

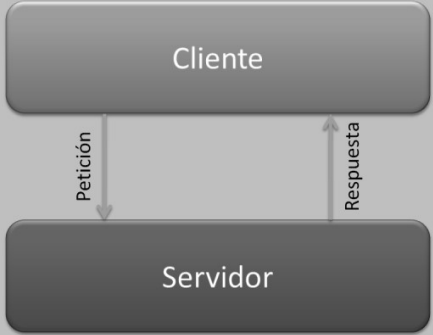
Estilos Arquitecturales

Los estilos arquitecturales son la herramienta básica de un arquitecto a la hora de dar forma a la arquitectura de una aplicación. Un estilo arquitectural se puede entender como un conjunto de principios que definen a alto nivel un aspecto de la aplicación. Un estilo arquitectural viene definido por un conjunto de componentes, un conjunto de conexiones entre dichos componentes y un conjunto de restricciones sobre cómo se comunican dos componentes cualesquiera conectados. Los estilos arquitecturales pueden organizarse en torno al aspecto de la aplicación sobre el que se centran. Los principales aspectos son: comunicaciones, despliegue, dominio, interacción y estructura.

Lo normal en una arquitectura es que no se base en un solo estilo arquitectural, **sino que combine varios de dichos estilos arquitecturales para obtener las ventajas de cada uno**. Es importante entender que los estilos arquitecturales son indicaciones abstractas de cómo dividir en partes el sistema y de cómo estas partes deben interactuar. En las aplicaciones reales los estilos arquitecturales se “instancian” en una serie de componentes e interacciones concretas. Esta es la principal diferencia existente entre un estilo arquitectural y un patrón de diseño. Los patrones de diseño son descripciones estructurales y funcionales de cómo resolver de forma concreta un determinado problema mediante orientación a objetos, mientras que los estilos arquitecturales son descripciones abstractas y no están atados a ningún paradigma de programación específico.

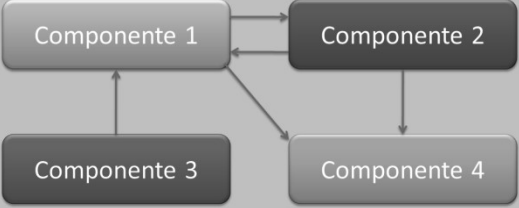
A continuación presentamos los principales estilos arquitecturales con su descripción, características, principios, beneficios y los factores que pueden determinar su uso.

Tabla 1.- Estilo Arquitectural Cliente/Servidor

Estilo Arquitectural	Cliente/Servidor	
Descripción	<p>El estilo cliente/servidor define una relación entre dos aplicaciones en las cuales una de ellas (cliente) envía peticiones a la otra (servidor fuente de datos).</p>	
Características	<ul style="list-style-type: none"> • Es un estilo para sistemas distribuidos. • Divide el sistema en una aplicación cliente, una aplicación servidora y una red que las conecta. • Describe una relación entre el cliente y el servidor en la cual el primero realiza peticiones y el segundo envía respuestas. • Puede usar un amplio rango de protocolos y formatos de datos para comunicar la información. 	
Principios Clave	<ul style="list-style-type: none"> • El cliente realiza una o más peticiones, espera por las respuestas y las procesa a su llegada. • El cliente normalmente se conecta solo a uno o a un número reducido de servidores al mismo tiempo. • El cliente interactúa directamente con el usuario, por ejemplo a través de una interfaz gráfica. 	

	<ul style="list-style-type: none">• El servidor no realiza ninguna petición al cliente.• El servidor envía los datos en respuesta a las peticiones realizadas por los clientes conectados.• El servidor normalmente autentifica y verifica primero al usuario y después procesa la petición y envía los resultados.
Beneficios	<ul style="list-style-type: none">• Más seguridad ya que los datos se almacenan en el servidor que generalmente ofrece más control sobre la seguridad.• Acceso centralizado a los datos que están almacenados en el servidor lo que facilita su acceso y actualización.• Facilidad de mantenimiento ya que los roles y las responsabilidades se distribuyen entre los distintos servidores a través de la red lo que permite que un cliente no se vea afectado por un error en un servidor particular.
Cuando usarlo	<ul style="list-style-type: none">• La aplicación se basa en un servidor y soportará múltiples clientes.• La aplicación está normalmente limitada a un uso local y área LAN controlada.• Estás implementando procesos de negocio que se usarán a lo largo de toda tu organización.• Quieres centralizar el almacenamiento, backup y mantenimiento de la aplicación.• La aplicación debe soportar distintos tipos de clientes y distintos dispositivos.


Tabla 2.- Estilo Arquitectural Basado en componentes

Estilo Arquitectural	Basado en componentes	
Descripción	<p>El estilo de arquitectura basada en componentes describe un acercamiento al diseño de sistemas como un conjunto de componentes que exponen interfaces bien definidas y que colaboran entre sí para resolver el problema.</p>	 <pre> graph TD C1[Componente 1] <--> C2[Componente 2] C3[Componente 3] --> C1 C2 --> C4[Componente 4] C1 --> C4 </pre>
Características	<ul style="list-style-type: none"> • Es un estilo para diseñar aplicaciones a partir de componentes individuales. • Enfatiza la descomposición del sistema en componentes con interfaces muy bien definidas. • Define una aproximación al diseño a través de componentes que se comunican mediante interfaces que exponen métodos, eventos y propiedades. 	
Principios Clave	<ul style="list-style-type: none"> • Los componentes son diseñados de forma que puedan ser reutilizados en distintos escenarios en distintas aplicaciones aunque algunos componentes son diseñados para una tarea específica. • Los componentes son diseñados para operar en diferentes entornos y contextos. Toda la información debe ser pasada al componente en lugar de incluirla en él o que este acceda a ella. • Los componentes pueden ser extendidos a partir de otros componentes para ofrecer nuevos 	

	<p>comportamientos.</p> <ul style="list-style-type: none">• Los componentes exponen interfaces que permiten al código usar su funcionalidad y no revelan detalles internos de los procesos que realizan o de su estado.• Los componentes están diseñados para ser lo más independientes posible de otros componentes, por lo que pueden ser desplegados sin afectar a otros componentes o sistemas.
Beneficios	<ul style="list-style-type: none">• Fácil despliegue ya que se puede sustituir un componente por su nueva versión sin afectar a otros componentes o al sistema.• Reducción de costes ya que se pueden usar componentes de terceros para abaratar los costes de desarrollo y mantenimiento.• Reusables ya que son independientes del contexto se pueden emplear en otras aplicaciones y sistemas.• Reducción de la complejidad gracias al uso de contenedores de componentes que realizan la activación, gestión del ciclo de vida, etc.
Cuando usarlo	<ul style="list-style-type: none">• Tienes componentes que sirvan a tu sistema o los puedes conseguir.• La aplicación ejecutará generalmente procedimientos con pocos o ningún dato de entrada.• Quieres poder combinar componentes escritos en diferentes lenguajes.• Quieres crear una arquitectura que permita

reemplazar o actualizar uno de sus componentes de forma sencilla.

Tabla 3.- Estilo Arquitectural En Capas (N-Layer)

Estilo Arquitectural	En Capas (<i>N-Layer</i>)
Descripción	<p>El estilo arquitectural en capas se basa en una distribución jerárquica de los roles y las responsabilidades para proporcionar una división efectiva de los problemas a resolver. Los roles indican el tipo y la forma de la interacción con otras capas y las responsabilidades la funcionalidad que implementan.</p> 
Características	<ul style="list-style-type: none"> • Descomposición de los servicios de forma que la mayoría de interacciones ocurre solo entre capas vecinas. • Las capas de una aplicación pueden residir en la misma máquina o pueden estar distribuidos entre varios equipos. • Los componentes de cada capa se comunican con los componentes de otras capas a través de interfaces bien conocidos. • Cada nivel agrega las responsabilidades y abstracciones del nivel inferior.


Principios Clave	<ul style="list-style-type: none">• Muestra una vista completa del modelo y a la vez proporciona suficientes detalles para entender las relaciones entre capas.• No realiza ninguna suposición sobre los tipos de datos, métodos, propiedades y sus implementaciones.• Separa de forma clara la funcionalidad de cada capa. <ul style="list-style-type: none">• Cada capa contiene la funcionalidad relacionada solo con las tareas de esa capa.• Las capas inferiores no tienen dependencias de las capas superiores.• La comunicación entre capas está basada en una abstracción que proporciona un bajo acoplamiento entre capas.
Beneficios	<ul style="list-style-type: none">• Abstracción ya que los cambios se realizan a alto nivel y se puede incrementar o reducir el nivel de abstracción que se usa en cada capa del modelo.• Aislamiento ya que se pueden realizar actualizaciones en el interior de las capas sin que esto afecte al resto del sistema.• Rendimiento ya que distribuyendo las capas en distintos niveles físicos se puede mejorar la escalabilidad, la tolerancia a fallos y el rendimiento.• Testeabilidad ya que cada capa tiene una interfaz bien definida sobre la que realizar las pruebas y la habilidad de cambiar entre diferentes implementaciones de una capa.• Independencia ya que elimina la necesidad de considerar el hardware y el despliegue así como las

dependencias con interfaces externas.

Cuando usarlo


- Ya tienes construidas capas de una aplicación anterior, que pueden reutilizarse o integrarse.
 - Ya tienes aplicaciones que exponen su lógica de negocio a través de interfaces de servicios.
 - La aplicación es compleja y el alto nivel de diseño requiere la separación para que los distintos equipos puedan concentrarse en distintas áreas de funcionalidad.
-
- La aplicación debe soportar distintos tipos de clientes y distintos dispositivos.
 - Quieres implementar reglas y procesos de negocio complejos o configurables.

Tabla 4.- Estilo Arquitectural Presentación Desacoplada

Estilo Arquitectural	Presentación Desacoplada
Descripción	<p>El estilo de presentación separada indica cómo debe realizarse el manejo de las acciones del usuario, la manipulación de la interfaz y los datos de la aplicación. Este estilo separa los componentes de la interfaz del flujo de datos y de la manipulación.</p>  <pre> graph TD A[Vista de Interfaz] <--> B[Lógica de Presentación] B <--> C[Lógica de Negocio] </pre>
Características	<ul style="list-style-type: none"> • Es un estilo, para diseñar aplicaciones, basado en patrones de diseño conocidos. • Separa la lógica para el manejo de la interacción de la representación de los datos con que trabaja el usuario. • Permite a los diseñadores crear una interfaz gráfica mientras los desarrolladores escriben el código para su funcionamiento. • Ofrece un mejor soporte para el testeo ya que se pueden testear los comportamientos individuales.
Principios Clave	<ul style="list-style-type: none"> • El estilo de presentación desacoplada separa el procesamiento de la interfaz en distintos roles. • Permite construir <i>mocks</i> que replican el comportamiento de otros objetos durante el testeo. • Usa eventos para notificar a la vista cuando hay datos

	<p>del modelo que han sido modificados.</p> <ul style="list-style-type: none">• El controlador maneja los eventos disparados desde los controles de usuario en la vista.
Beneficios	<ul style="list-style-type: none">• <i>Testeabilidad</i> ya que en las implementaciones comunes los roles son simplemente clases que pueden ser testeadas y reemplazadas por <i>mocks</i> que simulen su comportamiento.• Reusabilidad ya que los controladores pueden ser aprovechados en otras vistas compatibles y las vistas pueden ser aprovechadas en otros controladores compatibles.
Cuando usarlo	<ul style="list-style-type: none">• Quieres mejorar el testeo y el mantenimiento de la funcionalidad de la interfaz.• Quieres separar la tarea de crear la interfaz de la lógica que la maneja.• No quieres que la Interfaz contenga ningún código de procesamiento de eventos.• El código de procesamiento de la interfaz no implementa ninguna lógica de negocio.

Tabla 5.- Estilo Arquitectural N-Niveles (N-Tier)

Estilo Arquitectural	Presentación Desacoplada
Descripción	<p>El estilo arquitectural de N-Niveles define la separación de la funcionalidad en segmentos/niveles físicos separados, similar que el estilo en N-Capas pero sitúa cada segmento en una máquina distinta. En este caso hablamos de Niveles físicos (Tiers)</p> 
Características	<ul style="list-style-type: none"> • Separación de niveles físicos (Servidores normalmente) por razones de escalabilidad, seguridad, o simplemente necesidad (p.e. si la aplicación cliente se ejecuta en máquinas remotas, la Capa de Presentación necesariamente se ejecutará en un Nivel físico separado).
Principios Clave	<ul style="list-style-type: none"> • Es un estilo para definir el despliegue de las capas de la aplicación. • Se caracteriza por la descomposición funcional de las aplicaciones, componentes de servicio y su despliegue distribuido, que ofrece mejor escalabilidad, disponibilidad, rendimiento, manejabilidad y uso de recursos. • Cada nivel es completamente independiente de los

	<p>otros niveles excepto del inmediatamente inferior.</p> <ul style="list-style-type: none">• Este estilo tiene al menos 3 niveles lógicos o capas separados. Cada capa implementa funcionalidad específica y está físicamente separada en distintos servidores.• Una capa es desplegada en un nivel si uno o más servicios o aplicaciones dependen de la funcionalidad expuesta por dicha capa.
Beneficios	<ul style="list-style-type: none">• Mantenibilidad ya que cada nivel es independiente de los otros las actualizaciones y los cambios pueden ser llevados a cabo sin afectar a la aplicación como un todo.• Escalabilidad porque los niveles están basados en el despliegue de capas realizar el escalado de la aplicación es bastante directo.• Disponibilidad ya que las aplicaciones pueden redundar cualquiera de los niveles y ofrecer así tolerancia a fallos.
Cuando usarlo	<ul style="list-style-type: none">• Los requisitos de procesamiento de las capas de la aplicación difieren.• Los requisitos de seguridad de las capas de la aplicación difieren.• Quieres compartir la lógica de negocio entre varias aplicaciones.• Tienes el suficiente hardware para desplegar el número necesario de servidores en cada nivel.

Tabla 6.- Estilo Arquitectural Arquitectura Orientada al Dominio (DDD)

Estilo Arquitectura I	Arquitectura Orientada al Dominio (DDD)
Descripción	<p>DDD (<i>Domain Driven Design</i>) no es solo un estilo arquitectural, es también una forma de afrontar los proyectos a nivel de trabajo del equipo de desarrollo, Ciclo de vida del proyecto, identificación del ‘Lenguaje Ubicuo’ a utilizar con los Expertos en el negocio, etc. Sin embargo, DDD también identifica una serie de patrones de diseño y estilo de Arquitectura concreto.</p>
	<p>DDD es también, por lo tanto, una aproximación concreta para diseñar software basándonos sobre todo en la importancia del Dominio del Negocio, sus elementos y comportamientos y las relaciones entre ellos. Está muy indicado para diseñar e implementar aplicaciones empresariales complejas donde es fundamental definir un Modelo de Dominio expresado en el propio lenguaje de los expertos del Dominio de negocio real (el llamado Lenguaje Ubicuo). El modelo de Dominio puede verse como un marco dentro del cual se debe diseñar la aplicación.</p>

Características	<p>DDD identifica una serie de patrones de Arquitectura importantes a tener en cuenta en el Diseño de una aplicación, como son:</p> <ul style="list-style-type: none">• Arquitectura N-Capas (Capas específicas tendencias arquitectura DDD)• Patrones de Diseño:<ul style="list-style-type: none">○ Repository○ Entity○ Aggregate○ Value-Object○ Unit Of Work○ Services• En DDD también es fundamental el desacoplamiento entre componentes.
Principios Clave	<p>Para aplicar DDD se debe de tener un buen entendimiento del Dominio de Negocio que se quiere modelar, o conseguir ese conocimiento adquiriéndolo a partir de los expertos del Dominio real. Todo el equipo de desarrollo debe de tener contacto con los expertos del dominio (expertos funcionales) para modelar correctamente el Dominio. En DDD, los Arquitectos, desarrolladores, Jefe de proyecto y Testers (todo el equipo) deben acordar hacer uso de un único lenguaje sobre el Dominio del Negocio que esté centrado en cómo los expertos de dicho dominio articulan su trabajo. No debemos implementar nuestro propio lenguaje/términos internos de aplicación.</p> <p>El corazón del software es el Modelo del Dominio el cual es una proyección directa de dicho lenguaje acordado (Lenguaje Ubicuo) y permite al equipo de desarrollo el encontrar rápidamente áreas incorrectas en el software al analizar el lenguaje que lo rodea. La creación de dicho lenguaje común no es simplemente un ejercicio de aceptar información de los</p>

expertos del dominio y aplicarlo. A menudo, los problemas en la comunicación de requerimientos funcionales no es solo por mal entendidos del lenguaje del Dominio, también deriva del hecho de que dicho lenguaje sea ambiguo.

Es importante destacar que aunque DDD proporciona muchos beneficios técnicos, como mantenibilidad, debe aplicarse solamente en dominios complejos donde el modelo y los procesos lingüísticos proporcionan claros beneficios en la comunicación de información compleja y en la formulación de un entendimiento común del Dominio. Así mismo, la complejidad Arquitectural es también mayor que una aplicación orientada a datos, si bien ofrece una mantenibilidad y desacoplamiento entre componentes mucho mayor.

DDD ofrece los siguientes beneficios:

Beneficios

- **Comunicación:** Todas las partes de un equipo de desarrollo pueden usar el modelo de dominio y las entidades que define para comunicar conocimiento del negocio y requerimientos, haciendo uso de un lenguaje común.
- **Extensibilidad:** La Capa del Dominio es el corazón del software y por lo tanto estará completamente desacoplada de las capas de infraestructura, siendo más fácil así extender/evolucionar la tecnología del software
- **Mejor Testing:** La Arquitectura DDD también facilita el Testing y Mocking, debido a que la tendencia de diseño es a desacoplar los objetos de las diferentes capas de la Arquitectura, lo cual facilita el Mocking y Testing correctos.

Cuando usarlo

- Considerar DDD en aplicaciones complejas con mucha lógica de negocio, con Dominios complejos y donde se desea mejorar la comunicación y minimizar los malos entendidos en la comunicación del equipo de desarrollo.
- DDD es también una aproximación ideal en escenarios

empresariales grandes y complejos que son difíciles de manejar con otras técnicas.

- La presente Guía de Arquitectura N-Capas, está especialmente orientada a las tendencias de Arquitectura en DDD.
- Para tener más información sobre el proceso de trabajo del equipo de desarrollo (Ciclo de Vida en DDD, etc.), ver:
 - “Domain Driven Design Quickly” en <http://www.infoq.com/minibooks/domain-driven-design-quickly>
 - “Domain-Driven Design: Tackling Complexity in the Heart of Software” por Eric Evans (Addison-Wesley, ISBN: 0-321-12521-5)
 - “Applying Domain-Driven Design and Patterns” by Jimmy Nilsson

Tabla 7.- Estilo Arquitectural Orientado a Objetos

Estilo Arquitectural	Orientado a Objetos
Descripción	<p>El estilo orientado a objetos es un estilo que define el sistema como un conjunto de objetos que cooperan entre sí en lugar de como un conjunto de procedimientos. Los objetos son discretos, independientes y poco acoplados, se comunican mediante interfaces y permiten enviar y recibir mensajes.</p>



Características	<ul style="list-style-type: none">• Es un estilo para diseñar aplicaciones basado en un número de unidades lógicas y código reusable.• Describe el uso de objetos que contienen los datos y el comportamiento para trabajar con esos datos y además tienen un rol o responsabilidad distinta.• Hace hincapié en la reutilización a través de la encapsulación, la modularidad, el polimorfismo y la herencia.• Contrasta con el acercamiento procedimental donde hay una secuencia predefinida de tareas y acciones. El enfoque orientado a objetos emplea el concepto de objetos interactuando unos con otros para realizar las tareas.
Principios Clave	<ul style="list-style-type: none">• Permite reducir una operación compleja mediante generalizaciones que mantienen las características de la operación.• Los objetos se componen de otros objetos y eligen esconder estos objetos internos de otras clases o exponerlos como simples interfaces.• Los objetos heredan de otros objetos y usan la funcionalidad de estos objetos base o redefinen dicha funcionalidad para implementar un nuevo comportamiento, la herencia facilita el mantenimiento y la actualización ya que los cambios se propagan del objeto base a los herederos automáticamente.• Los objetos exponen la funcionalidad solo a través de métodos, propiedades y eventos y ocultan los detalles internos como el estado o las referencias a otros objetos. Esto facilita la actualización y el reemplazo de objetos asumiendo que sus interfaces son compatibles sin afectar al resto de objetos.

	<ul style="list-style-type: none">• Los objetos son polimórficos, es decir, en cualquier punto donde se espere un objeto base se puede poner un objeto heredero.• Los objetos se desacoplan de los objetos que los usan implementando una interfaz que los objetos que los usan conocen. Esto permite proporcionar otras implementaciones sin afectar a los objetos consumidores de la interfaz.
Beneficios	<ul style="list-style-type: none">• Comprensión ya que el diseño orientado a objetos define una serie de componentes mucho más cercanos a los objetos del mundo real.• Reusabilidad ya que el polimorfismo y la abstracción permiten definir contratos en interfaces y cambiar las implementaciones de forma transparente.• Testeabilidad gracias a la encapsulación de los objetos.• Extensibilidad gracias a la encapsulación, el polimorfismo y la abstracción.
Cuando usarlo	<ul style="list-style-type: none">• Quieres modelar la aplicación basándote en objetos reales y sus acciones.• Ya tienes objetos que encajan en el diseño y con los requisitos operacionales.• Necesitas encapsular la lógica y los datos juntos de forma transparente.

Tabla 8.- Estilo Arquitectural Orientación a Servicios (SOA)

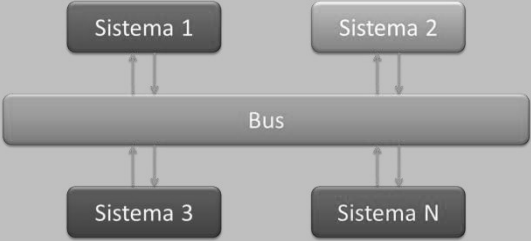
Estilo Arquitectural	Orientación a Servicios (SOA)
Descripción	<p>El estilo orientado a servicios permite a una aplicación ofrecer su funcionalidad como un conjunto de servicios para que sean consumidos. Los servicios usan interfaces estándares que pueden ser invocadas, publicadas y descubiertas. Se centran en proporcionar un esquema basado en mensajes con operaciones de nivel de aplicación y no de componente o de objeto.</p>
Características	<ul style="list-style-type: none"> • La interacción con el servicio está muy desacoplada. • Puede empaquetar procesos de negocio como servicios. • Los clientes y otros servicios pueden acceder a servicios locales corriendo en el mismo nivel. • Los clientes y otros servicios acceden a los servicios remotos a través de la red. • Puede usar un amplio rango de protocolos y formatos de datos.
Principios Clave	<ul style="list-style-type: none"> • Los servicios son autónomos, es decir cada servicio se mantiene, se desarrolla, se despliega y se versiona independientemente. • Los servicios pueden estar situados en cualquier nodo de una red local o remota mientras esta soporte los protocolos de comunicación necesarios. • Cada servicio es independiente del resto de servicios y puede ser reemplazado o actualizado sin afectar a las aplicaciones que lo usan mientras la interfaz sea compatible. • Los servicios comparten esquemas y contratos para

	<p>comunicarse, no clases.</p> <ul style="list-style-type: none">• La compatibilidad se basa en políticas que definen funcionalidades como el mecanismo de transporte, el protocolo y la seguridad.
Beneficios	<ul style="list-style-type: none">• Alineamiento con el dominio ya que la reutilización de servicios comunes con interfaces estándar aumenta las oportunidades tecnológicas y de negocio y reduce costes.• Abstracción ya que los servicios son autónomos y se accede a ellos a través de un contrato formal que proporciona bajo acoplamiento y abstracción.• Descubrimiento ya que los servicios exponen descripciones que permiten a otras aplicaciones y servicios encontrarlos y determinar su interfaz automáticamente.
Cuando usarlo	<ul style="list-style-type: none">• Tienes acceso a servicios adecuados o puedes comprarlos a una empresa.• Quieres construir aplicaciones que compongan múltiples servicios en una interfaz única.• Estás creando S+S, SaaS o una aplicación en la nube.• Necesitas soportar comunicación basada en mensajes para segmentos de la aplicación.• Necesitas exponer funcionalidad de forma independiente de la plataforma.• Necesitas utilizar servicios federados como servicios de autenticación.• Quieres exponer servicios que puedan ser descubiertos y usados por clientes que no tuviesen

conocimiento previo de sus interfaces.

- Quieres soportar escenarios de interoperabilidad e integración.

Tabla 9.- Estilo Arquitectural Bus de Servicios (Mensajes)

Estilo Arquitectural	Bus de Servicios (Mensajes)
Descripción	<p>El estilo de arquitectu- ral de bus de mensajes define un sistema software que puede enviar y recibir mensajes usando uno o más canales de forma que las aplicaciones pueden interactuar sin conocer detalles específicos la una de la otra.</p> 
Características	<ul style="list-style-type: none"> • Es un estilo para diseñar aplicaciones donde la interacción entre las mismas se realiza a través del paso de mensajes por un canal de comunicación común. • Las comunicaciones entre aplicaciones suelen ser asíncronas. • Se implementa a menudo usando un sistema de mensajes como MSMQ. • Muchas implementaciones consisten en aplicaciones

	<p>individuales que se comunican usando esquemas comunes y una infraestructura compartida para el envío y recepción de mensajes.</p>
Principios Clave	<ul style="list-style-type: none">• Toda la comunicación entre aplicaciones se basa en mensajes que usan esquemas comunes.• Las operaciones complejas pueden ser creadas combinando un conjunto de operaciones más simples que realizan determinadas tareas.• Como la interacción con el bus se basa en esquemas comunes y mensajes se pueden añadir o eliminar aplicaciones del bus para cambiar la lógica usada para procesar los mensajes.• Al usar un modelo de comunicación con mensajes basados en estándares se puede interactuar con aplicaciones desarrolladas para distintas plataformas.
Beneficios	<ul style="list-style-type: none">• Expansión ya que las aplicaciones pueden ser añadidas o eliminadas del bus sin afectar al resto de aplicaciones existentes.• Baja complejidad, la complejidad de la aplicación se reduce dado que cada aplicación solo necesita conocer cómo comunicarse con el bus.• Mejor rendimiento ya que no hay intermediarios en la comunicación entre dos aplicaciones, solo la limitación de lo rápido que entregue el bus los mensajes.• Escalable ya que muchas instancias de la misma

aplicación pueden ser asociadas al bus para dar servicio a varias peticiones al mismo tiempo.

- Simplicidad ya que cada aplicación solo tiene que soportar una conexión con el bus en lugar de varias conexiones con el resto de aplicaciones.

Cuando usarlo

- Tienes aplicaciones que interactúan unas con otras para realizar tareas.
- Estás implementando una tarea que requiere interacción con aplicaciones externas.
- Estás implementando una tarea que requiere interacción con otras aplicaciones desplegadas en entornos distintos.
- Tienes aplicaciones que realizan tareas separadas y quieres combinar esas tareas en una sola operación.

Referencias de estilos de arquitectura

<http://www.microsoft.com/architectureguide>

Evans, Eric. *Domain-Driven Design: "Tackling Complexity in the Heart of Software"*. Addison-Wesley, 2004.

Nilsson, Jimmy. *"Applying Domain-Driven Design and Patterns: With Examples in C# and NET"*. Addison-Wesley, 2006.

"An Introduction To Domain-Driven Design" at <http://msdn.microsoft.com/en-us/magazine/dd419654.aspx>

"Domain Driven Design and Development in Practice" at <http://www.infoq.com/articles/ddd-in-practice>

"Fear Those Tiers" at <http://msdn.microsoft.com/en-us/library/cc168629.aspx>.

"Layered Versus Client-Server" at

BORRADOR MARZO 2010

<http://msdn.microsoft.com/en-us/library/bb421529.aspx>

"Message Bus" at **<http://msdn.microsoft.com/en-us/library/ms978583.aspx>**

"Microsoft Enterprise Service Bus (ESB) Guidance" at
<http://www.microsoft.com/biztalk/solutions/soa/esb.msp>

"Separated Presentation" at
<http://martinfowler.com/eaDev/SeparatedPresentation.html>

"Services Fabric: Fine Fabrics for New-Era Systems" at
<http://msdn.microsoft.com/en-us/library/cc168621.aspx>