

Arquitectura Marco .NET

Microsoft Ibérica

I.- INTRODUCCIÓN

Microsoft Ibérica ha detectado en múltiples clientes y *partners* la necesidad de disponer de una “Guía de Arquitectura base .NET” en español, que sirva para marcar unas líneas maestras de diseño e implementación a la hora de desarrollar aplicaciones .NET complejas y con una vida y evolución de larga duración. Este marco de trabajo común (en muchas empresas denominado “Libro Blanco”) define un camino para diseñar e implementar aplicaciones empresariales de envergadura, con un volumen importante de lógica de negocio. Seguir estas guías ofrece importantes beneficios en cuanto a calidad, estabilidad y, especialmente, un incremento en la facilidad del mantenimiento futuro de las aplicaciones, debido al desacoplamiento entre sus componentes, así como por la homogeneidad y similitudes de los diferentes desarrollos a realizar.

Microsoft Ibérica define el presente ‘**Libro de Arquitectura Marco**’ como patrón y modelo base, sin embargo, en ningún caso este marco debe ser inalterable. Al contrario, se trata del primer peldaño de una escalera, un acelerador inicial, que debería ser personalizado y modificado por cada organización que lo adopte, enfocándolo hacia necesidades concretas, adaptándolo y agregándole funcionalidad específica según el mercado objetivo, etc.

I.1.- Audiencia del documento

Este documento está dirigido a las personas involucradas en todo el ciclo de vida de productos *software* o de aplicaciones corporativas desarrolladas a medida. Especialmente los siguientes perfiles:

- Arquitecto de Software
- Desarrollador

I.2.- Objetivos de la Arquitectura Marco .NET

Este documento pretende describir una arquitectura marco sobre la que desarrollar las aplicaciones a medida y establece un conjunto de normas, mejores prácticas y guías de desarrollo para utilizar .NET de forma adecuada y, sobre todo, homogénea.

DESCARGO DE RESPONSABILIDAD:

Queremos insistir en este punto y destacar que la presente propuesta de '*Arquitectura N-Capas Orientada al Dominio*' no es adecuada para cualquier tipo de aplicaciones, **solamente es adecuada para aplicaciones complejas empresariales con un volumen importante de lógica de negocio y una vida y evolución de aplicación de larga duración**, donde es importante implementar conceptos de desacoplamiento y ciertos patrones DDD. Para aplicaciones pequeñas y orientadas a datos, probablemente sea más adecuada una aproximación de arquitectura más sencilla implementada con tecnologías RAD.

Así mismo, esta guía (y su aplicación ejemplo asociada) es simplemente una propuesta a tener en cuenta y ser evaluada y personalizada por las organizaciones y empresas que lo deseen. *Microsoft Ibérica* no se hace responsable de problemas que pudieran derivarse de ella.

I.3.- Niveles de la documentación de la Arquitectura marco .NET

La documentación de esta arquitectura se diseña en dos niveles principales:

- **Nivel lógico de Arquitectura de Software:** Este primer nivel lógico, es una Arquitectura de software agnóstica a la tecnología, donde no se especifican tecnologías concretas de .NET. Para resaltar este nivel, se mostrará el icono:



- **Nivel de Implementación de Arquitectura .NET:** Este segundo nivel, es la implementación concreta de Arquitectura .NET, donde se enumerarán las tecnologías posibles para cada escenario con versiones concretas; normalmente se escogerá una opción y se explicará su implementación. Así mismo, la implementación de la arquitectura cuenta con una aplicación .NET ejemplo, cuyo alcance funcional es muy pequeño, pero debe implementar todas y cada una de las áreas tecnológicas de la Arquitectura marco. Para resaltar este nivel, se mostrará el icono de .NET al inicio del capítulo:



I.4.- Aplicación Ejemplo en CODEPLEX

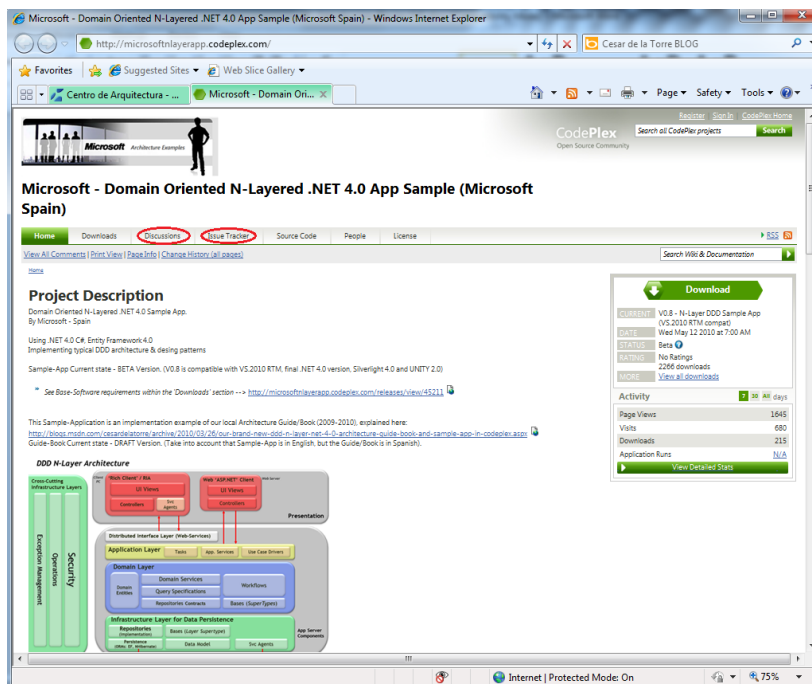
Es fundamental destacar que simultáneamente a la elaboración de este libro/guía de Arquitectura, también hemos desarrollado una aplicación ejemplo, que implementa los patrones expuestos en esta guía, con las últimas tecnologías actuales de Microsoft ('Ola .NET 4.0').

Así mismo, la mayoría de los *snippets* de código mostrados en este libro, son extractos de código precisamente de esta Aplicación ejemplo.

Esta aplicación ejemplo está publicada en CODEPLEX como código OSS y se puede descargar desde la siguiente URL:



En CODEPLEX disponemos no solo del código fuente de la aplicación ejemplo, también de cierta documentación sobre requerimientos (tecnologías necesarias como *Unity 2.0*, *PEX & MOLES*, *WPF Toolkit*, *Silverlight 4 Tools for Visual Studio 2010*, *Silverlight 4.0 Toolkit*, *AppFabric*, etc., links desde donde descargarlas en Internet, etc.) y de una página de **Discusiones/Foro**, algo muy interesante para poder colaborar con la comunidad, y poder también presentarnos preguntas, ideas, propuestas de evolución, etc.:



La aplicación ejemplo implementa los diferentes patrones de Diseño y Arquitectura DDD, pero con las últimas tecnologías Microsoft. También dispone de varios clientes (WPF, Silverlight, ASP.NET MVC) y otros a ser añadidos como OBA y Windows Phone 7.0, etc.

Es importante resaltar que la funcionalidad de la aplicación ejemplo, es lógicamente, bastante sencilla, pues lo que se quiere resaltar es la Arquitectura, no implementar un volumen grande de funcionalidad que complique el seguimiento y entendimiento de la Arquitectura.

La Capa de presentación y las diferentes implementaciones son simplemente un área más de la arquitectura y no son precisamente ‘el *core*’ de esta guía de referencia, donde nos centramos más en capas relativas al servidor de componentes (Capa del Dominio, de Aplicación, Infraestructura de acceso a datos, son sus respectivos patrones). Aun así, se hace también una revisión de los diferentes patrones en capa de presentación (MVC, M-V-VM, etc.) y como implementarlos con diferentes tecnologías.

Aquí mostramos algunas pantallas capturadas de la aplicación ejemplo:

Cliente Silverlight 4.0

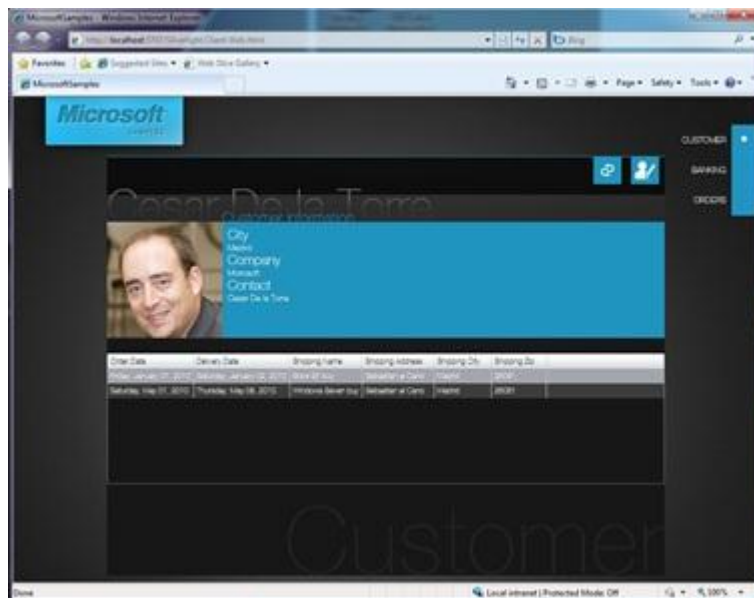
Silverlight – Lista de Clientes



Silverlight – Transición de Silverlight

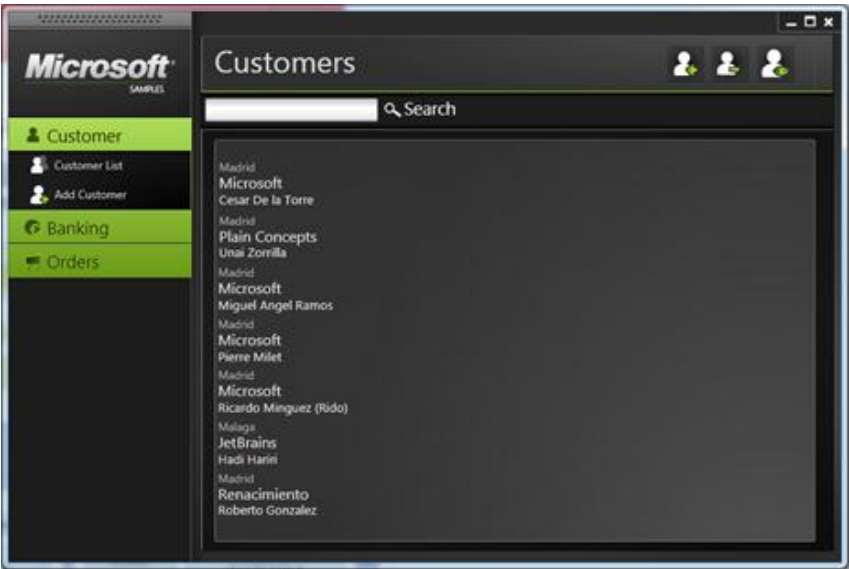


Silverlight – 'Vista de Cliente'



Cliente WPF 4.0

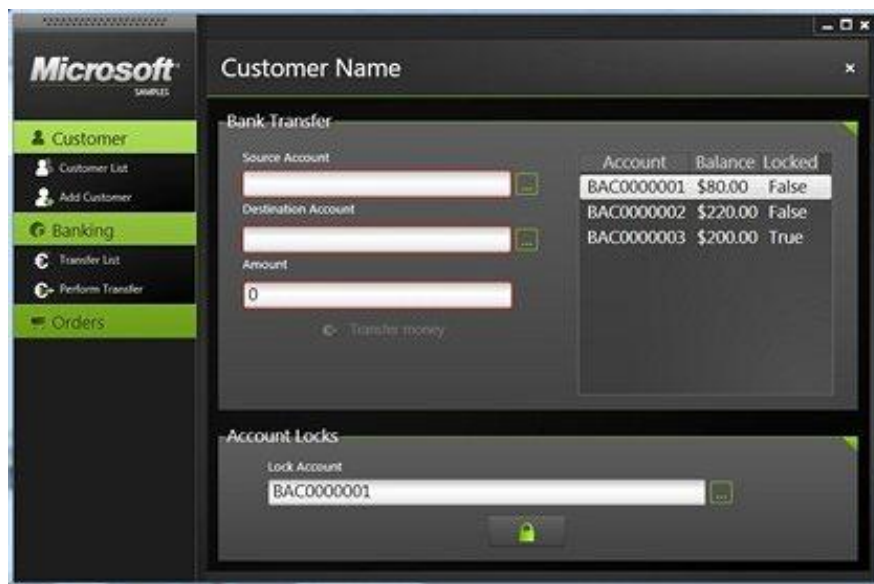
WPF – Vista de ‘Lista de Clientes’



WPF – Vista de Cliente

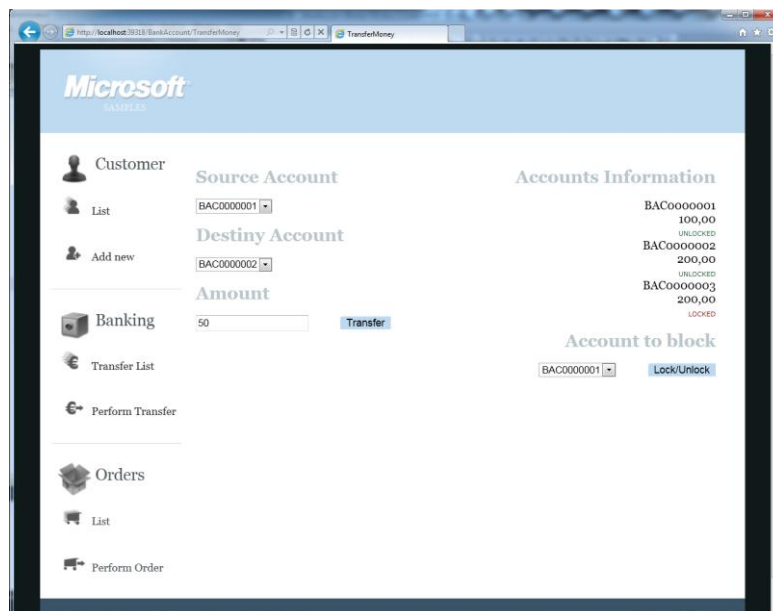


WPF – ‘Transferencias Bancarias’

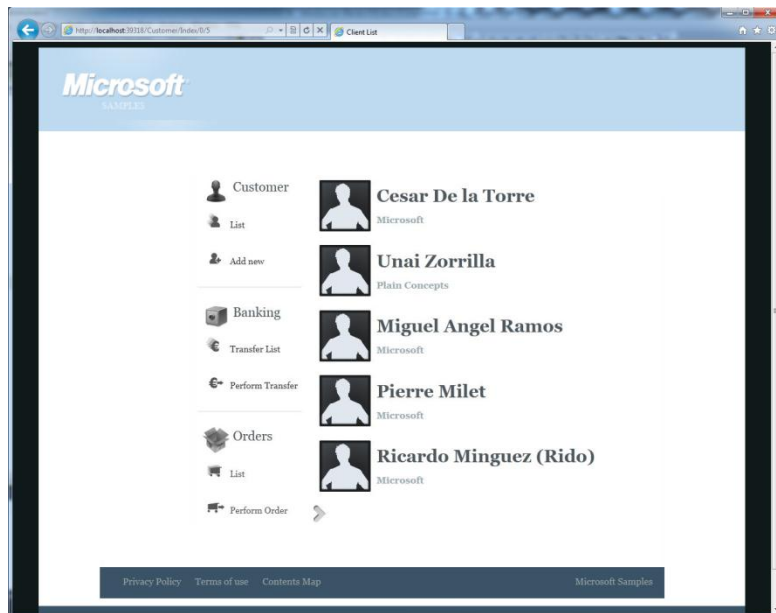


Cliente ASP.NET MVC

MVC – ‘Transferencias Bancarias’



MVC – Vista de ‘Lista de Clientes’



Por último, resaltar que tanto la aplicación como todo el código fuente e dicha aplicación, lo hemos elaborado en inglés, para poder ser aprovechada por toda la comunidad, a nivel mundial y no solo en Español.

Recomendamos ‘bajar’ de Internet esta aplicación ejemplo e irla investigando en paralelo según se lee la presente guía/libro de Arquitectura, especialmente cuando se está leyendo los apartados de implementación marcados con el siguiente logo de .NET:





CAPÍTULO

Fundamentos de Arquitectura de Aplicaciones



El diseño de la arquitectura de un sistema es el proceso por el cual se define una solución para los requisitos técnicos y operacionales del mismo. Este proceso define qué componentes forman el sistema, cómo se relacionan entre ellos, y cómo mediante su interacción llevan a cabo la funcionalidad especificada, cumpliendo con los criterios de calidad indicados como seguridad, disponibilidad, eficiencia o usabilidad.

Durante el diseño de la arquitectura se tratan los temas que pueden tener un impacto importante en el éxito o fracaso de nuestro sistema. Algunas preguntas que hay que hacerse al respecto son:

- ¿En qué entorno va a ser desplegado nuestro sistema?
- ¿Cómo va a ser nuestro sistema puesto en producción?
- ¿Cómo van a utilizar los usuarios nuestro sistema?
- ¿Qué otros requisitos debe cumplir el sistema? (seguridad, rendimiento, concurrencia, configuración...)
- ¿Qué cambios en la arquitectura pueden impactar al sistema ahora o una vez desplegado?

Para diseñar la arquitectura de un sistema es importante tener en cuenta los intereses de los distintos agentes que participan. Estos agentes son los usuarios del sistema, el propio sistema y los objetivos del negocio. Cada uno de ellos impone requisitos y restricciones que deben ser tenidos en cuenta en el diseño de la arquitectura y que pueden llegar a entrar en conflicto, por lo que se debe alcanzar un compromiso entre los intereses de cada participante.

Para los usuarios es importante que el sistema responda a la interacción de una forma fluida, mientras que para los objetivos del negocio es importante que el sistema

cueste poco. Los usuarios pueden querer que se implemente primero una funcionalidad útil para su trabajo, mientras que el sistema puede tener prioridad en que se implemente la funcionalidad que permita definir su estructura.

El trabajo del arquitecto es delinear los escenarios y requisitos de calidad importantes para cada agente así como los puntos clave que debe cumplir y las acciones o situaciones que no deben ocurrir.

El objetivo final de la arquitectura es identificar los requisitos que producen un impacto en la estructura del sistema y reducir los riesgos asociados con la construcción del sistema. La arquitectura debe soportar los cambios futuros del software, del hardware y de funcionalidad demandada por los clientes. Del mismo modo, es responsabilidad del arquitecto analizar el impacto de sus decisiones de diseño y establecer un compromiso entre los diferentes requisitos de calidad así como entre los compromisos necesarios para satisfacer a los usuarios, al sistema y los objetivos del negocio.

En síntesis, la arquitectura debería:

- Mostrar la estructura del sistema pero ocultar los detalles.
- Realizar todos los casos de uso.
- Satisfacer en la medida de lo posible los intereses de los agentes.
- Ocuparse de los requisitos funcionales y de calidad.
- Determinar el tipo de sistema a desarrollar.
- Determinar los estilos arquitecturales que se usarán.
- Tratar las principales cuestiones transversales.

Una vez vistas las principales cuestiones que debe abordar el diseño de la arquitectura del sistema, ahora vamos a ver los pasos que deben seguirse para realizarlo. En una metodología ágil como *Scrum*, la fase de diseño de la arquitectura comienza durante en el pre-juego (Pre-game) o en la fase de Inicio (*Inception*) en RUP, en un punto donde ya hemos capturado la visión del sistema que queremos construir. En el diseño de la arquitectura lo primero que se decide es el tipo de sistema o aplicación que vamos a construir. Los principales tipos son aplicaciones móviles, de escritorio, RIAs (*Rich Internet Application*), aplicaciones de servicios, aplicaciones web... Es importante entender que el tipo de aplicación viene determinado por la topología de despliegue y los requisitos y restricciones indicadas en los requisitos.

La selección de un tipo de aplicación determina en cierta medida el estilo arquitectural que se va a usar. El estilo arquitectural es en esencia la partición más básica del sistema en bloques y la forma en que se relacionan estos bloques. Los principales estilos arquitecturales son Cliente/Servidor, Sistemas de Componentes, Arquitectura en capas, MVC, N-Niveles, SOA... Como ya hemos dicho, el estilo

arquitectural que elegimos depende del tipo de aplicación. Una aplicación que ofrece servicios lo normal es que se haga con un estilo arquitectural SOA.

Por otra parte, a la hora de diseñar la arquitectura tenemos que entender también que un tipo de aplicación suele responder a más de un estilo arquitectural. Por ejemplo, una página web hecha con ASP.NET MVC sigue un estilo Cliente/Servidor pero al mismo tiempo el servidor sigue un estilo Modelo Vista Controlador.

Tras haber seleccionado el tipo de aplicación y haber determinado los estilos arquitecturales que más se ajustan al tipo de sistema que vamos a construir, tenemos que decidir cómo vamos a construir los bloques que forman nuestro sistema. Por ello el siguiente paso es seleccionar las distintas tecnologías que vamos a usar. Estas tecnologías están limitadas por las restricciones de despliegue y las impuestas por el cliente. Hay que entender las tecnologías como los ladrillos que usamos para construir nuestro sistema. Por ejemplo, para hacer una aplicación web podemos usar la tecnología ASP.NET o para hacer un sistema que ofrece servicios podemos emplear WCF.

Cuando ya hemos analizado nuestro sistema y lo hemos fragmentado en partes más manejables, tenemos que pensar como implementamos todos los requisitos de calidad que tiene que satisfacer. Los requisitos de calidad son las propiedades no funcionales que debe tener el sistema, como por ejemplo la seguridad, la persistencia, la usabilidad, la mantenibilidad, etc. Conseguir que nuestro sistema tenga estas propiedades va a traducirse en implementar funcionalidad extra, pero esta funcionalidad es ortogonal a la funcionalidad básica del sistema.

Para tratar los requisitos de calidad el primer paso es preguntarse ¿Qué requisitos de calidad requiere el sistema? Para averiguarlo tenemos que analizar los casos de uso. Una vez hemos obtenido un listado de los requisitos de calidad las siguientes preguntas son ¿Cómo consigo que mi sistema cumpla estos requisitos? ¿Se puede medir esto de alguna forma? ¿Qué criterios indican que mi sistema cumple dichos requisitos?

Los requisitos de calidad nos van a obligar a tomar decisiones transversales sobre nuestro sistema. Por ejemplo, cuando estamos tratando la seguridad de nuestro sistema tendremos que decidir cómo se autentican los usuarios, como se maneja la autorización entre las distintas capas, etc. De la misma forma tendremos que tratar otros temas como las comunicaciones, la gestión de excepciones, la instrumentación o el cacheo de datos.

Los procesos software actuales asumen que el sistema cambiará con el paso del tiempo y que no podemos saber todo a la hora de diseñar la arquitectura. El sistema tendrá que evolucionar a medida que se prueba la arquitectura contra los requisitos del mundo real. Por eso, no hay que tratar de formalizar absolutamente todo a la hora de definir la arquitectura del sistema. Lo mejor es no asumir nada que no se pueda comprobar y dejar abierta la opción de un cambio futuro. No obstante, sí que existirán algunos aspectos que podrán requerir un esfuerzo a la hora de realizar modificaciones. Para minimizar dichos esfuerzos es especialmente importante el concepto de desacoplamiento entre componentes. Por ello es vital identificar esas partes de nuestro sistema y detenerse el tiempo suficiente para tomar la decisión correcta. En síntesis las claves son:

- Construir ‘hasta el cambio’ más que ‘hasta el final’.
- Utilizar herramientas de modelado para analizar y reducir los riesgos.
- Utilizar modelos visuales como herramienta de comunicación.
- Identificar las decisiones clave a tomar.

A la hora de crear la arquitectura de nuestro sistema de forma iterativa e incremental, las principales preguntas a responder son:

- ¿Qué partes clave de la arquitectura representan el mayor riesgo si las diseño mal?
- ¿Qué partes de la arquitectura son más susceptibles de cambiar?
- ¿Qué partes de la arquitectura puedo dejar para el final sin que ello impacte en el desarrollo del sistema?
- ¿Cuáles son las principales suposiciones que hago sobre la arquitectura y como las verifico?
- ¿Qué condiciones pueden provocar que tenga que cambiar el diseño?

Como ya hemos dicho, los procesos modernos se basan en adaptarse a los cambios en los requisitos del sistema y en ir desarrollando la funcionalidad poco a poco. En el plano del diseño de la arquitectura, esto se traduce en que **definiremos la arquitectura del sistema final poco a poco**. Podemos entenderlo como un proceso de maduración, como el de un ser vivo. **Primero tendremos una arquitectura a la que llamaremos línea base y que es una visión del sistema en el momento actual del proceso. Junto a esta línea base tendremos una serie de arquitecturas candidatas que serán el siguiente paso en la maduración de la arquitectura.** Cada arquitectura candidata incluye el tipo de aplicación, la arquitectura de despliegue, el estilo arquitectural, las tecnologías seleccionadas, los requisitos de calidad y las decisiones transversales. Las preguntas que deben responder las arquitecturas candidatas son:

- ¿Qué suposiciones he realizado en esta arquitectura?
- ¿Qué requisitos explícitos o implícitos cumple esta arquitectura?
- ¿Cuáles son los riesgos tomados con esta evolución de la arquitectura?
- ¿Qué medidas puedo tomar para mitigar esos riesgos?
- ¿En qué medida esta arquitectura es una mejora sobre la línea base o las otras arquitecturas candidatas?

.....

Dado que usamos una metodología iterativa e incremental para el desarrollo de nuestra arquitectura, la implementación de la misma debe seguir el mismo patrón. La forma de hacer esto es mediante pruebas arquitecturales. Estas pruebas son pequeños desarrollos de parte de la aplicación (Pruebas de Concepto) que se usan para mitigar riesgos rápidamente o probar posibles vías de maduración de la arquitectura. Una prueba arquitectural se convierte en una arquitectura candidata que se evalúa contra la línea base. Si es una mejora, se convierte en la nueva línea base frente a la cual crear y evaluar las nuevas arquitecturas candidatas. Las preguntas que debemos hacerle a una arquitectura candidata que surge como resultado de desarrollar una prueba arquitectural son:

- ¿Introduce nuevos riesgos?
- ¿Soluciona algún riesgo conocido esta arquitectura?
- ¿Cumple con nuevos requisitos del sistema?
- ¿Realiza casos de uso arquitecturalmente significativos?
- ¿Se encarga de implementar algún requisito de calidad?
- ¿Se encarga de implementar alguna parte del sistema transversal?

Los casos de uso importantes son aquellos que son críticos para la aceptación de la aplicación o que desarrollan el diseño lo suficiente como para ser útiles en la evaluación de la arquitectura.

En resumen, el proceso de diseño de la arquitectura tiene que decidir qué funcionalidad es la más importante a desarrollar. A partir de esta decisión tiene que decidir el tipo de aplicación y el estilo arquitectural, y tomar las decisiones importantes sobre seguridad, rendimiento... que afectan al conjunto del sistema. El diseño de la arquitectura decide cuales son los componentes más básicos del sistema y como se relacionan entre ellos para implementar la funcionalidad. Todo este proceso debe hacerse paso a paso, tomando solo las decisiones que se puedan comprobar y dejando abiertas las que no. Esto significa mitigar los riesgos rápidamente y explorar la implementación de casos de uso que definan la arquitectura.

- Entorno de despliegue propuesto.

A partir de esta información deberemos generar los artefactos necesarios para que los programadores puedan implementar correctamente el sistema. Como mínimo, en el proceso de diseño de la arquitectura debemos definir:

- Casos de uso significativos a implementar.
- Riesgos a mitigar y cómo hacerlo.
- Arquitecturas candidatas a implementar.

Como ya hemos dicho, el diseño de la arquitectura es un proceso iterativo e incremental. En el diseño de la arquitectura repetimos 5 pasos hasta completar el desarrollo del sistema completo. Los pasos que repetimos y la forma más clara de verlos es esta:



Figura 1.- Diseño de Arquitectura

A continuación vamos a examinar en más detalle cada uno de estos pasos para comprender qué debemos definir y dejar claro en cada uno de ellos.

1.- IDENTIFICAR LOS OBJETIVOS DE LA ITERACIÓN

Los objetivos de la iteración son el primer paso para dar forma a la arquitectura de nuestro sistema. En este punto lo importante es analizar las restricciones que tiene nuestro sistema en cuanto a tecnologías, topología de despliegue, uso del sistema, etc... En esta fase es muy importante marcar cuales van a ser los objetivos de la arquitectura, tenemos que decidir si estamos construyendo un prototipo, realizando un diseño completo o probando posibles vías de desarrollo de la arquitectura. También hay que tener en cuenta en este punto a las personas que forman nuestro equipo. El tipo de documentación a generar así como el formato dependerá de si nos dirigimos a otros arquitectos, a desarrolladores, o a personas sin conocimientos técnicos.

El objetivo de esta fase del proceso de diseño de la arquitectura es entender por completo el entorno que rodea a nuestro sistema. Esto nos permitirá decidir en qué centraremos nuestra actividad en las siguientes fases del diseño y determinará el alcance y el tiempo necesarios para completar el desarrollo. Al término de esta fase deberemos tener una lista de los objetivos de la iteración, preferiblemente con planes para afrontarlos y métricas para determinar el tiempo y esfuerzo que requerirá completarlos. Tras esta fase es imprescindible tener una estimación del tiempo que invertiremos en el resto del proceso.

2.- SELECCIONAR LOS CASOS DE USO ARQUITECTURALMENTE IMPORTANTES

El diseño de la arquitectura es un proceso dirigido por el cliente y los riesgos a afrontar, esto significa que desarrollaremos primero los casos de uso (funcionalidad) que más valor tengan para el cliente y mitigaremos en primer lugar los riesgos más importantes que afronte nuestra arquitectura (requisitos de calidad). La importancia de un caso de uso la valoraremos según los siguientes criterios:

- Lo importante que es el caso de uso dentro de la lógica de negocio: Esto vendrá dado por la frecuencia de utilización que tendrá el caso de uso en el sistema en producción o el valor que aporte esa funcionalidad al cliente.
- El desarrollo del caso de uso implica un desarrollo importante de la arquitectura: Si el caso de uso afecta a todos los niveles de la arquitectura es un firme candidato a ser prioritario, ya que su desarrollo e implementación permitirán definir todos los niveles de la arquitectura aumentando la estabilidad de la misma.
- El desarrollo del caso de uso implica tratar algún requisito de calidad: Si el caso de uso requiere tratar temas como la seguridad, la disponibilidad o la tolerancia a fallos del sistema, es un caso de uso importante ya que permite tratar los aspectos horizontales del sistema a la vez que se desarrolla la funcionalidad.

- Lo que se adapte el caso de uso a los objetivos de la iteración: A la hora de seleccionar los casos de uso que vamos a implementar tenemos que tener en cuenta lo que se ajustan a los objetivos que nos hemos marcado para la iteración. No vamos a escoger casos de uso que desarrollen mucho el conjunto del sistema si nos hemos marcado como objetivo de la iteración reducir bugs o mitigar algún riesgo dado.

Es muy importante tener claro que no se debe tratar de diseñar la arquitectura del sistema en una sola iteración. En esta fase del proceso de diseño analizamos todos los casos de uso y seleccionamos solo un subconjunto, el más importante arquitecturalmente y procedemos a su desarrollo. En este punto, solo definimos los aspectos de la arquitectura que conciernen a los casos de uso que hemos seleccionado y dejamos abiertos el resto de aspectos para futuras iteraciones. Es importante recalcar que puede que en una iteración no definamos por completo algún aspecto del sistema, pero lo que tenemos que tener claro es que debemos intentar minimizar el número de cambios en futuras iteraciones. Esto no significa que no debamos “asumir que el software evoluciona”, sino que cuando desarrollemos un aspecto del sistema no nos atemos a una solución específica sino que busquemos una solución genérica que permita afrontar los posibles cambios en futuras iteraciones. En definitiva, todo esto se resume en dar pasos cortos pero firmes.

Es interesante a la hora de desarrollar el sistema tener en cuenta las distintas historias de usuario, sistema y negocio. Las historias de usuario, sistema y negocio son pequeñas frases o párrafos que describen aspectos del sistema desde el punto de vista del implicado. Las historias de usuario definen como los usuarios utilizarán el sistema, las historias de sistema definen los requisitos que tendrá que cumplir el sistema y como se organizará internamente y las historias de negocio definen como el sistema cumplirá con las restricciones de negocio.

Desmenuzar los casos de uso en varias historias de usuario, sistema y negocio nos permitirá validar más fácilmente nuestra arquitectura asegurándonos de que cumple con las historias de usuario, sistema y negocio de la iteración.

3.- REALIZAR UN ESQUEMA DEL SISTEMA

Una vez que están claros los objetivos de la iteración y la funcionalidad que desarrollaremos, podemos pasar a su diseño. Llegados a este punto, el primer paso es decidir qué tipo de aplicación vamos a desarrollar. El tipo de aplicación que elegiremos dependerá de las restricciones de despliegue, de conectividad, de lo compleja que sea la interfaz de usuario y de las restricciones de interoperabilidad, flexibilidad y tecnologías que imponga el cliente. Cada tipo de aplicación nos ofrece una serie de ventajas e inconvenientes, el arquitecto tiene que escoger el tipo de aplicación que mejor se ajuste a las ventajas que espera que tenga su sistema y que presente menos inconvenientes. Los principales tipos de aplicaciones que desarrollaremos son:

- **Aplicaciones para dispositivos móviles:** Se trata de aplicaciones web con una interfaz adaptada para dispositivos móviles o aplicaciones de usuario desarrolladas para el terminal.
- **Aplicaciones de escritorio:** Son las aplicaciones clásicas que se instalan en el equipo del usuario que la vaya a utilizar.
- **RIA (Rich Internet Applications):** Se trata de aplicaciones que se ejecutan dentro del navegador gracias a un plug-in y que ofrecen una mejor respuesta que las aplicaciones web y una interfaz de calidad similar a las aplicaciones de usuario con la ventaja de que no hay que instalarlas.
- **Servicios:** Se trata de aplicaciones que exponen una funcionalidad determinada en forma de servicios web para que otras aplicaciones los consuman.
- **Aplicaciones web:** Son aplicaciones que se consumen mediante un navegador y que ofrecen una interfaz de usuario estándar y completamente interoperable.

A modo de resumen y guía, la siguiente tabla recoge las principales ventajas y consideraciones a tener en cuenta para cada tipo de aplicación:

Tabla 1.- Ventajas y consideraciones tipos de aplicación

Tipo de aplicación	Ventajas	Consideraciones
Aplicaciones para dispositivos móviles	<ul style="list-style-type: none"> • Sirven en escenarios sin conexión o con conexión limitada. • Se pueden llevar en dispositivos de mano. • Ofrecen alta disponibilidad y fácil acceso a los usuarios fuera de su entorno habitual. 	<ul style="list-style-type: none"> • Limitaciones a la hora de interactuar con la aplicación. • Tamaño de la pantalla reducido.
Aplicaciones de escritorio	<ul style="list-style-type: none"> • Aprovechan mejor los recursos de los clientes. • Ofrecen la mejor respuesta a la interacción, una interfaz más potente y mejor experiencia de usuario. 	<ul style="list-style-type: none"> • Despliegue complejo. • Versionado complicado. • Poca interoperabilidad.

	<ul style="list-style-type: none"> • Proporcionan una interacción muy dinámica. • Soportan escenarios desconectados o con conexión limitada. 	
RIA (Rich Internet Applications)	<ul style="list-style-type: none"> • Proporcionan la misma potencia gráfica que las aplicaciones de escritorio. • Ofrecen soporte para visualizar contenido multimedia. • Despliegue y distribución simples. 	<ul style="list-style-type: none"> • Algo más pesadas que las aplicaciones web. • Aprovechan peor los recursos que las aplicaciones de escritorio. • Requieren tener instalado un plugin para funcionar.
Aplicaciones orientadas a servicios	<ul style="list-style-type: none"> • Proporcionan una interfaz muy desacoplada entre cliente y servidor. • Pueden ser consumidas por varias aplicaciones sin relación. • Son altamente interoperables 	<ul style="list-style-type: none"> • No tienen interfaz gráfica. • Necesitan conexión a internet.
Aplicaciones web	<ul style="list-style-type: none"> • Llegan a todo tipo de usuarios y tienen una interfaz de usuario estándar y multiplataforma. • Son fáciles de desplegar y de actualizar. 	<ul style="list-style-type: none"> • Dependen de la conectividad a red. • No pueden ofrecer interfaces de usuario complejas.

Una vez que tenemos decidido el tipo de aplicación que vamos a desarrollar, el siguiente paso es diseñar la arquitectura de la infraestructura, es decir, la topología de despliegue. La topología de despliegue depende directamente de las restricciones impuestas por el cliente, de las necesidades de seguridad del sistema y de la infraestructura disponible para desplegar el sistema. Definimos la arquitectura de la infraestructura en este punto, para tenerla en consideración a la hora de diseñar la arquitectura lógica de nuestra aplicación. Dado que las capas son más “maleables” que los niveles, encajaremos las distintas capas lógicas dentro de los niveles del sistema. Generalizando existen dos posibilidades, despliegue distribuido y despliegue no distribuido.

El despliegue no distribuido tiene la ventaja de ser más simple y más eficiente en las comunicaciones ya que las llamadas son locales. Por otra parte, de esta forma es más difícil permitir que varias aplicaciones utilicen la misma lógica de negocio al mismo tiempo. Además en este tipo de despliegue los recursos de la máquina son compartidos por todas las capas con lo que si una capa emplea más recursos que las otras existirá un cuello de botella.

El despliegue distribuido permite separar las capas lógicas en distintos niveles físicos. De esta forma el sistema puede aumentar su capacidad añadiendo servidores donde se necesiten y se puede balancear la carga para maximizar la eficiencia. Al mismo tiempo, al separar las capas en distintos niveles aprovechamos mejor los recursos, balanceando el número de equipos por nivel en función del consumo de las capas que se encuentran en él. El lado malo de las arquitecturas distribuidas es que la serialización de la información y su envío por la red tienen un coste no despreciable. Así mismo, los sistemas distribuidos son más complejos y más caros.

Tras decidir qué tipo de aplicación desarrollaremos y cuál será su topología de despliegue llega el momento de diseñar la arquitectura lógica de la aplicación. Para ello emplearemos en la medida de lo posible un conjunto de estilos arquitecturales conocidos. Los estilos arquitecturales son “patrones” de nivel de aplicación que definen un aspecto del sistema que estamos diseñando y representan una forma estándar de definir o implementar dicho aspecto. La diferencia entre un estilo arquitectural y un patrón de diseño es el nivel de abstracción, es decir, un patrón de diseño da una especificación concreta de cómo organizar las clases y la interacción entre objetos, mientras que un estilo arquitectural da una serie de indicaciones sobre qué se debe y qué no se debe hacer en un determinado aspecto del sistema. Los estilos arquitecturales se pueden agrupar según el aspecto que definen como muestra la siguiente tabla:

Tabla 2.- Aspectos estilos estructurales

Aspecto	Estilos arquitecturales
Comunicaciones	SOA, Message Bus, Tuberías y filtros.
Despliegue	Cliente/Servidor, 3-Niveles, N-Niveles.
Dominio	Modelo de dominio, Repositorio.
Interacción	Presentación separada.
Estructura	Componentes, Orientada a objetos, Arquitectura en capas.

Como se desprende de la tabla, en una aplicación usaremos varios estilos arquitecturales para dar forma al sistema. Por tanto, una aplicación será una combinación de muchos de ellos y de soluciones propias.

Ahora que ya hemos decidido el tipo de aplicación, la infraestructura física y la estructura lógica, tenemos una buena idea del sistema que construiremos. El siguiente paso lógico es comenzar con la implementación del diseño y para ello lo primero que tenemos que hacer es decidir qué tecnologías emplearemos. Los estilos arquitecturales que hemos usado para dar forma a nuestro sistema, el tipo de aplicación a desarrollar y la infraestructura física determinarán en gran medida estas tecnologías. Por ejemplo, para hacer una aplicación de escritorio escogeremos WPF o Silverlight 3, o si nuestra aplicación expone su funcionalidad como servicios web, usaremos WCF. En resumen las preguntas que tenemos que responder son:

- ¿Qué tecnologías ayudan a implementar los estilos arquitecturales seleccionados?
- ¿Qué tecnologías ayudan a implementar el tipo de aplicación seleccionada?
- ¿Qué tecnologías ayudan a cumplir con los requisitos no funcionales especificados?

Lo más importante es ser capaz al terminar este punto de hacer un esquema de la arquitectura que refleje su estructura y las principales restricciones y decisiones de diseño tomadas. Esto permitirá establecer un marco para el sistema y discutir la solución propuesta.

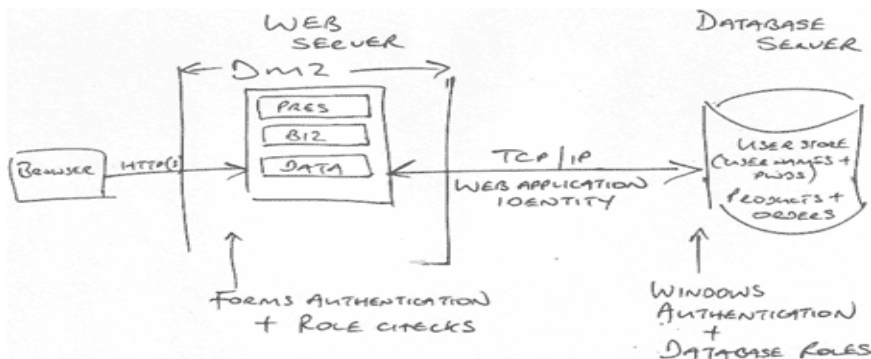


Figura 2.- Esquema de Arquitectura

4.- IDENTIFICAR LOS PRINCIPALES RIESGOS Y DEFINIR UNA SOLUCIÓN

El proceso de diseño de la arquitectura está dirigido por la funcionalidad, pero también por los riesgos a solventar. Cuanto antes minimicemos los riesgos, más probabilidades habrá de que tengamos éxito en nuestra arquitectura y al contrario.

La primera cuestión que surge es ¿Cómo identificar los riesgos de la arquitectura? Para responder a esta pregunta antes debemos tener claro qué requisitos no funcionales (o de calidad) tiene que tener nuestra aplicación. Esta información debería haber quedado definida tras la fase de inicio (*Inception*) y por lo tanto deberíamos contar con ella a la hora de realizar el diseño de la arquitectura.

Los requisitos no funcionales son aquellas propiedades que debe tener nuestra solución y que no son una funcionalidad, como por ejemplo: Alta disponibilidad, flexibilidad, interoperabilidad, mantenimiento, gestión operacional, rendimiento, fiabilidad, reusabilidad, escalabilidad, seguridad, robustez, capacidad de testeo y experiencia de usuario.

Es importante recalcar que normalmente nadie (un cliente normal) nos va a decir “la solución tiene que garantizar alta disponibilidad” sino que estos requisitos vendrán dados en el argot del cliente, por ejemplo “quiero que el sistema siga funcionando aunque falle algún componente”, y es trabajo del arquitecto traducirlos o mejor dicho, enmarcarlos dentro de alguna de las categorías.

Ahora que tenemos claros qué requisitos no funcionales (y por tanto riesgos) debemos tratar, podemos proceder a definir una solución para mitigar cada uno de ellos. Los requisitos no funcionales tienen impacto en como nuestra arquitectura tratará determinados “puntos clave” como son: la autenticación y la autorización, el cacheo de datos y el mantenimiento del estado, las comunicaciones, la composición, la concurrencia y las transacciones, la gestión de la configuración, el acoplamiento y la cohesión, el acceso a datos, la gestión de excepciones, el registro de eventos y la instrumentalización del sistema, la experiencia de usuario, la validación de la información y el flujo de los procesos de negocio del sistema. Estos puntos clave si tendrán una funcionalidad asociada en algunos casos o determinarán como se realiza la implementación de un aspecto del sistema en otros.

Como hemos dicho, **los requisitos no funcionales son propiedades** de la solución **y no funcionalidad**, pero influyen directamente en **los puntos clave** de la arquitectura que **sí son funcionalidad** del sistema. Podemos decir que los requisitos no funcionales son la especificación de las propiedades de nuestro sistema y los puntos clave la implementación.

Por lo general un requisito no funcional tiene asociados varios puntos clave que influyen positiva o negativamente en su consecución. Por tanto, lo que haremos será analizar cada uno de los requisitos no funcionales en base a los “puntos clave” a los que afecta y tomaremos las decisiones apropiadas. Es importante entender que la relación entre requisitos no funcionales y puntos clave es M:M, esto significa que se producirán situaciones en las que un punto clave afecte a varios requisitos no funcionales. Cuando el punto clave sea beneficioso para la consecución de todos los requisitos no funcionales no habrá problema, pero cuando influya positivamente en uno

y negativamente en otro es donde se tendrán que tomar decisiones que establezcan un compromiso entre ambos requisitos.

Cada uno de estos atributos se ve más a fondo en el capítulo dedicado a los aspectos horizontales/transversales de la arquitectura.

Como ya hemos dicho, en esta fase del proyecto de diseño mitigamos los principales riesgos creando planes para solventarlos y planes de contingencia para el caso de que no puedan ser solventados. Para diseñar un plan para un requisito de calidad nos basaremos en los puntos clave a los que afecta dicho requisito. El plan de un requisito consistirá en una serie de decisiones sobre los puntos clave. Siempre que se pueda es mejor expresar estas decisiones de forma gráfica, por ejemplo en el caso de la seguridad indicando en el diagrama de arquitectura física el tipo de seguridad que se utiliza en cada zona o en el caso del rendimiento donde se realiza el cacheo de datos.

5.- CREAR ARQUITECTURAS CANDIDATAS

Una vez realizados los pasos anteriores, tendremos una arquitectura candidata que podremos evaluar. Si tenemos varias arquitecturas candidatas, realizaremos la evaluación de cada una de ellas e implementaremos la arquitectura mejor valorada. Cualquier arquitectura candidata debería responder a las siguientes preguntas:

- ¿Qué funcionalidad implementa?
- ¿Qué riesgos mitiga?
- ¿Cumple las restricciones impuestas por el cliente?
- ¿Qué cuestiones deja en el aire?

Si no es así, es que la arquitectura todavía no está bien definida o no hemos concretado alguno de los pasos anteriores.

Para valorar una arquitectura candidata nos fijaremos en la funcionalidad que implementa y en los riesgos que mitiga. Como en todo proceso de validación tenemos que establecer métricas que nos permitan definir criterios de satisfacibilidad. Para ello existen multitud de sistemas, pero en general tendremos 2 tipos de métricas: Cualitativas y cuantitativas.

Las métricas cuantitativas evaluarán un aspecto de nuestra arquitectura candidata y nos darán un índice que compararemos con el resto de arquitecturas candidatas y con un posible mínimo requerido.

Las métricas cualitativas evaluarán si la arquitectura candidata cumple o no con un determinado requisito funcional o de calidad de servicio de la solución. Generalmente serán evaluadas de forma binaria como un sí o un no.

Con estas dos métricas podremos crear métricas combinadas, como por ejemplo métricas cuantitativas que solo serán evaluadas tras pasar el sesgo de una métrica cualitativa.

.....

Como ya hemos indicado existen multitud de sistemas para evaluar las arquitecturas software, pero todos ellos en mayor o menor medida se basan en este tipo de métricas. Los principales sistemas de evaluación de software son:

- *Software Architecture Analysis Method.*
- *Architecture Tradeoff Analysis Method.*
- *Active Design Review.*
- *Active Reviews of Intermediate Designs.*
- *Cost Benefit Analysis Method.*
- *Architecture Level Modifiability analysis.*
- *Family Architecture Assessment Method.*

Todas las decisiones sobre arquitectura deben plasmarse en una documentación que sea entendible por todos los integrantes del equipo de desarrollo así como por el resto de participantes del proyecto, incluidos los clientes. Hay muchas maneras de describir la arquitectura, como por ejemplo mediante ADL (*Architecture Description Language*), UML, Agile Modeling, IEEE 1471 o 4+1. Como dice el dicho popular, una imagen vale más que mil palabras, por ello nos decantamos por metodologías gráficas como 4+1. 4+1 describe una arquitectura software mediante 4 vistas distintas del sistema:

- **Vista lógica:** La vista lógica del sistema muestra la funcionalidad que el sistema proporciona a los usuarios finales. Emplea para ello diagramas de clases, de comunicación y de secuencia.
- **Vista del proceso:** La vista del proceso del sistema muestra cómo se comporta el sistema tiempo de ejecución, qué procesos hay activos y cómo se comunican. La vista del proceso resuelve cuestiones como la concurrencia, la escalabilidad, el rendimiento, y en general cualquier característica dinámica del sistema.
- **Vista física:** La vista física del sistema muestra cómo se distribuyen los distintos componentes software del sistema en los distintos nodos físicos de la infraestructura y cómo se comunican unos con otros. Emplea para ello los diagramas de despliegue.
- **Vista de desarrollo:** La vista lógica del sistema muestra el sistema desde el punto de vista de los programadores y se centra en el mantenimiento del software. Emplea para ello diagramas de componentes y de paquetes.

- **Escenarios:** La vista de escenarios completa la descripción de la arquitectura. Los escenarios describen secuencias de interacciones entre objetos y procesos y son usados para identificar los elementos arquitecturales y validar el diseño.

6.- ASPECTOS DE DOMAIN DRIVEN DESIGN



Figura 3.- Esquema de comunicación de Arquitectura

Hasta ahora hemos hablado del proceso de creación de la arquitectura, centrándonos en cómo elegir los casos de uso relevantes para la arquitectura, como decidir el tipo de aplicación que vamos a implementar y cómo afrontar los riesgos del proyecto dentro de la arquitectura. A continuación veremos aspectos claves que tenemos que tener en cuenta para conseguir una arquitectura que refleje nuestro dominio.

El objetivo de una arquitectura basada en Domain Driven Design es conseguir un modelo orientado a objetos que refleje el conocimiento de un dominio dado y que sea completamente independiente de cualquier concepto de comunicación, ya sea con elementos de infraestructura como de interfaz gráfica, etc. Buscamos construir un

.....

modelo a través del cual podamos resolver problemas expresados como la colaboración de un conjunto de objetos. Por ello, debemos tener claro qué:

- Todo proyecto software con lógica compleja y un dominio complicado debe disponer de un modelo que represente los aspectos del dominio que nos permiten implementar los casos de uso.
- El foco de atención en nuestra arquitectura debe estar en el modelo del dominio y en la lógica del mismo, ya que este es una representación del conocimiento del problema.
- El modelo que construimos tiene que estar íntimamente ligado con la solución que entregamos, y por tanto tener en cuenta las consideraciones de implementación.
- Los modelos de dominio representan conocimiento acumulado, y dado que el conocimiento se adquiere de forma gradual e incremental, el proceso de creación de un modelo que represente profundamente los conceptos de un dominio debe ser iterativo.

6.1.- El lenguaje ubicuo

Uno de los principales motivos de fracaso de los proyectos software es la ruptura de la comunicación entre los expertos del dominio y los desarrolladores encargados de construir un sistema. La falta de un lenguaje común para comunicarse entre expertos del dominio y desarrolladores, así como entre los propios desarrolladores, genera problemas como la diferente interpretación de conceptos o la múltiple representación de un mismo concepto. Es esto lo que acaba derivando en implementaciones desconectadas del dominio con el que trabajan o en el que intentan resolver problemas. Las implementaciones desconectadas del dominio con el que trabajan presentan dos síntomas claramente observables:

- El sistema no resuelve correctamente un problema.
- El sistema no resuelve el problema adecuado.

Es vital tener claro, que cualquier modelo que construyamos debe estar profundamente representado en la implementación que hagamos del sistema, es decir, en lugar de disponer de un modelo de análisis y un modelo de implementación, debemos disponer de un único modelo, el modelo de dominio.

Cualquier modelo que construyamos debe representar de forma explícita los principales conceptos del dominio de conocimiento con el que trabaja nuestro sistema. Debemos fomentar la construcción de un lenguaje de uso común tanto entre expertos del dominio y desarrolladores, como entre los propios desarrolladores, que contenga

los principales conceptos del dominio de conocimiento con el que trabaja el sistema, y que sea el lenguaje usado para expresar cómo se resuelven los distintos problemas objetivo de nuestro sistema. Al utilizar un lenguaje común para comunicarnos, fomentamos la transferencia de conocimiento de los expertos del dominio a los desarrolladores, lo que permite que estos implementen un modelo de dominio mucho más profundo. Los buenos modelos se consiguen cuando los desarrolladores tienen un profundo conocimiento del dominio que están modelando, y este conocimiento solo se adquiere con el tiempo y a través de la comunicación con los expertos del dominio. Razón por la cual es imprescindible el uso de un lenguaje común.

6.2.- Prácticas que ayudan a conseguir un buen modelo de dominio.

El punto clave para un proyecto exitoso es la transferencia efectiva de conocimiento desde los expertos del dominio a los desarrolladores del sistema. Para facilitar esta transferencia de conocimiento podemos emplear varias técnicas de desarrollo conocidas.

6.2.1.- Behavior Driven Development (BDD)

BDD es una práctica aplicable dentro de cualquier metodología que consiste en la descripción de los requisitos como un conjunto de pruebas ejecutables de forma automática. BDD ayuda a la transferencia del conocimiento al provocar que los principales conceptos del dominio presentes en los requisitos, pasen directamente al código, destacando su importancia dentro del dominio y creando un contexto o base para la construcción del modelo.

6.2.2.- Test Driven Development (TDD)

TDD es una práctica aplicable dentro de cualquier metodología que consiste en desarrollar un conjunto de pruebas que sirven como especificación y justificación de la necesidad de crear un componente para implementar una determinada funcionalidad. Estas pruebas permiten definir la forma del propio componente e indagan en las relaciones de este con otros componentes del dominio, lo que fomenta el desarrollo del modelo.

Arquitectura Marco N-Capas

1.- ARQUITECTURA DE APLICACIONES EN N-CAPAS



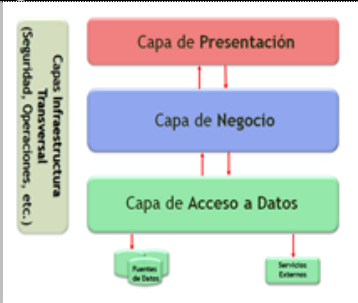
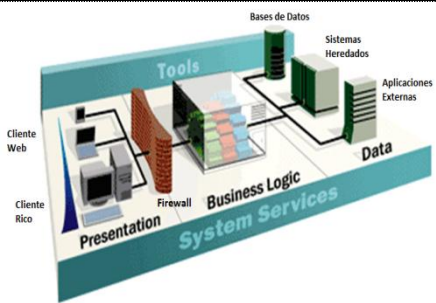
1.1.- Capas vs. Niveles (Layers vs. Tiers)

Es importante distinguir los conceptos de “Capas” (*Layers*) y “Niveles” (*Tiers*), pues es bastante común que se confundan y o se denominen de forma incorrecta.

Las Capas (*Layers*) se ocupan de la división lógica de componentes y funcionalidad, y no tienen en cuenta la localización física de componentes en diferentes servidores o en diferentes lugares. Por el contrario, los Niveles (*Tiers*) se ocupan de la distribución física de componentes y funcionalidad en servidores separados, teniendo en cuenta topología de redes y localizaciones remotas. Aunque tanto las Capas (*Layers*) como los Niveles (*Tiers*) usan conjuntos similares de nombres (presentación, servicios, negocio y datos), es importante no confundirlos y recordar que solo los Niveles (*Tiers*) implican una separación física. Se suele utilizar el término “*Tier*” refiriéndonos a patrones de distribución física como “2 *Tier*”, “3-*Tier*” y “N-*Tier*”.

A continuación mostramos un esquema **3-Tier** y un esquema **N-Layer** donde se pueden observar las diferencias comentadas (lógica vs. Situación física):

Tabla 1.- N-Tier vs. N-Layer

Arquitectura tradicional N-Capas (Lógica)	Arquitectura 3-Tier (Física)
	
Figura 1.- Arquitectura tradicional N-Layer (Lógica)	Figura 2.- Arquitectura 3-Tier (Física)

Por último, destacar que todas las aplicaciones con cierta complejidad, deberían implementar una arquitectura lógica de tipo N-Capas, pues proporciona una estructuración lógica correcta; sin embargo, no todas las aplicaciones tienen por qué implementarse en modo *N-Tier*, puesto que hay aplicaciones que no requieren de una separación física de sus niveles (*Tiers*), como pueden ser muchas aplicaciones web.



1.2.- Capas

Contexto

Se quiere diseñar una aplicación empresarial compleja compuesta por un número considerable de componentes de diferentes niveles de abstracción.

Problema

Cómo estructurar una aplicación para soportar requerimientos complejos operacionales y disponer de una buena mantenibilidad, reusabilidad, escalabilidad, robustez y seguridad.

Aspectos relacionados

Al estructurar una aplicación, se deben reconciliar las siguientes ‘fuerzas’ dentro del contexto del entorno de la aplicación:

- Localizar los cambios de un tipo en una parte de la solución minimiza el impacto en otras partes, reduce el trabajo requerido en arreglar defectos, facilita el mantenimiento de la aplicación y mejora la flexibilidad general de la aplicación.
- Separación de responsabilidades entre componentes (por ejemplo, separar la interfaz de usuario de la lógica de negocio, y la lógica de negocio del acceso a la base de datos) aumenta la flexibilidad, la mantenibilidad y la escalabilidad.
- Ciertos componentes deben ser reutilizables entre diferentes módulos de una aplicación o incluso entre diferentes aplicaciones.
- Equipos diferentes deben poder trabajar en partes de la solución con mínimas dependencias entre los diferentes equipos de desarrollo y para ello, deben desarrollar contra interfaces bien definidas.
- Los componentes individuales deben ser cohesivos
- Los componentes no relacionados directamente deben estar débilmente acoplados
- Los diferentes componentes de una solución deben poder ser desplegados de una forma independiente, e incluso mantenidos y actualizados en diferentes momentos.
- Para asegurar estabilidad y calidad, cada capa debe contener sus propias pruebas unitarias.

Las capas son agrupaciones horizontales lógicas de componentes de software que forman la aplicación o el servicio. Nos ayudan a diferenciar entre los diferentes tipos de tareas a ser realizadas por los componentes, ofreciendo un diseño que maximiza la reutilización y, especialmente, la mantenibilidad. En definitiva, se trata de aplicar el principio de ‘*Separación de Responsabilidades*’ (SoC - *Separation of Concerns principle*) dentro de una Arquitectura.

Cada capa lógica de primer nivel puede tener un número concreto de componentes agrupados en sub-capas. Dichas sub-capas realizan a su vez un tipo específico de tareas. Al identificar tipos genéricos de componentes que existen en la mayoría de las soluciones, podemos construir un patrón o mapa de una aplicación o servicio y usar dicho mapa como modelo de nuestro diseño.

El dividir una aplicación en capas separadas que desempeñan diferentes roles y funcionalidades, nos ayuda a mejorar el mantenimiento del código; nos permite también diferentes tipos de despliegue y, sobre todo, nos proporciona una clara

delimitación y situación de dónde debe estar cada tipo de componente funcional e incluso cada tipo de tecnología.

Diseño básico de capas

Se deben separar los componentes de la solución en capas. Los componentes de cada capa deben ser cohesivos y tener aproximadamente el mismo nivel de abstracción. Cada capa de primer nivel debe de estar débilmente acoplada con el resto de capas de primer nivel. El proceso es como sigue:

Comenzar en el nivel más bajo de abstracción, por ejemplo ‘Capa 1’. Esta capa es la base del sistema. Se continúa esta escalera abstracta con otras capas (Capa J, Capa J-1) hasta el último nivel (Capa-N):



Figura 3.- Diseño básico de capas

La clave de una aplicación en N-Capas está en la gestión de dependencias. En una arquitectura N-Capas tradicional, los componentes de una capa pueden interactuar solo con componentes de la misma capa o bien con otros componentes de capas inferiores. Esto ayuda a reducir las dependencias entre componentes de diferentes niveles. Normalmente hay **dos aproximaciones al diseño en capas: Estricto y Laxo**.

Un **‘diseño en Capas estricto’** limita a los componentes de una capa a comunicarse solo con los componentes de su misma capa o con la capa inmediatamente inferior. Por ejemplo, en la figura anterior, si utilizamos el sistema estricto, la capa J solo podría interactuar con los componentes de la capa J-1, la capa J-1 solo con los componentes de la capa J-2, y así sucesivamente.

Un **‘diseño en Capas laxo’** permite que los componentes de una capa interactúen con cualquier otra capa de nivel inferior. Por ejemplo, en la figura anterior, si utilizamos esta aproximación, la capa J podría interactuar con la capa J-1, J-2 y J-3.

El uso de la aproximación laxa puede mejorar el rendimiento porque el sistema no tiene que realizar redundancia de llamadas de unas capas a otras. Por el contrario, el uso de la aproximación laxa no proporciona el mismo nivel de aislamiento entre las

diferentes capas y hace más difícil el sustituir una capa de más bajo nivel sin afectar a muchas más capas de nivel superior (y no solo a una).

En soluciones grandes que involucran a muchos componentes de software, es habitual tener un gran número de componentes en el mismo nivel de abstracción (capas) pero que sin embargo no son cohesivos. En esos casos, cada capa debería descomponerse en dos o más subsistemas cohesivos, llamados también Módulos (parte de un módulo vertical en cada capa horizontal). El concepto de módulo lo explicamos en más detalle posteriormente dentro de la Arquitectura marco propuesta, en este mismo capítulo.

El siguiente diagrama UML representa capas compuestas a su vez por múltiples subsistemas:

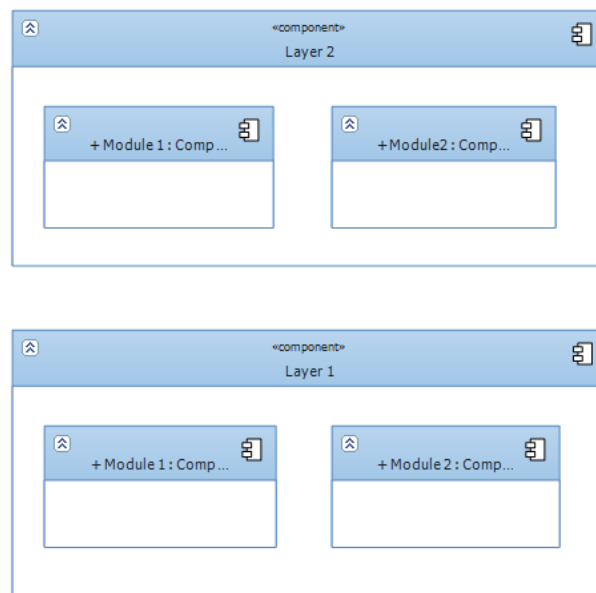


Figura 4.- Múltiples subsistemas en cada capa

Consideraciones relativas a Pruebas

Una aplicación en N-Capas mejora considerablemente la capacidad de implementar pruebas de una forma apropiada:

- Debido a que cada capa interactúa con otras capas solo mediante interfaces bien definidos, es fácil añadir implementaciones alternativas a cada capa (*Mock* y *Stubs*). Esto permite realizar pruebas unitarias de una capa incluso cuando las capas de las que depende no están finalizadas o, incluso, porque se quiera poder ejecutar mucho más rápido un conjunto muy grande de pruebas unitarias que al acceder a las capas de las que depende se ejecutan mucho más

lentamente. Esta capacidad se ve muy mejorada si se hace uso de clases base (Patrón ‘*Layered Supertype*’) e interfaces (Patrón ‘*Abstract Interface*’), porque limitan aún más las dependencias entre las capas., Es especialmente importante el uso de interfaces pues nos dará la posibilidad de utilizar técnicas más avanzadas de desacoplamiento, que exponemos más adelante en esta guía.

- Es más fácil realizar pruebas sobre componentes individuales porque las dependencias entre ellos están limitadas de forma que los componentes de capas de alto nivel solo pueden interaccionar con los de niveles inferiores. Esto ayuda a aislar componentes individuales para poder probarlos adecuadamente y nos facilita el poder cambiar unos componentes de capas inferiores por otros diferentes con un impacto muy pequeño en la aplicación (siempre y cuando cumplan los mismos interfaces).

Beneficios de uso de Capas

- El mantenimiento de mejoras en una solución será mucho más fácil porque las funciones están localizadas. Además las capas deben estar débilmente acopladas entre ellas y con alta cohesión internamente, lo cual posibilita variar de una forma sencilla diferentes implementaciones/combinaciones de capas.
- Otras soluciones deberían poder reutilizar funcionalidad expuesta por las diferentes capas, especialmente si se han diseñado para ello.
- Los desarrollos distribuidos son mucho más sencillos de implementar si el trabajo se ha distribuido previamente en diferentes capas lógicas.
- La distribución de capas (*layers*) en diferentes niveles físicos (*tiers*) puede, en algunos casos, mejorar la escalabilidad. Aunque este punto hay que evaluarlo con cuidado, pues puede impactar negativamente en el rendimiento.

Referencias

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerland, Peter; and Stal, Michael. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley & Sons, 1996.

Fowler, Martin. Patterns of Application Architecture. Addison-Wesley, 2003.

Gamma, Eric; Helm, Richard; Johnson, Ralph; and Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.



I.3.- Principios Base de Diseño a seguir

A la hora de diseñar un sistema, es importante tener presente una serie de principios de diseño fundamentales que nos ayudarán a crear una arquitectura que se ajuste a prácticas demostradas, que minimicen los costes de mantenimiento y maximicen la usabilidad y la extensibilidad. Estos principios clave seleccionados y muy reconocidos por la industria del *software*, son:



I.3.1.- Principios de Diseño ‘SOLID’

El acrónimo SOLID deriva de las siguientes frases/principios en inglés:

Principios ‘SOLID’ en Diseño

- ☐ **S**ingle Responsibility Principle
- ☐ **O**pen Close Principle
- ☐ **L**iskov Substitution Principle
- ☐ **I**nterface Segregation Principle
- ☐ **D**ependency Inversion Principle

De una forma resumida, los principios de diseño SOLID son los siguientes:

- **Principio de Única Responsabilidad ('Single Responsibility Principle'):**
Una clase debe tener una única responsabilidad o característica. Dicho de otra manera, una clase debe de tener una única razón por la que está justificado realizar cambios sobre su código fuente. Una consecuencia de este principio es que, de forma general, las clases deberían tener pocas dependencias con otras clases/tipos.

- **Principio Abierto Cerrado (*‘Open Closed Principle’*):** Una clase debe estar abierta para la extensión y cerrada para la modificación. Es decir, el comportamiento de una clase debe poder ser extendido sin necesidad de realizar modificaciones sobre el código de la misma.
- **Principio de Sustitución de Liskov (*‘Liskov Substitution Principle’*):** Los subtipos deben poder ser sustituibles por sus tipos base (interfaz o clase base). Este hecho se deriva de que el comportamiento de un programa que trabaja con abstracciones (interfaces o clases base) no debe cambiar porque se sustituya una implementación concreta por otra. Los programas deben hacer referencia a las abstracciones, y no a las implementaciones. Veremos posteriormente que este principio va a estar muy relacionado con la *Inyección de Dependencias* y la sustitución de unas clases por otras siempre que cumplan el mismo interfaz.
- **Principio de Segregación de Interfaces (*‘Interface Segregation Principle’*):** Los implementadores de Interfaces de clases no deben estar obligados a implementar métodos que no se usan. Es decir, los interfaces de clases deben ser específicos dependiendo de quién los consume y por lo tanto, tienen que estar granularizados/segregados en diferentes interfaces no debiendo crear nunca grandes interfaces. Las clases deben exponer interfaces separados para diferentes clientes/consumidores que difieren en los requerimientos de interfaces.
- **Principio de Inversión de Dependencias (*‘Dependency Inversion Principle’*):** Las abstracciones no deben depender de los detalles – Los detalles deben depender de las abstracciones. Las dependencias directas entre clases deben ser reemplazadas por abstracciones (interfaces) para permitir diseños top-down sin requerir primero el diseño de los niveles inferiores.



1.3.2.- Otros Principios clave de Diseño

- **El diseño de componentes debe ser altamente cohesivo:** no sobrecargar los componentes añadiendo funcionalidad mezclada o no relacionada. Por ejemplo, evitar mezclar lógica de acceso a datos con lógica de negocio perteneciente al Modelo del Dominio. Cuando la funcionalidad es cohesiva, entonces podemos crear ensamblados/*assemblies* que contengan más de un componente y situar los componentes en las capas apropiadas de la aplicación. Este principio está por lo tanto muy relacionado con el patrón ‘N-Capas’ y con el principio de *‘Single Responsibility Principle’*.

- **Mantener el código transversal abstraído de la lógica específica de la aplicación:** el código transversal se refiere a código de aspectos horizontales, cosas como la seguridad, gestión de operaciones, logging, instrumentalización, etc. La mezcla de este tipo de código con la implementación específica de la aplicación puede dar lugar a diseños que sean en el futuro muy difíciles de extender y mantener. Relacionado con este principio está AOP (*Aspect Oriented Programming*).
- **Separación de Preocupaciones/Responsabilidades** (*'Separation of Concerns'*): dividir la aplicación en distintas partes minimizando las funcionalidades superpuestas entre dichas partes. El factor fundamental es minimizar los puntos de interacción para conseguir una alta cohesión y un bajo acoplamiento. Sin embargo, separar la funcionalidad en las fronteras equivocadas, puede resultar en un alto grado de acoplamiento y complejidad entre las características del sistema.
- **No repetirse (DRY):** se debe especificar 'la intención' en un único sitio en el sistema. Por ejemplo, en términos del diseño de una aplicación, una funcionalidad específica se debe implementar en un único componente; esta misma funcionalidad no debe estar implementada en otros componentes.
- **Minimizar el diseño de arriba abajo** (*Upfront design*): diseñar solamente lo que es necesario, no realizar 'sobre-ingenierías' y evitar el efecto YAGNI (En inglés-slang: *You Ain't Gonna Need It*).



1.4.- Orientación a tendencias de Arquitectura DDD (*Domain Driven Design*)

El objetivo de esta arquitectura marco es proporcionar una base consolidada y guías de arquitectura para un tipo concreto de aplicaciones: '**Aplicaciones empresariales complejas**'. Este tipo de aplicaciones se caracterizan por tener una vida relativamente larga y un volumen de cambios evolutivos considerable. Por lo tanto, en estas aplicaciones es muy importante todo lo relativo al mantenimiento de la aplicación, la facilidad de actualización, o la sustitución de tecnologías y *frameworks/ORMs* (*Object-relational mapping*) por otras versiones más modernas o incluso por otros diferentes, etc. El objetivo es que todo esto se pueda realizar con el menor impacto posible sobre el resto de la aplicación. En definitiva, que los cambios de tecnologías de infraestructura de una aplicación no afecten a capas de alto nivel de la aplicación, especialmente, que afecten lo mínimo posible a la capa del 'Dominio de la aplicación'.

En las aplicaciones complejas, el comportamiento de las reglas de negocio (lógica del Dominio) está sujeto a muchos cambios y es muy importante poder modificar, construir y realizar pruebas sobre dichas capas de lógica del dominio de una forma fácil e independiente. Debido a esto, un objetivo importante es tener el mínimo acoplamiento entre el Modelo del Dominio (lógica y reglas de negocio) y el resto de capas del sistema (Capas de presentación, Capas de Infraestructura, persistencia de datos, etc.).

Debido a las premisas anteriores, las tendencias de arquitectura de aplicaciones que están más orientadas a conseguir este desacoplamiento entre capas, especialmente la independencia y foco preponderante sobre la capa del Modelo de Dominio, son precisamente las Arquitecturas N-Capas Orientadas al Dominio, como parte de DDD (*Domain Driven Design*).

DDD (Domain Driven Design) es, sin embargo, mucho más que simplemente una Arquitectura propuesta; es también una forma de afrontar los proyectos, una forma de trabajar por parte del equipo de desarrollo, la importancia de identificar un ‘Lenguaje Ubicuo’ proyectado a partir del conocimiento de los expertos en el dominio (expertos en el negocio), etc. Sin embargo, todo esto queda fuera de la presente guía puesto que se quiere limitar a una Arquitectura lógica y tecnológica, no a la forma de afrontar un proyecto de desarrollo o forma de trabajar de un equipo de desarrollo. Todo esto puede consultarse en libros e información relacionada con DDD.

Razones por las que no se debe orientar a Arquitecturas N-Capas Orientadas al Dominio

Debido a las premisas anteriores, se desprende que si la aplicación a realizar es relativamente sencilla y, sobre todo, si las reglas de negocio a automatizar en la aplicación cambiarán muy poco y no se prevén necesidades de cambios de tecnología de infraestructura durante la vida de dicha aplicación, entonces, probablemente la solución no debería seguir el tipo de arquitectura presentado en esta guía, y más bien se debería seleccionar un tipo de desarrollo/tecnología RAD (*Rapid Application Development*), como puede ser ‘*WCF RIA Services*’. Es decir, tecnologías de rápida implementación a ser utilizadas para construir aplicaciones sencillas donde el desacoplamiento entre todos sus componentes y capas no es especialmente relevante, pero sí lo es facilidad y productividad en el desarrollo y el ‘*time to market*’. De forma generalista se suele decir que son aplicaciones centradas en datos (*Data Driven*) y no tanto en un modelo de dominio (*Domain Driven Design*).

Razones por las que se si se debe orientar a Arquitectura N-Capas Orientada al Dominio

Es realmente volver hacer hincapié sobre lo mismo, pero es muy importante dejar este aspecto claro.

Así pues, las razones por las que es importante hacer uso de una ‘Arquitectura N-Capas Orientada al Dominio’ es especialmente en los casos donde el comportamiento

.....

del negocio a automatizar (lógica del dominio) está sujeto a muchos cambios y evoluciones. En este caso específico, disponer de un ‘Modelo de Dominio’ disminuirá el coste total de dichos cambios, y a medio plazo el TCO (Coste Total de la Propiedad) será mucho menor que si la aplicación hubiera sido desarrollada de una forma más acoplada, porque los cambios no tendrán tanto impacto. En definitiva, el tener todo el comportamiento del negocio que puede estar cambiando encapsulado en una única área de nuestro software, disminuye drásticamente la cantidad de tiempo que se necesita para realizar un cambio. Porque este cambio se realizará en un solo sitio y podrá ser convenientemente probado de forma aislada, aunque esto por supuesto dependerá de cómo se haya desarrollado. El poder aislar tanto como sea posible dicho código del Modelo del Dominio disminuye las posibilidades de tener que realizar cambios en otras áreas de la aplicación (lo cual siempre puede afectar con nuevos problemas, regresiones, etc.). Esto es de vital importancia si se desea reducir y mejorar los ciclos de estabilización y puesta en producción de las soluciones.

Escenarios donde utilizar el Modelo de Dominio

Las reglas de negocio que indican cuándo se permiten ciertas acciones son precisamente buenas candidatas a ser implementadas en el modelo de dominio.

Por ejemplo, en un sistema comercial, una regla que especifica que un cliente no puede tener pendiente de pago más de 2.000€, probablemente debería pertenecer al modelo de dominio. Hay que tener en cuenta que reglas como la anterior involucran a diferentes entidades y tienen que evaluarse en diferentes casos de uso.

Así pues, en un modelo de dominio tendremos muchas de estas reglas de negocio, incluyendo casos donde unas reglas sustituyen a otras. Por ejemplo, sobre la regla anterior, si el cliente es una cuenta estratégica o con un volumen de negocio muy grande dicha cantidad podría ser muy superior, etc.

En definitiva, la importancia que tengan en una aplicación las reglas de negocio y los casos de uso es precisamente la razón por la que orientar la arquitectura hacia el Dominio y no simplemente definir entidades, relaciones entre ellas y una aplicación orientada a datos.

Finalmente, para persistir la información y convertir colecciones de objetos en memoria (grafos de objetos/entidades) a una base de datos relacional, podemos hacer uso de alguna tecnología de persistencia de datos de tipo ORM (*Object-Relational Mapping*), como *NHibernate* o *Entity Framework*. Sin embargo, es muy importante que queden muy diferenciadas y separadas estas tecnologías concretas de persistencia de datos (tecnologías de infraestructura) del comportamiento de negocio de la aplicación, que es responsabilidad del Modelo del Dominio. Para esto, se necesita una arquitectura en Capas (*N-Layer*) que esté integrada de una forma desacoplada, como veremos posteriormente.



1.5.- DDDD (*Distributed Domain Driven Design*)

¿Cuatro ‘D’?. Bueno, sí, está claro que DDDD es una evolución/extensión de DDD donde se añaden aspectos de sistemas distribuidos. Eric Evans, en su libro de DDD obvia casi por completo los sistemas y tecnologías distribuidas, (Servicios Web, etc.) porque se centra mayoritariamente en el Dominio. Sin embargo, los sistemas distribuidos y Servicios remotos son algo que necesitamos en la mayoría de los escenarios.

Realmente, la presente propuesta de Arquitectura N-Layer, está basada en DDDD, porque tenemos en cuenta desde el principio a la capa de Servicios Distribuidos, e incluso lo mapeamos luego a implementación con tecnología Microsoft.

En definitiva, esta cuarta D añadida a DDD nos acerca a escenarios distribuidos, gran escalabilidad e incluso escenarios que normalmente se acercarán a ‘Cloud-Computing’ por su afinidad.



2.- ARQUITECTURA MARCO N-CAPAS CON ORIENTACIÓN AL DOMINIO

Queremos recalcar que hablamos de arquitectura con ‘Orientación al Dominio’, no hablamos de todo lo que cubre DDD (*Domain Driven Design*). Para hablar de DDD deberíamos centrarnos realmente no solo en la arquitectura (objetivo de esta guía), sino más bien en el proceso de diseño, en la forma de trabajar de los equipos de desarrollo, el ‘lenguaje ubicuo’, etc. Esos aspectos de DDD los tocaremos en la presente guía, pero de forma leve. **El objetivo de esta guía es centrarnos exclusivamente en una *Arquitectura N-Layer* que encaje con DDD, y como “mapearlo” posteriormente a las tecnologías Microsoft. No pretendemos exponer y explicar DDD, para esto último ya existen magníficos libros al respecto.**

Esta sección define de forma global la arquitectura marco en N-Capas así como ciertos patrones y técnicas a tener en cuenta para la integración de dichas capas.



2.1.- Capas de Presentación, Aplicación, Dominio e Infraestructura

En el nivel más alto y abstracto, la vista de arquitectura lógica de un sistema puede considerarse como un conjunto de servicios relacionados agrupados en diversas capas, similar al siguiente esquema (siguiendo las tendencias de Arquitectura DDD):

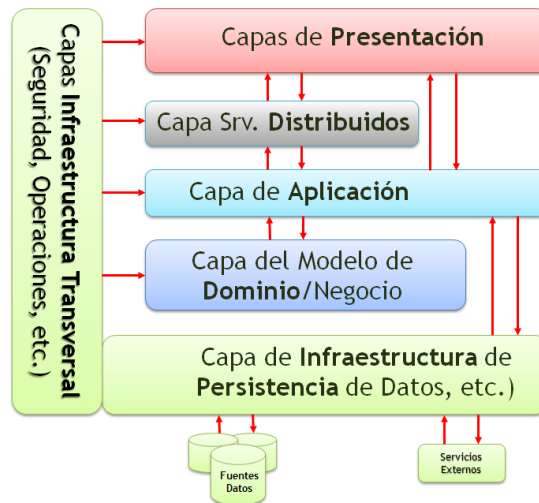


Figura 5.- Vista de arquitectura lógica simplificada de un sistema N-Capas DDD

En Arquitecturas ‘Orientadas al Dominio’ es crucial la clara delimitación y separación de la capa del Dominio del resto de capas. Es realmente un pre-requisito para DDD. *“Todo debe girar alrededor del Dominio”*.

Así pues, se debe particionar una aplicación compleja en capas. Desarrollar un diseño dentro de cada capa que sea cohesivo, pero delimitando claramente las diferentes capas entre ellas, aplicando patrones estándar de Arquitectura para que dichas dependencias sean en muchas ocasiones basadas en abstracciones y no referenciando una capa directamente a la otra. Concentrar todo el código relacionado con el modelo del dominio en una capa y aislarlo del resto de código de otras capas (Presentación, Aplicación, Infraestructura y Persistencia, etc.). Los objetos del Dominio, al estar libres de tener que mostrarse ellos mismos, persistirse/guardarse, gestionar tareas de aplicación, etc. pueden entonces centrarse exclusivamente en expresar el modelo de dominio. Esto permite que un modelo de dominio pueda evolucionar y llegar a ser lo suficientemente rico y claro para representar el conocimiento de negocio esencial y ponerlo realmente en ejecución dentro de la aplicación.

El separar la capa de dominio del resto de capas permite un diseño mucho más limpio de cada capa. Las capas aisladas son mucho menos costosas de mantener porque tienden a evolucionar a diferentes ritmos y responder a diferentes necesidades. Por ejemplo, las capas de infraestructura evolucionarán cuando evolucionen las tecnologías sobre las que están basadas. Por el contrario, la capa del Dominio evolucionará solo cuando se quieran realizar cambios en la lógica de negocio del Dominio concreto.

Adicionalmente, *la separación de capas ayuda en el despliegue de un sistema distribuido, permitiendo que diferentes capas sean situadas de forma flexible en diferentes servidores o clientes, de manera que se minimice el exceso de comunicación y se mejore el rendimiento (Cita de M. Fowler).*

La integración y desacoplamiento entre las diferentes capas de alto nivel es algo fundamental. Cada capa de la aplicación contendrá una serie de componentes que implementan la funcionalidad de dicha capa. Estos componentes deben ser cohesivos internamente (dentro de la misma capa de primer nivel), pero algunas capas (como las capas de Infraestructura/Tecnología) deben estar débilmente acopladas con el resto de capas para poder potenciar las pruebas unitarias, *mocking*, la reutilización y finalmente que impacte menos al mantenimiento. Este desacoplamiento entre las capas principales se explica en más detalle posteriormente, tanto su diseño como su implementación.



2.2.- Arquitectura marco N-Capas con Orientación al Dominio

El objetivo de esta arquitectura marco es estructurar de una forma limpia y clara la complejidad de una aplicación empresarial basada en las diferentes capas de la arquitectura, siguiendo el patrón *N-Layered* y las tendencias de arquitecturas en DDD. El patrón *N-Layered* distingue diferentes capas y sub-capas internas en una aplicación, delimitando la situación de los diferentes componentes por su tipología.

Por supuesto, esta arquitectura concreta N-Layer es personalizable según las necesidades de cada proyecto y/o preferencias de Arquitectura. Simplemente proponemos una Arquitectura marco a seguir que sirva como punto base a ser modificada o adaptada por arquitectos según sus necesidades y requisitos.

En concreto, las capas y sub-capas propuestas para aplicaciones '*N-Layered con Orientación al Dominio*' son:

Arquitectura N-Capas con Orientación al Dominio

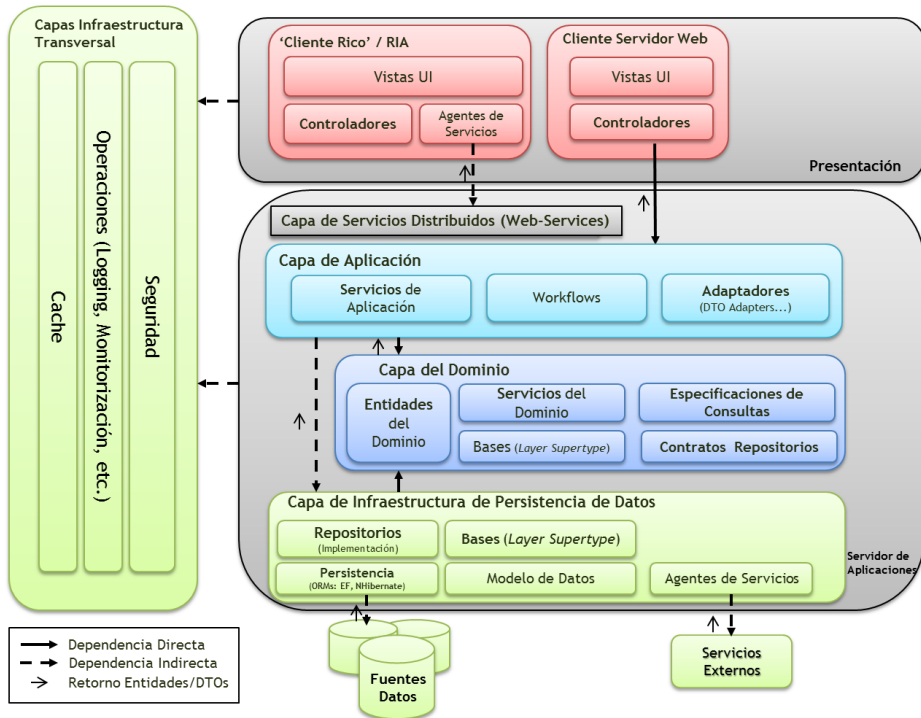


Figura 6.- Arquitectura N-Capas con Orientación al Dominio

- **Capa de Presentación**
 - o Subcapas de Componentes Visuales (Vistas)
 - o Subcapas de Proceso de Interfaz de Usuario (Controladores y similares)
- **Capa de Servicios Distribuidos (Servicios-Web)**
 - o Servicios-Web publicando las Capas de Aplicación y Dominio
- **Capa de Aplicación**
 - o Servicios de Aplicación (Tareas y coordinadores de casos de uso)
 - o Adaptadores (Conversores de formatos, etc.)
 - o Subcapa de *Workflows* (Opcional)
 - o Clases base de Capa Aplicación (Patrón Layer-Supertype)

- **Capa del Modelo de Dominio**
 - Entidades del Dominio
 - Servicios del Dominio
 - Especificaciones de Consultas (Opcional)
 - Contratos/Interfaces de *Repositorios*
 - Clases base del Dominio (Patrón *Layer-Supertype*)
- **Capa de Infraestructura de Acceso a Datos**
 - Implementación de Repositorios'
 - Modelo lógico de Datos
 - Clases Base (Patrón *Layer-Supertype*)
 - Infraestructura tecnología ORM
 - Agentes de Servicios externos
- **Componentes/Aspectos Horizontales de la Arquitectura**
 - Aspectos horizontales de Seguridad, Gestión de operaciones, Monitorización, Correo Electrónico automatizado, etc.

Todas estas capas se explican en el presente capítulo de forma breve y posteriormente dedicamos un capítulo a cada una de ellas; sin embargo, antes de ello, es interesante conocer desde un punto de vista de alto nivel cómo es la interacción entre dichas capas y por qué las hemos dividido así.

Una de las fuentes y precursores principales de DDD, es *Eric Evans*, el cual en su libro “*Domain Driven Design - Tackling Complexity in the Heart of Software*” expone y explica el siguiente diagrama de alto nivel con su propuesta de Arquitectura N-Layer:

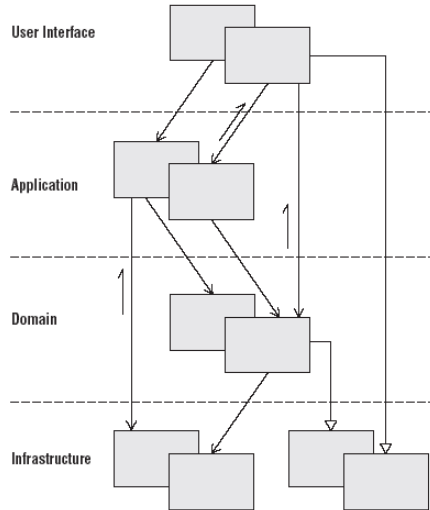


Figura 7.- Diagrama de Arquitectura N-Layer DDD

Es importante resaltar que en algunos casos el acceso a las otras capas es directo. Es decir, no tiene por qué haber un camino único obligatorio pasando de una capa a otra, aunque dependerá de los casos. Para que queden claros dichos casos a continuación mostramos el anterior diagrama de Eric-Evans, pero modificado y un poco más detallado, de forma que se relaciona con las sub-capas y elementos de más bajo nivel que proponemos en nuestra Arquitectura:

Interacción en Arquitectura DDD

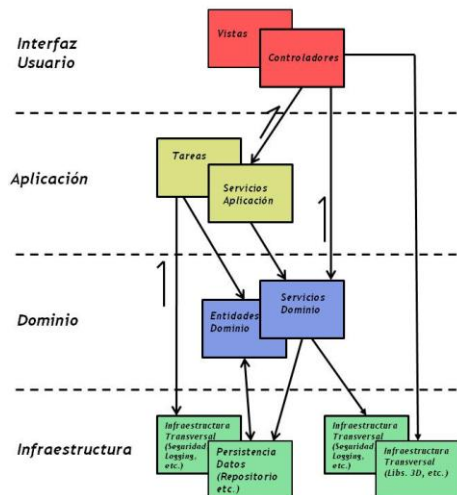


Figura 8.- Interacción en Arquitectura DDD

Primeramente, podemos observar que la **Capa de Infraestructura** que presenta una arquitectura con tendencia DDD, es algo muy amplio y para muchos contextos muy diferentes (Contextos de Servidor y de Cliente). La Capa de infraestructura contendrá todo lo ligado a tecnología/infraestructura. Ahí se incluyen conceptos fundamentales como Persistencia de Datos (Repositorios, etc.), pasando por aspectos transversales como Seguridad, *Logging*, Operaciones, etc. e incluso podría llegar a incluirse librerías específicas de capacidades gráficas para UX (librerías 3D, librerías de controles específicos para una tecnología concreta de presentación, etc.). Debido a estas grandes diferencias de contexto y a la importancia del acceso a datos, en nuestra arquitectura propuesta hemos separado explícitamente la Capa de Infraestructura de '**Persistencia de Datos**' del resto de capas de '**Infraestructura Transversal**', que pueden ser utilizadas de forma horizontal/transversal por cualquier capa.

El otro aspecto interesante que adelantábamos anteriormente, es el hecho de que el acceso a algunas capas no es con un único camino ordenado por diferentes capas. Concretamente podremos acceder directamente a las capas de Aplicación, de Dominio y de Infraestructura Transversal siempre que lo necesitemos. Por ejemplo, podríamos acceder directamente desde una Capa de Presentación Web (no necesita interfaces remotos de tipo Servicio-Web) a las capas inferiores que necesitemos (Aplicación, Dominio, y algunos aspectos de Infraestructura Transversal). Sin embargo, para llegar a la '*Capa de Persistencia de Datos*' y sus objetos *Repositorios* (puede recordar en algunos aspectos a la *Capa de Acceso a Datos (DAL)* tradicional, pero no es lo mismo), es recomendable que siempre se acceda a través de los objetos de coordinación (Servicios) de la Capa de Aplicación, puesto que es la parte que los orquesta.

Queremos resaltar que la implementación y uso de todas estas capas debe ser algo flexible. Relativo al diagrama, probablemente deberían existir más combinaciones de flechas (accesos). Y sobre todo, no tiene por qué ser utilizado de forma exactamente igual en todas las aplicaciones.

A continuación, en este capítulo describimos brevemente cada una de las capas y subcapas mencionadas. También presentamos algunos conceptos globales de cómo definir y trabajar con dichas capas (desacoplamiento entre algunas capas, despliegue en diferentes niveles físicos, etc.).

Posteriormente, en los próximos capítulos se procederá a definir y explicar en detalle cada una de dichas capas de primer nivel (Un capítulo por cada capa de primer nivel).

Capa de Presentación

Esta capa es responsable de mostrar información al usuario e interpretar sus acciones.

Los componentes de las capas de presentación implementan la funcionalidad requerida para que los usuarios interactúen con la aplicación. Normalmente es recomendable subdividir dichos componentes en varias sub-capas aplicando patrones de tipo MVC, MVP o M-V-VM:

- **Subcapa de Componentes Visuales (Vistas):** Estos componentes proporcionan el mecanismo base para que el usuario utilice la aplicación. Son componentes que formatean datos en cuanto a tipos de letras y controles visuales, y también reciben datos proporcionados por el usuario.
- **Subcapa de Controladores:** Para ayudar a sincronizar y orquestar las interacciones del usuario, puede ser útil conducir el proceso utilizando componentes separados de los componentes propiamente gráficos. Esto impide que el flujo de proceso y lógica de gestión de estados esté programada dentro de los propios controles y formularios visuales y permite reutilizar dicha lógica y patrones desde otros interfaces o ‘vistas’. También es muy útil para poder realizar pruebas unitarias de la lógica de presentación. Estos ‘*Controllers*’ son típicos de los patrones MVC y derivados.

Capa de Servicios Distribuidos (Servicios Web) –Opcional-

Cuando una aplicación actúa como proveedor de servicios para otras aplicaciones remotas, o incluso si la capa de presentación está también localizada físicamente en localizaciones remotas (aplicaciones *Rich-Client*, RIA, OBA, etc.), normalmente se publica la lógica de negocio (capas de negocio internas) mediante una capa de servicios. Esta capa de servicios (habitualmente Servicios Web) proporciona un medio de acceso remoto basado en canales de comunicación y mensajes de datos. Es importante destacar que esta capa debe ser lo más ligera posible y que no debe incluir nunca 'lógica' de negocio. Hoy por hoy, con las tecnologías actuales hay muchos elementos de una arquitectura que son muy simples de realizar en esta capa y en muchas ocasiones se tiende a incluir en ella propósitos que no le competen.

Capa de Aplicación

Esta capa forma parte de la propuesta de arquitecturas orientadas al Dominio. Define los trabajos que la aplicación como tal debe de realizar y redirige a los objetos del dominio y de infraestructura (persistencia, etc.) que son los que internamente deben resolver los problemas.

Realmente esta capa no debe contener reglas del dominio o conocimiento de la lógica de negocio, simplemente debe realizar tareas de coordinación de aspectos tecnológicos de la aplicación que nunca explicaríamos a un experto del dominio o usuario de negocio. Aquí implementamos la coordinación de la ‘fontanería’ de la aplicación, como coordinación de transacciones, ejecución de unidades de trabajo, y en definitiva llamadas a tareas necesarias para la aplicación (*software* como tal). Otros aspectos a implementar aquí pueden ser optimizaciones de la aplicación, conversiones de datos/formatos, etc. pero siempre nos referimos solo a la coordinación. El trabajo final se delegará posteriormente a los objetos de las capas inferiores. Esta capa tampoco debe contener estados que reflejen la situación de la lógica de negocio interna pero sí puede tener estados que reflejen el progreso de una tarea de la aplicación con el fin de mostrar dichos progresos al usuario.

Es una capa en algunos sentidos parecida a las capas “*Fachada de Negocio*”, pues en definitiva hará de fachada del modelo de Dominio, pero no solamente se encarga de simplificar el acceso al Dominio, hace algo más. Aspectos a incluir en esta capa serían:

- Coordinación de la mayoría de las llamadas a objetos *Repositorios* de la Capa de Persistencia y acceso a datos.
- Agrupaciones/agregaciones de datos de diferentes entidades para ser enviadas de una forma más eficiente (minimizar las llamadas remotas) por la capa superior de servicios web. Estos objetos a enviar son los DTOs, (*Data Transfer Object*) y el código en la capa de aplicación son *DTO-Adapters*.
- Acciones que consolidan o agrupan operaciones del Dominio *dependiendo de las acciones mostradas en la interfaz de usuario*, relacionando dichas acciones con las operaciones de persistencia y acceso a datos.
- Mantenimiento de estados relativos a la aplicación (no estados internos del Dominio).
- Coordinación de acciones entre el Dominio y aspectos de infraestructura. Por ejemplo, la acción de realizar una transferencia bancaria requiere obtener datos de las fuentes de datos haciendo uso de los Repositorios, utilizar posteriormente objetos del dominio con la lógica de negocio de la transferencia (abono y cargo) y a lo mejor finalmente mandar un e-mail a las partes interesadas invocando otro objeto de infraestructura que realice dicho envío del e-mail.
- **Servicios de Aplicación:** Es importante destacar que el concepto de Servicios en una arquitectura *N-Layer* con orientación al dominio, no tiene nada que ver con los Servicios-Web para accesos remotos. Primeramente, el concepto de servicio DDD existe diferentes capas, tanto en las capas de Aplicación, de Dominio e incluso en la de Infraestructura. **El concepto de servicios es simplemente un conjunto de clases donde agrupar comportamientos y métodos de acciones** que no pertenecen a una clase de bajo nivel concreta (como entidades y Servicios del Dominio u otro tipo de clase con identidad propia.). Así pues, los servicios normalmente coordinarán objetos de capas inferiores.

En cuanto a los ‘Servicios de Aplicación’, que es el punto actual, estos servicios normalmente coordinan el trabajo de otros servicios de capas inferiores (Servicios de Capas del Dominio o incluso Servicios de capas de Infraestructura transversal). Por ejemplo, un servicio de la capa de aplicación puede llamar a otro servicio de la capa del dominio para que efectúe la lógica de la creación de un pedido en las entidades ‘en memoria’. Una vez efectuadas dichas operaciones de negocio por la Capa del Dominio (la mayoría son cambios en objetos en memoria), la capa de aplicación podrá llamar a Repositorios de infraestructura delegando en ellos para que se encarguen de

persistir los cambios en las fuentes de datos. Esto es un ejemplo de coordinación de servicios de capas inferiores.

- **Workflows de Negocio (Opcional):** Algunos procesos de negocio están formados por un cierto número de pasos que deben ejecutarse de acuerdo a unas reglas concretas dependiendo de eventos que se puedan producir en el sistema y, normalmente, con un **tiempo de ejecución total de larga duración** (indeterminado, en cualquier caso), interactuando unos pasos con otros mediante una orquestación dependiente de dichos eventos. Este tipo de procesos de negocio se implementan de forma natural como flujos de trabajo (*workflows*) mediante tecnologías concretas y herramientas de gestión de procesos de negocio especialmente diseñadas para ello.

También esta capa de Aplicación puede ser publicada mediante la capa superior de servicios web, de forma que pueda ser invocada remotamente.

Capa del Dominio

Esta capa es responsable de representar conceptos de negocio, información sobre la situación de los procesos de negocio e implementación de las reglas del dominio. También debe contener los estados que reflejan la situación de los procesos de negocio. **Esta capa, ‘Dominio’, es el corazón del software.**

Así pues, estos componentes implementan la funcionalidad principal del sistema y encapsulan toda la lógica de negocio relevante (genéricamente llamado lógica del Dominio según nomenclatura DDD). Básicamente suelen ser clases en el lenguaje seleccionado que implementan la lógica del dominio dentro de sus métodos. Siguiendo los patrones de Arquitecturas N-Layer con Orientación al Dominio, esta capa tiene que ignorar completamente los detalles de persistencia de datos. Estas tareas de persistencia deben ser realizadas por las capas de infraestructura y coordinadas por la capa de Aplicación.

Normalmente podemos definir los siguientes elementos dentro de la capa de Dominio:

- **Entidades del Dominio:** Estos objetos son entidades desconectadas (datos + lógica) y se utilizan para alojar y transferir datos de entidades entre las diferentes capas. Pero adicionalmente, una característica fundamental en DDD es que contengan también la lógica del dominio relativo a cada entidad. Por ejemplo, en un abono bancario, la operación de sumar una cantidad de dinero al saldo de una cuenta la debemos realizar con lógica dentro de la propia entidad cuenta. Otros ejemplos son validaciones de datos relacionados con lógica de negocio, campos pre-calculados, relaciones con otras sub-entidades, etc. Estas clases representan al fin y al cabo las entidades de negocio del mundo real, como productos o pedidos. Las entidades de datos que la aplicación utiliza internamente, son en cambio objetos en memoria con datos y cierta lógica relacionada. Si usásemos ‘solo los datos’ de las entidades sin la lógica de la propia entidad dentro de la misma clase, estaríamos cayendo en el anti-patrón

llamado ‘*Anemic Domain Model*’, descrito originalmente sobre todo por *Martin Fowler*. Adicionalmente y siguiendo los patrones y principios recomendados, es bueno que estas clases entidad sean también objetos POCO (*Plain Old CLR Objects*), es decir, clases independientes de tecnologías concretas de acceso a datos, con código completamente bajo nuestro control. En definitiva, con este diseño (*Persistence Ignorance*) lo que buscamos es que las clases del dominio ‘no sepan nada’ de las interioridades de los repositorios ni de las tecnologías de acceso a datos. Cuando se trabaja en las capas del dominio, se debe ignorar cómo están implementados los repositorios.

Las clases entidad se sitúan dentro del dominio, puesto que son entes del dominio e independientes de cualquier tecnología de infraestructura (persistencia de datos, ORMs, etc.). En cualquier caso, las entidades serán objetos flotantes a lo largo de toda o casi toda la arquitectura.

Relativo a DDD, y de acuerdo con la definición de Eric Evans, “*Un objeto primariamente definido por su identidad se le denomina Entidad*”. Las entidades son fundamentales en el modelo del Dominio y tienen que ser identificadas y diseñadas cuidadosamente. Lo que en algunas aplicaciones puede ser una entidad, en otras aplicaciones no debe serlo. Por ejemplo, una ‘dirección’ en algunos sistemas puede no tener una identidad en absoluto, pues puede estar representando solo atributos de una persona o compañía. En otros sistemas, sin embargo, como en una aplicación para una empresa de Electricidad, la dirección de los clientes puede ser muy importante y debe ser una entidad, porque la facturación puede estar ligada directamente con la dirección. En este caso, una dirección tiene que clasificarse como una Entidad del Dominio. En otros casos, como en una aplicación de comercio electrónico, la dirección puede ser simplemente un atributo del perfil de una persona. En este último caso, la dirección no es tan importante y debería clasificarse como un ‘Objeto Valor’ (En DDD denominado ‘*Value-Object*’).

- **Servicios del Dominio:** En la capas del Dominio, los servicios son básicamente clases agrupadoras de comportamientos y/o métodos con ejecución de lógica del dominio. Estas clases normalmente no deben contener estados relativos al dominio (deben ser clases *stateless*) y serán las clases que coordinen e inicien operaciones compuestas contra las entidades del dominio. Un caso típico de un Servicio del Dominio es que esté relacionado con varias entidades al mismo tiempo. Pero también podemos tener un Servicio que esté encargado de interactuar (obtener, actualizar, etc.) contra una única entidad raíz (la cual sí puede englobar a otros datos relacionados siguiendo el patrón *Aggregate*).
- **Contratos de Repositorios:** Está claro que la implementación de los propios Repositorios no estará en el dominio, puesto que la implementación de los Repositorios no es parte del Dominio sino parte de las capas de Infraestructura (Los Repositorios están ligados a una tecnología de persistencia de datos, como un ORM). Sin embargo, los interfaces o ‘contratos’ de cómo deben estar contruidos dichos Repositorios, si deben formar parte del Dominio. En dichos contratos se especifica qué debe ofrecer cada Repositorio para que funcione y

se integre correctamente con el Dominio, sin importarnos como están implementados por dentro.

Dichos interfaces/contratos si son ‘agnósticos’ a la tecnología, aun cuando la implementación de los interfaces, por el contrario, si esté ligada a ciertas tecnologías. Así pues, es importante que los interfaces/contratos de los Repositorios estén definidos dentro de las Capas del Dominio. Esto es uno de los puntos recomendados en arquitecturas con orientación al Dominio y está basado en el patrón ‘*Separated Interface Pattern*’ definido por *Martin Fowler*.

Lógicamente, para poder cumplir este punto, es necesario que las ‘*Entidades del Dominio*’ y los ‘*Value-Objects*’ sean POCO; es decir, los objetos encargados de alojar las entidades y datos deben ser también completamente agnósticos a la tecnología de acceso a datos. Hay que tener en cuenta que las entidades del dominio son, al final, los ‘tipos’ de los parámetros enviados y devueltos por y hacia los Repositorios.

Capa de Infraestructura de Acceso a Datos

Esta capa proporciona la capacidad de persistir datos así como lógicamente acceder a ellos. Pueden ser datos propios del sistema o incluso acceder a datos expuestos por sistemas externos (Servicios Web externos, etc.). Así pues, esta capa de persistencia de datos expone el acceso a datos a las capas superiores, normalmente las capas del dominio. Esta exposición deberá realizarse de una forma desacoplada.

- **Implementación de ‘Repositorios’:** A nivel genérico, un Repositorio “*Representa todos los objetos de un cierto tipo como un conjunto conceptual*” (Definición de *Eric Evans*). A nivel práctico, un Repositorio será normalmente una clase encargada de realizar las operaciones de persistencia y acceso a datos, estando ligado por lo tanto a una tecnología concreta (p.e. ligado a un ORM como *Entity Framework*, *NHibernate*, o incluso simplemente *ADO.NET* para un gestor de bases de datos concreto). Haciendo esto centralizamos la funcionalidad de acceso a datos, lo cual hace más directo y sencillo el mantenimiento y configuración de la aplicación.

Normalmente debemos crear un *Repository* por cada ‘Entidad Raíz del Dominio’. Es casi lo mismo que decir que la relación entre un *Repository* y una entidad raíz es una relación 1:1. Las entidades raíz podrán ser a veces aisladas y otras veces la raíz de un ‘*Aggregate*’, que es un conjunto de entidades ‘*Object Values*’ más la propia entidad raíz.

El acceso a un Repositorio debe realizarse mediante un interfaz bien conocido, un contrato ‘depositado’ en el Dominio, de forma que podríamos llegar a sustituir un Repositorio por otro que se implemente con otras tecnologías y, sin embargo, la capa del Dominio no se vería afectada.

El punto clave de los Repositorios es que deben facilitar al desarrollador el mantenerse centrado en la lógica del modelo del Dominio y esconder por lo tanto la ‘fontanería’ del acceso a los datos mediante dichos ‘contratos’ de

repositorios. A este concepto se le conoce también como '*PERSISTENCE IGNORANCE*', lo cual significa que el modelo del Dominio ignora completamente cómo se persisten o consultan los datos contra las fuentes de datos de cada caso (Bases de datos u otro tipo de almacén).

Por último, es **fundamental diferenciar entre un objeto 'Data Access' (utilizados en muchas arquitecturas tradicionales N-Layer) y un Repositorio**. La principal diferencia radica en que un objeto '*Data Access*' realiza directamente las operaciones de persistencia y acceso a datos contra el almacén (normalmente una base de datos). Sin embargo, un Repositorio 'registra' en memoria (un contexto) las operaciones que se quieren hacer, pero estas no se realizarán hasta que desde la capa de Aplicación se quieran efectuar esas 'n' operaciones de persistencia/acceso en una misma acción, todas a la vez. Esto está basado normalmente en el patrón 'Unidad de Trabajo' o '*Unit of Work*', que se explicará en detalle en el capítulo de 'Capa de Aplicación'. Este patrón o forma de aplicar/efectuar operaciones contra los almacenes, en muchos casos puede aumentar el rendimiento de las aplicaciones, y en cualquier caso, reduce las posibilidades de que se produzcan inconsistencias. También reduce los tiempos de bloqueos en tabla debidos a transacciones.

- **Componentes Base (*Layer SuperType*):** La mayoría de las tareas de acceso a datos requieren cierta lógica común que puede ser extraída e implementada en un componente separado y reutilizable. Esto ayuda a simplificar la complejidad de los componentes de acceso a datos y sobre todo, minimiza el volumen de código a mantener. Estos componentes pueden ser implementados como clases base o clases utilidad (dependiendo del uso) y ser código reutilizado en diferentes proyectos/aplicaciones.

Este concepto es realmente un patrón muy conocido denominado '*Layered Supertype Pattern*' definido por *Martin Fowler*, que dice básicamente "Si los comportamientos y acciones comunes de un tipo de clases se agrupan en una clase base, esto eliminará muchos duplicados de código y comportamientos". El uso de este patrón es puramente por conveniencia y no distrae de prestar atención al Dominio en absoluto.

El patrón '*Layered Supertype Pattern*' se puede aplicar a cualquier tipo de capa (Dominios, Infraestructura, etc.), no solamente a los Repositorios.

- **'Modelo de Datos':** Normalmente los sistemas ORM (como *Entity Framework*) disponen de técnicas de definición del modelo de datos a nivel de diagramas 'entidad-relación', incluso a nivel visual. Esta subcapa deberá contener dichos modelos entidad relación, a ser posible, de forma visual con diagramas.
- **Agentes de Servicios remotos/externos:** Cuando un componente de negocio debe utilizar funcionalidad proporcionada por servicios externos/remotos

(normalmente Servicios Web), se debe implementar código que gestione la semántica de comunicaciones con dicho servicio particular o incluso tareas adicionales como mapeos entre diferentes formatos de datos. Los Agentes de Servicios aíslan dicha idiosincrasia de forma que, manteniendo ciertos interfaces, sería posible sustituir el servicio externo original por un segundo servicio diferente sin que nuestro sistema se vea afectado.

Capas de Infraestructura Transversal/Horizontal

Proporcionan capacidades técnicas genéricas que dan soporte a capas superiores. En definitiva, son ‘bloques de construcción’ ligados a una tecnología concreta para desempeñar sus funciones.

Existen muchas tareas implementadas en el código de una aplicación que se deben aplicar en diferentes capas. Estas tareas o aspectos horizontales (Transversales) implementan tipos específicos de funcionalidad que pueden ser accedidos/utilizados desde componentes de cualquier capa. Los diferentes tipos/aspectos horizontales más comunes, son: **Seguridad** (Autenticación, Autorización y Validación) y **tareas de gestión de operaciones** (políticas, *logging*, trazas, monitorización, configuración, etc.). Estos aspectos serán detallados en capítulos posteriores.

- **Subcapas de ‘Servicios de Infraestructura’:** En las capas de infraestructura transversal también existe el concepto de Servicios. Se encargarán de agrupar acciones de infraestructura, como mandar e-mails, controlar aspectos de seguridad, gestión de operaciones, *logging*, etc. Así pues, estos **Servicios**, agrupan cualquier tipo de actividad de infraestructura transversal ligada a tecnologías específicas.
- **Subcapas de objetos de infraestructura:** Dependiendo del tipo de aspecto de infraestructura transversal, necesitaremos los objetos necesarios para implementarlos, bien sean aspectos de seguridad, trazas, monitorización, envío de e-mails, etc.

Estas capas de ‘Infraestructura Transversal’ engloban una cantidad muy grande de conceptos diferentes, muchos de ellos relacionados con Calidad de Servicio (*QoS – Quality of Service*) y realmente, cualquier implementación ligada a una tecnología/infraestructura concreta. Es por ello que se definirá en detalle en un capítulo dedicado a estos aspectos transversales.

‘Servicios’ como concepto genérico disponible en las diferentes Capas

Debido a que los SERVICIOS están presentes en diferentes capas de una Arquitectura DDD, resumimos a continuación en un cuadro especial sobre el concepto de SERVICIO utilizado en DDD.

Tabla 1.- Servicios en Arquitecturas N-Layer Orientadas al Dominio

Servicios en Arquitecturas N-Layer Orientadas al Dominio

Como hemos visto en diferentes Capas (APLICACIÓN, DOMINIO e INFRAESTRUCTURA-TRANSVERSAL), en todas ellas podemos disponer de una sub-capa denominada Servicios. Debido a que es un concepto presente en diferentes puntos, es bueno tener un visón global sobre qué son los ‘Servicios’ enDDD.

Primeramente es importante aclarar, para no confundir conceptos, que los SERVICIOS enDDD no son los SERVICIOS-WEB utilizados para invocaciones remotas. Estos otros SERVICIOS-WEB estarán en una posible capa superior de ‘Capa de Servicios Distribuidos’ y podrían a su vez publicar las capas inferiores permitiendo acceso remoto a los SERVICIOS-DDD y también a otros objetos de la Capa de Aplicación y de Dominio.

Centrándonos en el concepto de SERVICIO enDDD, en algunos casos, los diseños más claros y pragmáticos incluyen operaciones que no pertenecen conceptualmente a objetos específicos de cada capa (p.e. operaciones que no pertenezcan de forma exclusiva a una entidad). En estos casos podemos incluir/agrupar dichas operaciones en SERVICIOS explícitos.

Dichas operaciones son intrínsecamente actividades u operaciones, no características de cosas u objetos específicos de cada capa. Pero debido a que nuestro modelo de programación es orientado a objetos, debemos agruparlos también en objetos. A estos objetos les llamamos SERVICIOS.

El forzar a dichas operaciones (normalmente operaciones de alto nivel y agrupadoras de otras acciones) a formar parte de objetos naturales de la capa, distorsionaría la definición de los objetos reales de la capa. Por ejemplo, la lógica propia de una entidad debe de estar relacionada con su interior, cosas como validaciones con respecto a sus datos en memoria, o campos calculados, etc., pero no el tratamiento de la propia entidad como un todo. Un ‘motor’ realiza acciones relativas al uso del motor, no relativas a cómo se fabrica dicho motor. Así mismo, la lógica perteneciente a una clase entidad no debe encargarse de su propia persistencia y almacenamiento.

Un SERVICIO es una operación o conjunto de operaciones ofrecidas como un interfaz que simplemente está disponible en el modelo, sin encapsular estados.

La palabra “Servicio” del patrón SERVICIO precisamente hace hincapié en lo que ofrece: *“Qué puede hacer y qué acciones ofrece al cliente que lo consuma y enfatiza la relación con otros objetos de cada capa”*.

A algunos SERVICIOS (sobre todo los de más alto nivel, en la Capa de Aplicación y/o algunos servicios del dominio coordinadores de lógica de negocio) se les suele nombrar con nombres de Actividades, no con nombres de objetos. Están por lo tanto relacionados con verbos de los Casos de Uso del análisis, no con sustantivos (objetos), aun cuando puede tener una definición abstracta de una operación concreta (Por ejemplo, un Servicio-Transferencia relacionado con la acción/verbo ‘Transferir Dinero de una cuenta bancaria a otra’).

Los servicios no deben tener estados (deben ser *stateless*). Esto no implica que la clase que lo implementa tenga que ser estática, podrá ser perfectamente una clase instanciable. Que un SERVICIO sea *stateless* significa que un programa cliente puede hacer uso de cualquier instancia de un servicio sin importar su historia individual como objeto.

Adicionalmente, la ejecución de un SERVICIO hará uso de información que es accesible globalmente y puede incluso cambiar dicha información (es decir, normalmente provoca cambios globales). Pero el servicio no contiene estados que pueda afectar a su propio comportamiento, como si tienen por ejemplo las entidades.

A modo de aclaración mostramos como particionar diferentes Servicios en diferentes capas en un escenario bancario simplificado:

APLICACIÓN


*Servicio de **Aplicación** de ‘BankingService’
(Operaciones Bancarias)*

- *Asimila y convierte formatos de datos de entrada (Como conversiones de datos XML)*
- *Proporciona datos de la transferencia a la Capa de Dominio para que sea allí realmente procesada la lógica de negocio.*
- *Coordina/invoca a los objetos de persistencia (Repositorios) de la capa de infraestructura, para persistir los cambios realizados en las entidades e cuentas bancarias, por la capa del dominio.*
- *Decide si se envía notificación (e-mail al usuario) utilizando servicios de infraestructura transversal.*
- *En definitiva, implementa toda la ‘coordinación de la fontanería tecnológica’*

	(como uso de transacciones y Unit of Work) para que la Capa de Dominio quede lo más limpia posible y exprese mejor y muy claramente su lógica.
DOMINIO	<p>Servicio de Dominio de 'Transferencia-Bancaria' (Verbo Transferir Fondos)</p> <ul style="list-style-type: none"> - Coordina el uso de los objetos entidad como 'CuentaBancaria' y otros objetos del Dominio bancario. - Proporciona confirmación del resultado de las operaciones de negocio.
INFRAESTRUCTURA-TRANSVERSAL	<p>Servicio de Infraestructura Transversal de 'Envío Notificaciones' (Verbo Enviar/Notificar)</p> <ul style="list-style-type: none"> - Envía un correo electrónico, mensaje SMS u otro tipo de comunicación requerido por la aplicación

De todo lo explicado hasta este punto en el presente capítulo, se desprende la primera regla a cumplir en un desarrollo de aplicación empresarial siguiendo esta guía de Arquitectura Marco:


Tabla 2.- Regla de Diseño D1

 <p>Regla N°: D1.</p>	<p>El diseño de <u>arquitectura lógica interna de una aplicación</u> se realizará siguiendo el modelo de arquitectura de aplicaciones en N-Capas (<i>N-Layered</i>) con Orientación al Dominio y tendencias y patrones DDD (<i>Domain Driven Design</i>)</p>
<p>○ <u>Normas</u></p> <ul style="list-style-type: none"> - Por regla general, esta regla deberá aplicarse en casi el 100% de aplicaciones empresariales complejas, con un cierto volumen y propietarias de mucha lógica de Dominio. 	



Cuándo SÍ implementar una arquitectura N-Capas con Orientación al Dominio

- Deberá implementarse en las aplicaciones empresariales complejas cuya lógica de negocio cambie bastante y la aplicación vaya a sufrir cambios y mantenimientos posteriores durante una vida de aplicación, como mínimo, relativamente larga.



Cuándo NO implementar una arquitectura N-Capas DDD

- En aplicaciones pequeñas que una vez finalizadas se prevén pocos cambios, la vida de la aplicación será relativamente corta y donde prima la velocidad en el desarrollo de la aplicación. En estos casos se recomienda implementar la aplicación con tecnologías RAD (como puede ser *'Microsoft RIA Services'*), aunque tendrá la desventaja de implementar componentes más fuertemente acoplados, la calidad resultante de la aplicación será peor y el coste futuro de mantenimiento probablemente será mayor dependiendo de si la aplicación continuará su vida con un volumen grande de cambios o no.



Ventajas del uso de Arquitecturas N-Capas

- Desarrollo estructurado, homogéneo y similar de las diferentes aplicaciones de una organización.
- Facilidad de mantenimiento de las aplicaciones pues los diferentes tipos de tareas están siempre situados en las mismas áreas de la arquitectura.
- Fácil cambio de tipología en el despliegue físico de una aplicación (2-Tier, 3-Tier, etc.), pues las diferentes capas pueden separarse físicamente de forma fácil.



Desventajas del uso de Arquitecturas N-Capas

- En el caso de aplicaciones muy pequeñas, estamos añadiendo una complejidad excesiva (capas, desacoplamiento, etc.). Pero este caso es muy poco probable en aplicaciones empresariales con cierto nivel.



Referencias

Eric Evans: Libro “Domain-Driven Design: Tackling Complexity in the Heart of Software”

Martin Fowler: Definición del ‘Domain Model Pattern’ y Libro “Patterns of Enterprise Application Architecture”

Jimmy Nilson: Libro “Applying Domain-Driven-Design and Patterns with examples in C# and .NET”

SoC - Separation of Concerns principle:
http://en.wikipedia.org/wiki/Separation_of_concerns

EDA - Event-Driven Architecture: SOA Through the Looking Glass – “The Architecture Journal”

EDA - Using Events in Highly Distributed Architectures – “The Architecture Journal”

Sin embargo, aunque estas son las capas inicialmente propuestas para cubrir un gran porcentaje de aplicaciones **N-Layered**, la arquitectura base está abierta a la implementación de nuevas capas y personalizaciones necesarias para una aplicación dada (por ejemplo capa EAI para integración con aplicaciones externas, etc.).

Así mismo, tampoco es obligatoria la implementación completa de las capas de componentes propuestas. Por ejemplo, en algunos casos podría no implementarse la capa de Servicios-Web por no necesitar implementar accesos remotos, etc.



2.3.- Desacoplamiento entre componentes

Es fundamental destacar que no solo se deben de delimitar los componentes de una aplicación entre diferentes capas. Adicionalmente, también debemos tener especial atención en cómo interaccionan unos componentes/objetos con otros, es decir, cómo se consumen y en especial cómo se instancian unos objetos desde otros.

En general, este desacoplamiento debería realizarse entre todos los objetos (con lógica de ejecución y dependencias) pertenecientes a las diferentes capas, pues existen

.....

ciertas capas las cuales nos puede interesar mucho el que se integren en la aplicación de una forma desacoplada. Este es el caso de la mayoría de capas de Infraestructura (ligadas a unas tecnologías concretas), como puede ser la propia capa de persistencia de datos, que podemos haber ligado a una tecnología concreta de ORM o incluso a un acceso a *backend* externo concreto (p.e. ligado a accesos a un Host, ERP o cualquier otro *backend* empresarial). En definitiva, para poder integrar esa capa de forma desacoplada, no debemos instanciar directamente sus objetos (p.e., no instanciar directamente los objetos Repositorio o cualquier otro relacionado con una tecnología concreta, de la infraestructura de nuestra aplicación).

Pero la esencia final de este punto, realmente trata del desacoplamiento entre cualquier tipo/conjunto de objetos. Bien sean conjuntos de objetos diferentes dentro del propio Dominio (p.e. para un país, cliente o tipología concreta, poder inyectar unas clases específicas de lógica de negocio), o bien, en los componentes de Capa de presentación poder simular la funcionalidad de Servicios-Web, o en la Capa de Persistencia poder también simular otros Servicios-Web externos y en todos esos casos realizarlo de forma desacoplada para poder cambiar de la ejecución real a la simulada o a otra ejecución real diferente, con el menor impacto. En todos esos ejemplos tiene mucho sentido un desacoplamiento de por medio.

En definitiva, es conseguir un ‘*state of the art*’ del diseño interno de nuestra aplicación: *“Tener preparada toda la estructura de la Arquitectura de tu aplicación de forma desacoplada y en cualquier momento poder inyectar funcionalidad para cualquier área o grupo de objetos, no tiene por qué ser solo entre capas diferentes”*.

Un enfoque exclusivo de “desacoplamiento entre capas” probablemente no es el más correcto. El ejemplo de conjuntos de objetos diferentes a inyectar dentro del propio Dominio, que es una única capa (p.e. para un país, cliente o tipología concreta, un módulo incluso vertical/funcional), clarifica bastante.

En la aplicación ejemplo anexa a esta Guía de Arquitectura hemos optado por realizar desacoplamiento entre todos los objetos de las capas internas de la aplicación, porque ofrece muchas ventajas y así mostramos la mecánica completa.

Las técnicas de desacoplamiento están basadas en el **Principio de Inversión de Dependencias**, el cual establece una forma especial de desacoplamiento donde se invierte la típica relación de dependencia que se suele hacer en orientación a objetos la cual decía que las capas de alto nivel deben depender de las Capas de más bajo nivel. El propósito es conseguir disponer de capas de alto nivel que sean independientes de la implementación y detalles concretos de las capas de más bajo nivel, y por lo tanto también, independientes de las tecnologías subyacentes.

El **Principio de Inversión de Dependencias** establece:

- A. Las capas de alto nivel no deben depender de las capas de bajo nivel. Ambas capas deben depender de abstracciones (Interfaces)
- B. Las abstracciones no deben depender de los detalles. Son los Detalles (Implementación) los que deben depender de las abstracciones (Interfaces).

El objetivo del principio de inversión de dependencias es desacoplar los componentes de alto nivel de los componentes de bajo nivel de forma que sea posible llegar a reutilizar los mismos componentes de alto nivel con diferentes implementaciones de componentes de bajo nivel. Por ejemplo, poder reutilizar la misma Capa de Dominio con diferentes Capas de Infraestructura que implementen diferentes tecnologías (“diferentes detalles”) pero cumpliendo los mismos interfaces (abstracciones) de cara a la Capa de Dominio.

Los contratos/interfaces definen el comportamiento requerido a los componentes de bajo nivel por los componentes de alto nivel y además dichos contratos/interfaces deben existir en los *assemblies* de alto nivel.

Cuando los componentes de bajo nivel implementan los interfaces/contratos a cumplir (que se encuentran en las capas de alto nivel), significa que los componentes/capas de bajo nivel son las que dependen, a la hora de compilar, de los componentes de alto nivel, invirtiendo la tradicional relación de dependencia. Por eso se llama “*Inversión de Dependencias*”.

Existen varias técnicas y patrones que se utilizan para facilitar el ‘aprovisionamiento’ de la implementación elegida de las capas/componentes de bajo nivel, como son *Plugin*, *Service Locator*, *Dependency Injection* e *IoC (Inversion of Control)*.

Básicamente, las técnicas principales que proponemos utilizar para habilitar el desacoplamiento entre capas, son:

- Inversión de control (IoC)
- Inyección de dependencias (DI)
- Interfaces de Servicios Distribuidos (para consumo/acceso remoto a capas)

El uso correcto de estas técnicas, gracias al desacoplamiento que aportan, potencia los siguientes puntos:

- Posibilidad de sustitución, en tiempo de ejecución, de capas/módulos actuales por otros diferentes (con mismos interfaces y similar comportamiento), sin que impacte a la aplicación. Por ejemplo, puede llegar a sustituirse en tiempo de ejecución un módulo que accede a una base de datos por otro que accede a un sistema externo tipo HOST o cualquier otro tipo de sistema, siempre y cuando cumplan unos mismos interfaces. No sería necesario el añadir el nuevo módulo, especificar referencias directas y recompilar nuevamente la capa que lo consume.
- Posibilidad de uso de **STUBS/MOLES y MOCKS** en pruebas: Es realmente un escenario concreto de cambio de un módulo por otro. En este caso consiste por ejemplo, en sustituir un módulo de acceso a datos reales (a bases de datos o cualquier otra fuente de datos) por un módulo con interfaces similares pero que simplemente simula que accede a las fuentes de datos. Mediante la inyección de dependencias puede realizarse este cambio incluso en tiempo de ejecución, sin llegar a tener que recompilar la solución.



2.4.- Inyección de dependencias e Inversión de control

Patrón de Inversión de Control (IoC): Delegamos a un componente o fuente externa, la función de seleccionar un tipo de implementación concreta de las dependencias de nuestras clases. En definitiva, este patrón describe técnicas para soportar una arquitectura tipo ‘plug-in’ donde los objetos pueden buscar instancias de otros objetos que requieren y de los cuales dependen.

Patrón Inyección de Dependencias (*Dependency Injection, DI*): Es realmente un caso especial de IoC. Es un patrón en el que se suplen objetos/dependencias a una clase en lugar de ser la propia clase quien cree los objetos/dependencias que necesita. El término fue acuñado por primera vez por *Martin Fowler*.

Entre las diferentes capas no debemos de instanciar de forma explícita las dependencias. Para conseguir esto, se puede hacer uso de una clase base o un interfaz (nos parece más claro el uso de interfaces) que defina una abstracción común que pueda ser utilizada para inyectar instancias de objetos en componentes que interactúen con dicho interfaz abstracto compartido.

Para dicha inyección de objetos, inicialmente se podría hacer uso de un “Constructor de Objetos” (*Patrón Factory*) que crea instancias de nuestras dependencias y nos las proporciona a nuestro objeto origen, durante la creación del objeto y/o inicialización. Pero, la forma más potente de implementar este patrón es mediante un “*Contenedor DI*” (En lugar de un “Constructor de Objetos” creado por nosotros). El contenedor DI inyecta a cada objeto las dependencias/objetos necesarios según las relaciones o registro plasmado bien por código o en ficheros XML de configuración del “Contenedor DI”.

Típicamente este contenedor DI es proporcionado por un *framework* externo a la aplicación (como *Unity*, *Castle-Windsor*, *Spring.NET*, etc.), por lo cual en la aplicación también se utilizará Inversión de Control al ser el contenedor (almacenado en una biblioteca) quien invoque el código de la aplicación.

Los desarrolladores codificarán contra un interfaz relacionado con la clase y usarán un contenedor que inyectará las instancias de los objetos dependientes en la clase en base al interfaz o clase declarada de los objetos dependientes. Las técnicas de inyección de instancias de objetos son ‘inyección de interfaz’, ‘inyección de constructor’, ‘inyección de propiedad’ (*setter*), e ‘inyección de llamada a método’.

Cuando la técnica de ‘Inyección de Dependencias’ se utiliza para desacoplar objetos de nuestras capas, el diseño resultante aplicará por lo tanto el “*Principio de Inversión de Dependencias*”.

Un escenario interesante de desacoplamiento con IoC es internamente dentro de la Capa de Presentación, para poder realizar un *mock o stub/mole* de una forma aislada y configurable de los componentes en arquitecturas de presentación tipo MVC y MVVM, donde para una ejecución rápida de pruebas unitarias podemos querer simular un consumo de Servicio Web cuando realmente no lo estamos consumiendo, sino simulando.

Y por supuesto, la opción más potente relativa al desacoplamiento es hacer uso de IoC y DI entre prácticamente todos los objetos pertenecientes a las capas de la arquitectura, esto nos permitirá en cualquier momento inyectar simulaciones de comportamiento o diferentes ejecuciones reales cambiándolo en tiempo de ejecución y/o configuración.

En definitiva, los contenedores IoC y la Inyección de dependencias añaden flexibilidad y conllevan a ‘tocar’ el menor código posible según avanza el proyecto. Añaden comprensión y mantenibilidad del proyecto.

Tabla 3.- Inyección de Dependencias (DI) y Desacoplamiento entre objetos como ‘Mejor Práctica’

Inyección de Dependencias (DI) y Desacoplamiento entre objetos como ‘Mejor Práctica’

El principio de 'Única responsabilidad' (*Single Responsibility Principle*) establece que cada objeto debe de tener una única responsabilidad.

El concepto fue introducido por *Robert C. Martin*. Se establece que una responsabilidad es una razón para cambiar y concluye diciendo que una clase debe tener una y solo una razón para cambiar.

Este principio está ampliamente aceptado por la industria del desarrollo y en definitiva promueve el diseño y desarrollo de clases pequeñas con una única responsabilidad. Esto está directamente relacionado con el número de dependencias (objetos de los que depende) cada clase. Si una clase tiene una única responsabilidad, sus métodos normalmente deberán tener pocas dependencias con otros objetos. Si hay una clase con muchísimas dependencias (por ejemplo 15 dependencias), esto nos estaría indicando lo que típicamente se dice como un 'mal olor' del código. Precisamente, haciendo uso de Inyección de dependencias en el constructor, por sistema nos vemos obligados a declarar todas las dependencias de objetos en el constructor y en dicho ejemplo veríamos muy claramente que esa clase en concreto parece que no sigue el principio de '*Single Responsibility*', pues es bastante raro que la clase tenga una única responsabilidad y sin embargo en su constructor veamos declaradas 15 dependencias. Así pues, DI es también una forma de guía que nos conduce a realizar buenos diseños e implementaciones en desarrollo, además de ofrecernos un desacoplamiento que podemos utilizar para inyectar diferentes ejecuciones de forma transparente.

Mencionar también que es factible diseñar e implementar una Arquitectura Orientada al Dominio (siguiendo patrones con tendencias DDD) sin implementar técnicas de desacoplamiento (Sin IoC ni DI). No es algo ‘obligatorio’, pero sí que

favorece mucho el aislamiento del Dominio con respecto al resto de capas, lo cual si es un objetivo primordial en DDD. La inversa también es cierta, es por supuesto también factible utilizar técnicas de desacoplamiento (*IoC* y *Dependency Injection*) en Arquitecturas no Orientadas al Dominio. En definitiva, hacer uso de IoC y DI, es una filosofía de diseño y desarrollo que nos ayuda a crear un código mejor diseñado y que favorece, como decíamos el principio de '*Single Responsibility*'.

Los contenedores IoC y la inyección de dependencias favorecen y facilitan mucho el realizar correctamente Pruebas Unitarias y Mocking. Diseñar una aplicación de forma que pueda ser probada de forma efectiva con Pruebas Unitarias nos fuerza a realizar 'un buen trabajo de diseño' que deberíamos estar haciendo si realmente sabemos qué estamos haciendo en nuestra profesión.

Los interfaces y la inyección de dependencias ayudan a hacer que una aplicación sea extensible (tipo *pluggable*) y eso a su vez ayuda también al *testing*. Podríamos decir que esta facilidad hacia el testing es un efecto colateral 'deseado', pero no el más importante proporcionado por IoC y DI.

Sin embargo, IoC y DI no son solo para favorecer las Pruebas Unitarias, como remarcamos aquí:

Tabla 4.- IoC y DI no son solo para favorecer las Pruebas Unitarias

¡¡IoC y DI no son solo para favorecer las Pruebas Unitarias!!

Esto es fundamental. ¡La Inyección de Dependencias y los contenedores de Inversión de Control no son solo para favorecer el *Testing* de Pruebas Unitarias e Integración! Decir eso sería como decir que el propósito principal de los interfaces es facilitar el *testing*. Nada más lejos de la realidad.

DI e IoC tratan sobre desacoplamiento, mayor flexibilidad y disponer de un punto central donde ir que nos facilite la mantenibilidad de nuestras aplicaciones. El *Testing* es importante, pero no es la primera razón ni la más importante por la que hacer uso de Inyección de Dependencias ni IoC.

Otro aspecto a diferenciar es dejar muy claro que DI y los contenedores IoC no son lo mismo.

Tabla 5.- Diferenciamiento entre DI e IoC

DI e IoC son cosas diferentes

Hay que tener presente que DI e IoC son cosas diferentes.

DI (Inyección de dependencias mediante constructores o propiedades) puede sin duda ayudar al *testing* pero el aspecto útil principal de ello es que guía a la aplicación hacia el **Principio de Única Responsabilidad** y también normalmente hacia el principio de '**Separación de Preocupaciones/Responsabilidades**' (*Separation Of*

Concerns Principle). Por eso, DI es una técnica muy recomendada, una mejor práctica en el diseño y desarrollo de software.

Debido a que implementar DI por nuestros propios medios (por ejemplo con clases *Factory*) puede llegar a ser bastante farragoso, se usan contenedores IoC para proporcionar flexibilidad a la gestión del grafo de dependencias de objetos.

Tabla 6.- Regla de Diseño N° D2



El consumo y comunicación entre los diferentes objetos pertenecientes a las capas de la arquitectura deberá ser desacoplado, implementando los patrones de ‘Inyección de dependencias’ (DI) e ‘Inversión de Control’ (IoC).

○ Normas

- Por regla general, esta regla deberá aplicarse en todas la arquitecturas N-Capas de aplicaciones medianas/grandes. *Por supuesto, debe de realizarse entre los objetos cuya función mayoritaria es la lógica de ejecución (de cualquier tipo) y que tienen dependencias con otros objetos. Un ejemplo claro son los Servicios, Repositorios, etc. No tiene mucho sentido hacerlo con las propias clases de Entidades.*



Cuándo SÍ implementar ‘Inyección de dependencias’ e ‘Inversión de Control’

- Deberá implementarse en prácticamente la totalidad de las aplicaciones empresariales N-Capas que tengan un volumen mediano/grande. Es especialmente útil entre las capas del Dominio y las de Infraestructura así como en la capa de presentación junto con patrones tipo MVC y M-V-VM.



Cuándo NO implementar ‘Inyección de dependencias’ e ‘Inversión de Control’

- A nivel de proyecto, normalmente, no se podrá hacer uso de DI e IoC en aplicaciones desarrolladas con tecnologías RAD (*Rapid Application Development*) que no llegan a implementar realmente una aplicación N-Capas flexible y no hay posibilidad de introducir este tipo de desacoplamiento. Esto pasa habitualmente en aplicaciones pequeñas.
- A nivel de objetos, en las clases que son ‘finales’ o no tienen dependencias (como las ENTIDADES), no tiene sentido hacer uso de

IoC.



Ventajas del uso de ‘Inyección de dependencias’ e ‘Inversión de Control’

- Posibilidad de sustitución de Capas/bloques, en tiempo de ejecución.
- Facilidad de uso de STUBS/MOCKS/MOLES para el *Testing* de la aplicación.
- Añaden flexibilidad y conllevan a ‘tocar’ el menor código posible según avanza el proyecto.
- Añaden comprensión y mantenibilidad al proyecto.



Desventajas del uso de ‘Inyección de dependencias’ e ‘Inversión de Control’

- Si no se conocen las técnicas IoC y DI, se añade cierta complejidad inicial en el desarrollo de la aplicación, pero una vez comprendidos los conceptos, realmente merece la pena en la mayoría de las aplicaciones, ya que añade mucha flexibilidad y finalmente calidad de software.



Referencias

Inyección de Dependencias: MSDN -
<http://msdn.microsoft.com/enus/library/cc707845.aspx>

Inversión de Control: MSDN - *<http://msdn.microsoft.com/enus/library/cc707904.aspx>*

Inversion of Control Containers and the Dependency Injection pattern (By Martin Fowler) - *<http://martinfowler.com/articles/injection.html>*



2.5.- Módulos

En las aplicaciones grandes y complejas, el modelo de Dominio tiende a crecer extraordinariamente. El modelo llega a un punto donde es complicado hablar sobre ello como ‘un todo’, y puede costar bastante entender bien todas sus relaciones e interacciones entre todas sus áreas. Por esa razón, se hace necesario organizar y *particionar* el modelo en diferentes módulos. Los módulos se utilizan como un método de organización de conceptos y tareas relacionadas (normalmente bloques de negocio diferenciados) para reducir la complejidad desde un punto de vista externo.

El concepto de módulo es realmente algo utilizado en el desarrollo de software desde sus orígenes. Es más fácil ver la foto global de un sistema completo si lo subdividimos en diferentes módulos verticales y después en las relaciones entre dichos módulos. Una vez que se entienden las interacciones entre dichos módulos, es más sencillo focalizarse en más detalle de cada uno de ellos. Es una forma simple y eficiente de gestionar la complejidad. El lema “Divide y vencerás” es la frase que mejor lo define.

Un buen ejemplo de división en módulos son la mayoría de los ERPs. Normalmente están divididos en módulos verticales, cada uno de ellos responsable de un área de negocio específico. Ejemplos de módulos de un ERP podrían ser: Nómina, Gestión de Recursos Humanos, Facturación, Almacén, etc.

Otra razón por la que hacer uso de módulos está relacionada con la calidad del código. Es un principio aceptado por la industria el hecho de que **el código debe tener un alto nivel de cohesión y un bajo nivel de acoplamiento**. Mientras que la cohesión empieza en el nivel de las clases y los métodos, también puede aplicarse a nivel de módulo. Es recomendable, por lo tanto, agrupar las clases relacionadas en módulos, de forma que proporcionemos la máxima cohesión posible. Hay varios tipos de cohesión. Dos de las más utilizadas son “Cohesión de Comunicaciones” y “Cohesión Funcional”. La cohesión relacionada con las comunicaciones tiene que ver con partes de un módulo que operan sobre los mismos conjuntos de datos. Tiene todo el sentido agruparlo, porque hay una fuerte relación entre esas partes de código. Por otro lado, la cohesión funcional se consigue cuando todas las partes de un módulo realizan una tarea o conjunto de tareas funcionales bien definidas. Esta cohesión es el mejor tipo.

Así pues, el uso de módulos en un diseño es una buena forma de aumentar la cohesión y disminuir el acoplamiento. Habitualmente los módulos se dividirán y repartirán las diferentes áreas funcionales diferenciadas y que no tienen una relación/dependencia muy fuerte entre ellas. Sin embargo, normalmente tendrá que existir algún tipo de comunicación entre los diferentes módulos, de forma que deberemos definir también interfaces para poder comunicar unos módulos con otros. En lugar de llamar a cinco objetos de un módulo, probablemente es mejor llamar a un interfaz (p.e. un Servicio DDD) del otro módulo que agrega/agrupa un conjunto de funcionalidad. Esto reduce también el acoplamiento.

Un bajo acoplamiento entre módulos reduce la complejidad y mejora sustancialmente la mantenibilidad de la aplicación. Es mucho más sencillo también entender cómo funciona un sistema completo, cuando tenemos pocas conexiones entre módulos que realizan tareas bien definidas. En cambio, si tenemos muchas conexiones de unos módulos a otros es mucho más complicado entenderlo, y si es necesario tenerlo así, probablemente debería ser un único módulo. Los módulos deben ser bastante independientes unos de otros.

El nombre de cada módulo debería formar parte del '*Lenguaje Ubicuo*' de DDD, así como cualquier nombre de entidades, clases, etc. Para más detalles sobre qué es el '*Lenguaje Ubicuo*' en DDD, leer documentación sobre DDD como el libro de *Domain-Driven Design* de Eric Evans.

A continuación mostramos el esquema de arquitectura propuesta pero teniendo en cuenta diferentes posibles módulos de una aplicación:

Módulos en Arquitectura N-Capas con Orientación al Dominio

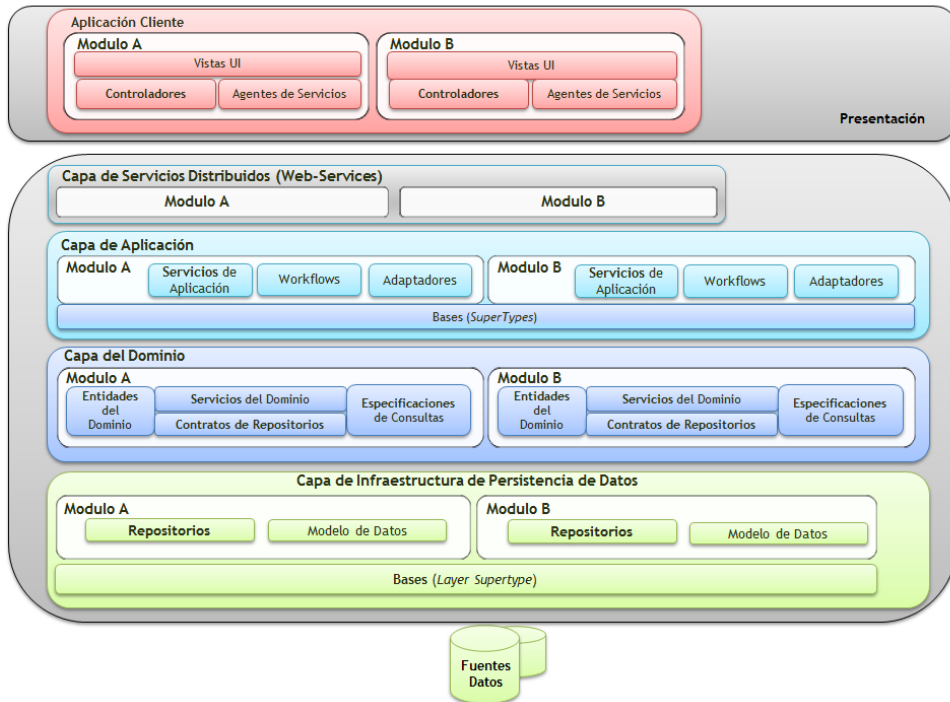



Figura 9.- Módulos en Arquitectura N-Capas con Orientación al Dominio

A nivel de interfaz de usuario, el problema que surge cuando hay diferentes grupos de desarrollo trabajando en los diferentes módulos es que, al final, la capa de presentación (la aplicación cliente) es normalmente solo una y los cambios a realizar en ella por unos grupos de desarrollo pueden molestar/estorbar a los cambios a hacer por otros grupos de desarrollo.

Debido a esto, los módulos tienen mucho que ver con el concepto de aplicaciones compuestas (*'Composite Applications'*), donde diferentes grupos de desarrollo pueden estar trabajando sobre la misma aplicación pero de una forma independiente, cada equipo de desarrollo en un módulo diferente. Pero finalmente todo se tiene que integrar en el mismo interfaz de usuario. Para que esa integración visual sea mucho menos traumática, es deseable hacer uso de conceptos de *'Composite Applications'*, es decir, definir interfaces concretos que cada módulo visual debe cumplir (áreas de menús, áreas de contenido, carga/descarga de módulos visuales a partir de un punto configurable de la aplicación, etc.), de forma que sea una integración muy automatizada y reglada y no algo traumático al hacer la integración de los diferentes módulos en una única aplicación cliente.

Tabla 7.- Regla de Diseño N° D3



Regla N°: D3.

Definir y diseñar Módulos de Aplicación que engloben áreas funcionales diferenciadas.

○ **Normas**

- Por regla general, esta regla deberá aplicarse en la mayoría de las aplicaciones con cierto volumen y áreas funcionales diferenciadas.

✓ **Cuándo SÍ diseñar e implementar Módulos**

- Deberá implementarse en prácticamente la totalidad de las aplicaciones empresariales que tengan un volumen mediano/grande y sobre todo donde se pueda diferenciar diferentes áreas funcionales que sean bastante independientes entre ellas.

✗ **Cuándo NO diseñar e implementar Módulos**

- En aplicaciones donde se disponga de una única área funcional muy cohesionada entre ella y sea muy complicado separarlo en módulos funcionales independientes y desacoplados a nivel funcional.

👍 **Ventajas del uso de 'Módulos'**

→ El uso de módulos en un diseño es una buena forma de aumentar la cohesión

y disminuir el acoplamiento.

- Un bajo acoplamiento entre módulos reduce la complejidad y mejora sustancialmente la mantenibilidad de la aplicación.



Desventajas del uso de ‘Módulos’

- Si las entidades de un hipotético módulo tienen muchas relaciones con otras entidades de otro/s módulos, probablemente debería ser un único módulo.
- Cierta inversión en tiempo inicial añadido de diseño que obliga a definir interfaces de comunicación entre unos módulos y otros. Sin embargo, siempre que encaje bien la definición y separación de módulos (existen áreas funcionales diferenciadas), será muy beneficioso para el proyecto.



Referencias

Modules: Libro DDD – Eric Evans

Microsoft - Composite Client Application Library: <http://msdn.microsoft.com/en-us/library/cc707819.aspx>



2.6.- Subdivisión de modelos y contextos de trabajo

En esta sección veremos cómo trabajar con modelos de gran tamaño, expondremos técnicas para mantener la coherencia de los modelos mediante la división de un modelo de gran tamaño en varios modelos más pequeños con fronteras bien definidas. En esta sección nos centraremos en los bounded context. Es vital tener claro que un bounded context no es lo mismo que un contexto de un ORM tipo *Entity Framework* o sesiones de *NHibernate*, sino que representa un concepto completamente distinto, de contexto de trabajo de un grupo de desarrollo, como veremos a continuación.



2.7.- Bounded Contexts

En aplicaciones de gran tamaño y complejidad nuestros modelos crecen muy rápidamente en términos de número de elementos y relaciones entre los mismos. Mantener la coherencia en modelos tan grandes es muy complicado debido tanto al tamaño de los mismos como a la cantidad de personas trabajando al mismo tiempo en ellos. Es muy fácil que dos personas tengan interpretaciones distintas de un mismo concepto, o que repliquen un concepto en otro objeto por no saber que dicho concepto está ya implementado en otro objeto. Para solucionar estos problemas debemos poner un límite al tamaño de los modelos definiendo un contexto dentro del cual dichos modelos son válidos.

La idea de tener un modelo único para todo el sistema es tentadora pero irrealizable debido a que mantener la coherencia dentro de un modelo tan grande es casi imposible y no merece la pena en términos de coste. De hecho, la primera pregunta que debemos hacernos al afrontar el desarrollo de un modelo de gran tamaño es ¿Necesitamos total integración entre cada una de las funcionalidades de nuestro sistema? La respuesta a esta pregunta será no en el 90% de los casos.

Por tanto, los modelos grandes los vamos a separar en varios modelos de menor tamaño, estableciendo que dado un elemento determinado de nuestro sistema, este solo tiene sentido dentro del contexto (o submodelo) donde está definido. Nos centraremos en mantener la coherencia dentro de estos contextos y trataremos aparte las relaciones entre contextos. Los contextos son particiones del modelo destinadas a mantener la coherencia, no una simple partición funcional del mismo. Las estrategias para definir contextos pueden ser múltiples, como por ejemplo dividir en contextos por equipos de trabajo (la idea es fomentar la comunicación y la integración continua dentro de un contexto), por funcionalidades de alto nivel del sistema (uno o varios módulos funcionales), etc. Por ejemplo, en un proyecto donde estamos construyendo un sistema nuevo que debe funcionar en paralelo con un sistema en mantenimiento, está claro que el sistema antiguo tiene su contexto, y que no queremos que nuestro nuevo sistema esté en su mismo contexto, ya que esto influiría en el diseño de nuestro nuevo sistema. Otro posible ejemplo es la existencia de un algoritmo optimizado para algún tipo cálculo donde se utiliza un modelo completamente distinto, como por ejemplo cualquier tipo de cálculo matemático complejo que queramos realizar sobre los elementos de nuestro modelo.

Establecer contextos dentro de un sistema tiene el inconveniente de que perdemos la visión global del mismo, esto provoca que cuando dos contextos tienen que comunicarse para implementar una funcionalidad tiendan a mezclarse. Por este motivo es fundamental definir simultáneamente a los contextos un mapa de contextos, donde se establecen claramente los distintos contextos existentes en el sistema y las relaciones entre los mismos. De esta forma obtenemos las ventajas de coherencia y cohesión que

nos ofrecen los contextos y preservamos la visión global del sistema estableciendo claramente las relaciones entre contextos.



2.8.- Relaciones entre contextos

Las distintas relaciones que se dan entre dos o más contextos dependen fundamentalmente del grado de comunicación que exista entre los distintos equipos de cada contexto y del grado de control que se tenga de los mismos. Por ejemplo, puede ocurrir que no podamos realizar modificaciones en un contexto, como puede ser el caso de un sistema en producción o descontinuado, o puede ocurrir que nuestro sistema se apoye en otros sistemas para funcionar. A continuación veremos algunas relaciones que típicamente se dan entre contextos, pero es importante entender que no debemos forzar estas relaciones entre contextos en nuestro sistema a no ser que se presenten de forma natural.



2.8.1.- Shared Kernel

Cuando tenemos dos o más contextos en los que trabajan equipos que pueden comunicarse de forma fluida, es interesante establecer una responsabilidad compartida sobre los objetos que ambos contextos utilizan para relacionarse con el otro contexto. Estos objetos pasan a formar lo que se denomina el shared kernel o núcleo compartido de ambos contextos, y queda establecido que para realizar una modificación en cualquier objeto del shared kernel se requiere la aprobación de los equipos de todos los contextos implicados. Es recomendable crear conjuntamente entre todos los equipos de los contextos implicados pruebas unitarias para cada objeto del shared kernel, de forma que el comportamiento del shared kernel quede completamente definido. Favorecer la comunicación entre los distintos equipos es crítico, por lo que una buena práctica es hacer circular a algunos miembros de cada equipo por los equipos de los otros contextos, de manera que el conocimiento acumulado en un contexto se transmita al resto.



2.8.2.- Customer/Supplier

Es bastante frecuente encontrar que estamos desarrollando un sistema que depende de otros sistemas para hacer su trabajo, como por ejemplo puede ser un sistema de análisis o un sistema de toma de decisiones. En este tipo de sistemas suelen existir dos

contextos, en un contexto se encuentra nuestro sistema, que “consume” al sistema del que depende y que se encuentra en el otro contexto.

Las dependencias entre los dos contextos son en una sola dirección, del contexto “cliente” al contexto “proveedor”, del sistema dependiente hacia el sistema dependido.

En este tipo de relaciones el cliente puede verse limitado por necesitar funcionalidades del sistema proveedor, y al mismo tiempo el contexto proveedor puede cohibirse a la hora de realizar cambios por miedo a provocar la aparición de bugs en el contexto o contextos clientes. Para solucionar este tipo de problemas la clave es la comunicación entre los equipos de los distintos contextos. Los miembros del equipo de los contextos clientes deberían participar como clientes en las reuniones de planificación del equipo proveedor para priorizar las historias de usuario del sistema proveedor, y se debería crear conjuntamente un juego de pruebas de aceptación para el sistema proveedor, de forma que quede perfectamente definida la interfaz que esperan los contextos clientes, y el contexto proveedor pueda realizar cambios sin miedo a cambiar por error la interfaz que esperan los contextos clientes.



2.8.3.- Conformista

La relación cliente/proveedor requiere de la colaboración entre los equipos de los distintos contextos. Esta situación suele ser bastante ideal, y en la mayoría de los casos el contexto proveedor tiene sus propias prioridades y no está dispuesto a atender a las necesidades del contexto cliente. En este tipo de situaciones donde nuestro contexto depende de otro contexto sobre el cual no tenemos control alguno, (no podemos realizar modificaciones ni pedir funcionalidades) y con el que tenemos una estrecha relación, (el coste de la traducción de las comunicaciones de un contexto a otro es elevado) podemos emplear un acercamiento conformista, que consiste en acomodar nuestro modelo al expuesto por el otro contexto. Esto limita nuestro modelo a hacer simples adiciones al modelo del otro contexto, y limita la forma que puede tomar nuestro modelo. No obstante no es una idea descabellada, ya que posiblemente el otro modelo incorpore el conocimiento acumulado en el desarrollo del otro contexto. La decisión de seguir una relación de conformismo depende en gran medida de la calidad del modelo del otro contexto. Si no es adecuado, debe seguirse un enfoque más defensivo como puede ser un Anti-corruption layer o Separate ways como veremos a continuación.



2.8.4.- Anti-corruption Layer

Todas las relaciones que hemos visto hasta ahora presuponen la existencia de una buena comunicación entre los equipos de los distintos contextos o un modelo de un

contexto bien diseñado que puede ser adoptado por otro. ¿Pero qué ocurre cuando un contexto está mal diseñado y no queremos que este hecho influya sobre nuestro contexto? Para este tipo de situaciones podemos implementar un anti-corruption layer, que consiste en crear una capa intermedia entre contextos que se encarga de realizar la traducción entre nuestro contexto y el contexto con el que tenemos que comunicarnos. Generalmente esta comunicación la vamos a iniciar nosotros, aunque no tiene porqué ser así.

Un anti-corruption layer se compone de tres elementos: adaptadores, traductores y fachadas. Primero se diseña una fachada que simplifica la comunicación con el otro contexto y que expone solo la funcionalidad que nuestro contexto va a utilizar. Es importante tener claro que la fachada debe definirse en términos de elementos del modelo del otro contexto, ya que si no estaríamos mezclando la traducción con el acceso al otro sistema. Frente a la fachada de sitúa un adaptador que modifica la interfaz del otro contexto para adaptarla a la interfaz que espera nuestro contexto, y que hace uso de un traductor para mapear los elementos de nuestro contexto a los que espera la fachada del otro contexto.



2.8.5.- Separate ways

La integración está sobrevalorada, y muchas veces no merece la pena el coste que conlleva. Por este motivo, dos grupos de funcionalidades que no tengan relación pueden desarrollarse en contextos distintos sin comunicación entre los mismos. Si tenemos funcionalidades que necesitan hacer uso de los dos contextos siempre podemos realizar esta orquestación a más alto nivel.



2.8.6.- Open Host

Típicamente cuando desarrollamos un sistema y decidimos realizar una separación en contextos, lo normal es crear una capa intermedia de traducción entre contextos. Cuando el número de contextos es elevado la creación de estas capas de traducción supone una carga extra de trabajo bastante importante. Cuando creamos un contexto lo normal es que éste presente una fuerte cohesión y que las funcionalidades que ofrece puedan verse como un conjunto de servicios. (No hablamos de servicios web, sino simplemente servicios).

En estas situaciones lo mejor es crear un conjunto de servicios que definan un protocolo de comunicación común para que otros contextos puedan utilizar la funcionalidad del contexto. Este servicio debe mantener la compatibilidad entre versiones, aunque puede ir aumentando las funcionalidades ofrecidas. Las funcionalidades expuestas deben ser generales, si otro contexto necesita una

funcionalidad específica se crea en una capa de traducción independiente para no contaminar el protocolo de nuestro contexto.



2.9.- Implementación de bounded contexts en .NET

Como hemos indicado al principio de la sección, los bounded context son unidades organizativas destinadas a mantener la coherencia de modelos de gran tamaño. Por este motivo un bounded context puede representar desde un área de funcionalidad del sistema, hasta un sistema externo o un grupo de componentes destinados a realizar una tarea de forma óptima. No existe por tanto una regla general para implementar un bounded context, pero en este apartado trataremos los aspectos más importantes y pondremos algunos ejemplos de situaciones típicas.

En nuestra arquitectura dividimos el dominio y las funcionalidades en módulos de gran tamaño. Cada módulo es lógico que vaya asignado a un grupo de trabajo distinto, y que presente un conjunto de funcionalidades muy cohesivas, que pueden exponerse como un conjunto de servicios. Lo más lógico cuando tenemos que tratar con varios módulos es utilizar una relación “separate ways” entre módulos. Cada módulo a su vez será un “open host” que ofrecerá al resto un conjunto de funcionalidades en forma de servicios. De esta forma, cualquier funcionalidad que implique varios módulos se orquestrará desde un nivel superior. Cada módulo se encargará por tanto de su propio modelo de objetos, y de la gestión de la persistencia del mismo. En caso de utilizar Entity Framework esto significa que tendremos una correspondencia de 1 a 1 entre módulo y contexto de entity framework.

Dentro de cada módulo es bastante probable que exista complejidad suficiente como para que podamos seguir partiendo el sistema en contextos más pequeños. No obstante, estos contextos de trabajo estarán más relacionados y presentarán una relación de comunicación basada en un “shared kernel” o “customer/supplier”. En estos casos, el contexto es más una unidad organizativa que funcional. Los distintos contextos compartirán el mismo modelo de entity framework, pero la modificación de ciertos objetos clave estará sujeta al acuerdo entre los dos equipos de los distintos contextos.

Por último, queda tratar el aspecto específico de la relación de nuestro sistema con sistemas externos o componentes de terceros, que claramente son bounded context distintos. Aquí el enfoque puede ser aceptar el modelo del sistema externo, adoptando una postura “conformista” o podemos proteger nuestro dominio mediante un “anti-corruption layer” que traduzca nuestros conceptos a los conceptos del otro contexto. La decisión entre seguir un enfoque conformista u optar por un anti-corruption layer depende de la calidad del modelo del otro contexto y del coste de la traducción de nuestro contexto al otro contexto.



2.9.1.- ¿Cómo partir un modelo de Entity Framework?

El primer paso para partir un modelo es identificar los puntos donde existen entidades con poca relación entre ellas. No es imprescindible que no exista relación alguna entre entidades, y ahora veremos por qué. Examinemos en detalle una relación. ¿Cuál es la utilidad de que exista una relación entre dos entidades? Típicamente que una de las entidades hace uso de funcionalidad de la otra para implementar su propia funcionalidad. Como por ejemplo puede ser una entidad cuenta y una entidad cliente, en la que el patrimonio de un cliente se calcula a través de la agregación del balance de todas sus cuentas y propiedades.

De forma genérica, una relación entre dos entidades puede sustituirse por una consulta en el repositorio de una de las dos entidades. Esta consulta representa la asociación. En los métodos de la otra entidad podemos añadir un parámetro extra que contiene la información de la asociación como el resultado de la consulta al repositorio y puede operar de la misma forma que si la relación existiese.

La interacción entre las dos entidades se orquestará a nivel de servicio, ya que este tipo de interacción no es muy común y la lógica no suele ser compleja. En caso de que haya que modificar una asociación (por ejemplo añadiendo o eliminando algún elemento) tendremos en dichas entidades métodos de consulta que devolverán valores booleanos para indicar si dicha acción se debe llevar a cabo o no. En lugar de tener métodos para modificar la asociación que hemos eliminado. Siguiendo con nuestro ejemplo de las cuentas y los clientes, imaginemos que queremos calcular los intereses a pagar a un determinado cliente, que variarán dependiendo de las características del cliente. Este servicio además debe guardar los intereses en una nueva cuenta si exceden en una determinada cantidad dependiendo de la antigüedad del cliente. (Ya sabemos que en realidad no se hace así, pero es solo un caso ilustrativo) Tendríamos un servicio con la siguiente interfaz:

```
public interface IInterestRatingService
{
    void RateInterests(int clientId);
}
```

```
public class InterestRatingService : IInterestRatingService
{
    public InterestRatingService(IClientService clients,
                                IBankAccountService accounts)
    {
        ...
    }
    public void RateInterests(int clientId)
    {
        Client client = _clients.GetById(clientId);
        IEnumerable<BankAccount> clientAccounts =
            accounts.GetByClientId(clientId);
```

```

double interests = 0;
foreach(var account in clientAccounts)
{
    interests += account.calculateRate(client);
}
if(client.ShouldPlaceInterestsInaNewAccount(interests))
{
    BankAccount newAccount = new Account(interests);
    accounts.Add(newAccount);
}else{
    clientAccounts.First().Charge(interests);
}
}
}

```



2.9.2.- Relación entre bounded contexts y ensamblados

La existencia de un bounded context no implica directamente la creación de un ensamblado específico para él, sino que dependiendo de las relaciones del mapa de contextos, unos bounded contexts irán en el mismo ensamblado mientras que otros estarán separados. Lo normal, es que cuando dos bounded contexts tienen una relación fuerte, como por ejemplo la determinada por un shared kernel o un customer/supplier, dichos contextos se encuentren dentro del mismo ensamblado. En relaciones más débiles como pueden ser la interacción entre módulos, existen dos aproximaciones.

Una aproximación es tener todos los módulos en un mismo ensamblado, y utilizar solo los ensamblados para la división por capas. De esta manera se facilita la interacción entre módulos, al poder estos albergar referencias a elementos de cualquier otro módulo. Además tenemos la ventaja de tener todo nuestro dominio en un solo ensamblado, lo que simplifica el despliegue y la reutilización del dominio en otras aplicaciones. Hay que destacar que el hecho de que todos los módulos estén en el mismo ensamblado no significa que compartan el mismo contexto de Entity Framework. Este es el enfoque que hemos seguido en el ejemplo de interacción entre módulos.

La otra aproximación es tener cada módulo en un ensamblado distinto. Con esto no solo mejoramos sino que garantizamos el aislamiento entre módulos, pero las comunicaciones entre módulos se vuelven un poco complicadas. Cada módulo debería definir sus propias abstracciones de las entidades de otro módulo que necesite (que debieran reducirse al máximo), y en un nivel superior, a través de un anticorruption layer, crear un adaptador de las entidades del otro módulo a las abstracciones definidas en el módulo.



Mapeo de Tecnologías 'Wave .NET 4.0'

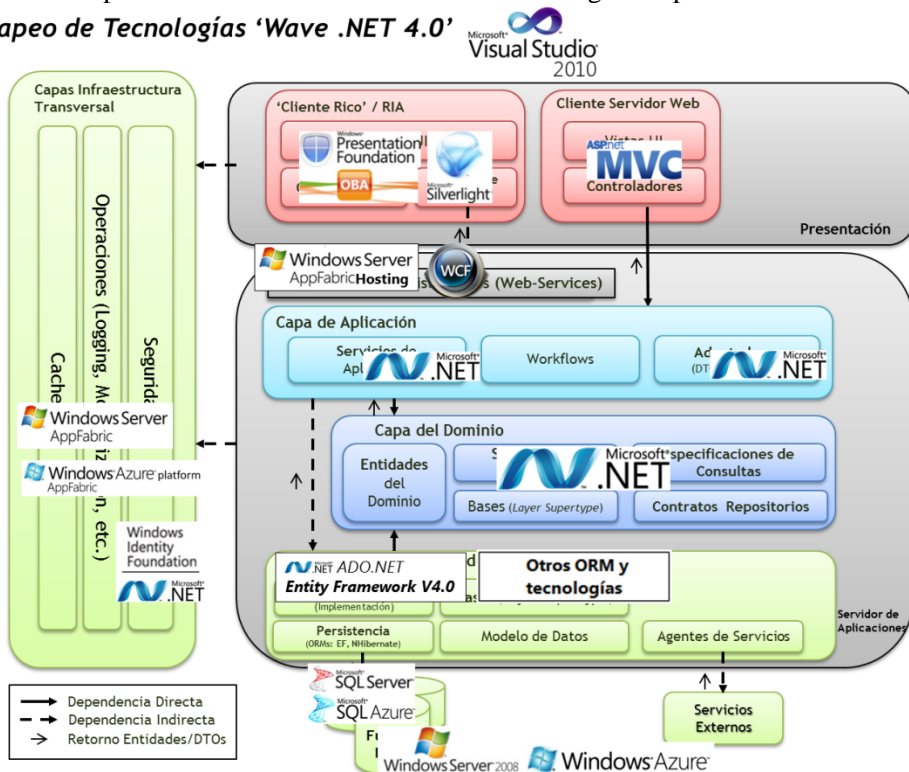


Figura 10.- Mapeo de tecnologías ‘OLA.NET 4.0’

2.11.-Implementación de Estructura de Capas en Visual Studio 2010

Para implementar una Arquitectura en Capas (según nuestro modelo lógico definido, orientado a Arquitecturas N-Capas DDD), hay una serie de pasos que debemos realizar:

- 1.- La solución de Visual Studio debe estar organizada y mostrar de forma clara y obvia donde está situada la implementación de cada capa y sub-capas.
- 2.- Cada capa tiene que estar correctamente diseñada y necesita incluir los patrones de diseño y tecnologías de cada capa.
- 3.- Existirán capas transversales de patrones y tecnologías a ser utilizados a lo largo de toda la aplicación, como la implementación de la tecnología escogida para IoC, o aspectos de seguridad, etc. Estas capas transversales (Infraestructura Transversal de DDD) serán capas bastante reutilizables en diferentes proyectos que se realicen en el futuro. Es un mini-framework, o mejor llamado *seedwork*, en definitiva, un código fuente que será reutilizado también en otros proyectos futuros, así como ciertas clases base (*Core*) de las capas del Dominio y Persistencia de Datos.



2.12.-Aplicación ejemplo N-Layer DDD con .NET 4.0

Prácticamente todos los ejemplos de código y estructuras de proyecto/*solutions* que se muestran en la presente guía pertenecen a la aplicación ejemplo que se ha desarrollado para acompañar este libro. Recomendamos encarecidamente bajar el código fuente de Internet e irlo revisando según se explica en el libro, pues siempre se podrán observar más detalles directamente en el código real.

La aplicación ejemplo está publicada en CODEPLEX, con licencia OPEN SOURCE, en esta URL:



<http://microsoftnlayerapp.codeplex.com/>



2.13.-Diseño de la solución de Visual Studio

Teniendo una ‘Solución’ de Visual Studio, inicialmente crearemos la estructura de carpetas lógicas para albergar y distribuir los diferentes proyectos. En la mayoría de los casos crearemos un proyecto (.DLL) por cada capa o sub-capa, para disponer así de una mayor flexibilidad y facilitar los posibles desacoplamientos. Sin embargo, esto ocasiona un número de proyectos considerable, por lo que es realmente imprescindible ordenarlos/jerarquizarlos mediante carpetas lógicas de Visual Studio.

La jerarquía inicial sería algo similar a la siguiente:

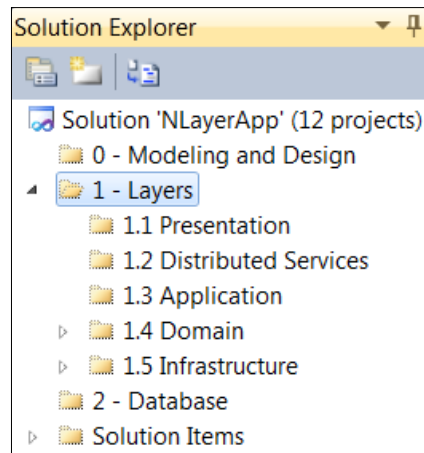


Figura 11.- Jerarquía de Carpetas en Solución de Visual Studio

Empezando por arriba, la primera carpeta (‘0 – Modeling & Design’) contendrá los diferentes diagramas de Arquitectura y Diseño realizados con VS.2010, como diagrama Layer de la Arquitectura, y diferentes diagramas UML de diseño interno. Estos diagramas los iremos utilizando para representar la implementación que hagamos.

La numeración de las capas es simplemente para que aparezcan en un orden adecuado siguiendo el orden real de la arquitectura y sea más sencillo buscar cada capa dentro del *solution* de Visual Studio.

La siguiente carpeta ‘1 Layers’, contendrá las diferentes capas de la Arquitectura N-Layer, como se observa en la jerarquía anterior.

Capa de Presentación

La primera capa, Presentación, contendrá los diferentes tipos de proyectos que pudiera haber, es decir, proyectos *Windows-Rich* (WPF, WinForms, OBA), RIA (Silverlight), Web (ASP.NET) o Windows Phone, etc.:

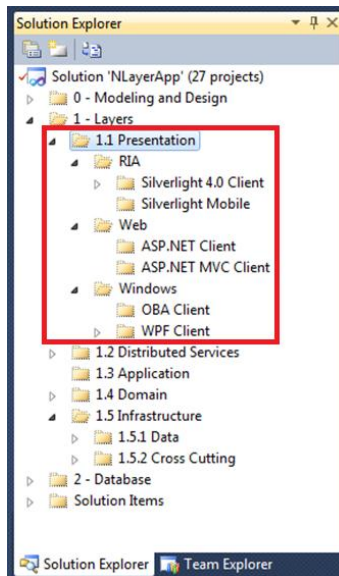


Figura 12.- Capas de Presentación

Posteriormente, tenemos las capas de componentes que normalmente están situadas en un servidor de aplicaciones (aunque ahí estaríamos hablando de despliegue, y eso puede variar, por lo que a nivel de organización en VS, no especificamos detalles de despliegue). En definitiva, dispondremos de las diferentes Capas principales de una Arquitectura *N-Layered* Orientada al Dominio, con diferentes proyectos para cada subcapa:

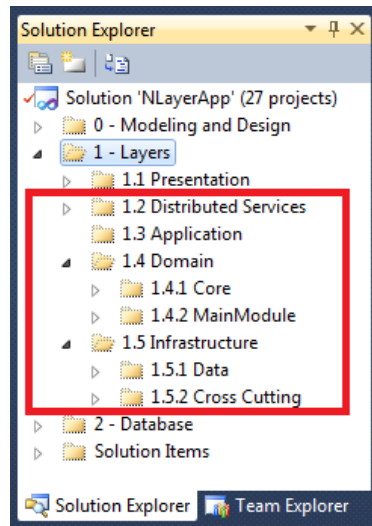


Figura 13.- Capas del Servidor de Aplicaciones

Dentro de cada una de dichas carpetas, añadiremos los proyectos necesarios según los elementos típicos de cada capa. Esto también viene determinado dependiendo de los patrones a implementar (explicados posteriormente a nivel lógico e implementación en la presente guía).

Capa de Servicios Distribuidos (Servicios WCF)

Esta Capa es donde implementaremos los Servicios WCF (normalmente Servicios-Web) para poder acceder remotamente a los componentes del Servidor de aplicaciones. Es importante destacar que esta capa de Servicios Distribuidos es opcional, puesto que en algunos casos (como capa de presentación web ASP.NET), es posible que se acceda directamente a los componentes de APPLICATION y DOMAIN, si el servidor Web de ASP.NET está en el mismo nivel de servidores que los componentes de negocio.

En el caso de hacer uso de servicios distribuidos para accesos remotos, la estructura puede ser algo similar a la siguiente:

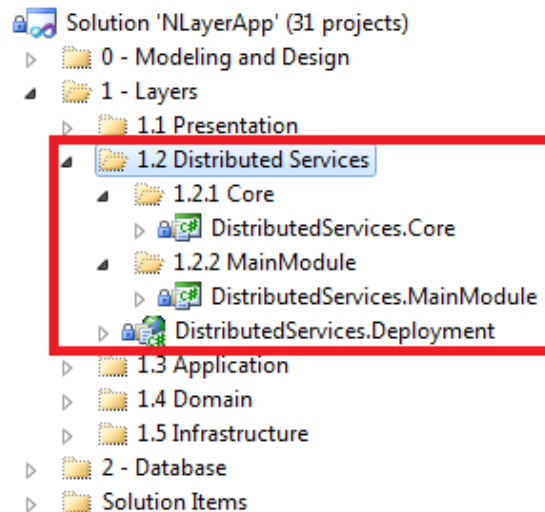


Figura 14.- Uso de servicios distribuidos

Un proyecto para el *Hoster* del Servicio WCF, es decir, el proceso donde se ejecuta y publica el servicio WCF. Ese proyecto/proceso puede ser de tipo *WebSite* de IIS (o *Casini* para desarrollo), un Servicio Windows, o realmente, cualquier tipo de proceso.

Y donde realmente está la funcionalidad del Servicio-Web es en los Servicios que exponen la lógica de cada módulo, es decir, dispondremos de **un proyecto de Servicio WCF (.DLL) por cada MÓDULO funcional de la aplicación**. En nuestro ejemplo, tenemos solo un módulo llamado '**MainModule**'.

En el caso de *hosting* de Servidor Web, internamente se añadirá un .SVC por cada MÓDULO de la aplicación.

Adicionalmente, deberá haber también un proyecto de clases de *Testing* (Pruebas Unitarias), dentro de esta capa.

Para un Servicio WCF en producción, se recomienda que el proyecto sea de tipo *WebSite* desplegado en IIS (**IIS 7.x**, a ser posible, para tener como posibilidad el utilizar *bindings* como *NetTCP* y no solamente *bindings* basados en HTTP), e incluso en el mejor escenario de despliegue, con IIS más **Windows Server AppFabric** para disponer de la monitorización e instrumentalización de los Servicios WCF, proporcionado por **AppFabric**.

Capa de Aplicación

Como se ha explicado anteriormente en la parte de Arquitectura lógica de esta guía, esta capa no debe contener realmente reglas del dominio o conocimiento de la lógica de negocio, simplemente debe realizar tareas de coordinación de aspectos tecnológicos de la aplicación que nunca explicaríamos a un experto del dominio o usuario de negocio. Aquí implementamos la coordinación de la 'fontanería' de la aplicación, como coordinación de transacciones, ejecución de unidades de trabajo, uso mayoritario de Repositorios y llamadas a objetos del Dominio.

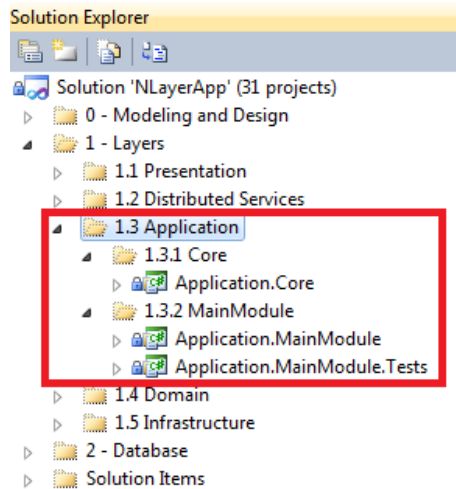


Figura 15.- Sub-Capas de Aplicación

Cada capa con clases lógicas tendrá a su vez un proyecto de clases de *Testing* (Pruebas Unitarias).

Capa de Dominio

Esta es la Capa más importante desde el punto de vista de la problemática de la aplicación, puesto que es aquí donde implementamos toda la lógica del dominio, entidades del dominio, etc.

Esta Capa tiene internamente varias sub-capas o tipos de elementos. Se recomienda consolidar al máximo el número de proyectos requerido dentro de una misma capa. Sin embargo, en este caso, es bueno disponer de un ensamblado/proyecto específico para las entidades, para que no estén acopladas a los Servicios del Dominio:

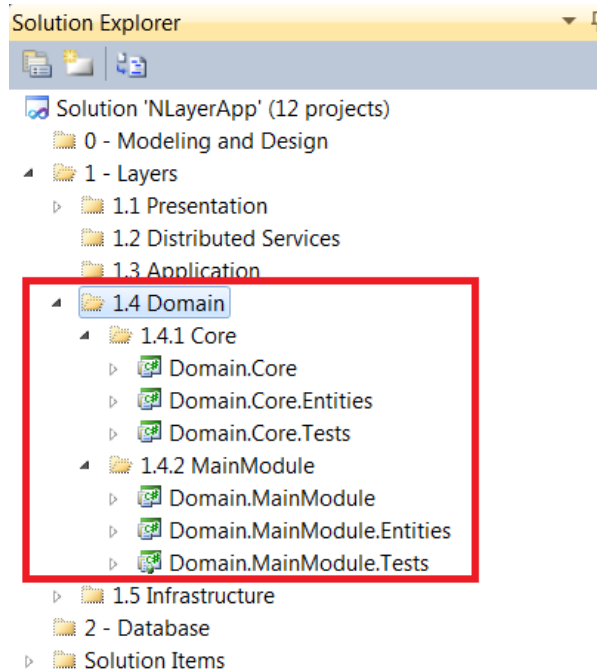


Figura 16.- Sub-Capas del Dominio

A nivel general, podemos disponer de un proyecto ‘Core’ de clases base y otras clases reutilizables de forma horizontal en todos los módulos funcionales del Dominio.

Por cada MODULO funcional de la aplicación (en el ejemplo, en este caso el llamado ‘MainModule’), implementaremos toda la lógica del módulo (Servicios, Especificaciones y Contratos de Repositorios) dentro de un único proyecto (en este caso Domain.MainModule), pero necesitamos un proyecto aislado para las ‘**Entidades del Dominio**’, por cada MÓDULO, donde Entity-Framework nos genere nuestras clases entidad POCO o Self-Tracking.

Este es el contenido de los proyectos de *Domain* a nivel de un módulo:

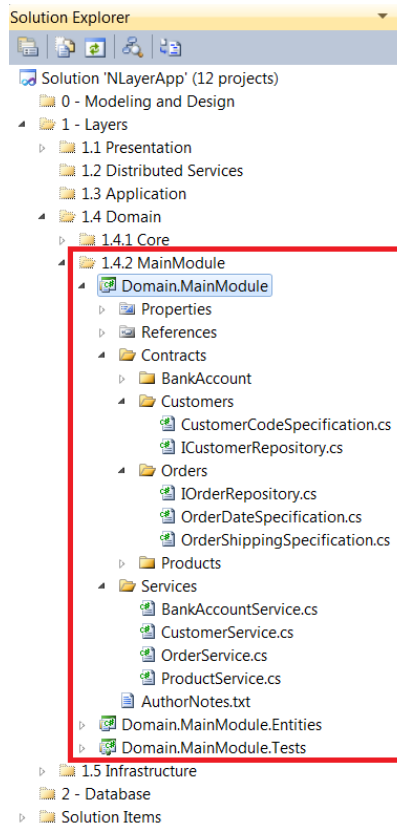


Figura 17.- Contenido de los proyectos de Dominio

Cada proyecto con clases lógicas tendrá a su vez un proyecto de clases de *Testing* (Pruebas Unitarias) y pudiéramos tener otros proyectos de pruebas de integración y funcionales.

Esta capa de Dominio se explica tanto a nivel lógico como de implementación en un capítulo completo de la guía.

Capa de Infraestructura de Persistencia de Datos

La parte más característica de esta capa es la implementación de REPOSITARIOS para realizar la persistencia y acceso a datos. En este módulo es también donde por lo tanto implementamos todo lo relacionado con el modelo y conexiones/acciones a la base de datos de ENTITY FRAMEWORK.

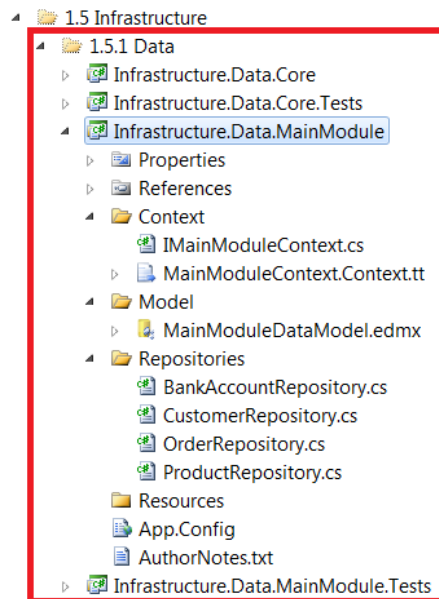


Figura 18.- Capa de Infraestructura de Persistencia de Datos

A nivel de cada MODULO funcional (en este caso, *MainModule*) dispondremos de un único proyecto con los siguientes elementos:

- **‘DataModel’**: Contendrá el modelo de *EntityFramework*. Si bien, las clases que genera Entity Framework (*Container* y Entidades POCO/IPOCO) las extraeremos a otros proyectos para poder desacoplarlo según el diseño del Dominio en DDD. Aquí solo estará el modelo de datos (En nuestro caso, *MainModuleDataModel.edmx*).
- **‘Context’** implementa una abstracción del contexto/contenedor de EntityFramework, para poder sustituirlo por un *fake/mock* y realizar pruebas unitarias.
- **Repositorios (Repositories)**: Clases encargadas de la lógica de persistencia de datos.

También dispondremos de otro proyecto para los Tests de todo el módulo.

Los proyectos de tipo **‘Core’** son proyectos a utilizar para implementar clases base y extensiones que son válidos para reutilizar de forma horizontal en la implementación de Capa de Persistencia de todos los módulos funcionales de la aplicación.

Esta capa de ‘Persistencia de Datos’ se explica tanto a nivel lógico como de implementación en un capítulo completo de la guía.



2.14.- Arquitectura de la Aplicación con Diagrama Layer de VS.2010

Para poder diseñar y comprender mejor la Arquitectura, en VS.2010 disponemos de un diagrama donde podemos plasmar la Arquitectura N-Layered que hayamos diseñado y adicionalmente nos permite mapear las capas que dibujemos visualmente con *namespaces* lógicos y/o *assemblies* de la solución. Esto posibilita posteriormente el validar la arquitectura con el código fuente real, de forma que se compruebe si se están realizando accesos/dependencias entre capas no permitidas por la arquitectura, e incluso ligar estas validaciones al proceso de control de código fuente en TFS.

En nuestro ejemplo de aplicación, este sería el diagrama de **Arquitectura N-Layer**:

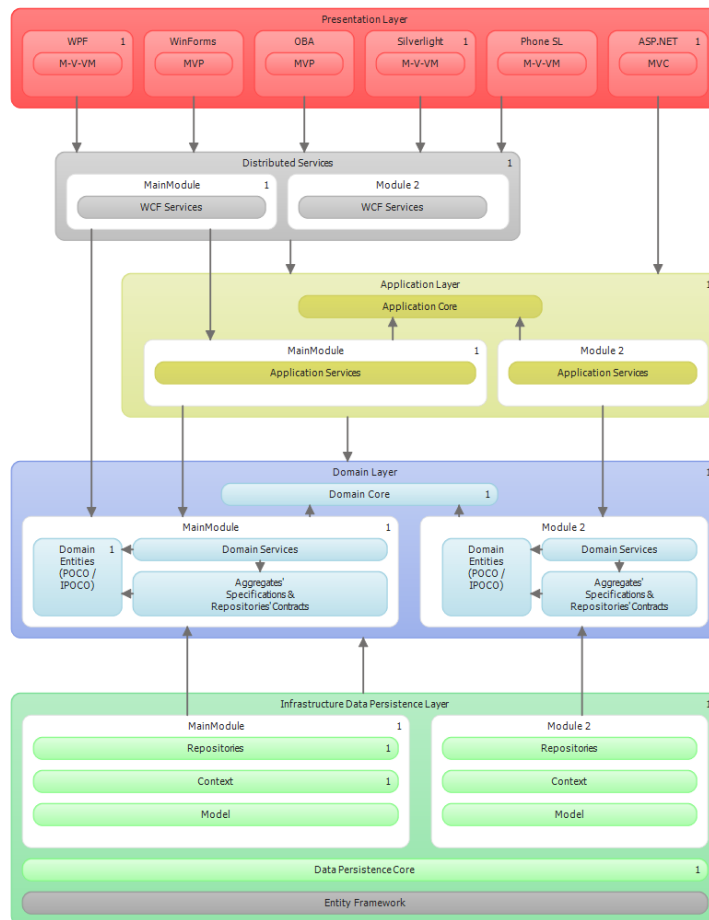


Figura 19.- Arquitectura N-Layer DDD en VS.2010

En los siguientes capítulos de la guía analizamos la lógica e implementación de cada una de estas capas y sub-capas. Sin embargo, resaltamos algunos aspectos globales a continuación.

Como se puede observar en el diagrama de Arquitectura, la Capa Central sobre la que gira toda la Arquitectura, es la Capa de Dominio. Esto es también apreciable a nivel de las dependencias. La mayoría de las dependencias finalizan en la Capa de Dominio (p.e. dependencias con las Entidades del Dominio, etc.). Y la Capa de Dominio, a su vez tiene mínimas dependencias con otras capas (Infraestructura, Persistencia de Datos), y en esos casos, son ‘dependencias desacopladas’, es decir, basadas en abstracciones (interfaces) y a través de contenedores de IoC, por lo que no aparecen esas dependencias de forma directa como ‘flechas’ en el diagrama.

Cabe destacar, que por claridad en el diagrama anterior, no se han especificado todas las dependencias reales de más bajo nivel que tiene la aplicación, ejemplo de la cual se ha obtenido este diagrama de Capas.

Otro aspecto a mencionar es que la Capa de ‘Servicios Remotos’ o ‘Servicios Distribuidos’ (Servicios WCF, en definitiva, en .NET), es una capa opcional dependiendo del tipo de Capa de Presentación a utilizar. Si la capa de presentación se ejecuta en un entorno remoto (Silverlight, WPF, Winforms, OBA, se ejecutan en el ordenador cliente), está claro que será necesario. Pero por ejemplo, en el caso de un cliente Web (ASP.NET o ASP.NET MVC), cabe la posibilidad más normal de que el servidor web de capa de presentación esté en el mismo nivel físico de servidores que los componentes de negocio. En ese caso, no tiene sentido hacer uso de servicios WCF, puesto que impactaría innecesariamente en el rendimiento de la aplicación.

En cuanto a la ‘Capa de Aplicación’, va a ser normalmente nuestra capa ‘Fachada’, donde se exponen los Servicios de Aplicación que coordinan las tareas y acciones a efectuar contra el Dominio así como contra la persistencia y consulta de datos.



2.15.- Implementación de Inyección de Dependencias e IoC con UNITY

En la presente sección se pretende explicar las técnicas y tecnologías para realizar una implementación específica del desacoplamiento entre capas de la Arquitectura. En concreto, explicar las técnicas **DI (Inyección de dependencias)** e **IoC (Inversión de Control)** con una tecnología concreta de **Microsoft Pattern & Practices**, llamada **Unity**.

DI e **IoC** se pueden implementar con diversas tecnologías y frameworks de diferentes fabricantes, como:

Tabla 8.- Implementaciones de Contenedores IoC

Framework	Implementador	Información
Unity http://msdn.microsoft.com/en-us/library/dd203101.aspx http://unity.codeplex.com/	Microsoft Pattern & Practices	Es actualmente el <i>framework</i> ligero de Microsoft más completo para implementar IoC y DI. Es un proyecto <i>OpenSource</i> . Con licenciamiento de tipo Microsoft Public License (Ms-PL)
Castle Project (Castle Windsor) http://www.castleproject.org/	CastleStronghold	Castle es un proyecto OpenSource. Es uno de los mejores frameworks para IoC y DI.
MEF (Microsoft Extensibility Framework) http://code.msdn.microsoft.com/mef http://www.codeplex.com/MEF	Microsoft (Forma parte de .NET 4.0)	Es actualmente un <i>framework</i> para extensibilidad automática de herramientas y aplicaciones, no está tan orientado a desacoplamiento entre Capas de Arquitectura utilizando IoC y DI.
Spring.NET http://www.springframework.net/	SpringSource	Spring.NET es un proyecto OpenSource. Es uno de los mejores frameworks con AOP (Aspect Oriented Programming), ofreciendo también capacidades IoC.

StructureMap http://structuremap.sourceforge.net/Default.htm	Varios desarrolladores de la comunidad .NET	Proyecto OpenSource.
Autofac http://code.google.com/p/autofac/	Varios desarrolladores de la comunidad .NET	Proyecto OpenSource.
LinFu http://code.google.com/p/linfu/downloads/list http://www.codeproject.com/KB/cs/LinFuPart1.aspx	Varios desarrolladores de la comunidad .NET	Proyecto OpenSource. Aporta IoC, AOP y otras características.

Para el ejemplo de aplicación N-Capas de nuestra Arquitectura marco, hemos escogido UNITY por ser actualmente el *framework* IoC y DI más completo ofrecido por Microsoft. Pero por supuesto, en una arquitectura marco empresarial, podría hacerse uso de cualquier *framework* IoC (listado o no en la tabla anterior).



2.15.1.- Introducción a Unity

El *Application Block* denominado **Unity** (implementado por *Microsoft Patterns & Practices*), es un contenedor de inyección de dependencias extensible y ligero (Unity no es un gran framework pesado). Soporta inyección en el constructor, inyección de propiedades, inyección en llamadas a métodos y contenedores anidados.

Básicamente, *Unity* es un contenedor donde podemos registrar tipos (clases, interfaces) y también mapeos entre dichos tipos (como un mapeo de un interfaz hacia una clase) y además el contenedor de *Unity* puede instanciar bajo demanda los tipos concretos requeridos.

Unity está disponible como un *download* público desde el site de Microsoft (es gratuito) y también está incluido en la Enterprise Library 4.0/5.0 y en PRISM (*Composite Applications Framework*), los cuales hacen uso extensivo de *Unity*.

Para hacer uso de *Unity*, normalmente registramos tipos y mapeos en un contenedor de forma que especificamos las dependencias entre interfaces, clases base y tipos concretos de objetos. Podemos definir estos registros y mapeos directamente por código fuente o bien, como normalmente se hará en una aplicación real, mediante

XML de ficheros de configuración. También se puede especificar inyección de objetos en nuestras propias clases haciendo uso de atributos que indican las propiedades y métodos que requieren inyección de objetos dependientes, así como los objetos especificados en los parámetros del constructor de una clase, que se inyectan automáticamente.

Incluso, se puede hacer uso de las extensiones del contenedor que soportan otras cosas como la extensión “*Event Broker*” que implementa un mecanismo de publicación/suscripción basado en atributos, que podemos utilizar en nuestras aplicaciones. Podríamos incluso llegar a construir nuestras propias extensiones de contenedor.

Unity proporciona las siguientes ventajas al desarrollo de aplicaciones:

- Soporta abstracción de requerimientos; esto permite a los desarrolladores el especificar dependencias en tiempo de ejecución o en configuración y simplifica la gestión de aspectos horizontales (*crosscutting concerns*), como puede ser el realizar pruebas unitarias contra *mocks* y *stubs*, o contra los objetos reales de la aplicación.
- Proporciona una creación de objetos simplificada, especialmente con estructuras de objetos jerárquicos con dependencias, lo cual simplifica el código de la aplicación.
- Aumenta la flexibilidad al trasladar la configuración de los componentes al contenedor IoC.
- Proporciona una capacidad de localización de servicios; esto permite a los clientes el guardar o cachear el contenedor. Es por ejemplo especialmente útil en aplicaciones web ASP.NET donde los desarrolladores pueden persistir el contenedor en la sesión o aplicación ASP.NET.



2.15.2.- Escenarios usuales con Unity

Unity resuelve problemas de desarrollo típicos en aplicaciones basadas en componentes. Las aplicaciones de negocio modernas están compuestas por objetos de negocio y componentes que realizan tareas específicas o tareas genéricas dentro de la aplicación, además solemos tener componentes que se encargan de aspectos horizontales de la arquitectura de la aplicación, como pueden ser trazas, logging, autenticación, autorización, cache y gestión de excepciones.

La clave para construir satisfactoriamente dichas aplicaciones de negocio (aplicaciones N-Capas), es conseguir un diseño desacoplado (*decoupled* / *very loosely coupled*). Las aplicaciones desacopladas son más flexibles y fácilmente mantenibles y especialmente son más fáciles de probar durante el desarrollo (Pruebas Unitarias). Se pueden realizar **mocks** (simulaciones) de objetos que tengan fuertes

dependencias concretas, como conexiones a bases de datos, conexiones a red, conexiones a aplicaciones externas como ERPs, etc. De forma que las pruebas unitarias se puedan realizar contra los mocks o contra los objetos reales cambiándolo de una forma dinámica o basado en configuración.

La inyección de dependencias es una técnica fundamental para construir aplicaciones desacopladas. Proporciona formas de gestionar dependencias entre objetos. Por ejemplo, un objeto que procesa información de un cliente puede depender de otros objetos que acceden a la base de datos, validan la información y comprueban que el usuario está autorizado para realizar actualizaciones. Las técnicas de inyección de dependencias pueden asegurar que la clase 'Cliente' instancie y ejecute correctamente dichos objetos de los que depende, especialmente cuando las dependencias son abstractas.



2.15.3.- Patrones Principales

Los siguientes patrones de diseño definen aproximaciones de arquitectura y desarrollo que simplifican el proceso:

Patrón de Inversión de Control (IoC). Este patrón genérico describe técnicas para soportar una arquitectura tipo 'plug-in' donde los objetos pueden buscar instancias de otros objetos que requieren.

Patrón de Inyección de Dependencias (DI). Es realmente un caso especial de IoC. Es una interfaz de técnica de programación basada en alterar el comportamiento de una clase sin cambiar el código interno de la misma. Los desarrolladores codifican contra un interfaz relacionado con la clase y usan un contenedor que inyecta instancias de objetos dependientes en la clase basada en el interfaz o tipo de objeto. Las técnicas de inyección de instancias de objetos son 'inyección de interfaz', 'inyección de constructor', inyección de propiedad (*setter*), e inyección de llamada a método.

Patrón de Intercepción. Este patrón introduce otro nivel de indirección. Esta técnica sitúa un objeto entre el cliente y el objeto real. Se utiliza un proxy entre el cliente y el objeto real. El comportamiento del cliente es el mismo que si interactuara directamente con el objeto real, pero el proxy lo intercepta y resuelve su ejecución colaborando con el objeto real y otros objetos según requiera.



2.15.4.- Métodos principales

Unity expone dos métodos para registrar tipos y mapeos en el contenedor:

RegisterType(): Este método registra un tipo en el contenedor. En el momento adecuado, el contenedor construye una instancia del tipo especificado. Esto puede ser en respuesta a una inyección de dependencias iniciada mediante atributos de clase o cuando se llama al método *Resolve*. El tiempo de vida (*lifetime*) del objeto corresponde

al tiempo de vida que se especifique en los parámetros del método. Si no se especifica valor al *lifetime*, el tipo se registra de forma transitoria, lo que significa que el contenedor crea una nueva instancia en cada llamada al método **Resolve ()**.

RegisterInstance(): Este método registra en el contenedor una instancia existente del tipo especificado, con un tiempo de vida especificado. El contenedor devuelve la instancia existente durante ese tiempo de vida. Si no se especifica un valor para el tiempo de vida, la instancia tiene un tiempo de vida controlada por el contenedor.



2.15.5.- Registro Configurado de tipos en Contenedor

Como ejemplo de uso de los métodos **RegisterType** y **Resolve**, a continuación realizamos un registro de un mapeo de un interfaz llamado **ICustomerService** y especificamos que el contenedor debe devolver una instancia de la clase **CustomerService** (la cual tendrá implementado el interfaz **ICustomerService**).

```
C#
//Registro de tipos en Contenedor de UNITY
IUnityContainer container = new UnityContainer();
container.RegisterType<ICustomerManagementService,
CustomerManagementService>();
...
//Resolución de tipo a partir de Interfaz
ICustomerManagementService customerSrv =
container.Resolve<ICustomerManagementService>();
```

Normalmente en la versión final de aplicación, el registro de clases, interfaces y mapeos en el contenedor, se puede realizar de forma declarativa en el XML de los ficheros de configuración, quedando completamente desacoplado. Sin embargo, tal y como se muestra en las líneas de código anteriores, durante el desarrollo probablemente es más cómodo realizarlo de forma '*Hard-coded*', pues así los errores tipográficos se detectarán en tiempo de compilación en lugar de en tiempo de ejecución (como pasa con el XML).

Con respecto al código anterior, la línea que siempre estará en el código de la aplicación, sería la que instancia propiamente el objeto resolviendo la clase que debe utilizarse mediante el contenedor, es decir, la llamada al método **Resolve()** (Independientemente de si se realiza el registro de tipos por XML o '*Hard-Coded*').



2.15.6.- Inyección de dependencias en el constructor

Como ejemplo de inyección en el constructor, si instanciamos una clase usando el método **Resolve()** del contenedor de Unity, y dicha clase tiene un constructor con uno o

más parámetros (dependencias con otras clases), el contenedor de Unity creará automáticamente las instancias de los objetos dependientes especificados en el constructor.

A modo de ejemplo, tenemos un código inicial que no hace uso de Inyección de Dependencias ni tampoco de Unity. Queremos cambiar esta implementación para que quede desacoplado, utilizando IoC mediante Unity. Tenemos en principio un código que utiliza una clase de negocio llamada **CustomerManagementService**. Es una simple instanciación y uso:

```
C#
...
{
    CustomerManagementService custService =
        new CustomerManagementService ();
    custService.SaveData("0001", "Microsoft", "Madrid");
}
```

Este código es importante tener en cuenta que sería el código a implementar en el inicio de una acción, por ejemplo, sería el código que implementaríamos en un método de un Servicio-Web WCF.

A continuación tenemos la definición de dicha clase de Servicio inicial sin inyección de dependencias (**CustomerManagementService**), que hace uso a su vez de una clase de la capa de acceso a datos, llamada **CustomerRepository** (clase repositorio o de acceso a datos):

```
C#

public class CustomerManagementService
{
    //Members
    private ICustomerRepository _custRepository;
    //Constructor
    public CustomerManagementService ()
    {
        _custRepository = new CustomerRepository();
    }
    public SaveData()
    {
        custRepository.SaveData("0001", "Microsoft", "Madrid");
    }
}
```

Hasta ahora, en el código anterior, no tenemos nada de IoC ni DI, no hay inyección de dependencias ni uso de Unity, es todo código tradicional orientado a objetos. Ahora vamos a modificar la clase de negocio **CustomerManagementService** de forma que la creación de la clase de la que dependemos (**CustomerRepository**) no lo hagamos nosotros, sino que la instanciación de dicho objeto sea hecha automáticamente por el contenedor de Unity. Es decir, tendremos un código haciendo uso de inyección de dependencias en el constructor.

C#

```

public class CustomerManagementService
{
    //Members
    private ICustomerRepository custRepository;
    //Constructor
    public CustomerManagementService (ICustomerRepository
                                     customerRepository)
    {
        custRepository = customerRepository;
    }
    public SaveData()
    {
        _custRepository.SaveData("0001", "Microsoft", "Madrid");
    }
}

```

Es importante destacar que, como se puede observar, no hemos hecho ningún ‘new’ explícito de la clase CustomerRepository. Es el contenedor de Unity el que automáticamente creará el objeto de CustomerRepository y nos lo proporcionará como parámetro de entrada a nuestro constructor. **Esa es precisamente la inyección de dependencias en el constructor.**

En tiempo de ejecución, el código de instanciación de **CustomerManagementService** se realizaría utilizando el método Resolve() del contenedor de Unity, el cual origina la instanciación generada por el *framework* de Unity de la clase CustomerRepository dentro del ámbito de la clase **CustomerManagementService**.

El siguiente código es el que implementaríamos en la capa de primer nivel que consumiría objetos del Dominio. Es decir, sería probablemente la capa de Servicios Distribuidos (WCF) o incluso capa de presentación web ejecutándose en el mismo servidor de aplicaciones (ASP.NET):

C# (En Capa de Servicio WCF ó Capa de Aplicación o en aplicación ASP.NET)

```

...
{
    IUnityContainer container = new UnityContainer();
    CustomerManagementService custService =
        container.Resolve<ICustomerManagementService>();
    custService.SaveData("0001", "Microsoft", "Madrid");
}

```

Como se puede observar en el uso de **Resolve<>()**, en ningún momento hemos creado nosotros una instancia de la clase de la que dependemos (CustomerRepository) y por lo tanto nosotros no hemos pasado explícitamente un objeto de CustomerRepository al constructor de nuestra clase CustomerManagementService. Y sin embargo, cuando se instancie la clase de servicio (CustomerManagementService), automáticamente se nos habrá proporcionado en el constructor una instancia nueva de CustomerRepository. Eso lo habrá hecho precisamente el contenedor de Unity al detectar la dependencia. Esta es la inyección de dependencias, y nos proporciona la

flexibilidad de poder cambiar la dependencia en tiempo de configuración y/o ejecución. Por ejemplo, si en el fichero de configuración hemos especificado que se creen objetos Mock (simulación) en lugar de objetos reales de acceso a datos (Repository), la instanciación de la clase podría haber sido la de **CustomerMockRepository** en lugar de **CustomerRepository** (ambas implementarían el mismo interfaz **ICustomerRepository**).



2.15.7.- Inyección de Propiedades (Property Setter)

A continuación mostramos la inyección de propiedades. En este caso, tenemos una clase llamada **ProductService** que expone como propiedad una referencia a una instancia de otra clase llamada **Supplier** (clase entidad/datos, no definida en nuestro código). Para forzar la inyección de dependencias del objeto dependiente, se debe aplicar el atributo “Dependency” a la declaración de la propiedad, como muestra el siguiente código.

```
c#

public class ProductService
{
    private Supplier supplier;
    [Dependency]
    public Supplier SupplierDetails
    {
        get { return supplier; }
        set { supplier = value; }
    }
}
```

Entonces, al crear una instancia de la clase **ProductService** mediante el contenedor de Unity, automáticamente se generará una instancia de la clase **Supplier** y se establecerá dicho objeto como el valor de la propiedad **SupplierDetails** de la clase **ProductService**.

Para más información sobre ejemplos de programación con Unity, estudiar la documentación y labs de:

Unity 1.2 Hands On Labs

<http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=93a5e18f-3211-44ef-b785-c59bcec4cd6f>

Webcast Demos

<http://unity.codeplex.com/Wiki/View.aspx?title=Webcast%20demos>

MSDN Technical Article & Sample Code

<http://msdn.microsoft.com/en-us/library/cc816062.aspx>



2.15.8.- Resumen de características a destacar de Unity

Unity proporciona los siguientes puntos/características que merece la pena destacar:

- Unity proporciona un mecanismo para construir (o ensamblar) instancias de objetos, los cuales pueden contener otras instancias de objetos dependientes.
- Unity expone métodos “RegisterType<>()” que permiten configurar el contenedor con mapeos de tipos y objetos (incluyendo instancias singleton) y métodos “Resolve<>()” que devuelven instancias de objetos construidos que pueden contener objetos dependientes.
- Unity proporciona **inversión de control (IoC)** permitiendo inyección de objetos preconfigurados en clases construidas por el *application block*. Podemos especificar un interfaz o clase en el constructor (inyección en constructor), o podemos aplicar atributos a propiedades y métodos para iniciar inyección de propiedades e inyección de llamadas a métodos.
- Se soporta jerarquía de contenedores. Un contenedor puede tener contenedores hijo, lo cual permite que las consultas de localización de objetos pasen de los contenedores hijos a los contenedores padre.
- Se puede obtener la información de configuración de sistemas estándar de configuración, como ficheros XML, y utilizarlo para configurar el contenedor.
- No se requiere definiciones específicas en las clases. No hay requerimientos a aplicar a las clases (como atributos), excepto cuando se usa la inyección de llamada a métodos o la inyección de propiedades.
- Unity permite extender las funcionalidades de los contenedores; por ejemplo, podemos implementar métodos que permitan construcciones adicionales de objetos y características de contenedores, como cache.



2.15.9.- Cuándo utilizar Unity

La inyección de dependencias proporciona oportunidades para simplificar el código, abstraer dependencias entre objetos y generar instancias de objetos dependientes de una forma automatizada. Sin embargo, el proceso puede tener un pequeño impacto en el rendimiento (normalmente es insignificante cuando en paralelo tenemos dependencias a recursos externos como bases de datos y consumo de servicios distribuidos, que son

realmente los cuellos de botella de la mayoría de las aplicaciones). En otros casos trabajando solamente con objetos en memoria, sí que podría impactar significativamente en el rendimiento.

Así mismo, también se incrementa algo la complejidad donde simplemente existían dependencias directas.

En general:

Se debe utilizar *Unity* en las siguientes situaciones:

- Tus objetos y clases pueden tener dependencias sobre otros objetos y clases.
- Tus dependencias son complejas o requieren abstracción.
- Se quiere hacer uso de características de inyección en constructor, método o propiedad.
- Se quiere gestionar el tiempo de vida de las instancias de los objetos.
- Se quiere poder configurar y cambiar dependencias en tiempo de ejecución.
- Se quiere realizar pruebas unitarias sobre mocks/stubs.
- Se quiere poder cachear o persistir las dependencias a lo largo de *postbacks* en una aplicación Web.

No se necesita utilizar *Unity* en las siguientes situaciones:

- Tus objetos y clases no tienen dependencias con otros objetos y clases.
- Tus dependencias son muy simples o no requieren abstracción.



3.- ORIENTACIÓN A ARQUITECTURA EDA (EVENT DRIVEN ARCHITECTURE)

EDA (*Event-Driven Architecture*) es un patrón de arquitectura de software que promociona fundamentalmente el uso de eventos (generación, detección, consumo y reacción a eventos) como hilo conductor principal de ejecución de cierta lógica del Dominio. Es un tipo de Arquitectura genérica orientada a eventos, por lo que puede ser implementada con lenguajes de desarrollo multidisciplinar y no es necesario/obligatorio hacer uso de tecnologías especiales (si bien, las tecnologías especialmente diseñadas para implementar *Workflows* y orquestaciones de procesos de negocio, pueden ayudar mucho a esta tendencia de Arquitectura).

En la presente guía de arquitectura, EDA va a incluirse como una posibilidad complementaria, no como algo obligatorio a diseñar e implementar, pues la idoneidad de una fuerte orientación a eventos depende mucho del tipo de aplicación a crear.

Un evento puede definirse como “un cambio significativo de estado”. Por ejemplo, una petición de vacaciones puede estar en estado de “en espera” o de “aprobado”. Un sistema que implemente esta lógica podría tratar este cambio de estado como un evento que se pueda producir, detectar y consumir por varios componentes dentro de la arquitectura.

El patrón de arquitectura EDA puede aplicarse en el diseño y la implementación de aplicaciones que transmitan eventos a lo largo de diferentes objetos (componentes y servicios débilmente acoplados, a ser posible). Un sistema dirigido por eventos normalmente dispondrá de emisores de eventos (denominados también como ‘Agentes’) y consumidores de eventos (denominados también como ‘sumidero’ o *sink*). Los *sinks* tienen la responsabilidad de aplicar una reacción tan pronto como se presente un evento. Esa reacción puede o no ser proporcionada completamente por el propio *sink*. Por ejemplo, el *sink* puede tener la responsabilidad de filtrar, transformar y mandar el evento a otro componente o él mismo puede proporcionar una reacción propia a dicho evento.

El construir aplicaciones y sistemas alrededor del concepto de una orientación a eventos permite a dichas aplicaciones reaccionar de una forma mucho más natural y cercana al mundo real, porque los sistemas orientados a eventos son, por diseño, más orientados a entornos asíncronos y no predecibles (El ejemplo típico serían los *Workflows*, pero no solamente debemos encasillar EDA en *Workflows*).

EDA (Event-Driven Architecture), puede complementar perfectamente a una arquitectura *N-Layer DDD* y a arquitecturas orientadas a servicios (SOA) porque la lógica del dominio y los servicios-web pueden activarse por disparadores relacionados con eventos de entrada. Este paradigma es especialmente útil cuando el *sink* no proporciona él mismo la reacción/ejecución esperada.

Esta ‘inteligencia’ basada en eventos facilita el diseño e implementación de procesos automatizados de negocio así como flujos de trabajo orientados al usuario (*Human Workflows*); incluso es también muy útil para procesos de maquinaria, dispositivos como sensores, actuadores, controladores, etc. que pueden detectar cambios en objetos o condiciones para crear eventos que puedan entonces ser procesados por un servicio o sistema.

Por lo tanto, **se puede llegar a implementar EDA en cualquier área orientada a eventos, bien sean *Workflows*, procesos de reglas del Dominio, o incluso capas de presentación basadas en eventos (como MVP y M-V-VM), etc.**

EDA también está muy relacionado con el patrón **CQRS** (*Command and Query Responsibility Segregation*) que introduciremos posteriormente.

Finalmente, resaltar que en la presente propuesta de Arquitectura, así como en nuestra aplicación ejemplo publicada en *CODEPLEX*, no estamos haciendo uso de **EDA (Event-Driven Architecture)**, simplemente lo introducimos aquí como un aspecto de arquitectura para escenarios avanzados hacia los que se puede evolucionar. Es también posible que en siguientes versiones lleguemos a evolucionar esta arquitectura hacia **EDA**.



4.- ACCESO DUAL A FUENTES DE DATOS

En la mayoría de los sistemas, los usuarios necesitan ver datos y realizar todo tipo de búsquedas, ordenaciones y filtros, al margen de las operaciones transaccionales y/o de actualización de datos.

Para realizar dichas consultas cuyo objetivo es únicamente visualizar (informes, consultas, etc.), podríamos hacer uso de las mismas clases de lógica del dominio y repositorios de acceso a datos relacionados que utilizamos para operaciones transaccionales (en muchas aplicaciones lo haremos así), sin embargo, si se busca la máxima optimización y rendimiento, probablemente esta no sea la mejor opción.

En definitiva, mostrar información al usuario no está ligado a la mayoría de los comportamientos del dominio (reglas de negocio), ni a problemáticas de tipos de concurrencia en las actualizaciones (Gestión de Concurrencia Optimista ni su gestión de excepciones), ni por lo tanto tampoco a entidades desconectadas *self-tracking*, necesarias para la gestión de concurrencia optimista, etc. Todas estas problemáticas impactan en definitiva en el rendimiento puro de las consultas y lo único que queremos realmente hacer en este caso es realizar consultas con muy buen rendimiento. Incluso aunque tengamos requerimientos de seguridad u otro tipo que sí estén también relacionados con las consultas puras de datos (informes, listados, etc.), esto también se puede implementar en otro sitio.

Por supuesto que se puede hacer uso de un único modelo y acceso a las fuentes de datos, pero a la hora de escalar y optimizar al máximo el rendimiento, esto no se podrá conseguir. En definitiva *“un cuchillo está hecho para la carne y una cuchara para la sopa”*. Con mucho esfuerzo podremos cortar carne con una cuchara, pero no es lo más óptimo, ni mucho menos.

Es bastante normal que los Arquitectos de Software y los Desarrolladores definan ciertos requerimientos, a veces innecesarios, de una forma inflexible, y que además, a nivel de negocio no se necesitan. Este es probablemente uno de esos casos. La decisión de utilizar las entidades del modelo del dominio para solo mostrar información (solo visualización, informes, listados, etc.) es realmente algo auto-impuesto por los desarrolladores o arquitectos, pero no tiene por qué ser así.

Otro ejemplo diferente, es el hecho de que en muchos sistemas multiusuario, los cambios no tienen por qué ser visibles inmediatamente al resto de los usuarios. Si esto es así, ¿por qué hacer uso del mismo dominio, repositorios y fuentes de datos transaccionales?. Si no se necesita del comportamiento de esos dominios, ¿por qué pasar a través de ellos?. Es muy posible, por ejemplo, que las consultas (para informes, y consultas solo visualización) sean mucho más óptimas en muchos casos si se utiliza una segunda base de datos basada en cubos, BI (p.e. SQL Server OLAP, etc.) y que para acceder a ello se utilice el mecanismo más sencillo y ligero para realizar consultas (una simple librería de acceso a datos, probablemente para conseguir el máximo rendimiento, el mejor camino no sea un ORM.).

En definitiva, en algunos sistemas, la mejor arquitectura podría estar basada en dos pilares internos de acceso a datos:

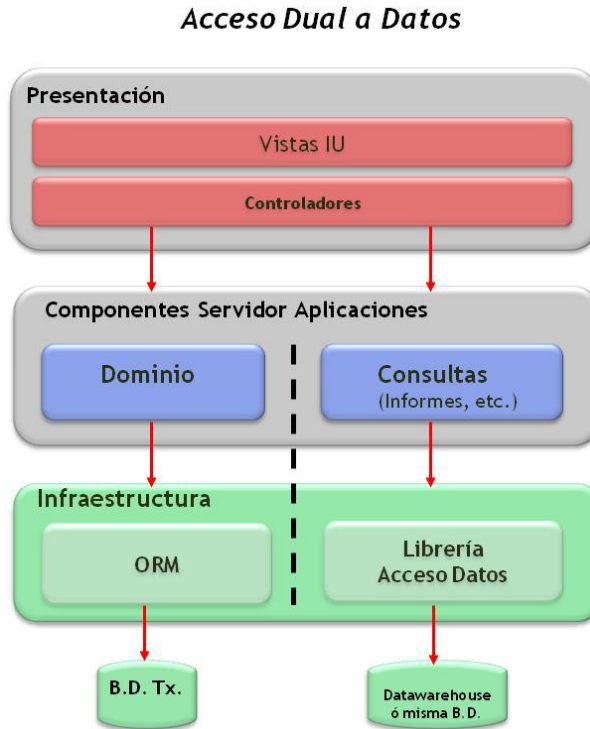


Figura 20.- Acceso Dual a Datos

Lo importante a resaltar de este modelo/arquitectura es que el pilar de la derecha se utiliza solo para consultas puras (informes, listados, visualizaciones). En cambio, el pilar de la izquierda (Dominio+ORM) seguirá realizando consultas para casos en los que dichos datos consultados pueden ser modificados por el usuario, utilizando *databinding*, etc.

Así mismo, la viabilidad de disponer o no de una base de datos diferente (incluso de tipo diferente, relacional vs. cubos), depende mucho de la naturaleza de la aplicación, pero en caso de ser viable, es la mejor opción, pues las escrituras no interferirán nunca con las 'solo lecturas', esto finalmente maximiza al máximo la escalabilidad y el rendimiento de cada tipo de operación. Sin embargo, en este caso se requerirá de algún tipo de sincronización de datos entre las diferentes bases de datos.

En definitiva, el objetivo final es *“situar todo el código en cada parte adecuada del sistema, de una forma granularizada, focalizada y que se pueda probar de forma automatizada”*.



5.- NÍVELES FÍSICOS EN DESPLIEGUE (*TIER*S)

Los Niveles representan separaciones físicas de las funcionalidades de presentación, negocio y datos de nuestro diseño en diferentes máquinas, como servidores (para lógica de negocio y bases de datos) y otros sistemas (PCs para capas de presentación remotas, etc.). Patrones de diseño comunes basados en niveles son “2-Tier”, “3-Tier” y “N-Tier”.

2-Tier

Este patrón representa una estructura básica con dos niveles principales, un nivel cliente y un servidor de bases de datos. En un escenario web típico, la capa de presentación cliente y la lógica de negocio co-existen normalmente en el mismo servidor, el cual accede a su vez al servidor de bases de datos. Así pues, en escenarios Web, el nivel cliente suele contener tanto la capa de presentación como la capa de lógica de negocio, siendo importante por mantenibilidad que se mantengan dichas capas lógicas internamente.

Arquitectura ‘2-Tier’ (Cliente-Servidor)

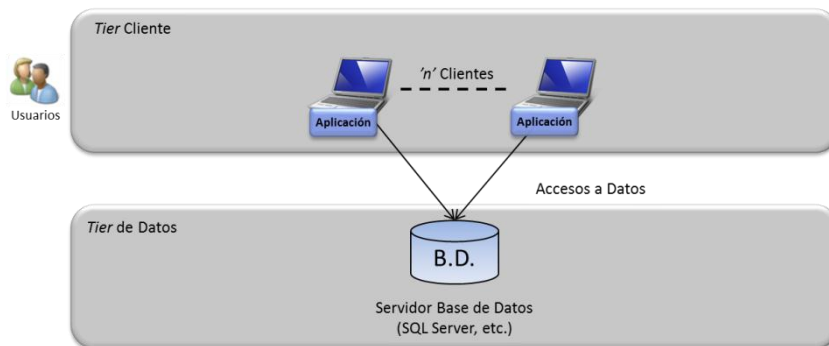


Figura 21.- Nivel/Tier Cliente

3-Tier

En un diseño de patrón “3-Tier”, el usuario interacciona con una aplicación cliente desplegada físicamente en su máquina (PC, normalmente). Dicha aplicación cliente se comunica con un servidor de aplicaciones (Tier Web/App) que tendrá embebidas las capas lógicas de lógica de negocio y acceso a datos. Finalmente, dicho servidor de

aplicaciones accede a un tercer nivel (Tier de datos) que es el servidor de bases de datos. Este patrón es muy común en todas las aplicaciones Rich-Client, RIA y OBA. También en escenarios Web, donde el cliente sería ‘pasivo’, es decir, un simple navegador.

El siguiente gráfico ilustra este patrón “**3-Tier**” de despliegue:

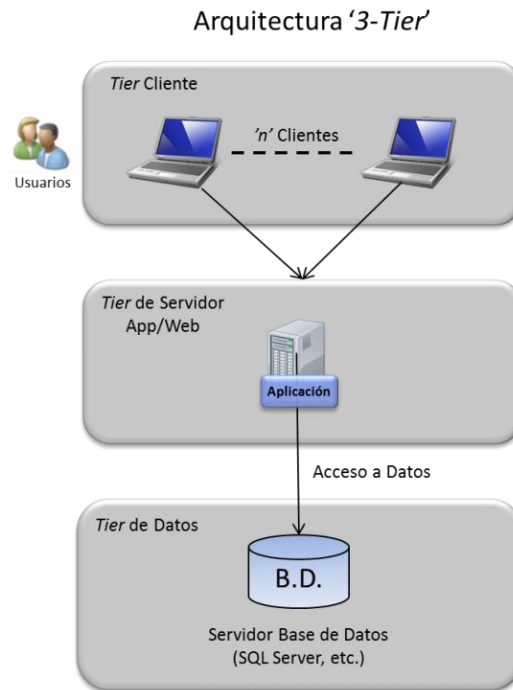
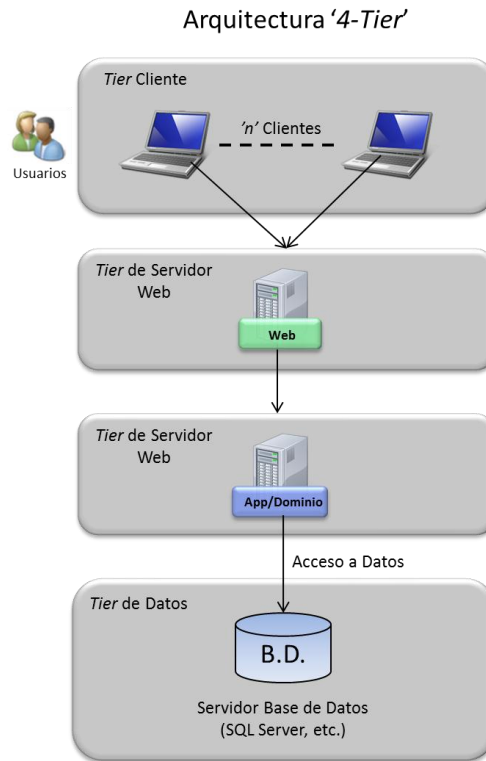


Figura 22.- Nivel/Tier Cliente

N-Tier

En este escenario, el servidor Web (que contiene la capa de lógica de presentación) se separa físicamente del servidor de aplicaciones que implementa ya exclusivamente lógica de negocio y acceso a datos. Esta separación se suele hacer normalmente por razones de políticas de seguridad de redes, donde el servidor Web se despliega en una red perimetral y accede al servidor de aplicaciones que está localizado en una subred diferente, separados probablemente por un firewall. También es común que exista un segundo *firewall* entre el nivel cliente y el nivel Web.

La siguiente figura ilustra el patrón de despliegue “**N-Tier**”:

**Figura 23.- Nivel/Tier Cliente**

Elección de niveles/tiers en la arquitectura

La elección de niveles/*tiers* separando capas lógicas de nuestra aplicación en niveles físicos separados, impacta en el rendimiento de las aplicaciones (debido a la latencia de las comunicaciones remotas entre los diferentes niveles), si bien, puede beneficiar a la escalabilidad al distribuir la carga entre diferentes servidores. También puede mejorar la seguridad al separar los componentes más sensibles de la aplicación a diferentes redes. Sin embargo, hay que tener siempre presente que la adición de niveles/*tiers* incrementa la complejidad de los despliegues y en ocasiones impacta sobre el rendimiento, por lo que no se deben añadir más niveles de los necesarios.

En la mayoría de los casos, se debe localizar todo el código de la aplicación en un mismo servidor o mismo nivel de servidores balanceados. Siempre que se haga uso de comunicaciones remotas, el rendimiento se ve afectado por la latencia de las comunicaciones así como por el hecho de que los datos deben serializarse para viajar por la red. Sin embargo, en algunos casos podemos necesitar dividir funcionalidad en diferentes niveles de servidores a causa de restricciones de seguridad o requerimientos de escalabilidad. En esos casos, siempre es deseable elegir protocolos de comunicación optimizados para maximizar el rendimiento (TCP vs. HTTP, etc.).

.....

Considera el patrón “2-Tier” si:

- **Aplicación Web.** Se quiere desarrollar una aplicación Web típica, con el máximo rendimiento y sin restricciones de seguridad de redes. Si se requiere aumentar la escalabilidad, se clonaría el Servidor Web en múltiples servidores balanceados.
- **Aplicación Cliente-Servidor.** Se quiere desarrollar una aplicación cliente-servidor que acceda directamente a un servidor de bases de datos. Este escenario es muy diferente, pues todas las capas lógicas estarían situadas en un nivel cliente que en este caso sería el PC cliente. Esta arquitectura es útil cuando se requiere un rendimiento muy alto y accesos rápidos a la base de datos, sin embargo, las arquitecturas cliente-servidor ofrecen muchos problemas de escalabilidad y sobre todo de mantenimiento y detección de problemas, pues se mueve toda la lógica de negocio y acceso a datos al nivel del PC cliente del usuario, estando a merced de las diferentes configuraciones de cada usuario final. Este caso no se recomienda en la mayoría de las ocasiones.

Considera el patrón “3-Tier” si:

- Se quiere desarrollar una aplicación “3-Tier” con cliente remoto ejecutándose en la máquina cliente de los usuarios (“Rich-Client”, RIA, OBA, etc.) y un servidor de aplicaciones con servicios web publicando la lógica de negocio.
- Todos los servidores de aplicación pueden estar localizados en la misma red.
- Se está desarrollando una aplicación de tipo ‘intranet’ donde los requerimientos de seguridad no exigen separar la capa de presentación de las capas de negocio y acceso a datos.
- Se quiere desarrollar una aplicación Web típica, con el máximo rendimiento.

Considera el patrón “N-Tier” si:

- Existen requerimientos de seguridad que exigen que la lógica de negocio no pueda desplegarse en la red perimetral donde están situados los servidores de capa de presentación.
- Se tienen código de aplicación muy pesado (hace uso intensivo de los recursos del servidor) y para mejorar la escalabilidad se separa dicha funcionalidad de componentes de negocio a otro nivel de servidores.

