# Section 4 – Data Access Layer:

The storage and retrieval of data from a datastore is arguably the most critical part of an Enterprise Application. This section provides guidelines for implementing data storage and retrieval with minimum effort and the total elimination of repetitive SQL or mapping code. In addition to showing how to do this we will also discuss how this is done so as to not only achieve portability between vendors of various relational databases who implement slightly different dialects of SQL but also portability between different datastored e.g. XML and Memory dumps. Habanero tackles this difficult enterprise application problem by implementing an Object Relational Mapping strategy (ORM). This strategy is steadily gaining acceptance as the preferred data access strategy for developing enterprise applications.

Although we recommend that you at least skim through this chapter. If you just want to get started using Habanero then you can skip this entire chapter since firestarter automates the generation of all database mappings. You can then use this chapter as a reference when you come across one of the more complex mappings namely Many to Many and Inheritance hierachies.Although this book is about developing Enterprise applications using Habanero to fully utilise the power of Object Oriented Design (Domain drive design) while storing your data in a relational database requires at least a basic understanding of the various options and design patterns that are used in designing the various solutions. In this section we are therefore going to show these problems and how they can be solved in Habanero.

Chapter ix: Object relational mapping

All the examples developed so far in Section 2 where developed using an In Memory Datastore. This shows the incredible power and flexibility of Habanero in being able to plug in any datastore e.g. an XML datastore.

The most commonenterprise data store is still however the relational database. As mentioned previously there is inherently a significant impedance mismatch due to the different objectives of objects and tables. Objects represent and model behavior whereas tables model data. At the same time objects also have state the state of the object may last over the lifetime of a single object. In these cases this state needs to be persisted. In the case of a relational database normalized to the third normal (or Boyc Codd Normal) form the objects state will be stored in one or more tables and each of the objects persistant properties will be stored as a field on one or more of these tables. For the purpose of retrieval each object must be uniquely identifiable ideally globally but at least within its class.

What is a relational database.

A relational database …Normalisation of data…The most common enterprise datastore and is thus almost always used in an enterprise application. Tabular representation of data in tables with rows and columns

The impedence mismatch

modern object oriented software uses sophisticated domain models that are not represented by rows and columns in a table. The rich domain model allows the application developer to develop an application using sophisticated object oriented concepts and patterns in the language of the domain. The habanero ORM ensures that the developers interaction with fjdla;fjdsa.

SQL

Habanero totally isolates the developer from having to write SQL for data access. We will therefore not go into any detail on SQL even though you may use it to create custom views or stored procedures.

The domain model is inheritantly richer than the Relational model.

Non entity types e.g. Address that is stored in same table as user but is a specific type (money is another example) See evans for more discussion.

Unique identity of an object map to the Primary key of a table.

The fit between object technology and RDB technology is not perfect. In the early 1990s, the difference between the two approaches was labeled the *object/relational impedance mismatch*, also referred to as the *O/R impedance mismatch* or simply the *impedance mismatch*, terms still in common use today. Why does a technological impedance mismatch exist? The object-oriented paradigm is based on proven software engineering principles. The relational paradigm, however, is based on proven mathematical principles. Because the underlying paradigms are different, the two technologies do not work together seamlessly. The impedance mismatch becomes apparent when you look at the preferred approach to access: With the object paradigm you traverse objects via their relationships, whereas with the relational paradigm you join rows of tables. This fundamental difference results in a nonideal combination of the two technologies, although when have you ever used two different things together without a few hitches? Scott Ambler

There is also a cultural impedance mismatch between developers in the object community and data professionals in the data community. Object developers have been taking an evolutionary (iterative and incremental) approach to development for years and are now quickly moving towards agile development methods such as extreme programming (Beck 2000), Scrum (Beedle and Schwaber 2001), feature-driven development (FDD) (Palmer and Felsing 2002), and agile modeling (Ambler 2002). Unfortunately many within the data community look upon evolutionary development as a questionable approach, and agile approaches (Ambler 2003b) are just now being considered. Object developers typically take an object-oriented approach to modeling using unified modeling language (UML) diagrams, whereas data professionals focus on data-oriented models such as logical data models (LDMs) and physical data models (PDMs). Furthermore, many object developers are not skilled at data modeling and data professionals are not skilled at object modeling. When it gets right down to it the real issue is that data professionals view the world as data to be manipulated, whereas object developers view it as objects to be combined to perform behavior.

Due to these two impedance mismatches, one technological and one cultural, it can be very difficult to succeed at object persistence. Unfortunately we have little choice in the matter: these are the two technologies that we typically have to work with, and so we need techniques to overcome the inherent challenges. The goal of this chapter is to describe how you can effectively develop the data aspects of your business software.

#### Object persistence: Object Relational mapping, Concurrency control, Transacitonal support.

The term "mapping" refers to how objects and their relationships are stored in a relational database

Some attributes of an object are objects in their own right; in the class model *Address.state* is actually a *State* object. As you saw in [Chapter 13](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0131.html#850) this really reflects an association between the two classes; association mapping is described in [Section 14.2.3](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P1517D074CFD-5182-4BF8-AB91-930CAE575BC9). For now trust me that the two models in [Fig. 14.1](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P297D074CFD-5182-4BF8-AB91-930CAE575BC9) map to one another.

Remember that not all attributes are persistent; some are used for temporary calculations. For example, a *Student* object may have an *averageMark* attribute that is calculated by the object but is not saved into the database. Similarly, as you see in [Fig. 14.1](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P297D074CFD-5182-4BF8-AB91-930CAE575BC9) not all table columns map back to your object schema—there is no corresponding attribute for the *State.Capital* column. The important thing to understand is that at some point each attribute of a class will be mapped to zero or more columns.

Mapping is only this clean when you have control over both your object and data schemas. In situations where you cannot easily change the database schema, which is typical when you are working with one or more legacy databases, you will find that the mappings are much more complex. In these situations mappings between classes and tables become many-to-many—the attributes of a class will map to the columns of several tables, and any given table will have several classes mapping to it. You may even have several attributes mapping into the same column (and worse yet vice versa). When you are working with a legacy database you may need to do significant legacy analysis ([Section 14.7](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0145.html#1056)) if adequate documentation is not in place. You will also find that you need to write code to encapsulate these complex mappings, perhaps in the form of data access objects (DAOs) discussed in [Section 14.3](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0141.html#1000).

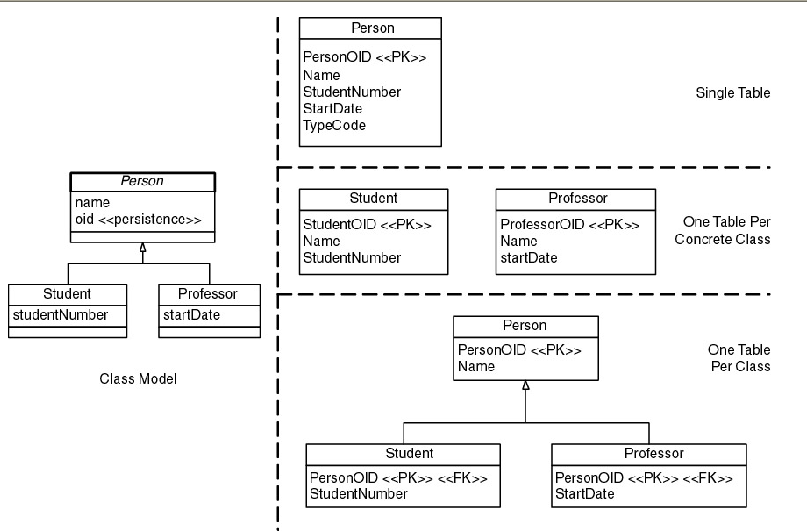
### Mapping Inheritance Structures

Relational databases do not natively support inheritance, forcing you to map the inheritance structures within your object schema to your data schema.

There are three primary solutions for mapping inheritance into a relational database:

1. Map the entire class hierarchy to a single table.
2. Map each concrete class to its own table.
3. Map each class to its own table.

Having said that, I have found that the single table per hierarchy approach is often your best option.



| Table 14.1: Comparing the Three Techniques | | |
| --- | --- | --- |
| **Technique** | **Advantages** | **Disadvantages** |
| Single table | * Simple approach. | * Coupling within the class hierarchy is increased because all classes are directly coupled to the same table. A change in one class can affect the table, which can then affect the other classes in the hierarchy and their related mappings. |
|  | * Easy to add new classes: you just need to add new columns for the additional data. | * Potential space wasted in the database because many rows will have empty columns. |
|  | * Supports object polymorphism by simply changing the type of the row. | * Indicating the type becomes complex when significant overlap between types exists (e.g., someone is both a professor and a student). |
|  | * Data access is fast because the data are in one table. | * Table can grow quickly for large hierarchies. |
|  | * Ad hoc reporting ([Section 14.5.6](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0143.html#1048)) is very easy because all of the data are found in one table. |  |
| One table per concrete class | * Good performance to access a single object's data. | * Modifications to classes require you to modify its table and the table of any of its subclasses. For example if you were to add *birthDate* to the *Person* class you would need to add columns to the *Person, Student*, and *Professor* tables. |
|  | * Ad hoc reporting simple because all of the data you need about a single class are stored in only one table. | * Whenever an object changes its role, e.g., perhaps the university hires one of its students, you need to copy the data into the appropriate table. |
|  |  | * It is difficult to support multiple roles and still maintain data integrity. For example, where would you store the name of someone who is both a professor and a student? |
| One table per class | * Simple mapping because it is one-to-one. | * There are many tables in the database, one for every class (plus tables to maintain relationships). |
|  | * Easily supports object polymorphism because you merely have records in the appropriate tables for each type. | * Potentially takes longer to read and write data because you need to access multiple tables. |
|  | * Very easy to modify superclasses and add new subclasses as you merely need to modify/add one table. | * Ad hoc reporting is difficult, unless you add views to simulate the desired tables. |
|  | * Data size grows in direct proportion to growth in the number of objects. |  |

Doing a good job of object relational mapping requires a good understanding of object modelling and relational database modelling. Object relational mapping is thus difficult it is further complicated by the fact that not all object oriented programming environments support the same features. Programming environments implement different data access and data management capabilities. Relational databases are deficient in implementing core relational theory. In addition to this all relational database vendors support different features and capabilities.

For the purposes of this book we will thus deal with OR mapping between the .Net environment and the major database vendors Oracle, SQL server, MSSQL and MSAccess Jet database engine.

Relational theory is primarily with the access and management of data (knowledge). Object Modelling is primarily concerned with behaviour. It is here that we find what is commonly referred to as the Object relational dissonance or mismatch.

##### Object Modelling

Basic principles

Identity (OID):- Objects have a unique and immutable identity which distinguishes them from all other objects in the given system for all time. The objects identity is independent of the state of the object. OID’s are never reused.

State:- The current value of the object. The state of the object should be encapsulated and should only be visible by examining its behaviour.

Behaviour:- Objects provide an abstraction that other objects (‘Clients’) can deal with. The behaviour of an object is the collection of operations (‘methods’) that the object provides (its interface). All interactions with an object is via its interface.

Encapsulation:- abstraction of behaviour that prevents the implementation details being seen from outside of the object.

Persistence:- An object can persist beyond the lifetime of the application session. During this time an object must be stored in a file or Database.

It is way beyond the scope of this book to discuss methodologies and processes for capturing requirements, Object Design techniques etc. There are many excellent books on these subjects including (The Object Primer – Third Edition by Scott Ambler. Enterprise applications). The philosophy of Design has also changed in the last few years with Agile methodologies being applied more and more (An Excellent book that summarizes the various methodologies and how to select one is ‘Rapid Application development’ Steve m) For the purposes of this book We assume that you have a number of classes that you wish to model in application and which you wish to persist in the database. See chapter xxx which will go through a brief walkthrough of how firestarter an application modeler and generator can be used to support both agile and traditional software development methodologies.

Relational Database modelling:- normalisation.

##### ORM

### Mapping Relationships

For the purposes of our discussion the term relationship shall apply to either associations or composition associations as there is no significant difference between the two from a mapping point of view. Relationships in object schemas are implemented by a combination of references to objects and operations, whereas in RDBs they are maintained through the use of foreign keys. A foreign key is a data attribute(s) that appears in one table that may be part of or is coincidental with the key of another table.

The way that you map a relationship depends on its multiplicity:

1. **One-to-one relationships.** To map a one-to-one relationship you have two strategies. If each class corresponds to a table, which we see in [Fig.14.4](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P1507D074CFD-5182-4BF8-AB91-930CAE575BC9) with the *Course* and *CourseVideo* classes, then a foreign key needs to be implemented in one of the two tables. In this case the *CourseV- ideo.CourseNumber* column acts as the foreign key to the *Course* table. From a data point of view it does not really matter which table the foreign key goes in, although if the relationship is unidirectional association then it makes sense to have the foreign key in the table corresponding to the class that maintains the relationship. For example, if there is a unidirectional association from *Person* to *Address* then add the foreign key into the *Person* table. The second mapping strategy is to simply store the data for both classes in a single table, the approach we took to map the relationship between the *Course* and *CourseOverview* classes.

Figure 14.4: Mapping associations.

1. **One-to-many relationships.** One-to-many relationships, such as *offering of* in [Fig. 14.4](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P1507D074CFD-5182-4BF8-AB91-930CAE575BC9), are implemented in classes via a collection such as a *Hashset* on the many side of the relationship and by an object reference on the one side of the relationship, in this case *Course.seminars* and *Seminar.course*, respectively. Within the database the relationship is implemented via a foreign key on the many side of the relationship, in this case *Seminar.CourseNumber.*
2. **Many-to-many relationships.** A many-to-many relationship between two classes is mapped to an associative table within a database. This table comprises the primary keys of the two business tables. In [Fig. 14.4](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P1507D074CFD-5182-4BF8-AB91-930CAE575BC9) the *instructs* relationship between the *Professor* and *Seminar* classes is mapped to the *Instructs* table within the database. By adding an associative table like this we say that we have resolved the many-to-many relationship by converting it into two one-to-many associations.

A general rule of thumb with relationship mapping is that you should keep the multiplicities the same; i.e., a one-to-one object relationship maps to a one-to-one data relationship, a one-to-many maps to a one-to-many, and a many-to-many maps to a many-to-many. The fact is that this does not have to be the case; you can implement a one-to-one object relationship with to a one-to-many or even a many-to-many data relationship. This is because a one-to-one data relationship is a subset of a one-to-many data relationship and a one-to-many relationship is a subset of a many-to-many relationship. However, implementing a one-to-one or a one-to-many as a many-to-many is an example of overbuilding your software so I discourage it.

A recursive relationship, also called self or reflexive relationships, is one where the same entity is involved with both ends of the relationship. For example, the *prerequisites* association in [Fig. 14.4](mk:@MSITStore:C:\Brett\_All%20Info\_Programming\_Books\agile\The%20Object%20Primer%20Agile%20Model-Driven%20Development%20with%20UML%202.0%203rd%20Edition.chm::/0140.html#wbp17Chapter14P1507D074CFD-5182-4BF8-AB91-930CAE575BC9) is recursive, representing the concept that a course may be a prerequisite to other courses. There is nothing special about recursive relationships; you map them the exact same way that you would a nonrecursive relationship. For example, because the *prerequisites* association is many-to-many I simply map it to the *Prerequisites* associative table.

From the above discussion

Overcoming impedence

(Start with the simple case ignoring inheritance we will tackle inheritance just now and ignoring relationships)

E.g. To map a person class to the database



1. For each persistable class create a table.
2. For each persistable field (attribute, property) of the class create a column (attribute) for this table.
3. Add an additional attribute to the table for the object ID.
4. Set the object ID as the primary key.
5. Set any other natural keys as alternate keys.

Person Table



Mapping relationships

Mapping Class relationships to a relational database is now relatively easy.

1. Mapping aggregation and composition 1:M

Composition and aggregation is where one object is made up of or contains other objects. The common examples are a file consists of the papers in the file. A team is made up of the players in the team. A house is composed of walls, floors, windows. (See Object modelling above)

If a Team is an aggregation of the players in the team then the

* 1. The Team class is mapped to the database as for the person class.
  2. The Team ID is mapped as a foreign Key in the person table.
  3. The aggregation relationship is mapped as a 1:M Database relationship.



1. Mapping composition and aggregation 1:1

This is a somewhat artificial example but lets say we treat the person an aggregation of their parts i.e. the Person has one Heart.



* 1. This mapping is very similar to the 1:M above.



1. Map 1:M relationships



* 1. The person can own one or more cars.
  2. The car can be owned by one and only one person.
  3. This relationship is shown as navigable in both directions. Whether the relationship is navigable in one direction or both directions does not affect the mapping to a relational database.
  4. We map this the same way we did the 1:M aggregation above by creating a Person\_ID foreign Key in tbCar.

Note there is nothing in the database to show the difference between the Team – Person relationship and the Person – Car relationship. You can give the database relationship a name but this adds little value.



1. Map 1:1 relationships

We will map the relationship that a Team has a home ground.



* 1. This relationship is shown to be navigable in only one direction.

There are a few choices in doing this mapping.

* + 1. Option 1. We create the Team\_ID as a foreign Key in tbGround.
    2. Option 2. We create the Ground\_ID as a foreign Key in tbTeam.
    3. Either of these solutions will be equally acceptable from an Object relational mapping perspective.
    4. Our standard has become to place the foreign key in on the navigable relationship if there is only one.



1. Map M:M relationships.

For this example we have a relationship between the player position (Full Back) and Player. In this case we have a Position which can have zero or more players but we also have Players who can play one or more Positions. This type of relationship is very common, however it is very important to differentiate between this relationship and two 1:M relationships the difference comes is whether the relationship contains any data e.g. in the example below if the Player has a first position and alternate positions e.g. the player is a centre but can stand in as a full back then this relationship is not a many to many but is instead two 1:M relationships one between the Player and the PlayerPosition and the Other between Position and PlayerPosition. In the case of a true M:M the Position will have a collection of players and the player a collection of positions.



* 1. When mapping a many to many to an ERD or a database model the rules are fairly simple an associative entity is created between the Position and the Person. Thus creating two 1:M database relationships instead of the many to many.
  2. The associative entity will hold the primary key of both tables as a foreign key and the composite of the two primary keys will be a unique constraint on the associative entity.
  3. The only additional decision that needs to be made for the Database mapping is whether the primary key for the TeamPerson should be a composite key consisting of Team\_id and person\_ID or whether Team\_ID, Person\_ID should be set up an alternate key and a Team\_Person\_Id is created.
     1. The rules for this are fairly simple if the M:M class relationship is implemented as an associative class then you must create a Team\_Person\_Id since an object id is necessary.
     2. If the M:M class relationship is implemented via two collections then there is no need to create a new ID.



For more info see Association Table Mapping pattern – Fowler 248. The Link Table (Player Position should have no corresponding in memory object as a result it does not require its own object ID although I have seen a number of database designs which still do this. The primary Key for PlayerPosition is a compound key consisting of PersonID, PositionId.

1. Lookups

Lookups are not really relationships between two objects but will be discussed here. In reality a lookup is a rule for limiting the validation for a particular property to limit to a particular set of data e.g. The values in the lookup set of data are very seldom changed or updated e.g. a person may have a Title e.g. Mr, Mrs, Doctor, Honour. The available options are often provided to the user via a Combo box or equivalent. The business object then merely needs to validate the input value against the list of available values. Typically lookup lists are not modeled in the object model but will be modeled as a separate table with a 1:M relationship in the Relational Database Model.



1. Self referencing relationships.

This relationship is simple an Object has a relationship to itself e.g. A Person Managed Other People. The normal rules apply to this relationship i.e. it can be a 1:1, 1:M or M:M. In the database is represented by having the primary key as a foreign Key.



1. Inheritance relationships

One of the powerful features of object oriented programming is the ability to have classes inherit behaviour from other objects. The classical example often used is A Circle Inherits from Shape and a Filled circle inherits from a Circle.



This is easy enough to implement in C# the complexities however build when mapping to a database. Typically Business Objects that inherit from each other will not only inherit behaviour but will also be required to inherit properties. Each inherited class can also have additional properties. In these cases there are a number of commonly used Inheritance hierarchy - Database mappings. The Inheritance mappings supported by Habanero are ‘single table inheritance’, ‘Class table Inheritance’ and ‘Concrete table Inheritance’. A summary of the advantages and disadvantages of each mapping strategy will be given the discussion of each strategy.

1. Single Table Inheritance – Fowler (278) – ‘Represents an inheritance hierarchy of classes as a single table that has all the columns for all the fields of the various Domain classes’. This has the simplest implementation mapping of inheritance -Database mapping that can be implemented. Every field required by any class in the inheritance hierarchy is mapped to a related field in only one table. Any column that is not required by a particular class is left empty and is neither updated or loaded when that class is loaded. I.e. Firstname will not be loaded when a Contact of type ContactOrganisation is loaded. When loading or saving an object to/from the database no joins are required. Each row of the Contact table will have one additional field (The Discriminator field) that will identify the name of the class that this row represents e.g. ‘ContactPerson’ or ‘ContactOrganisation’. When loading from the database this field is used to determine which sub class to load. When inserting the record the Discriminator field is updated.



1. Class Table Inheritance – Fowler (285) – ‘Represents an inheritance hierarchy of classes with one table for each class’. This is conceptually a simple mapping since every class is mapped to one table and the database relationships between the tables map to the inheritance relationships. The fields unique to each sub class are represented in only the relevant table. There are two options option 1: Each table has as its primary key the ID from the super class. In our example below each table will have its primary key as the ContactID. This is the preferred option where the developer has control over the database design since the inheritance relationship is more obvious when viewing the database design alone. Option 2: each sub class table has its own primary key and in addition to this has a foreign key that references the Super class table. The complexity of this mapping strategy relates to the loading since loading all the contacts requires complex logic to load the correct sub class (i.e. ContactOrganisation or ContactPerson).



1. Concrete Table Inheritance – Fowler (293) – ‘Represents an inheritance hierarchy of classes with only one table for each concrete class in the heirachy’. This is conceptually a simple mapping since every concrete class is mapped to only one table. All the fields for every concrete class are represented in each table that is mapped. The complexity of this mapping strategy relates to the loading since loading all the contacts requires complex logic to load the correct sub class (i.e. ContactOrganisation or ContactPerson). 
2. When to use each inheritance mapping strategy.

Our suggestion is that you always start with single table inheritance since these results in the least complexity in the database and is easiest to adapt and modify as your system evolves. If you have any sub classes that have a lot of fields and data that are different from the other classes then use Class table inheritance for this class only. Only use Concrete table inheritance when the sub classes only share common behaviour without sharing many common fields or persistant relationships to other business objects.

For a detailed discussion of when to use each inheritance mapping strategy see Fowler – Patterns of Enterprise application architecture (pg 278 – 301).

##### Concurrency control

When several users/applications try to read and write to a single object simultaneously then you create a contention for an object. This can result in several disastrous consequences such as losing changes made by one user (commonly known as lost updates).

The types of concurrency control issues that can occur are.

Note: In all these cases we refer to user but this could just as well be a program etc.

1. Lost Updates
   1. User 1 and User 2 read an object from the database.
   2. User 1 makes edits to the data and persists these edits to the database.
   3. User 2 makes edits to the data and persists these edits to the database thus overriding User 1’s edits.
2. Stale Data for an object.
   1. User 1 reads an object’s from the database e.g. The details of an individual soccer player.
   2. User 2 edits the object that User 1 is viewing. E.g. the player is updated as injured.
   3. User 1 now has stale data any decisions made reports printed etc are made with ‘out of date’ information.
3. Stale set of objects
   1. User 1 reads one or more objects from the database e.g. A list of all players in a soccer team.
   2. User 2 edits one or more of the objects that User 1 is using or modifies the set of objects that user 1 is using. E.g. A new player is added to the soccer team or an existing player is updated as injured.
   3. User 1 now has stale data any decisions made reports printed etc are made with ‘out of date’ information.
4. Lost delete (this is in fact a specialised version of lost updates)
   1. User 1 and User 2 read an object from the database.
   2. User 1 deletes the object and persists the delete to the database.
   3. User 2 edits the object and persists the object to the database. One of two things happen depending on the way that the programmer has written the code.
      1. User 2’s updates are lost since the object has been deleted.
      2. User 2’s updates are persisted to the database and the User 1’s delete is lost.
5. Duplicate data
   1. User 1 captures data for a new object. E.g. For a new soccer player.
   2. User 2 captures data for the same new object. E.g. for the same new soccer player.
   3. A duplicate soccer player is created in the database with two different object ids.

Strategies to Concurrency control issues.

1. Pessimistic Check out – Check in (Single write access, multiple read access)
   1. Many programs can read the data for an object.
   2. When a client object wants to edit the data it must first obtain a checkout.
   3. While this client object has the data checked out no other client object can edit the data, many other objects can however still read the data.

This strategy for concurrency control eliminates lost updates and lost deletes but does not prevent or help duplicate data or stale data.

1. Pessimistic Check out – Check in (Single read and single write access).
   1. Only one client object can read data at a time.
   2. The client object will check out the object when it begins reading and will check in the object when it is completed.

This strategy for concurrency control eliminates lost updates and lost deletes and stale data for an individual object but does not assist with stale data for a set of objects or duplicate data.

This strategy is seldom generally usable for normal business objects since it would result in an excessively restrictive environment but is useful in certain instances such as generating a unique invoice number that must be guaranteed to be unique and sequential.

1. Optimistic locking
   1. All client objects have read and write access to all the data.
   2. The client can update the data to the database if the data has not been changed since it read the data.

This strategy is useful where the severity of a conflict is low and/or where the probability of a conflict is low. This is generally the easiest strategy to implement and is the least restrictive.

This strategy resolves the lost updates and may or may not solve the lost deletes depending on how it is implemented.

1. Data Polling
   1. Client objects or sets of client objects have the ability to poll the server (database) to ensure that its data is still current and to refresh it if required.
   2. This type of solution can be employed where an object or a set of objects poll the database at specific points. E.g. when an object is currently displayed in a form or a collection of objects is displayed in a grid then it will regularly poll the database for any changes or at certain specified points the object will be required to refresh the data if it has gone stale.

This strategy primarily addresses the stale data issues and can be used in conjunction with pessimistic read multiple write once locking or with optimistic locking.

This can have significant performance overheads and must be used with circumspection.

1. Last save wins

This is not really a concurrency control mechanism but is the mechanism most novice programmer’s use (often unintentionally). This mechanism basically ignores concurrency control issues and the last user to save to the database wins and overwrites all previous users’ data.

1. Messaging
   1. When a client object or a set of client objects is created from the server it can register for messages from the server.
   2. Any time that this object’s data is changed by another client object it is notified via the message.
   3. The client object can then react accordingly.

This strategy is similar to the data polling solution.

In a high transactional volume system this can add significant performance overhead.

The architectural requirements of this type of system are more limiting than the other strategies in that an application server of some type is required so that client objects can register for messages and be sent messages.

As we can see the duplicate data is the most difficult concurrency problem to solve.

The data polling and messaging do however go some way towards preventing this type of error since the data that the user is viewing is fresher. This is particularly true when messaging and polling is used for a set of object e.g. a set (collection) of all players playing for a team is loaded. The set (collection) then polls the database for any changes to its data if an object has been deleted, added or removed from the set the set (collection can be updated or the appropriate warnings can be given).

Hence the reasons for concurrency control mechanisms.

Pessimistic concurrency control: Pessimistic locking only one application/user can have access to a single object at any one time. Pessimistic write locking many users/applications can obtain read permissions to an object but only one can have write permissions at any one time.

Optimistic concurrency control: ……

Transaction Committer:- Implements the Unit of Work Pattern (Fowler 184 ‘Maintains a list of objects affected by a business transaction and co-ordinates the writing out of changes and the detection and resolution of concurrency problems’).

Custom loading e.g. for Blobs to implement lazy load.

Need flexibility to extend domain-DB mappers for loading and updating. This can be maintained in the ClassDef if need some bizarre loading for one or more object can write custom loader this will be used instead of standard.

Relational database implementation.

Implementation:-

There will be many cases where the developer is working in environments where he/she will not be able to determine the database design to organizational standards or due to working with old databases (That cannot be changed because of all the reports running off of that etc). The ideal is that each object is globally uniquely identified via a Guid (In SQL Server a Unique Identifier). The reasons are numerous but are briefly other techniques (e.g. autonumbers) do not assign the id until the data is inserted into the database. This means that the identity of the object is unknown until it is persisted for the first time the complexity of managing and persisting related objects is therefore significantly increased. Techniques using composite natural keys provide an even worse set of problems typically the key values can be modified resulting in significant overhead in matching the updated object to the original record in the database as well as the overhead of managing relationships a huge increase in the potential and severity of concurrency control issues. However having said this because Habanero has evolved from real life practices over many years it has been developed to be able to handle all these scenarios.

I think it is generally better to err on the side of simplicity whenever possible. Writing a bunch of mindless stored procedures to perform every operation you think you may need is definitely not simple. I’m certainly not ruling out the use of stored procedures, but to start with procs? That seesm like a fairly extreme case of premature optimization to me. Jeff Atwood, codinghorror.com

Stored Procedures vs Dynamically generated SQL: In the Habanero architecture we prefer to use generated SQL to retrieve and update transactions into the database the reasons for this are many but are summarized below. Having however said this the Habanero Framework is sufficiently well designed and responsibility segregated that stored procedures can easily be used. It is however slightly more work for the application developer since the powerfull capabilities where the Habanero Framework dynamically creates optimised SQL is lost. With stored procedures the true ease of use of the Habanero Object relational mapping capabilities will not be realised. All the other advantages and benefits of Habanero are still however realised

Karl Seguin (codebetter.com)

Three major benefits of an O/R mapper

* + - 1. You end up writing less code – This results in more maintainable systems and quicker deliver.
      2. You gain true abstraction from the underlying data source – You are querying the OR mapper for your data not a database and the mapping between the table schema and domain objects is abstracted.
      3. Your code becomes simpler and less repetitive (DRY).
      4. DRY. The mapping is done in one place the updating, inserting, loading and searching all use the same mapping so you eliminate duplicate mappings.

Dynamic SQL is generated on the fly by a generic piece of code which gets various data as input and generates a parametrized query from it. Dynamically generated paramaterised sql will be cached on the server. In our case the SQL is dynamically produced from the Habanero Business Objects.

SQL dynamically generated from the Business object definition is the least brittle interface to the database possible.

|  |  |  |
| --- | --- | --- |
| **Issue** | **Dynamic SQL** | **Stored Procedures** |
| Maintainability | To add a new field onto a table and support it as a property in the object the database must be updated and the business object definition must be updated. That’s it two places. (As close to DRY as we can get)  Dynamically generated SQL provides for better testability. I.e. The system can run automated self test by testing that each and every property required by each and every class is represented appropriately in the database thus ensuring that the system is more robust than using stored procedures or non dynamically generated SQL. | Stored Procedures are in effect an API changes to an API must always be treated with extreme respect.  Adding a new field to a database that uses stored procedures requires one of two options  1) the existing stored procedure(s) are modified. This can cause a nightmare of ensuring that all versions of the software and all users of the stored procedure are updated at the same time.  2) A new stored procedure is created with the new field. This results in an explosion of stored procedures and difficult maintenance since you are never certain which are being used by whom.  Remember that there will usually be at least 3 stored procedures involved in this change, the select, update and insert.  In addition to the updating the stored procedure the business object have to have the new property and the business object reading updating and writing all need the new property added. |
| Sorting and searching. | Dynamic SQL can easily handle any combination of sorting and querying with minimal effort. Search by a different field no problem. Once again conforming to DRY. | The combinations of searching and sorting have to be known up from so that the stored procedure can have it built in. This add significant overhead to making relatively simple changes. |
| Sql Injection | Dynamically generated parameterized SQL cannot be subjected to SQL injection attacks. | Stored Procedures Cannot be subjected to SQL injection attacks. |
| Security | There are many options for implementing excellent security using Dynamic SQL. We have implemented all of these an even more sever security mechanisms when appropriate. Dynamic SQL can be used to provide even superior security to Stored procedures where the user has no permission to anything in the database see option 4.  1) Setting up role based security in the database where each user is assigned a role and that role is given permissions to certain tables.  2) Even each application and not each user is given permissions to the database. The password for the application is stored in the application using the appropriate encryption. No user ever has any permissions to do anything in the database.  3) Use views. Views can be set up in such a way that the users only ever have permissions to the views and never to any underlying tables. The users with inappropriate permissions can therefore never view sensitive information.  4) Based on the windows or custom security roles that the user has for the application the user is assigned to database roles for the particular connection being used. This provides the ultimate security but with performance and complexity overheads. | You set up the stored procedures and apply appropriate execution rights. The user need not have any permissions to any underlying tables. |
| Portability | Dynamic SQL can be built in such a way that it is fully portable to any relational database. The Habanero architecture allows the developer to move from MySQL to MS SQL, Oracle, FireBird, MSAccess etc with only a configuration change. Any currently unsupported Database can be supported with less than 4 hours work and once supported all systems developed using the Habanero framework will work on the new database. | Stored Procedures are difficult to port from on database to another and require rewriting and retesting to move. |
| Performance | For years we assumed that stored procedures gave performance advantages over Dynamic SQL. On one project we were however requested to test the performance loss as part of the decision making in choosing between stored procedures and Dynamic SQL. We know that ever since SQL 2000 the execution plan for Parameterised Dynamic SQL is cached in exactly the same manner as are stored procedures.  Well yes even knowing this we and everyone else was surprised with the dynamically generated parameterized SQL performed at least as well as stored procedures and in many cases significantly better. | |
| Using Dynamic SQL provides the capability to read data from ADO using or ordinal position instead of named variables. This has an order of magnitude performance advantage over using named fields. (see Section 4) |  |
| Execution Plan Cached | Execution Plan Cached |
| Updating data can easily be optimized to only update fields that are actually dirty (this is done with Habanero). As an extreme example if a Boolean field is changed then only that field is updated to the database and not the BLOB field.  In a similar manner loading and refreshing objects can be easily optimized to only load and refresh relevant data from the database. | The ability to have a stored procedure for every combination of fields that could be updated for each table would be prohibitive. In specialized cases e.g. the blob example two stored procedures could be created. This however makes the maintenance problems even worse than before. |
| Supports Incremental development and delivery | Dynamic SQL and Business objects provide the optimal condition for incremental development and incremental delivery preferred by agile development. The arguments are the same or similar to the maintainability arguments above.  Agile development methodologies are therefore significantly easier if you at least start with dynamic sql. Later you can change to stored procedures if there is an advantage. | Stored procedures are usable for incremental development and delivery but present more challenges. |

For all the above reasons we have selected to use Dynamic parameterized SQL as our primary mechanism for interfacing to Relational databases.

The Habanero architecture is designed such that stored procedures are fully supported it just requires the additional effort on behalf of the application developer to develop stored procedures.