National College of Ireland

Mobile Architecture & Security

Data Access Layer

Mikhail.Timofeev@ncirl.ie

8. Data Access Layer

① Introduction

② Requirements

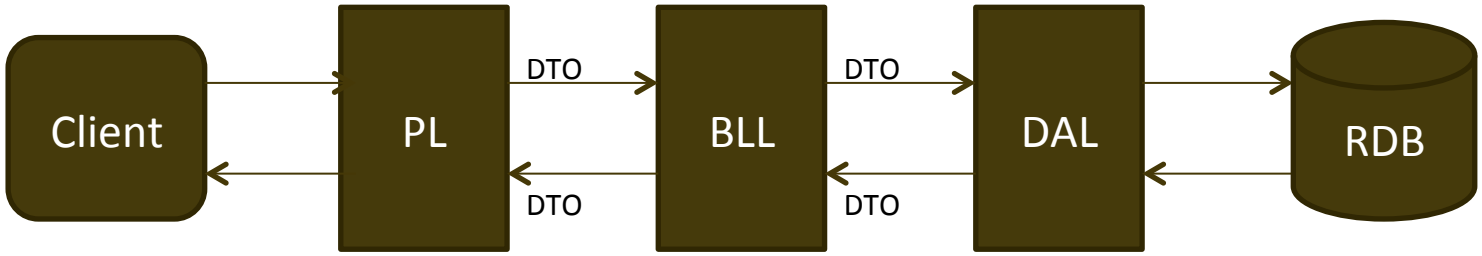
③ Interface

④ Patterns

Introduction

- Data Access Layer (DAL) is responsible for reading data from and persisting data to a data-store.
  - Typically, this is a relational database

- Ideally, the DAL is separated from each of the other layers in a layered architecture system.



Introduction

- In a less than ideal world…
  - We have seen, however, that the degree of separation is dependent on the manner in which we architect the solution
    - Transaction Script may make direct calls via prepared SQL statements to operate on the database
    - Table Module is also coupled with a given database instance through the use of objects representing database tables.
      - Recommended approach would be to implement the Table Data Gateway pattern to offer some degree of separation.
    - Active Record is also coupled with a given database instance through the use of objects representing database table rows.
      - Recommended approach would be to implement the Row Data Gateway pattern to offer some degree of separation.

Introduction

- In an world approaching an ideal …
  - Domain Model differs in that the creation of a Domain Model is driven from implementing a model-first design approach.
    - Domain Model is created independently of any particular database considerations
  - With the implementation of a Domain Model we have the opportunity to realise the Data Access Layer as a completely separate layer.
  - There is a major difficulty with this approach though …

The Object / Relational Impedance Mismatch

Requirements

- Requirements of the DAL
  - The DAL must offer the following capabilities
    - CRUD
      - Record creation
      - Record reading
      - Record updating
      - Record deletion
    - Transactional Integrity and Concurrency Control
      - Logically group a set of database operations as a single transaction
      - Manage database operations in such a manner to prevent reading or writing of invalid data
    - Query based operations
      - Ability to return sets of data that meet a specified set of criteria
    - Acceptable Performance
      - DAL must operate in accordance with expected service levels

Requirements

- Requirements of the DAL
  - We would also like the DAL to be independent of any particular physical database implementation
    - Ability to replace the RDBMS without having to make major changes to the DAL
    - Components that offer specific access to one vendor's database or another's should ideally be pluggable components into the DAL
      - Develop code to target an interface
      - Use an OR/M

Requirements

- CRUD
  - The DAL must have the capability to create new records and also have the capability to operate on records that already exist in the database
  - An object in the BLL for which a corresponding record(s) exists in the database is called a *persistent object*
    - E.g., a persistent record is created as a result of calling into the DAL's database READ capabilities to access some data
  - Those objects that exist in the BLL without a corresponding record(s) in the database are called *transient objects*
    - E.g., When creating a new record the BLL will pass a *transient object* to the DAL for ultimate insertion of the data into the database

Requirements

- CRUD
  - Reading Data from the Database
    - Many read type operations will be required time-and-time again from within the BLL for each of the BLL objects
      - Read Account by Id
      - Get all Customers from Ireland
      - Use a Repository to contain the queries
    - We would also like the capability to issue ad-hoc queries against the database if required
      - Find all Customers in the Leinster region with more than €10000 in outstanding invoices
      - Use a Query Object to generate the required SQL based on a given set of criteria. [Note: a Query Object is often placed in the Repository.]

Requirements

- Transaction Management
  - The DAL should offer the capability of logically grouping together a set of associated data operations
    - The Unit of Work pattern can be implemented to satisfy this requirement
    - Transaction based semantics are catered for
      - Begin transaction
      - Commit transaction
      - Rollback transaction
    - Corresponding database transactional operations are generated

Requirements

- Concurrency
  - In the BLL of the application object data will be modified. There will be direct connection with the actual database during such object-based operations leading to the possibility of violation of database ACID rules
    - Atomicity – all or nothing
    - Consistency – database must remain in a consistent state after transaction
    - Isolation – Incomplete transactions cannot impact on other transactions
    - Durability – Successfully committed transactions must persist

  - Usually, an optimistic standpoint is taken with respect to concurrency control within applications
    - Implementation of the Optimistic Offline Lock pattern

Requirements

- Ease of Use
  - In general, the DAL services and components can be wrapped into a super class structure that is an aggregation of the various different components of the DAL
    - ORM tools will generally offer some type of *context* object that is used to invoke DAL functionality

Interface

- Program to an Interface
  - Abstract out the DAL context to an interface

    - Separate out the actual DAL implementation from an interface to an implementation – Separated Interface Pattern
    - Consumers of the DAL are unaware of concrete implementations
    - Capability to plug-in different DALs without impacting on the other layers
      - New RDBMS implementation
      - Alternate data-storage technology
      - Increased testability and lesser maintenance effort required

    The interface should include declarations for CRUD operations, query based operations, and transaction operations support.

Interface

- Program to an Interface
  - Plug-in can be coded using a Factory implementation to provide a concrete implementation of a particular DAL
    - Clients consume the interface
    - Based on a specified set of configuration parameters
    - Associated libraries can be loaded dynamically at run-time
- Example

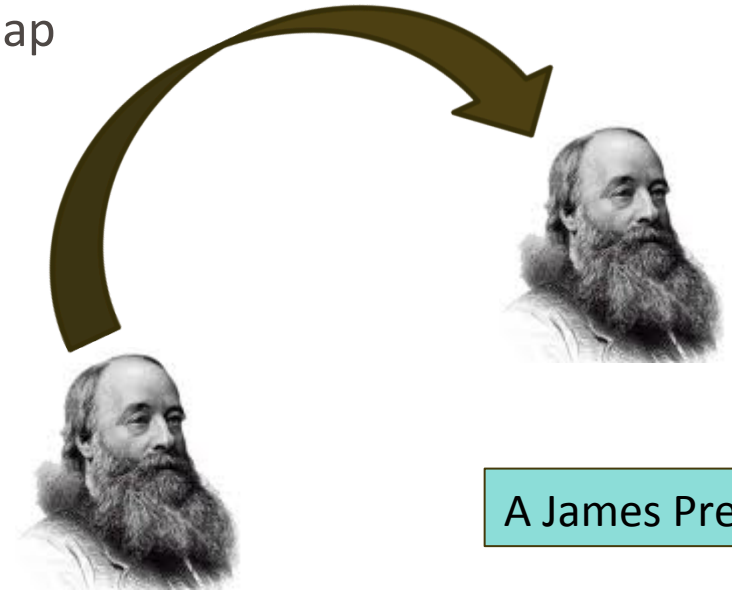
```
Public class DALFactory {
  private static IDataContext _instance = null;

  static DALFactory() {
     string asm = ConfigurationManager.AppSettings[“DALAssembly”];
     string classtype = ConfigurationManager.AppSettings[“DALType”];
     Assembly asmbly = Assembly.Load(asm);
     _instance = (IDataContext) asmbly.CreateInstance(classtype);
  }
  public IDataContext GetDataContext() {
     return _instance;
  }
```

Interface

- Program to an Interface
  - As an alternative to the plugin option we can also consider using *Dependency Injection* via an *Inversion of Control* (IoC) container.
  - A number of IoC containers exist in the marketplace
    - They can inject concrete object references into classes/objects automatically at run-time
    - Typically, there are three injection options available
      - Constructor Injection
      - Property Injection
      - Interface Injection

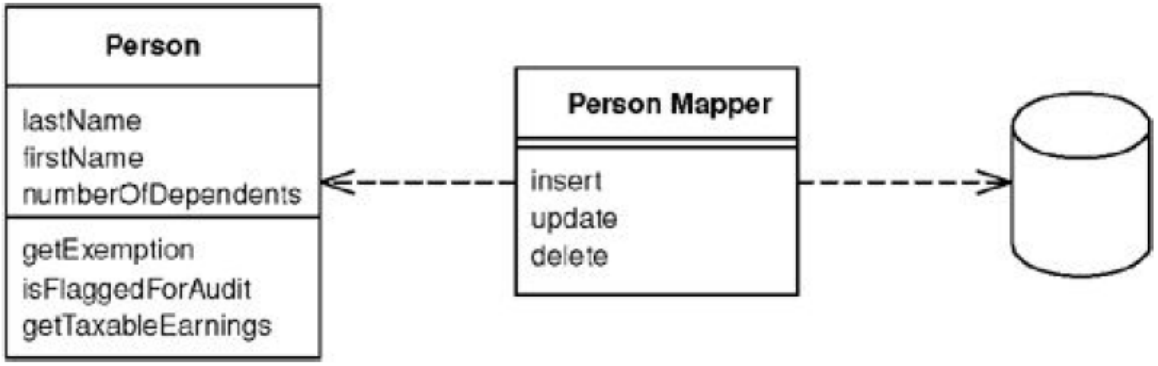
Patterns

- A Closer Look At Some DAL Patterns
  - DataMapper
  - Query Objects
  - Repository
  - Unit of Work
  - Identity Map



A James Prescott Joule Identity Map

Patterns: DataMapper



Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. Doing so leads to variant schemas; that is, the object schema and the relational schema don't match up.

You still need to transfer data between the two schemas, and this data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, changes in one tend to ripple to the other.

Martin Fowler – Patterns of Enterprise Application Architecture

Patterns: DataMapper

- In-memory objects can remain persistence ignorant
  - All database correspondence occurs through DataMapper
- Writing DataMappers is difficult
  - Classes have multiple fields
  - Classes have multiple tables
  - Classes with inheritance
  - Cater for concurrency control
  - Implement Unit-of-Work
  - Lazy-Loading strategies
  - Circular references
  - Implementation of Identity-Map
  - Etc.

- Recommendation is to BUY rather than BUILD

Patterns: DataMapper

- Typically, there will be a separate DataMapper for each domain entity
- DataMapper will implement CRUD operations, Query based operations, and Transactional operations => DataContext will utilise a DataMapper to implement functionality
- Can also code a factory for DataMappers
  - Concrete instance of a DataMapper delivered according to entity type
- An IoC container may also be used to inject concrete DataMapper references into application code in accordance with entity types

Patterns: DataMapper

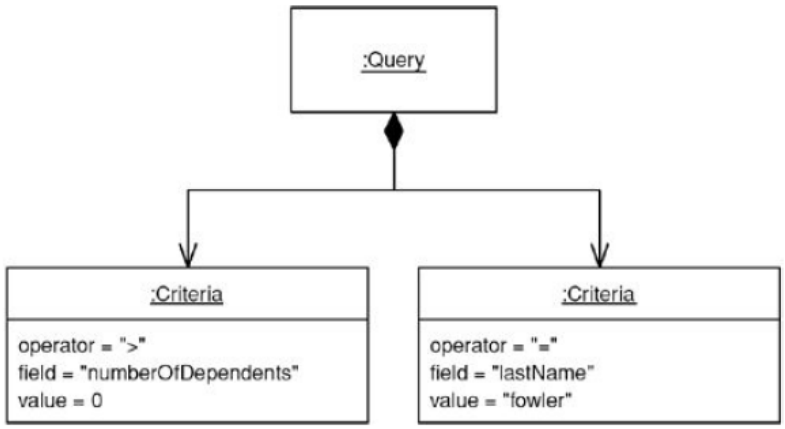
- DataMapper code will contain SQL commands to implement database operations
- DataMappers must also bridge the mismatch between the object graph and the relational database structure
- Must be aware also of connection management
  - Are we opening and closing connections successively
  - Can we batch transactions
  - Must be aware of transaction scope
  - What if multiple databases are involved?

Patterns: DataMapper

- The Repository pattern can be used to offer an interface to a DAL consumer via the DataMapper for regular query-based operations
  - i.e., the DataMapper also contains canned query-based methods

- For ad-hoc queries a Query Object may be created

An object that represents a database query.



SQL can be an involved language, and many developers aren't particularly familiar with it. Furthermore, you need to know what the database schema looks like to form queries. You can avoid this by creating specialized finder methods that hide the SQL inside parameterized methods, but that makes it difficult to form more ad hoc queries. It also leads to duplication in the SQL statements should the database schema change.

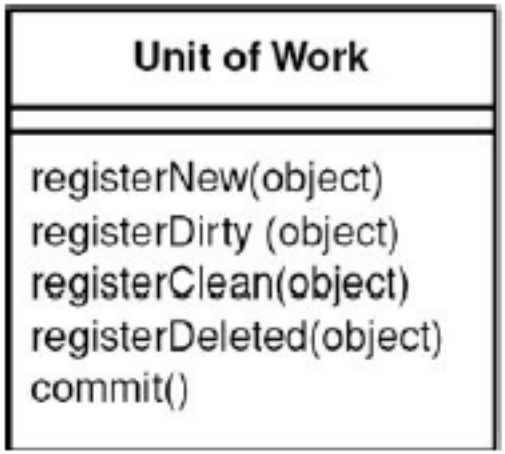
A Query Object is an interpreter [Gang of Four], that is, a structure of objects that can form itself into a SQL query. You can create this query by referring to classes and fields rather than tables and columns. In this way those who write the queries can do so independently of the database schema and changes to the schema can be localized in a single place.

Patterns: DataMapper

- Query Object
  - Flexible
  - ORM tools provide these capabilities out-of-the-box
  - Specify a set of criteria, members, and sort order etc.
  - May not be capable of taking care of all scenarios
    - Hand-code SQL
    - Use stored procedures

Patterns:
Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.

You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.

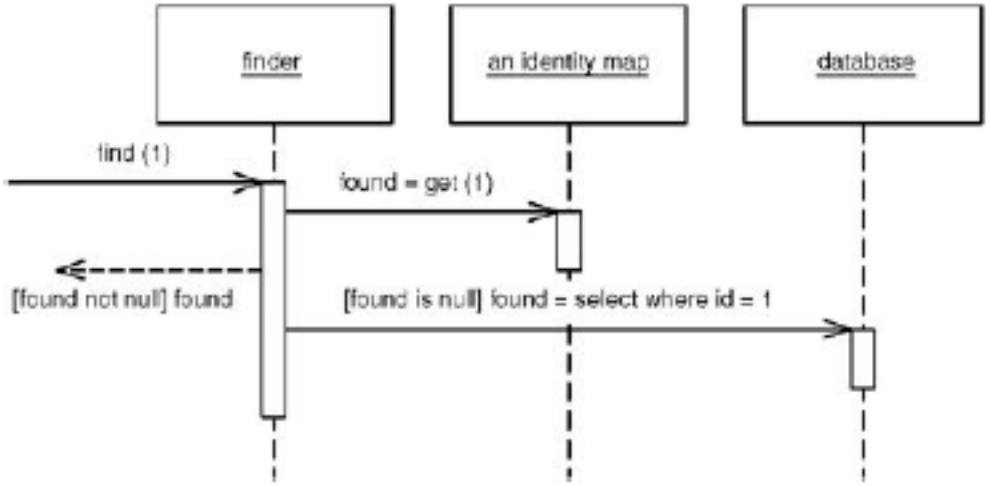
A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

Patterns:
Unit of Work

- The Service Layer will typically script business transactions and make calls into the DAL
- We use a Unit of Work to keep track of the changes that are affected during the course of running a business transaction
  - Unit-of-Work is informed of object creation, update, and delete
  - Unit-of-Work can also keep track of reads if required (to ensure that no records that have been read have changed since the Unit of Work started)
  - At *Commit* time the Unit-of-Work will
    - open a transaction
    - Check concurrency controls (Optimistic Offline Lock usually implemented)
  - Unit-of-Work informed of changes either by the caller or by the objects that are undergoing operation

Patterns:
Identity Map

Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.



An old proverb says that a man with two watches never knows what time it is. If two watches are confusing, you can get in an even bigger mess with loading objects from a database. If you aren't careful you can load the data from the same database record into two different objects. Then, when you update them both you'll have an interesting time writing the changes out to the database correctly.

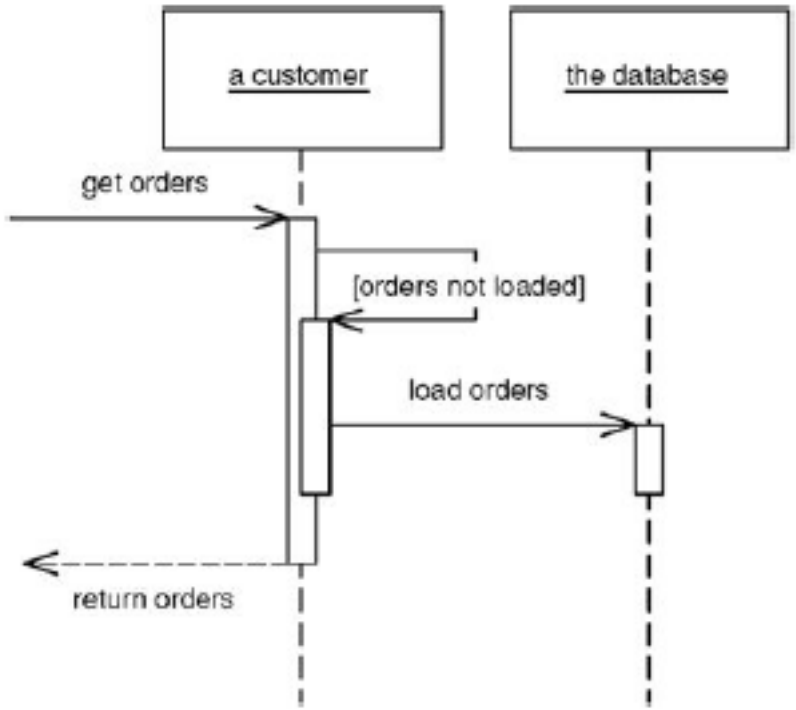
An Identity Map keeps a record of all objects that have been read from the database in a single business transaction. Whenever you want an object, you check the Identity Map first to see if you already have it.

Patterns:
Identity Map

- May be implemented as either an Explicit or Generic Identity Map
  - Generic will hold map data for all types of objects
  - Explicit will involve having methods associated with each specific domain entity
    - E.g.
      - Find(Person, 1)
      - findPerson(1)
- May have one map per class or one map per session
- Unit of Work is probably the best place to house the Identity Maps
- Can also split between Read-Only and Updatable

Patterns: Lazy Load

An object that doesn't contain all of the data you need but knows how to get it.



For loading data from a database into memory it's handy to design things so that as you load an object of interest you also load the objects that are related to it. This makes loading easier on the developer using the object, who otherwise has to load all the objects he needs explicitly.

However, if you take this to its logical conclusion, you reach the point where loading one object can have the effect of loading a huge number of related objects—something that hurts performance when only a few of the objects are actually needed.

A Lazy Load interrupts this loading process for the moment, leaving a marker in the object structure so that if the data is needed it can be loaded only when it is used. As many people know, if you're lazy about doing things you'll win when it turns out you don't need to do them at all.

Patterns:
Lazy Load

- Lazy Load
  - Lazy Initialization
    - Check if field null.
    - If null then calculate field value
      - What if null is valid?
    - Forces a dependency between database and object
  - Virtual Proxy
    - Takes place of target object – stands in as a proxy
    - When property is accessed then correct object is loaded
    - May be necessary to create a large number of proxy classes

Patterns:
Lazy Load

- Lazy Load
  - Ghost
    - Real object partially loaded
    - When loaded the object holds just the Id
    - When a field is accessed the rest of the data is loaded
  - Careful of lazy-loading collections
    - May take a performance hit

National College of Ireland

Questions?

Mikhail Timofeev
Office 3.18
MTimofeev@ncirl.ie