**ISW NOTES 2**

**UNIT 6: PERSISTENCE DESIGN**

1.-INTRODUCTION

In most applications the storage of non-volatile information is essential. A specific format may be used for each application. A structured or relational format based on DB may be used. The use of DBs results in using libraries to manage the access to data.

2.-DAO, REPOSITORY AND UOW PATTERNS

DAO, repository and UoW patterns are **implementation bridges between**:

* Data stored in objects.
* Data stored in a relational DB.
* Having methods to add, update, search and remove records.
* Encapsulating the necessary logic to copy data values from classes of the problem domain to the DB and vice versa.

The **elements of a DAO pattern** are:

* **BusinessObject**: object of the business layer that needs access to the data storage to read or write information.
* **DataAccessObject**: abstraction of the implementation of the data access layer. BusinessObject delegates the DAO all read write operations.
* **DataTransferObject** (DTO): represents an object holding data. DAO may return data to BusinessObject by means of a DTO. The DAO may receive data in a DTO to update the DB.
* **DataSource**: implementation of the data source.

For simplicity, the **BusinessObject** and **TransferObject** may **become just the first** one.

**Steps for a DAO pattern implementation**:

1. Take as starting point the Business Logic classes
2. Define an interface for each DAO. A DAO interface for each domain class with CRUD operations Create Read Update Delete and any other needed operations.
3. Define a class for each interface implementing its functionality. This class will know the details about how to access the data.

\*\* TYPICAL QUESTION \*

The main **advantages** of the DAO pattern are:

* **Encapsulation**: objects of the business layer do not know specific details of the implementation of the data access.
* **Easier migration**: migrating to a different DBMS just involves changing the DAO layer.
* **Less complexity** in the business layer because the access to data is isolated.
* Data access **centralized** in a layer.

The main **disadvantages** of the DAO pattern are:

* **Software architecture** slightly **more complex**.
* **Additional code for the layer** must be developed.
* From an efficiency perspective the **process may be slower**.
* **Coupled to the fields of domain objects**.
* May **affect maintainability of the code**.

\*\*

A **Repository** mediates between the domain and data mapping layers using a collection like interface for accessing domain objects. The interface is fixed (well defined contract) and independent of the fields of classes.

A **Repository** provides an abstraction so that data can be accessed as if it was an in-memory collection. Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with database concerns. Using this pattern can help achieve loose coupling and make business objects persistence ignorant.

The **simplest approach**, especially with an existing system, is to **create a new Repository** **implementation** **for each business object** you need to store to or retrieve from your persistence layer. Further, you should only implement the specific methods you are calling in your application. The biggest benefit of this approach is YAGNI (**You ain't gonna need it**), you won’t waste any time implementing methods that never get called.

**Another approach** is to go ahead and **create a simple, generic interface** for your Repository.

**DAO** is much closer to the underlying storage, it’s really data centric. That’s why in many cases you’ll have DAOs matching DB tables or views 1 on 1. A DAO allows for a simpler way to get data from a storage, hiding the ugly queries. But the important part is that they return data as in **object state**.

A **repository** sits at a **higher level**. It deals with data too and hides queries and all that but, a repository deals with business/domain objects. A repository will use a DAO to get the data from the storage and uses that data to restore a business object. Or it will take a business object and extract the data that will be persisted.

A **Unit of Work** (UoW) maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

**The Unit of Work pattern isn't necessarily something that you will explicitly build yourself**, but the pattern shows up in almost every modern persistence tool:

* The **ITransaction** interface in Nhibernate.
* The **DbContext** class in the Entity Framework.

Persistence provides **access to a data source**. The services provided by the persistence layer are specified again as an **interface**. There are **different implementations** of the interface may be given depending on the concrete data source. By using an interface any change in the implementation of IDAL **does not affect** the business logic layer. IDAL implements a generic repository pattern.

3.-PERSISTENCE IN ORDB AND OODB

In complex systems it is tedious to convert data between OO and Relational models, like mapping features from programming language to SQL and vice versa. There are several tools that perform automatically this mapping for several languages (Entity Framework).

Instead of implementing a relational DB an **OODB is provided**:

* The internal storage represents objects as such.
* No object relational or object SQL middleware.
* Most operations are implemented in a more efficient way, no need to manage data in different relational tables.

There are important associated **advantages**:

* The power of objects with the flexibility of query languages all together.
* The development of layered applications is simplified. No need for a persistence layer based on SQL and a business logic based on objects. The logic layer communicates with the persistence layer by means of objects without conversion mapping. Constant access in terms of “object.member”.

**UNIT 7: GRAPHICAL USER INTERFACE**

1.-CREATING A BASIC WINDOWS APPLICATION

The creation of **Visual Apps for Windows** may be done, among others with the namespace System.Windows.Forms which includes classes, structures, interfaces, etc. to develop these types of applications.

The namespace System.Windows.Forms includes the following classes:

* **Application**: the core of a Windows app. Its methods are used to process Windows messages and visual apps are created and destroyed.
* **Form**: represents a window or a dialog box in a visual
* **Button, ListBox, TextBox, PictureBox, Label**,...: Providing the functionality of common Windows controls.
* **StatusBar, ToolBar**,...: windows utilities.
* **ColorDialog, FileDialog**,...: standard dialog
* **StripMenu, StripMenuItem**,…: use to create different types of menus.
* **ToolTip, Timer**,...: to ease the interactivity of.

2.-FORMS WITH CONTROLS

3.-EVENTS IN FORMS

4.-DESIGNING AND USING MENUS

5.-APPS WITH SEVERAL FORMS

5.1.-DESIGNED BY THE CODER

5.2.-DIALOG FORMS

6.-DISPLAYING DATA SETS

7.-ADVANCED OPERATIONS: VISUAL INHERITANCE