Learning patterns of application architecture by looking at code

PAULO SOUSA
GECAD – Knowledge Engineering and Decision Support Group
Instituto Superior de Engenharia do Porto
R. Dr. António Bernardino de Almeida, 431
4200-072 Porto
PORTUGAL

Abstract: - Typical approaches to design patterns present them in the catalog form. For undergraduate students this abstract description of the pattern makes it even more difficult to understand its purpose and need. In this paper we describe a tool used at a university course to enable the students to learn architectural patterns by looking at code. The tool presented in this paper shows the solution to a specific problem using different implementations, each following a different architectural style. The results of informal surveys from users of the tool are presented.

Key words: Design patterns, application architecture

1. Introduction

The author teaches a course on software architecture and patterns and faced a recurring problem when presenting this subject: the difficulty of the students to understand the use of patterns in “real” problems.
The concept of patterns appeared 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 solution 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 (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].
Typical approaches to patterns are based on pattern catalogs which are somewhat abstract in the sense that they lack a coding context. Alan Holub [6] also notes this problem and presents the *Gang of Four* [5] patterns “by looking at code”. In what relates to enterprise application architecture patterns, Martin fowler’s book [4] collects and discusses the most common patterns in use. However, from the author’s experience it is sometimes difficult for 4th year students to understand the connection between the different patterns cited in the book and how they relate to each other to build complete applications.
In this paper we present a tool, the *PoEAA Workbench* which presents several implementations of the same problem (revenue recognition), each following a different architectural style and patterns.

2. The Workbench

2.1 General description

The workbench uses the example problem of *Revenue Recognition* described in [4]. The main goal is to calculate future revenues (amount and date of occurrence) of sales contracts of three kinds (each product with different payment conditions):

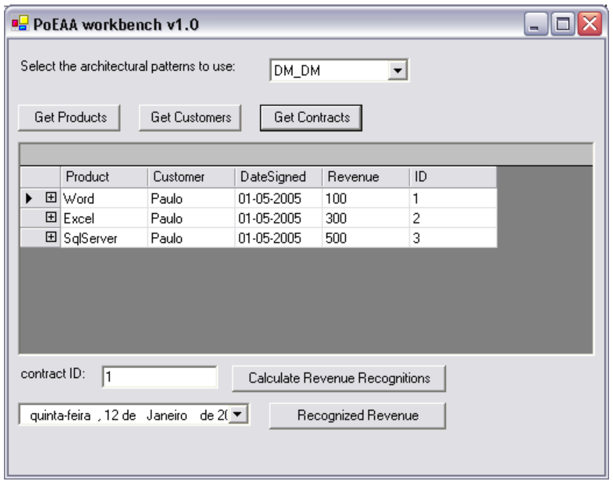
- *Word Processors* – paid in full at acquisition
- *Databases* – three “equal” payments at acquisition, 30 days and 60 days after
- *Spreadsheets* – three “equal” payments at acquisition, 30 days and 90 days after

The business interface defined for the problem is:

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

    object GetContracts();
    object GetCustomers();
    object GetProducts();
}
```

The workbench application (Figure 1) was developed using Microsoft Visual Studio and the .Net framework (downloadable from http://w2ks.dei.isep.ipp.pt/psousa/GetFile.aspx?file=PoEAAWorkbench.zip).



PoEAA workbench v1.0

Select the architectural patterns to use: DM_DM

Get Products | Get Customers | Get Contracts

	Product	Customer	DateSigned	Revenue	ID
▶ ⊞	Word	Paulo	01-05-2005	100	1
⊞	Excel	Paulo	01-05-2005	300	2
⊞	SqlServer	Paulo	01-05-2005	500	3

contract ID: 1 | Calculate Revenue Recognitions

quinta-feira , 12 de Janeiro de 2(| Recognized Revenue

Figure 1 - workbench application's GUI

Figure 2 shows the package structure of the application.

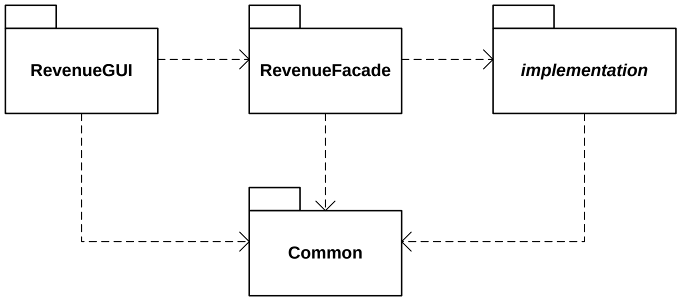


Figure 2 - Workbench package structure

There is a separate layer 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 "implementation" package represents the several packages with specific implementations of an architectural style (transaction script, table module, domain model).

2.2. The Transaction Script layers

A *Transaction Script* [4] organizes all logic as a single procedure, making calls directly to the database or through a thin database wrapper such as a *Row Data Gateway* [4] (an object that mimics the structure of a database record and provides methods for saving and loading this data). A Transaction Script is a very simple approach to decomposition of functionality; in the simplest case, it's just a collection of procedures callable by the presentation and makes no separation of business logic from data access logic.

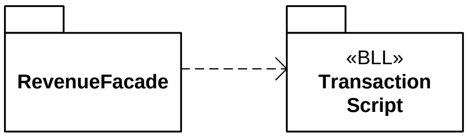


Figure 3 - Transaction script

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

    public void CalculateRevenueRecognitions(
                      int contractID)

    public DataSet GetContracts()
    public DataTable GetProducts()
    public DataTable GetCustomers()
}
```

A more sophisticated version can use a thin database wrapper in a separate package (Figure 4) or even a Row Data Gateway [4].

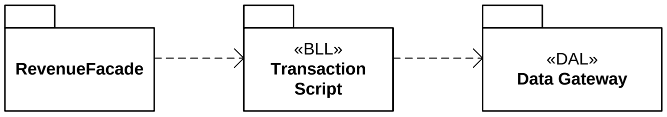


Figure 4 - Transaction script + data gateway

The package for the Row Data Gateway scenario exposes one Row Data Gateway and one Finder class for each table in the database. Following is an example of the gateway and finder for the contract table:

```
public class ContractFinder
{
    private Contract CreateContractObject(DataRow r)

    public Contract GetContractByID(int contractID)
    public IList GetContracts()
}

public class ContractGateway
{
    public int CustomerID;
    public DateTime DateSigned;
    public int ID;
    public int ProductID;
    public decimal Revenue;

    public int Insert()
    public bool Update()
    public bool Delete()
}
```

The main difference of these two approaches reside in the data access API: the data gateway has procedures to insert, update, and fetch database records, while the Row Data Gateway is an object that handles *one* single record of the database but makes no use of OO techniques.

2.3. The Table Module layers

In this architectural style 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. This style organizes the structure of the program according to the table or views of the database. A *Table Module* [4] 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* [4] 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. They are particularly useful in conjunction with Record Set [4] (e.g., ADO.net's DataSet or JDBC's ResultSet) and are a good opportunity for code generation techniques.
Figure 5 shows the package structure for the "table module + table data gateway with record set" implementation.

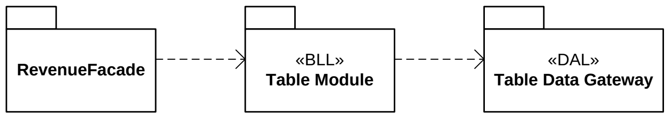


Figure 5 – TM + TDG with record set

There is one class for each table in the database: contract, product and customer. The classes of the business logic layer expose the following API:

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

    public void CalculateRevenueRecognitions(
                        int contractID)

    public DataSet GetContracts()
}
```

```
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 GetContracts()
}


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 6 shows the package structure for the table module variant with custom classes. The main difference between this style and the previous one is 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` and the methods in the data access classes were modified to receive the custom classes as parameters.

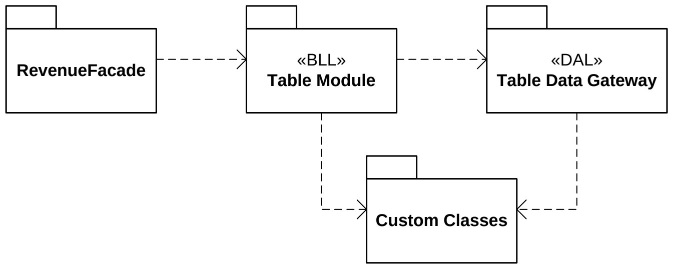


Figure 6 - TM + TDG with custom classes

The data access layer classes have additional 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.

2.4. The Domain Model layers

This style organizes the structure of the program following an object oriented approach of the domain of the problem. A *Domain Model* [4]) is an object model of the domain that incorporates both behavior and data. If this object encapsulates a row in a database and its access along with the business logic methods it is called an *Active Record* [4]). 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 behavior.
In the domain model / active record scenario, both the business logic and the data access logic are placed in the same classes (Figure 7).

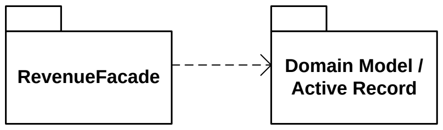


Figure 7 - Domain Model / Active Record

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 itself from and save itself to the database).

```
public class Contract : ActiveRecord
{
    private int _CustomerID;
    private DateTime _DateSigned;
    private int _ProductID;
    private decimal _Revenue;
    private IList _RevenueRecognitions
    private Customer _Customer;
    private Product _Product;

    // PROTECTED CONSTRUCTOR/MAPPERS
    protected Contract(DataRow row)

    // PUBLIC BUSINESS API
    public Contract()
    public Money RecognizedRevenue(DateTime asOf)
    public void CalculateRecognitions()

    // PUBLIC DATA ACCESS API
    public static Contract LoadById(int contractID)
    public static IList LoadAll()
    public override void Save()
}

public class Customer : ActiveRecord
{
    private string _Name;

    // PROTECTED CONSTRUCTOR/MAPPER
    protected Customer(DataRow row)

    // PUBLIC BUSINESS API
    public Customer()

    // PUBLIC DATA ACCESS API
    public static Customer LoadById(int customerID)
    public static IList LoadAll()
    public override void Save()
}

public class Product : ActiveRecord
{
    private string _name;
    private string _type;

    // PROTECTED CONSTRUCTOR/MAPPER
    protected Product(DataRow row)

    // PUBLIC BUSINESS API
    public Product()

    // PUBLIC DATA ACCESS API
    public static Product LoadById(int productID)
    public static IList LoadAll()
    public override void Save()
}
```

A Contract object is now in charge of representing a 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. 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 the class's attributes and the table fields are similar). However, in some situations one wants to be independent of the database schema; thus, a data mapper can be used to hide the persistence details from the business logic (Figure 8).

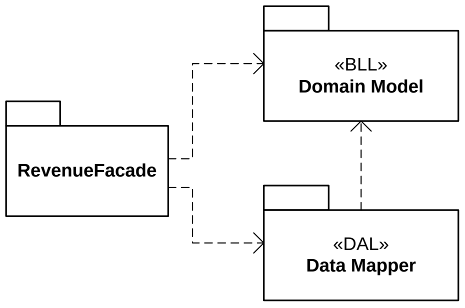


Figure 8 - Domain Model + Data Mapper

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
{
    protected Customer ApplyMap(DataRow row)

    public Customer LoadById(int customerID)
    public IList LoadAll()
}

public class ProductMapper
{
    protected Product ApplyMap(DataRow row)

    public Product LoadById(int productID)
    public IList LoadAll()
}
```

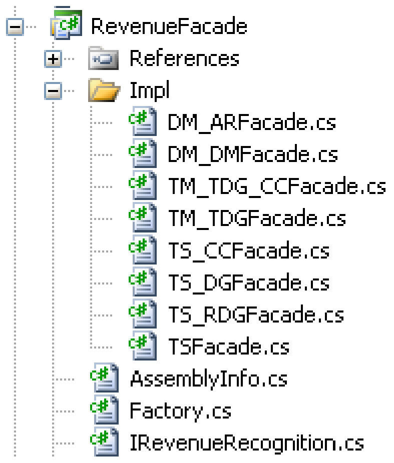
The mappers' protected method `ApplyMap` constructs a business object given a `DataRow` read from the database. The `Save` method perform the mapping between the business object and ADO.net.

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.

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]. Contract objects use the lazy load pattern regarding its `Customer` and `Product` attribute.

2.5. The RevenueFacade layer

The `RevenueFacade` layer defines a factory class which uses reflection to dynamically create an object with the desired implementation. This object is a realization of the *Façade* pattern [5] hiding the complexity of calling the real business layer and data access layer objects. There is a sub-package with a façade for each architectural style (Figure 9)



```
RevenueFacade
  References
  Impl
    DM_ARFacade.cs
    DM_DMFacade.cs
    TM_TDG_CCFacade.cs
    TM_TDGFacade.cs
    TS_CCFacade.cs
    TS_DGFacade.cs
    TS_RDGFacade.cs
    TSFacade.cs
  AssemblyInfo.cs
  Factory.cs
  IRevenueRecognition.cs
```

Figure 9 - RevenueFacade classes

Since the workbench application provides implementations of the various architectural styles and the classes and operations to call for each case are different, the `RevenueFacade` layer hides these details.

The façade to the *Table Module* with *Table Data Gateway* using *Record Set* implements the "Calculate Revenue" business operation in the following way:

```
using TM.BLL;

public void CalculateRevenueRecognitions(
                          int contractID)
{
    Contract bll = new Contract();
    bll.CalculateRevenueRecognitions(
                          contractID);
}
```

The façade for *Domain Model* with *Active Record* does it by calling operations in the Contract object:

```
using DM_AR;


public void CalculateRevenueRecognitions(
                          int contractID)
{
    Contract c = Contract.LoadById(contractID);
    if (c != null)
    {
      c.CalculateRevenueRecognitions();
      c.Save();
    }
}
```

While the façade to the *Domain Model* with *Data Mapper* implements it in the following way:

```
using DM.BLL;
using DM.DAL;


public void CalculateRevenueRecognitions(
                           int contractID)
{
    ContractMapper mapper = new ContractMapper();
    Contract c = mapper.LoadById(contractID);
    if (c != null)
    {
     c.CalculateRevenueRecognitions();
     mapper.Save(c);
    }
}
```

This package is useful for showing the interaction of the business logic and data access logic layers with the calling layer (typically the presentation layer). The transaction script and table module are the most straight forward implementations since the caller only needs to know about the business logic layer (it's the BLL class that invokes the data access logic that it needs). Both Active Record and Data Mapper require the caller to know about the business and data access logic layer; the main difference is if there is only one object for both function sets (such as in Active Record) or two different objects (as in domain model and data mapper).

3. Conclusions and future work

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*.

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 indeed is a reason why code generators for patterns are almost useless [6].

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. In the years before the introduction of this tool, the majority of the students had some difficulties when presented only with the general description of the pattern (as from the pattern catalog). From an informal survey conducted with the students, the majority of the students have acknowledged that the workbench allowed them to understand the patterns and their differences as well as their use in a coherent and related way. The post-graduate students were all full time employed in the area of software development. Even this group had little or no previous contact with the concept of design patterns; the tool helped them to understand the patterns and to use then in the day-to-day application development of their own.
Future work for this project includes the enhancement of the workbench with more architectural styles as well as other GoF patterns and patterns for transaction management and data access concurrency. This tool will also be extended to show the use of ORM patterns and tools (e.g., Hibernate [7]) and code generators for the data access layer.

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

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/dnbda/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.

[7] www.hibernate.org