National College of Ireland

Mobile Architecture & Security

Business Logic Layer

Mikhail.Timofeev@ncirl.ie

7. Business Logic Layer

① Business Logic Overview

② Key Business Components

③ General Design Considerations

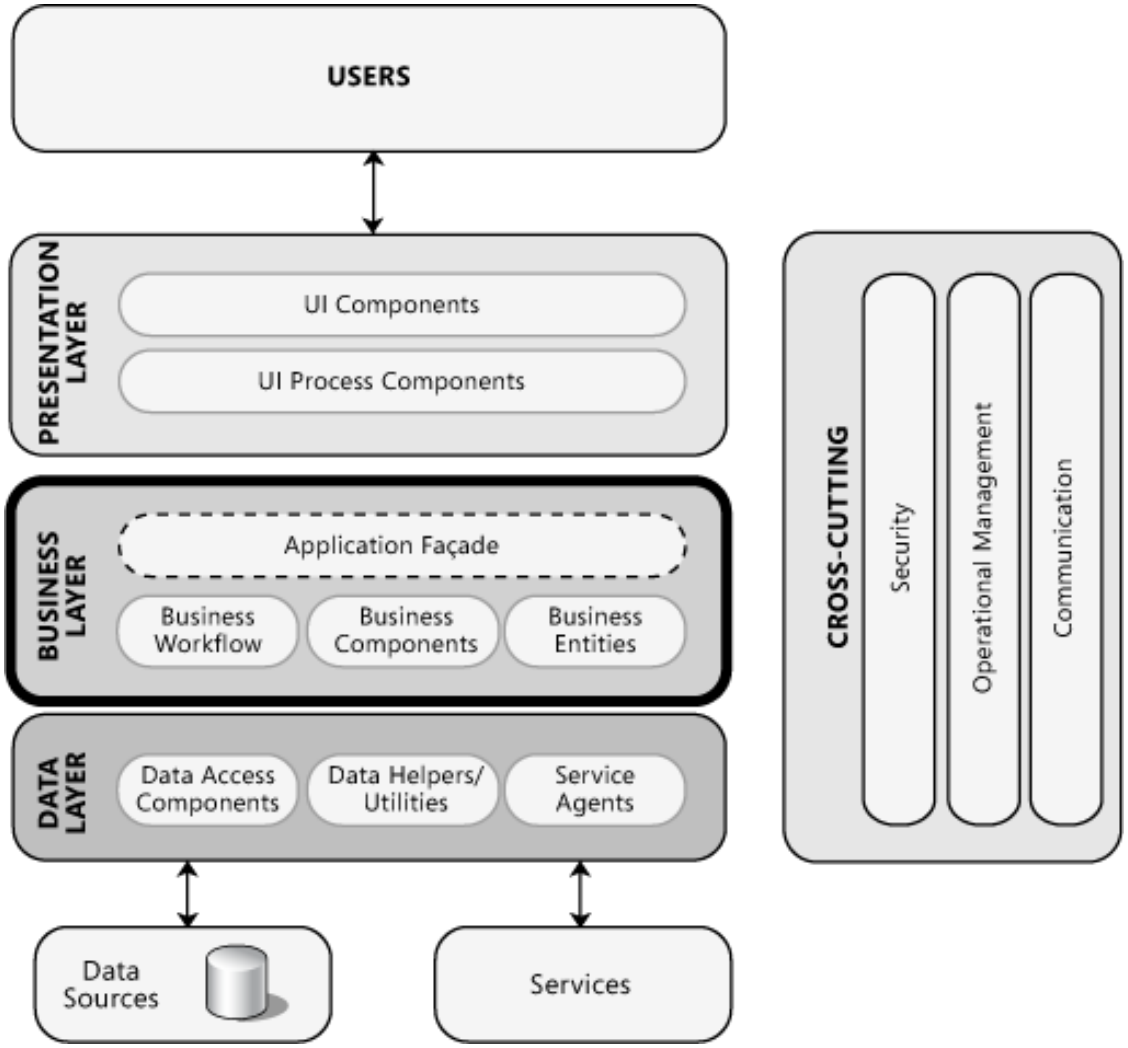
④ BLL Architectural Patterns

⑤ Specific Design Issues

⑥ Deployment Considerations

⑦ Design Steps for the Business Layer

Business Layer Overview



Business Layer Overview

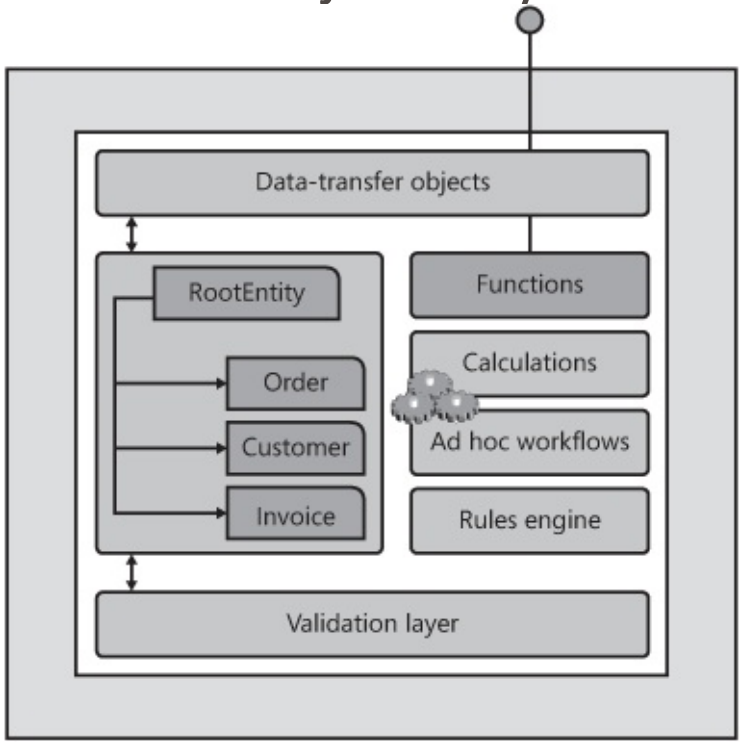
- Any software of any reasonable level of complexity is organised in layers
- Each layer represents a logical section of the system
- The modules in the business layer include all the functional algorithms and calculations that make the system work and interact with other layers, including the data access layer (DAL) and presentation layer
- The business layer is the nerve system of any layered system and contains most of its core logic
- It is often referred to as the Business Logic Layer (BLL)

Business Layer Overview

- BLL is the part of the software system that deals with the performance of business-related tasks
- It consists of an array of operations to execute on some data
- Data is modeled after the real entities in the problem's domain – invoices, customers, orders, etc.
- Operations try to model business processes or individual steps of a single process – creating an invoice, adding a customers, posting an orders, etc.
- In BBL you will find:
  - Object Model, which models business entities
  - Business Rules, that expresses all the customer's policies and requirements
  - Services, to implement autonomous functionalities
  - Workflows, that define how documents and data are passed around

Business Layer Overview

- Security is also a serious matter and must be addresses in all layers, especially BLL, where the code acts as a gatekeeper
- Security in the BLL means essentially role-base security to restrict access to business object only to authorised users



Key Business Components

The main components within the business layer:

- **Application façade.** This optional component typically provides a simplified interface to the business logic components, often by combining multiple business operations into a single operation that makes it easier to use the business logic. It reduces dependencies because external callers do not need to know details of the business components and the relationships between them.
- **Business Logic components**. Business logic is defined as any application logic that is concerned with the retrieval, processing, transformation, and management of application data; application of business rules and policies; and ensuring data consistency and validity. To maximize reuse opportunities, business logic components should not contain any behavior or application logic that is specific to a use case or user story. Business logic components can be further subdivided into the following two categories:

Key Business Components

- **Business entities**. Business components used to pass data between other components are considered business entities. The data can represent real-world business entities, such as products and orders, or database entities, such as tables and views. Consider using scalar values as business entities. Alternatively, business entities can be implemented using data structures such as DataSets and Extensible Markup Language (XML) documents.
- **Business workflows**. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. Business workflows define and coordinate long-running, multi-step business processes, and can be implemented using business process management tools.

General Design Considerations

- **Decide if you need a separate business layer**
  - It is always a good idea to use a separate business layer where possible to improve the maintainability of your application. The exception may be applications that have few or no business rules (other than data validation).
- **Identify the responsibilities and consumers of your business layer**
  - This will help you to decide what tasks the business layer must accomplish, and how you will expose your business layer. Use a business layer for processing complex business rules, transforming data, applying policies, and for validation. If your business layer will be used by your presentation layer and by an external application, you may choose to expose your business layer through a service.
- **Do not mix different types of components in your business layer**
  - Use a business layer to avoid mixing presentation and data access code with business logic code, to decouple business logic from presentation and data access logic, and to simplify testing of business functionality. Also, use a business layer to centralize common business logic functions and promote reuse.

General Design Considerations

- **Reduce round trips when accessing a remote business layer**
  - If the business layer is on a separate physical tier from layers and clients with which it must interact, consider implementing a message-based remote application façade or service layer that combines fine-grained operations into a smaller number of coarse-grained operations. Consider using coarse-grained packages for data transported over the network, such as Data Transfer Objects (DTOs).
- **Avoid tight coupling between layers**
  - Use the principles of abstraction to minimize coupling when creating an interface for the business layer. Techniques for abstraction include using public object interfaces, common interface definitions, abstract base classes, or messaging. For Web applications, consider a message-based interface between the presentation layer and the business layer.

BLL Architectural Patterns

- Architectural Patterns
  - Transaction Script
    - Simplest pattern for business logic

> Organizes business logic by procedures where each procedure handles a single request from the presentation.

> Most business applications can be thought of as a series of transactions. A transaction may view some information as organized in a particular way, another will make changes to it. Each interaction between a client system and a server system contains a certain amount of logic. In some cases this can be as simple as displaying information in the database. In others it may involve many steps of validations and calculations.
>
> A Transaction Script organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures.

Martin Fowler, Patterns of Enterprise Application Architecture

BLL Architectural Patterns

- Architectural Patterns
  - Transaction Script
    - Implements a direct mapping of business components to user actions
    - For each user action available through the Presentation Layer a method is created which is called a *Transaction Script*

Advantages	Disadvantages
Simplicity	Simplicity
Good choice for small applications with uncomplicated business logic	Code duplication
Transaction Scripts may be composed of reusable subroutines and components	Refactoring code to reuse common routines and components becomes more difficult as application complexity increases
	Violates OO design principles

BLL Architectural Patterns

- Architectural Patterns
  - Using Transaction Script
    - Transaction Script requirements can be based on use-cases
    - Each use-case action maps to a Transaction Script represented by a method in some *business component*
    - Business Components
      - May be derived from a common business component base class that implements common exception handling, authorisation, logging functionality etc.
      - Transaction Scripts can be grouped logically according to related functionality with related Transaction Scripts coded into the same business component. This is the most common approach
      - Alternatively, each Transaction Script can be associated with a single business component class
        - Implementation of the *Command* design pattern

BLL Architectural Patterns

- Architectural Patterns
  - Using Transaction Script
    - Transaction Scripts grouped in a business component

```
public class RevenueAPI
{
    public decimal RecognisedRevenue(int contractID, DateTime statusDate)
    {
        // Initialise result
        // Find recognitions for contractID as at statusDate
        // Sum recognitions
        // Return result
    }

    public void CalculateRevenueRecognitions(int contractID)
    {
        // Find contract for contractID
        // Compute total revenue for contractID
        // Get contract date
        // Create recognitions
    }
}
```

BLL Architectural Patterns

- Architectural Patterns
  - Using Transaction Script
    - Transaction Scripts using the Command pattern

```
public interface IApplicationCommand
{
    int Run();
}

public class RecognisedRevenue : IApplicationCommand
{
    private int _contractID;
    private DateTime _statusDate;
    private decimal _result;

    public RecognisedRevenue(int contractID, DateTime statusDate)
    {
        _contractID = contractID;
        _statusDate = statusDate;
    }

    public decimal Result { get {return _result;} }

    public int Run()
    {
        // Initialise result
        // Find recognitions for contractID as at statusDate
        // Sum recognitions
        // Return result
    }
}
```

Using the Command pattern may lead to a large number of small classes, however, implementing this pattern also allows for bundling up the individual Transaction Scripts into larger transactions

Data passed to Transaction Scripts via method signature, constructor, DTO, or some other technique

BLL Architectural Patterns

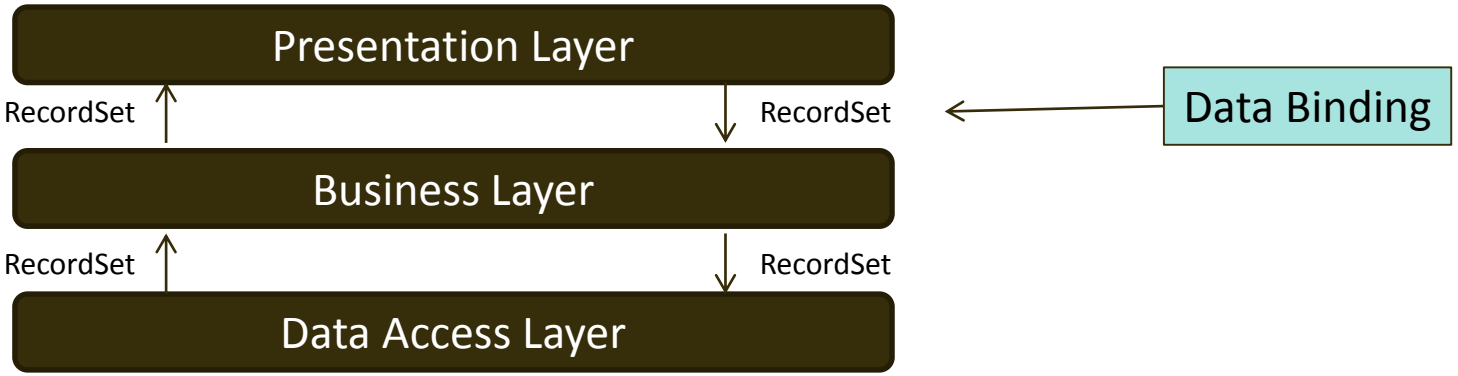
- Architectural Patterns
  - Table Module
    - Built around notion of a table of data

> A Table Module organizes domain logic with one class per table in the database, and a single instance of the class contains the various procedures that will act on the data.
>
> Patterns of Enterprise Application Architecture

  - Define a business component for each database table
  - Business methods exist at the table level

BLL Architectural Patterns

- Architectural Patterns
  - Using Table Module
    - Define a business component called a *Table Module* for each database table (or view)
    - Table Module packages data and behaviour together and plays to the strengths of a relational database
    - Table data usually held in a *RecordSet* object that mimics a database table
    - Data and related behaviours are close to each other – good encapsulation.
    - Good choice if Presentation Layer and Data Access Layer are based on tabular data structures



BLL Architectural Patterns

- Architectural Patterns
  - Using Table Module

Advantages	Disadvantages
Relatively simple to implement given a class library support (e.g., ADO.NET)	Cannot be used to express a complex graph of entities especially when significant gap between object model and data model
Degree of Object-Orientation & encapsulation	Difficult to implement from scratch
	TM uses objects but the objects do not model the business rules

BLL Architectural Patterns

- Architectural Patterns
  - Using Table Module
    - Class initialised with data
    - Methods on a Table Module do not match one-to-one with actions on the Presentation Layer => may require a simple service layer to script the actions on the Table Modules

```
public class CustomersManager
{
    private DataSet _data;

    public CustomersManager(DataSet data)
    {
        _data = data;
    }

    public DataTable Customers()
    {
        return _data.Tables[0];
    }

    public DataRow GetRow(int index)
    {
        return _data.Tables[0].Rows[index];
    }

    public DataRow GetRowByID(int customerID)
    {
        return _data.Tables[0].Select(...);
    }

    public int Update(DataRow row)
    {
        . . .
    }

        .
        .
        .
}
```

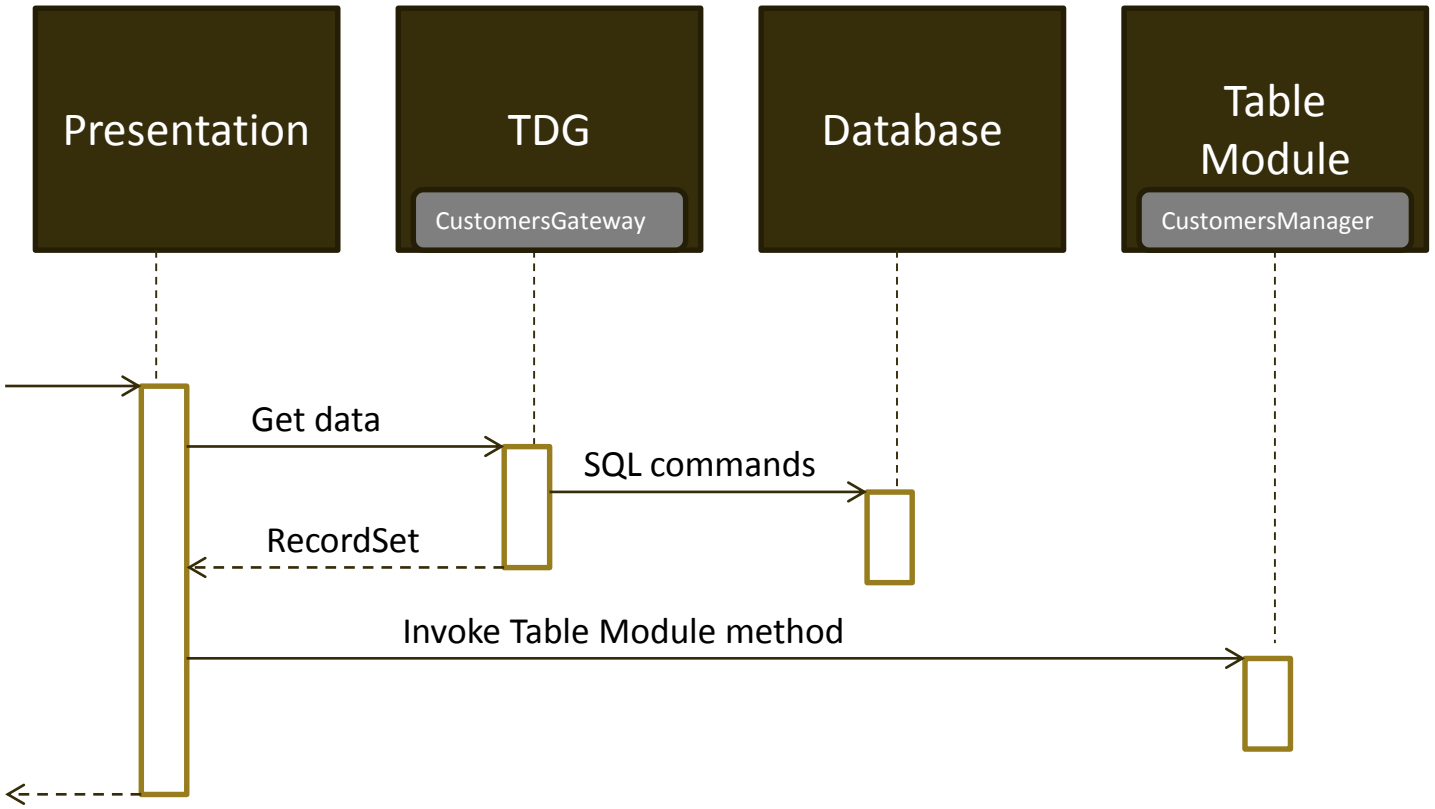
BLL Architectural Patterns

- Architectural Patterns
  - Using Table Module
    - Table Module requires some helper classes to physically access the database
      - Thin layer of code to fill required data structures with results of one or more queries
      - Data access helper classes are part of the DAL but are consumed by the Table Module classes
    - We call this layer of code the *Table Data Gateway* pattern

```
RecordSet customers = CustomersGateway.LoadByDate(dateFrom, dateTo);
CustomersManager customersManager = new CustomersManager(customers);
```

BLL Architectural Patterns

- Architectural Patterns
  - Table Module and the Table Data Gateway pattern



BLL Architectural Patterns

- Architectural Patterns
  - Active Record
    - Simple object model where the objects are essentially records with some extra methods.

> An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.
>
> Patterns of Enterprise Application Architecture

Typically, an Active Record class has the following methods:

- Construct an instance of the Active Record from a SQL result set row
- Construct a new row for later insertion into table
- Finder methods to wrap commonly used SQL queries
- Update the database and insert into it the data in the Active Record
- Get and set the fields
- Implement business logic

BLL Architectural Patterns

- Architectural Patterns
  - Using Active Record
    - Application viewed as a set of inter-related objects representing domain entities and objects responsible for performing calculations etc.
    - Active Record comes to the fore when the domain entities map closely to tables in the data model and domain logic is quite simple
    - Typical methods present in an Active Record:
      - CRUD operations
      - Validation
      - Domain specific functions
    - Active Record is a good choice when the correspondence between domain entities and relational tables is strong

BLL Architectural Patterns

- Architectural Patterns
  - Using Active Record

Advantages	Disadvantages
Conceptually simple and effective	Requires a DataMapper if Active Record classes require different perspective on data other than data model
Framework support (LINQ-to-SQL, Castle ActiveRecord)	Increasingly harder to modify if domain model and data model diverge
	Performance implications if dealing with large collections of Active Records

BLL Architectural Patterns

- Architectural Patterns
  - Using Active Record
    - Property names and types match database table column names and types
      - Data may be converted into more workable types if required
      - If further classes are to group related data columns from table we start to move towards a domain model approach
      - May not have option if relying on framework support
    - Multiple Active Records can be grouped in arrays or collection classes
    - Active Records are POCO / POJO
      - Plain Old CLR Objects
      - Plain Old Java Objects
    - Use a *Row Data Gateway* pattern to access physical data that is consumed by the Active Record class – many frameworks will provide this capability

BLL Architectural Patterns

- Architectural Patterns
  - Using Active Record

    - In memory object can be created to be inserted to database at later time
    - Load method loads object using data from database
    - Collections are managed through Arrays or Collection types

```
public class Customer
{
    public Customer()
    {
    }

    public Customer(int customerID)
    {
        // ...
    }

    public int CustomerID { get; set; }
    public string CustomerFirstName { get; set; }
    public string CustomerSurname { get; set; }
    public string AddressLine1 { get; set; }
    public string AddressLine2 { get; set; }
    public string AddressLine3 { get; set; }
    public string City { get; set; }
    public string CountryID { get; set; }
    public DateTime DOB { get; set; }

    // . . .

    public void Load(int customerID)
    {
        // Load Customer
    }

    public void Insert()
    {
        // Insert Customer record
    }

    public void Update()
    {
        // Update Customer record
    }

    // . . .
}
```

BLL Architectural Patterns

- Architectural Patterns
  - Using Active Record and the Foreign-Key Mapping Pattern
    - Active Record can be extended to hold references to other Active Record objects based on the foreign-key relationships that exist in the database
    - This is implemented as the Foreign-Key Mapping Pattern
      - Many frameworks support this pattern by default
      - May use deferred loading

```
public string City { get; set; }
public Country Country { get; set; }
public DateTime DOB { get; set; }
```

BLL Architectural Patterns

- Architectural Patterns
  - Domain Model
    - A more abstract object model representing the problem domain

> Putting a Domain Model in an application involves inserting a whole layer of objects that model the business area you're working in. You'll find objects that mimic the data in the business and objects that capture the rules the business uses. Mostly the data and process are combined to cluster the processes close to the data they work with.
>
> Patterns of Enterprise Application Architecture

    - Created as a result of a Domain Driven Design approach
    - Model-First design approach
    - Objects are not aware of any behaviour relating to persistence of information to a database
      - Persistence designated to a distinct de-coupled Data Access Layer

BLL Architectural Patterns

- Architectural Patterns
  - Using Domain Model
    - The domain model describes the business entities, their relationships with each other, and how data flows between them
    - Abstract domain model created prior to incorporation in software
    - Collaboration between domain experts, business analysts, and application architects key to successful description of domain model
    - Business logic can be reused between applications
    - Domain Model has higher initialisation costs than other approaches but the cost associated with enhancing the model tends to grow in a linear manner as a function of the complexity

BLL Architectural Patterns

- Architectural Patterns
  - Using Domain Model

Advantages	Disadvantages
OO design based (encapsulation, inheritance, composition, etc.)	May be difficult to describe and formulate the domain model
No constraint due to database	The O/R Impedance Mismatch
Requirements driven	

Recall: The Dependency Inversion Principle

BLL Architectural Patterns

- Architectural Patterns
  - The Object/Relational Impedance Mismatch
    - O/R Mapping refers to mapping a relational model to an object model.
    - There is a mismatch due to the differences between a domain model's object hierarchy and a normalised relation database
    - There are also differences in
      - data-types that need to be accounted for
      - Syntactical differences between query based language vs. OO programming language (declarative vs. imperative)
      - Refactoring of code cannot be necessarily reflected in the database schema (e.g., changing property names)
      - Data Independence Principle – database is designed, administered, maintained, secured, catalogued, accessed independently from any single application program that operates with the database.

Electronic Engineering:
Impedance is the measure of the amount that some object impedes (or obstructs) the flow of a current. Impedance might refer to *resistance*, *reactance*, or some complex combination of the two.

BLL Architectural Patterns

- Architectural Patterns
  - Designing the Domain Model
    - Domain classes may be derived from a *layer super-type*
    - The layer super-type can define how cross-cutting concerns will be implemented for domain entity classes
      - Validation
      - Logging
      - Security
      - Etc.
    - Use object composition to establish relationships
    - Domain Model does not contain code related to physical data access
      - Domain model responsible for exposing properties and methods
        - E.g., List of Customers, How to Find particular set of Customers
      - Persistence Layer responsible for loading data into domain objects.

BLL Architectural Patterns

- Architectural Patterns
  - Domain Model Logic
    - Logic to load or save state to storage is not contained within the domain objects
    - Code to create domain objects loaded with database data occurs in other classes within the Business Layer
    - Only business logic is implemented in domain objects

```
public virtual decimal GetOrdersTotal()
{
    decimal total = 0;
    foreach (Order order in this.Orders)
    {
        total += order.CalculatePrice();
    }
    return total;
}
```

Specific Design Issues

- Authentication
- Authorization
- Caching
- Coupling and Cohesion
- Exception Management
- Logging, Auditing, and Instrumentation
- Validation

Specific Design Issues: Authentication

- Avoid authentication in the business layer if it will be used only by a presentation layer or by a service layer on the same tier within a trusted boundary.
- If your business layer will be used in multiple applications, using separate user stores, consider implementing a single sign-on mechanism.
- If the presentation and business layers are deployed on the same machine and you must access resources based on the original caller's access control list (ACL) permissions, consider using impersonation. If the presentation and business layers are deployed to separate machines and you must access resources based on the original caller's ACL permissions, consider using delegation.

Specific Design Issues: Authorization

- Protect resources by applying authorization to callers based on their identity, account groups, roles, or other contextual information. For roles, consider minimizing the granularity of roles as far as possible to reduce the number of permission combinations required.
- Consider using role-based authorization for business decisions; resource-based authorization for system auditing; and claims-based authorization when you need to support federated authorization based on a mixture of information such as identity, role, permissions, rights, and other factors.
- Avoid using impersonation and delegation where possible because it can significantly affect performance and scaling opportunities. It is generally more expensive to impersonate a client on a call than to make the call directly.
- Do not mix authorization code and business processing code in the same components.
- As authorization is typically pervasive throughout the application, ensure that your authorization infrastructure does not impose any significant performance overhead.

Specific Design Issues: Caching

- Consider caching static data that will be reused regularly within the business layer, but avoid caching volatile data. Consider caching data that cannot be retrieved from the database quickly and efficiently, but avoid caching very large volumes of data that can slow down processing. Cache only the minimum required.
- Consider caching data in a ready to use format within your business layer.
- Avoid caching sensitive data if possible, or design a mechanism to protect sensitive data in the cache.
- Consider how Web farm deployment will affect the design of your business layer caching solution. If any server in the farm can handle requests from the same client, your caching solution must support the synchronization of cached data.

Specific Design Issues: Coupling and Cohesion

- Avoid circular dependencies. The business layer should know only about the layer below (the data access layer), and not the layer above (the presentation layer or external applications that access the business layer directly).
- Use abstraction to implement a loosely coupled interface. This can be achieved with interface components, common interface definitions, or shared abstraction where concrete components depend on abstractions and not on other concrete components (the principle of Dependency Inversion).
- Design for tight coupling within the business layer unless dynamic behavior requires loose coupling.
- Design for high cohesion. Components should contain only functionality specifically related to that component. Always avoid mixing data access logic with business logic in your business components.
- Consider using message-based interfaces to expose business components to reduce coupling and allow them to be located on separate physical tiers if required.

Specific Design Issues: Exception Management

- Only catch internal exceptions that you can handle, or if you need to add information. For example, catch data conversion exceptions that can occur when trying to convert null values. Do not use exceptions to control business logic or application flow.
- Design an appropriate exception propagation strategy. For example, allow exceptions to bubble up to boundary layers where they can be logged and transformed as necessary before passing them to the next layer. Consider including a context identifier so that related exceptions can be associated across layers when performing root cause analysis of errors and faults.
- Ensure that you catch exceptions that will not be caught elsewhere (such as in a global error handler), and clean up resources and state after an exception occurs.
- Design an appropriate logging and notification strategy for critical errors and exceptions that logs sufficient detail from exceptions and does not reveal sensitive information.

Specific Design Issues: Logging, Auditing and Instrumentation

- Centralize the logging, auditing, and instrumentation for your business layer. Consider using a library such as patterns & practices Enterprise Library, or a third party solutions such as the Apache Logging Services "log4Net" or Jaroslaw Kowalski's "NLog," to implement exception handling and logging features.
- Include instrumentation for system critical and business critical events in your business components.
- Do not store business sensitive information in the log files.
- Ensure that a logging failure does not affect normal business layer functionality.
- Consider auditing and logging all access to functions within business layer.

Specific Design Issues: Validation

- Validate all input and method parameters within the business layer, even when input validation occurs in the presentation layer.
- Centralize your validation approach to maximize testability and reuse.
- Constrain, reject, and sanitize user input. In other words, assume that all user input is malicious. Validate input data for length, range, format, and type.

Deployment Considerations

- Consider deploying the business layer on the same physical tier as the presentation layer in order to maximize application performance, unless you must use a separate tier due to scalability or security concerns.
- If you must support a remote business layer, consider using the TCP protocol to improve application performance.
- Consider using Internet Protocol Security (IPSec) to protect data passed between physical tiers.
- Consider using Secure Sockets Layer (SSL) encryption to protect calls from business layer components to remote Web services.

Design Steps for the Business Layer

- **Create a high level design for your business layer**. Identify the consumers of your business layer, such as the presentation layer, a service layer, or other applications. This will help you to determine how to expose your business layer. Next, determine the security requirements for your business layer, and the validation requirements and validation strategy. Use the guidelines in the "Specific Design Issues" section earlier in this chapter to ensure that you consider all of the relevant factors when creating the high level design.
- **Design your business components**. There are several types of business components you can use when designing and implementing an application. Examples of these components include business process components, utility components, and helper components. Different aspects of your application design, transactional requirements, and processing rules affect the design you choose for your business components.

Design Steps for the Business Layer

- **Design your business entity components**. Business entities are used to contain and manage business data used by an application. Business entities should provide validation of the data contained within the entity. In addition, business entities provide properties and operations used to access and initialize data contained within the entity.
- **Design your workflow components**. There are many scenarios where tasks must be completed in an ordered way based on the completion of specific steps, or coordinated through human interaction. These requirements can be mapped to key workflow scenarios. You must understand how requirements and rules affect your options for implementing workflow components.

National College of Ireland

Questions?

Mikhail Timofeev

Office 3.18

MTimofeev@ncirl.ie