

A Workbench for Learning Enterprise Patterns

Paulo Sousa

GECAD – Knowledge Engineering and Decision Support Group

Instituto Superior de Engenharia do Porto

Porto, Portugal

psousa@dei.isep.ipp.pt

Abstract – design patterns capture proven solutions for recurring problems. However the solution itself is not reusable off-the-self. It must be adapted for each specific problem. Most businesses run enterprise applications with mission critical data and functionality. Developers of these enterprise applications have recognized the value of design patterns in this scenario and detected specific design patterns for Enterprise Application Architectures. This paper presents a learning tool showing several architectural styles currently used in undergraduate and graduate programs.

Keywords: patterns, enterprise application architecture.

1 Introduction

When starting the development of a new enterprise application, one is faced with questions such as how to divide the application? How to represent the business entities? How to code the business logic? How to persist the business entities? How to guarantee the coherence of data in the database? How to handle the interaction with the user? and how to handle distribution aspects of the application? These are common problems for every enterprise application project and the solutions for some of these questions were already found in praxis of diverse development teams. Martin Fowler in his book “Patterns of enterprise Application Architecture” [4], describes exactly the patterns one can use for solving these problems. This is a great book with lots of knowledge sharing and advice, and a great catalogue of patterns that I use as a textbook for a class on Design Patterns and software Architectures on a post-graduate course on Enterprise Application Engineering done at ISEP. The work presented in this paper tries to provide a concrete example for the patterns described in Martin Fowler’s catalogue of patterns for Enterprise Application Architecture for layered enterprise application architectures.

The concept of patterns appears in the architecture field by the 1970s by an architect called Christopher Alexander [1] [2]. In his work “A pattern language” [1], Christopher Alexander defines that “each pattern describes a *problem that occurs over and over again* in our environment and then describes the *core of the solution* to that problem in such a way that you can *use this solu-*

tion a million times over without ever doing it the same way twice.”. The concept has many similarities with the software development industry and as such, by 1996, Erich Gamma, Richard Helm, Ralph Johnson and the late John Vissides (who would become known as the Gang of Four) define the concept of a design pattern such that it “names, abstracts, and identifies the key aspects of a common *design structure* that make it useful for creating a *reusable object-oriented design*” [5].

Three layered architectures divide the application in presentation, business logic and data access logic. The presentation layer is mainly responsible for interaction handling and performs no logic whatsoever (except maybe some basic forms validation for usability issues); the business logic layer is responsible for implementing the business rules of the application, by ignoring how the data is presented or captured from the user and ignoring how the data is persisted and read; the data access layer is responsible for accessing the persistence mechanisms such as database management systems or file systems (one must note, that a generic package such as JDBC or ADO.net is not a data access layer, neither is a DBMS. The data access layer is the code we built – or use code generators – to call such generic packages).

The paper is organized as follows: section 1 just presented the motivation and background for the work. Section 2 will describe the example problem and the general architecture of the workbench application. Section 3 will describe four architectural styles for implementing the example problem; two of them are data oriented, and the other two are object oriented. Finally in section 4 some concluding remarks are presented.

2 The workbench

2.1 Example Problem

The workbench uses the example problem of *Revenue Recognition* described in [4]. In this problem, the main goal is to calculate future revenues a company will have based on the payment conditions of its sale agreements.

The company sells three kinds of products (word processors, spreadsheets, and databases), each with different payment conditions:

- *Word Processors* – paid in full at acquisition (t)

- *Databases* – three “equal” payments at acquisition, 30 days and 60 days after ($t, t+30, t+60$)
- *Spreadsheets* – three “equal” payments at acquisition, 30 days and 90 days after ($t, t+30, t+90$)

For each contract with a customer, it is necessary to calculate the revenue and respective date it will occur. Figure 3 presents a conceptual model of the problem.

The requirements posed to the application are essentially:

- Calculate the revenue recognition of a contract
- Return the revenue of a contract as of a certain point in time

In order to answer these requirements one can define a business interface along the lines of:

```
public interface IRevenueRecognition
{
    Money RecognizedRevenue (
        int contractID, DateTime asOf);

    void CalculateRevenueRecognitions (
        int contractID);
}
```

2.2 Workbench Architecture

The workbench application (Figure 1) was developed using Microsoft Visual Studio 2003 and the .Net framework version 1.1, although the model and code can, in its majority, be used in other platforms (e.g., Java).

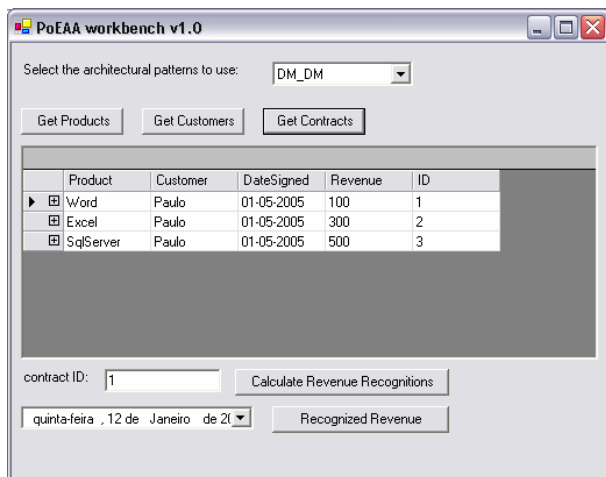


Figure 1 - workbench application's GUI

There is a separated layer (.net assembly) for the user interface and a layer with common data types (such as Money) to be used by all the layers in the application. The *RevenueFacade* layer defines the business interface as previously described and also defines an *Abstract Factory* [5] for the creation of business layer implementations according to the desired architectural style.

The current implementation of the workbench provides examples of the following architectural styles:

- Transaction Script using Record Set
- Transaction Script + Data Gateway (Record Set)
- Transaction Script using Custom Classes

- Transaction Script + Row Data Gateway (Custom classes)
- Table Module + Table Data Gateway (Record Set)
- Table Module + Table Data Gateway (Custom Classes)
- Domain Model + Active Record
- Domain Model + Data Mapper

The *RevenueFacade* layer defines an enumerated type for these architectural styles in the factory class. The factory method then uses reflection to dynamically create an object with the desired implementation. This object is a realization of the *Facade* pattern [5] hiding the complexity of calling the real business layer and data access layer objects. The next section will provide a detailed description of the last four combinations, and the last sub-section describes in more detail the implementation of the *RevenueFacade* layer.

3 Architectural Styles

3.1 Table Module + Table Data Gateway with Record Set

This architectural style basically divides the application in three layers (Figure 2) where the business logic is organized around the *Table Module* [4] pattern and the data access layer is organized around the *Table Data Gateway* [4] pattern.

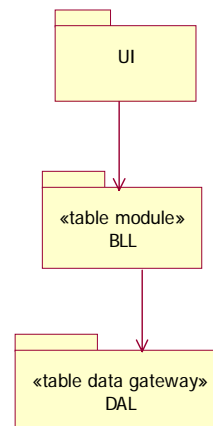


Figure 2 - TM + TDG

A *Table Module* is a business logic pattern where a single instance handles the business logic for all rows in a database table or view. A *Table Data Gateway* is an object that acts as a *Gateway* to a database table (one instance handles all the rows in the table). These two patterns provide a decomposition of the business and data layer directly related to the database schema, providing a good balance between decomposition, ease of maintenance and flexibility.

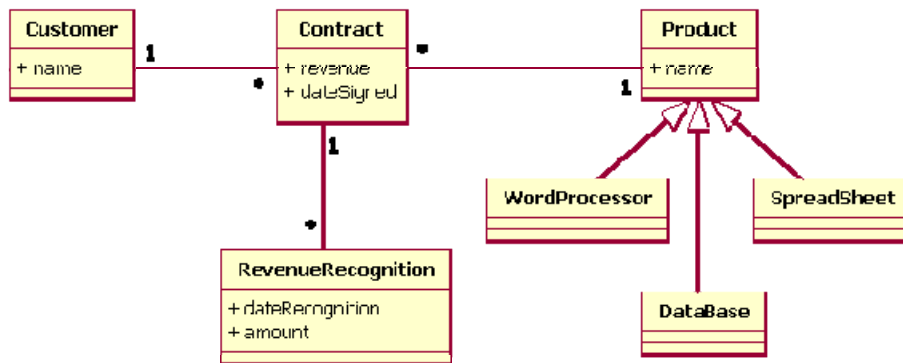


Figure 3 - conceptual model

They are particularly useful in conjunction with *Record Set* [4] (e.g., ADO.net's *DataSet* or JDBC's *ResultSet*) specially if there is good support for data binding between Record Sets and UI widgets.

Table Data Gateway is a good opportunity for code generation techniques

The classes of the business logic layer are:

```

public class Contract
{
    public Money RecognizedRevenue(
        int contractID, DateTime asOf)

    public void CalculateRevenueRecognitions(
        int contractID)

    public DataSet GetContracts()

    public DataSet GetContractsByProduct(
        int productID)

    public DataSet GetContractsByCustomer(
        int customerID)
}

public class Customer
{
    public DataTable GetCustomers()
}

public class Product
{
    public DataTable GetProducts()
}
  
```

In the data access layer, there are the following classes:

```

public class ContractGateway
{
    public int InsertRecognition(
        int contractID,
        DateTime recognitionDate,
        decimal amount)

    public void DeleteRecognitions(
        int contractID)

    public DataSet GetContractByID(
        int contractID)

    public DataSet GetContractsByProduct(
        int productID)

    public DataSet GetContracts()
}
  
```

```

public DataSet GetContractsByCustomer(
    int customerID)
}

public class CustomerGateway
{
    public DataTable GetCustomers()
}

public class ProductGateway
{
    public DataTable GetProducts()
}
  
```

Additionally there is an abstract base class for all gateways that provides common data access functionality (e.g., open connection, fill a dataset).

Figure 4 present the sequence of messages exchanged between the objects in the application in order to perform the *Calculate Revenue* business operation. In this scenario, the business entities (the data) are passed along layers in record set objects (e.g., *DataTable*).

3.2 Table Module + Table Data Gateway with custom Classes

This architectural style is similar to the one previously described, with an additional package for the business entities in the problem. Instead of using a *Record Set* for passing data around, *Custom Classes* [3] are built to that specific purpose. These classes are typical data structures with no behaviour associated (since the business logic is coded in the table Module classes).

The main difference between this style and the previous one is in the use of the custom classes as data holders. As such, the methods of the business logic layer classes have been modified to return *IList* instead of *DataSet/DataTable*.

The data access layer classes are also very similar to the ones presented. An important difference in this Table Data Gateways is the presence of private methods for constructing the custom class object given a *DataRow* read from the database. These methods perform the mapping between the data access library (ADO.net) and the program specific classes for holding data.

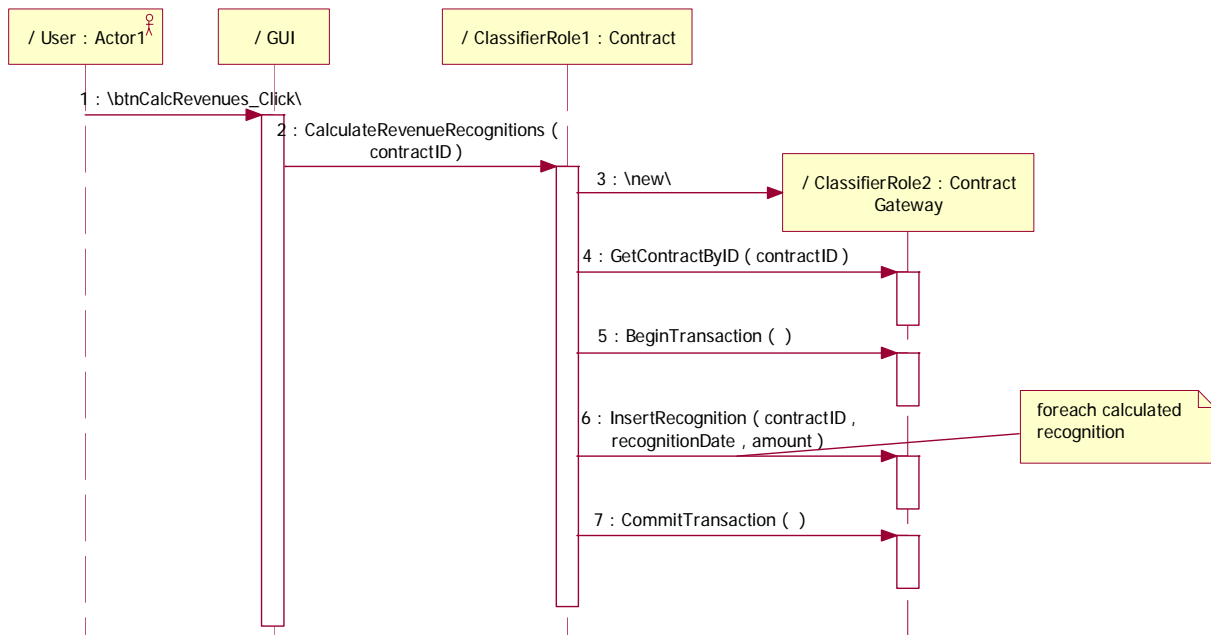


Figure 4 - TM + TDG: sequence CalculateRevenues

3.3 Domain Model + Active Record

In this scenario, the structure of packages is simplified (Figure 5).

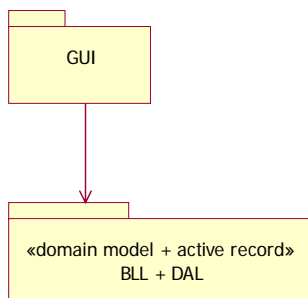


Figure 5 - DM (AR): packages

The business logic is implemented in a *Domain Model* (an object model of the domain that incorporates both behavior and data – a classic OO model of the problem [4]) that acts as an *Active Record* (encapsulating a row in a database and its access along with the business logic methods [4]).

This combination is very interesting in the sense that it is object oriented while simplifying the decomposition of the application and providing a natural place for the data access logic (an objects knows how to handle its business as well as how to load it self from and save itself to the database). Typically, load methods are implemented as class methods (*static* in java or C#) or in separated finder classes. To some, this is an unnatural three layer architecture since there are only two layers. The question here is that a layer is something logical; as long as the data access code is separated from the business logic code it's a layered architecture, even in this

situation where the separation is only at method level and not at class level as usual.

The class diagram for the business classes as active records is a classic OO model where you have a network of related classes for modelling the problem at hand. For instance, a *Contract* object is now in charge of representing a specific contract and has a relation with a *Product* object and customer object for that specific contract. You also use the contract object to save the changes you made to the database as well as use class methods in the contract class to load a specific object in to memory.

Figure 6 you can see the sequence of operations for *Calculate Revenue*.

In this scenario a protected constructor is used by the class load methods to perform the mapping between the *DataRow* and the object's internal data structure. This pattern provides a good way to do OOP without abstracting the database to much. It is particularly useful when the database schema is relatively stable and adequate from an OO perspective (that is, the class's attributes and the table fields are similar). However, in some situations one wants to be independent of the database schema and that's what the next architectural style is all about.

3.4 Domain Model + Data Mapper

The final scenario (Figure 9) uses the same *Domain Model* pattern for the business logic but the data access logic is organized around the *Data Mapper* pattern. A *Data Mapper* is an object that moves data between objects and a database while keeping them independent of each other and the mapper itself [4]. In this scenario, the business classes only have attributes and business related behaviour.

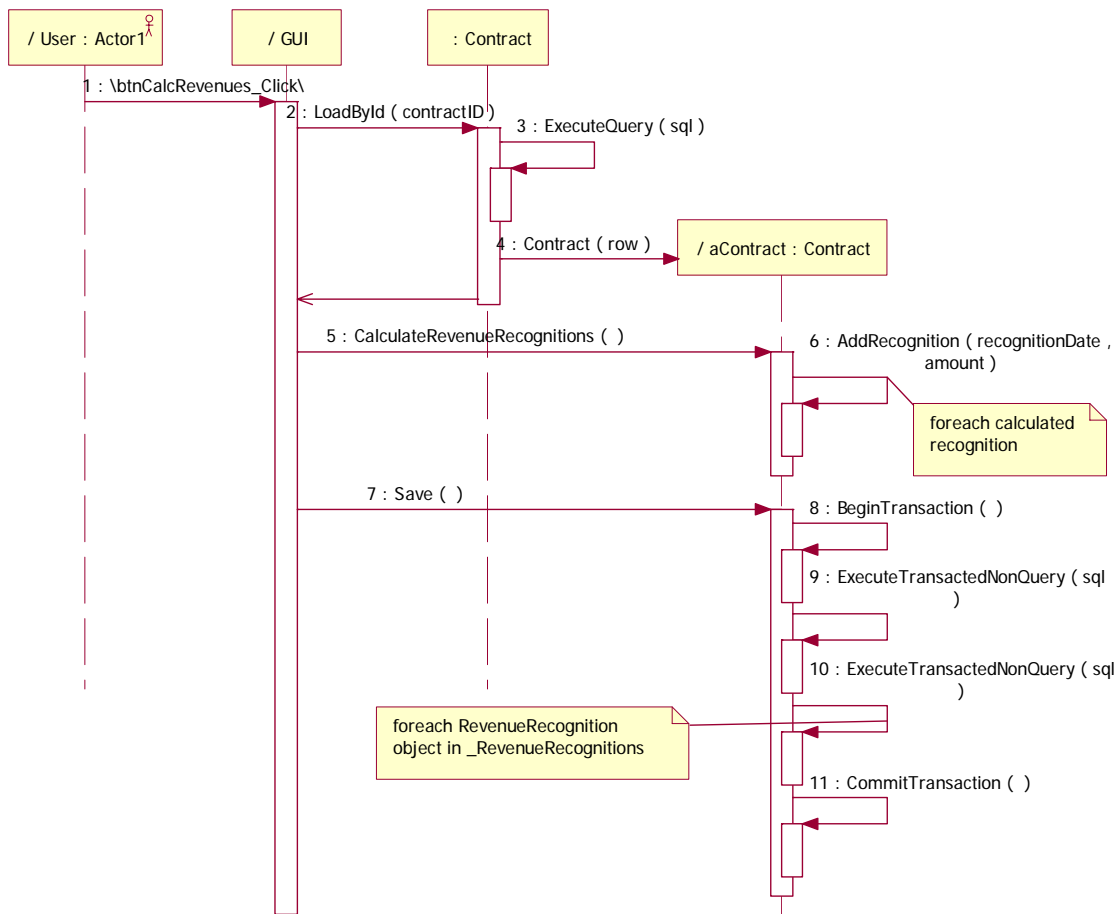


Figure 6 - DM (AR): sequence CalculateRevenues

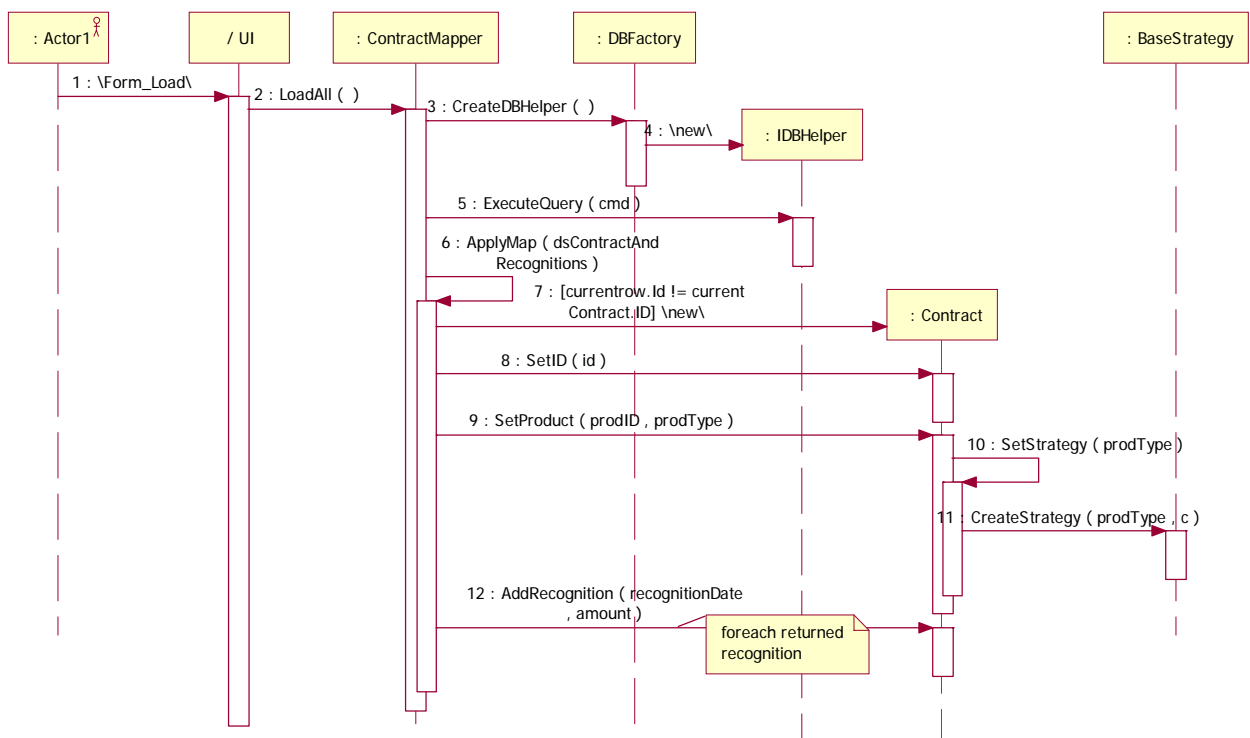


Figure 7 - DM + DM: sequence GetContracts

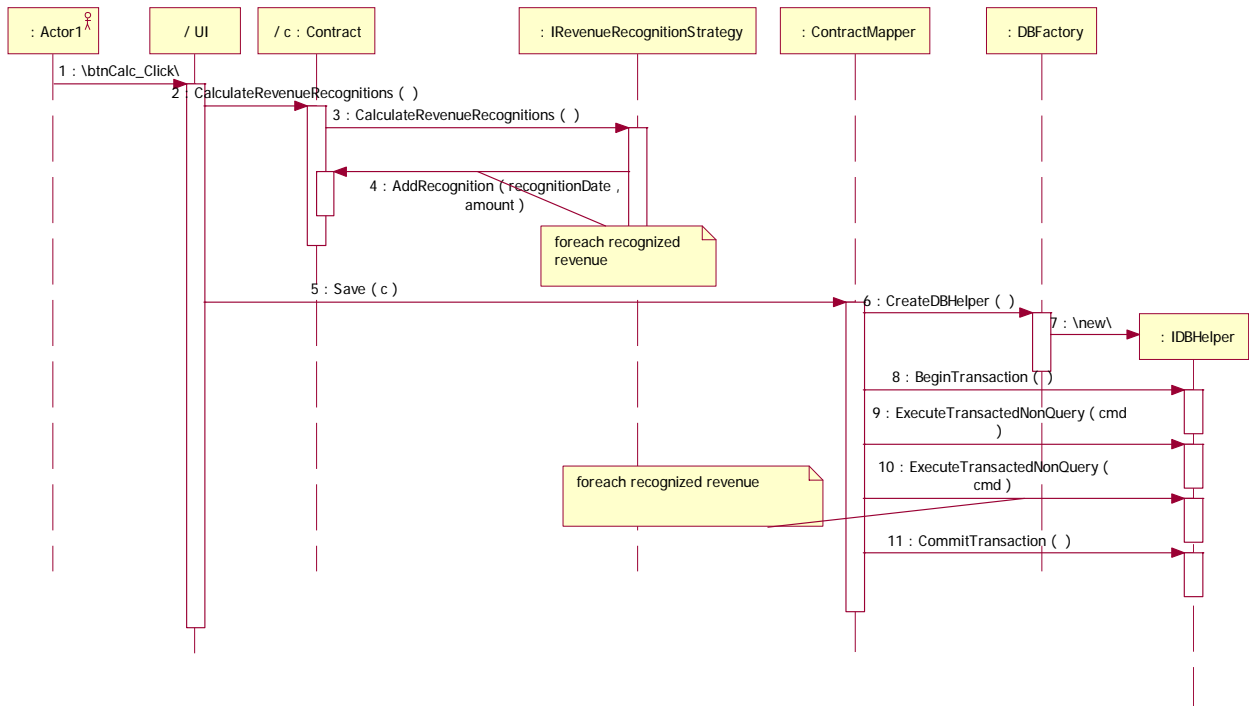


Figure 8 - DM + DM: sequence CalculateRevenues

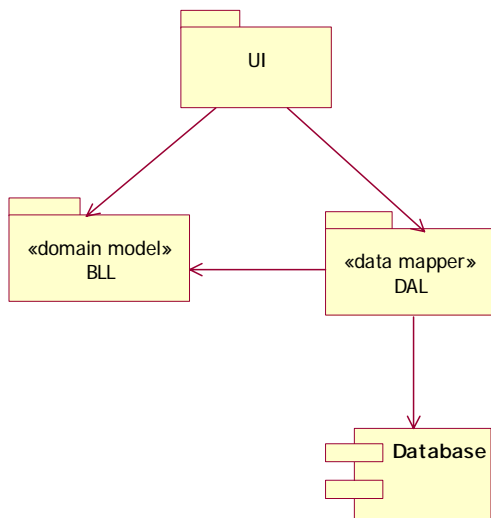


Figure 9 - DM + DM: packages

In this implementation we decided to apply the *Strategy* pattern [5] to separate the rules for calculating the revenue recognition from the *Contract* class, allowing to dynamically attach the correct strategy object at time of creation. This is accomplished by providing three different implementations (one for each product type) of the following interface:

```
public interface IRevenueRecognitionStrategy
{
    void CalculateRevenueRecognitions();
}
```

The data access layer provides the following data mapper classes:

```
public class ContractMapper
{
    protected Contract ApplyMap(
        DataSet dsContractAndRecognitions)

    public Contract LoadById(int contractID)

    public IList LoadAll()

    public void Save(Contract c)
}

public class CustomerMapper : IdentityMap
{
    protected Customer ApplyMap(DataRow row)

    public Customer LoadById(int customerID)

    public IList LoadAll()
}

public class ProductMapper : IdentityMap
{
    protected Product ApplyMap(DataRow row)

    public Product LoadById(int productID)

    public IList LoadAll()
}
```

The mappers' protected method `ApplyMap` constructs a business object given a `DataRow` read from the database. The `save` method perform the mapping between the business object and ADO.net.

The use of a domain model brings additional problems such as how to guarantee that no duplicate objects are read into memory (for instance, if we load all the

contracts of a customer, only one customer object will exist in memory). In order to solve this problem we can apply the *Identity Map* pattern: 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 [4].

In this scenario, both `CustomerMapper` and `ProductMapper` are realizations of the *Identity Map* and *Data Mapper* patterns. One thing to note about patterns is that there are similar patterns and that typically patterns are used in conjunction with each other and not alone [6]. This in deed is a reason why code generators for patterns are almost useless [6].

Figure 7 and Figure 8 shows the sequence of operations for *Get Contracts* and *Calculate Revenue Recognitions* using a domain model and data mappers.

4 Summary

This paper presented several pattern based implementations of the Revenue Recognition problem in order provide a better understanding of the different architectural styles available for layered enterprise applications.

As a summary, you can use the following rules of thumb:

- If the complexity of the problem is moderate and you have a good understanding of relational model and a good support for Record sets in your development environment – choose *Table Module* and *Table Data Gateway*
- If the complexity of the problem is moderate to high but are at ease with OO concepts and your database schema is similar to your OO model, choose *Domain Model* with *Active Record*
- If the complexity of the problem is moderate to high, there is dissonance between your relational model and your OO model or you need independence from each other, choose *Domain Model* and *Data Mapper*.

Using a domain model you also have to guarantee that loading an object won't bring into memory all related objects. For instance, when loading all the orders for a month we probably must avoid loading all the referred products. This can be solved using the *Lazy Load* pattern: an object that doesn't contain all of the data you need but knows how to get it [4].

This project has been used in a university level course on design patterns to help the students understand the differences between each enterprise architecture pattern. My experience shows that the students have some difficulties when presented only with the general description of the pattern. From an informal survey conducted with my students, the majority of the students have acknowledge that the workbench allowed them to understand the patterns and their differences.

Acknowledgements

The author would like to acknowledge FCT, FEDER, POCTI, POSI, POCI and POSC for their support to R&D Projects and GECAD Unit.

5 References

- [1] Alexander, C. (1977) *A Pattern Language : Towns, Buildings, Construction*. Oxford University Press.
- [2] Alexander, C. (1979) *The Timeless Way of Building*. Oxford University Press.
- [3] Crocker, A., Olsen, A. and Jezierski E. (2002) "Designing Data Tier Components and Passing Data Through Tiers". <http://msdn.microsoft.com/library/en-us/dnnda/html/BOAGag.asp>
- [4] Fowler, M. (2002), *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- [5] Gamma, E., Helm, R., Johnson, R. and Vissides, J. (1995) *Design patterns : elements of reusable object-oriented software*. Addison-Wesley
- [6] Holub, A. (2004) *Holub on patterns: learning design patterns by looking at code*. Apress.