# Patterns for Making Business Objects Persistent in a Relational Database

Presented By:
Joseph W. Yoder
joeyoder@joeyoder.com
www.refactory.com

## More Info

For additional information see
http://www.joeyoder.com/Research/objectmappings



## Collaborators

Ralph Johnson johnson@cs.uiuc.edu

Quince Wilson qwilson@issintl.com

The Refactory, Inc. www.refactory.com

 2

## Goals

- Look at patterns for making objects persistent in a non-object world
- Learn a framework for mapping your objects to a relational database
- Take some sample code or ideas back with you that you can use in your development environment

 3

## Overview

- Motivation
- The Way it Used to Be
- Patterns for Mapping Objects to RDBMS's
- How we Developed our Framework
- The Design of our Framework
- The Relational Database Side of Things
- Meta-architecture for mapping objects to RDBMS's
- Summary

 4

## Motivation

Systems are often developed where mappings to a relational database for domain values are needed. Quite often relational calculus and the maturity of relational databases are exactly what one needs. Other times it might be that the corporate policy is to use a relational database rather than an object -oriented database.

 5

## Problems with OO-RDBMS Mappings

- Impedance Mismatch of technologies

Objects - hierarchies, types, composition, polymorphism, relate code and data

Relations - rows, tables, relational calculus permanent storage, data access

 6

## Illinois Department of Public Health "Relational Mappings"

Many common systems that need to share data in similar domains, want to reuse base "Enterprise Components" (need to share a similar persistence mechanism)

- New Born Screening
- Food, Drug, and Dairy
- High Risk
- Refugee
- ….

 7

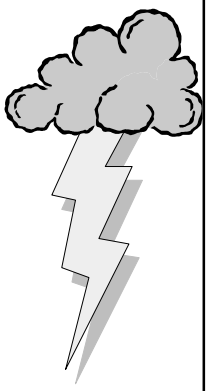
## The Way It Used To Be

VisualAge with Parts (Using Brokers)

- Pros:
  - GUI Builders (don't need to understand much Smalltalk)
  - Code automatically generated
  - Good for quick VisualBasic type of solutions.

 8

# The Way It Used To Be

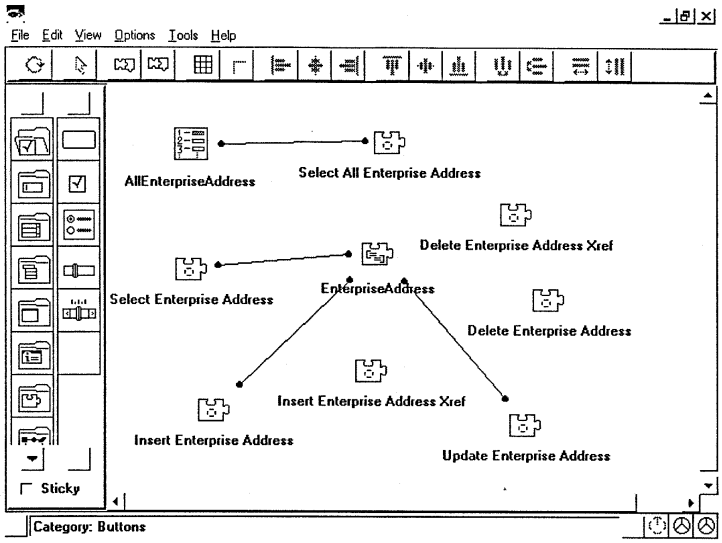


VisualAge with Parts (Using Brokers)

- Cons:
  - Complicated when your number of tables grows.
  - Hard to Debug and Maintain
  - Visual Languages have their limits

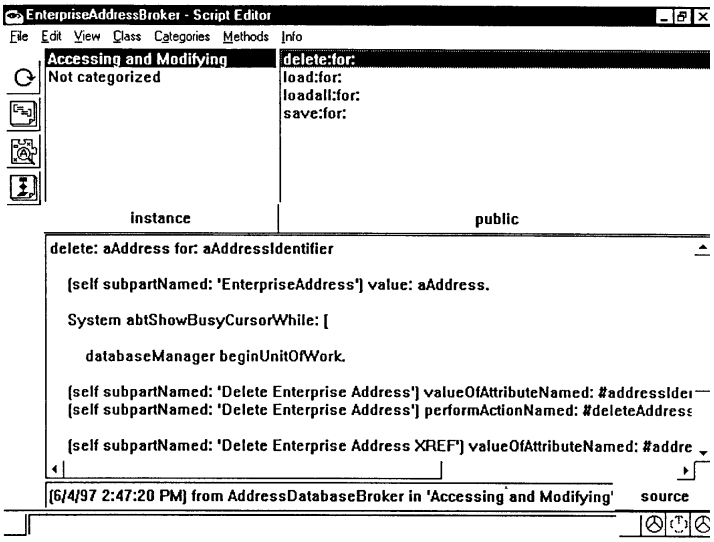
 9

# The Way It Used To Be



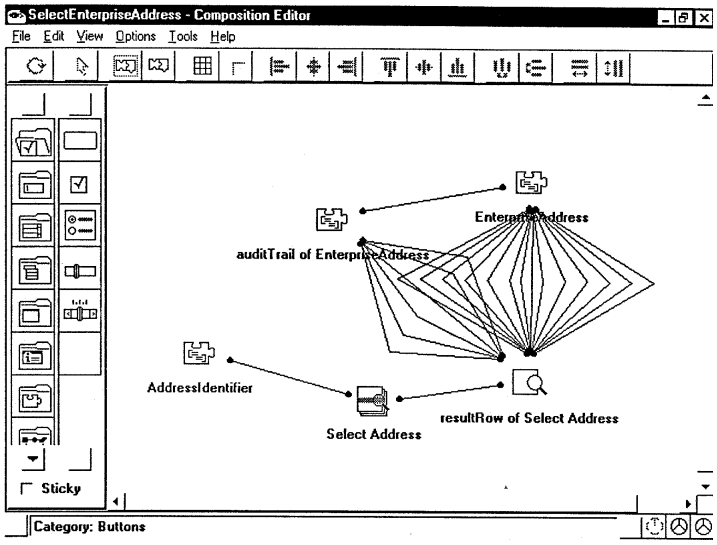

 10

# The Way It Used To Be



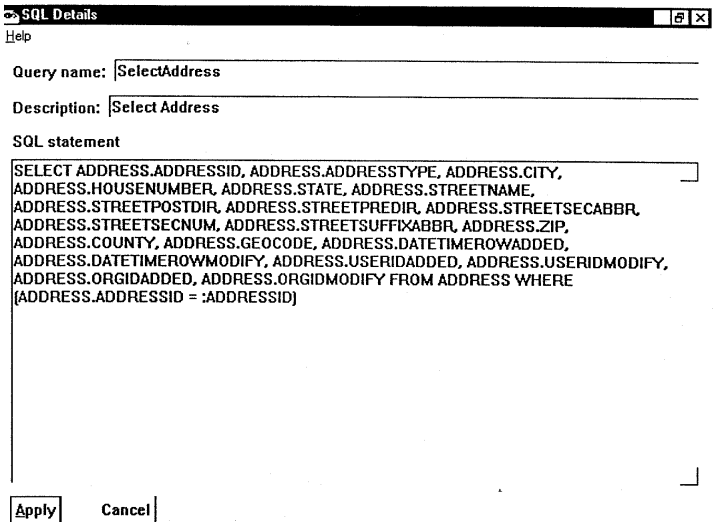

 11

# The Way It Used To Be




 12

# The Way It Used To Be




 13

# Mapping Objects To RDBMS Persistence Pattern Language

- Persistence Layer
- CRUD
- SQL Code
- Attribute Mapping Methods
- Type Conversion

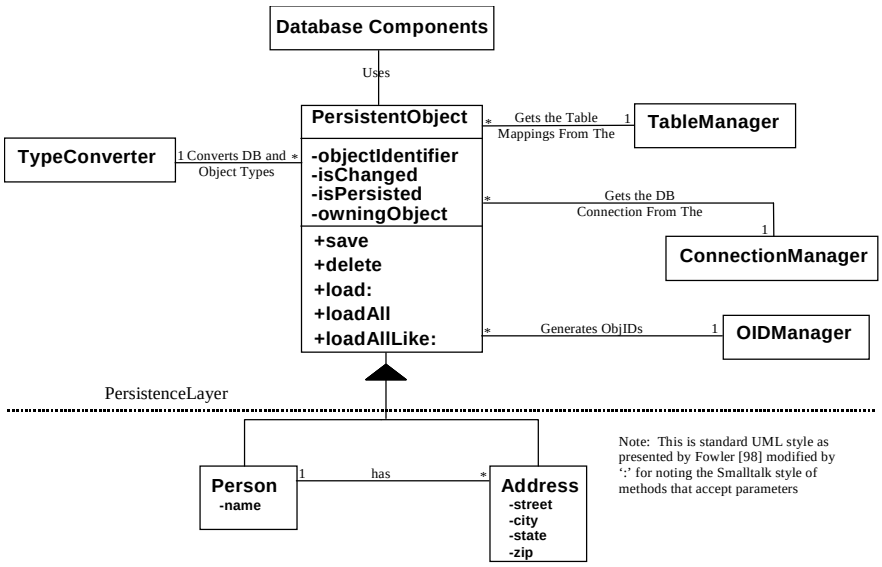
 14

# Mapping Objects To RDBMS Persistence Pattern Language

- Changed Manager
- OID Manager
- Transaction Manager
- Connection Manager
- Table Manager

 15

# The Patterns in Action




 16

# Persistence Layer

***Problem:***

How to make objects persistent to a non object-oriented storage application such as a relational database. This should be accomplished in such a fashion as to relieve the developers from having to know the exact implementation.

 17

# Persistence Layer

***Solution***:

*Provide a Persistent Layer in which objects are able to populate themselves from a data storage source as well as save themselves back to the data storage source.* This is really a special case of building a layer to protect you from changes. It is similar to *Adapters* and *Facades*. A standard interface is provided in which all objects that need to be persisted interface to.

 18

# Persistence Layer

***Discussion***:

- Use a Layered Object. Subclass each domain object from an abstract PersistentObject.
- Provide a *Broker* that can read or write domain objects to or from the database.
- Compose each domain object from a set of data objects that have a one to one mapping to the database tables.

 19

# Persistence Layer

***Using a Layered Object***:

- Inherit behavior for persisting to database.
- Overwrite methods for reading and writing values to and from the database.
- Layer isolates developer from database details.
- Easy to Write SQL mapping
- Have to inherit from a PersistenceObject

 20

# Persistence Layer

***Using a Broker***:

- Similar to a Layered Object in that you go through a separate layer to persist your objects.
- SQL Code is kept separate from your domain.
- Can inherit from any object and still have a persistent mapping.
- All SQL code is conglomerated together.

 21

# Persistence Layer

***Using Data Objects***:

- Create a one-to-one mapping between tables in the database and simple data objects.
- Create domain objects by using data objects.
- Simple and easy to map to…can easily map to any database.
- Let data objects know when they are dirty and save themselves when you commit.
- Have to handle complicated queries through views or multiple data objects

 22

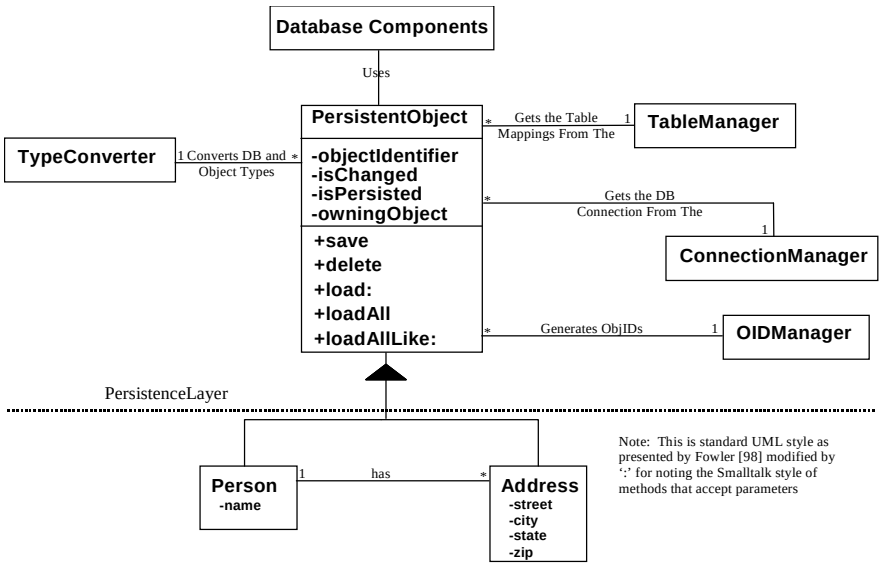
# Persistence Layer

***Our Solution***:

We create a Persistent object that any object that we want to be persisted can inherit from. The details of how the persistence is done is hidden from the user. It also provides a place where mapping to new types of persistence storage can be created and integrated into the system without affecting the application code.

 23

# Persistence Class Diagram




 24

## Persistence Object

- Abstract class
- Standard Interface to Persistence Layer
- Supports General CRUD Operations
  - (create, read, update, and delete)
- Sub-Classes overwrite the specific mappings to the database

 25

## Persistence Object (an example)

```
AbtObservableObject subclass: PersistentObject
    instanceVariableNames: 'objectIdentifier
                isPersisted isChanged owningObject '
    classVariableNames: ''
    poolDictionaries: ''
```

 26

## Persistence Object (an example)

```
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]
```

 27

## Persistence Object (an example)

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

 28

## Persistence Object (an example)

```
Protocol for Public Interface
  PersistentObject (instance)
save
  "Saves self to the database wrapped
  in a transaction."
  self class beginTransaction.
  self saveAsTransaction.
  self class endTransaction.
```

 29

## Persistence Object (an example)

```
Protocol for Public Interface
  PersistentObject (instance)
delete
  "Deletes self from the database
  wrapped in a transaction."
  self class beginTransaction.
  self deleteAsTransaction.
  self class endTransaction.
```

 30

## Persistence Object (an example)

```
buildSqlStatement: aString
    "Answer that each subclass must
     implement a queryStream method."
    ^self subclassResponsibility
```

 31

## Persistence Object (an example)

```
executeSql: aSqlStatement
  "Execute a sql statement using an custom error
  block in runtime or use the default debugger
  during development."
  | aQuery |
  aQuery:=AbtQuerySpec new statement:
                                    aSqlStatement.
  ..........
```

 32

## Persistence Object (an example)

```
executeSql: aSqlStatement (continued)
  ...
  (System startUpClass isRuntime)
      ifTrue:[self databaseConnection
            executeQuerySpec: aQuery
            ifError:[:error| ErrorCollector
                                    dbmError: nil ]]
      ifFalse:[self databaseConnection
            executeQuerySpec: aQuery].
```

 33

## CRUD (Create Read Update Delete)

***Problem:***

What minimal operations are needed for a persistence object?

***Solution*** :

*Provide the basic CRUD (create, read, update, and delete) operations for persistent objects.*

 34

## CRUD (example)

```
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

saveAsTransaction
      "Save self to the database."
  self isPersisted ifTrue: [self update]
                   ifFalse: [self create].
      self makeClean
```

 35

## CRUD (example)

```
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
```

 36

## CRUD (example)

```
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]
```

 37

## CRUD (example)

```
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)).
```

 38

# SQL Code

***Problem:***

How to maintain the consistency between the values from objects and the persistent storage? Where do you store the actual SQL statement necessary to read and write to the data source? How to provide a means where by a embattled programmer is less likely to forget to update a SQL statement when a domain object is modified?

 39

# SQL Code

***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).

 40

## SQL Code (example)

```
PersistentObject subclass: #Person
    instanceVariableNames: ' last first middle
                             email  organization phone '
    classVariableNames: ''
    poolDictionaries: ''
```

 41

## SQL Code (example)

```
SELECT * FROM table_name.

INSERT INTO table_name ( column_names )
  VALUES ( values ).

UPDATE table_name SET column_name = xyz
  WHERE key_value = someKeyValue.

DELETE FROM table_name [whereClause].
```

 42

# SQL Code (Person)

```
Person>>buildSqlStatement: aWhereCondition

    "This method builds the actual sql statement
    required to read records from the table.

    The aString passed in is the selection where
    clause produced by the selectionClause

    method of the object. The conditional check for
    where clause.  Provides the ability to

    load all (PPLPersistentObject>>loadAll ) the
    records from the table."
```

 43

# SQL Code (Person)

```
Person>>buildSqlStatement: aWhereCondition
   | 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.
   (aWhereCondition isNil or: [aWhereCondition trimBlanks
   isEmpty])
        ifFalse: [aStream setToEnd; nextPutAll: ' WHERE ';
                                nextPutAll: aWhereCondition].
   ^aStream contents
```

 44

## SQL Code (example)

```
updateRowSQL
   "Build the update sql statement from the object."
   | aStream |
   aStream := WriteStream on:(String new).
   aStream   nextPutAll: 'UPDATE ';
             nextPutAll: self class table;
             nextPutAll: ' SET NAM_FST=';
             nextPutAll: (self prepForSql:(self first
                                          asUppercase));
             nextPutAll: ', NAM_LST=';
             nextPutAll: (self prepForSql:(self last
                                          asUppercase));
```

 45

## SQL Code (example)

```
updateRowSQL
   "Build the update sql statement from the object."
   ...       nextPutAll: ', NAM_MID=';
             nextPutAll: (self prepForSql:(self middle
                                          asUppercase));
   ...
             nextPutAll: ' WHERE ID_OBJ=';
             nextPutAll: (self prepForSql:
                                    self objectIdentifier).

   ^aStream contents.
```

 46

## SQL Code (example)

```
insertRowSQL
   "Build the insert statement 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 (’…’);  ...
```

 47

## Attribute Mapping Methods

***Problem:***

Where and how does the developer describe the mappings between database values and attributes? When values are brought in from the database, it needs to be defined which attributes the values are mapped to and vice-versa.

 48

# Attribute Mapping Methods

***Solution*** :

*For every domain object that needs to be persistent, create a means to describe the mappings between the database columns and object attributes.* In our case, we write a method that describes the mappings from the database values to the object's attributes and write a method which maps the values from the object back to the database.

 49

# Attribute Mapping Methods

***Discussion*** :

The Persistence Layer will use this method to take the returned values from the database and stored them in the appropriate object attribute. Similarly, when the persistent object is being saved, the Persistent Layer will use a similar method for taking values from the object and putting them out to the database.

 50

## Attribute Mapping (example)

```
initialize: aRow
    "Initializes an instance from the database row."
    objectIdentifier := self asNumber:( aRow at: 'ID_OBJ' ).
    owningObject := ( aRow at: 'ID_OBJ_OWN' ).
    isPersisted:=true.
    first := self asUpperString:( aRow at: 'NAM_FST' ).
    middle := self asUpperString:( aRow at: 'NAM_MID' ).
    last := self asUpperString:( aRow at: 'NAM_LST' ).
    email := self asString:( aRow at: 'EML_ADR').
    organization := self asUpperString:( aRow at: 'ORG_NAM').
    phone := self asNumber:( aRow at: 'NUM_PHO')
```

 51

## Attribute Mapping (example)

```
updateRowSQL
  "Build the update sql statement from the object."
  | aStream |
  aStream := WriteStream on:(String new).
  aStream   nextPutAll: 'UPDATE ';
            nextPutAll: self class table;
            nextPutAll: ' SET NAM_FST=';
            nextPutAll: (self prepForSql:(self first
                                          asUppercase));
            nextPutAll: ', NAM_LST=';
            nextPutAll: (self prepForSql:(self last
                                          asUppercase));
```

 52

# Type Conversion

***Problem:***

There is an impedance mismatch between RDB-types & object-types. How do we take objects that may not have a type in a database and allow for them to map to a database type? How do we ensure the data read from the data source will work with our object? How do we ensure the data written to the data source will comply with the data source's rules and maintain data integrity?

 53

# Type Conversion

***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 data source the data formats can vary. This pattern ensures the data retrieved from the data source is appropriate for the object. This can also be a *Strategy* for pluggable type converters

 54

## Type Conversion (example)

```
Object subclass: # TypeConverter
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

 55

## Type Conversion (example)

prepForSql: anObject (continued)

```
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]]….
```

 56

# Type Conversion (example)

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

 57

# Changed Manager

***Problem:***

Many objects need access to shared values, but the values are not unique throughout the system. How to tell that an object has changed and needs to be saved to the data source? How to prevent unnecessary access to the data source?

 58

# Changed Manager

***Solution*** :

*Create a means to keep track of what objects are dirty so you can insure to save these changes when desired. In our case, we inherit from a Persistent object, which has a dirty bit that gets set whenever one of its attributes that maps to the database is changed.* This dirty bit is usually an instance variable with a boolean value which indicates when an objects values have changed.

 59

# Changed Manager

***Discussion*** :

When the boolean value is set the Persistent object will save the new values to the data source. If the boolean value is not set the Persistent object will bypass the write to the data source. Dependent upon the class hierarchy the implementation can vary. One solution is to modify the setter methods to set the flag whenever an object's values are changed.

 60

## Changed Manager

```
PersistenceObject>>makeDirty
   "Indicates an object needs to be saved to the db."
   isChanged:=true.

Person>>email: aString
   "Save the value of email."
   self makeDirty.
   email := aString.
   self signalEvent: #email
        with: aString.
```

 61

## OID Manager (Key Manager)

***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?

 62

# OID Manager

***Solution*** :

*Provide a Object Identity 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 data source a unique identifier is generated. The generation process needs to be quick and ensure uniqueness.

 63

# OID Manager

***Discussion*** :

*The OID Manager is usually an object that just encapsulates the key generation algorithm*. The *OID Manager* can use a *Strategy* to generate its unique key.

 64

## OID Manager (example)

```
PersistenceObject >>getKeyValue
  "Get a unique key for a new db row."
  (self objectIdentifier isNil  or:
                  [self objectIdentifier  =0])
     ifTrue:[^OIDManager getKey ]
     ifFalse:[^self objectIdentifier ].
```

 65

## OID Manager (example)

```
PersistenceObject  subclass: #OIDManager
   classInstanceVariableNames :
                              'singletonInstance '
   instanceVariableNames : 'increment highKey
                               currentKey lowKey '
   classVariableNames : ''
   poolDictionaries : ''
```

 66

## OID Manager (example)

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

 67

## OID Manager (example)

```
readKey
    "Generate a unique key value for use in
     storing  an object in the database."
    | newKey aQuerySpec aResultSet aMaxKey prep |
    PersistentObject beginTransaction.
    aQuerySpec := AbtQuerySpec new
                        statement: ' SELECT  NUM_SEQ
    FROM SEQUENCE '.
    aResultSet := PersistentObject databaseConnection
                        resultTableFromQuerySpec:
    aQuerySpec.
    aMaxKey := aResultSet first at: 'NUM_SEQ'.
    newKey := aMaxKey + self increment.
     ……...
```

 68

# OID Manager (example)

```
readKey (continued)
    "Generate a unique key value for use in
     storing  an object in the database."
     ……...
    PersistentObject
          executeSql: 'UPDATE SEQUENCE
                       SET NUM_SEQ = ' , newKey printString.
    PersistentObject endTransaction.
    prep := self class siteKey * self keySize.
    self lowKey: prep + aMaxKey.
    self currentKey: prep + aMaxKey.
    self highKey: prep + (newKey - 1).
```

 69

# Transaction Manager

***Problem:***

How do we group together the saving of multiple objects in such a way that if the saving of one object fails, then we want the other objects to not be saved?

 70

## Transaction Manager

***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.

 71

## Transaction Manager (an example)

```
beginTransaction
  "Tell Persistence Mechanism that a
    Transaction is beginning."
  self databaseConnection
  beginUnitOfWorkIfError : [ self
    databaseConnection rollbackUnitOfWork  ]
```

 72

## Transaction Manager (an example)

endTransaction

"Tell Persistence Mechanism that a Transaction is ending."

self databaseConnection commitUnitOfWork

 73

## Connection Manager

***Problem:***

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

 74

# Connection Manager

***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.

 75

# Connection Manager

***Discussion*** :

The Connection Manager establishes the connections to the databases. A *Strategy* can be used for deciding which connection is needed if multiple database servers are being used. The Connection Manager can use a *Session*.

 76

## Connection Manager (example)

```
Object subclass: ConnectionManager
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

 77

## Connection Manager (example)

```
databaseConnection

    (TableManager get local)
        ifTrue:[ ^self localConnection ]
        ifFalse:[ ^self remoteConnection ]
```

 78

# Table Manager

***Problem:***

How does an object know what table name(s) to use especially when multiple tables are used to persist the object? The persistent storage that objects map to may evolve over time, or there may be multiple stores for objects. This magnitude of the problem increases when multiple databases are needed.

 79

# Table Manager

***Solution*** :

*Provide the object a means to retrieve the necessary tables it needs to persist itself.* A *Table Manager* describes the mappings of databases to tables, thus keeping the details away from the developer. This pattern uses a *Singleton* instance to return to the domain object the table name(s) it needs.

 80

## Table Manager

***Discussion*** :

The instance contains arrays to store the names and when the object sends a message requesting the name, the instance methods will check a instance variable to decide which array to address. This provides the ability for the object to send the same message regardless of which data source it must access.

 81

## Table Manager

```
Object subclass: # TableManager
    classInstanceVariableNames :
                                'singletonInstance '
    instanceVariableNames : 'localTables
                                remoteTables local '
    classVariableNames : ''
    poolDictionaries : ''
```

 82

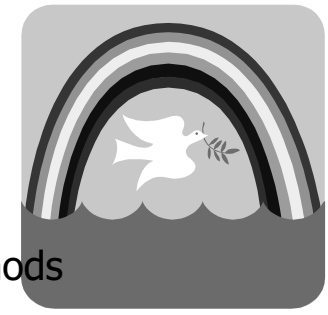
## Table Manager

Methods are provided to initialize the table manager and to access the desired table given a value.

 83

## Mapping Objects To RDBMS Persistence Pattern Language

- Persistence Layer
- CRUD
- SQL Code
- Attribute Mapping Methods
- Type Conversion



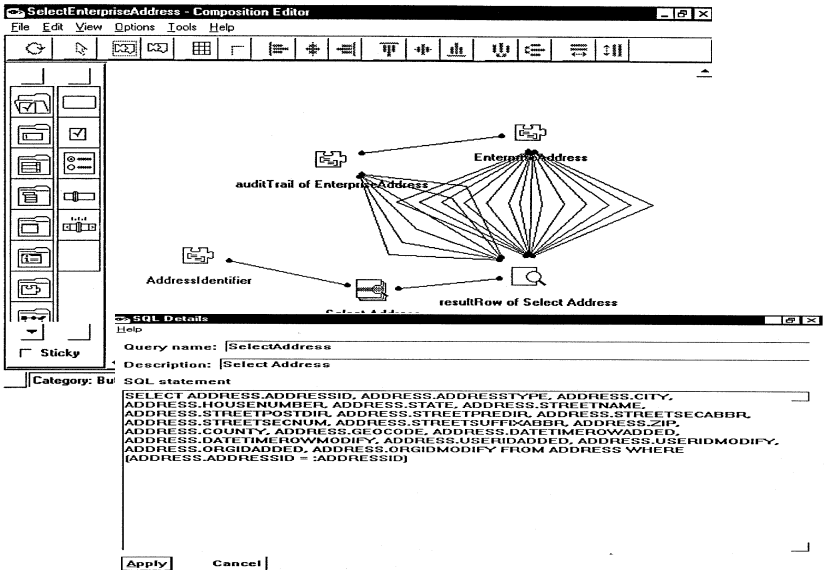
 84

# Mapping Objects To RDBMS Persistence Pattern Language

- Changed Manager
- OID Manager
- Transaction Manager
- Connection Manager
- Table Manager

 85

# Before PersistentObject




 86

## After PersistentObject

- Subclass from PersistentObject
- Overwrite:
  - TableName
  - Initialize Method
  - Create, Update, Read
- Much simpler and easier to understand and maintain

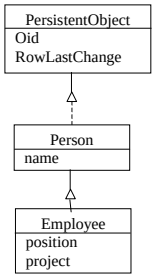
 87

## Mapping Objects (the relational side)

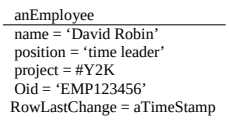
- Object Identification
  - uniquely generated
  - "HIGH/LOW OID" presented in [Ambler98].
- Classes to Tables [Brown Whitenack 96].
- Handling Related Objects
  - owned objects
  - knowledge relationships
  - many-to-many
- Optimizations (denormalization)

 88

# One Table for each Concrete Class




a) Object Model

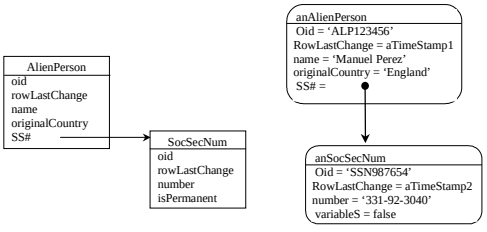



b) Instance Diagram

| oid | ownerOid | rowLastChange | name | position | project |
|---|---|---|---|---|---|
| | | | | | |
| EMP123456 | null | aTimeStamp | 'David Robin' | 'time leader' | #Y2K |
| | | | | | |

 89

# Owned Objects




a) Object Model

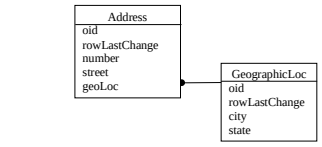
b) Instance Diagram

| oid | ownerOid | rowLastChange | name | originalCountry |
|---|---|---|---|---|
| | | | | |
| ALP123456 | null | aTimeStamp1 | 'Manuel Perez' | 'England' |
| | | | | |

| oid | ownerOid | rowLastChange | variableT | variableS |
|---|---|---|---|---|
| | | | | |
| SSN987654 | ALP123456 | aTimeStamp2 | '331-92-3040' | false |
| | | | | |

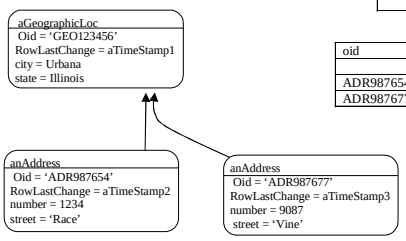
 90

# Knowledge Relationships




a) Object Model

| oid | ownerOid | rowLastChange | city | state |
|---|---|---|---|---|
| | | | | |
| GEO123456 | null | aTimeStamp1 | Urbana | Illinois |
| | | | | |

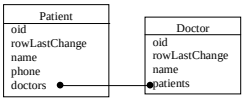



b) Instance Diagram

| oid | ownerOid | geoLocOid | rowLastChange | number | street |
|---|---|---|---|---|---|
| | | | | | |
| ADR987654 | null | GEO123456 | aTimeStamp2 | 1234 | Race |
| ADR987677 | null | GEO123456 | aTimeStamp3 | 9087 | Vine |

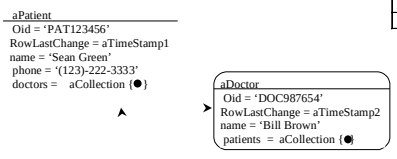
 91

# Many to Many Relationships




a) Object Model

| oid | ownerOid | rowLastChange | name | phone |
|---|---|---|---|---|
| | | | | |
| PAT123456 | null | aTimeStamp1 | Sean Green | (123)-222-3333 |
| | | | | |




| oid | ownerOid | rowLastChange | name |
|---|---|---|---|
| | | | |
| DOC987654 | null | aTimeStamp2 | Bill Brown |
| | | | |

b) Instance Diagram

| oid | table1OID | table2OID | table1ClassType | table2ClassType | rowLastChanged |
|---|---|---|---|---|---|
| | … | … | | | … |
| 77777 | DOC987654 | PAT123456 | Doctor | Patient | TimeStamp*N* |
| | … | … | | | … |

 92

## Mapping Objects (meta-architecture)

- Most of the time the only difference between the SQL generated is the table names, column names, and the rdb-types, attribute names, object-types
- When creating CRUD there are many places where the code looks very similar
- Can use a specification to parameterize the differences between objects

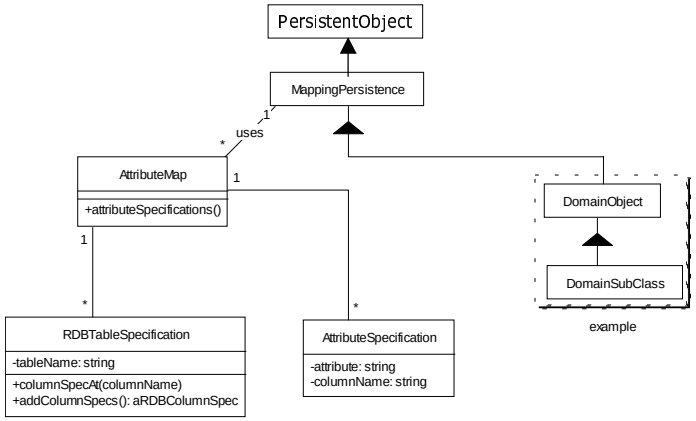
 93

## SQL Code (example)

```
SELECT * FROM table_name.

INSERT INTO table_name ( column_names )
  VALUES ( values ).

UPDATE table_name SET column_name = xyz
  WHERE key_value = someKeyValue.

DELETE FROM table_name [whereClause].
```

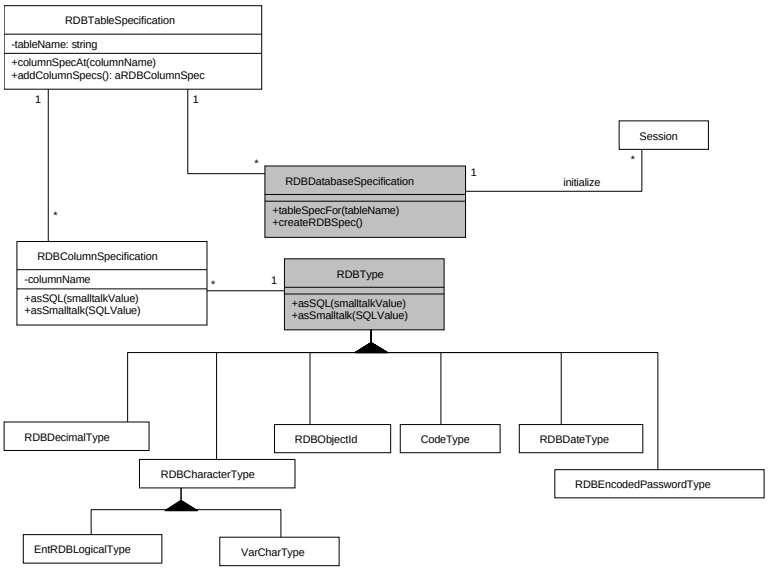
 94

# Mapping Objects (meta-architecture)




 95

# Mapping Objects (meta-architecture)



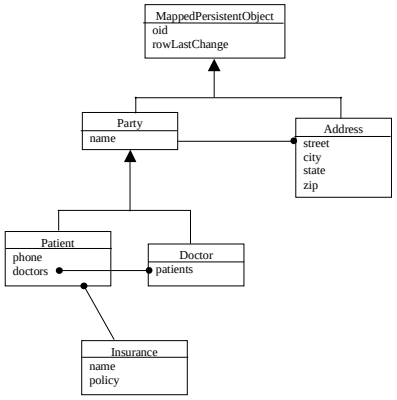

 96

# Attribute Map Specification

- Specification for the attribute map needs
  - Table Name(s)
  - Attribute Names
  - Column Names
  - RDB Types with Parameters
  - Type of Relationship

 97

# Attribute Mapping Example




Class Diagram

 98

## Attribute Map Example

```
#(#PATIENT_T
    #(#name #NAM_STR #(RDBVarChar 20) #simple)
    #(#address #Address #(RDBObjectId) #owned)
    #(#phone #PHONE_STR #(RDBVarChar 10) #simple)
    #(#doctors #Doctor #(RDBManyType) #many)
    #(#insurance #Insurance #(RDBKnowledgeType #INS_OID)
                                                #knowledge))
```

This spec could be XML or whatever…it is used to generate the SQL Code anytime a CRUD operation is executed

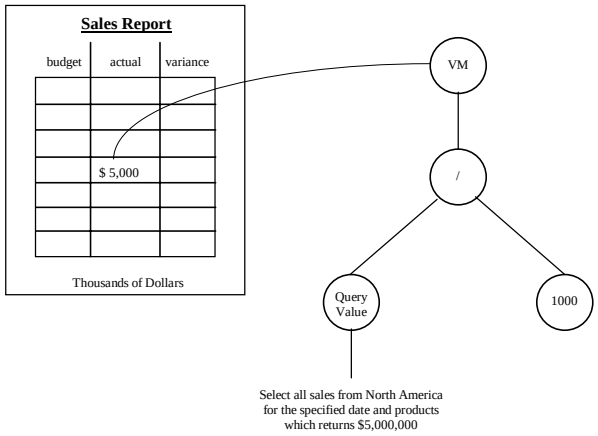
 99

## Hot Spots

- Find aspects that change, and make them objects
- Often are patterns from *Design Patterns: Elements of Reusable Object-Oriented Software*
- QueryObjects: Interpreter pattern

 100

# Typical Values in a Report




 101

# Query Objects

Business logic is equations expressed with ValueModel and QueryObjects

- Values = functions of other values
- Values = queries from the database

variable margin = net sales - variable cost

net sales = gross sales - warrantee

gross sales = sum *sales* column from *sales_and_transfer* table

 102

## Interpreter Pattern

- Need to represent SQL to manipulate query:

  SELECT SUM(sales) FROM sales_and_transfer

  WHERE family='MWL' AND date < '1/1/96' AND date > '1/1/97'

- Problem: how do you represent a simple language?

 103

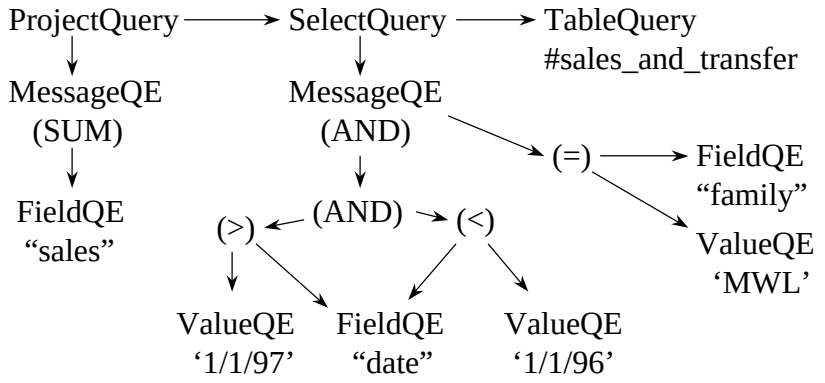
## Interpreter Pattern

1) make a class hierarchy that represents nodes in abstract syntax tree (SELECT, AND, <, tables, field names)
2) define methods to construct and manipulate tree
3) define method to compute value of query (this is the "interpreter")

 104

# Instance Hierarchy

SELECT SUM(sales) FROM sales_and_transfer
WHERE family='MWL' AND date < '1/1/96' AND date > '1/1/97'




 105

# QueryObjects

QueryObject
- TableQuery
- JoinQuery
- WrapperQuery
  - RenamingQuery
  - ExpressionQuery
    - SelectQuery
    - ProjectQuery
    - OrderQuery

QueryExpression
- ValueQE
- MessageQE
- FieldQE
  - RenamedFieldQE

 106

## QueryObject Protocol

- values - answer collection of tuples
- fieldNames
- join: aQueryObject
- select: aQueryExpression
- project:, renameColumnsTo:, outerJoin:, groupBy:, orderBy:, asDistinct

 107

## Creating a QueryObject

```
salesQ := #sales_and_transfer asQuery.
dateQ := salesQ select:
        ((salesQ @@ 'family') = 'MWL') &
        ((salesQ @@ 'date') > '1/1/96') &
        ((salesQ @@ 'date') < '1/1/97').
dateQ project: (dateQ @@ 'sales') Sum
```

 108

# QueryExpression Protocol

+, -, <, =, &, |, Sum, Average, Count, …

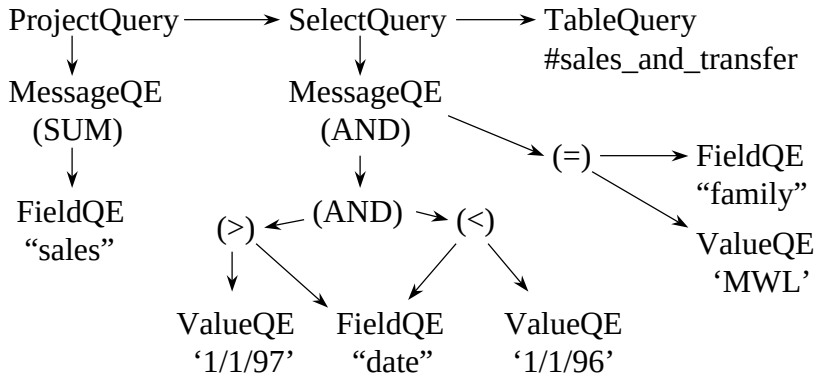
Sending one of these messages to a QueryExpression builds a MessageQE with the appropriate operands, and with the message as the operator.

 109

# Observer and QueryObjects

Let ValueQE refer to a ValueModel.

Let each QueryObject observe its components.




 110

# Triggering Changes

Old way - route change through report

*Update*

```
budget value: (query1 values first first).
actual value: (query 2 values first first).
difference value: budget value - actual value
```

New way - route change directly to ValueModel

*Initialization*

```
budget := QueryHolder on: query1
actual := QueryHolder on: query2
difference := budget - actual
```

Requires:

- QueryHolder - adapts QueryObject to ValueModel
- ValueModel understands +, -, *, /, etc

 111

# QueryHolder

Adaptor pattern - subclass of ValueModel that lets QueryObject act like ValueModel.

*instance variables*: query, values

```
query: aQuery
    query := aQuery.
    aQuery addDependent: self
```

 112

## QueryHolder

```
update
    values := aQuery values
    self changed

value
    ^values first first
```

 113

## Arithmetic on ValueModels

ValueModel implements arithmetic by creating ValueModels that compute function.

```
+ anObject
    ^BlockValue
        on: [:a :b | a value + b value]
        with: (Array with: self with: anObject)
```

 114

## Summary

- Visual Languages are powerful but have their limits
- A Persistent Layer helps hide database technology details from the application developer and makes it easier to change persistent storage technologies without affecting the application code

 115

## Summary

- Patterns are good for documenting a framework and for describing how to build a similar framework.
- This pattern language follows what one needs to do when dealing with persisting objects in a non-object world.

 116

# Related Links

- The following link discusses the details of the framework
  http://www.joeyoder.com/Research/objectmappings
- Object-Oriented page with framework references
  http://www.joeyoder.com/Research/Frameworks/
- Joe's Patterns Paper
  http://www.joeyoder.com/papers/patterns
- The reporting patterns describing query-objects - PLoP '96
  http://www.joeyoder.com/papers/patterns/Reports/
- Evolving Frameworks - PLoP '97
  http://st-www.cs.uiuc.edu/users/droberts/evolve.html

 117

# Related Links

- Security Patterns describing Sessions - PLoP '97
  http://www.joeyoder.com/papers/patterns/
- Crossing Chasms
  http://www.ksccary.com/ordbjrnl.htm
- Mapping Object to Relational Databases
  http:// www.ambysoft.com/mappingObjects.pdf
- Relational Database Access Layers
  http://www.sdm.de/g/arcus/cookbook/relzs/
- Metadata and Active Object-Models
  http://www.joeyoder.com/Research/metadata

 118

# That's All

 119