Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

UDA – Utilidades de desarrollo de aplicaciones

Arquitectura conceptual

Fecha: 06/06/2011

Referencia:

EJIE S.A.
Mediterráneo, 14
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejie.es

Control de documentación

Título de documento: Arquitectura conceptual

Histórico de versiones

Código:	Versión:	Fecha:	Resumen de cambios:
	1.0.0	06/06/2011	Primera versión.
	1.1.0	17/10/2011	Corrección de formato

Cambios producidos desde la última versión

Formato del control de documentación

Control de difusión

Responsable: Ander Martínez

Aprobado por:

Firma: Fecha:

Distribución:

Referencias de archivo

Autor:

Nombre archivo:

Localización:

Contenido

1 Introducción.

El objeto del presente documento es la definición de arquitectura conceptual de los desarrollos Java EE.

La estructuración del presente documento se ha realizado siguiendo los estándares marcados por el modelo "IEEE Recommended Practice for Architectural Description of Software Intensive Systems".

Los requisitos fundamentales que deben ser respetados por la arquitectura (y el conjunto de entregables del proyecto) se describen en el documento de requisitos.

Principios fundamentales

El objetivo que se persigue con el sistema que se describe en este documento es construir una arquitectura, que facilite un proceso de desarrollo de aplicaciones web lo más **productivo** posible. Tratamos además de reutilizar, mantener las cosas simples, no limitar la evolución e independizar la solución de las tecnologías en la medida de lo posible. En definitiva estamos describiendo un sistema basado en **componentes reutilizables** que unidos a **generadores de código** nos permitan producir software de calidad con poco esfuerzo.

Dejando claro que la productividad es nuestro primer objetivo hay otros aspectos que se han tenido en cuenta a la hora de elaborar esta solución. Es importante recalcar que se prima la selección de herramientas existentes frente al desarrollo propietario. De entre éstas prevalecen aquellas que se apoyan en **estándares**. Si además éstas son **Open Source** mejor. Este concepto es igualmente válido a la hora de componer nuestra arquitectura conceptual ya que está basada en arquitecturas de referencia actuales. Así pues, optamos por un diseño por capas que maximice la **separación de responsabilidades** y por tanto haga más fácil conseguir los requerimientos técnicos que se presuponen a una arquitectura de este tipo, los denominados "ilities" (exensibility, flexibility, testability, reliability…).

Dicha arquitectura debe dar solución tanto a entornos Intranet como a entornos Internet para lo cual el aspecto de accesibilidad se ha debatido y tenido en cuenta especialmente. De la misma forma se han tenido en cuenta aspectos que tanto afectan al rendimiento en estos entornos como minimizar el tráfico a través de la red. Para ello una vez más utilizaremos estándares en forma de **patrones de diseño** (facade, value object…).

La arquitectura también cubre la relación entre aplicaciones, es decir la forma en la que unas aplicaciones exponen servicios para que otras las consuman, incluyendo la integración con frameworks anteriores de EJIE (Geremua 2).

2 Identificación de los interesados y los objetivos de la arquitectura

En los siguientes puntos se establecen los roles o perfiles a los que va dirigido el presente documento además de los objetivos perseguidos por el mismo.

2.1 Interesados

El presente documento está dirigido a los responsables técnicos de cada proyecto de desarrollo, concretamente a los jefes de proyecto o responsables de la arquitectura técnica de los proyectos. Es decir, se trata de un documento dirigido a describir la arquitectura técnica diseñada, justificando punto por punto las decisiones tomadas en su proceso de elaboración. Por lo tanto se trata de un documento fundamentalmente teórico que se centra especialmente en la descripción y justificación de la arquitectura y no en la forma de programar las aplicaciones.

Para más información sobre la forma de programar las aplicaciones consultar la Guía de Desarrollo.

2.2 Objetivos

El objetivo del presente documento es ofrecer una visión clara de los elementos que conforman la arquitectura de aplicaciones Java EE, independientemente de tecnologías concretas.

3 Referencias y vistas utilizadas

El contenido y formato del documento presente es una evolución del estudio de modelos de arquitectura llamado "20090216_Modelos_Arquitectura.doc". En función de las conclusiones extraídas de dicho estudio se establecieron las siguientes normas o reglas de cara a documentar la arquitectura:

- Respetar en lo posible el modelo presentado por IEEE para organizar la documentación de arquitectura
- El uso de varias vistas para documentar la arquitectura incluyendo como mínimo las siguientes vistas obtenidas de la agregación de vistas de las diferentes metodologías de documentación de arquitecturas analizadas: vista del entorno de desarrollo (propuesta por RUP y SEI), vista del entorno de despliegue (propuesta por todas las referencias analizadas), vista estática o estructural (SEI, IEEE y RUP), vista dinámica (propuesta por SEI, IEEE y RUP) y vista de procesos (propuesta por RUP).

Una vez realizado el trabajo más teórico de recopilación de referencias y haber consensuado las vistas a utilizar, se detectó una carencia desde el punto de vista metodológico a la hora de afrontar el proyecto de arquitectura. Es decir, los diferentes estándares establecían un estándar de creación de vistas pero no aportan los pasos a seguir para llegar a dichos planos.

Para intentar mejorar este último aspecto se ha tomado como referencia la propuesta realizada por Volter donde se propone la definición de la arquitectura en varios pasos, concretamente:

- Definición de la arquitectura conceptual: definición de la arquitectura desde un punto de vista lógico intentando en lo posible aislar de las tecnologías a utilizar
- Justificación de la arquitectura conceptual
- Definición de las tecnologías a utilizar para implementar el modelo conceptual
- Definición del modelo de programación
- Arquitectura de aplicación: se trata de la organización de la arquitectura desde un punto de vista funcional o del dominio sobre el que se está trabajando. Por ejemplo la división del código en varios módulos o proyectos.

En los siguientes puntos se describe la arquitectura propuesta, haciendo uso de los planos o vistas acordados y siguiendo los pasos establecidos por la metodología de Volter.

4 Definición de la arquitectura conceptual

Entendemos como arquitectura conceptual al proceso de definición de la arquitectura intentando aislar en lo posible las tecnologías de implementación finales. Es decir, se trata de la definición de la estructura y comportamiento de la arquitectura utilizando en primer lugar las ideas base o patrones de diseño de la misma. Posteriormente se irá concretando. En primer lugar mediante la selección de tecnologías (qué tecnologías) y finalmente mediante el modelo de programación (como utilizamos las tecnologías).

El hecho de intentar trabajar sobre ideas o patrones en primer lugar facilita la sustitución de las tecnologías seleccionadas en primera instancia. Además simplifica el desarrollo de aplicaciones al trabajar sobre conceptos funcionales y no sobre tecnologías concretas.

Por ejemplo, dentro de la definición de la arquitectura conceptual podemos hablar de que queremos una arquitectura local o que es necesario aislar cada módulo mediante interfaces (sin depender de la implementación) evitando en lo posible la forma o el lenguaje en el que se implementará posteriormente.

4.1 Requerimientos de la arquitectura

Como punto de partida de la definición de la arquitectura conceptual es necesario cumplir con los requerimientos habituales de toda arquitectura:

- Fomentar la Productividad del desarrollo
- Reutilización de código
- Rendimiento
- Escalabilidad
- Fiabilidad
- Disponibilidad
- Extensibilidad
- Mantenibilidad
- Fácil manejo
- Seguridad

4.2 Bases de la arquitectura conceptual

4.2.1. División de responsabilidades

Uno de los principios básicos en la definición de toda arquitectura para posibilitar gran parte de los requisitos establecidos en el punto anterior (extensibilidad, mantenibilidad, facilidad de manejo) es sin duda alguna la división de responsabilidades o la creación de módulos con una finalidad en concreto, evitando mezclar diferentes aspectos de la creación de una aplicación en un mismo módulo.

Capas

Dentro del desarrollo de aplicaciones software, en este caso dirigido fundamentalmente al desarrollo de aplicaciones web, existen tareas de muy diferente índole, por ejemplo: la creación del interface gráfico, la persistencia y consulta de datos, etc.,..

Con el objeto de dividir la complejidad de la arquitectura en módulos más manejables se han establecido las siguientes capas lógicas:

- **Capa de presentación**: Es la responsable de ofrecer a los usuarios la posibilidad de interactuar con la aplicación de forma sencilla e intuitiva, proporcionando una experiencia de usuario Web satisfactoria.

  La capacidad de desacoplar una interfaz de usuario y reemplazarla con otra suele ser un requerimiento clave en todas las capas de presentación.

  A grandes rasgos se podría decir que la capa de presentación se compone de:

  1. La interfaz de usuario: ofrece a los usuarios información, sugerencias, acciones y captura los datos de entrada a través del teclado y el ratón.
  2. La lógica de presentación: se ejecuta en cliente y hace referencia a todo el procesamiento requerido para mostrar datos y transformar los datos de entrada en acciones que podemos ejecutar contra el servidor.
  3. La lógica de control de peticiones: reside en el servidor y se encarga de interactuar con lógica de presentación, encargándose principalmente de transformar el modelo de datos para la recepción/envío, gestionar las reglas de navegación y enlazar con la capa de servicios de negocio. También se conoce como la subcapa de control.

- **Capa de servicios de negocio**: En esta capa es donde se deben implementar todas aquellas reglas obtenidas a partir del análisis funcional del proyecto. Un servicio se compone de funciones sin estado, auto-contenidas, que aceptan una(s) llamada(s) y devuelven una(s) respuesta(s) mediante una interfaz bien definida. Los servicios no dependen del estado de otras funciones o procesos. Los servicios pueden también enmarcarse en unidades discretas de trabajo, como son las transacciones, que coordinan el uso de la capa de acceso a datos.
- **Capa de acceso a datos:** La capa de acceso a datos es la responsable de la gestión de la persistencia de la información manejada en las capas superiores. En un modelo académicamente purista, la interfaz de esta capa estaría compuesta por vistas de las entidades a persistir (una vista de “factura”, otra de “cliente”), pero a efectos prácticos, y con objeto de aprovechar la habitual potencia de los gestores de bases de datos, la interfaz muestra una serie de métodos que pueden agrupar operaciones en lo que se puede denominar “lógica de persistencia”, como insertar cliente o inserción factura, en la que podrían darse de alta al mismo tiempo una factura y todos las entidades que dependan de dicha factura (porque no, el mismo cliente).
- **Capa de Remoting**: Su propósito será el de permitir ofrecer un acceso a los servicios de negocio debidamente empaquetados a aplicaciones diferentes a la que contiene dichos servicios. Para ello, se utilizan técnicas de Remoting o invocación remota a Objetos. En general en las aplicaciones que no necesiten accesos remotos esta capa no es necesaria. Por esto último se deduce que la capa de Remoting **no puede contener lógica de negocio** y ha de actuar como Proxy (consultar patrón Proxy más adelante) de la capa de servicio.
- **Capa de modelo de datos**: Uno de los elementos fundamentales del diseño de una aplicación es la definición de los objetos que conforman el modelo de datos de la aplicación. En un modelo de capas cada una de ellas tiene acceso únicamente a las funciones expuestas por la capa siguiente con la excepción del modelo. El modelo es una capa vertical utilizada por todas las capas horizontales, es por ello que el modelo no utiliza las funciones de ninguna de las capas horizontales. Cuando se dice que el modelo es utilizado por todas las capas horizontales se hace referencia a que habitualmente forma parte de los parámetros de entrada y salida que definen sus interfaces.

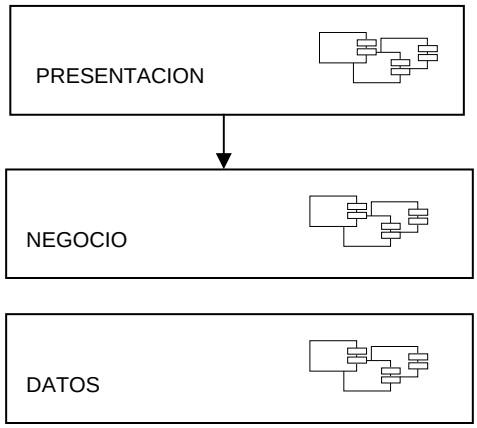
  El modelo de datos es una abstracción del modelo relacional de base de datos representado en clases Java.

Con el objeto de simplificar y por tanto ser más productivo, las aplicaciones serán locales siempre que no haya requisitos específicos que apunten lo contrario. Con el término local lo que se trata de indicar es que sus servicios no se exponen a otras aplicaciones. Este modelo requerirá de menor infraestructura y facilitará el desarrollo como se verá una vez se plasme la arquitectura en un modelo real.

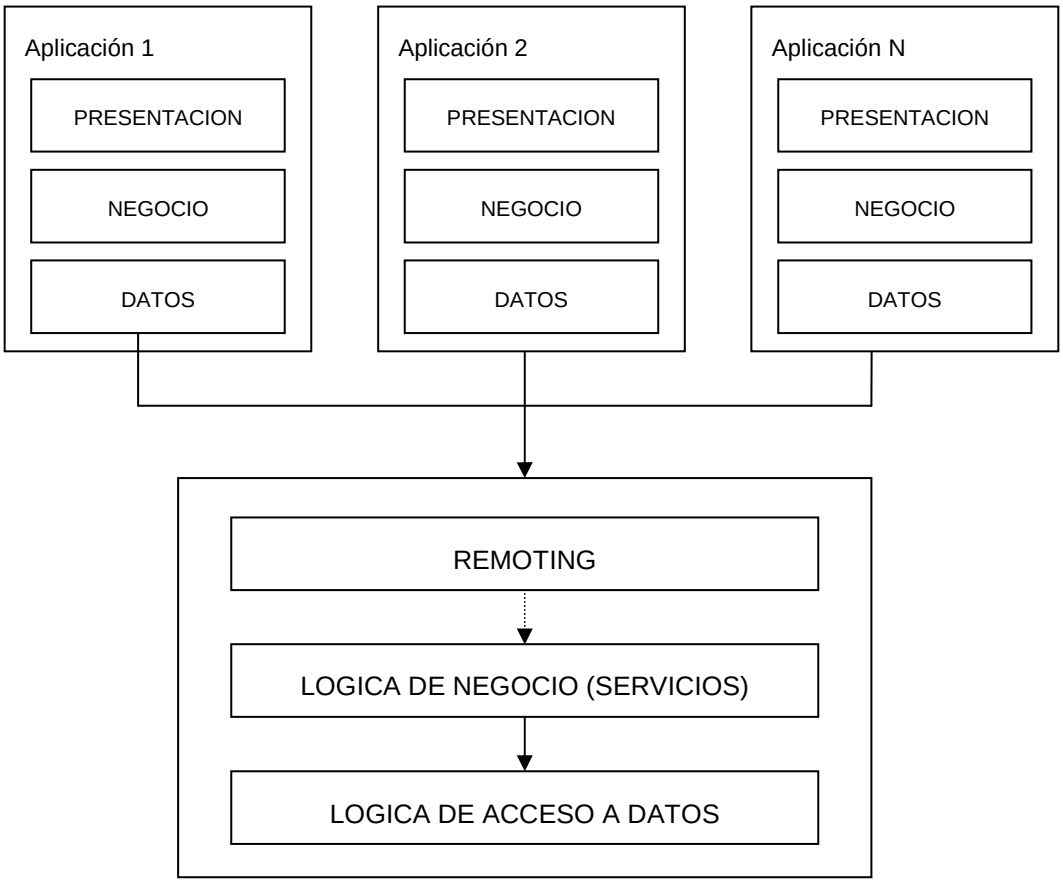
Así pues proponemos una distinción entre dos tipos de aplicaciones. Aplicaciones que exponen algunos de sus servicios a otras aplicaciones y aplicaciones sencillas, autocontenidas

Las siguientes figuras ilustran ejemplos de la relación entre las diferentes capas en un modelo simple o local y en un modelo en el que los servicios son puestos a disposición de otras aplicaciones. En este último caso se puede ver que la combinatoria es mayor existiendo modelos como el primero donde los servicios comunes son centralizados y expuestos por una sola aplicación y en el segundo caso un modelo en el que cada aplicación ofrece sus propios servicios al resto (modelo 2). Por supuesto existen combinaciones del modelo 1 y 2. Si el número de aplicaciones que exponen servicios es grande una aplicación centralizadora para facilitar la gestión de los servicios (bus de servicios) sería otra posibilidad a tener en cuenta.

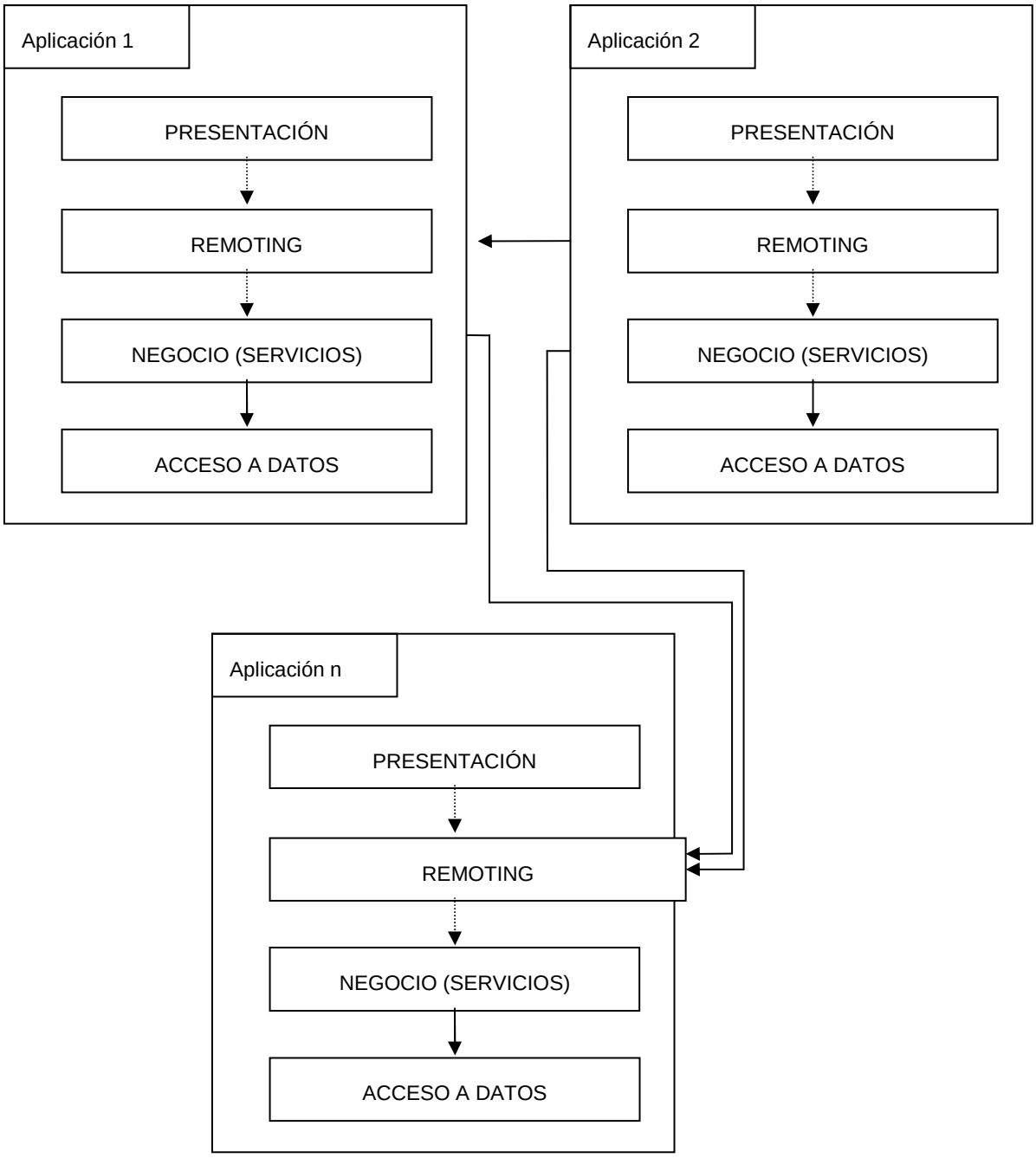
Modelo de aplicación (local)



Modelo de aplicación con exposición de servicios remotos (modelo 1):



Modelo de aplicación con exposición de servicios remotos (modelo 2):



Las ventajas ofrecidas por esta base de la arquitectura son las siguientes:

- Permite sustituir un módulo sin afectar al resto, favoreciendo la mantenibilidad de las aplicaciones
- Reduce la complejidad al dividir las aplicaciones

4.2.2. Lenguaje de comunicación de datos uniforme

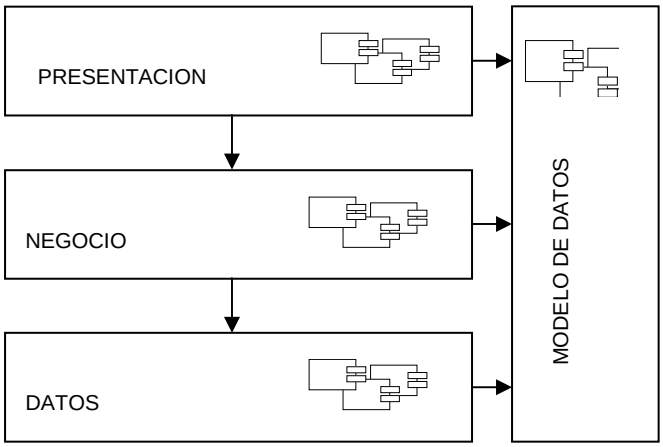
El modelo de capas anterior refleja el principio de Separación de Responsabilidades permitiendo una clara organización de los desarrollos y posibilitando además la especialización de desarrolladores. Sin embargo existen una serie de abstracciones cuyo desempeño es transversal (cross-cutting).a las anteriores. Es decir, así como La capa de presentación únicamente es consciente que existe la capa de negocio a través de su interface y no es consciente de que existe una capa de datos, existen elementos como los objetos del modelo de datos que son accesibles desde cualquier capa.

En toda aplicación se requiere el manejo de datos. A pesar de que se trata del desarrollo de aplicaciones informáticas no podemos obviar que dichos datos representan a “cosas” del mundo real, más concretamente a “cosas” existentes dentro de la semántica que acompaña a los datos que se quieren representar.

Por lo tanto otra de las bases predefinidas dentro de la arquitectura conceptual es el uso de los conceptos o palabras utilizadas en el mundo real, evitando en lo posible la creación de un doble lenguaje para referenciar a las mismas “cosas”.

Las ventajas ofrecidas por esta base de la arquitectura son las siguientes:

- Mejora la comunicación entre los creadores de las aplicaciones informáticas y los expertos en la lógica del negocio (profesores, alumnos, ascensores, expedientes, voladuras,…).
- Minimiza la duplicidad de código al no crear dos representaciones diferentes de los mismos datos.



4.2.3. Bajo Acoplamiento

El bajo acoplamiento permite mejorar la programación y el diseño de sistemas informáticos y aplicaciones. Es un factor muy importante en el incremento de la reutilización del código.

El acoplamiento informático indica el **nivel de dependencia** entre las capas de una aplicación, es decir, el grado en que una capa puede funcionar sin recurrir a otras; dos capas son absolutamente independientes entre sí (el nivel más bajo de acoplamiento) cuando una puede hacer su trabajo completamente sin recurrir a la otra. En este caso se dice que ambas están desacopladas.

El bajo acoplamiento entre las capas de una aplicación es el estado ideal que siempre se intenta obtener para lograr una buena programación o un buen diseño. Cuanto menos dependientes sean las partes que constituyen un sistema informático, mejor será el resultado. Sin embargo, es imposible un desacoplamiento total de las unidades.

Por ello, uno de los objetivos de la arquitectura propuesta es reducir al máximo el acoplamiento entre componentes. Para ello, lo más importante es saber **eliminar el acoplamiento que no sea funcional o arquitectónico**.

El caso del bajo acoplamiento funcional puede ser ilustrado por un componente de cálculo de probabilidades que dependa de un componente de cálculo matemático básico, ya que para calcular probabilidades será necesario aplicar fórmulas matemáticas.

En el caso del acoplamiento arquitectónico, únicamente se permite el acoplamiento de las capas de la manera indicada en el capítulo de División de responsabilidades, explicado anteriormente en este mismo documento. El proceso de acoplamiento será llevado a cabo mediante Inyección de Dependencias (Dependency Injection), de manera que los componentes únicamente referencian a los Interfaces de los componentes de los que dependen, desconociendo por completo su(s) implementación(es) e invirtiendo así el control del proceso de resolución de dependencias (Inversion of Control), ya que queda en manos del contenedor IoC. Este último, se encarga de resolver, enriquecer (con Proxies Dinámicos) e inyectar dichas dependencias.

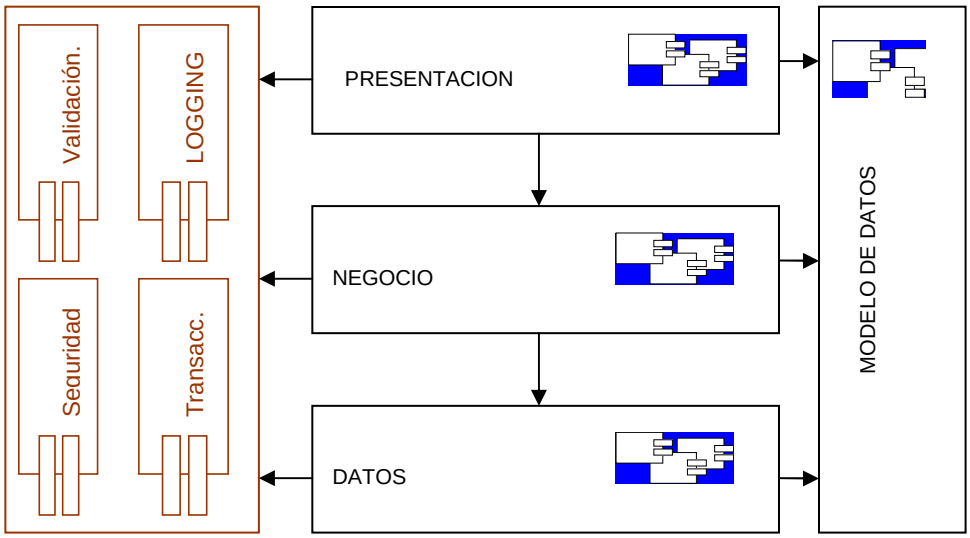
En resumidas cuentas, el bajo acoplamiento permite:

- Mejorar la mantenibilidad de las unidades de software.
- Aumentar la reutilización de las unidades de software.
- Evitar el efecto onda, ya que un defecto en una componente puede propagarse a otros, haciendo incluso más difícil de detectar dónde está el problema.
- Minimiza el riesgo de tener que cambiar múltiples componentes de software cuando se debe alterar uno.

4.2.4. Externalización de responsabilidades globales o transversales.

El modelo de capas anteriormente expuesto (presentación, negocio, datos, modelo de datos) está conformado por componentes con responsabilidades muy claras e independientes. Sin embargo todos ellos comparten la necesidad de efectuar operaciones comunes y por tanto reutilizables. ¿Ejemplos? Pues son los ampliamente conocidos: logging, seguridad, transaccionalidad, excepciones. Separar estas responsabilidades en módulos nos permitirá reutilizarlos y por tanto simplificar el desarrollo. Si además estos módulos son proporcionados por la arquitectura estaremos simplificando aún más el desarrollo permitiendo a los desarrolladores centrarse en escribir el código que describe los casos de uso de su aplicación. Siguiendo el principio de minimizar el acoplamiento. Estimamos que la mejor forma de aportar estas características a un desarrollo es utilizando AOP.

Así pues podríamos completar la figura anterior que resume la arquitectura conceptual.



La figura superior muestra en fondo azul los componentes cuya implementación corre a cargo de los desarrolladores, mientras que los componentes de color granate son implementados por la arquitectura y la responsabilidad de desarrollador es aplicarlos y configurarlos únicamente.

4.2.5. Minimizar los costes de comunicación entre módulos

Dentro de una aplicación informática es habitual la existencia de dependencias entre diferentes módulos, ya sea dentro de una misma aplicación o entre aplicaciones diferentes. Existen diferentes tipos de comunicaciones en función del protocolo utilizado para la comunicación o la localización física del mismo.

Otro de los aspectos fundamentales es la estandarización de los datos intercambiados entre diferentes módulos.

Dado que siempre puede existir la dependencia entre aplicaciones, el objeto de esta regla base de la arquitectura es minimizar el impacto de la comunicación entre módulos, intentando en lo posible mantener las siguientes pautas:

- **Fomentar la comunicación local** en lo posible, evitando las llamadas remotas.
- Fomentar el uso del mismo lenguaje de manejo de datos (punto 4.2.2) para **evitar el costo de transformación de datos** entre módulos diferentes.

Las ventajas ofrecidas por esta base de la arquitectura son las siguientes:

- Mejorar el rendimiento de las aplicaciones gracias al uso de llamadas locales (siempre que sea posible) y evitando la transformación de datos.

Evidentemente, estas reglas aplican únicamente a módulos Software que se ejecutan en el ámbito de un mismo servidor de aplicaciones.

Es inevitable el costo de la transformación de datos y la comunicación remota entre el servidor y los clientes Web o navegadores.

4.2.6. Fomentar el uso de comunicaciones desatendidas o asíncronas

Tal y como se ha presentado en el punto anterior las dependencias entre diferentes módulos pueden ser resueltas de forma diferente en función del protocolo y la situación física de cada módulo. Uno de los sistemas de comunicación existente es el uso de comunicaciones asíncronas, aquellas que no requieren esperar el resultado final.

Este tipo de comunicaciones ofrecen varias ventajas respecto al modelo de comunicación síncrono:

- Es la solución que requiere menor grado de acoplamiento No requiere la disponibilidad del módulo dependiente, es decir, el destinatario de la llamada final puede no estar disponible en el momento que se realiza la llamada al módulo y puede procesar el mensaje posteriormente.
- El módulo consumidor puede seguir ejecutando su lógica de negocio sin depender del tiempo de respuesta del proveedor de servicios, mejorando la escalabilidad de las aplicaciones en momentos de carga intensa.

Este modelo de comunicación es precisamente el utilizado por AJAX para crear aplicaciones interactivas. Ajax realiza peticiones Web (HTTP) asíncronas que se lanzan desde el navegador del usuario, y mantiene comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre la misma página sin necesidad de recargarla. Esto significa aumentar la interactividad, velocidad y usabilidad en la misma.

4.2.7. Independizar el código de las tecnologías de cada momento

Uno de los aspectos clave en los sistemas informáticos es la velocidad de evolución de las tecnologías. Es decir, las plataformas tecnológicas evolucionan cada año y dado que las plataformas sobre las que se sostienen las aplicaciones cambian, también deben cambiar en muchos casos las aplicaciones desarrolladas sobre las mismas.

En consecuencia, cualquier arquitectura muy dependiente de la plataforma específica de cada momento tiene un nivel de riesgo considerable dado que el cambio de la tecnología base puede provocar el cambio o modificación en las aplicaciones ya desarrolladas.

Para poner un ejemplo más claro, la evolución de las tecnologías en el mundo Java es extremadamente rápida. Cada año se publican nuevas especificaciones y se modifican las existentes. Dentro del mundo Java existen tecnologías muy cambiantes (WebServices, EJBs,…) y otras como la JDK que disponen de una estabilidad y una compatibilidad hacía atrás mucho mayor.

Por otro lado, aunque las tecnologías evolucionan rápidamente en el corto plazo, las necesidades funcionales de un usuario de una aplicación informática son mucho más constantes y duraderas en el tiempo, es decir, aquello que funcionalmente se implementa en un momento concreto en el tiempo es probable que desde el punto de vista funcional se trate de una solución válida durante varios años.

El objetivo de esta base de la arquitectura es independizar el código generado lo máximo posible del entorno de ejecución final, mejorando la adaptabilidad y reutilización del código en un futuro y en lo posible igualar el tiempo de caducidad del código al tiempo de validez funcional del mismo. Para ello resulta fundamental ligar la menor cantidad de código posible a especificaciones o tecnología concretas y utilizar soportes más universales, como por ejemplo los POJO (Plain Old Java Objects).

Las ventajas ofrecidas por esta base de la arquitectura son las siguientes:

- Minimizar las dependencias sobre las especificaciones o implementaciones de cada momento
- Mejorar la reutilización del código generado
- Facilitar el cambio tecnológico, posibilitando el intercambio de piezas sobre las que se sustenta la arquitectura.

4.2.8. Independencia respecto al entorno de ejecución final

En un entorno tecnológico con tanta diversidad como el del mundo Java, resulta muy difícil ejercer de visionario y marcar una estrategia ganadora a largo plazo. Es por ello que conviene no casarse (tecnológicamente hablando) demasiado con un proveedor/fabricante de software concreto y disponer de cierta flexibilidad a la hora de adaptarse a una u otra plataforma.

Dicha flexibilidad viene dada, sin duda, por el nivel de ligazón que sufre el código desarrollado con respecto a las tecnologías sobre las que se ejecuta. En consecuencia, se puede afirmar que el nivel de portabilidad de una aplicación es directamente proporcional al nivel de compatibilidad que tienen las tecnologías en las que se sustenta con los diferentes entornos de ejecución disponibles en el mercado.

Por lo tanto, hay que actuar con suma cautela a la hora de ligar el código generado con tecnologías no estándares o de limitada adopción por parte de la industria.

Como mínimo, hay que procurar que a grandes rasgos, las aplicaciones generadas sean compatibles tanto con Servidores de Aplicaciones JEE como con Contenedores de Servlets.

Existen casos en los que resulta necesario ligarse a una tecnología que a la larga, pueda desaparecer del entorno de ejecución de la empresa, como son los EJBs, ya que solo están soportados por los Servidores de Aplicaciones JEE y no pueden ser desplegados bajo ningún Contenedor de Servlets. Para evitar estas situaciones, se recomienda (tal y como se ha comentado anteriormente) desarrollar aplicaciones autocontenidas, que no requieran del consumo de servicios remotos proporcionados por EJBs.

4.2.9. Exposición del negocio mediante librerías y servicios remotos

Para los casos extremos en los que no sea suficiente con aplicaciones autocontenidas, la arquitectura final debe posibilitar el encapsulamiento de la lógica de negocio (todo aquello que no sea creación de pantallas) en formato librería (formato de empaquetado que es posible moverlo físicamente) o en un formato que posibilite su consumo de forma remota mediante un protocolo de red.

La arquitectura no debe condicionar en ningún caso los protocolos utilizados para exportar los servicios de lógica de negocio.

Los servicios remotos deben posibilitar en la medida de lo posible el soporte para transacciones distribuidas. Las ventajas ofrecidas por esta base de la arquitectura son las siguientes:

- La lógica de negocio puede ser consumida por cualquier aplicación (local o remota)

4.2.10. Testabilidad de los componentes

Otro de los aspectos clave de la arquitectura es la capacidad de poder testear cada componente de la arquitectura de forma independiente. Es decir, debe de posibilitar la implementación y ejecución de pruebas unitarias y de integración. Las ventajas ofrecidas por esta base de la arquitectura son las siguientes:

- Es posible probar de forma independiente cualquier componente software de la arquitectura
- Reduce la existencia de errores gracias a la detección temprana de errores

4.2.11. Uso de patrones

Los patrones de diseño tratan problemas que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas. Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes. Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.

A modo introductorio, cabe mencionar que los patrones de diseño analizados a continuación fueron definidos literariamente por primera vez de la mano de un grupo de expertos denominado “The Gang of Four”, también conocidos como GoF.

GoF clasificó los patrones en 3 grandes categorías basadas en su **propósito**: creacionales, estructurales y de comportamiento.

- **Creacionales**: Los patrones creacionales tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
- **Estructurales**: Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades.
- **Comportamiento**: Los patrones de comportamiento ayudan a definir la comunicación e interacción entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

Propósito / Ámbito	Creacional	Estructural	Comportamiento
Clase	✓ Factory Method	✓ Adapter	× Interpreter ✓ Template Method
Objeto	✓ Prototype ✓ Singleton × Abstract Factory × Builder	✓ Adapter × Bridge × Composite ✓ Decorator ✓ Facade × Flyweight ✓ Proxy	✓ Chain of Responsibility × Command ✓ Iterator × Mediator × Memento ✓ Observer × State × Strategy × Visitor

En la tabla superior se indican (con el símbolo ✓) los patrones sobre los que se erige la arquitectura conceptual propuesta y se ignoran (con el símbolo ×) los patrones que no influyen directamente en el diseño de la arquitectura, aunque pueden ser usados a la hora de desarrollar los casos de uso concretos de las aplicaciones, si se estima oportuno.

Además de los patrones de la tabla, se hace uso del patrón Delegation.

4.2.11.1. Patrón Delegation

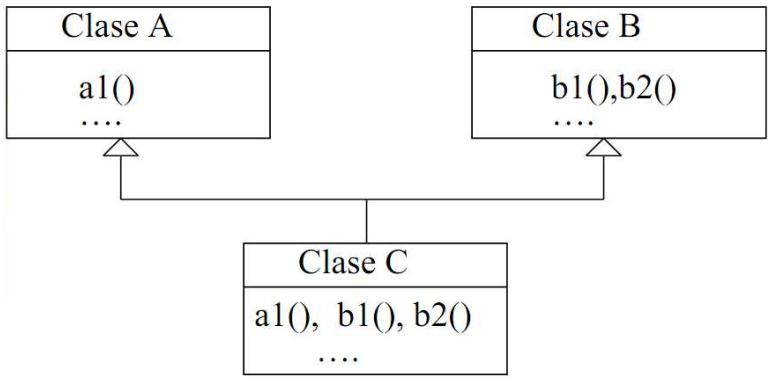
Utilidad:

- Cuando se quiere extender y reutilizar la funcionalidad de una clase sin utilizar la herencia.

Ventajas:

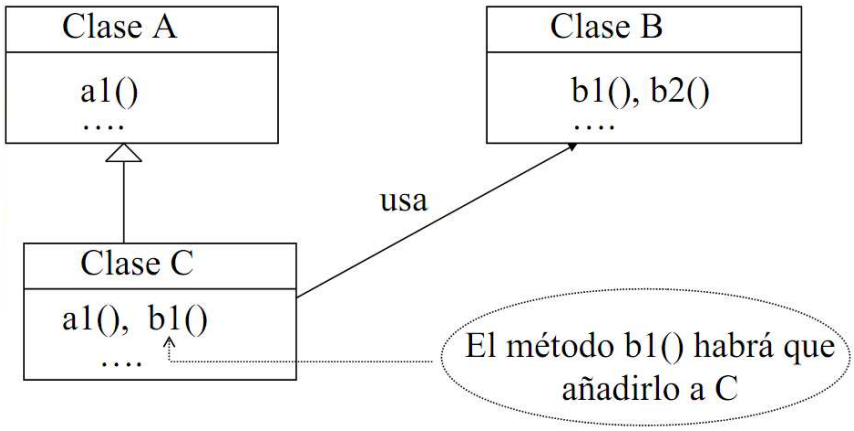
- Emula hasta cierto punto la herencia múltiple, no soportada por Java.
- Por lo tanto, este patrón fomenta la compartición de código que no se puede heredar.

Problemática:



En Java, solo es posible que las clases hereden de una única clase. Además, es posible que no se desee que la Clase C herede todos los métodos de la Clase B.

Solución:



4.2.11.2. Patrón Factory Method

Utilidad:

- Patrón que sirve para fabricar instancias de objetos en tiempo de ejecución, ocultando a los clientes los detalles del proceso de creación de instancias.

Ventajas:

- Permite encapsular y desacoplar el código referente a la creación e inicialización de objetos, de manera que los clientes se ligan únicamente a un interface, despreocupándose por completo del proceso de fabricación de la instancia que implementará ese interface en un contexto de ejecución determinado.

Problemática:

- El código de fabricación de clases se dispersa y los objetos se desvían de su eje funcional, teniendo que preocuparse de la instanciación de otros. Resulta difícil realizar cambiar la implementación de ciertas clases en las aplicaciones, sin que esto requiera reprogramar sus clases dependientes.

Solución:

- Básicamente se trata de un método estático que, mediante un parámetro, tomará la decisión de devolver una instancia de una clase (o generalmente una subclase).

4.2.11.3. Patrón Singleton

Utilidad:

- Asegurar que una clase tiene una sola instancia y proporcionar un punto de acceso global a ella.

Ventajas:

- Es necesario cuando hay clases que tienen que gestionar de manera centralizada un recurso.
- Una variable global no garantiza que sólo se instancie una vez.

Solución:

- Antiguamente se solucionaba con un constructor privado y un método estático (típicamente getInstance()) que devuelve la única instancia de la clase.
- Actualmente, los contenedores Inversion of Control (IoC) proporcionan utilidades de creación de instancias de Objetos Singleton y gestión de dicha instancia mediante Dependency Injection.

4.2.11.4. Patrón Prototype

Utilidad:

- Permitir que las instancias de las clases se creen bajo demanda.

Ventajas:

- Este patrón propone la creación de distintas variantes del objeto que la aplicación necesite, en el momento y contexto adecuado. Toda la lógica necesaria para la decisión sobre el tipo de objetos que usará la aplicación en su ejecución debería localizarse aquí. Luego, el código que utiliza estos objetos solicitará una copia del objeto que necesite. En este contexto, una copia significa otra instancia del objeto. El único requisito que debe cumplir este objeto es suministrar la prestación de clonarse.

Solución:

- Antiguamente, se utilizaba el interface Cloneable para construir este tipo de clases.
- Actualmente, al igual que con el patrón Singleton, los contenedores IoC modernos proporcionan la infraestructura necesaria para fabricar este tipo de instancias.

4.2.11.5. Patrón Adapter

Utilidad:

- Se desea usar una clase existente, y su interfaz no se iguala con la necesitada por la clase cliente, por lo que se crea una clase intermedia que envuelve a la necesitada y la hace accesible al cliente.
- Permite que clases con interfaces incompatibles se comuniquen.

Ventajas:

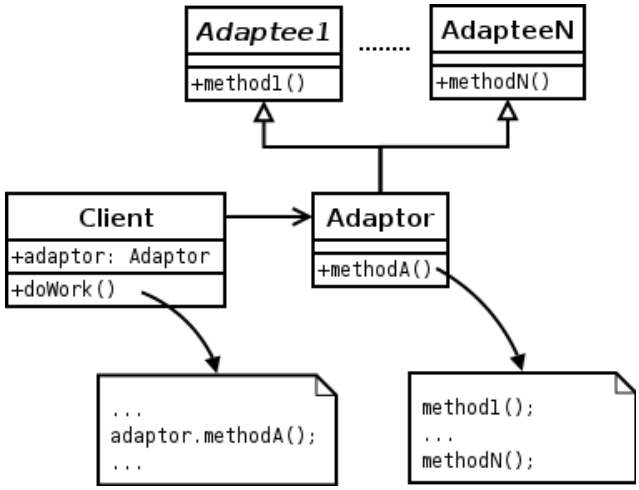
- Posibilita utilizar una clase ya existente y su interfaz aunque no se corresponda con la interfaz que se necesita.
- Permite envolver código no orientado a objeto con forma de clase.

Problemática:

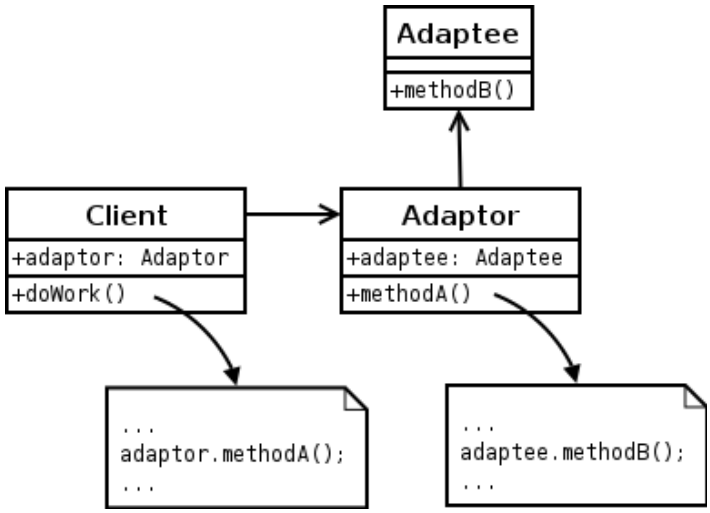
- En ocasiones, cuando se trabaja con aplicaciones modeladas utilizando la filosofía de Orientación a Objetos se dan situaciones en las que cierta lógica queda encapsulada tras una clase que cumple un contrato (o interface) que no es apto para ser utilizado por otra clase que necesita reutilizar dicha lógica.

Solución:

- Para evitar la duplicación de dicho código encapsulado, o la redefinición del contrato de la clase requerida (lo cual podría ocasionar problemas a terceras clases que se encuentran atadas al actual interface) se crea una clase intermedia que actúa como envoltorio o adaptador de la clase necesitada.
- Desde el punto de vista de clase, la problemática mencionada se soluciona de la siguiente manera:



  - Se crea una clase adaptadora con múltiples clases herederas. Cada una de las herederas se adapta al contrato de la clase final requerida por el cliente.
- Desde el punto de vista de objeto, la problemática mencionada se soluciona de la siguiente manera:



  - El objeto adaptador está, en esta instancia, ligado a una “clase adaptada” en concreto y realiza las labores de envoltorio de esta, posibilitando al objeto cliente el uso indirecto de la adaptada.

4.2.11.6. Patrón Decorator

Utilidad:

- A veces se desea añadir responsabilidades a un objeto pero no a toda una clase. Las responsabilidades se pueden incrementar por medio del mecanismo de Herencia, pero este mecanismo no es flexible porque dicha responsabilidad es añadida estáticamente. El patrón Decorator permite añadir dinámicamente responsabilidades a objetos.

Ventajas:

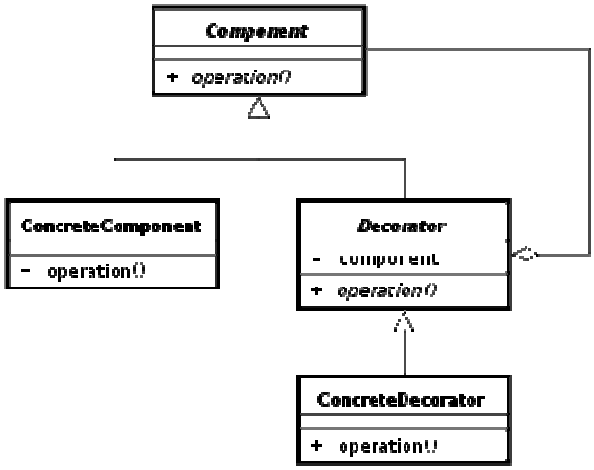
- Posibilita añadir responsabilidades a objetos individuales dinámicamente sin afectar otros objetos.
- Permite que las responsabilidades puedan ser retiradas.
- Ofrece alternativas cuando no es práctico añadir responsabilidades por medio de la herencia.

Problemática:

- En ocasiones resulta necesario implementar una clase Y que herede de una clase X con motivo de extender sus responsabilidades. A continuación, es posible que sea necesario añadir una nueva responsabilidad a la clase X, por lo que se implementaría una nueva clase Z que herede de la clase X. Este esquema de crecimiento puede dar lugar a N clases herederas de X, cada una con su responsabilidad específica, pero no solucionaría el escenario donde hiciera falta un objeto que se encargara de las responsabilidades de la clase Z y la clase Y conjuntamente.

Solución:

- La solución flexible es la de rodear el objeto con otro objeto que es el que aporta la nueva responsabilidad. Este nuevo objeto es el denominado Decorator. La forma más sencilla de explicarlo es la de mostrar el diagrama UML estático y pseudocódigo Java que refleja el comportamiento dinámico de los objetos.




```
abstract class Componente
{
	abstract public void operacion();
}
class ComponenteConcreto extends Componente
{
	public void operacion()
	{
		System.out.println("ConcreteComponent.Operation()");
	}
}
abstract class Decorador extends Componente
{
	protected Componente componente;
	public void setComponente(Componente component)
	{
```

```
                this.componente = component;
        }
        public void operacion()
        {
                if (componente != null) componente.operacion();
        }
}
class DecoradorConcretoA extends Decorador
{
        private String estadoAgregado;
        public void operacion()
        {
                super.operacion();
                estadoAgregado = "Nuevo estado";
                System.out.println("DecoradorConcretoA.Operacion()");
        }
}
class DecoradorConcretoB extends Decorador
{
        public void operacion()
        {
                super.operacion();
                AgregarComportamiento();
                System.out.println("DecoradorConcretoB.Operacion()");
        }
        void AgregarComportamiento()
        {
                System.out.println("comportamiento añadido B");
        }
}
public class Cliente
{
        public static void main(String[] args)
        {
                ComponenteConcreto c = new ComponenteConcreto();
                DecoradorConcretoA d1 = new DecoradorConcretoA();
                DecoradorConcretoB d2 = new DecoradorConcretoB();
                d1.setComponente(c);
                d2.setComponente(d1);
                d2.operacion();
        }
}
```

4.2.11.7. Patrón Facade

Utilidad:

- Proporcionar una interfaz simplificada para un grupo de subsistemas o un sistema complejo.

Ventajas:

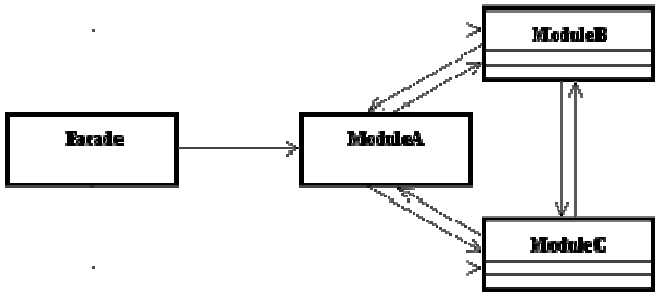
- Simplifica el acceso a un conjunto de clases proporcionando una única clase que todos utilizan para comunicarse con dicho conjunto de clases.
- Reducir la complejidad y minimizar dependencias.

Problemática:

- Las aplicaciones actuales se diseñan de manera modular, pero muchas veces el diseño modular se encuentra mal concebido, por lo que suele ser muy útil crear un representante que gestione la interacción con un módulo o conjunto de módulos, de forma que exponga al mundo exterior (a los clientes) un interfaz bien definido que encapsule todo el uso del API subyacente en forma de **servicio**.

Solución:

- Se trata de definir clases que las responsabilidades de los módulos del subsistema y respondan a las peticiones de los clientes combinando los servicios que ofrecen dichos módulos del subsistema.



4.2.11.8. Patrón Proxy

Utilidad:

- El patrón Proxy se utiliza como intermediario para acceder a un objeto, permitiendo controlar el acceso a él.

Ventajas:

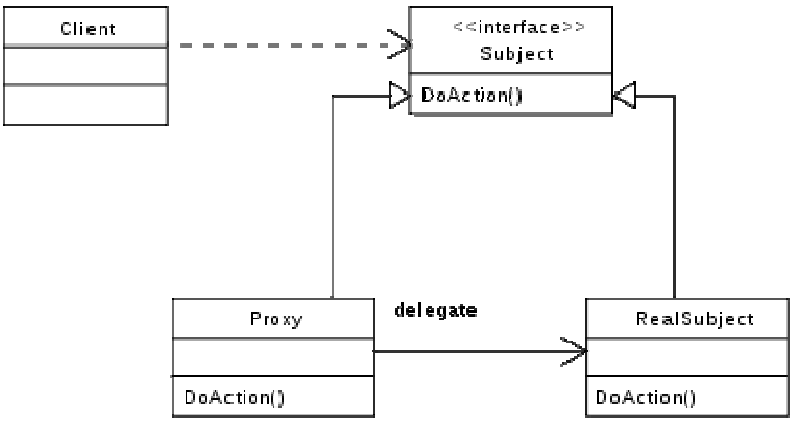
- Posibilita retardar la creación de un objeto pesado, creándolo bajo demanda.
- Permite aislar de la complejidad de las comunicaciones. En el caso de las llamadas remotas, se suele llamar a una Proxy local que encapsula la complejidad referente a la comunicación con el objeto remoto al que representa.
- Puede delimitar el acceso a un recurso.

Problemática:

- Por lo general, este patrón se aplica en los casos en los que existe una complejidad implícita muy alta a la hora de acceder o invocar a un objeto. En estos casos se opta por crear un Proxy a ese objeto para localizar ahí ese código y que pueda ser reutilizado de manera transparente por cualquier cliente.

Solución:

- Por lo general, se define un contrato (en forma de interface) que han de cumplir tanto el Proxy como la clase representada por este. De esa manera, el cliente se liga únicamente al contrato y en tiempo de ejecución, invoca al objeto Proxy en vez de al objeto final.



4.2.11.9. Patrón Template Method

Utilidad:

- Usando el Template Method, se define una estructura de herencia en la cual la superclase sirve de plantilla de los métodos en las subclases.

Ventajas:

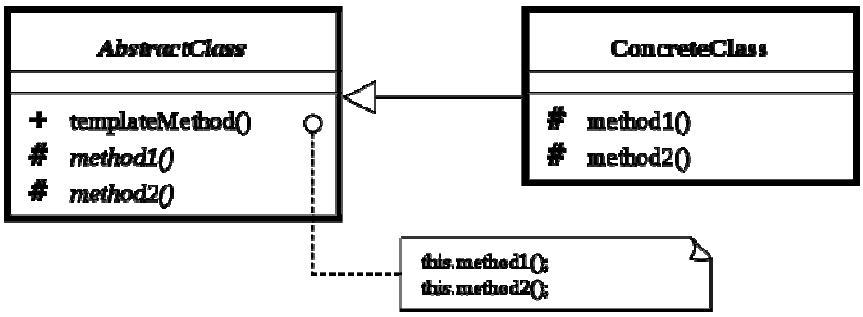
- La principal ventaja de este patrón es que evita la repetición de código.
- Otra gran ventaja es que evita la aparición de errores, ya que se parte de código bien estructurado y probado.

Problemática:

- Este patrón se vuelve de especial utilidad cuando es necesario realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre una y otras. Por ejemplo, en un Data Access Object.

Solución:

- Se trata de proporcionar una clase abstracta con los métodos primitivos que las clases concretas tendrán que implementar. Además esta clase abstracta proporciona un método template, definiendo el esqueleto de un algoritmo.
- Por otro lado, se implementan las clases concretas que implementan los métodos primitivos.



4.2.11.10. Patrón Chain of Responsibility

Utilidad:

- Permite establecer una cadena de objetos receptores a través de los cuales pasa una petición formulada por un objeto emisor. Todos estos receptores aportan su valor añadido a la petición que pasa de un receptor a otro. Además cualquiera de los objetos receptores puede responder a la petición en función de un criterio establecido.

Ventajas:

- Reducción del acoplamiento:
  Ni el receptor ni el emisor se conocen explícitamente. Un objeto sólo tiene que saber que una petición será manejada.
- Añade flexibilidad para asignar responsabilidades a objetos:
  Las responsabilidades de los mensajes pueden cambiar mediante la organización del proceso de ejecución.

Problemática:

- Cubre los casos en los que una petición ha de pasar por una tubería de procesamiento, en la cual se pueden añadir o quitar manejadores y modificar el orden de estos. Todo ello sucede sin que el cliente sea consciente de ello.

Solución:

- Todos los objetos receptores implementarán la misma interfaz o extenderán la misma clase abstracta. En ambos casos se proveerá de un método que permita obtener el sucesor y así el paso de la petición por la cadena será lo más flexible y transparente posible.

4.2.11.11. Patrón Iterator

Utilidad:

- Proporcionar una forma de acceder a los elementos de una colección de objetos de manera secuencial sin revelar su representación interna. Define una interfaz que declara métodos para acceder secuencialmente a la colección.

Ventajas:

- La clase que accede a la colección solamente a través de dicho interfaz permanece independiente de la clase que implementa la interfaz.

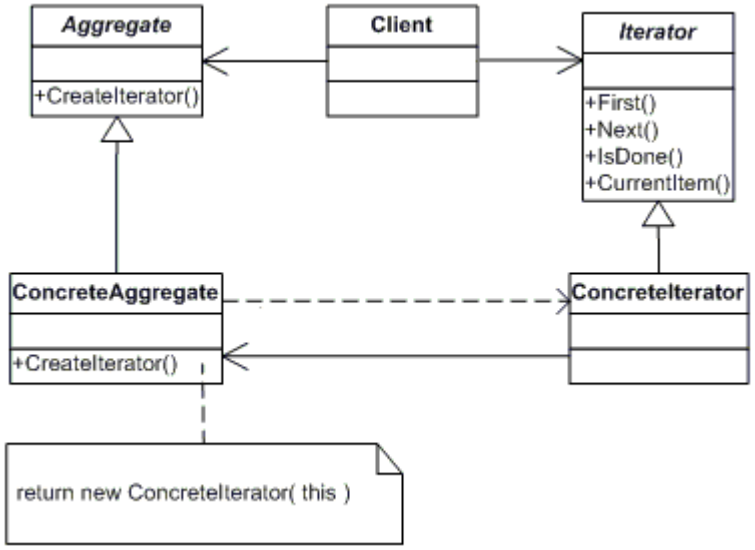
Problemática:

- Cuando las aplicaciones trabajan con listas o colecciones de objetos de grandes dimensiones, es conveniente disponer de utilidades que solucionen la problemática que implica la gestión de dichas colecciones, como por ejemplo, añadir elementos a la lista, removerlos, ordenarlos, seleccionar un elemento concreto, etc.
- Si las utilidades que se encargaran de todo ello fueran especificas para cada clase correspondiente al modelo de datos, sería necesario desarrollar y mantener mucho código dedicado exclusivamente a ello por cada aplicación. Por ello, nace el patrón Iterator con el fin de facilitar el diseño de utilidades genéricas de gestión de colecciones, independientemente del modelo de datos de cada aplicación.

Solución:

- Una implementación muy extendida del patrón Iterator, es la siguiente:
  - **Iterator:**
    - El cual define una interfaz para el acceso y recorrido de los elementos.
  - **ConcreteIterator:**
    - El cual implementa la interfaz de Iterator y mantiene la pista de la posición actual en el recorrido del agregado.

- **Aggregate:**
  - El cual define una interfaz para la creación de un objeto Iterator.
- **ConcreteAggregate:**
  - El cual implementa la interfaz de creación de Iterator para retornar una instancia del ConcreteIterator adecuado.



4.2.11.12. Patrón Observer

Utilidad:

- Definir una dependencia 1:N de forma que cuando el objeto 1 cambie su estado, los N objetos sean notificados y se actualicen automáticamente.

Ventajas:

- Sirve para separar los objetos de presentación (vistas) de los objetos de datos, de forma que se puedan tener varias vistas sincronizadas de los mismos datos (editor-subscriptor).
- El objetivo de este patrón es desacoplar la clase de los objetos clientes del objeto, aumentando la modularidad del lenguaje, así como evitar bucles de actualización, como la espera activa o el polling.

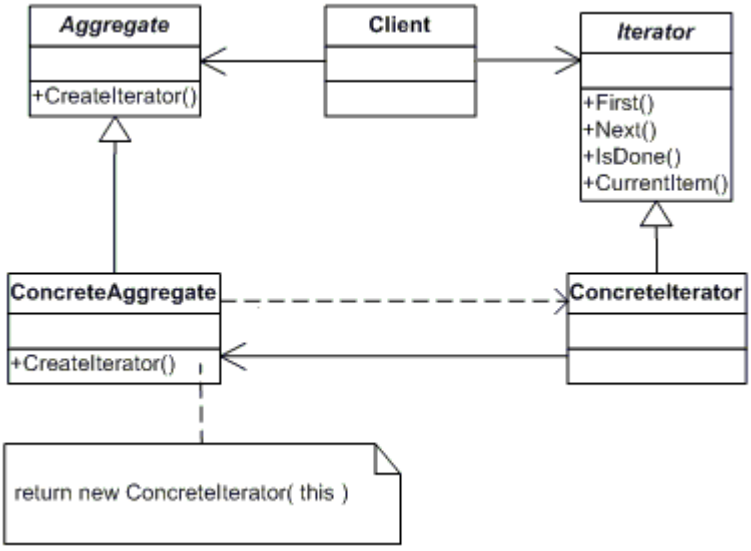
Problemática:

- Propone soluciones para los casos en los que los clientes de un backend necesitan esperar a eventos que el servidor produce para actualizar su estado y su presentación.

Solución:

- Este patrón también se conoce como el patrón de publicación-inscripción. Este nombre sugiere la idea básica del patrón, que es bien sencilla: el objeto de datos, llamémoslo "Sujeto" a partir de ahora, contiene atributos mediante los cuales cualquier objeto observador o vista se puede suscribir a él pasándole una referencia a sí mismo. El Sujeto mantiene así una lista de las referencias a sus observadores.
- Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el Sujeto es capaz de notificar a sus observadores "suscritos" los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el Sujeto, ejecutado, por ejemplo, por alguno de los observadores, el objeto de datos puede recorrer la lista de observadores avisando a cada uno.

- Este patrón suele observarse en los frameworks de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir 'listeners' a los objetos que pueden disparar eventos.



4.2.12. Componentes de Arquitectura

Existen ciertas funcionalidades que la arquitectura ha de proporcionar a todas las aplicaciones que se erijan sobre ella, como son la seguridad, la transaccionalidad, la validación y la generación de trazas. Todas ellas, han de ser transparentes al código de la aplicación, ya que se trata de aspectos de uso común (o Cross Cutting Concerns), por lo que se inyectarán en el código a través de aspectos AOP.

En líneas generales, los Componentes de Arquitectura se basan en estándares JEE y estándares de facto que rodean a dicha especificación, aunque en determinados aspectos, como el de la seguridad y la generación de trazas, se propone la inclusión de sendos módulos propietarios.

Ambos módulos propietarios han de ser dispensables, de manera que sean de uso opcional y su uso (o no uso) no afecte en absoluto al resto de piezas de la arquitectura.

- Módulo se seguridad: Se propone la inclusión de un conector que encapsule el uso del sistema de seguridad propio de Ejie (XLNetS), de manera que el código de las aplicaciones finales sea independiente del mencionado proveedor de seguridad. El código de las aplicaciones sólo ha de ligarse con la solución estándar de alto nivel que encapsule el uso de XLNetS, pudiendo esta ligarse con cualquier otro proveedor de seguridad, a través del conector correspondiente.
- Modulo de trazas: Se propone utilizar un módulo de gestión de trazas que genere Logs de aplicación con el formato definido por Ejie. Este módulo será opcional y no influirá en el resto de piezas de la arquitectura. Además, será capaz de generar Logs automáticos de diferente índole:
    - Logs de usuario: Trazas incluidas en código por los programadores.
    - Logs de incidencia: Trazas generadas automáticamente por el módulo de trazas cuando detecta una excepción no capturada.
    - Logs de aplicación: Trazas generadas automáticamente por el módulo de trazas, reflejando los pasos que una operación ha realizado a través de las diferentes capas del sistema de información.

4.2.13. Gestión de vulnerabilidades

La arquitectura debe encargarse de controlar las típicas vulnerabilidades de arquitecturas Web, como son el SQL Injection, Code Injection, Cross Site Scripting, etc.

En concreto, la arquitectura delega la responsabilidad de tapar dichas vulnerabilidades en los productos software en los que se basa. No obstante, si se descubre una vulnerabilidad en alguno de esos productos, se actualizará la versión de dicho producto, para así disponer del parche que tape dicho problema.

Además, se hará uso de las buenas prácticas recomendadas por los fabricantes para no dar pié a exponer las vulnerabilidades que las diferentes tecnologías puedan tener.

4.2.14. Empaquetado y organización del código

La arquitectura define un modelo de empaquetado, organización del código y despliegue que facilita el desarrollo, mantenimiento y el desacoplamiento del código y de los artefactos generados.

En cuanto a las aplicaciones que se desarrollen sobre la arquitectura, se homologa el formato de proyectos divididos por módulos de despliegue y distribución:

- Módulo JAR: Las Clases de cada aplicación estarán contenidas en un artefacto JAR. Dicho artefacto se extrae de la compilación del proyecto Java que contiene todas las clases desarrolladas ad-hoc para una aplicación. Estos artefactos deberán ser utilizados en aplicaciones de mayor nivel (aplicaciones Web o Enterprise).
- Módulo WAR: El contenido dinámico Web se encuentran contenidos en un artefacto WAR. Este WAR se corresponde con un proyecto dinámico Web que contiene ficheros de configuración, controladores y Servlets o JSPs.
- Módulo EJB: Contiene Enterprise Java Beans que se encargan exclusivamente de recubrir la capa de servicios, dotándola de la capacidad de realizar llamadas remotas. Solo aplica en Servidores de Aplicaciones JEE.
- Módulo EAR: Este módulo puede contener varias aplicaciones Web (artefactos WAR) y varias aplicaciones Java (artefactos JAR o EJB) en sí mismo. Además, puede contener librerías comunes a todos los artefactos que contiene y ficheros de configuración de la aplicación. Se corresponde con un proyecto Enterprise que incluye todo el contenido mencionado. Se podrá desplegar en un Servidor de Aplicaciones.

Los proyectos Java de cada aplicación tendrán que estar divididos en paquetes. Existirá un paquete por cada capa de la aplicación.

Los proyectos Web contendrán el contenido Web estático al primer nivel y el contenido dinámico que se ejecuta en servidor (junto con los ficheros de configuración) a segundo nivel, bajo WEB-INF.

4.2.15. Orientación a la Web

La arquitectura propuesta esta específicamente diseñada para aplicaciones Web ricas (RIA) basadas en navegador. La parte visual de dichas aplicaciones está compuesta por componentes gráficos que cumplen con las reglas de accesibilidad definidas en el documento de Accesibilidad de Aplicaciones, que básicamente propone lo siguiente:

"…se propone adoptar las recomendaciones de la WCAG 2.0 y las especificaciones WAI-ARIA para poder presentar a los usuarios unas aplicaciones más ricas, interactivas y potentes haciendo uso de tecnologías accesibles actuales como puede ser AJAX."

Por otro lado, la capa visual de las aplicaciones hace uso intensivo de Ajax, de forma que se establecen múltiples comunicaciones HTTP asíncronas entre el servidor y los clientes RIA. Es por ello, que se

proporcionan mecanismos de (de)serialización del modelo de datos haciendo que los objetos Java que se utilizan en el servidor se sigan utilizando en el cliente, sin necesidad de que los datos que contienen hayan de ser pasados a otros objetos de transferencia. Por lo tanto, se puede afirmar que los mismos objetos Java que se crean en el servidor por cada petición realizada (objetos que por lo tanto cumplen con el patrón Prototype) cruzan todas las capas de la aplicación, desde la capa de acceso a datos hasta la capa de presentación.

La usabilidad de los componentes RIA está definida en el documento de Análisis de Patrones de Presentación. En el se definen un conjunto de patrones que responden a necesidades básicas de la mayoría de aplicaciones.

En cuanto a la parte de la capa de presentación que se encarga de gestionar las peticiones de los clientes RIA, es decir el Controlador, se propone el uso de componentes que gestionen entidades de negocio a través de URIs estandarizadas y acciones HTTP. Además, los datos que se envían en las acciones HTTP han de estar representados en un formato estandarizado, como puede ser XML o JSON. Si se tienen en cuenta todas estas características, se obtiene que los Controladores que se utilizan están orientados al paradigma REST (Representational State Transfer).

Por lo tanto, la capa de presentación se compone de componentes RIA que consumen servicios Web RESTFul a través de peticiones asíncronas Ajax. El consumo de los recursos Web (o gestión de entidades de negocio) ha de darse siguiendo con el siguiente esquema de interacción:

- Se usa POST para crear un recurso en el servidor.
- Se usa GET para obtener un recurso.
- Se usa PUT para cambiar el estado de un recurso o actualizarlo.
- Se usa DELETE para eliminar un recurso.

De esta manera, una petición clásica de creación de un usuario que podría tener este aspecto:

```
GET /agregarusuario?nombre=Ane HTTP/1.1
```

Pasa a tener este aspecto muchísimo más elegante y explicito en REST:

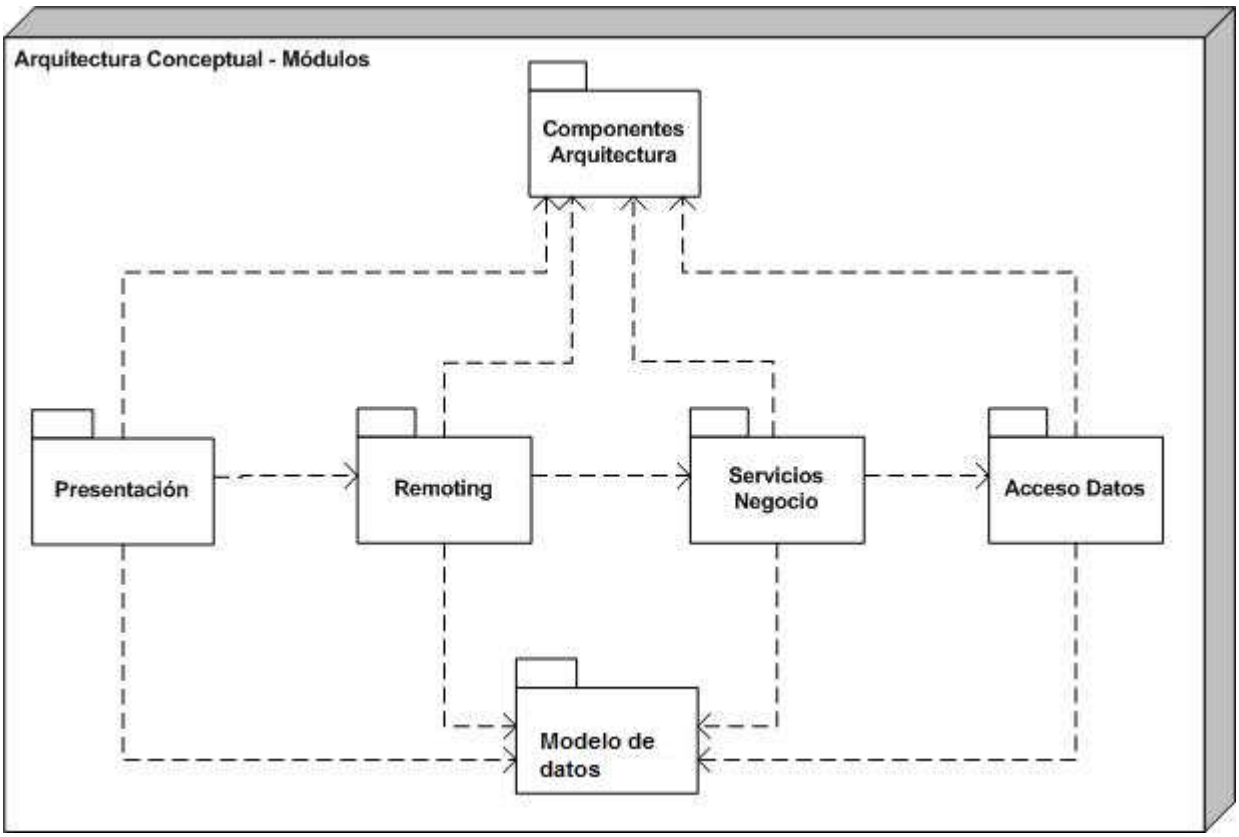
```
POST /usuarios HTTP/1.1
Host: miservidor
Content-type: application/xml

<usuario>
    <nombre>Ane</nombre>
</usuario>
```

Por último, hay que tener en cuenta la los teóricos de REST sugieren que las peticiones HTTP realizadas han de ser autocontenidas, ya que el servidor Web habría de mantener estado. Sin embargo, no se propone el uso de REST como una limitación, sino como una herramienta, por lo que no se restringe el uso de las variables de sesión que proporciona el API de Servlets, ni tampoco se elimina el uso de las Cookies.

4.3 Vistas de la arquitectura conceptual

Partiendo de las premisas teóricas establecidas en la arquitectura conceptual, en los siguientes puntos se describe mediante las vistas acordadas en el punto inicial, la arquitectura conceptual resultante.

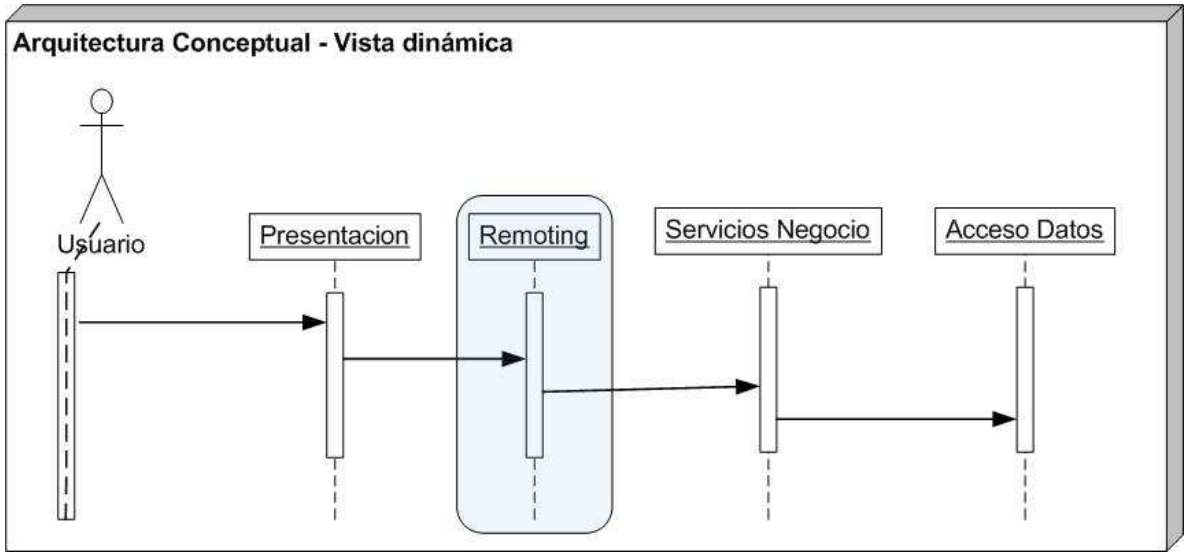
4.3.1. Vista estática o estructural



Esta figura muestra claramente los grupos de componentes y las relaciones que existen entre ellos. El paquete componentes de arquitectura recoge las funcionalidades que aporta esta que básicamente coincide con los aspectos (seguridad, trazas, transaccionalidad y validación).

Lo más destacable en cuanto a relaciones es que las capas expuestas en el centro únicamente se relacionan con la siguiente capa de izquierda a derecha. De esta forma se facilita la sustitución de implementaciones, ya que aunque la figura no lo muestra, las relaciones se establecen por interfaces y no por implementaciones. De esta forma se consigue un acoplamiento mínimo. Por otro lado tanto los componentes de la arquitectura como la capa de modelo de datos son accesibles desde el resto. La función de los primeros es simplificar el desarrollo del resto de capas mientras que los componentes de la capa de modelo de datos ofrecen un lenguaje común dentro del dominio del negocio en el que se enmarca la aplicación. Finalmente destacar que la capa de Remoting es opcional. Su función es puramente tecnológica y no de negocio y se aplicará a aquellas aplicaciones que deseen exportar funcionalidad a otras aplicaciones.

4.3.2. Vista dinámica



El esquema dinámico es el omnipresente en los desarrollos por capas. Destacar únicamente como la capa de funciones de integración es opcional como ya se indicó anteriormente.

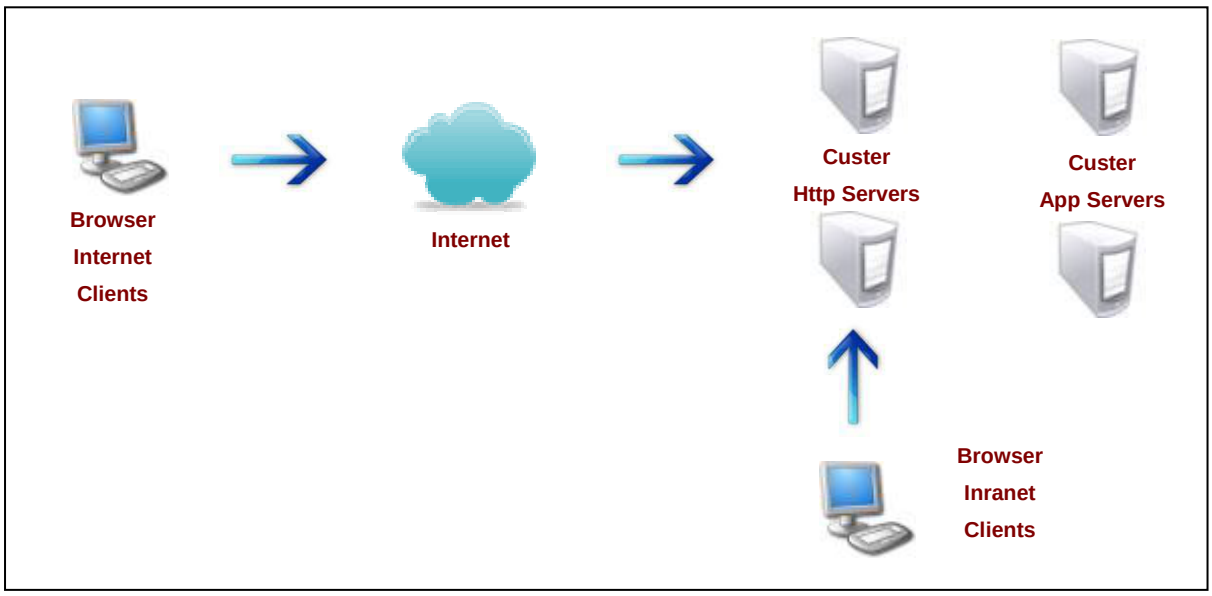
4.3.3. Vista de despliegue

La arquitectura da solución a dos entornos diferentes intranet e internet. Sin embargo esto no afecta a la forma en la que las aplicaciones son empaquetadas para sus despliegues (artefactos WAR y EAR anteriormente mencionados).

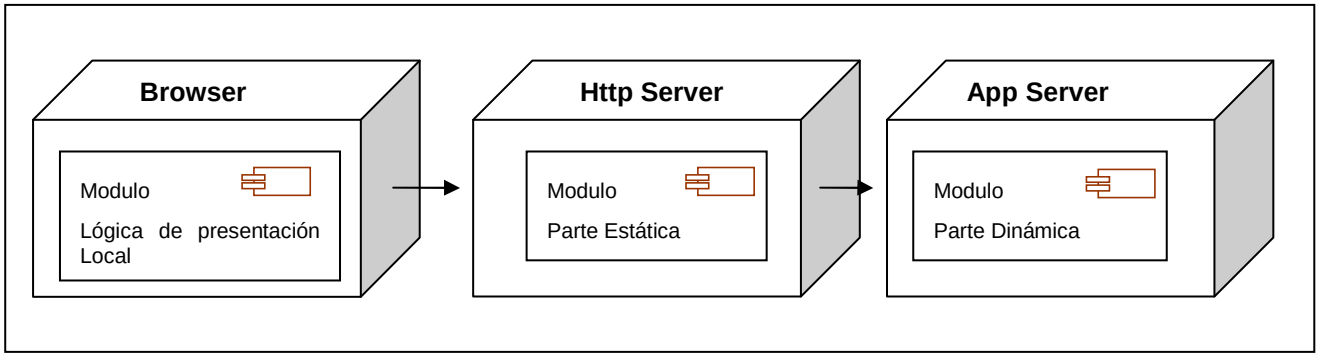
Las aplicaciones tendrán dos partes básicamente. La parte estática de estas residirá en un Http Server (clusterizable) y será el punto de entrada tanto para accesos desde internet como desde intranet (no hablamos de comunicaciones si no de los elementos que entran a formar parte del despliegue de la aplicación). La parte de lógica de la aplicación se desplegará en los servidores de aplicaciones (clusterizables también). Es en esta parte en la que el abanico de posibilidades se abre.

Aunque no a la hora de desplegar, pero sí, a la hora de ejecutar la aplicación, parte de la lógica de presentación se desplegará en los clientes para mejorar la experiencia de usuario tanto en refrescos de pantallas como en tiempo de espera de ejecución.

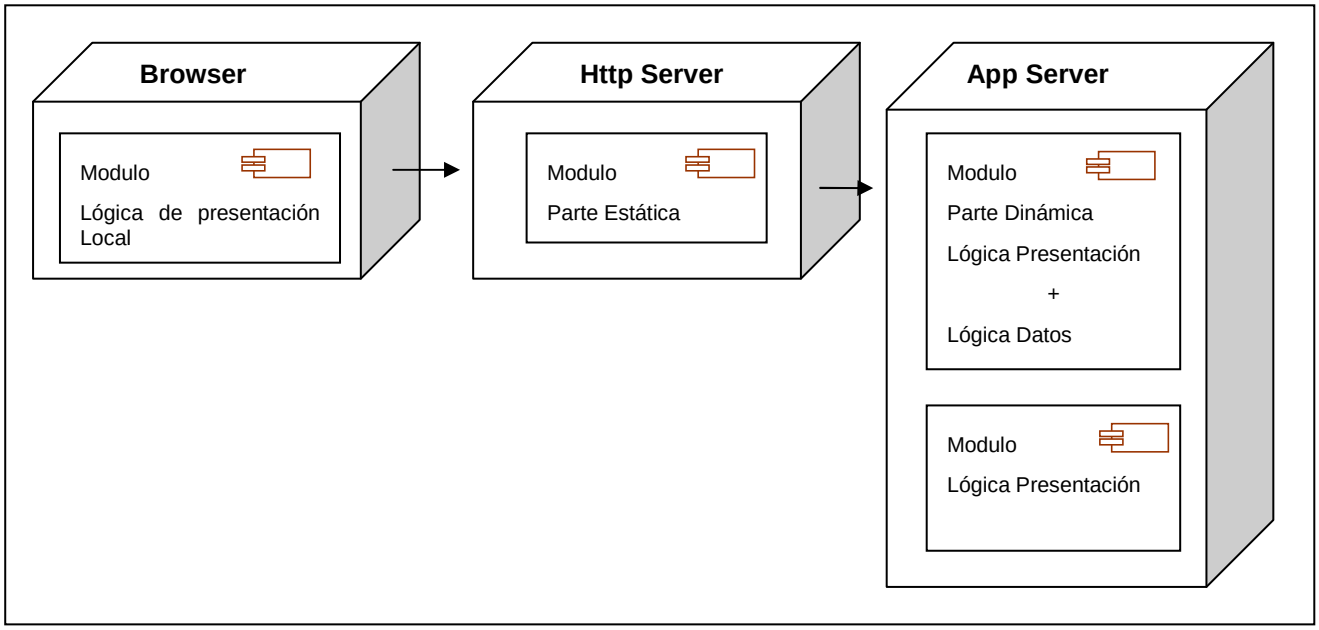
El servidor http delegará en el servidor de aplicaciones cuando sea necesario pero no estará directamente accesible para los clientes browser.



Ya dentro de un servidor de aplicaciones podremos desplegar aplicaciones distribuidas o de forma compacta o local. Una aplicación en un único módulo con lógica de presentación, negocio y datos en un único módulo. La conexión entre sus elementos será local en todos los casos minimizando la complejidad de los accesos y maximizando el rendimiento.



En otros casos, generalmente cuando necesitemos compartir información entre aplicaciones, podremos optar por dividir la lógica en varios módulos de despliegue (dentro de un mismo servidor o en distintos servidores). Esta opción nos aportará un mayor grado de integración y por tanto de reutilización que la anterior a cambio de añadir complejidad en el despliegue y en el uso de las comunicaciones.

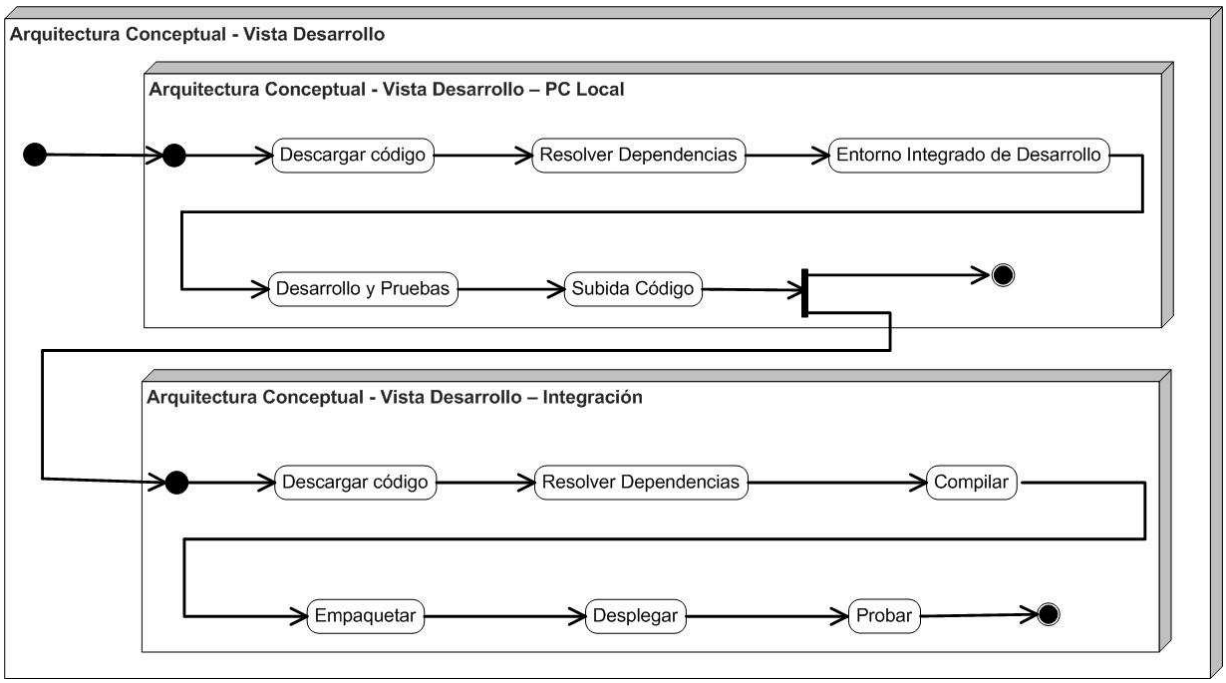


4.3.4. Vista de desarrollo

La arquitectura define que el desarrollo de aplicaciones ha de ser modular e iterativo, de manera que a medida que se va desarrollando el Software, este sea desplegado y testeado en un entorno de Integración con características muy similares a las del entorno de producción final. De esta manera, se favorece el desarrollo de Software correctamente testado, que cumpla con los requisitos del entorno real.

En cada una de estas iteraciones, se tendrán en cuenta como mínimo dos escenarios, que son el entorno de PC local del que dispondrá cada uno de los desarrolladores y el entorno de Integración, el cual será común para todos los desarrolladores de una aplicación.

- Entorno PC Local: Corresponde con el PC de cada desarrollador. Supone el punto de inicio de cada iteración del proceso de desarrollo. Este tendrá que disponer de las siguientes prestaciones:
    - Cliente que permita interactuar con el repositorio de código.
    - Cliente que permita interactuar con el repositorio de dependencias.
    - Entorno Integrado de Desarrollo.
        - Maquina Virtual de Java compatible.
        - Servidor de Aplicaciones / Contenedor de Servlets compatible.
- Entorno de Integración: Se trata de un servidor central que proporciona un entorno muy similar al entorno final de producción. Tendrá que disponer de las siguientes prestaciones:
    - Cliente que permita interactuar con el repositorio de código.
    - Cliente que permita interactuar con el repositorio de dependencias.
    - Plataforma de ejecución de tareas que permita compilar, empaquetar y desplegar el código.
        - Maquina Virtual de Java compatible.
        - Servidor de Aplicaciones / Contenedor de Servlets compatible.


Arquitectura Conceptual - Vista Desarrollo
Arquitectura Conceptual - Vista Desarrollo – PC Local
Descargar código
Resolver Dependencias
Entorno Integrado de Desarrollo
Desarrollo y Pruebas
Subida Código
Arquitectura Conceptual - Vista Desarrollo – Integración
Descargar código
Resolver Dependencias
Compilar
Empaquetar
Desplegar
Probar