26

PROXY and STAIRWAY TO HEAVEN: Managing Third Party APIs

Does anybody remember laughter?

--Robert Plant, The Song Remains the Same.

There are many barriers in software systems. When we move data from our program into the database, we are crossing the database barrier. When we send a message from one computer to another we are crossing the network barrier.

Crossing these barriers can be complicated. If we aren't careful, our software will be more about the barriers than about the problem to be solved. The patterns in this chapter help us cross such barriers while keeping the program centered on the problem to be solved.

PROXY

Imagine that we are writing a shopping cart system for a website. Such a system might have objects for the customer, the order (the cart), and the products in the order. Figure 26-1 shows a possible structure. This structure is simplistic, but will serve for our purposes.

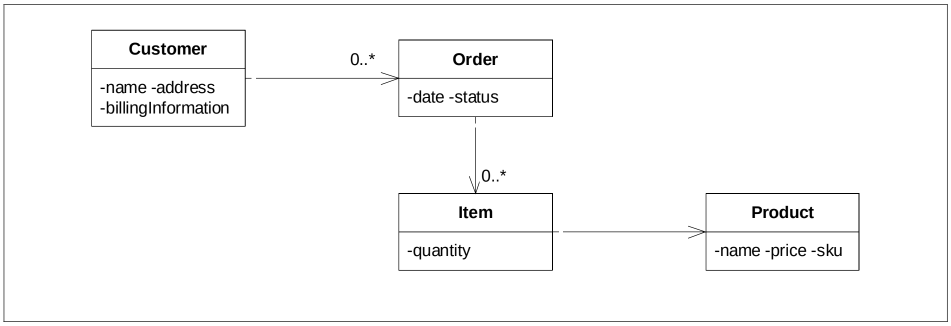

Figure 26-1
Simple shopping card object model.

If we consider the problem of adding a new item to an order, we might come up with the code in Listing 26-1. The `addItem` method of class `Order` simply creates a new `Item` holding the appropriate `Product` and quantity. It then adds that `Item` to its internal `Vector` of `Items`.

Listing 26-1
Adding an item to the Object Model.

```
public class Order
{
  private Vector itsItems = new Vector();
  public void addItem(Product p, int qty)
  {
    Item item = new Item(p, qty);
    itsItems.add(item);
  }
}
```

Now imagine that these objects represent data that are kept in a relational database. Figure 26-2 shows the tables and keys that might represent the objects. To find the orders for a given customer, you find all orders that have the customer's `cusid`. To find all the items in a given order, you find the items that have the order's `orderId`. To find the products referenced by the items, you use the product's `sku`.

If we want to add an item row for a particular order, we'd use something like Listing 26-2. This code makes JDBC calls to directly manipulate the relational data model.

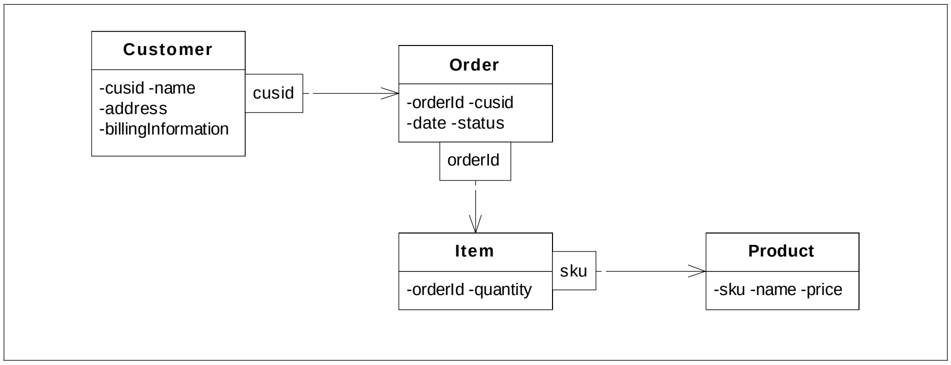

Figure 26-2
Shopping Card Relational Data Model

Listing 26-2
Adding an item to the relational model.

```
public class AddItemTransaction extends Transaction
{
  public void addItem(int orderId, String sku, int qty)
  {
    Statement s = itsConnection.CreateStatement();
    s.executeUpdate("insert into items values(" +
                    orderId + "," + sku + "," +
                    qty + ")");
  }
}
```

These two code snippets are very different, and yet they perform the same logical function. They both connect an item to an order. The first ignores the existence of a data base and the second glories in it.

Clearly the shopping cart program is all about orders, items, and products. Unfortunately, if we use the code in Listing 26-2 we make it about SQL statements, database connections and piecing together query strings. This is a significant violation of SRP and possibly the CCP. Listing 26-2 mixes together two concepts that change for different reasons. It mixes the concept of the items and orders with the concept of relational schemas and SQL. If either concept must change for any reason, the other concept will be affected. Listing 26-2 also violates the DIP since the policy of the program depends upon the details of the storage mechanism.

The PROXY pattern is a way to cure these ills. To explore this, lets set up a test program that demonstrates the behavior of creating an order and calculating the total price. The salient part of this program is shown in Listing 26-3

Listing 26-3
Test program creates order and verifies calculation of price.

```
public void testOrderPrice()
  {
    Order o = new Order("Bob");
    Product toothpaste = new Product("Toothpaste", 129);
```

Listing 26-3 (Continued)
Test program creates order and verifies calculation of price.

```
    o.addItem(toothpaste, 1);
    assertEquals(129, o.total());
    Product mouthwash = new Product("Mouthwash", 342);
    o.addItem(mouthwash, 2);
    assertEquals(813, o.total());
  }
```

The simple code that passes this test is shown in Listing 26-4 through Listing 26-6. It makes use of the simple object model in Figure 26-1. It does not assume that there is a database anywhere.

Listing 26-4
order.java

```
public class Order
{
  private Vector itsItems = new Vector();

  public Order(String cusid)
  {
  }

  public void addItem(Product p, int qty)
  {
    Item item = new Item(p,qty);
    itsItems.add(item);
  }

  public int total()
  {
    int total = 0;
    for (int i = 0; i < itsItems.size(); i++)
    {
      Item item = (Item) itsItems.elementAt(i);
      Product p = item.getProduct();
      int qty = item.getQuantity();
      total += p.getPrice() * qty;
    }
    return total;
  }
}
```

Listing 26-5
product.java

```
public class Product
{
  private int itsPrice;

  public Product(String name, int price)
  {
    itsPrice = price;
  }

  public int getPrice()
  {
```

Listing 26-5 (Continued)
product.java
```
    return itsPrice;
  }
}
```

Listing 26-6
item.java
```
public class Item
{
  private Product itsProduct;
  private int itsQuantity;

  public Item(Product p, int qty)
  {
    itsProduct = p;
    itsQuantity = qty;
  }

  public Product getProduct()
  {
    return itsProduct;
  }

  public int getQuantity()
  {
    return itsQuantity;
  }
}
```

Figure 26-3 and Figure 26-4 show how the PROXY pattern works. Each object that is to be proxied is split into three parts. The first is an interface that declares all the methods that clients will want to invoke. The second is an implementation that implements those methods without knowledge of the database. The third is the proxy that knows about the database.

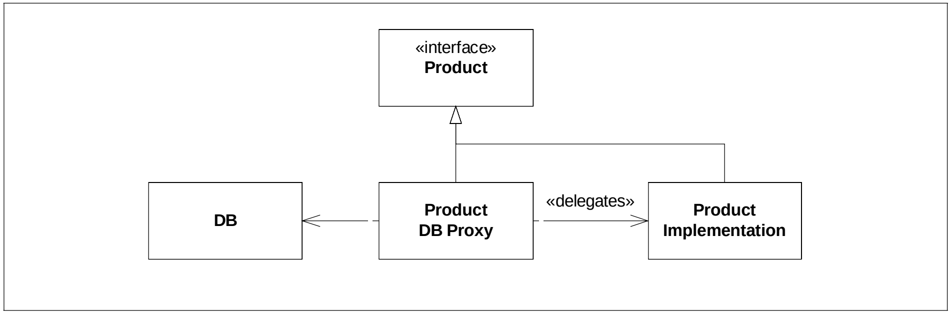

Figure 26-3
PROXY static model

Consider the `Product` class. We have proxied it by replacing it with an interface. This interface has all the same methods that `Product` has. The `ProductImplemen-`

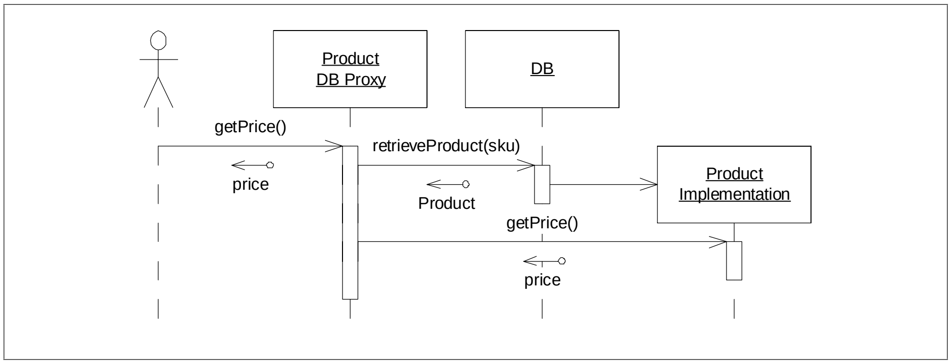

Figure 26-4
PROXY dynamic model

`tation` class implements the interface almost exactly as before. The `ProductDBProxy` implements all the methods of `Product` to fetch the product from the database, create an instance of `ProductImplementation` and then delegate the message to it.

The sequence diagram in Figure 26-4 shows how this works. The client sends the `getPrice` message to what it thinks is a `Product`, but what is really a `Product-DBProxy`. The `ProductDBProxy` fetches the `ProductImplementation` from the database. It then delegates the `getPrice` method to it.

Neither the client nor the `ProductImplementation` knows that this has happened. The database has been inserted into the application without either party knowing about it. That's the beauty of the PROXY pattern. In theory it can be inserted in between two collaborating objects without those objects having to know about it. Thus, it can be used to cross a barrier like a database or a network without either of the participants knowing about it.

In reality using proxies is non-trivial. To get an idea what some of the problems are, lets try to add the PROXY pattern to the simple shopping cart application.

Proxifying the Shopping Cart.

The simplest Proxy to create is for the `Product` class. For our purposes the product table represents a simple dictionary. It will be loaded in one place with all the products. There is no other manipulation of this table; and that makes the proxies relatively trivial.

To get started, we need a simple database utility that stores and retrieves product data. The proxy will use this interface to manipulate the database. Listing 26-7 shows the test program for what I have in mind. Listing 26-8 and Listing 26-9 make that test pass.

Listing 26-7
DBTest.java

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class DBTest extends TestCase
```

Listing 26-7 (Continued)
DBTest.java

```
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"DBTest"});
  }

  public DBTest(String name)
  {
    super(name);
  }

  public void setUp() throws Exception
  {
    DB.init();
  }

  public void tearDown() throws Exception
  {
    DB.close();
  }

  public void testStoreProduct() throws Exception
  {
    ProductData storedProduct = new ProductData();
    storedProduct.name = "MyProduct";
    storedProduct.price = 1234;
    storedProduct.sku = "999";
    DB.store(storedProduct);
    ProductData retrievedProduct = DB.getProductData("999");
    DB.deleteProductData("999");
    assertEquals(storedProduct, retrievedProduct);
  }
}
```

Listing 26-8
ProductData.java

```
public class ProductData
{
  public String name;
  public int price;

  public ProductData()
  {
  }

  public ProductData(String name, int price, String sku)
  {
    this.name = name;
    this.price = price;
    this.sku = sku;
  }

  public String sku;

  public boolean equals(Object o)
```

Listing 26-8 (Continued)
`ProductData.java`

```
  {
    ProductData pd = (ProductData)o;
    return name.equals(pd.name) &&
           sku.equals(pd.sku) &&
           price==pd.price;
  }
}
```

Listing 26-9
`DB.java`

```
import java.sql.*;

public class DB
{
  private static Connection con;

  public static void init() throws Exception
  {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    con = DriverManager.getConnection(
       "jdbc:odbc:PPP Shopping Cart");
  }

  public static void store(ProductData pd) throws Exception
  {
    PreparedStatement s = buildInsertionStatement(pd);
    executeStatement(s);
  }

  private static PreparedStatement
  buildInsertionStatement(ProductData pd) throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
      "INSERT into Products VALUES (?, ?, ?)");
    s.setString(1, pd.sku);
    s.setString(2, pd.name);
    s.setInt(3, pd.price);
    return s;
  }

  public static ProductData getProductData(String sku)
  throws Exception
  {
    PreparedStatement s = buildProductQueryStatement(sku);
    ResultSet rs = executeQueryStatement(s);
    ProductData pd = extractProductDataFromResultSet(rs);
    rs.close();
    s.close();
    return pd;
  }

  private static PreparedStatement
  buildProductQueryStatement(String sku) throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
      "SELECT * FROM Products WHERE sku = ?;");
```

Listing 26-9 (Continued)
DB.java

```
    s.setString(1, sku);
    return s;
  }

  private static ProductData
  extractProductDataFromResultSet(ResultSet rs)
  throws SQLException
  {
    ProductData pd = new ProductData();
    pd.sku = rs.getString(1);
    pd.name = rs.getString(2);
    pd.price = rs.getInt(3);
    return pd;
  }

  public static void deleteProductData(String sku)
  throws Exception
  {
    executeStatement(buildProductDeleteStatement(sku));
  }

  private static PreparedStatement
  buildProductDeleteStatement(String sku) throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
      "DELETE from Products where sku = ?");
    s.setString(1, sku);
    return s;
  }

  private static void executeStatement(PreparedStatement s)
  throws SQLException
  {
    s.execute();
    s.close();
  }

  private static ResultSet
  executeQueryStatement(PreparedStatement s)
  throws SQLException
  {
    ResultSet rs = s.executeQuery();
    rs.next();
    return rs;
  }

  public static void close() throws Exception
  {
    con.close();
  }
}
```

The next step in implementing the proxy is to write a test that shows how it works. This test adds a product to the database. It then creates a `ProductProxy` with the `sku` of the stored product, and attempts to use the accessors of `Product` to acquire the data from the proxy. See Listing 26-10

Listing 26-10
`ProxyTest.java`

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class ProxyTest extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"ProxyTest"});
  }

  public ProxyTest(String name)
  {
    super(name);
  }

  public void setUp() throws Exception
  {
    DB.init();
    ProductData pd = new ProductData();
    pd.sku = "ProxyTest1";
    pd.name = "ProxyTestName1";
    pd.price = 456;
    DB.store(pd);
  }

  public void tearDown() throws Exception
  {
    DB.deleteProductData("ProxyTest1");
    DB.close();
  }

  public void testProductProxy() throws Exception
  {
    Product p = new ProductProxy("ProxyTest1");
    assertEquals(456, p.getPrice());
    assertEquals("ProxyTestName1", p.getName());
    assertEquals("ProxyTest1", p.getSku());
  }
}
```

In order to make this work we have to separate the interface of `Product` from its implementation. So I changed `Product` to an interface and created `ProductImp` to implement it (See Listing 26-11 and Listing 26-12). This forced me to make changes to `TestShoppingCart` (not shown) to use `ProductImp` instead of `Product`.

Notice that I have added exceptions to the `Product` interface. This is because I was writing `ProductProxy` (Listing 26-13) at the same time as I was writing `Product`, `ProductImp`, and `ProxyTest`. I implemented them all one accessor at a time. As we will see, the `ProductProxy` class invokes the database which throws exceptions. I did not want those exceptions to be caught and hidden by the proxy, so I decided to let them escape from the interface.

Listing 26-11
`Product.java`

```
public interface Product
{
  public int getPrice() throws Exception;
  public String getName() throws Exception;
  public String getSku() throws Exception;
}
```

Listing 26-12
`ProductImp.java`

```
public class ProductImp implements Product
{
  private int itsPrice;
  private String itsName;
  private String itsSku;

  public ProductImp(String sku, String name, int price)
  {
    itsPrice = price;
    itsName = name;
    itsSku = sku;
  }

  public int getPrice()
  {
    return itsPrice;
  }

  public String getName()
  {
    return itsName;
  }

  public String getSku()
  {
    return itsSku;
  }
}
```

Listing 26-13
`ProductProxy.java`

```
public class ProductProxy implements Product
{
  private String itsSku;
  public ProductProxy(String sku)
  {
    itsSku = sku;
  }
  public int getPrice() throws Exception
  {
    ProductData pd = DB.getProductData(itsSku);
    return pd.price;
```

Listing 26-13 (Continued)
ProductProxy.java

```
  }

  public String getName() throws Exception
  {
    ProductData pd = DB.getProductData(itsSku);
    return pd.name;
  }

  public String getSku() throws Exception
  {
    return itsSku;
  }
}
```

The implementation of this proxy is trivial. In fact, it doesn't quite match the canonical form of the pattern shown in Figure 26-3 and Figure 26-4. This was an unexpected surprise. My intent was to implement the PROXY pattern. But when the implementation finally materialized, the canonical pattern made no sense.

As shown below, the canonical pattern would have had `ProductProxy` create a `ProductImp` in every method. It would then have delegated that method to the `ProductImp`.

```
public int getPrice() throws Exception
{
  ProductData pd = DB.getProductData(itsSku);
  ProductImp p = new ProductImp(pd.sku, pd.name, pd.price);
  return p.getPrice();
}
```

The creation of the `ProductImp` is a complete waste of programmer and computer resources. The `ProductProxy` already has the data that the `ProductImp` accessors would return. So there is no need to create, and then delegate to, the `ProductImp`. This is yet another example of how the code may lead you away from the patterns and models you expected.

Notice that the `getSku` method of `ProductProxy` in Listing 26-13 takes this theme one step further. It doesn't even bother to hit the database for the `sku`. Why should it? It already has the `sku`.

You might be thinking that the implementation of `ProductProxy` is very inefficient. It hits the database for each accessor. wouldn't it be better if it cached the `ProductData` item in order to avoid hitting the database?

This change is trivial; but the only thing driving us to do it is our fear. At this point we have no data to suggest that this program has a performance problem. And besides, we know the database engine is doing some caching too. So it's not clear what building our own cache would buy us. We should wait until we see indications of a performance problem before we invent trouble for ourselves.

Proxyifying Relationships. Our next step is to create the proxy for `Order`. Each `Order` instance contains many `Item` instances. In the relational schema (Figure 26-2) this relationship is captured within the `Item` table. Each row of the `Item` table contains the key of the `Order` that contains it. In the object model, however, the relationship is implemented by a `Vector` within `Order` (See Listing 26-4). Somehow the proxy is going to have to translate between the two forms.

We begin by posing a test case that the proxy must pass. This test adds a few dummy products to the database. It then obtains proxies to those products, and uses them to invoke `addItem` on an `OrderProxy`. Finally, it asks the `OrderProxy` for the total price (See Listing 26-14). The intent of this test case is to show that an `OrderProxy` behaves just like an `Order`, but that it obtains its data from the database instead of from in-memory objects.

Listing 26-14
ProxyTest.java

```
public void testOrderProxyTotal() throws Exception
  {
    DB.store(new ProductData("Wheaties", 349, "wheaties"));
    DB.store(new ProductData("Crest", 258, "crest"));
    ProductProxy wheaties = new ProductProxy("wheaties");
    ProductProxy crest = new ProductProxy("crest");
    OrderData od = DB.newOrder("testOrderProxy");
    OrderProxy order = new OrderProxy(od.orderId);
    order.addItem(crest, 1);
    order.addItem(wheaties, 2);
    assertEquals(956, order.total());
  }
```

In order to make this test case work, we have to implement a few new classes and methods. The first we'll tackle is the `newOrder` method of `DB`. It looks like this method returns an instance of something called an `OrderData`. `OrderData` is just like `Product-Data`. It is a simple data structure that represents a row of the `Order` database table. It is shown in Listing 26-15.

Listing 26-15
OrderData.java

```
public class OrderData
{
  public String customerId;
  public int orderId;

  public OrderData()
  {
  }

  public OrderData(int orderId, String customerId)
  {
    this.orderId = orderId;
    this.customerId = customerId;
  }
}
```

Don't be offended by the use of public data members. This is not an object in the true sense. It is just a container for data. It has no interesting behavior that needs to be encapsulated. Making the data variables private, and providing getters and setters would just be a waste of time.

Now we need to write the `newOrder` function of `DB`. Notice that when we call it in Listing 26-14, we provide the id of the owning customer; but we do not provide the `orderId`. Each `Order` needs an `orderId` to act as its key. What's more, in the relational schema, each `Item` refers to this `orderId` as a way to show its connection to the `Order`. Clearly the `orderId` must be unique. How does it get created? Lets write a test to show our intent. See Listing 26-16.

Listing 26-16
DBTest.java

```
public void testOrderKeyGeneration() throws Exception
{
  OrderData o1 = DB.newOrder("Bob");
  OrderData o2 = DB.newOrder("Bill");
  int firstOrderId = o1.orderId;
  int secondOrderId = o2.orderId;
  assertEquals(firstOrderId+1, secondOrderId);
}
```

This test shows that we expect the `orderId` to somehow automatically increment every time a new `Order` is created. This is easily implemented by querying the database for the maximum `orderId` currently in use, and then adding one to it. See Listing 26-17

Listing 26-17
DB.java

```
  public static OrderData newOrder(String customerId)
  throws Exception
  {
    int newMaxOrderId = getMaxOrderId() + 1;
    PreparedStatement s = con.prepareStatement(
      "Insert into Orders(orderId,cusid) Values(?,?);");
    s.setInt(1, newMaxOrderId);
    s.setString(2,customerId);
    executeStatement(s);
    return new OrderData(newMaxOrderId, customerId);
  }

  private static int getMaxOrderId() throws SQLException
  {
    Statement qs = con.createStatement();
    ResultSet rs = qs.executeQuery(
      "Select max(orderId) from Orders;");
    rs.next();
    int maxOrderId = rs.getInt(1);
    rs.close();
    return maxOrderId;
  }
```

Now we can start to write `OrderProxy`. As with `Product,` we need to split `Order` into an interface and an implementation. So `Order` becomes the interface and `OrderImp` becomes the implementation. See Listing 26-18 and Listing 26-19.

Listing 26-18
`Order.java`

```
public interface Order
{
  public String getCustomerId();
  public void addItem(Product p, int quantity);
  public int total();
}
```

Listing 26-19
`OrderImp.java`

```
import java.util.Vector;

public class OrderImp implements Order
{
  private Vector itsItems = new Vector();
  private String itsCustomerId;

  public String getCustomerId()
  {
    return itsCustomerId;
  }

  public OrderImp(String cusid)
  {
    itsCustomerId = cusid;
  }

  public void addItem(Product p, int qty)
  {
    Item item = new Item(p,qty);
    itsItems.add(item);
  }

  public int total()
  {
    try
    {
      int total = 0;
      for (int i = 0; i < itsItems.size(); i++)
      {
        Item item = (Item) itsItems.elementAt(i);
        Product p = item.getProduct();
        int qty = item.getQuantity();
        total += p.getPrice() * qty;
      }
      return total;
    }
    catch (Exception e)
    {
      throw new Error(e.toString());
    }
```

Listing 26-19 (Continued)
OrderImp.java

```
  }
}
```

I had to add some exception processing to `OrderImp` because the `Product` interface throws exceptions. I'm getting frustrated with all these exceptions. The implementations of proxies behind an interface should not have an effect upon that interface; and yet the proxies are throwing exceptions that propagate out through the interface. So, I resolve to change all the `Exceptions` to `Errors` so that I don't have to pollute the interfaces with `throws` clauses, and the users of those interfaces with `try/catch` blocks.

How do I implement `addItem` in the proxy? Clearly the proxy cannot delegate to `OrderImp.addItem`! Rather, the proxy is going to have to insert an `Item` row in the database. On the other hand, I *really want* to delegate `OrderProxy.total` to `Order-Imp.total`, because I want the business rules, i.e. the policy for creating totals, to be encapsulated in `OrderImp`. The whole point of building proxies is to separate database implementation from business rules.

In order to delegate the `total` function, the proxy is going to have to build the complete `Order` object along with all its contained `Items`. Thus, in `OrderProxy`.total we are going to have to read in all the items from the database, call `addItem` on an empty `OrderImp` for each item we find. And then call `total` on that `OrderImp`. Thus, the `OrderProxy` implementation ought to look something like Listing 26-20.

Listing 26-20
OrderProxy.java

```
import java.sql.SQLException;

public class OrderProxy implements Order
{
  private int orderId;

  public OrderProxy(int orderId)
  {
    this.orderId = orderId;
  }

  public int total()
  {
    try
    {
      OrderImp imp = new OrderImp(getCustomerId());
      ItemData[] itemDataArray = DB.getItemsForOrder(orderId);
      for (int i = 0; i < itemDataArray.length; i++)
      {
        ItemData item = itemDataArray[i];
        imp.addItem(new ProductProxy(item.sku), item.qty);
      }
      return imp.total();
    }
    catch (Exception e)
    {
```

Listing 26-20 (Continued)
OrderProxy.java

```
      throw new Error(e.toString());
    }
  }

  public String getCustomerId()
  {
    try
    {
      OrderData od = DB.getOrderData(orderId);
      return od.customerId;
    }
    catch (SQLException e)
    {
      throw new Error(e.toString());
    }
  }

  public void addItem(Product p, int quantity)
  {
    try
    {
      ItemData id =
        new ItemData(orderId, quantity, p.getSku());
      DB.store(id);
    }
    catch (Exception e)
    {
      throw new Error(e.toString());
    }
  }

  public int getOrderId()
  {
    return orderId;
  }
}
```

This implies the existence of an `ItemData` class, and a few `DB` functions for manipulating `ItemData` rows. These are shown in Listing 26-21 through Listing 26-23

Listing 26-21
ItemData.java

```
public class ItemData
{
  public int orderId;
  public int qty;
  public String sku = "junk";

  public ItemData()
  {
  }

  public ItemData(int orderId, int qty, String sku)
  {
    this.orderId = orderId;
```

Listing 26-21 (Continued)
`ItemData.java`

```
    this.qty = qty;
    this.sku = sku;
  }

  public boolean equals(Object o)
  {
    ItemData id = (ItemData)o;
    return orderId == id.orderId &&
           qty == id.qty &&
           sku.equals(id.sku);
  }
}
```

Listing 26-22
`DBTest.java`

```
  public void testStoreItem() throws Exception
  {
    ItemData storedItem = new ItemData(1, 3, "sku");
    DB.store(storedItem);
    ItemData[] retrievedItems = DB.getItemsForOrder(1);
    assertEquals(1, retrievedItems.length);
    assertEquals(storedItem, retrievedItems[0]);
  }

  public void testNoItems() throws Exception
  {
    ItemData[] id = DB.getItemsForOrder(42);
    assertEquals(0, id.length);
  }
```

Listing 26-23
`DB.java`

```
  public static void store(ItemData id) throws Exception
  {
    PreparedStatement s = buildItemInsersionStatement(id);
    executeStatement(s);
  }

  private static PreparedStatement
  buildItemInsersionStatement(ItemData id) throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
      "Insert into Items(orderId,quantity,sku) " +
      "VALUES (?, ?, ?);");
    s.setInt(1,id.orderId);
    s.setInt(2,id.qty);
    s.setString(3, id.sku);
    return s;
  }

  public static ItemData[] getItemsForOrder(int orderId)
  throws Exception
  {
    PreparedStatement s =
      buildItemsForOrderQueryStatement(orderId);
```

Listing 26-23 (Continued)
DB.java

```
    ResultSet rs = s.executeQuery();
    ItemData[] id = extractItemDataFromResultSet(rs);
    rs.close();
    s.close();
    return id;
  }

  private static PreparedStatement
  buildItemsForOrderQueryStatement(int orderId)
  throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
      "SELECT * FROM Items WHERE orderid = ?;");
    s.setInt(1, orderId);
    return s;
  }

  private static ItemData[]
  extractItemDataFromResultSet(ResultSet rs)
  throws SQLException
  {
    LinkedList l = new LinkedList();
    for (int row = 0; rs.next(); row++)
    {
      ItemData id = new ItemData();
      id.orderId = rs.getInt("orderid");
      id.qty = rs.getInt("quantity");
      id.sku = rs.getString("sku");
      l.add(id);
    }
    return (ItemData[]) l.toArray(new ItemData[l.size()]);
  }

  public static OrderData getOrderData(int orderId)
  throws SQLException
  {
    PreparedStatement s = con.prepareStatement(
      "Select cusid from orders where orderid = ?;");
    s.setInt(1, orderId);
    ResultSet rs = s.executeQuery();
    OrderData od = null;
    if (rs.next())
      od =  new OrderData(orderId, rs.getString("cusid"));
    rs.close();
    s.close();
    return od;
  }
```

Summary of PROXY

This example should have dispelled any false illusions about the elegance and simplicity of using proxies. Proxies are not trivial to use. The simple delegation model implied by the canonical pattern seldom materializes so neatly. Rather, we find ourselves short circuiting the delegation for trivial getters and setters. For methods that manage `1:N` relationships, we find ourselves *delaying* the delegation and moving it into other methods, just as the delegation for `addItem` was moved into `total`. Finally we face the specter of caching.

We didn't do any caching in this example. The tests all run in less than a second, so there was no need to worry overmuch about performance. But in a real application the issue of performance, and the need for intelligent caching is likely to arise. I do not suggest that you automatically implement a caching strategy because you fear performance will otherwise be too slow. Indeed, I have found that adding caching too early is a very good way to *decrease* performance. Rather, if you fear performance may be a problem, I recommend that you conduct some experiments to *prove* that it will be a problem. Once proven, and *only once proven*, you should start considering how to speed things up.

The benefit of Proxy. For all the troublesome nature of proxies, they have one very powerful benefit: *the separation of concerns*. In our example the business rules and the database have been completely separated. `OrderImp` has no dependence whatever on the database. If we want to change the database schema, or change the database engine, we can do so without affecting `Order`, `OrderImp`, or any of the other business domain classes.

In those instances where separation of business rules from database implementation is critically important, PROXY can be a good pattern to employ. For that matter, PROXY can be used to separate business rules from *any* kind of implementation issue. It can be used to keep the business rules from being polluted by such things as COM, CORBA, EJB, etc. It is a way to keep the business rule assets of your project separate from the implementation mechanisms that are currently in vogue.

Dealing with Databases, Middleware, and other Third Party interfaces.

Third party APIs are a fact of life for software engineers. We buy database engines, middleware engines, class libraries, threading libraries, etc. Initially we use these APIs by making direct calls to them from our application code (See Figure 26-5).

Over time, however, we find that our application code becomes more and more polluted with such API calls. In a database application, for example, we may find more and more SQL strings littering the code that also contains the business rules.

This becomes a problem when the third party API changes. For databases it also becomes a problem when the schema changes. As new versions of the API or Schema are released, more and more of the application code has to be reworked to align with those changes.

Eventually the developers decide that they must insulate themselves from these changes. So they invent a layer that separates the application business rules from the third party API (See Figure 26-6). They concentrate into this layer all the code that uses the

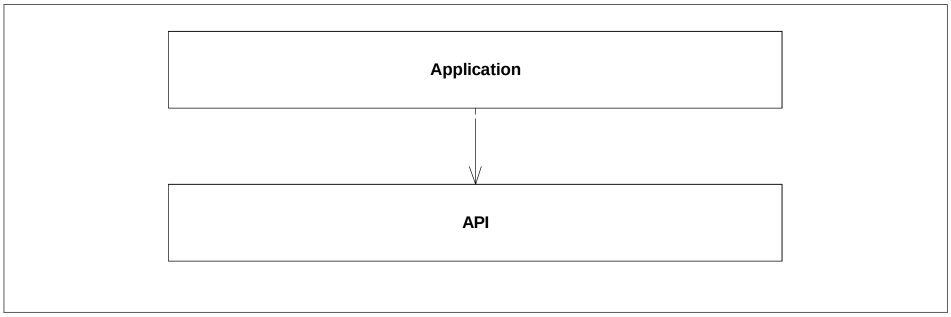

Figure 26-5
Initial relationship between an application and a third party API

third party API, and all of the concepts that related to the API rather than to the business rules of the application.

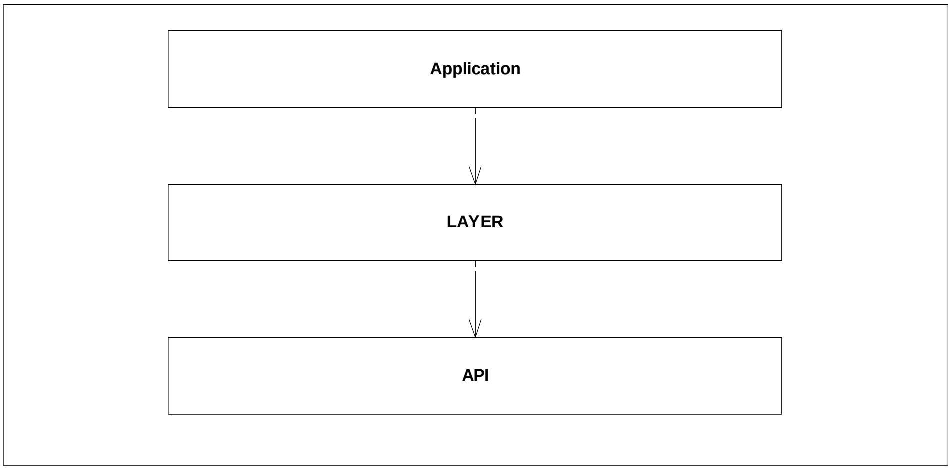

Figure 26-6
Introducing an insulation layer

Such layers can sometimes be purchased. ODBC or JDBC are such layers. They separate the application code from the actual database engine. Of course they are also third party APIs in and of themselves, and therefore the application may need to be insulated even from them.

Notice that there is a transitive dependency from the `Application` to the `API`. In some applications that indirect dependence is still enough to cause problems. JDBC, for example, does not insulate the application from the details of the schema.

In order to attain even better insulation, we need to invert the dependency between the Application and the Layer (See Figure 26-7). This keeps the application from knowing anything at all about the third party API, either directly or indirectly. In the case of a database, it keeps the application from direct knowledge of the schema. In the case of a middleware engine, it keeps the application from knowing anything about the datatypes used by that middleware processor.

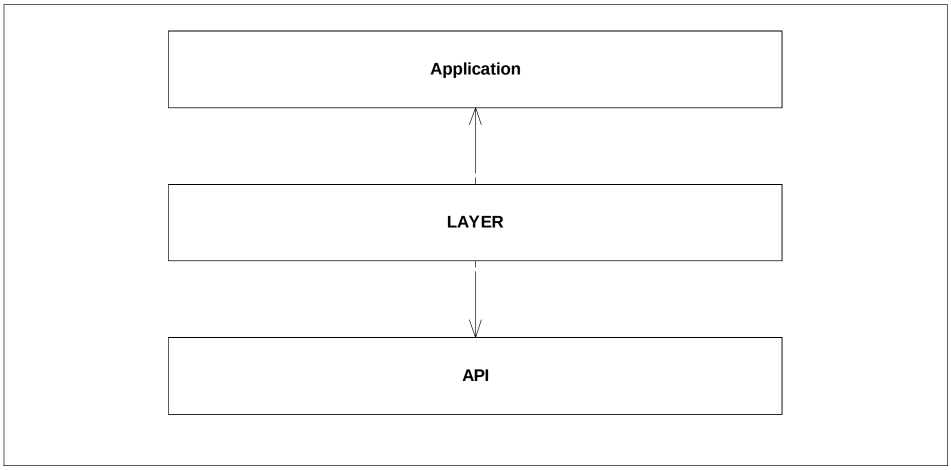

Figure 26-7
Inverting the dependency between the Application and the Layer

This arrangement of dependencies is precisely what the PROXY pattern achieves. The application does not depend upon the proxies at all. Rather the proxies depend upon the application, and upon the API. This concentrates all knowledge of the mapping between the application and the API into the proxies.

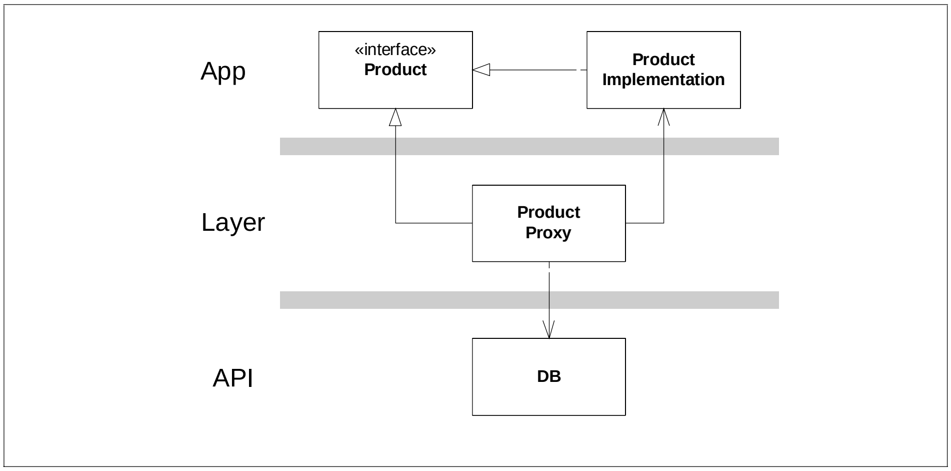

Figure 26-8
How the Proxy inverts the dependency between the Application and the Layer.

This concentration of knowledge means that the proxies are nightmares. Whenever the API changes, the proxies change. Whenever the application changes the proxies change. The proxies can become very hard to deal with.

It's good to know where your nightmares live. Without the proxies, the nightmares would be spread throughout the application code.

Most applications don't need proxies. Proxies are a very heavy weight solution. When I see proxy solutions in use, my recommendation in most cases is to take them out and use something simpler. But there are cases when the intense separation between the application and the API afforded by proxies is beneficial. Those cases are almost always in very large systems that undergo frequent schema and/or API thrashing. Or in systems that can ride on top of many different database engines or middleware engines.

STAIRWAY TO HEAVEN[1]

STAIRWAY TO HEAVEN is another pattern that achieves the same dependency inversion as PROXY. It employs a variation on the class form of the ADAPTER pattern. See Figure 26-9.

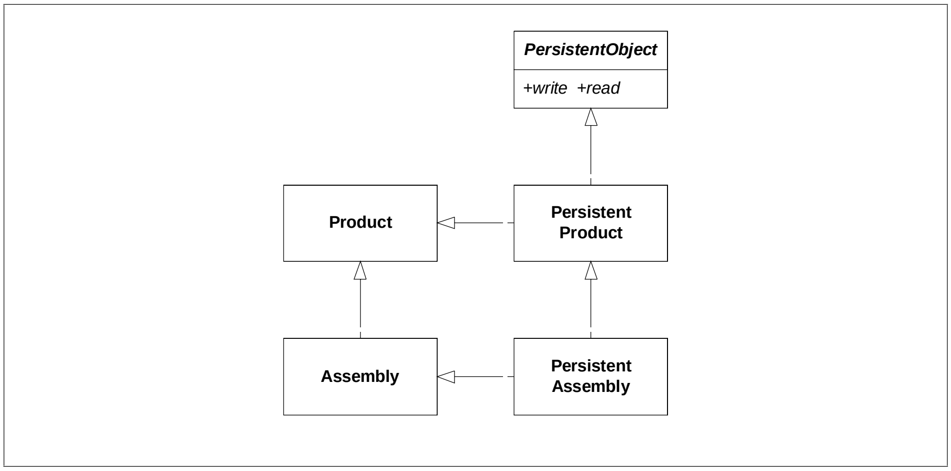

Figure 26-9
Stairway to Heaven

`PersistentObject` is an abstract class that knows about the database. It provides two abstract methods: `read` and `write`. It also provides a set of implemented methods that provide the tools needed to implement `read` and `write`. `PersistentProduct`, for example, uses these tools to implement `read` and `write` to read and write all the data

1. [Martin97]

fields of Product from and to the database. By the same token, PersistentAssembly implements read and write to do the same for the extra fields within Assembly. It inherits the ability to read and write the fields of Product from PersistentProduct and structures the read and write methods so as to take advantage of that fact.

This pattern is only useful in languages that support multiple inheritance. Note that both PersistentProduct and PersistentAssembly inherit from two implemented base classes. What's more, PersistentAssembly finds itself in a *diamond* inheritance relationship with Product. In C++ we would have to use virtual inheritance to prevent two instances of Product from being inherited into PersistentAssembly.

The need for virtual inheritance, or similar relationships in other languages, means that this pattern is somewhat intrusive. It makes itself felt in the Product hierarchy. But the intrusion is minimal.

The benefit of this pattern is that it completely separates knowledge of the database away from the business rules of the application. Those small bits of the application that need to invoke read and write can do so through the following exigency:

```
PersistentObject* o = dynamic_cast<PersistentObject*>(product);
if (o)
  o->write();
```

In other words, we ask the application object if it conforms to the Persistent-Object interface; and if so we invoke either read or write. This keeps that part of the application that does not need to know about reading and writing completely independent of the PersistentObject side of the hierarchy.

Example of STAIRWAY TO HEAVEN

Listing 26-24 through Listing 26-34 show an example of STAIRWAY TO HEAVEN in C++. As usual, it is best to start with the test case. CppUnit[1] is a bit wordy if shown in its entirety, so I have only included the test case methods in Listing 26-24. The first test case verifies that a PersistentProduct can be passed around the system as a Product, and then converted to a PersistentObject and written at will. We assume that the PersistentProduct will write itself in a simple XML format. The second test case verifies the same for PersistentAssembly, the only difference being the addition of a second field in the Assembly object.

Listing 26-24
productPersistenceTestCase.cpp {abridged}

```
void ProductPersistenceTestCase::testWriteProduct()
{
  ostrstream s;
  Product* p = new PersistentProduct("Cheerios");
  PersistentObject* po = dynamic_cast<PersistentObject*>(p);
  assert(po);
```

1. One of the XUnit family of unit test frameworks. See www.junit.org, and www.xprogramming.com for more information.

Listing 26-24 (Continued)
productPersistenceTestCase.cpp {abridged}

```
  po->write(s);
  char* writtenString = s.str();
  assert(strcmp("<PRODUCT><NAME>Cheerios</NAME></PRODUCT>",
                writtenString) == 0);
}

void ProductPersistenceTestCase::testWriteAssembly()
{
  ostrstream s;
  Assembly* a = new PersistentAssembly("Wheaties", "7734");
  PersistentObject* po = dynamic_cast<PersistentObject*>(a);
  assert(po);
  po->write(s);
  char* writtenString = s.str();
  assert(strcmp("<ASSEMBLY><NAME>Wheaties"
                "</NAME><ASSYCODE>7734</ASSYCODE></ASSEMBLY>",
                 writtenString) == 0);

}
```

Next, in Listing 26-25 through Listing 26-28 we see the definitions and implementations of both `Product` and `Assembly`. In the interest of saving space in our example, these classes are nearly degenerate. In a normal application these classes would contain methods that implemented business rules. Note that there is no hint of persistence in either of these classes. There is no dependence whatever from the business rules to the persistence mechanism. This is the whole point of the pattern.

While the dependency characteristics are good, there is an artifact in Listing 26-27 that is present solely because of the STAIRWAY TO HEAVEN pattern. `Assembly` inherits from `Product` using the `virtual` keyword. This is necessary in order to prevent duplicate inheritance of `Product` in `PersistentAssembly`. If you refer back to Figure 26-9 you'll see that `Product` is the apex of a diamond[1] of inheritance involving `Assembly`, `PersistentProduct`, and `PersistentObject`. To prevent duplicate inheritance of `Product` it must be inherited virtually.

Listing 26-25
product.h

```
#ifndef STAIRWAYTOHEAVENPRODUCT_H
#define STAIRWAYTOHEAVENPRODUCT_H

#include <string>

class Product
{
 public:
  Product(const string& name);
  virtual ~Product();
  const string& getName() const {return itsName;}
 private:
```

1. Sometimes facetiously known as the Deadly Diamond of Death.

Listing 26-25 (Continued)
`product.h`

```
  string itsName;
};

#endif
```

Listing 26-26
`product.cpp`

```
#include "product.h"

Product::Product(const string& name)
  : itsName(name)
{
}

Product::~Product()
{
}
```

Listing 26-27
`assembly.h`

```
#ifndef STAIRWAYTOHEAVENASSEMBLY_H
#define STAIRWAYTOHEAVENASSEMBLY_H

#include <string>
#include "product.h"

class Assembly : public virtual Product
{
 public:
  Assembly(const string& name, const string& assyCode);
  virtual ~Assembly();

  const string& getAssyCode() const {return itsAssyCode;}
 private:
  string itsAssyCode;
};

#endif
```

Listing 26-28
`assembly.cpp`

```
#include "assembly.h"

Assembly::Assembly(const string& name, const string& assyCode)
  :Product(name), itsAssyCode(assyCode)
{
}

Assembly::~Assembly()
{
}
```

Listing 26-29 and Listing 26-30 show the definition and implementation of `PersistentObject`. Note that while `PersistentObject` knows nothing of the

`Product` hierarchy, it does seem to know something about how to write XML. At least it understands that objects are written by writing a header, followed by the fields, followed by a footer.

The `write` method of `PersistentObject` uses the TEMPLATE METHOD[1] pattern to control the writing of all its derivatives. Thus the persistent side of the STAIRWAY TO HEAVEN pattern makes use of the facilities of the `PersistentObject` base class.

Listing 26-29
`persistentObject.h`

```
#ifndef STAIRWAYTOHEAVENPERSISTENTOBJECT_H
#define STAIRWAYTOHEAVENPERSISTENTOBJECT_H

#include <iostream>

class PersistentObject
{
 public:
  virtual ~PersistentObject();
  virtual void write(ostream&) const;

 protected:
  virtual void writeFields(ostream&) const = 0;

 private:
  virtual void writeHeader(ostream&) const = 0;
  virtual void writeFooter(ostream&) const = 0;
};

#endif
```

Listing 26-30
`persistentObject.cpp`

```
#include "persistentObject.h"

PersistentObject::~PersistentObject()
{
}

void PersistentObject::write(ostream& s) const
{
  writeHeader(s);
  writeFields(s);
  writeFooter(s);
  s << ends;
}
```

Listing 26-31 and Listing 26-32 show the implementation of `PersistentProduct`. This class implements the `writeHeader`, `writeFooter`, and `writeField` functions to create the appropriate XML for a `Product`. It inherits the fields and accessors from `Product`, and is driven by the write method of its base class `PersistentObject`.

1. See Chapter 14: *Template Method & Strategy: Inheritance vs. Delegation*, on page 193.

Listing 26-31
persistentProduct.h

```
#ifndef STAIRWAYTOHEAVENPERSISTENTPRODUCT_H
#define STAIRWAYTOHEAVENPERSISTENTPRODUCT_H

#include "product.h"
#include "persistentObject.h"

class PersistentProduct : public virtual Product
                        , public PersistentObject
{
 public:
  PersistentProduct(const string& name);
  virtual ~PersistentProduct();

 protected:
  virtual void writeFields(ostream& s) const;

 private:
  virtual void writeHeader(ostream& s) const;
  virtual void writeFooter(ostream& s) const;
};

#endif
```

Listing 26-32
persistentProduct.cpp

```
#include "persistentProduct.h"

PersistentProduct::PersistentProduct(const string& name)
:Product(name)
{
}

PersistentProduct::~PersistentProduct()
{
}

void PersistentProduct::writeHeader(ostream& s) const
{
  s << "<PRODUCT>";
}

void PersistentProduct::writeFooter(ostream& s) const
{
  s << "</PRODUCT>";
}

void PersistentProduct::writeFields(ostream& s) const
{
  s << "<NAME>" << getName() << "</NAME>";
}
```

Finally, Listing 26-33 and Listing 26-34 show how `PersistentAssemly` unifies `Assembly` and `PersistentProduct`. Just like `PersistentProduct` it overrides `writeHeader`, `writeFooter`, and `writeFields`. However, it implements `write-`

Fields to invoke PersistentProduct::writeFields. Thus it inherits the ability to write the Product part of Assembly from PersistentProduct; and inherits the Product and Assembly fields and accessors from Assembly.

Listing 26-33
persistentAssembly.h

```
#ifndef STAIRWAYTOHEAVENPERSISTENTASSEMBLY_H
#define STAIRWAYTOHEAVENPERSISTENTASSEMBLY_H

#include "assembly.h"
#include "persistentProduct.h"

class PersistentAssembly : public Assembly, public
PersistentProduct
{
 public:
  PersistentAssembly(const string& name,
                     const string& assyCode);
  virtual ~PersistentAssembly();

 protected:
  virtual void writeFields(ostream& s) const;

 private:
  virtual void writeHeader(ostream& s) const;
  virtual void writeFooter(ostream& s) const;
};

#endif
```

Listing 26-34
persistentAssembly.cpp

```
#include "persistentAssembly.h"

PersistentAssembly::PersistentAssembly(const string& name,
const string& assyCode)
: Assembly(name, assyCode)
, PersistentProduct(name)
, Product(name)
{
}

PersistentAssembly::~PersistentAssembly()
{
}

void PersistentAssembly::writeHeader(ostream& s) const
{
  s << "<ASSEMBLY>";
}

void PersistentAssembly::writeFooter(ostream& s) const
{
  s << "</ASSEMBLY>";
}

void PersistentAssembly::writeFields(ostream& s) const
```

Listing 26-34 (Continued)
persistentAssembly.cpp

```
{
  PersistentProduct::writeFields(s);
  s << "<ASSYCODE>" << getAssyCode() << "</ASSYCODE>";
}
```

Conclusion. I've seen STAIRWAY TO HEAVEN used in many different scenarios with good results. The pattern is relatively easy to set up, and has a minimum impact on the objects that contain the business rules. On the other hand, it requires a language, like C++, that supports multiple inheritance of implementation.

OTHER PATTERNS THAT CAN BE USED WITH DATABASES

EXTENSION OBJECT. Imagine an Extension Object[1] that knows how to write the extended object on a database. In order to write such an object you would ask it for an Extension Object that matched the "Database" key, cast it to a `DatabaseWriterExtension`, and then invoke the `write` function.

```
Product p = /* some function that returns a Product */
ExtensionObject e = p.getExtension("Database");
if (e)
{
  DatabaseWriterExtension dwe = (DatabaseWriterExtension) e;
  e.write();
}
```

VISITOR[2]. Imagine a visitor hierarchy that knows how to write the visited object on a database. You would write an object on the database by creating the appropriate type of visitor, and then calling `accept` on the object to be written.

```
Product p = /* some function that returns a Product */
DatabaseWriterVisitor dwv = new DatabaseWritierVisitor();
p.accept(dwv);
```

DECORATOR[3]. There are two ways to use a decorator to implement databases. You can decorate a business object and give it read and write methods; or you can decorate a data object that knows how to read and write itself and give it business rules. The latter approach is not uncommon when using Object Oriented Databases. The business rules are kept out of the OODB schema and added in with decorators.

FACADE. This is my favorite starting point. It's simple and effective. On the down side, it couples the business rule objects to the database. Figure 26-10 shows the structure. The `DatabaseFacade` class simply provides methods for reading and writing all the neces-

1. See "Extension Object" on page 498
2. See "VISITOR" on page 478
3. See "Decorator" on page 494

sary objects. This couples the objects to the `DatabaseFacade` and vice versa. The objects know about the facade because they are often the ones that call the read and write functions. The facade knows about the objects because it must use their accessors and mutators to implement the read and write functions.

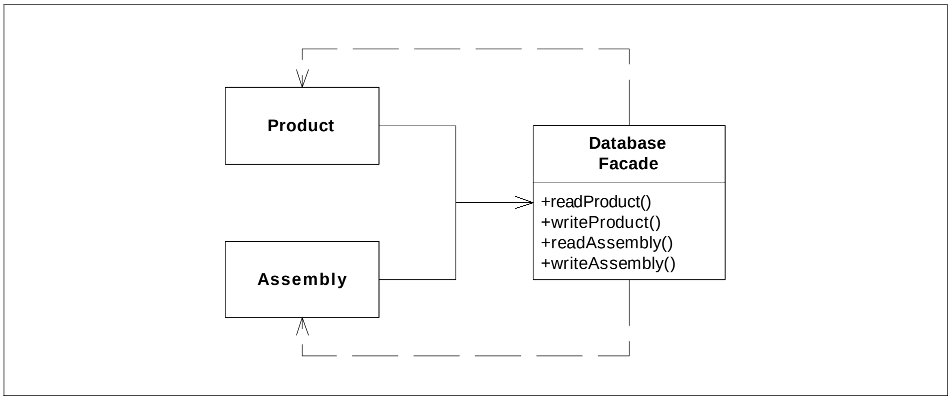

Figure 26-10
Database Facade

This coupling can cause a lot of problems in larger applications; but in smaller apps, or in apps that are just starting to grow, it's a pretty effective technique. If you start using a facade and then later decide to change to one of the other patterns to reduce coupling, the facade is pretty easy to refactor.

CONCLUSION.

It is very tempting to anticipate the need for PROXY or STAIRWAY TO HEAVEN, long before the need really exists. This is almost never a good idea; especially with PROXY. I recommend starting with FACADE and then refactoring as necessary. You'll save yourself time and trouble if you do.

BIBLIOGRAPHY

[GOF95]: Design Patterns, Gamma, et. al., Addison Wesley, 1995

[Martin97]: *Design Patterns for Dealing with Dual Inheritance Hierarchies*, Robert C. Martin, C++ Report, April, 1997.