Evolutionary Design with Databases

Gareth M Reeves
Lante Corporation
600 W Fulton
Chicago, IL 60661 USA
gxr@lante.com

XP has an evolutionary approach to building systems. The evolutionary design model grows a system over time as more functionality is added. This is more akin to growing a Redwood tree than building a bridge. To keep the design clean and simple we must continuously review the structure of the code and change it when it starts to smell[1]. We call this Refactoring, and it is as essential to evolutionary software design as water and light are to the Redwood tree.

This paper is about my experiences with applying evolutionary design to databases. In his book Refactoring [1] Martin Fowler talks about databases as being a troublesome area for refactoring. On p63 Martin says '*Most business applications are tightly coupled to the database schema that supports them. That's one reason that the database is difficult to change. Another is data migration*'.

In addition, there is some evidence that our findings are consistent with those of others. On the Object Technology User Group (OTUG) mailing list, Ian Chamberlian said "We use SQL Server but have a independent persistence layer. At the moment the structures are fluid and subject to change, so we don't optimize. In fact I constantly recommend ['Doing The Simplest Thing That Could Possibly Work' (DTSTTCPW)[2]] and ['You Aren't Gonna Need It' (YAGNI)[3]]. When the structure settles down and we can profile workload, then we can optimize. This works great for an independent development team." He goes on to say, "I see no reason why an XP team cannot develop their data requirements independently of a corporate DB. A good [Object Oriented (OO)] design will abstract the actual object to [Relational Database Management System (RDBMS)] mapping into a separate tier anyway."

Databases are commonly thought of as an area of the system that needs to be designed and locked down in a schema before coding can begin. We have found that this is not always the case and that a small set of techniques can enable you to build a database schema one table at a time and take advantage of the benefits of evolutionary design.

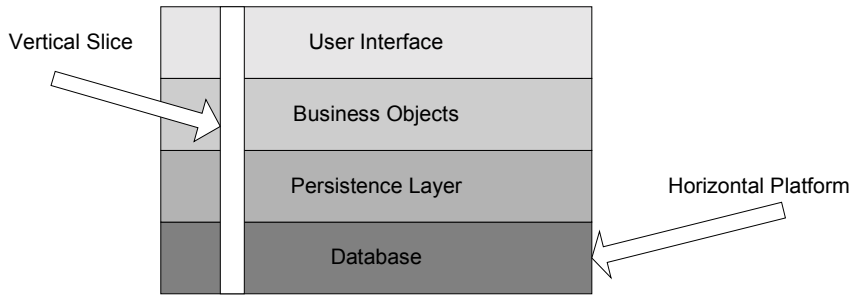
More on evolutionary design

Evolutionary Design takes a different approach to the way activities involved in building a system are scheduled. Evolutionary design encourages chunking the work into thin vertical slices of business functionality that spike all the way through the infrastructure of the system. The alternative and more common 'phased'[4] approach encourages building horizontal platforms to eventually layer business functionality and a user interface on top of at the end. Evolutionary Design is more like the way that a

---

[1] Code starts to smell when it becomes un-maintainable and hard to understand.
[2] An Evolutionary Design Practice - http://www.c2.com/cgi/wiki?DoTheSimplestThingThatCouldPossiblyWork
[3] An Evolutionary Design Practice - http://www.c2.com/cgi/wiki?YouArentGonnaNeedIt
[4] A phased approach to building systems is the opposite of the evolutionary approach. The word phased is used because the approach separates the work activities of Analysis, Design, Implementation and Testing. All of the Analysis is done at the start of the project and then when it is completed the Design phase starts, when the system is designed it can then be built and finally tested.

tree grows layers of rings on top of other rings than the way that engineers build the infrastructure of a bridge before layering the road on top.



There are two major advantages of the evolutionary approach to building a system with vertical slices over the horizontal platforms. The first is one of risk mitigation. Vertical slices spike through all of the layers of the infrastructure as well as the business functionality and force developers to address problems earlier. Vertical slices also mitigate the risk of misunderstood or poorly communicated requirements by getting the functionality in front of the user sooner.

The second advantage is that it allows stakeholders[5] to 'steer' the direction of the system based on real world usage and feedback. When you build one vertical slice at a time it is easy to change the direction of the system and incorporate feedback from the people who actually use the system. The net result is that a customer gets more of what they want today, with a clearer idea of the end product, as opposed to getting what they wanted back when the requirements specifications were written and the end product was merely conceptual.

"This all sounds good, but what is the catch?" I hear you say. Well, the catch is that you have get **really** good at evolving a system and give up the erroneous yet comforting idea that the majority of a system has to be designed before it can be coded.

A Story about a big Refactoring

I am a big advocate of evolutionary design and so when tasked with a new project to builds a system we started with something very simple. The system started with a single domain object. Amongst other things this domain object was responsible for persisting itself to the database. We implemented the logic for doing this within the domain object in a method called 'storeInDB()'. There was a similar method for retrieving the object from the database both methods contained the required SQL code to interact with the database. This worked well for a single domain object and just as well when we had two domain objects but it started to smell after we introduced some simple relationships between the objects. Sometimes we would want to retrieve objects based on 'where' clauses on other objects. Joins were required and the SQL code had to be written twice. This clearly violates the 'once and only once' rule and cost us dearly when we would add, remove or change an attribute on one of the objects. We also needed to write a test for each and every query and that meant that the information was duplicated again. It was clear that this was not a scalable way of persisting our objects.

To improve the design we introduced an object that was responsible for building SQL statements given a mapping object. The mapping object would explain how to map attributes in a domain object to tables and columns in the database. Now adding, removing or changing an attribute was simply

[5] The ultimate stakeholder would be the person paying for the system that is being built but other stakeholders could be CIO, product manager, project manager, end user, visual designer, interaction designer, tester etc.

done in one place, in the mapping object. Relationships between objects and where clauses were now represented separately from the attributes on the objects, this allowed us to conform to 'once and only once[6]. The really cool part about this was reducing the complexity of the tests. We removed the majority of the SQL tests and replaced them with a single test for each of the mapping objects.

Database Isolation Layer

The story above introduces the concept of a database layer that maps objects in the system to tables in the relational database. With object mappings we are able to decouple our business objects from the database. Changes in database schema are isolated to two places, the schema itself and the mapping object.

Using these mappings we can dynamically generate the required SQL to load from and store objects into a database. All this can be done at run time meaning that the SQL for a given query is never hard coded into the system. In addition to automatically generating and running the queries we can use Java Reflection to set the data from the result set onto the object. We call this process of instantiating objects from the database, Object Hydration.

A basic query works like this:

1. When a request is made of the layer to hydrate an object the mapping is supplied along with additional information about the request. For example, we may want to get an instance with a specific primary key or we may want to get all instance when a field equals a certain value.
2. The SQL generator is used to build a query based on the information supplied.
3. A connection is requested from the connection factory and the statement is prepared based on the SQL output of the SQL generator.
4. The statement is executed and a result set is returned. A new object is created for each row in the result set in preparation for the new data. The mapping object is used to find the associated 'setter' method for each of the data elements in the result set and Reflection is used to set the data onto the object.
5. The hydrated object(s) are returned to the application.

Example of an Object Mapping:

```
mapping = new Mapping();
mapping.setObjectClass( TestObject.class );
mapping.setTableName( "Test" );
mapping.addAttribute( new Attribute( "attr", "Attribute", String.class ) );
mapping.addAttribute( new Attribute( "tableId", "TableId", Long.TYPE ) );
```

The mapping contains three things.

1. The type of object that it is representing. We use the class object for this.
2. The name of the table that the object is persisted in.
3. A collection of the attributes to be persisted.

The attributes also contain three things.

1. The name of the instance variable to be persisted.
2. The column name to persist it in.

[6] A good Object Oriented design practice - http://www.c2.com/cgi/wiki?OnceAndOnlyOnce

3. The type of the instance variable. Again we use the class objects to represent objects and Primitive.TYPE to represent primitive values (where Primitive is the java.lang object representation of the primitive value).

Object Mappings are one of the most important classes in the system. Given an Object Mapping we can build other objects to generate the required SQL to persist of load that object. We can also build something to execute that SQL and transfer the data from the ResultSet back into Object form.

Here is an extract of some code that builds SQL statements given a object mapping.

```
public String getSelectSql(Mapping mapping)
{
	StringBuffer sql = new StringBuffer();
	sql.append( "SELECT " );

	Iterator attributes = mapping.getAttributes();
	appendFirstAttribute( attributes, mapping, sql );
	appendAttributes( attributes, mapping, sql );

	sql.append( " FROM " + mapping.getTableName() );
	return sql.toString();
}
```

The method appendAttributes adds each of the attributes to the string buffer and separates them with a comma.

```
public void appendAttributes(Iterator attributes, Mapping mapping,
	StringBuffer stringBuffer)
{
	while( attributes.hasNext() )
	{
	Attribute attribute = (Attribute) attributes.next();
	stringBuffer.append( ", " + mapping.getTableName() + "." +
	attribute.getColumn() + " AS " + mapping.getTableName() + "_" +
	attribute.getColumn());
	}
}
```

When building the SQL statements we give each value a specific name that is comprised of the table name and the column name. You will see why we do this when we get into Aggregation.

The appendFirstAttributes method does the same thing but only takes the first attribute off of the iterator and does not append the comma to the beginning of the string.

The next code extract comes from the object that executes the SQL.

```
public Iterator doSelect(Mapping mapping) throws Exception
{
	String sql = selectSql.getSelectSql( mapping );

	Connection connection = getConnection();
	Vector returnObjects = new Vector();

	Statement statement = connection.createStatement();
	ResultSet resultSet = statement.executeQuery( sql );

	while( resultSet.next() )
```

```
	{
		Object object = mapping.getObjectClass().newInstance();
		returnObjects.add( object );
		setValuesOnObject(mapping.getAttributes(), mapping, resultSet, object );
	}

	return returnObjects.iterator();
}
```

This is probably the simplest way to interact with the database isolation layer. The above method will return all of the objects in the database as specified in the mapping. The setValuesOnObject method transfers the data from the ResultSet and sets them onto the object using reflection.

```
public void setValuesOnObject(Iterator attributes, Mapping mapping, ResultSet resultSet,
Object object) throws Exception
{
	while( attributes.hasNext() )
	{
		Attribute attribute = (Attribute) attributes.next();
		if( attribute.getType() == String.class )
		{
			String value = resultSet.getString( getAsName( mapping, attribute ) );
			attribute.getSetter().invoke( object, new Object[] { value } );
		}
		if( attribute.getType() == Long.TYPE )
		{
			long value = resultSet.getLong( getAsName( mapping, attribute ) );
			attribute.getSetter().invoke( object, new Object[] {new Long(value)} );
		}
		// Other types omitted to reduce size
	}
}
```

Although retrieving a list of a single object type from the database is sometimes all that is needed, it is often more interesting to retrieve objects based on the relationship that they have with other objects. This requires us to do join's in the SQL.

```
public Iterator doAggregateSelect(Mapping mapping, Mapping secondMapping, String field1,
String field2 ) throws Exception
{
	String sql = selectSql.getSelectSql( mapping, secondMapping, field1, field2 );

	Connection connection = getConnection();
	Vector returnObjects = new Vector();

	Statement statement = connection.createStatement();
	ResultSet resultSet = statement.executeQuery( sql );

	while( resultSet.next() )
	{
		Object firstObject = mapping.getObjectClass().newInstance();
		Object secondObject = secondMapping.getObjectClass().newInstance();
		returnObjects.add( secondObject );
		Iterator firstAttributes = mapping.getAttributes();
		setValuesOnObject( firstAttributes, mapping, resultSet, firstObject );
		Iterator secondAttributes = secondMapping.getAttributes();
		setValuesOnObject( secondAttributes, secondMapping, resultSet, secondObject );
		aggregate( secondObject, firstObject );
	}
```

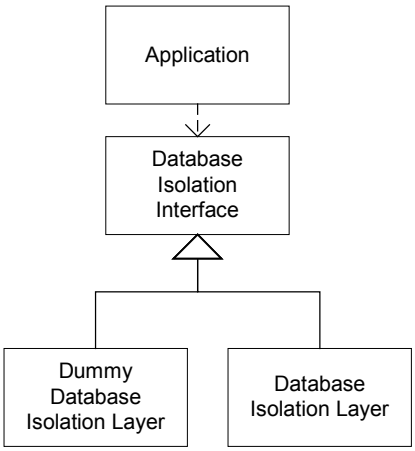
```
        return returnObjects.iterator();
}
```

We have a second method for building SQL statements that now takes two mappings and two fields to join on. Now the use of identifiers that are combination of both the table name and column name is more apparent. We do this to protect against column names that are the same in each of the tables being joined. Finally, the aggregate method uses reflection on the second object to find a setter method that takes the first object as a parameter. If a setter is found it will invoke and join the two objects together.

These same techniques can be used to update and insert objects into the database by adding some information about primary and composite keys to mapping (not shown in the example). We have also been successful in implementing techniques like concurrency timestamps[7] without having to modify our application code at all.

Unit Testing

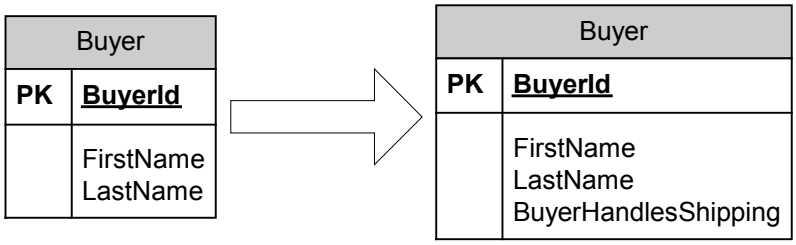
The database isolation layer also simplifies unit testing. The layer offers another place to use the 'Mock Object' [2] concept, which enables us to isolate parts of the system for testing. In this case we want to isolate our application code from the database.



One way to do this is to implement Mock or Dummy implementations of the database API (JDBC). The dummy implementations fool the application into thinking that it is working with a real database and capture the information that passes back and forth for testing purposes. Implementing the dummy classes for JDBC is quite simple but implementing it for the isolation layer is even easier. This is because isolation layer returns fully hydrated objects instead of lower level database types.

Database refactoring example 1

Lets start with a simple one. We are building an RFQ exchange system and as part of the continuous requirements process it has been decided that Buyers can specify that they want to handle shipping. This is a new attribute of the RFQ class and should be persisted (at least for now) in the RFQ table under a new column BuyerHandlesShipping.



In order for this change to take affect we need to add the new field to the RFQ class and a getter and setter for the field. Then we need to add the attribute to the RFQ mapping object and finally add the

[7] Concurrency stamps are a simple concurrency management technique.

column to the database. When we add the column we should probably give it a default value of false. That's it. The isolation layer takes care of loading, creating and updating the RFQ objects.

Adding a new attribute to our mapping object looks something like this:

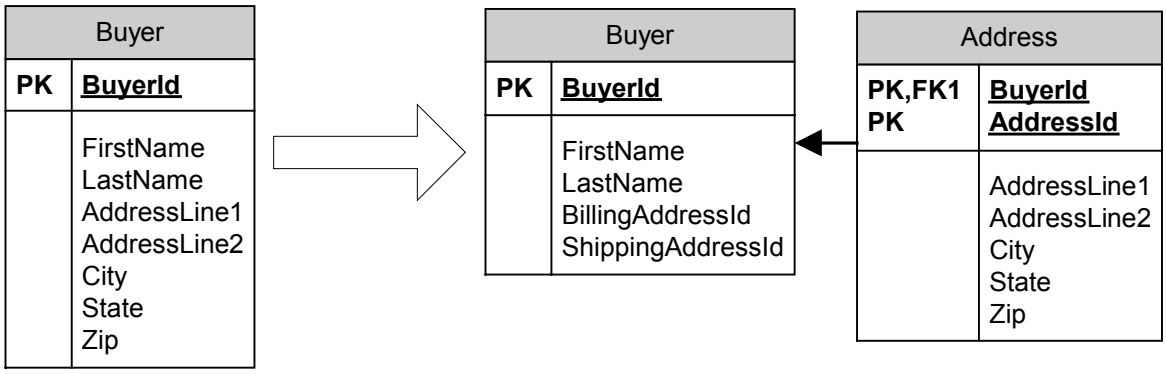
```
mapping.addAttribute(
new Attribute( "buyerHandlesShipping", Boolean.TYPE, "BuyerHandlesShipping ) );
```

The first parameter is the name of the field on the object, the second is the type of the field that is used for JDBC and the last parameter is the column name in the table. In addition to the attributes the mapping will need to contain a table name and object type.

Database refactoring example 2

Now lets take a slightly more complicated example. Until now the Buyer has only required a single address in the system. This address has served the purpose of both Shipping Address and Billing Address but now there is a new requirement that will allow the buyer to supply a different Shipping Address from Billing Address.

Side Note: The Extreme in Extreme programming is about taking the best practices to the extreme. In this case I am assuming that when the original requirement only specified a one to one mapping of Buyer to Address the developers included the Address attributes as part of the Buyer table because that was the simplest thing to do. Working this way can be cost effective if cost of changing it later can be reduced so that it is equal to adding it in the first place. Now the business team has come up with a new requirement that means a one to many relationship.



Here are the steps we take for the refactoring:

1. Create new class Address and add the required fields to the class along with their corresponding getters and setters.
2. Create the new Address table in the database
3. Replace the address fields in the Buyer table with foreign keys to the billing address and shipping address.
4. Replace the address attributes in the Buyer object with getters and setters for billing and shipping address.
5. Create the mapping object that maps the Address fields to the address table.

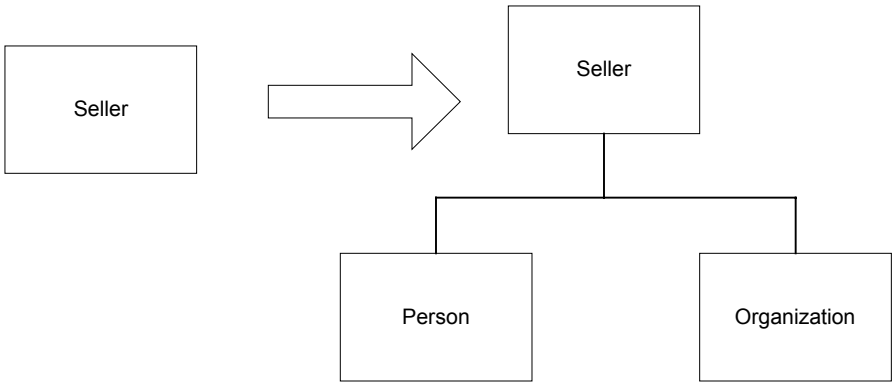
Here are some additional things that you could do:

- Instead of replacing the getters and setters in Buyer for Address getters and setters you could just use simple delegation.

- If you wanted to be able to do a reverse lookup on Address to find out which Buyer it is associated with you could add a Buyer foreign key and set it when the buyer's setter saves the address.
- If you wanted to be able to determine whether an Address was a billing or shipping address you could Address type field. The type value could either come from a constant in the code or be a foreign key to an AddressType table.

Database refactoring example 3

Until now, sellers could only be recognized as individual people. A new requirement is to allow organizations to also sell on the exchange. We now have a situation where a buyer could be buying from an individual or a organization. Lets assume that the data model contains two entities, Buyer and Seller. We can use the Party pattern to associate the buyer with either an individual or an organization. The Party pattern requires the addition of three new tables, a Seller table that will act as the Party, a Person table and an Organization table. The Seller table will contain a primary key, a foreign key to either the Person or the Organization and a descriptor that will describe whether the seller is a person or an organization. Person and Organization will require primary keys and the relevant data attributes.



In application we are looking for some polymorphic behavior. There are some scenarios where we would just like to print out some information about the seller. When displaying a person we want to print out the name and contact information and when we display a organization we want to include company name and primary contact information. Ideally we would have a Seller super class and Person and Organization subclasses that the application need not be concerned with. The display code will simply ask the Seller to print itself and the correct behavior will be executed.

Here are the steps to do the refactoring:

1. Create the two new classes Person and Organization. Modify the Seller class to only contain the attributes that are going to be in the Seller table.
2. Create the two new tables. Modify the Seller table.
3. Create the mapping objects for the two new classes. Modify the Seller mapping object.
4. Implement print on Person and Organization classes.
5. Change print method on seller to check for the type of Seller, load the specific seller type and delegate print to that object.

Some things to note:

- You could lazily instantiate the internal Person or Organization to avoid reading from the database each time it is printed or other operations are performed.
- The database isolation layer that we have been using does not yet hydrate objects across inheritance hierarchies. True polymorphic behavior would be achieved by having the Person and Organization classes inherit from Seller but as previously stated, we cannot load objects from the database this way. The composition approach that I describe achieves the same thing but is less flexible.

Example Schema Evolution

The table below shows how our database grew over the course of a 5 iteration project. You can see how the project started with a single table and grew to 4 tables by the second iteration. The second iteration introduced some new domain objects into the system that required persistence. Not all iterations required new tables but the number of rows in the database continued to increase. This was because we were adding new test cases that required data in the system.

Iteration Number	Number of Tables	Number of Columns	Number of Rows
1	1	7	3
2	4	23	10
3	5	26	12
4	6	29	20
5	6	29	25

Data

The above examples assume that there is no data in the database when you do the refactoring. This is rarely the case.

When working in development, our approach to data migration is to do it as little as possible. Instead of migrating one database schema to the next we rebuild our database with new test data. Scripts are used to build the database and insert test data so that making a change is very easy. When we make a change like the ones described above we change the database build script to reflect the changes in the schema. We then change the data insert scripts to populate the new schema with our test data.

We configure and manage our own databases for development and that frees us up from having to answer to organizational data standards and DBA controls every time we want to make a change. We evolve our schema as needed over our development cycle and then address DBA acceptance when it is time to move the system into production. At this time we can use standard data migration techniques to move the data over.

Conclusion

Traditional data modeling has had to be very forward thinking. It has been essential to make the schema as generic, flexible and robust as possible to be able to cope with a potential future

requirement. The high cost of changing the database schema has dictated this up front design mentality, and rightly so. What results is a long initial phase to design the ultimate database schema and then a resistance to changing it.

Ron Jeffries has commented on the need for XP practitioners to overcome this mentality on the Object Technology User Group. His comments were "Defense of the existing structure is a VERY bad way to develop new structure. Like it or not, XP is a process for building a new software thing, and does it by evolving from a simple structure to a well-formed final one. A team doing XP needs the rapid ability to evolve its solution."

Our findings have shown that the cost of changing the database does not have to be so high. The following techniques have helped us in reducing that cost:

- Decoupling the application from the database with a database isolation layer or an Object/Relational mapping tool.
- Working with an instance of a database that is entirely under our control to modify and rebuild as needed.
- Using scripts to automate the creation of the database and to insert the test data.
- Using common OO techniques.

These findings are also consistent with Robert Martin's advice on a recent IT Forum. "Decouple the application from the database. Create polymorphic interfaces that contain methods that give you the queries you need. Allow no SQL in the application code itself. Then you can write tests by creating mock implementations of these interfaces. You can run your unit tests without having to ask the DBA to make any changes at all.

He continues to talk about the advantages of local database. "Create a local database for pre-release testing. Make sure you have rights to change the schema whenever you like. Update it whenever you have to, while evolving your software. At some point during the iteration, the schema will settle down. Then you can present the schema changes to the DBAs."[8]

The advantages of the evolutionary design approach are clear. Spreading the design work equally over the project allows you to mitigate risk areas towards the start of a project. In addition, adding functionality incrementally to a system allows for changes in scope, meaning that a customer can get more of what they want without paying an exorbitant cost for changing their mind.

References

1. Fowler, M. Refactoring
2. Freeman, Mackinnon, Craig. Endo-Testing: Unit Testing with Mock Objects http://www.sidewize.com/company/mockobjects.pdf

[8] http://forums.itworld.com/webx?14@35.PmmvaWpiiyV^0@.ee6eed4/56!skip=