Polymorphic Stored Procedure?

Venkat Subramaniam
venkats@durasoftcorp.com
http://www.durasoftcorp.com/download

Abstract

A typical user of JDBC or ASP.NET issues a SQL query to the underlying database, grabs the fields returned by the record set/result set (or dataset) and then populates an object with the data fetched. Not considering the use of Entity Beans and JDO in Java, what does one do if the object being fetched is one of several types derived from a common base type? This article addresses one way this can be solved in an extensible manner.

A Simple Data Access Scenario

For some one who has spent significant time developing OO applications that used object-oriented databases and those that did not use any database at all, it is painful to use a relational database with objects. I am sure you have heard people with OODBMS experience talk about data access impedance mismatch when it comes to storing an object in a relational database. You hear less of this now than you did in the early nineties. While there is some market for ODBMS still, I have come to agree that relational databases are here to stay. After developing a couple of applications using relational databases, I was pondering about arriving at a solution to the problem of mapping the inheritance hierarchy in an extensible way. Is this really a good approach to follow, is for you to decide. I simply present here my thoughts on what I tried recently. Your comments are most welcome.

Let's consider a database with four tables: Customer, Account, CheckingAccount, SavingsAccount. For the sake of simplicity, assume a Customer has only one Account. An Account may be either a CheckingAccount or SavingsAccount. In UML notation, the relationship may be expressed as shown in Figure 1.

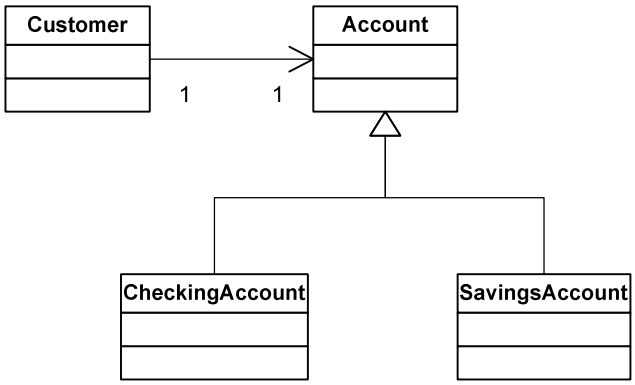


Figure 1. Relationship between entities in our example

First we access a Customer object from the database table Customer. This object holds the account number which is the account belonging to this customer that we fetched. This

account, however, may either be a checking account or a savings account. How do we fetch the appropriate account from the database?

First Attempt

One approach, given the account id, would be to query the database for the account type. Then based on the type of the account, you may create an object of CheckingAccount or SavingsAccount. Figure 2 shows a pseudo code to do just that.

```
1. fetch account type as accountType from Account table for given account id
2. If accountType == "CheckingAccount" acct = new CheckingAccount(id);
3. If accountType == "SavingsAccount" acct = new SavingsAccount(id);
4. acct.LoadDataFromDatabase();
```

Figure 2. Pseudo code to fetch an account given its id

In step 4 of Figure 2, we call the LoadDataFromDatabase on either the CheckingAccount or the SavingsAccount and the appropriate method will load the data from the appropriate table.

There are two disadvantages with the above approach. First, we have to run to the database twice: once to find the type of the account and again to get the actual data. Second, if more types of accounts are introduced, the code to create the appropriate type of account is affected. In short, this code fails OCP[1].

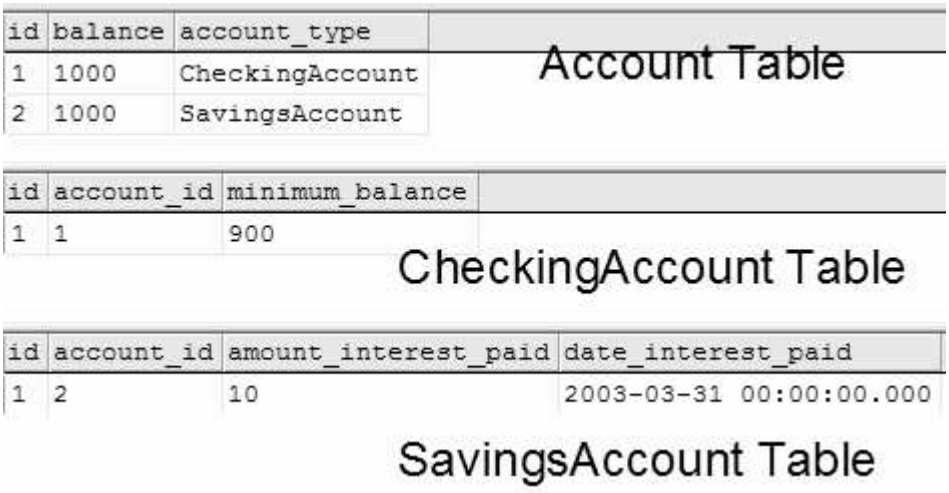
An Extensible Approach

We can think of an alternate way to approach this problem. The approach presented here requires some handling at the stored procedure level and at the code level as well. The idea is fairly simple (I hear you say “it is always simple, isn’t it!”). We will write a stored procedure that examines the type of account and dynamically returns the appropriate information from the correct table (details presented later). Then at the Java code level (as much as in C# code if you choose to), you can use Reflection to dynamically create the object of the appropriate type and load it up. The rest of this article will focus on illustrating this. Let’s first start with the database stored procedure level.

Polymorphic Stored Procedure?

Quickly we will first present the columns of our table. We only focus on the Account table, CheckingAccount table and SavingsAccount table. Figure 3 shows the columns of these tables along with some sample data. Note that the id is the primary key in the Account table and the account_id in the CheckingAccount and SavingsAccount are foreign keys representing the id of the account. Our goal is to write a stored procedure that returns the columns from the appropriate table based on the type of the account. At the same time, we want that stored procedure to be extensible as well. In other words, we do not want to modify that stored procedure if a new type of account is introduced. First we write two stored procedures selCheckingAccount and selSavingsAccount. We will adhere to a strict naming convention here (If you do not want to stick to a naming convention, then you can write a mapping table to map your table names to the store

procedures you will have to use. You should be able to figure that out easily once you understand what's going on). The two stored procedures are shown in Figure 4.



id	balance	account_type
1	1000	CheckingAccount
2	1000	SavingsAccount

Account Table

id	account_id	minimum_balance
1	1	900

CheckingAccount Table

id	account_id	amount_interest_paid	date_interest_paid
1	2	10	2003-03-31 00:00:00.000

SavingsAccount Table

Figure 3. Account related tables with sample data

```
CREATE PROCEDURE selCheckingAccount (@iID int) AS

SELECT
        Account.account_type, Account.id, Account.balance,
        CheckingAccount.minimum_balance
FROM
        Account, CheckingAccount
WHERE
        account_id = @iID AND Account.[id] = @iID

GO

CREATE PROCEDURE selSavingsAccount (@iID int) AS

SELECT
        Account.account_type, Account.id, Account.balance,
        SavingsAccount.amount_interest_paid, SavingsAccount.date_interest_paid
FROM
        Account, SavingsAccount
WHERE
        account_id = @iID AND Account.[id] = @iID
GO
```

Figure 4. Stored procedures to access CheckingAccount and SavingsAccount

Now, based on the type of the accounts we would like to execute one of these two stored procedures (or others that may be added in the future if new tables related to account

were introduced). Figure 5 shows the stored procedure to select an account, what I call as the Polymorphic Stored Procedure!

```
CREATE  PROCEDURE selAccount(@iID int) AS

DECLARE @sAccountType AS varchar(256)

SELECT
        @sAccountType = account_type
FROM
        Account
WHERE
        [id] = @iID

DECLARE @sStoredProc varchar(270)
SET @sStoredProc = 'sel' + @sAccountType

exec @sStoredProc @iID
GO
```

Figure 5. "Polymorphic Stored Procedure" to select an Account

The selAccount stored procedure first queries the Account table to find the type of an account. Then it prefixes the account type with the letters "sel" to decide which stored procedure it should call. The effect of running the following query is shown in Figure 6:
exec selAccount 1
exec selAccount 2

	account_type	id	balance	minimum_balance
1	CheckingAccount	1	1000	900

Result of exec selAccount 1

	account_type	id	balance	amount_interest_paid	date_interest_paid
1	SavingsAccount	2	1000	10	2003-03-31 00:00:00.000

Result of exec selAccount 2

Figure 6. Result of running two queries on the selAccount stored procedure

From Figure 6 we can see that the columns from CheckingAccount table are returned for the first query, while the columns from the SavingsAccount table are being returned for the second query. This of course is in addition to the columns of the Account table as well. The selAccount stored procedure appears to be polymorphic, isn't it? This approach, reminds me of the Factory Method pattern[2].

Dynamic Creation of Account Objects

Now we will look at the Java code that will dynamically create the object of the appropriate type. We will use reflection as a vehicle to apply the Abstract Factory pattern

in Java[2, 3]. A class AccountFactory issues a query to the selAccount stored procedure. It then creates the appropriate account object and asks that object to load the data. Let's understand this bottom up. The Account, CheckingAccount and SavingsAccount classes are shown below:

```
// Account.java

import java.sql.*;

public abstract class Account
{
	private int id;
	private double balance;

	protected Account() {}

	protected void load(ResultSet rset) throws SQLException
	{
		id = rset.getInt("id");
		balance = rset.getDouble("balance");
	}

	public String toString()
	{
		return "Account id: " + id + ", balance $" + balance;
	}
}


// CheckingAccount.java
import java.sql.*;

public class CheckingAccount extends Account
{
	private double minimumBalance;
	protected CheckingAccount() {}

	protected void load(ResultSet rset) throws SQLException
	{
		super.load(rset);
		minimumBalance = rset.getDouble("minimum_balance");
	}

	public String toString()
	{
		return "Checking" + super.toString() +
```

```
                        ", with minimum balance of $" + minimumBalance;
        }
}

// SavingsAccount.java
import java.sql.*;
import java.util.*;
import java.util.Date;

public class SavingsAccount extends Account
{
        private double interestAmount;
        private Date interestDate;

        protected SavingsAccount() {}

        protected void load(ResultSet rset) throws SQLException
        {
                super.load(rset);
                interestAmount = rset.getDouble("amount_interest_paid");
                interestDate = rset.getDate("date_interest_paid");
        }

        public String toString()
        {
                return “Savings” + super.toString() +
                        ", Interest of $" + interestAmount + " paid on " +
                        interestDate;
        }
}
```

Note that the load method of Account is overridden by the load methods of CheckingAccount and SavingsAccount. Each of these methods deal with ResultSet which is populated with the data for the appropriate account by the AccountFactory as shown below:

```
// AccountFactory.java
import java.sql.*;

public class AccountFactory
{
        public Account get(int id)
        {
                Account acct = null;

                try
```

```
		{
			Connection connection = DBManager.getConnection();

			CallableStatement stmt =
				connection.prepareCall("{ call selAccount (?) }");

			stmt.setInt(1, id);
			ResultSet resultSet = stmt.executeQuery();
			if (resultSet.next())
			{
				String accountType =
					resultSet.getString("account_type");

				acct = (Account)
					Class.forName(accountType).newInstance();
				acct.load(resultSet);
			}
			resultSet.close();
			stmt.close();
		}
		catch (Exception e)
		{
			//Write appropriate error reporting code.
			System.out.println("Error: " + e);
		}

		return acct;
	}
}
```

The DBManager class above simply keeps a connection to the database as shown below:

```
//DBManager.java
import java.sql.*;

public class DBManager
{
	private static Connection connection = null;
	public static Connection getConnection()
	{
		if (connection == null)
		{
			try
			{
				Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
				connection = DriverManager.getConnection(
```

```
                                        "jdbc:odbc:InheritanceDataAccessDB");
                }
                catch (Exception e)
                {
                        System.out.println("Error: " + e);
                        System.exit(0);
                }
        }

        return connection;
    }

    public static void closeConnection() throws SQLException
    {
        if (connection != null)
        {
            connection.close();
            connection = null;
        }
    }
}
```

Test Case

Let's run a test case for the above code to see if this is going to work. The test code is given below.

```
public class TestCode
{
    public static void displayInfo(int id)
    {
        Account acct =
            new AccountFactory().get(id);

        System.out.println(acct);
    }

    public static void main(String[] args)
    {
        try
        {
            displayInfo(1);
            System.out.println("------------------");
            displayInfo(2);

            DBManager.closeConnection();
        }
        catch (Exception e)
```

```
                {
                        System.out.println(e);
                }
        }
}
```

The output from the above code is shown here:
CheckingAccount id: 1, balance $1000.0, with minimum balance of $900.0

SavingsAccount id: 2, balance $1000.0, Interest of $10.0 paid on 2003-03-31

Conclusion

Given a choice, we may want to utilize an OR mapping API/product to access the database from OO application. However, there are still applications out there that use JDBC or ASP.NET to access the data. This article shows how one could take advantage of Inheritance hierarchy by writing extensible code and stored procedures.

References

1. Robert C Martin, *The Open-Closed Principle*, C++ Report, 1996. http://www.objectmentor.com/resources/articles/ocp.pdf.
2. Erich Gamma, et. al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1994.
3. Ken Arnold, James Gosling, David Holmes, *The Java Programming Language, Third Edition*. Addison-Wesley, Boston, MA, 2000. ISBN:0-201-70433-1.