



## CHAPTER

# 6

# Application Layer



## 1.- APPLICATION LAYER

Following the trend of DDD architecture style, the Application layer should be a Layer that coordinates the activities of the Application as such, but it is essential that it does not include any domain logic or business/domain state. However, it can contain progress states of the application tasks.

The SERVICES that typically reside within this layer (remember that the SERVICE pattern is applicable to different Architecture layers), are services that usually coordinate the SERVICES and objects of other lower level layers.

The most common case of an Application Service is a Service that coordinates all the “plumbing” of the application, that is, orchestration of calls to the Domain Services and later, calls to Repositories to perform persistence, using UoW, transactions, etc.

Another more collateral case would be a SERVICE of the APPLICATION layer responsible for receiving E-Commerce purchase orders in a specific XML format. In this scenario, the APPLICATION layer will be responsible for reformatting/rebuilding such Purchase Orders from the original XML received and converting them into Domain Model ENTITY objects. This example is a typical case of APPLICATION logic. We need to perform a format conversion, which is a requirement of the application and not something that forms part of the Domain logic, so it should not be located in the Domain layer but rather in the Application layer.

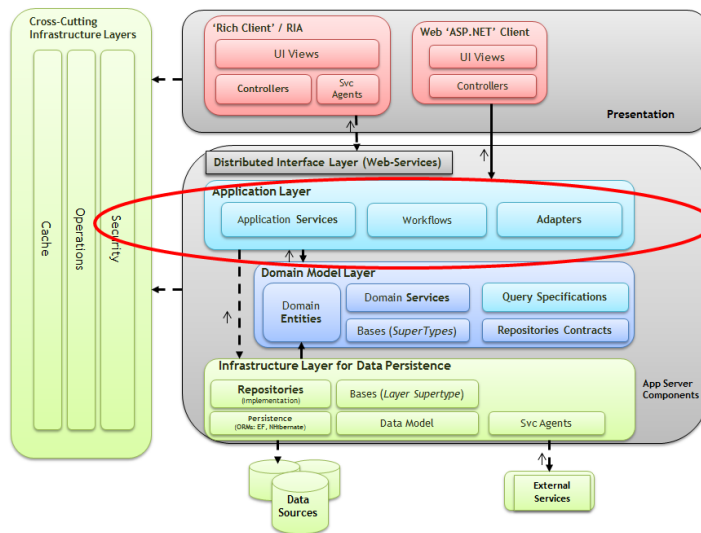
**We will locate all the coordination necessary to perform complete operations/scenarios in the Application layer SERVICES. In other words, we put here the kind of the actions we would never discuss with a Domain-Expert. Those actions are informally known as application "plumbing" coordination. The Application layer is also useful as a *Façade Layer* exposing the server components to consumer layers or external applications (Presentation layers or other remote services).**



## 2.- LOGICAL DESIGN AND ARCHITECTURE OF THE APPLICATION LAYER

In the following diagram we show how the Application layer typically fits into our *N-Layered Domain Oriented* architecture:

**DDD N-Layered Architecture**



**Figure I.- Location of the Application layer in DDD N-Layered Architecture**

The Application Layer defines the tasks that the software is supposed to perform. These are usually related to making calls to the Domain and Infrastructure Layers. Again, we mean the application coordination, not the Domain rules coordination. Good examples are coordination when calling to Repositories to persist data in databases, data conversion, implementation of DTO Adapters to perform data conversions from/to Domain Entities, etc.

Therefore, the elements to be included in the Application Layer might be:

- **Application Services** (the most common element in this layer.)
- **Adapters/Converters** (e.g., Converters from DTO to Domain entities and vice versa)
- **Workflows** (for long executions processes.)

Table I.- Framework Architecture Guide



Rule N°: D17.

**An Application Layer will be designed and implemented for coordination of tasks related to coordinating technical Application requirements.**

### Rules

- The Application logic should not include any Domain logic, but only coordination of the tasks related to technical requirements of the application, such as coordination of calls to Repositories in order to persist data, Domain Entities data format conversion, and ultimately, calls to the Infrastructure components so that they perform complimentary tasks to the application.
- It should not have states reflecting the business process status; however, it may have states that reflect the progress of the application task.



### Advantages of using the Application Layer

- We satisfy the “*Separation of Concerns*” principle, that is, we isolate the Domain layer from tasks/requirements of the application itself also known as “plumbing tasks”. These are actually not business logic, but rather technological integration aspects, data formats, performance optimization, data persistence coordination, etc.



### References

*‘Application’ Layer. By Eric Evans in his book DDD.*



## **2.1.- Application Layer Design Process**

When designing the server component layers, we should consider the requirements of the application design. In this section we briefly explain the main activities associated with the design of components usually located in the Application Layer. When designing the business layers, the following key activities should be performed on each area:

### **1.- Creating a general application layer design:**

- a. Identify consumers of the application layer.
- b. Determine the security requirements of the application layer.
- c. Determine requirements and data validation strategies in the application layer.
- d. Determine the Cache strategy.
- e. Determine the exception management system of the application layer.

## 2.- Design of application logic components (Coordination):

- a. Identify domain components that will be coordinated from the application layer.
- b. Make decisions on localization, coupling and interactions of the business components.
- c. Choose an adequate transactional support.
- d. Identify how to implement the coordination of business rules.
  - a. Directly in the code, using Application services.
  - b. Workflows (scenarios with long executions).
- e. Identify Application layer design patterns that meet the requirements.



## 2.2.- The importance of decoupling the Application Layer from the Infrastructure

In the Application layer design, and generally in the design of the rest of the internal layers, we should ensure that we implement a decoupling mechanism between the objects of these layers. This will make it easier to support a high level of business rules in business application scenarios with a large amount of domain logic components (business rules) and a high level of data source access. This is usually accomplished by **decoupling component layers through contracts or interfaces, and even beyond that, by using DI (*Dependency Injection*) and IoC (*Inversion of Control*). These techniques provide excellent maintainability.**

For more information on core concepts of Inversion of Control and Dependency Injection, see the initial chapter about global N-Layer Architecture.



### 3.- APPLICATION LAYER COMPONENTS

The Application layer may include different types of components. However, the main component is the Application SERVICE, as discussed below.



#### 3.1.- Application Services

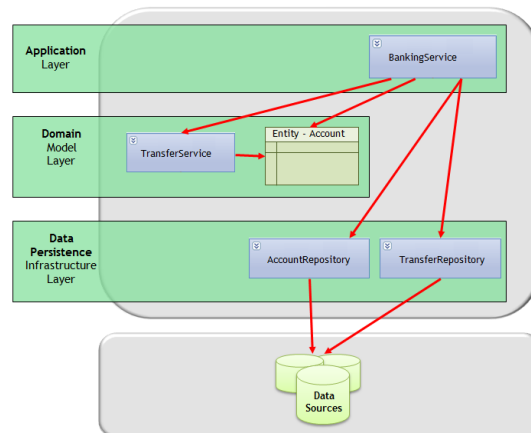
The Application SERVICE is another type of Service, complying with the guidelines of its pattern, the “*SERVICE pattern*”. Basically, they should be stateless objects that coordinate certain operations. In this case they would be operations and tasks related to the Application layer (Tasks required by the application, not by the Domain logic.).

Another function of the Application Services is to encapsulate and isolate the data persistence infrastructure layer. Therefore, in the application layer, we perform **coordination** of **transactions** and **data persistence** (only coordination or calls to Repositories), **data validation** and **security issues** such as authentication requirements and authorization for the execution of specific components, etc.

In addition, the SERVICES should be the only entry point in the architecture through which higher layers access the data persistence infrastructure classes (*Repositories*) when working with data. For example, it is not advisable to directly invoke Repository objects from the Presentation layer.

As we can see in the following diagram, the interaction between the different objects of the layers will usually start in an Application Service, which will be the concentrator or hub of the different types of application actions.

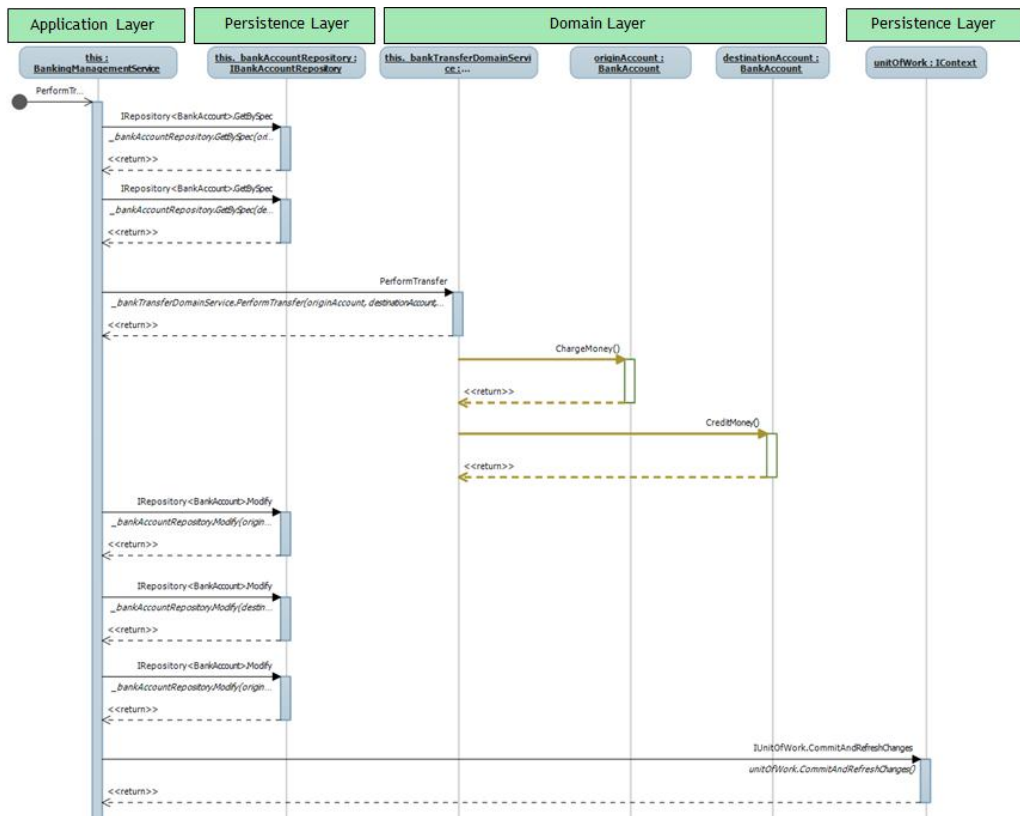
*Application Services interacting with Domain Services and Infrastructure Repositories*



**Figure2.- Interaction Sample between the objects of different layers**

The fundamental aspect of this layer is not to mix Software requirements (persistence coordination, conversions to different data formats, optimizations, Service Quality, etc.) with the Domain layer, which should only contain pure Business logic.

The following UML sequence diagram shows an Application layer object (the Service that originates the bank transfer in the application), the Domain objects and how the Repository and UoW objects are invoked from the Application Service at the end.



**Figure 3.- Sequence Diagram**

In this interaction between objects, only the calls to *ChargeMoney()* and *CreditMoney()* methods are purely business/Domain calls. The rest of the interactions are necessary for the application (e.g., query the data of every account and data persistence through Repositories, use of UoW, etc.) and are therefore coordinated from the Application Layer.



Rule N°: D18.

**Application SERVICE classes are mainly responsible for the intercommunication with the Data Persistence Infrastructure Layer Classes (Repository classes) and other infrastructure objects.**

### ○ Recommendation

- It is recommended for the application SERVICE classes to have the coordination and the communication (interlocutors or access paths) with the Repository classes (lower layer of the Infrastructure) as one of their main responsibilities. For example, there should be no access to a Repository class directly from Web Services or the Presentation layer. Usually, we do not instantiate Repositories from a domain class either, although there may be exceptions, like when we need to query some data from a Domain Service depending on variable states.

**Note1:** Coordination of Repositories, UoW, transactions, etc. could be implemented from the Domain objects/services themselves. In fact, there are many implementations of N-Layer architectures, including those which are DDD, that do so. Placing the Repository coordination in one layer or another is simply for reasons of design preferences. However, by leaving these aspects in the Application layer (as we prefer to do), the Domain layer becomes much cleaner and more simplified, containing only the domain logic.

**Note2:** Additionally, and although as a rule Repositories are only used from the Application layer, it is also possible to make exceptions, and make queries invoking Repositories from the Domain services as necessary. But this should be avoided as much as possible in order to achieve homogeneity in our developments.



**Rule N°: D19.**

**Do not implement persistence/data access code in the Application Services**

○ **Rule**

- Never implement any data access or persistence code (such as code directly using 'LINQ to Entities', 'LINQ to SQL', ADO.NET classes, etc.) or SQL statements code or stored procedures names, directly in a method of an Application layer class. For data access, you should only invoke classes and methods from the Infrastructure layer (i.e., Repository classes).



**References**

“Separation of Concerns” principle

[http://en.wikipedia.org/wiki/Separation\\_of\\_concerns](http://en.wikipedia.org/wiki/Separation_of_concerns)



**Rule N°: D20.**

**Implement the *Layer Supertype* pattern**

○ **Recommendation**

- It is common and very useful to have “base classes” for each layer to group and reuse common functionalities which otherwise would be duplicated in different parts of the system. This simple pattern is called *Layer SuperType*.
- However, it should only be implemented if necessary (YAGNI).



**References**

*'Layer Supertype' pattern.* By Martin Fowler.

<http://martinfowler.com/eaCatalog/layerSupertype.html>





### 3.2.- Decoupling between APPLICATION SERVICES and REPOSITORIES

When decoupling all the objects with dependencies through **IoC and DI**, the **application layer should also be decoupled from the lower layers, such as Repositories (belonging to the Data persistence infrastructure layer)**. This would allow us, for example, to dynamically configure (or when compiling and testing) whether or not we actually want to access the real data repositories (usually relying on databases), a secondary system of repositories/storages or even “fake repositories” (stub or fake). Thus, if we simply want to run a large number of unit tests right after making changes to the business logic, this can be done easily and quickly (without slowing down development) because we will not be accessing databases (tests are run only against the mock or stub repositories).

An application **SERVICE** class method usually invokes other objects (domain Services or data persistence Repositories), setting up rules or complete transactions. **As we saw in the example, an Application Service Method implementing a Bank transfer**, called *BankingManagementService.PerformTransfer()*, made a call to the Domain to carry out the business operation related to the transfer (internally within the Domain entities logic by *Credit()* and *Debit()* methods), and later on, following the Application Service execution, it called the Repository persistence methods so that the transfer was recorded/reflected in the persistent storage (most likely a database). **All of these calls between different objects of different layers (especially those regarding the infrastructure) should be decoupled calls made through interfaces and usually based on dependency injection.** The only case in which it is pointless to decouple through DI is when we use Domain entities, as there is no point in replacing entities to another version that complies with the same interface. Also, POCO Domain entities have no dependencies, not it doesn't make sense to create entity objects using IoC containers.



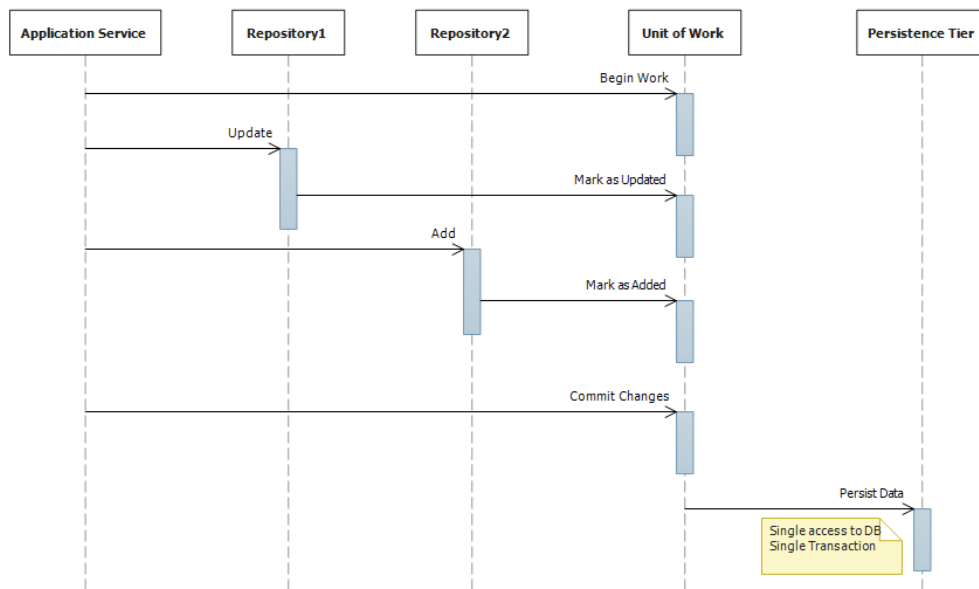
### 3.2.1.- Consuming a UNIT OF WORK from Application Layer Services

One of the most common design patterns in enterprise software development is the Unit of Work. According to Martin Fowler, the Unit of Work pattern "*maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.*"

We previously defined and explained how to define your Unit of Work in the persistence infrastructure layer, but now, in the **Application Services methods** is where we really **consume/use the Unit of Work**.

One of the best ways to use the Unit of Work pattern is to allow several classes and services to take part in a single logical transaction. The key point here is that you want the disparate classes and services to remain ignorant of each other while being able to enlist in a single transaction. It makes the code more explicit, easier to understand, and simpler to unit test.

The following image shows how possible interactions using a UoW and two Repositories from the Application Layer.





### 3.2.2.- DTO (Data Transfer Object)

Because a DTO is part of the required ‘plumbing’ we might require for decoupling the Domain Model from consumer applications (Presentation Layer or external applications) and we might want to use it through different channels (directly or through Distributed Services), a good place to define and place DTOs is within the Application Layer.

Like mentioned, to **decouple** clients/consumers from the Domain Model, and ultimately **from the Domain Entities**, the most common option is to use DTOs (*Data Transfer Objects*). This is a design pattern that consists in packaging multiple data structures in a single data structure to be transferred through different *boundaries* (in most cases ‘*Physical Boundaries*’ using remote communication between servers and/or remote machines). DTOs are especially useful when the application that uses our services has a data representation or even a model that does not exactly match our internal Domain Entity Model (Which happens most of the times). This pattern allows to the development team to evolve the Domain entities with no impact to external consumers, as long as the interfaces/contracts and DTOs structure do not change. So, in many cases, changes in the server might not affect the consumer application. It also supports a more comfortable version management towards external consumers. This design approach is, therefore, the most suitable when there are external clients/consumers using data from our sub-system and also when the development team does not have control over the development of the client applications (Consumer client applications could be developed by other teams with a different development speed).

## Using DTOs

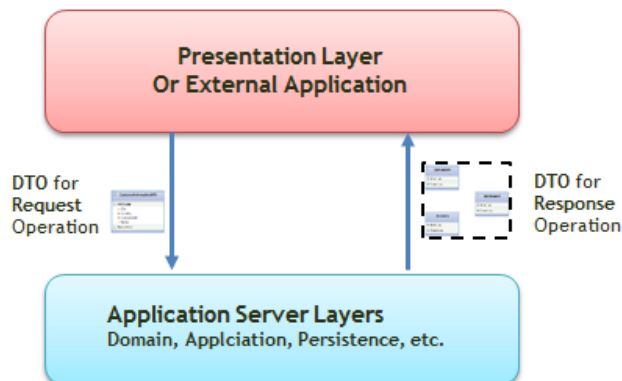


Figure 5.- Using DTOs

The typical design of DTOs tries to adapt to the hypothetical needs of the consumer (either presentation layer, or another type of external application). It is also important to design them so that they minimize the number of calls to the web service (minimize round-trips), therefore improving the performance of the application.



### 3.2.3.- DTO Adapters

Working with DTOs requires having certain adaptation/conversion logic from DTOs to Domain Entities and vice versa. In the DDD N-layered architecture, these Adapters would be typically placed by us within the Application layer, since this is a requirement of the Application Architecture and not part of the Domain. Placing them within the Web Services would not be the best option either, since this layer should be as thin as possible, and also, we could want to use DTOs from a different consumer who is not using Distributed-Services, like traditional Web-app clients.

## Architecture using DTOs

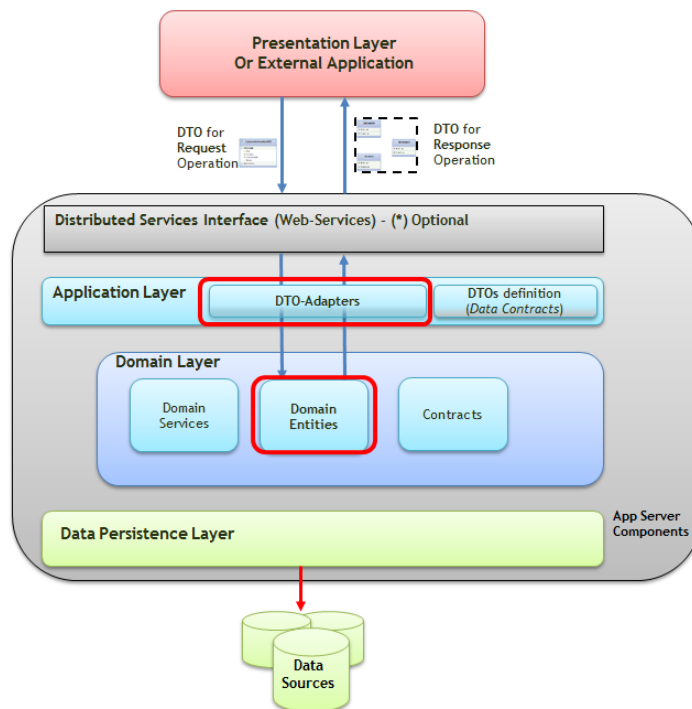


Figure 6.- Architecture Diagram using DTOs (Data Transfer Objects)

The DTOs promote coarse grain operations by accepting DTOs that are designed to carry data usually between different physical levels (Tiers).

This is a good approach from a pure Software Architecture point of view, since we decouple Domain data entities from “outside the domain world”. In the long term, decoupling Domain entities from consumer applications (using DTOs) has greater benefits than propagating Domain Entities to the presentation layer or external consumers. However, the use of DTOs requires significantly more initial work than when using directly serialized domain entities (which in some cases can even perform concurrency handling tasks for us), as we will see in the section about Distributed Services implementation in .NET.

DTO-Adapters need to be developed and the conversion and mapping from DTOs to Domain Entities and vice versa needs also to be programmed. However, there are libraries available that can help us with that mapping work (We’ll see several options in the Application Layer implementation section). Still, if we want to implement a complete optimistic concurrency exception handling, this usually needs to be implemented by ourselves, as well.

There can also be mixed approaches; for example, using serialized domain entities for end-to-end controlled presentation layers, and using DTOs for façades to be consumed by unknown clients.

But, if the application is large and complex enough (like most DDD apps that might even have several autonomous BOUNDED-CONTEXTS), the use of DTOs is highly recommended for its de-coupling capacities regarding data models.



## References about DTOs

*Pros and Cons of Data Transfer Objects (Dino Esposito)*

**<http://msdn.microsoft.com/en-us/magazine/ee236638.aspx>**

*Building N-Tier Apps with EF4 (Danny Simons):*

**<http://msdn.microsoft.com/en-us/magazine/ee335715.aspx>**



### 3.2.4.- Application Layer Workflow Services (optional)

Actually, this sub-layer is a special case of Application SERVICES that provides a solution to certain casuistry in different software solutions. Long running processes or processes where there is interaction both with humans and other software systems are clear examples for the use of workflows. Modeling a process with human interaction directly in the code will often obscure its true purpose and in many cases hinder the possibility of understanding it, thereby reducing its readability. On the other hand, the workflow layer allows for modeling different interactions through activities and a control designer gives a clear visual idea of the purpose of the process to be carried out.



**Rule N°: D21.**

#### **Designing and implementing a Workflow Service Sub-layer in the Application Layer**

##### **Recommendations**

This layer is optional. In fact, it is not commonly found in applications that focus to a great degree on data without business processes with human interactions.

Try to encapsulate the processes in “Activities” in a workflow so they are reusable in other flows.

Although the workflows may implement “business logic”, they should always rely on domain services and repositories in order to perform the different tasks assigned to their activities.



##### **References**

**Workflow Patterns** - <http://www.workflowpatterns.com/>.

When we refer to workflows, we generally talk about the steps of these workflows, typically referred to as activities. When there is an implementation of this sub-layer, it is very important to make the best of its reusability and to pay attention to how they are implemented. The following shows the most important highlights:

- If the activities use persistence mechanisms, they should use the already defined repositories as much as possible.
- The activities may orchestrate different application sub-layers and domain service methods.

Most of the workflow engines that exist today have mechanisms to ensure their durability in long-running processes and/or in the case of a system outage. These systems are usually based on relational databases; therefore, they have different operations that could be incorporated in this sub-layer. Some examples of operations are as follows:

- Rehydration of workflows from the persistence system to the memory.
- Download of workflows from memory to the persistence system.
- Checking the existence of certain workflows in the persistence system.
- Workflow instance correlation storage in the persistence system.



### 3.3.- Application Layer Errors and Anti-patterns

There are certain problematic items and common errors in many applications, which should be analyzed when designing the Application layer. This table lists these items grouped by categories.

**Table 2.- Application Layer Anti-patterns**

Category	Common Mistakes
Authentication	<ul style="list-style-type: none"> <li>- Applying the authentication of the application itself in the application layers when it is not required and when a global authentication outside the application could be used.</li> <li>- Designing an authentication mechanism of its own.</li> <li>- Failure to have a 'Single-Sign-on' when appropriate.</li> </ul>
Authorization	<ul style="list-style-type: none"> <li>- Incorrect use of role granularity.</li> <li>- Use of impersonation and delegation when not required.</li> <li>- Mixing up authorization code with business process code.</li> </ul>
Application components	<ul style="list-style-type: none"> <li>- Mixing up data access logic (TSQL, LINQ, etc.) in the Application Services.</li> <li>- Overload of business components when mixing non-related functionalities.</li> </ul>

	<ul style="list-style-type: none"> <li>- Failure to consider the use of interfaces based on messages (Web-Services) when exposing the business components.</li> </ul>
Cache	<ul style="list-style-type: none"> <li>- Making a volatile data cache.</li> <li>- Caching too much data in the application layers.</li> <li>- Failure to get to cache data in a ready to use format.</li> <li>- Caching sensitive/confidential information in a non-encrypted format.</li> </ul>
Coupling and cohesion	<ul style="list-style-type: none"> <li>- Designing layers that are tightly-coupled with each other.</li> <li>- There is no clear separation of responsibilities (<i>concerns</i>) between the different layers.</li> </ul>
Concurrency and transactions	<ul style="list-style-type: none"> <li>- A correct model of data concurrency was not chosen.</li> <li>- Use of ACID transactions that are too long and causing too many blockages in the database.</li> </ul>
Data access	<ul style="list-style-type: none"> <li>- Access to the database directly from the business/application layers.</li> <li>- Mixing-up data access logic with business logic in the business components.</li> </ul>
Exception management	<ul style="list-style-type: none"> <li>- Showing confidential information to the end user (such as connection strings when there are errors).</li> <li>- Use of exceptions to control application flow.</li> <li>- Failure to show the user error messages with useful information.</li> </ul>
Instrumentalization and <i>Logging</i>	<ul style="list-style-type: none"> <li>- Failure to adapt the instrumentation in business components.</li> <li>- Failure to log critical business events or critical system events.</li> </ul>



Validation	<ul style="list-style-type: none"> <li>- Exclusively relying on validation made in the presentation layer.</li> <li>- Failure to correctly validate length, range, format and type.</li> <li>- Failure to reuse validation logic.</li> </ul>
Workflow	<ul style="list-style-type: none"> <li>- Failure to consider the application management requirements.</li> <li>- Choosing an incorrect workflow pattern.</li> <li>- Failure to consider how to manage all the state exceptions.</li> <li>- Choosing incorrect workflow technology.</li> </ul>



### 3.4.- Design aspects related to the Application Layer

The following sections are specifically cross-cutting aspects of an Architecture and are explained in detail in the chapter on ***'Cross-Cutting/Horizontal Infrastructure Layers.'*** However, it is important to show which of these aspects are related to the Application Layer.



### 3.4.1.- Authentication

Designing an effective authentication strategy for the Application Layer is essential when dealing with application security and reliability concerns. If this is not correctly designed and implemented, the application may be vulnerable to attacks. The following guidelines should be followed when defining the type of application authentication to be used:

- Do not perform authentication in the Application layer if you are only going to use it for a presentation layer or at a Distributed Services level (Web Services, etc.) within the same trusted boundary. In these cases (commonly in business applications), the best solution is to propagate the client's identity to the Application and Domain layers when it should be authorized based on the initial client's identity.
- If the Application and Domain layers are used in multiple applications with different user's storages, then the implementation of a "*single sign-on*" system should be considered. Avoid designing your own authentication mechanisms; it is preferable to use a generic platform.
- Consider using a "Claims based" security, especially for applications based on Web Services. This way, the benefits of federated identity can be used, and different types and technologies of authentication may be integrated.

This cross-cutting aspect (Authentication) is further explained in the chapter "*Cross-Cutting/Horizontal Infrastructure Layers*'.



### 3.4.2.- Authorization

Designing an effective authorization strategy for the Application Layer is essential when dealing with security and reliability of the application. If this is not correctly designed and implemented, the application can be vulnerable to attacks. The following guidelines should be considered when defining the type of application authorization to be used:

- Protect the resources of the Application and Domain Layer (services classes, etc.) by applying the authorization to consumers (clients) based on their identity, roles, claims of role type, or other contextual information. If roles are used, try to minimize the number of roles in order to reduce the number of combinations of required permissions.

- Consider use of authorization based on roles for business decisions, authorization based on resources for system audits, and authorization based on claims when the support of federated authorization is necessary and is based on a mix of information such as identity, role, permissions, rights and other factors.
- Avoid using impersonation and delegation insofar as possible because it may significantly affect performance and scalability. Generally, as regards performance, it is more expensive to impersonate a client in a call than to make the call itself.
- Do not mix the authorization code.

This cross-cutting aspect (Authorization) is further explained in the chapter '***Cross-Cutting/Horizontal Infrastructure Layers***'.



### **3.4.3.- Cache**

Designing an effective cache strategy for the application is essential when dealing with the performance and scalability concerns of the application. Cache should be used to optimize master data queries, to avoid unnecessary remote calls through the network and to eliminate duplicated processes and queries. As a part of the strategy there should be a decision on when and how to upload data to the cache. This depends entirely on the nature of the Application and Domain, since it depends on each entity.

To avoid unnecessary client wait time, load the data in an asynchronous manner or use batch processes.

The following guidelines should be considered when defining the cache strategy of the application:

- Make a cache of static data that will be regularly reused in the different layers (in the end, they will be used/handled in the Domain and Presentation Layers), but avoid making a cache of very volatile data. Consider making a cache of data that cannot be quickly and efficiently retrieved from the database (sometimes normalized data in databases can be hard to obtain, we could therefore cache de-normalized data). At the same time, however, avoid making a cache containing very large volumes of data that may slow down the caching process. Make a cache of minimally required volume.
- Avoid cache of confidential data or design a protection mechanism of such data in the cache (such as encryption of such confidential data).
- Consider deployments in “Web Server Farms”, which may affect standard caches in the memory space of the Services. If any server in the Web-Farm can

handle queries of the same client (Balancing without affinity), the cache to be implemented should support data synchronization between different servers of the *Web-Farm*. Microsoft has appropriate technologies for this purpose (Distributed cache) as explained further on in this guide.

This cross-cutting aspect (Cache) is further explained in the chapter '*Cross-cutting/Horizontal Infrastructure Layers*'.



### **3.4.4.- Exception Handling**

Designing an effective strategy of Exception Handling for the application layer can be essential when dealing with stability and even security concerns of the application. If exceptions are not properly managed, the application may be vulnerable to attacks; it may disclose confidential information about the application, etc. Also, originating business exceptions and exception Handling itself are operations with a relatively expensive process cost, so we should consider the impact on performance in our design.

When designing an exception Handling system, the following guidelines should be followed:

- Catch only the exceptions that can actually be managed or if it is necessary to add information.
- Under no circumstance should the exception Handling system be used to control the flow of the application or business logic. The implementation of exceptions catching (Catch, etc.) has very low performance and in these cases (normal execution flow of the application) it would cause a negative impact on application performance.
- Design a proper strategy for exception Handling; for example, allow exceptions to flow up to the “boundary” layers (e.g., the last level of the component server) where they can (should) be persisted in a logging system and/or transformed as necessary before passing to the presentation layer. It is also best to include a context identifier so that the related exceptions may be associated throughout the different layers, making it easier to identify the origin/cause of the errors.

This cross-cutting aspect (Exception Handling) is further explained in the chapter '*Cross-Cutting/Horizontal Infrastructure Layers*'.



### 3.4.5.- Logging, Audit and Instrumentalization

Designing an effective strategy of *Logging, Audit and Instrumentalization* for the Domain and Application layer is important for application security, stability and maintainability. If it is not properly designed and implemented, the application may be vulnerable to rejection actions when certain users deny their actions. The log/record files may be requested to test incorrect actions in legal procedures. The Audit is considered more accurate if the information log is generated at the exact moment of access to the resource and through the routine that accesses the resource.

The instrumentation may be implemented with events and performance counters as well as the subsequent use of monitoring tools to provide administrators with information on the state, performance and health of the application.

Consider the following guidelines:

- Centralize the *logging*, audits and instrumentation in the Application and Domain layers.
- Make use of simple and reusable classes/libraries. For more advanced aspects (clear publication in different repositories and even in SNMP traps), we recommend using libraries such as '*Microsoft Patterns & Practices Enterprise Library*' or those of third parties, such as *Apache Logging Services "log4Net"* or Jarosław Kowalski's "*NLog*".
- Include instrumentation in the system and/or critical business events within the components of the Application Layer and Domain Layer.
- Do not store confidential information in the log files.
- Make sure the failures in the logging system do not affect the normal operation of the Application and Domain layers.



### 3.4.6.- Validations

Designing an effective strategy for validations in the Application and Domain layer is not only important for the stability of the application, but also for the use of the application by the end user. If it is not properly designed, it may lead to data inconsistencies and violations of the business rules, and finally, to a mediocre user

experience due to the errors subsequently originated, which should have been detected earlier. In addition to this, if it is not correctly built, the application may also be vulnerable to security aspects like ‘*Cross-Site-Scripting*’ attacks in web applications, SQL injections attacks, ‘*buffer overflow*’, etc.

Consider the following guidelines:

- Validate all the input data and method parameters in the Application layer, even when data validation has been previously performed in the presentation layer. The data validation in the presentation layer is more closely related to user experience and the validation performed in the Application layer is more associated with aspects of application security.
- Centralize the validation strategy to enable tests and reuse.
- Assume that all the input data of the users may be “malicious”. Validate data length, ranges, formats and types as well as other more advanced concepts of the business/domain rules.



### **3.4.7.- Deployment Aspects of the Application Layer**

When deploying the Application and Domain layer, consider performance aspects and security of the production environment. Consider the following guidelines:

- If you want to maximize performance, consider deploying the application and domain layer in the same physical level as the Web presentation level. It should only be taken to another physical level for security reasons and in some special cases of scalability.



### **3.4.8.- Concurrency and Transactions**

When we design Concurrency and Transactions aspects, it is important to identify the proper model of concurrency and determine how to handle transactions. For concurrency you may choose between the optimistic and the pessimistic model.

#### **Optimistic Concurrency Model**

In this model, the blocks are not kept in the database (only the minimum required while updating, but not while the user is working or simply with the update window open), and therefore updates are required to check that the data have not been changed

in the database since the original retrieval of the data to be modified. Generally, it is articulated based on *timestamps*.

### Pessimistic Concurrency Model

Data to be updated are blocked in the database, and they cannot be updated by other operations until unblocked.

Consider the following guidelines related to concurrency and transactions:

- The transaction boundaries should be considered so that retries and compositions can be made.
- When a commit or rollback cannot be applied, or if long execution transactions are used, choose the option to implement compensatory methods to undo operations performed on data and leave it in the previous state in case an operation is about to fail. The reason for this is that you cannot keep the database blocked due to a long running transaction.
- Avoid maintaining blockage for long periods of time; for example, do not perform long running transactions that are '*Two Phase Commit*'.
- Choose a suitable isolation level for the transaction. This level defines how and when changes will be available for other operations.



### 3.5.- Map of possible patterns to be implemented in the Application layer

In this table you can see the key patterns for the application layers, organized by categories. The use of these patterns should be considered when decisions are made for each category.

**Table 3.- Key patterns**

Categories	Patterns
Application layer components	<ul style="list-style-type: none"><li>• Application Façade</li><li>• Chain of Responsibility</li><li>• Command</li></ul>

Concurrency and transactions	<ul style="list-style-type: none"> <li>• Capture Transaction Details</li> <li>• Coarse-Grained Lock</li> <li>• Implicit Lock</li> <li>• Optimistic Offline Lock</li> <li>• Pessimistic Offline Lock</li> <li>• Transaction Script</li> </ul>
Workflows	<ul style="list-style-type: none"> <li>• Data-driven workflow</li> <li>• Human workflow</li> <li>• Sequential workflow</li> <li>• State-driven workflow</li> </ul>



## Patterns References

Information on ‘Command’, ‘Chain of Responsibility’ and ‘Façade’ patterns or “data & object factory” at <http://www.dofactory.com/Patterns/Patterns.aspx>

Information on “Entity Translator” pattern:

<http://msdn.microsoft.com/en-us/library/cc304800.aspx>

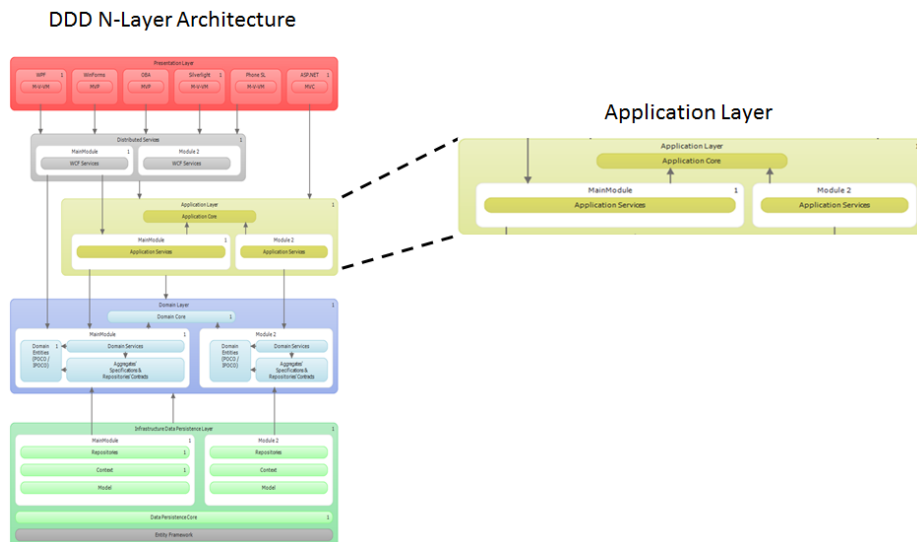
“Capture Transaction Details pattern”, see “Data Patterns” in:

<http://msdn.microsoft.com/en-us/library/ms998446.aspx>





In the following diagram we emphasize the location of the Application Layer with a *Layer Diagram* implemented with Visual Studio 2010:



**Figure 6.- Application Layer Situation diagram**

Steps to be followed:

- 1.- After identifying the application characteristics and the software requirements (not the Domain requirements), we should create a structure for this layer, that is, the project or projects in Visual Studio that will host the .NET classes implementing the Application SERVICES and other Application artifacts (DTO-Adapters, etc.).
- 2.- We will add and implement Application SERVICES as .NET classes. It is important to remember that we should also continue working with abstractions (interfaces) in this layer. Therefore, for each SERVICE class (implementation) we should also have an interface declaring all its operations (operations contract). This interface will be used from the higher layer (Web services or Presentation in ASP.NET) through the Unity container. When resolving dependencies, the UNITY container will resolve every object dependency

related to the Service interface that we order. The process is similar to the one followed in a Domain SERVICE implementation.

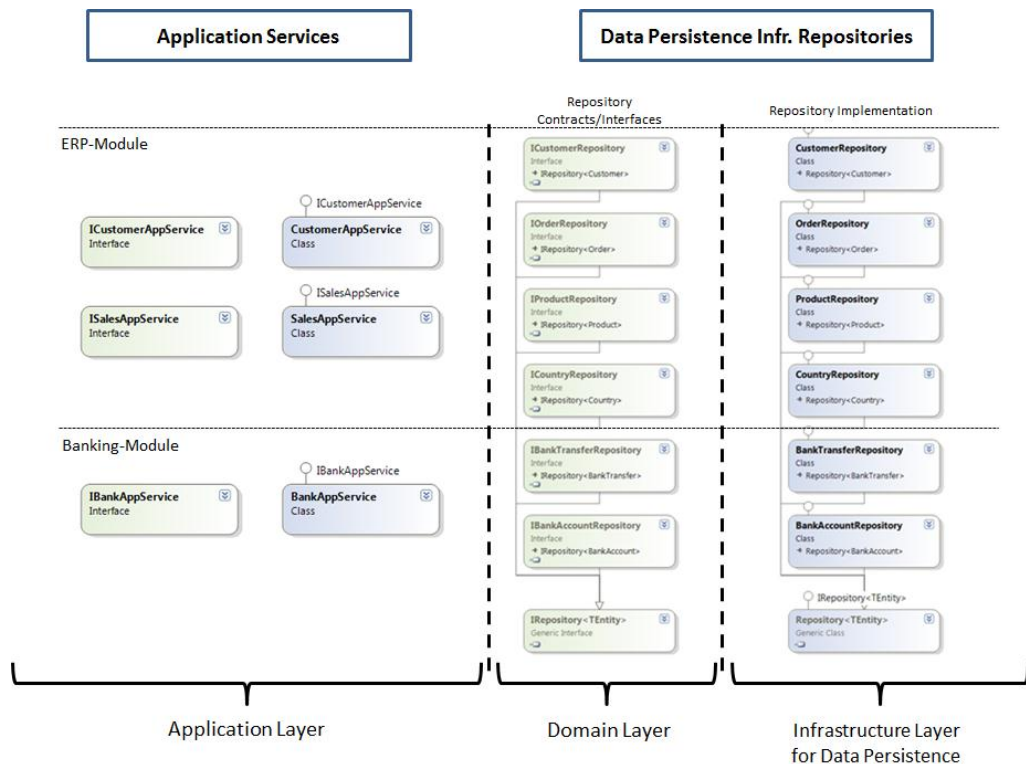
- 3.- SERVICES of the application layer may be implemented with WORKFLOW technologies, and not only through .NET classes, for special cases with a long-time executions and scenarios where a workflow fits better.



## 4.1.- Implementation of Application Layer Services

In general and with a few exceptions, the APPLICATION SERVICES should be the only item or type of component of the architecture through which there is access to the data persistence infrastructure classes (*Repositories*). There should not be direct access to the Repositories from the presentation layers or Web services. Otherwise, we would be bypassing the application coordination logic, as well as the business/domain logic.

The following figure shows several application Services and their related Repository classes (Repositories implementation are part of the Data Persistence Infrastructure layer):



**Figure 7.- Repositories and Application Service Classes Diagram**

In addition to the Application Services and Repository classes, we also have the Domain Services within our application (but they are situated within the Domain Layer). However, these are not shown in the diagram above because the interaction with Repositories (creation and use of Repositories) will generally be done from the Application service Layer.

Below is shown an example of an Application SERVICE class which coordinates all the 'plumbing' (UoW and Repositories) related to the **Customer** entity:

```
C#

Application Service

Interface for abstraction and instantiation through IoC container
(Unity), from higher layers (e.g. WCF or ASP.NET)

public class CustomerAppService
    : ICustomerAppService
{
    //Members
    ITypeAdapter _typesAdapter;
    ICountryRepository countryRepository;
    ICustomerRepository customerRepository;

    Using 'Constructor Injection', with main dependencies instantiated
    and injected automatically by the IoC Container (Unity)

    public CustomerAppService(ITypeAdapter typesAdapter,
        ICountryRepository countryRepository,
        ICustomerRepository customerRepository)
    {
        //Validations Ommitted
        //...
        typesAdapter = typesAdapter;
        countryRepository = countryRepository;
        _customerRepository = customerRepository;
    }

    Application coordination logic for 'Customer' related operations.

    public List<CustomerListDTO> FindCustomers(int pageIndex,
        int pageCount)
    {
        //Ommitted method validations

        //get customers
        var customers = _customerRepository.GetEnabled(pageIndex,
            pageCount);

        Access to Database through an injected
        Repository. (Get enabled customers)

        if (customers != null
            &&
            customers.Any())
        {
            Transform Entity Collection to a DTO Collection.
            Internally it is using AutoMapper.

            return _typesAdapter.Adapt<IEnumerable<Customer>,
                List<CustomerListDTO>>(customers);
        }
        else
        {
            Return the DTO collection to consumers (Higher layers like WCF or ASP.NET)

            return null;
        }
    }
    // ...Other CustomerAppService methods (updates, etc.)
}
```

The code above is quite straightforward, except maybe for one detail (if you are not used to IoC): **Where is the 'ICustomerRepository' Repository being instantiated and created?** (And the rest of dependencies).

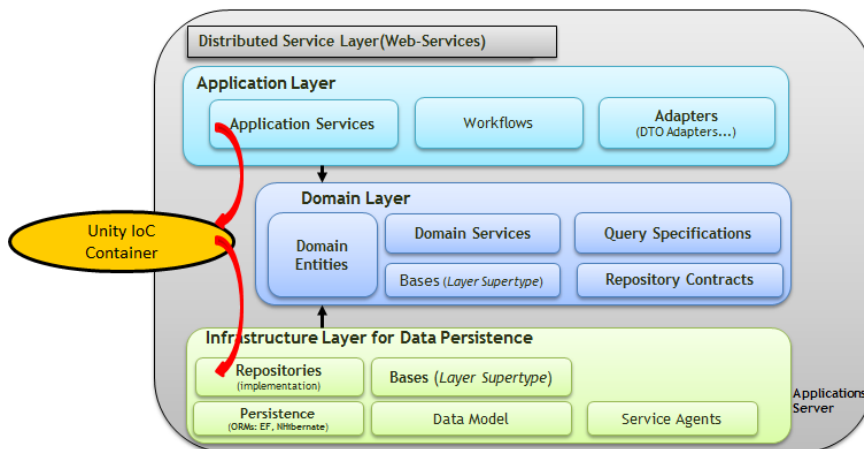
This is precisely related to the Dependency Injection and decoupling between objects through the Unity IoC container which is discussed below.



#### 4.1.1.- Decoupling and using Dependency Injection between Application Services and Repositories through UNITY IoC

The following scheme shows how Dependency Injection is being implemented with UNITY, between the “Application Service” classes and the “Data Persistence Infrastructure” layer Repositories (In reality we used DI for all dependencies, not only for Persistence objects dependencies):

##### *Repository Dependency Injection in Application Services*



**Figure 8.- Domain Service Scheme**

We highlight this decoupled integration between application and Repositories, but it is really done for all dependencies all around within the architecture, starting the objects graph at the very inception of the first call to our application server (WCF or ASP.NET). If you are not familiar with Unity, you should first read the section about “Implementation of Dependency Injection and IoC with UNITY”, which is part of this Architecture and implementation guide (In Chapter 3).

## Interface and Class Registration in the Unity Container

Before instantiating any class through the Unity container, we need to “register” the types in the Unity IoC container, both the interfaces and the classes. **This registration process may be done through code (e.g. C#) or as a statement through the Unity configuration XML.**

In the case of registering class types and mappings using XML, you may choose to mix the Unity configuration XML with the XML of web.config or App.config of the process that hosts your application/service. However, a better (and cleaner) approach would be to have a specific XML file for Unity linked to the app.config/web.config configuration file. In the sample implementation we used a specific configuration file for Unity, called Unity.config.

This would be the XML link from the web/app.config to the Unity configuration file:

```
Web.config (From a WCF Service or an ASP.NET app, etc.)
```

```
...
<!-- Unity configuration for solving dependencies-->
<unity configSource="Unity.config"/>
...
```

This is the XML configuration to register the interface and Repository class:

```
Web.config (WCF Service config, etc.)
...
<!-- Unity configuration for solving dependencies-->
<unity configSource="Unity.config"/>
...
XML - Unity.config
<?xml version="1.0" encoding="utf-8" ?>
<unity>
  <typeAliases>
    ...
    <typeAlias alias="ICustomerRepository" type="
Microsoft.Samples.NLayerApp.Domain.MainBoundedContext.ERPModule.Aggregat
es.CustomerAgg.ICustomerRepository, NLayerApp.Domain.MainBoundedContext"
/>
    ...
    <typeAlias alias="CustomerRepository"
type="Microsoft.Samples.NLayerApp.Infrastructure.Data.MainBoundedContext
.ERPModule.Repositories.CustomerRepository,
NLayerApp.Infrastructure.Data.MainBoundedContext" />
    ...
  
```

```
...
  
```

Below is where the interesting part comes in, that is, the mapping that we can specify in the container between the contracts/interfaces and the class to be instantiated by the Unity container. In other words, a mapping that states: “*When I command that I want to get an ICustomerRepository object , instantiate and give me an object of the*

*CustomerRepository class*". The interesting part is that at a given moment, if we want to run isolated unit tests against a fake implementation (a stub/mock), it could specify something similar to the following statement: "When I command that I want to get an *ICustomerRepository* object , instantiate an object of the *CustomerFakeRepository class*".

So the XML statement in the Unity.config file where we specified this mapping for our sample Repository is the following:

#### XML - Unity.config

```
<?xml version="1.0" encoding="utf-8" ?>
<unity>
<typeAliases>
  ...
</typeAliases>
  ...
  ...
<!-- UNITY CONTAINERS -->
<containers>
<container name="RootContainer">
<types>
  ...
<type type="ICustomerRepository" mapTo="CustomerRepository">
</type>
  ...
</types>
</container>
  ...
  ...
```

**Container:** We can have a containers hierarchy, created by program. Here we simply define the mapping of each container.

Mapping to the class that will be instantiated by the Unity container.

This registry of types and mappings from interfaces to classes may also be done through .NET code (C#, VB, etc.), which is probably the most suitable way when we are in the middle of the project development, because we'll detect typing errors at compilation time instead of at run time. In the sample application we are using C# code for these registration tasks, with a code similar to the following snippet:

```
//Registering Types & mappings in a single step
//-> Unit of Work and repositories
currentContainer.RegisterType<IMainBCUnitOfWork,
    MainBCUnitOfWork>(new PerResolveLifetimeManager());

_currentContainer.RegisterType<IBankAccountRepository,
    BankAccountRepository>();
currentContainer.RegisterType<ICountryRepository, CountryRepository>();
currentContainer.RegisterType<ICustomerRepository,
    CustomerRepository>();
_currentContainer.RegisterType<IOrderRepository, OrderRepository>();
_currentContainer.RegisterType<IProductRepository, ProductRepository>();

//Other Types Registrations...
//...
```

Once we have defined the mappings (either by XML or .NET code), we may implement the code where we really command to the Unity container to instantiate an object for a given interface. We could do something similar to the following code (Please note that we must not explicitly invoke the `Resolve()` method to instantiate `Repositories`, as `Repositories` will usually be resolved based on higher level classes constructors based on the Dependency Injection objects graph).

```
C#
IUnityContainer container = new UnityContainer();
ICustomerRepository customerRep =
    container.Resolve<ICustomerRepository>();
```

**Keep in mind that if we want to apply the DI (Dependency Injection) correctly, we will usually invoke the `Resolve()` method only from the highest level classes of our application server, that is, from the incoming or upper layers that are usually Web services (WCF) and/or ASP.NET presentation layer. We should not do an explicit `Resolve()` against `Repositories`, because we would be using the container almost exclusively as a ‘Type Locator’ (also called *ServiceLocator* pattern). It would not be correct from a DI point of view.**

In short, as we should have a chain of built-in layers decoupled from each other through Unity, the best option is to let Unity detect our dependencies through each class constructor. That is, **if our Application Service class has a dependency to a Repository class (and several other dependencies), we simply specify those types in our constructor and the Unity container will instantiate an object for each dependency, and it will provide them as parameters of our constructor.**

Simplifying our Application SERVICE class called 'CustomerAppService', the code related to its DI by constructor will be like the following code.

```
C#
Application Service
Interface for abstraction and instantiation through IoC container
(Unity), from higher layers (e.g. WCF or ASP.NET)

public class CustomerAppService
    : ICustomerAppService
{
    //Members
    ITypeAdapter _typesAdapter;
    ICountryRepository countryRepository;
    ICustomerRepository customerRepository;

    Using 'Constructor Injection', with main dependencies instantiated
    and injected automatically by the IoC Container (Unity)

    public CustomerAppService(ITypeAdapter typesAdapter,
                             ICountryRepository countryRepository,
                             ICustomerRepository customerRepository)
    {
        //Validations Ommitted
        //...
        _typesAdapter = typesAdapter;
        countryRepository = countryRepository;
        customerRepository = customerRepository;
    }
}
```

It is important to point out that, as shown, we have not made any explicit “new” instantiating an object of the CustomerRepository class. The Unity container is the one that automatically instantiates the CustomerRepository object and provides it as a parameter to our constructor. **This is precisely the dependency injection in the constructor.**

Then, within the constructor, we store the dependency (Repository in this case) in a field inside the object so that we can use it from different methods of our Application Layer Service class.

Finally, the following *code snippet* shows how a graph of objects would be instantiated using dependency injection based on constructors.

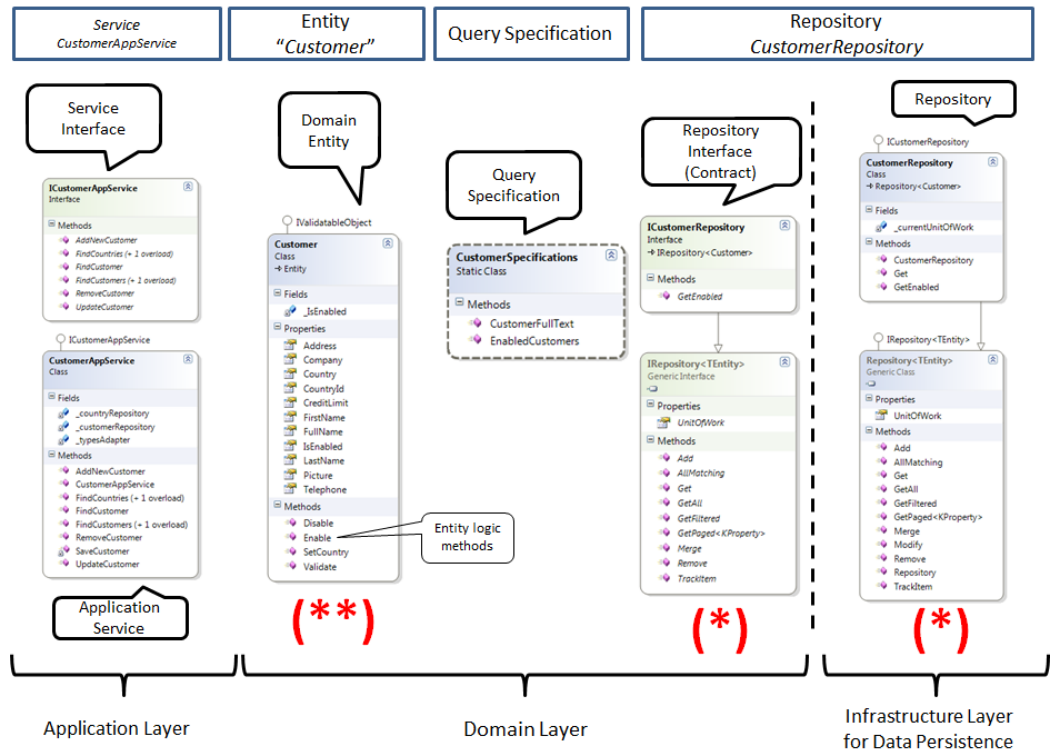
This consumer code is not part of the Application Layer. Typically, this kind of code would be implemented within the starting point of a WCF Services Layer or in an ASP.NET web presentation layer.

```
C#
//Code within the InstanceProvider of WCF service layer or in ASP.NET
//It must be the inception point or entry point to your appserver
//...
//This method is part of a WCF Instance point
public object GetInstance(InstanceContext instanceContext,
                        System.ServiceModel.Channels.Message message)
{
    //This is the only call to UNITY container in the whole solution
    return _container.Resolve(_serviceType);
}
```

In the Distributed services layer chapter we will explicitly explain where this line of code has to be implemented in order to have a single call (Resolve<>) to the IoC container (Within a WCF InstanceProvider).



The following diagram shows the most related classes to the "Customer" Domain Entity:



**Figure 9.- App Service and Repository Classes Diagram**

Although it may seem that there are many classes and interfaces associated with a single Domain entity, they are necessary if there is a need for decoupling; in fact, it does not require much work to be implemented, because:

- Of all these classes, the ones marked with one asterisk (\*) are base classes, so they are only implemented once for the entire project.
- The "Customer" Domain entity class marked with two asterisks (\*\*) is also implemented once for each MODEL, so it does not require any work right now.
- Interfaces are only method declarations or signatures, just like a contract, very quick to create and modify.

Therefore, we only need to implement the "CustomerAppService" service class itself, together with the Application layer logic that we require, and also the "CustomerRepository" repository with its persistence logic and data access if we cannot reuse the methods already contained in the base repository class.



## 4.2.- Implementing Transactions and using the UoW pattern in the Application Layer Services

Before showing the internal implementation of the sample Service, we will first show the different options for transaction implementation in .NET. Then we will implement it in the code of the sample Service “BankTransferService” because this sample implementation is closely associated with the implementation of transactions in .NET.



### 4.2.1.- Transactions in .NET

A transaction is an exchange of information and associated sequential actions treated as an atomic unit in order to satisfy a request, and simultaneously ensuring, specific data integrity. A transaction is only deemed complete if all the transaction information and actions have been completed and all the associated changes to the database are permanently applied. The transactions support the "undo" action (*rollback*) when there is a mistake, which helps to preserve the data integrity in the databases.

Historically, there have been many possible ways of implementing transactions in .NET. Basically, the following options are available:

- Transactions in TSQL (In their own SQL statement).
- ADO.NET transactions (Based on the Connection and Transaction objects)
- Enterprise Services transactions (Distributed transactions based on COM+)
- **Transactions System.Transactions (local transactions and upgradable to distributed transactions).**

The first type (transactions in SQL statements and/or stored procedures) is feasible for any programming language and platform (.NET, VB, Java, etc.) and is the one that may achieve the best performance. Therefore, for special and specific cases, it may be the most suitable approach. However, using it in an N-layer architecture business application is not recommended because it has the huge disadvantage of having the transaction concept completely coupled (a business concept such as a transfer) with the data access code (SQL statements). Remember that one of the basic rules of a Domain Oriented application is that the application code and domain/business should be completely separated and decoupled from the persistence and data access code. **Transactions should be exclusively declared/implemented in the Application layer** (or Domain layer depending on the preference, but we think that transactions code and classes (kind of ‘technical plumbing’) are not really pure Domain Logic but technical

implementation of our application, therefore we think it should be done within the Application Layer).

On the other hand, in .NET 1.x we basically had two main options, ADO.NET and COM+ transactions with *Enterprise Services*. If the ADO.NET transactions were used in an application, we should keep in mind that these transactions are closely linked to the *Transaction and Database Connection* objects, which are associated with the data access level. It is therefore very difficult to define transactions exclusively in the business component level (only through a Framework itself based on aspects, etc.) In short, we have a problem similar to using transactions in SQL statements. Now, however, instead of defining transactions in the SQL itself, we would be tightly coupled to the ADO.NET objects implementation. It is not the ideal context for transactions that should be defined exclusively at the business level.

Another option enabled by .NET Framework 1.x was to use *Enterprise Services* transactions (based on *COM+*), which can be exclusively specified at the business class level (through .NET attributes). However, in this case we have the problem wherein its use seriously impacts performance (*Enterprise Services* are based on *COM+* and therefore *COMInterop* is used from .Net, as well as an inter-process communication with *DTC*). The development also becomes more tedious because the components must be signed with a safe name (*strong-name*) and recorded as *COM* components in *COM+*.

However, starting on **.NET 2.0** (also continued in .NET 3.0, 3.5 and 4.0) we have the '***System.Transactions***' namespace. This is generally the most recommended way of implementing transactions because of its flexibility and higher performance when dealing with Enterprise Services. This is especially true as of SQL Server 2005 where there is a **possibility of “automatic promotion from a local transaction to a distributed transaction”**.

The below Table summarizes the different technological options to coordinate transactions in .NET:

**Table 4.- Technological options to coordinate transactions in .NET**

Type of transaction	V. Framework .NET	Description
<b>Internal transactions with T-SQL (in DB)</b>	From .NET Framework 1.0, 1.1	Transactions internally implemented in its own SQL statements (it can also be defined in stored procedures)
<b>Enterprise Service transactions (COM+)</b>	From .NET Framework 1.0, 1.1	<ul style="list-style-type: none"> <li>- Enterprise Services (COM+)</li> <li>- Web ASP.NET transactions</li> <li>- XML Web Service (WebMethod) transactions</li> </ul>
<b>ADO.NET transactions</b>	From .NET Framework 1.0, 1.1	Implemented with ADO.NET Transaction and Connection objects.
<b>System.Transactions</b>	.NET Framework 2.0,	Allows local transactions that are

<b>transactions</b>	3.0, 3.5 and 4.0	upgradable to distributed transactions. It is the preferred option.
---------------------	------------------	---

The next table shows the resources and objectives as well as assumptions and the transaction technology to be used:

**Table 5.- Resources and goal premises**

<b>What do I have? + Objectives</b>	<b>What to use</b>
<ul style="list-style-type: none"> <li>- A SQL Server 2005/2008/2008R2 for most transactions and there could also be distributed transactions with other DBMS and/or transactional environments 'Two Phase Commit'</li> <li>- Objective: Maximum performance in local transactions</li> </ul>	→ System.Transactions (From .NET 2.0)
<ul style="list-style-type: none"> <li>- Only one older DBMS server (e.g. SQL Server 2000), for the same transactions</li> <li>- Objective: Maximum flexibility in the business components design.</li> </ul>	→ System.Transactions (From .NET 2.0)
<ul style="list-style-type: none"> <li>- Only one older DBMS server (e.g. SQL Server 2000), for the same transactions</li> <li>- Objective: Maximum performance in local transactions</li> </ul>	→ ADO.NET transactions
<ul style="list-style-type: none"> <li>- 'n' DBMS servers and Transactional Data Sources for Distributed Transactions.</li> <li>- Objective: Maximum integration with other Transactional environments (HOST, MSMQ transactions, etc.)</li> </ul>	→ System.Transactions (From .NET 2.0)  → Enterprise Services (COM+) could be used too, but it is an older technology associated with COM+ and COM components.
<ul style="list-style-type: none"> <li>- Any DBMS and execution of very critical specific transaction regarding its maximum performance.</li> <li>- Objective: Maximum full performance, even when rules of design in N-layers are broken.</li> </ul>	→ Internal transactions with Transact-SQL

Therefore, as a general rule and with a few exceptions, **the best option is System.Transactions.**

**Table 6.- Framework Architecture Guide**



**The transaction management system to be used by default in .NET will be 'System.Transactions'**

#### Rule

- The most powerful and flexible system for transaction implementation in .NET is **System.Transactions**. It offers aspects such as upgradable transactions and maximum flexibility by supporting local and distributed transactions.
- For most transactions of an N-layer application, the recommendation is to use the **implicit model** of **System.Transactions**, that is to say, using '**TransactionScope**'. Although this model is not at the same performance level as manual or explicit transactions, they are the easiest and clearest to develop, so they adapt very well to the Domain layers. If we do not want to use the Implicit Model (TransactionScope), we can implement the Manual Model by using the **Transaction** class of the **System.Transactions namespace**. Consider it for certain cases or those with heavier transactions.



#### **References**

##### ***ACID Properties***

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconacidproperties.asp>

##### ***System.Transactions***

<http://msdn.microsoft.com/en-us/library/system.transactions.aspx>

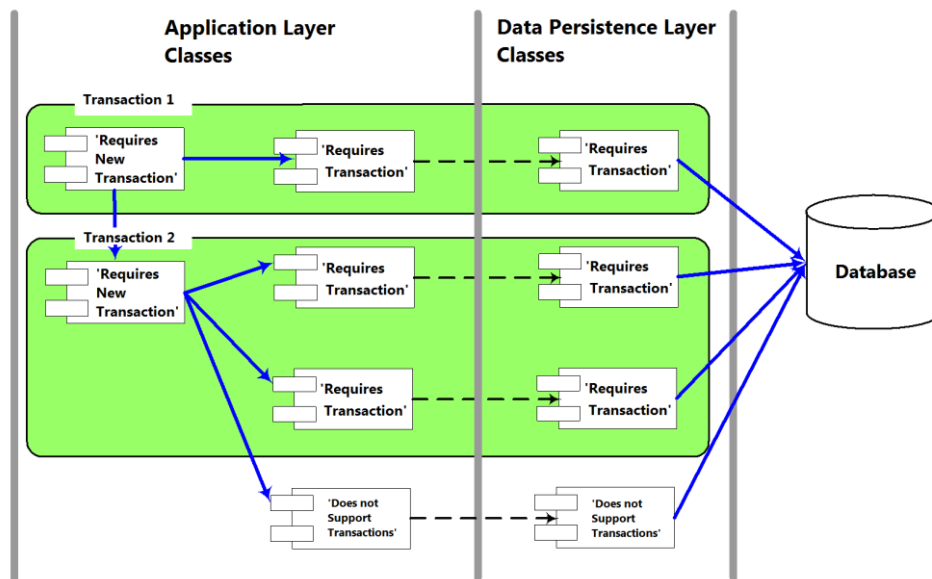


#### 4.2.2.- Transaction Implementation in the Application Services Layer

The initiation and coordination of transactions following a correct design should generally be done in the SERVICES of the APPLICATION LAYER. This is also feasible in the Domain layer, as preferred. However, in this guide, as we've explained, we propose to perform all the 'plumbing coordination', such as use of Repositories and UoW from the application layer in order to leave the Domain layer much cleaner with only pure business logic.

Most application designs with business transactions should include transaction management in its implementation, so that a **sequence of operations can be performed as a single unit of work and be completely or unitarily applied or revoked if there is a mistake being made.**

**Any N-layer application should have the ability to establish transactions at the business or application component levels and not embed them within the data layer, as shown in this scheme:**



**Figure 10.- Transactions diagram - Classes level**

All native standard transactions (not compensatory transactions) must satisfy the **ACID principles**:

- **Atomicity:** A transaction should be an atomic unit of work, that is, everything is done or nothing is done.

- **Consistency:** It should leave data in a consistent and coherent state once the transaction is over.
- **Isolation:** Modifications made by transactions are independently treated, as if there were only one user of the database.
- **Durability:** After the transaction is completed, its effects will be permanent and they will not be undone.



### 4.2.3.- Concurrency Model during Updates

It is important to identify the proper concurrency model and to determine how to manage the transactions. For concurrency, we can mostly choose between an optimistic or pessimistic model. Using the optimistic concurrency model, there are no blockages maintained in the data sources for long periods; however, the updates require certain checking code, generally against a ‘*timestamp*’ in order to verify that the original data to be modified has not changed in the data source (DB) since the original time they were obtained. In the pessimistic concurrency model, data are blocked and they cannot be updated by any other operation until they are unblocked. The pessimistic model is quite typical for Client/Server applications where support for a great scalability of concurrent users is not required (e.g., thousands of concurrent users). On the other hand, **the optimistic concurrency model is much more scalable because it does not maintain such a high level of blockages in the database and is therefore the model to be chosen by most Web applications, N-Tier, and SOA.**

Table 7.- Architecture Framework Guide



The concurrency model by default will be “Optimistic Concurrency”.

#### ○ Rule

- By default, the concurrency model in DDD N-layer applications with SOA, N-Tier or Web deployment will be the ‘Optimistic Concurrency’ model. At the implementation level, it is much easier to implement an Optimistic Concurrency exception management using *Entity Framework Self Tracking entities (STE)*. But it can also be implemented manually using DTOs plus

Domain entities.

Of course, if there are important reasons for using the pessimistic concurrency model in specific cases, then it should be used but as an exception.



### **Advantages**

- Higher scalability and independence of data sources.
- Fewer blockages in database than the pessimistic model.
- For applications that require high scalability, such as Internet applications, it is mandatory to use this type of concurrency model.



### **Disadvantages**

- Higher effort in managing exceptions while developing, if there is not additional help such as *Entity Framework 'Self-Tracking Entities'*.
- In certain on-off operations where the concurrency control and the operation order are critical and we do not intend to depend on the end user's decisions when exceptions occur, the pessimistic concurrency model always offers a stronger and tighter concurrency control.
- If there is a high possibility of data conflicts due to concurrent users working, then consider using the pessimistic concurrency to avoid a high number of exceptions to be decided by the end users.



## **4.2.4.- Types of Transaction Isolation**


Use a suitable isolation level for the transaction. There should be a balance between consistency and containment. That is, a high level of transaction isolation will offer a high level of data consistency, but it will have a higher level of blockages. On the other hand, a lower transaction isolation level will improve overall performance by lowering containment, but the level of consistency may be lower.

Therefore, when executing a transaction, it is important to know the different types of isolation options available in order to apply the most suitable for the operation to be performed. These are the most common ones:



- **Serialized:** data read by the current transaction will not be modified by other transactions until the current transaction is completed. No new data will be inserted during the execution of this transaction.
- **Repeatable Read:** data read by the current transaction will not be modified by other transactions until the current transaction is completed. New data could be inserted during the execution of this transaction.
- **Read Committed:** a transaction will not read data being modified by another transaction if it is not reliable. This is the Microsoft SQL Server and Oracle's default isolation level.
- **Read Uncommitted:** a transaction will read any data, even though it is being modified by another transaction. This is the lowest possible level of isolation, although it allows higher data concurrency.

**Table 8.- Architecture Framework Guide**

 <b>Rule N°: I10.</b>	<p>The level of isolation should be considered in each application and application area. The most common are 'Read-Committed' or 'Serialized'.</p>
<p>○ <b><u>Recommendation</u></b></p>	<p>In cases where the transaction has a critical level of importance, use of the 'Serialized' level is recommended, although we should be aware that this level will decrease performance and increase the surface blockage in the database.</p> <p>In any case, the transaction isolation level should be analyzed depending on the particular case of each application.</p>

Consider the following guidelines when designing and implementing transactions:

- Consider what the boundaries of transactions are, and activate them only if necessary. In general, the queries will not require explicit transactions. It is also convenient to know the database transaction isolation level. By default, SQL Server runs each individual SQL statement as an individual transaction (*auto-commit transactional mode*).

- Transactions should be as short in duration as possible to minimize the blockages time maintained in the database tables. Also, avoid blockages in shared data as much as possible because they may block access to another code. Avoid using exclusive blockage because it may cause inter-blocking.
- Avoid blockages in long running transactions. In cases where we have long-running processes but we would like them to behave as one transaction, compensatory methods should be implemented in order to return data to the initial state in case an operation fails.

Below, there is a sample of an Application SERVICE class method (**BankTransferService**) that initiates a transaction involving operations of associated Repositories, Domain Services and Domain Entities to persist changes in the operation:

```

C#
...
namespace
Microsoft.Samples.NLayerApp.Application.MainBoundedContext.BankingModule.Services

public class BankAppService : IBankAppService
{
    IBankAccountRepository _bankAccountRepository;
    ICustomerRepository _customerRepository;
    IBankTransferService _transferService;
    ITypeAdapter adapter;

    public BankAppService (IBankAccountRepository bankAccountRepository,
        ICustomerRepository customerRepository,
        IBankTransferService transferService,
        ITypeAdapter adapter)
    {
        //...Ommited preconditions checking...

        _bankAccountRepository = bankAccountRepository;
        customerRepository = customerRepository;
        adapter = adapter;
        transferService = transferService;
    }
    //...Other AppService methods ommitted...

    public void PerformBankTransfer (BankAccountDTO fromAccount,
        BankAccountDTO toAccount,
        decimal amount)
    {
        //Application-Logic Process:
        // 1° Get Accounts objects from Repositories
        // 2° Start Transaction
        // 3° Call PerformTransfer() method in the Domain Service
        // 4° If no exceptions, commit the unit of work and complete transaction

        if (BankAccountHasIdentity(fromAccount)
            &&
            BankAccountHasIdentity(toAccount))
    }
}

```

*Namespace of the Application Layer Services in a sample module*

**Application Service**

**Contract/Interface to comply**

**Constructor with Dependencies Injection**

**Application Service Method to perform a Transaction. Only 'Plumbing coordination', NO Domain logic**



Some considerations regarding the example above are found below:

- As shown, this Application Layer Service is where we implement all the ‘plumbing’ coordination. In other words, the creation of a transaction and configuration of its type, use of ‘Unit of Work’, calls to Repositories to obtain entities and to finally persist them, etc. Ultimately, this includes all the necessary coordination of the application which is basically the aspects that we would not discuss with a business/domain expert. Instead, the entire Domain logic (BankTransfer operations) is encapsulated within the Domain Service and business logic of the entities themselves (in this case, the BankAccount entity and the BankTransfer Domain Service).
- Since **using** is being employed with the *TransactionScope*, it is not necessary to manually manage the transaction *rollback*. Any exception being thrown during insertion of any of the regions will cause the transaction to be aborted.

- The **UoW** (*Unit of work*) enables a context where the Repositories mark/record the persistence operations intended to perform, but they are not actually persisted on the database until we explicitly call to the method 'unitOfWork.Commit ()'.

## Transactions Nesting

*System.Transactions* allows nesting transactions. A common example is having another "TransactionScope" within an internal method (for example, in one of the methods of the "BankAccount" class, etc.). The original transaction will be extended with the new TransactionScope in one way or another, depending on the specified 'TransactionScopeOption' in the internal TransactionScope.

As shown, the advantage of this model resides in its flexibility and ease of development.

**Table 9.- Transaction isolation type**



**The type of TransactionScope by default will be 'Required'.**

### ○ Recommendation

- If a transaction scope is not specified in the services at the lowest level, that is, the ones using REPOSITORIES, then operations will be listed to the highest transaction level that can be created. But if in these SERVICES we also implement *TransactionScope*, it should be configured as 'Required.'

This is because, in the case of calling the Service with our transaction from a code that has not created any transaction yet, then a new transaction will be created with the corresponding operations. However, if it is called from another class/service that has already created a transaction, this call will simply extend the current transaction. Then, as 'Required' (TransactionScopeOption.Required) it will be correctly aligned to the existing transaction. On the other hand, if it appears as "RequiredNew", although there is an initial transaction in existence, a new transaction would be created by calling this transaction. Of course, all this depends on the specific business rules involved. In some cases, we might be interested in this other behavior.

This transaction configuration is implemented through the *System.Transactions* '*TransactionScope()*' syntax.



## References

Introducing System.Transactions in the .NET Framework 2.0:

**<http://msdn2.microsoft.com/en-us/library/ms973865.aspx>**

Concurrency Control at **<http://msdn.microsoft.com/enus/library/ms978457.aspx>**.

Integration Patterns at **<http://msdn.microsoft.com/enus/library/ms978729.aspx>**.



### 4.3.- Testing Implementation in the Application Layer

The application layer tests should normally be *tested*, especially the Application Services.

The application service tests are relatively complex because they involve dependencies to other internal elements or other services (application or domain services) while, of course, invoking domain entity logic.

```
C#
[TestClass()]
public class BankManagementServiceTests
{
    [TestMethod()]
    public void PerformBankTransfer()
    {
        //Arrange
        var sourceId = new Guid("3481009C-A037-49DB-AE05-44FF6DB67DEC");
        var bankAccountNumberSource = new BankAccountNumber("4444",
                                                                "5555",
                                                                "3333333333",
                                                                "02");

        var source = BankAccountFactory.CreateBankAccount(Guid.NewGuid(),
                                                            bankAccountNumberSource);
        source.Id = sourceId;
        source.DepositMoney(1000, "initial");

        var sourceBankAccountDTO = new BankAccountDTO()
        {
            Id = sourceId,
            BankAccountNumber = source.Iban
        };

        var targetId = new Guid("8A091975-F783-4730-9E03-831E9A9435C1");
        var bankAccountNumberTarget = new BankAccountNumber("1111",
                                                                "2222",
                                                                "3333333333",
                                                                "01");

        var target = BankAccountFactory.CreateBankAccount(Guid.NewGuid(),
                                                            bankAccountNumberTarget);
        target.Id = targetId;

        var targetBankAccountDTO = new BankAccountDTO()
        {
            Id = targetId,
            BankAccountNumber = target.Iban
        };

        var accounts = new List<BankAccount>() { source, target };
        var accountsDTO = new List<BankAccountDTO>() { sourceBankAccountDTO,
                                                         targetBankAccountDTO };

        SIBankAccountRepository bankAccountRepository =
```

```

        new SIBankAccountRepository();
bankAccountRepository.GetGuid = (guid) =>
{
    return accounts.Where(a => a.Id == guid).SingleOrDefault();
};
bankAccountRepository.UnitOfWorkGet = () =>
{
    var unitOfWork = new SIUnitOfWork();
    unitOfWork.Commit = () => { };

    return unitOfWork;
};

SICustomerRepository customerRepository =
    new SICustomerRepository();

IBankTransferService transferService = new BankTransferService();
ITypeAdapter adapter = PrepareTypeAdapter();

IBankAppService bankingService = new
BankAppService(bankAccountRepository, customerRepository, transferService,
adapter);

//Act
bankingService.PerformBankTransfer(sourceBankAccountDTO,
    targetBankAccountDTO, 100M);

}

}

```