barnes, Anjali Sharma, Daniel Long, and Imad Zorob; analysts from the Illinois Department of Public Health: Rick Kirchgesner, Twyla Hulskotter, Sree Nair, John Loftus, Keith Hogan, and Sandy Sanders; our shepherd, Wolfgang Keller; our program committee member Bob Hamner; who reviewed earlier drafts and provided valuable feedback.

## References


[Ambler 97] **Scott W. Ambler.** *Mapping Object to Relational Databases.* URL: http:// www.AmbySoft.com/mappingObjects.pdf

[Beck 97] **Kent Beck.** *SMALLTALK Best Practice Patterns*, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[Brant & Yoder 96] **John Brant and Joseph Yoder.** "Reports," *Collected papers from the PLoP '96 and EuroPLoP '96 Conference*, Technical Report #wucs-97-07, Dept. of Computer Science, Washington University Department of Computer Science, February 1997. URL: http://www.cs.wustl.edu/~schmidt/PLoP-96/yoder.ps.gz.

[BMRSS 96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal.** *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons Ltd., Chichester, UK, 1996.

[Foote & Yoder 96] **Brian Foote & Joseph Yoder.** "Evolution, Architecture, and Metamorphosis," *Pattern Languages of Program Design 2*, John M. Vlissides, James O. Coplien, and Norman L. Kerth, eds., Addison-Wesley, Reading, MA., 1996.

[Brown & Whitenack 96] **Kyle Brown & Bruce Whitenack.** "Crossing Chasms: A Pattern Language for Object-RDBMS Integration," *Pattern Languages of Program Design 2*, John M. Vlissides, James O. Coplien, and Norman L. Kerth, eds., Addison-Wesley, Reading, MA., 1996.

[Foote & Yoder 98] **Brian Foote & Joseph Yoder.** "Metadata," Submitted to PLoP '98. URL: http://www-cat.ncsa.uiuc.edu/~yoder/Research/metadata.

[Fowler 97-1] **Martin Fowler.** *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997.

[Fowler 97-2] **Martin Fowler.** *Dealing with Roles*, Proceedings of PLoP '97, Monticello, IL, October 1997. URL: http://www.aw.com/cp/roles2-1.html.

[GHJV 95] **Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides.** *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[GemStone 96] **Gemstone Systems, Inc.** *GemBuilder for VisualWorks, Version 5*. July 1996. URL: http://www.gemstone.com/Products/gbs.htm.

[GemConn 96] **Gemstone Systems, Inc.** *GemConnect for VisualWorks, Version 5*. July 1996. URL: http://www.gemstone.com/Products/gbs.htm.

[Keller 97-1] **Wolfgang Keller**: *Mapping Objects to Tables: A Pattern Language*, in "Proceedings of the 1997 European Pattern Languages of Programming Conference," Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

[Keller 97-2] **Wolfgang Keller, Jens Coldewey**: *Relational Database Access Layers: A Pattern Language*, in "Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences" Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

[Keller 98-1] **Wolfgang Keller, Jens Coldewey**: *Accessing Relational Databases: A Pattern Language*, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[Keller 98-2] **Wolfgang Keller.** "Object/Relational Access Layers - A Roadmap, Missing Links and More Patterns," Submitted to EPLoP '98.

[OE 98] **IBM, Corporation.** *Object Extender for VisualAge.* 1998. URL: http://www.software.ibm.com/ad/smalltalk/about/persfact.html.

[Orfali & Harkey 98] **Robert Orfali & Dan Harkey.** *Client/Server Programming with Java and {CORBA}*, 2nd Edition, John Wiley & Sons, 1998.

[OS 95] **ObjectShare, Inc.** *VisualWorks User's Guide*. 1998. URL: http://www.objectshare.com/vw30abt.htm.

[RSBMZ 98] **Dirk Riehle, Wolf Siberski, Dirk Baeumer, Daniel Megert, & Heinz Zuellighoven.** *Serializer*, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.

[TopLink 97-1] **The Object People Inc.:** TOPLink Version 4.0 - A White Paper, 1997. URL: http://www.objectpeople.com/.

[TopLink 97-2] **The Object People Inc.:** TOPLink Version 4.0 - User Manual, 1997.

[VA 98] **IBM, Corporation.** *VisualAge Smalltalk*. 1998. URL: http://www.software.ibm.com/ad/smalltalk.

[Yoder & Barcalow 97] **Joseph Yoder & Jeffrey Barcalow.** "Security," *Fourth Conference on Patterns Languages of Programs (PLoP '97)* Monticello, Illinois, September 1997. Technical report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997. URL: http://www-cat.ncsa.uiuc.edu/~yoder/papers/patterns/#YoderBarcalow1997.

[Yoder 97] **Joseph Yoder.** *A Framework to Build Financial Models*. URL: http://www-cat.ncsa.uiuc.edu/~yoder/financial_framework.

[Yoder & Wilson 98] **Joseph Yoder & Quince Wilson.** *A Framework for Persisting Objects to Relational Databases*. URL: http://www-cat.ncsa.uiuc.edu/~yoder/Research/objectmappings.

[You+ 95] **Joseph Yoder & Quince Wilson.** *Mainstream Objects, An Analyusis*. URL: http://www-cat.ncsa.uiuc.edu/~yoder/Research/objectmappings.