

Arquitecturas Software

Desarrollo de Software Basado en Componentes y Servicios

M.I. Capel

ETS Ingenierías Informática y
Telecomunicación
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada
Email: manuelcapel@ugr.es

TOADADTR
Máster Universitario en Desarrollo de Software



Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Índice

- 1 Conceptos fundamentales
- 2 Modelos de Arquitecturas Software
- 3 Estilos Arquitectónicos
- 4 Patrones Arquitectónicos y Calidad del Software
- 5 Arquitecturas Orientadas a Servicios
- 6 Arquitecturas Dirigidas por Eventos
- 7 Grid Computing

Conceptos fundamentales

Modelos de Arquitecturas Software

Estilos Arquitectónicos

Patrones Arquitectónicos y Calidad del Software

Arquitecturas Orientadas a Servicios

Arquitecturas Dirigidas por Eventos

Grid Computing

Componentes

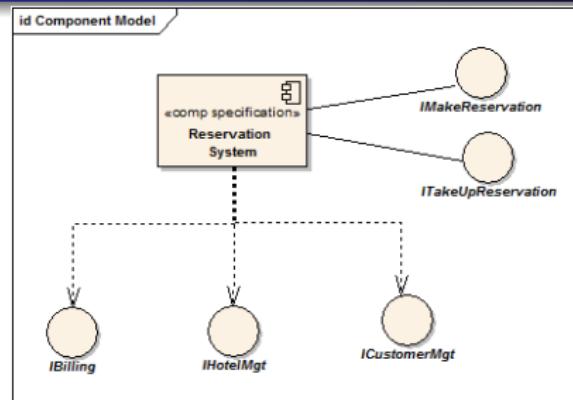


Figura: Relación entre componente y middleware

Componente Software

“Una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio”.

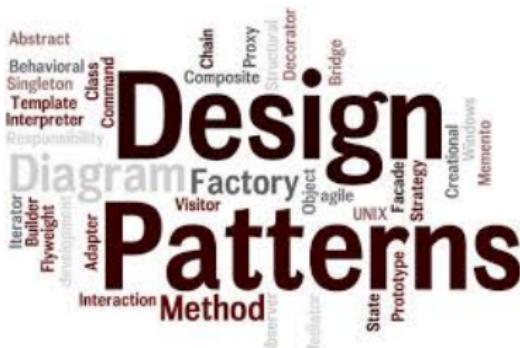
Elementos de un componente



- 1 Extensibilidad independiente
- 2 Requisitos/requerimientos: determinan las necesidades en cuanto a recursos
- 3 Interfaces: conjuntos de operaciones



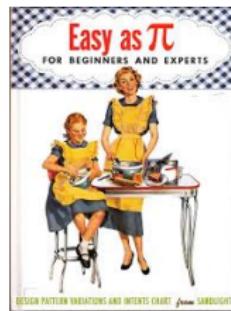
Patrón de Diseño



Definición

“Un elemento de análisis de software que ofrece una solución abstracta a un problema de diseño que aparece muy frecuentemente, expresada mediante un conjunto de relaciones e interacciones entre componentes”

Patrones de Diseño



- Son útiles para diseñar la arquitectura de los marcos de trabajo
- Se utilizan también como documentación de éstos
- Existen catálogos con patrones de diseño, tanto dependientes como independientes del dominio



Conceptos fundamentales

Modelos de Arquitecturas Software

Estilos Arquitectónicos

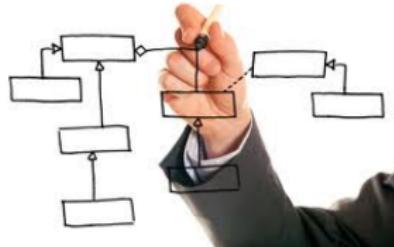
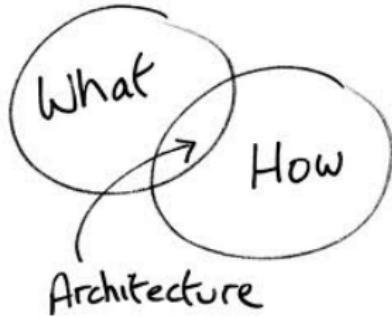
Patrones Arquitectónicos y Calidad del Software

Arquitecturas Orientadas a Servicios

Arquitecturas Dirigidas por Eventos

Grid Computing

Arquitectura Software

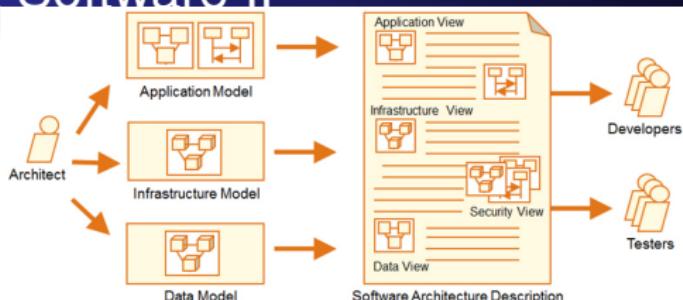


Definición

“Representación de alto nivel de la estructura de un sistema o aplicación, que describe las partes que la integran, las interacciones entre ellas, los patrones que supervisan su composición y las restricciones a la hora de aplicar estos patrones”.



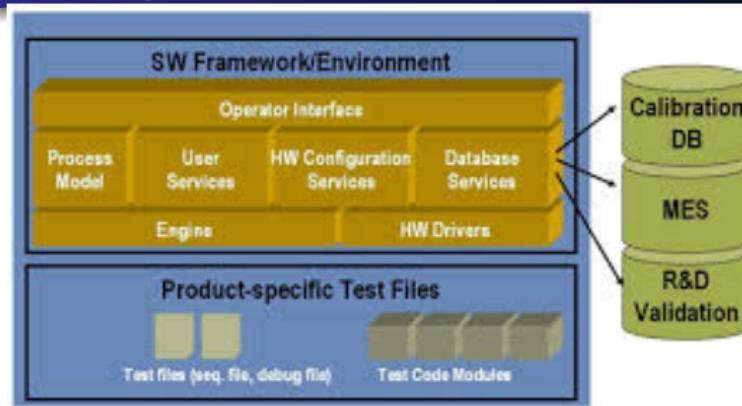
Arquitectura Software II



Objetivos

- Comprender y gestionar mejor la estructura de aplicaciones complejas
- Reutilizar dicha estructura o partes de ella
- Planificar la evolución de la aplicación, identificando las partes mutables e inmutables de la misma así como los costes de los futuros posibles cambios

Marcos de Trabajo



Definición

“Diseño reutilizable de todo o parte de un sistema, representado por un conjunto de clases abstractas y la forma en que estas interactúan”.



Marcos de Trabajo II

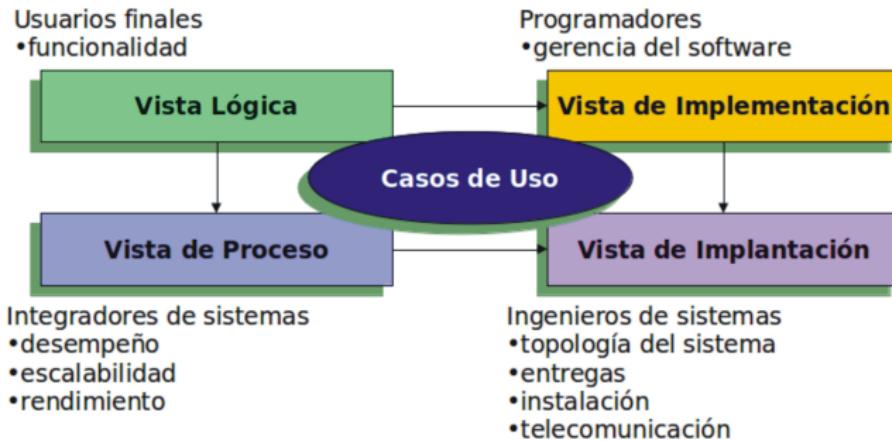


Características

- Pueden ser vistos como un esqueleto de clases que debe ser adaptado por los desarrolladores a necesidades concretas de una aplicación
- Un marco de trabajo se extiende a partir de un conjunto de punto de entrada o ganchos, que conforman la parte variable de la aplicación



Modelos abstractos de Arquitecturas de Software

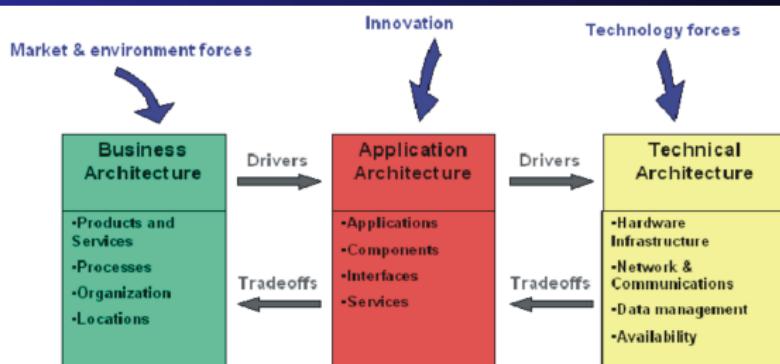


Modelo 4+1 Vistas de Krutchen

En la actualidad el desarrollo de sistemas se centra en la arquitectura software, que es especificada utilizando el modelo siguiente



Ciclo de Arquitecturas de Negocio (ABC)



- Ciclo de retroalimentación entre una AS y un negocio
- Según [Bass et al., 2012], estos y otros mecanismos de retroalimentación forman lo que se llama “ABC”
- En [Kruchten, 1995] también se indica esta relación de la AS con el negocio: se habla de *tres campos* de arquitectura muy relacionados durante el desarrollo.

Estilos Arquitectónicos

- Define una familia de sistemas de software en términos de su organización estructural.
- Representa los componentes y las relaciones entre ellos con las restricciones de su aplicación y las asociaciones y reglas de diseño para su construcción
- Define un vocabulario de componentes y tipos de conectores.

Ejemplos de Estilos Arquitectónicos

Pipes and filters, Tipos de datos abstractos y OO, Repositorios, Capas, Basados en Eventos, Intérpretes, etc.

Estilos II

- La imposición de ciertos estilos arquitectónicos mejora o disminuye las posibilidades de satisfacción de ciertos atributos de calidad del sistema [Jansen and Bosch, 2005]
- Cada estilo propicia atributos de calidad, y la decisión de implementar alguno de los existentes depende de los requerimientos de calidad del sistema.
- Un criterio importante del éxito de los estilos es la forma en que estos alcanzan de manera satisfactoria los objetivos de la Ingeniería de Software [Buschmann and et al., 2004].

Diferencias entre estilos y patrones arquitectónicos

Estilo Arquitectónico

- Sólo describe el esqueleto estructural y general de las aplicaciones
- Son independientes del contexto al que puedan ser aplicados
- Cada estilo es independiente de los otros
- Expresan técnicas de diseño desde una perspectiva independiente de la situación actual de un diseño
- Pueden comprenderse también como una clasificación de los sistemas software

Patrón Arquitectónico

- Varios rangos de escala, comenzando con la definición de la estructura básica de una aplicación
- Necesitan de la definición de un contexto del problema
- Dependen de patrones más pequeños, con los que interactúan; o de patrones que los contengan
- Expresan un problema recurrente de diseño, muy específico, presentando una solución, desde el punto de vista del contexto en el que se presenta
- Son soluciones generales a problemas comunes

Patrón Arquitectónico

Definición de Buschmann (1996) [[Buschmann and et al., 2004](#)]

Una regla que consta de tres partes, la cual expresa una relación entre un contexto, un problema y una solución.

- ① *Contexto:* Es una situación de modelado de una parte del sistema en la que aparece un problema de diseño.
- ② *Problema:* Es un conjunto de fuerzas que aparecen repetidamente en el contexto.
- ③ *Solución:* Es una configuración que equilibra estas fuerzas.

Patrones II

Para Buschmann son plantillas para arquitecturas de software concretas, que especifican las propiedades estructurales de una aplicación y tienen un impacto en la arquitectura de subsistemas.

El uso de ciertos mecanismos, como los estilos y patrones arquitectónicos, permite mejorar las características de calidad en el software [[Jansen and Bosch, 2005](#)], bien sean éstas observables o no en tiempo de ejecución [[Bass et al., 2012](#)].

- El ámbito del patrón es menos general que el del estilo arquitectónico
- Un patrón impone una regla a la arquitectura, describiendo cómo el software gestionará algún aspecto de su funcionalidad (p.e.: la concurrencia).
- Los patrones arquitectónicos tienden a abordar problemas *comportamentales* específicos dentro del contexto de una arquitectura
- Los patrones y los estilos arquitectónicos se pueden utilizar de forma conjunta para dar forma a la estructura completa de un sistema.

Diferencias según el nivel de abstracción

Nivel de Abstracción



Estilo Arquitectónico

- ✓ Descripción del esqueleto estructural y general para aplicaciones
- ✓ Es independiente de otros estilos
- ✓ Expresa componentes y sus relaciones

Patrón Arquitectónico

- ✓ Define la estructura básica de una aplicación
- ✓ Puede contener o estar contenido en otros patrones
- ✓ Provee un subconjunto de subsistemas predefinidos, incluyendo reglas y pautas para su organización
- ✓ Es una plantilla de construcción

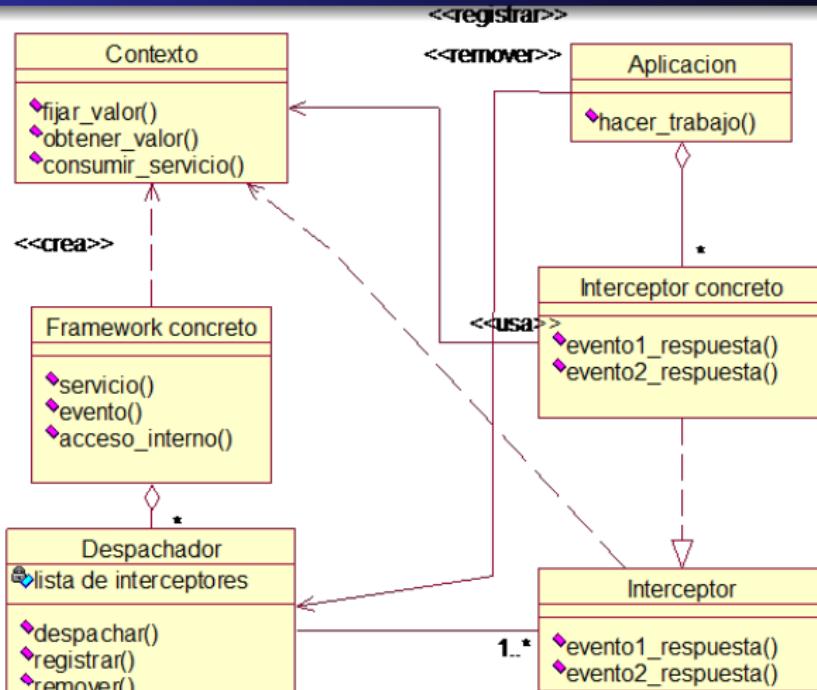
Patrón de Diseño

- ✓ Esquema para refinar subsistemas o componentes

Patrón Interceptor

- Permite a determinados servicios ser añadidos de manera transparente a un marco de trabajo y ser disparados automáticamente cuando ocurren ciertos eventos
- Contexto: Desarrollo de marcos de trabajo que puedan ser extendidos de manera transparente
- Problema: Los marcos de trabajo, arquitecturas software, etc. ha de poder anticiparse a las demandas de servicios concretos que deben ofrecer a sus usuarios
- Integración dinámica de nuevos componentes sin afectar a la AS o a otros componentes
- Solución: Registro offline de servicios a través de una interfaz predefinida del marco de trabajo, posteriormente se disparan estos servicios cuando ocurran determinados eventos

Interceptor



Estructura de clases de “Interceptor”

- Un *FrameworkConcreto* que instancia una arquitectura genérica y extensible
- Los *Interceptores* que son asociados con un evento particular
- *InterceptoresConcretos* que especializan las interfaces del interceptor e implementan sus métodos de enlace
- *Despachadores* para configurar y disparar interceptores concretos
- Una *Aplicación* que se ejecuta por encima de un *framework concreto* y utiliza los servicios que éste le proporciona

Implicaciones en la calidad del patrón Interceptor

Beneficios	Atributo de calidad	Características ISO 9126
Cambiar/incluir servicios de un framework sin que sea preciso cambiarlo	Extensibilidad, Flexibilidad,Dinamismo	Mantenibilidad Facilita cambios
Añadir interceptores sin afectar al código de la aplicación	Acoplamiento	Mantenibilidad Facilita cambios
Obtención dinámica inform. del framework con intercept. y objetos-contexto	Monitorización Control	Tolerancia a fallas Uso de recursos
Infraestructura de servicios estratificada con interceptores correspondientes simétricos	Encapsulamiento	Mantenibilidad Facilita cambios,análisis
Reutilización de interceptores en diferentes aplicaciones	Reusabilidad	Mantenibilidad Facilita cambios

Implicaciones en la calidad 2

Inconvenientes	Atributo de calidad	Características ISO 9126
Difícil ajuste del número de despachadores e interceptores	Complejidad Flexib., extensibilidad	Facilita cambios Facilita análisis
Bloqueo aplicación por fallo del interceptor	Disponibilidad Modificabilidad	Mantenibilidad Madurez, Tolerancia Fallas
Degrado rendimiento por cascadas de interceptores	Rendimiento Bloqueo	Eficiencia, madurez Tolerancia fallas

- Insuficientes interceptores y despachadores reduce la flexibilidad y extensibilidad del framework concreto.
- Sistema demasiado grande e ineficiente, complejo de aprender, implementar, usar, y optimizar, utilizando demasiados interceptores
- Para evitar el bloqueo completo de la aplicación pueden utilizarse estrategias de *time-out*, pero esto puede complicar el diseño del framework concreto
- Las cascadas de intercepción pueden conllevar una degradación del rendimiento o un bloqueo de la aplicación

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio Facilidad de análisis	+	Reusabilidad Modificabilidad Encapsulamiento Extensibilidad Flexibilidad Acoplamiento Dinamismo
	Facilidad de cambio Facilidad de análisis	-	Extensibilidad Flexibilidad Complejidad Modificabilidad
Eficiencia	Tiempo de respuesta	-	Desempeño
	Uso de recursos	+	Monitoreo Control
Fiabilidad	Tolerancia a fallas	-	Disponibilidad Bloqueo
		+	Monitoreo Control
	Madurez	-	Disponibilidad Bloqueo

Resumen características de calidad

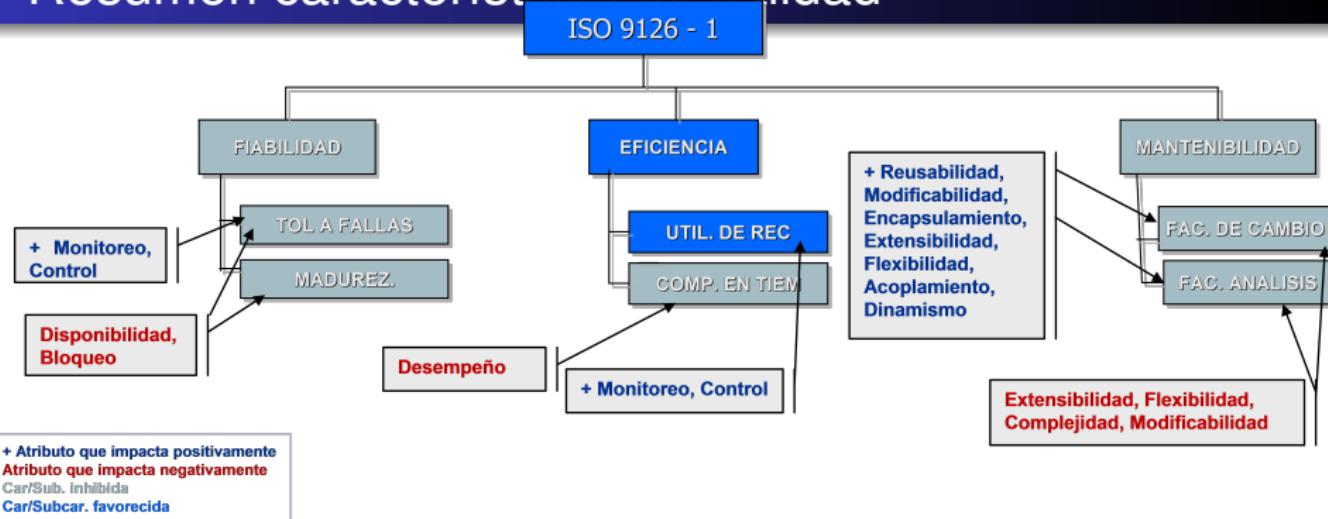
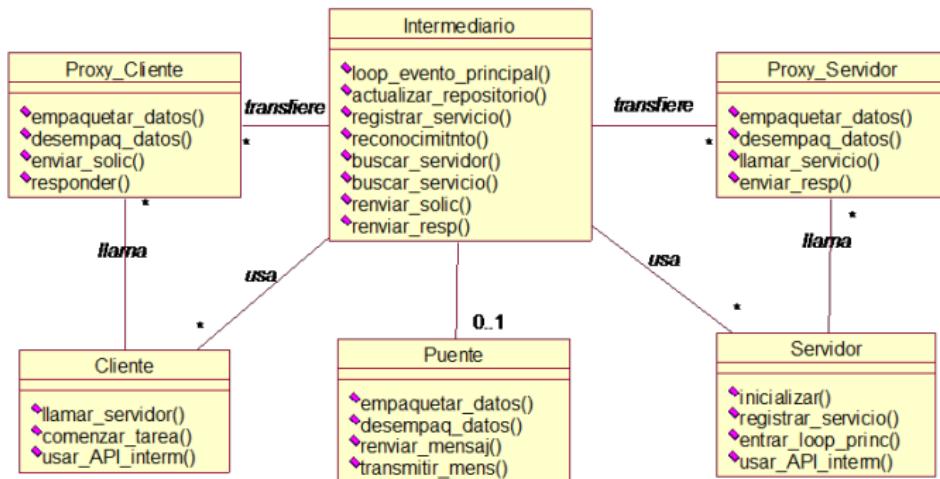


Figura: Características ISO del patrón “Interceptor”

Patrón Broker

- Para modelar sistemas distribuidos compuesto de componentes software totalmente desacoplados
- Contexto: cualquier sistema distribuido y, posiblemente, heterogéneo con componentes cooperando dentro de un sistema de información
- Problema: ¿Cómo estructurar componentes configurables dinámicamente e independientes de los mecanismos concretos de comunicación de un sistema distribuido?
- Solución: Introducir un componente *Broker* para mejor desacoplamiento entre clientes y servidores
- Las tareas de los Brokers incluyen la localización del servidor apropiado, envío de la petición al servidor y transmisión de los resultados

Broker



Estructura de clases del patrón Broker

- *Intermediario*: admite las solicitudes, asigna los servidores y responde a las peticiones de los clientes
- *Servidor*: se registra en el *Intermediario* e implementa el servicio
- *Cliente*: accede a los servicios remotos
- *Proxy Cliente* y *Proxy Servidor*, que proporcionan transparencia, ocultando los detalles de implementación del patrón
- *Puente*: le proporciona interoperabilidad al *Intermediario*

Implicaciones en la calidad del patrón Broker

Beneficios	Atributos de calidad	Característica ISO 9126
Separación código de comunicaciones.	Acoplamiento Modificabilidad	Facilita cambios Fac.análisis, pruebas
Independencia de la plataforma de ejecución para la aplicación	Escalabilidad	Facilita cambios Uso de recursos
Mejor descomposición del espacio del problema	Interoperabilidad Simplicidad	Interoperabilidad Facilita análisis
Independencia de la implementación concreta de cada capa	Modificabilidad Flexibilidad	Mantenibilidad Fac.cambios
Interacciones basadas en el paradigma de objetos	Transparencia	Funcionalidad Interoperabilidad
Arquitectura software muy flexible	Dinamismo	Facilita cambios Facilita análisis
Reducción complejidad de programación distribuida	Modificabilidad Flexibilidad	Facilita cambios Facilita análisis
Integración de tecnologías	Reusabilidad	Facilita cambios Facilita análisis
Distribución del modelo de	Interoperabilidad	Portabilidad

Implicaciones en la calidad 2

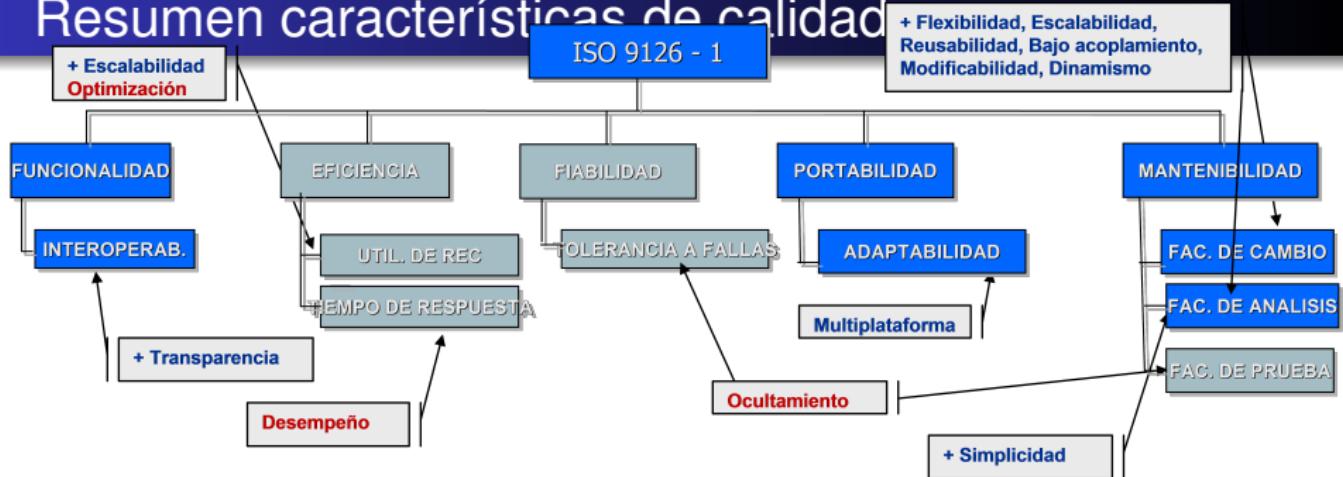
Inconvenientes	Atributos de calidad	Características ISO 9126
Empeora el rendimiento de la aplicación	Rendimiento	Eficiencia Tiempo respuesta
Impide la verticalidad en acceso a capas internas	Optimización Ocultamiento	Facilidad pruebas Uso recursos Tolerancia fallas

- Las capas de abstracción a las que conduce este patrón pueden perjudicar el desempeño
- Usar una capa separada del *Broker* puede ocultar los detalles sobre cómo la aplicación utiliza la capa más baja
- Eventualmente, optimizaciones específicas del rendimiento de algunas capas podrían resultar perjudicadas

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio Facilidad de análisis	+	Flexibilidad Escalabilidad Reusabilidad Bajo acoplamiento Modificabilidad Dinamismo
	Facilidad de análisis	+	Simplicidad
	Facilidad de prueba	-	Ocultamiento
Eficiencia	Uso de recursos	+	Escalabilidad
		-	Optimización
	Tiempo de respuesta	-	Desempeño
Funcionalidad	Interoperabilidad	+	Transparencia
Fiabilidad	Tolerancia a fallas	-	Ocultamiento
Portabilidad	Adaptabilidad	+	Multiplataforma

Resumen características de calidad



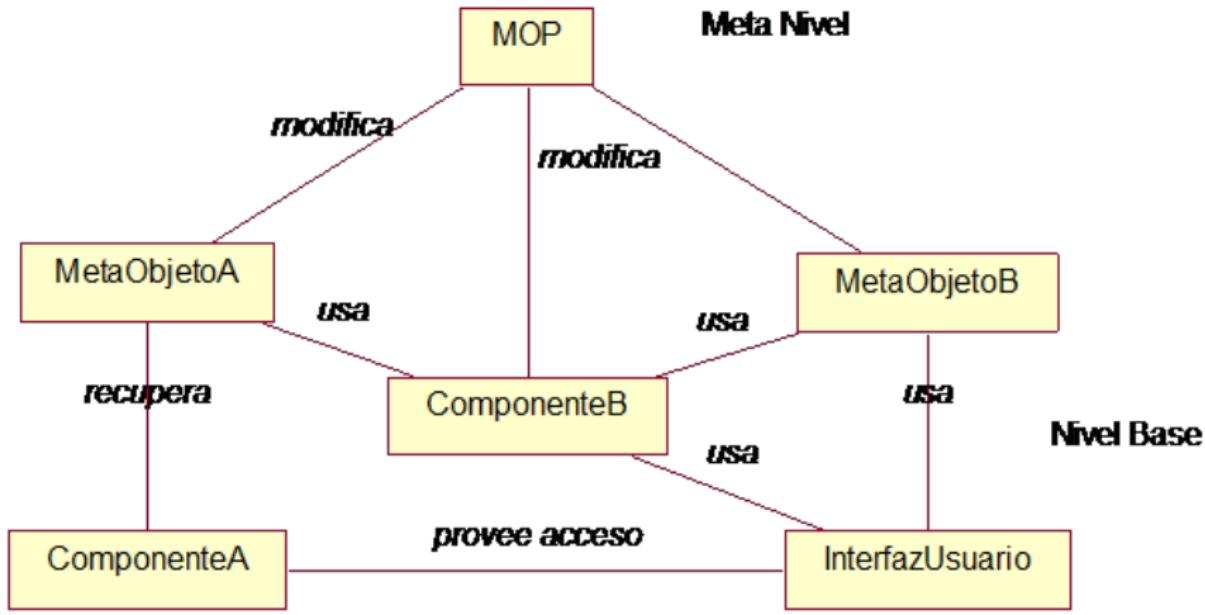
+ Atributo que impacta positivamente
 - Atributo que impacta negativamente
 Car/Sub. Inhibida
 Car/Subcar. favorecida

Figura: Características ISO del patrón "Broker"

Patrón Reflection

- Proporciona un mecanismo para cambiar la estructura y comportamiento de sistemas software dinámicamente
- Soporta la modificación de aspectos fundamentales, tales como estructuras y mecanismos de llamadas a funciones.
- Contexto: Cualquier sistema que necesita soporte para realizar cambios propios y para conseguir persistencia de sus entidades
- Problema: ¿Cómo se puede modificar el comportamiento de los objetos de una jerarquía dinámicamente sin afectar a los propios objetos en su configuración actual?
- Solución: Hacer que el software sea “auto-consciente” de su función y comportamiento, haciendo que los aspectos seleccionados sean accesibles para su adaptación y cambio dinámico.

Reflection



Estructura de clases del patrón Reflection

- *Meta Nivel*: proporciona la información acerca de las propiedades escogidas del sistema y hace que el sistema sea auto-consciente
- *Meta Objetos*: encapsulan y representan información acerca del software
- *Nivel Base*: incluye la lógica de la aplicación
- Los cambios mantenidos en el *Meta Nivel* afectan consecuentemente al comportamiento del *Nivel Base*
- La implementación del *Meta Nivel* utiliza *Meta Objetos* para mantenerse independiente de aquellos aspectos que son propensos a cambiar

Implicaciones en la calidad del patrón Reflection

Beneficios	Atributos de calidad	Características ISO 9126
Comprobación correctitud desde el nivel de <i>MetaObjetos</i>	Correctitud	Funcionalidad Precisión
Ejecución de los cambios desde el nivel de <i>MetaObjetos</i>	Dinamismo	Facilidad cambios Facilidad análisis
Modificación y extensión de componentes software facilitada	Modificabilidad Extensibilidad	Mantenibilidad Facilidad cambios
Cambios en el software del NivelBase facilitados	Modificabilidad Extensibilidad	Mantenibilidad Facilidad cambios
Asume modificaciones no explícitas del código fuente	Extensibilidad	Mantenibilidad Facilidad cambios
Soporte para muchos tipos de cambios	Modificabilidad	Mantenibilidad Facilidad cambios

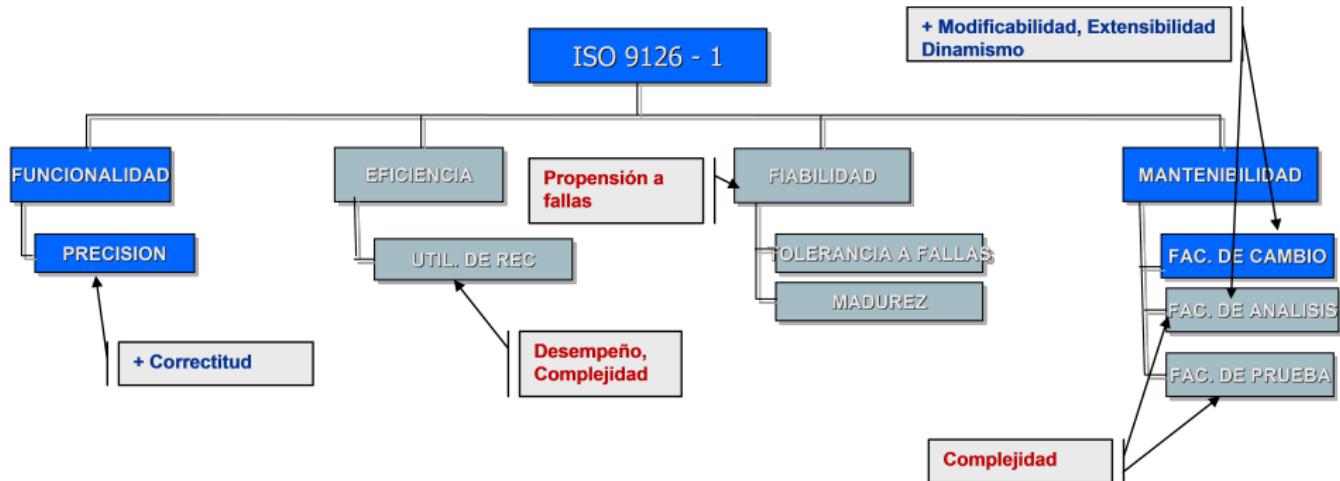
Implicaciones en la calidad 2

Inconvenientes	Atributos de calidad	Características ISO 9126
Complejidad de diseño e implementación por los niveles y protocolo meta–objetos	Complejidad	Facilidad análisis Facilidad pruebas
Demasiados meta–objetos en la aplicación final	Rendimiento	Uso recursos Eficiencia
Modificaciones en meta–nível pueden causar daños comportamiento del sistema	Fallas	Madurez Tolerancia a fallas
Rendimiento sistema puede ser afectado por complejidad diseño del patrón	Complejidad	Eficiencia Tiempo respuesta
Difícil adaptación a la evolución de los productos generados con el patrón	Adaptabilidad	Tiempo respuesta Eficiencia

Resumen características de calidad

Característica	Sub-característica	Impacto	Atributo
Mantenibilidad	Facilidad de cambio	+	Modificabilidad
	Facilidad de análisis		Extensibilidad
	Facilidad de análisis		Dinamismo
Funcionalidad	Precisión	+	Complejidad
Eficiencia	Uso de recursos	-	Correctitud
Fiabilidad	Madurez	-	Desempeño
	Tolerancia a fallas		Complejidad
			Propensión a fallas

Resumen características de calidad



+ Atributo que impacta positivamente
 Atributo que impacta negativamente
 Car/Sub. Inhibida
 Car/Subcar. favorecida

Figura: Características ISO del patrón "Reflection"

Concepto de Servicio

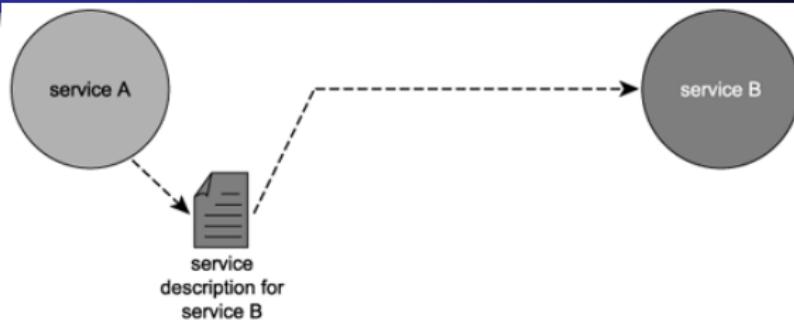


Figura: Acceso entre servicios

Servicio

Unidad de funcionalidad, independiente, autocontenido y débilmente acoplada a otros.



Concepto de Servicio II

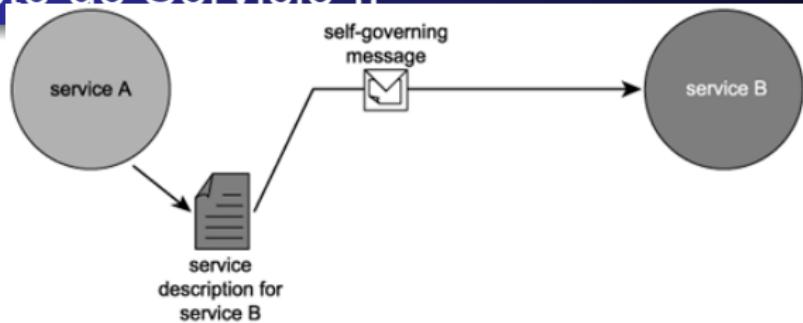


Figura: Acceso entre servicios

- Descripción de servicios y acoplamiento débil
- Objetivo de un marco de trabajo de comunicaciones
- Mensajes como *unidades de comunicación independientes*
- Papel de la *orientación a servicios*

SOA

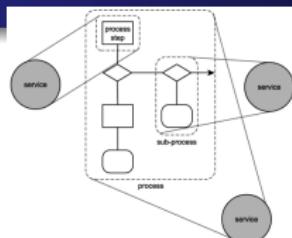
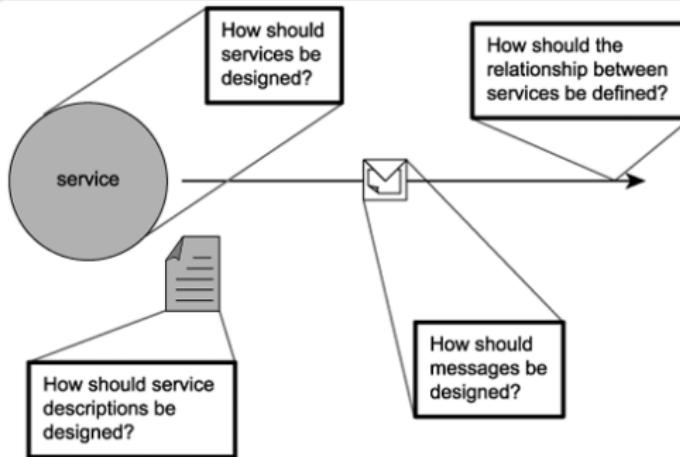


Figura: Aplicación con varios servicios

Motivación de utilizar SOA

- Evitar acoplamiento excesivo entre servicios
- Propician la estandarización y el lenguaje común entre proveedores y consumidores
- Nuevo principio útil para el diseño arquitectónico
- Encapsulación *ágil* de la lógica de negocio
- Extensibilidad independiente de los servicios

Arquitecturas orientadas a servicios



Estas arquitecturas abordan los problemas de diseño de sistemas software complejos aplicando los principios de *orientación a servicios*



Definiciones de SOA primitivo

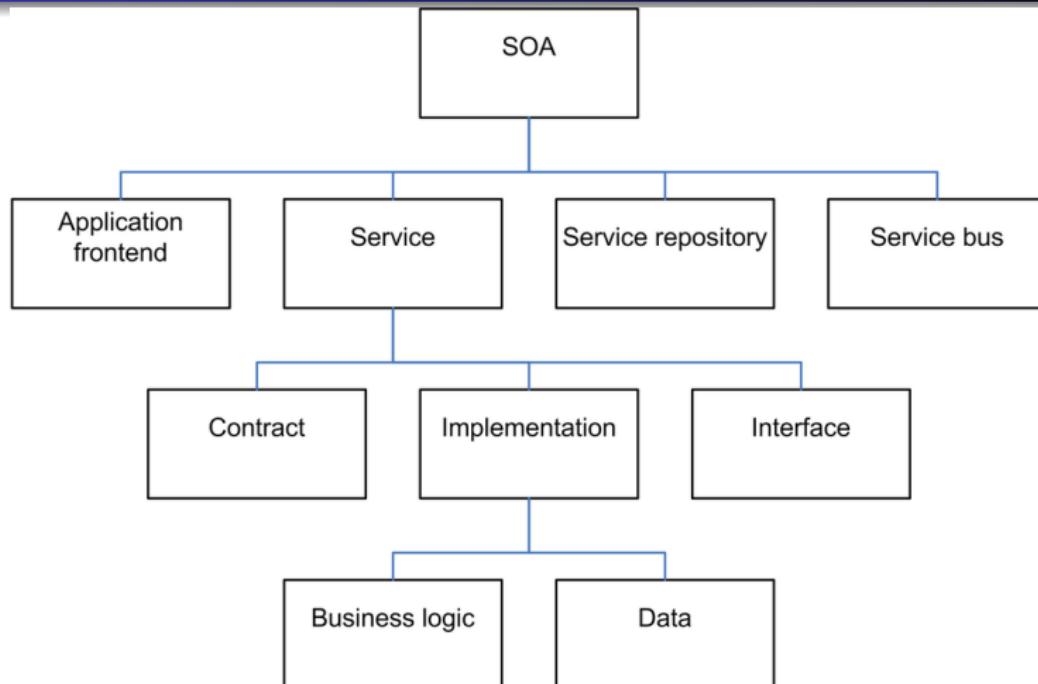
OASIS

“A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations”

Principios de Orientación a Servicios

- Bajo acoplamiento
- Contrato de servicio
- Autonomía
- Abstracción
- Reusabilidad
- Composicionalidad
- Ausencia de estado
- Facilidad de descubrimiento

Elementos de un SOA



Tecnologías basadas en apertura

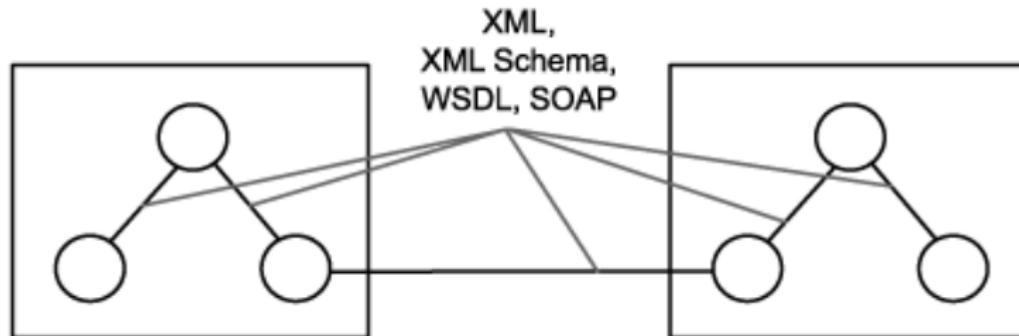


Figura: Tecnologías que superan los límites de la aplicación

Interoperabilidad entre plataformas

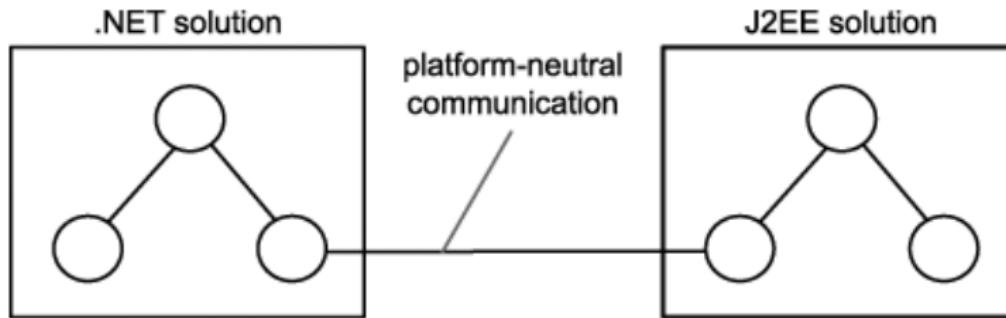


Figura: Tecnologías de plataforma interoperables

Descubrimiento de servicios

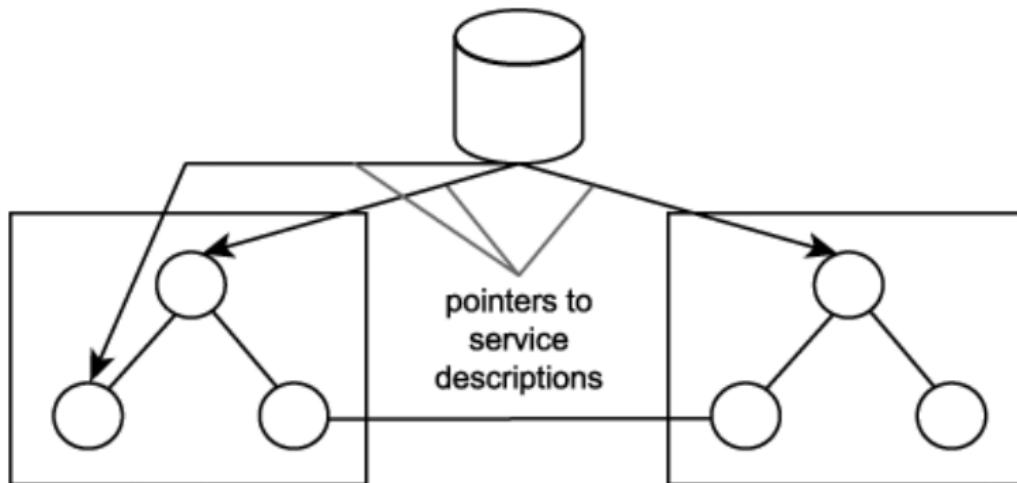


Figura: Mecanismo de descubrimiento de servicios implementado con registros

Integración de aplicaciones y servicios

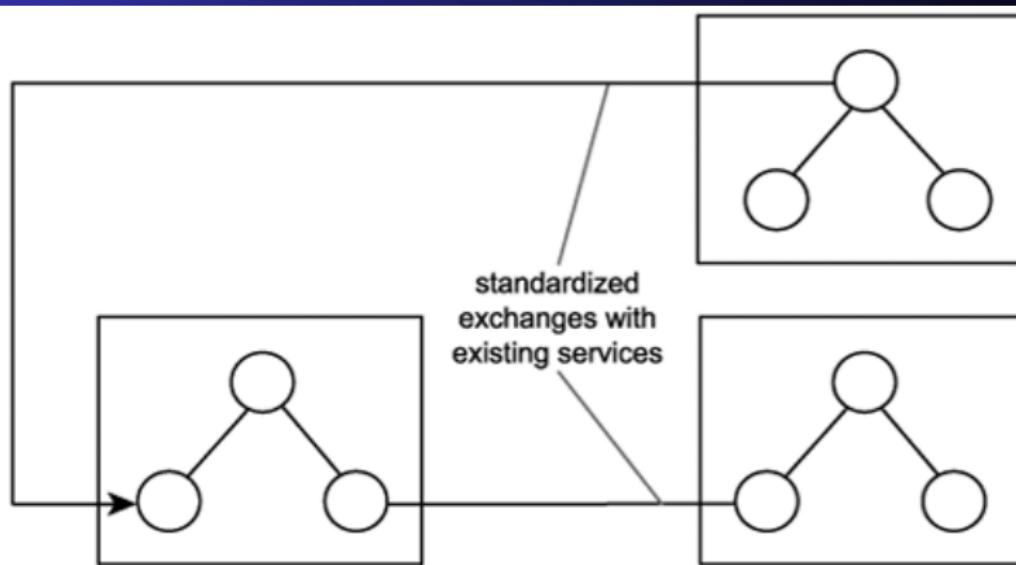


Figura: Servicios interoperables como potenciales puntos finales de integración

Soluciones híbridas

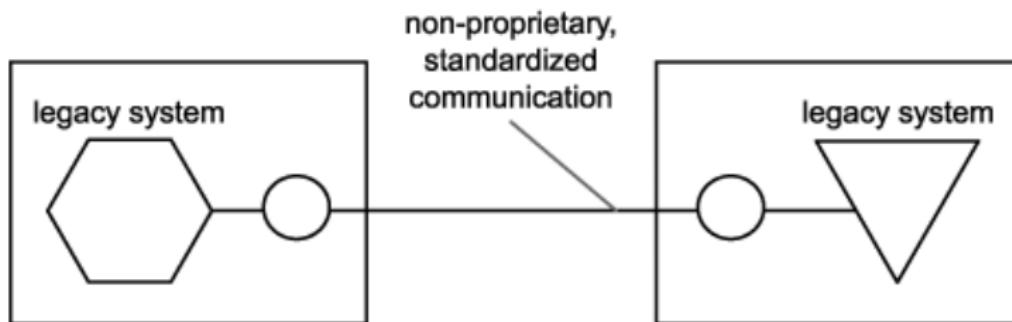


Figura: Federación estándar de sistemas legados mediante SOA

Reutilización

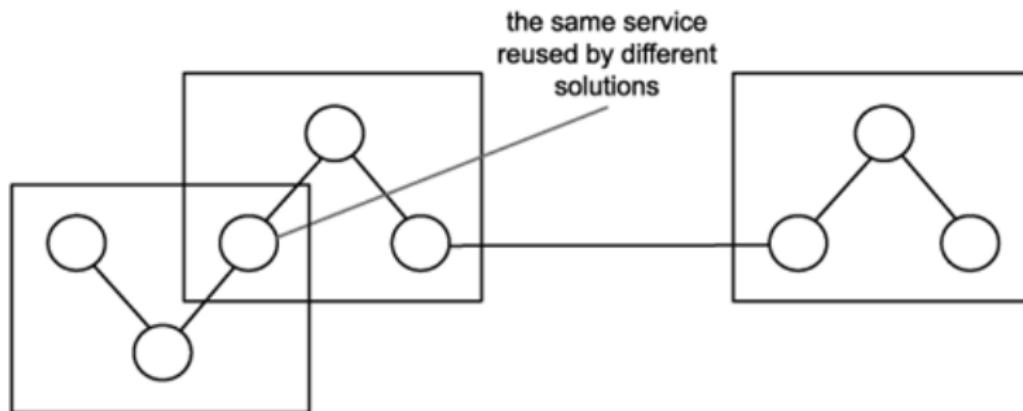


Figura: Reutilización inherente óptima

Niveles de reutilización

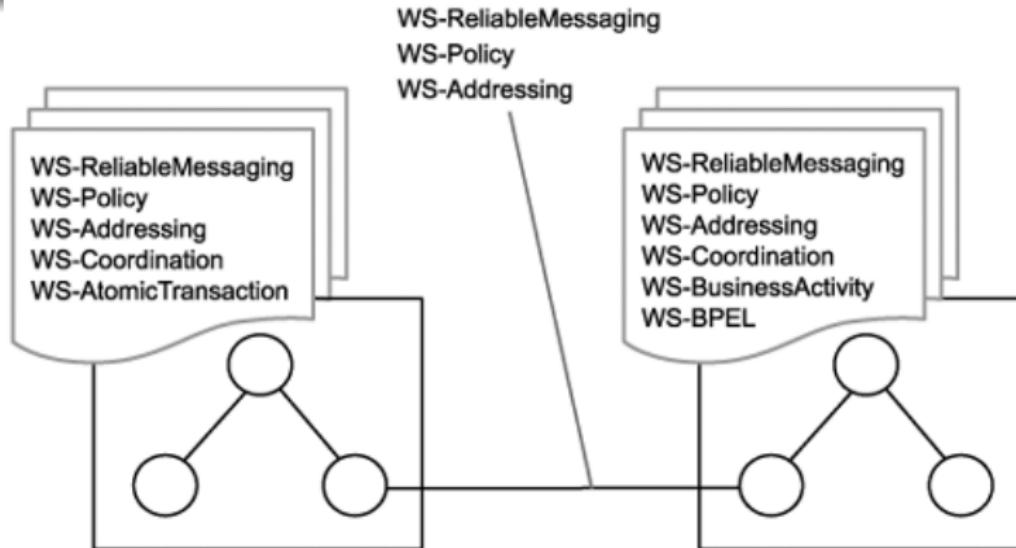


Figura: Composición de soluciones interoperable y extensible con escalabilidad ilimitada

Extensibilidad

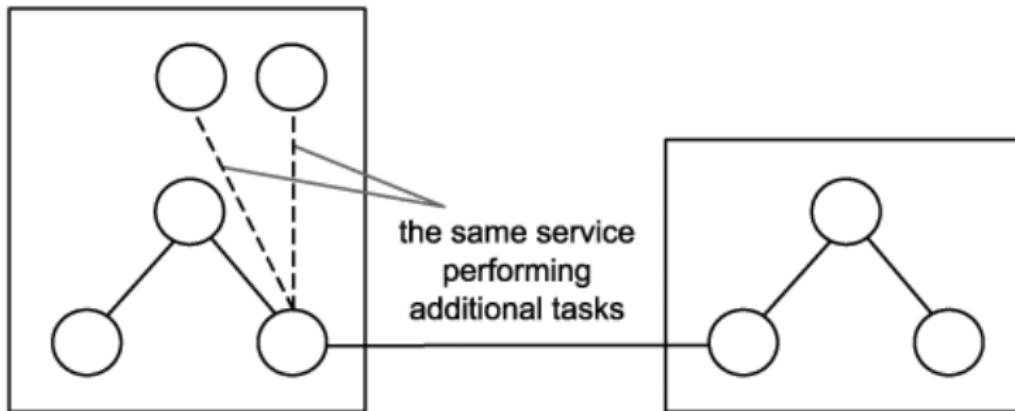


Figura: Expansión máxima de funcionalidad con mínimo impacto

Definición actual de SOA

SOA contemporáneo

“Representa una arquitectura con facilidad para la composición, federada, extensible y abierta, que promueve la orientación a servicios y que se compone de servicios potencialmente reutilizables, descubribles, interoperables, de varios vendedores, capaces de proporcionar calidad de servicio y autónomos, que son implementados mediante servicios Web”.

Encapsulación de la “lógica de negocio”

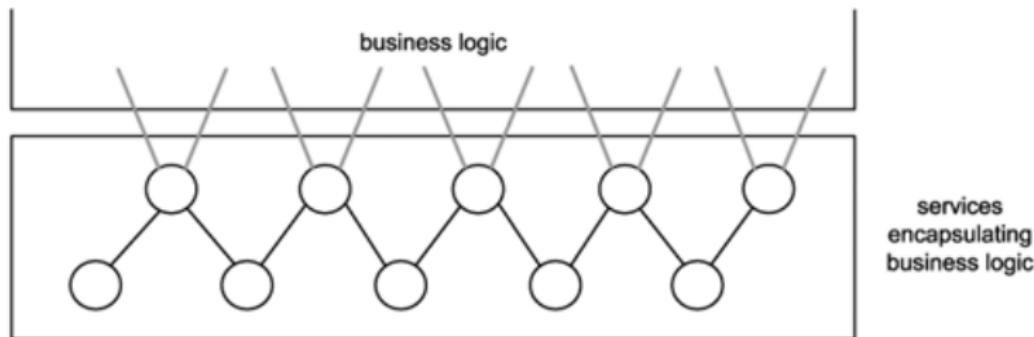


Figura: Capa de servicios que encapsulan la lógica de negocio de una aplicación

Encapsulación de la “lógica de negocio” II

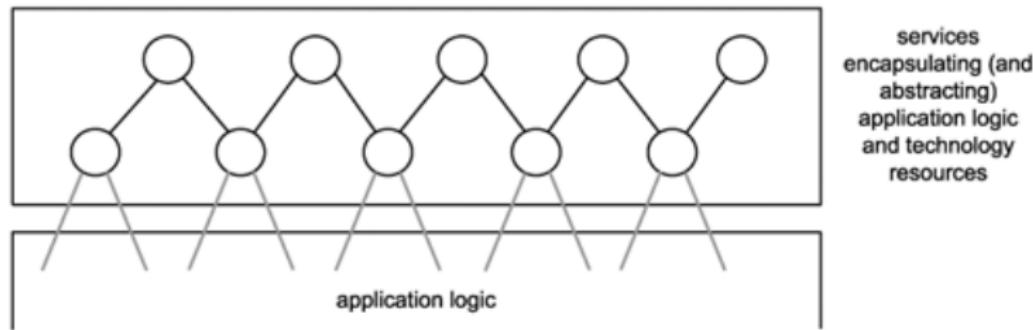


Figura: Abstracción de una lógica de negocio representada con tecnología privativa

Encapsulación de la “lógica de negocio” III

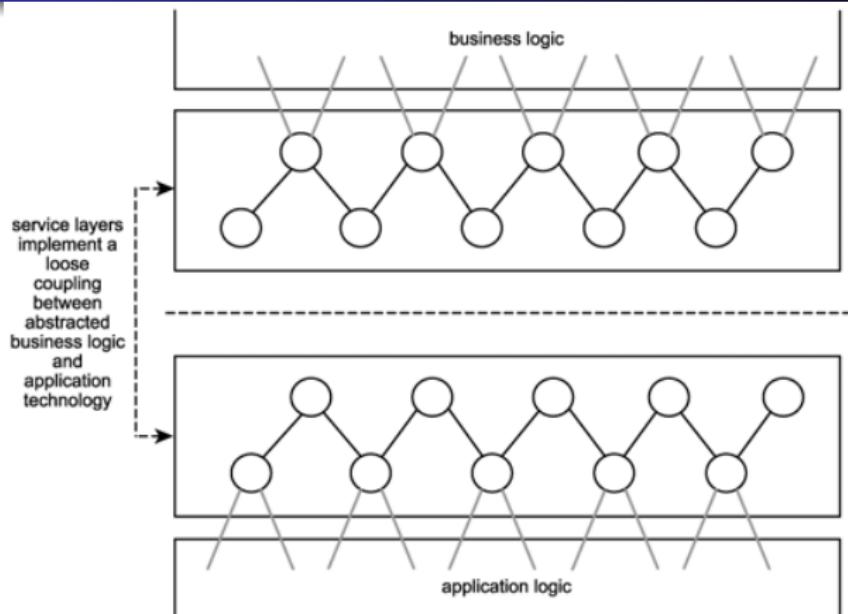


Figura: Abstracción de lógica de negocio en capas para implantar una SOE

Implementación de *agilidad tecnológica* en una organización

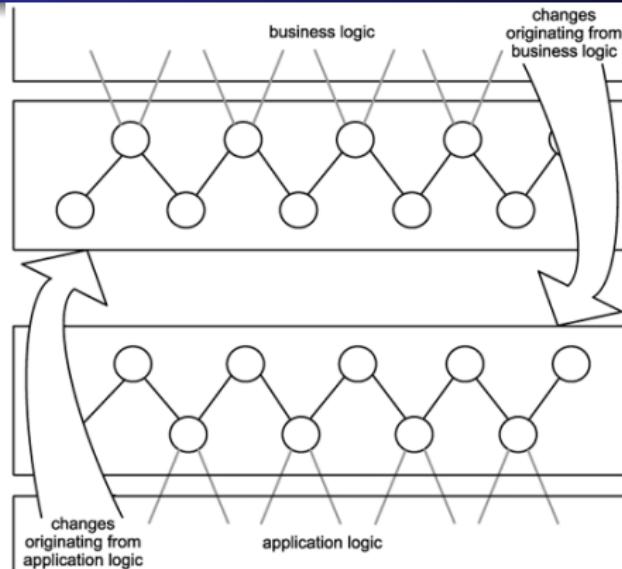


Figura: Facilidad de cambios mediante acoplamiento débil entre

Extensión de la definición actual de SOA

Un SOA contemporáneo además

“Puede establecer una abstracción de la lógica de negocio y de la tecnología lo que le permite introducir cambios en el modelado de un proceso de negocio y en su arquitectura como sistema de información, lo que tiene como consecuencia un acoplamiento débil entre estos modelos. Estos cambios fomentan la orientación a servicios que sirve como apoyo al nuevo concepto de *empresa orientada a servicios (SOE)*”.

Ventajas de un SOA contemporáneo

- Facilitar cambios en los negocios y en las empresas para responder a la variabilidad de las condiciones del mercado
- Reutilización de *macroservicios* más que *reusabilidad de micronivel* (Objetos, clases, etc.)
- Integración de sistemas software legados
- Modelado de todo el negocio, no del sistema de información de una empresa
- Evolución arquitectónica de los sistemas de información hacia arquitecturas más complejas y eficientes
- Propicia la interoperabilidad de plataformas y la separación del negocio de las implementaciones de los servicios

Inconvenientes

- Contribuye a la confusión que existe entre el paradigma arquitectónico de orientación a servicios y los denominados *servicios Web*
- ¿Un SOA es sólo el resultado de añadir varias capas XML a las aplicaciones y componentes software de las empresas?
- Sobrecarga debido al uso excesivo de RPCs que hacen algunas implementaciones actuales de SOA en el mercado

Inconvenientes II

- Hay tecnologías actualmente que no dependen de RPCs y su traducción a través de XML a un SOA: Java Business Integration (JBI), Windows Communication Foundation (WCF) o Data Distribution Service (DDS) y tecnologías emergentes de exploración de fuentes XML (VTD-XML)
- Si los servicios han de mantener un estado de la aplicación, se incurre en sobrecarga
- Incremento de acoplamiento no deseable entre proveedor y consumidor de servicios
- El utilizar un SOA suele impedir la realización de *modificaciones inmediatas* de un sistema software

Introducción a ADE

Concepto de “evento”

Cambio de estado en un sistema que provoca posteriormente la transmisión de un mensaje o notificación del cambio. No se ha de confundir con la *notificación del evento* (lo que se transmite, detecta o consume).

Event Driven Architecture (EDA)

Paradigma arquitectónico que se aplica en el diseño e implementación de sistemas y aplicaciones que transmiten eventos entre componentes software.

Introducción a ADE II

Motivación de los EDA

- Complementan a los SOA porque la notificación de eventos pueden ser iniciada, gestionada o los mensajes reenviados por *servicios* y también pueden activarlos.
- Favorecen la adaptación a comportamientos impredecibles del entorno y/o asíncronicidad de estímulos recibidos.
- SOA 2.0 define un nuevo *patrón de eventos* basado en las relaciones entre SOA y EDA.

Introducción a ADE III

Componentes de los EDA

- Generadores de eventos y transformación de formato
- Canal de eventos
 - mecanismo de transferencia entre *generadores* y *sumideros*
 - No confundir con el soporte del canal: conexión TCP, etc.
 - Asincronicidad y concurrencia en el acceso
- Motor de procesamiento de eventos y *reglas de negocio*.
- Actividades derivadas.

Composición de eventos

Elementos de un sistema diseñado como EDA

- Emisores de eventos o *agentes*
- Consumidores de eventos o *sumideros*:
 - Filtros
 - Transformadores
 - Actores

Composición de las notificaciones

- *Cabecera*: tipo–evento, identificador, tiempo ocurr., ...
- *Cuerpo*: ¿qué ocurrió exactamente?
- No confundir el *cuerpo* de un evento con la *lógica* que se activa como consecuencia de la ocurrencia del evento.

Estilos

Proceso de eventos

- Proceso Simple
- Proceso de flujo
- Proceso complejo: correlación entre eventos asíncronos

Distribución de eventos

- *Acoplamiento débil* de los EDA
- Desconexión entre evento y las consecuencias
- Ligadura en tiempo, espacio, de sincronización casi nula
- Se propicia la escalabilidad de las arquitecturas
- Heterogeneidad semántica y desarrollo confiable

Ejemplo: API–Java Swing

Swing

Proporciona funcionalidad y componentes software relacionados con el desarrollo de interfaces gráficas de usuario (GUI).

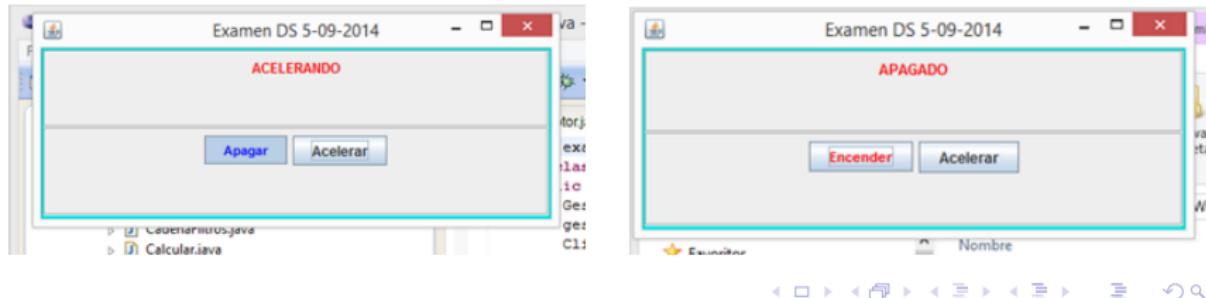
Swing y EDA

Utiliza nombres de componentes (`ActionListener`, `ActionEvent`) que están muy relacionados con el procesamiento de eventos. Los conceptos de la API de Swing se basan en EDA.

Swing y EDA II

Elementos de Swing

- Las clases de *generadores* actúan como ActionListener
- Las clases *recogedores* de eventos actúan como ActionPerformed



Ejemplo de ADE: SCACV con Swing

Programación de *escuchadores* de eventos para ambos botones como clases de Swing

```
BotonEncender.setText("Encender");
BotonEncender.addActionListener(new java.awt.event.
    ActionListener(){
    public void actionPerformed(java.awt.event.
        ActionEvent evt){
        BotonEncenderActionPerformed(evt);}}));
subpanel.add(BotonEncender);
BotonEncender.setBounds(10,10,90,23);
BotonAcelerar.setText("Acelerar");
BotonAcelerar.addActionListener(new java.awt.event.
    ActionListener(){
    public void actionPerformed(java.awt.event.
```



Ejemplo de ADE: SCACV con Swing II

Programación de *recogedores* de eventos para ambos botones como clases de Swing

```
synchronized private void BotonEncenderActionPerformed(  
    java.awt.event.ActionEvent evt){  
    if (BotonEncender.isSelected()) {  
        BotonEncender.setText("Apagar");  
        BotonEncender.setForeground(Color.blue);  
        EtiqMostrarEstado.setForeground(Color.red);  
        EtiqMostrarEstado.setText("APAGADO");  
    }  
    else{  
        BotonEncender.setText("Encender");  
        BotonEncender.setForeground(Color.red);  
        EtiqMostrarEstado.setText("APAGADO");  
    }  
}
```



Conceptos generales sobre GC

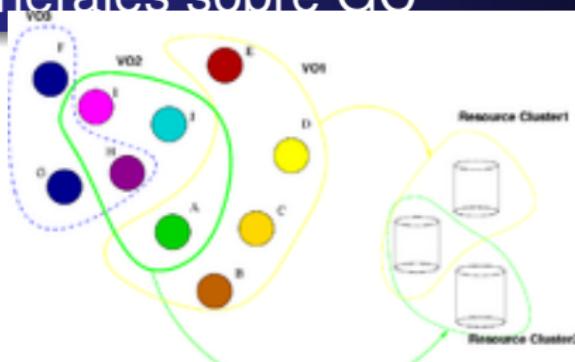


Figura: Clusters y organizaciones virtuales (VO) utilizando CaaS.

“Grid Computing”

Es un término que se utiliza para expresar la combinación de varios computadores desde plataformas separadas que colaboran para resolver una única tarea y, posteriormente, desaparecen.

Motivación de GC

Fundamentos

- GC está basada en conceptos y técnicas de Programación Distribuida, Programación Orientada a Objetos y Servicios Web para conseguir gran capacidad de cálculo.
- GC hace posible resolución de problemas que constituyen grandes retos para la Programación Paralela y Distribuida (plegamiento de proteínas, modelado financiero, predicción meteorológica, ...)
- Promueve la *computación desinteresada* y la *búsqueda de ciclos perdidos*

Computación con “grids”

Características fundamentales

- A diferencia de “Cloud Computing”(CC), con GC donamos desinteresadamente nuestros recursos de computación
- Virtualización de recursos *a demanda*
- Los recursos no se administran centralizadamente
- indicada para aplicaciones que presenten acoplamiento débil entre cálculos paralelos
- Uso de estándares abiertos
- “Calidad de servicio” (QoS) se consigue transitoriamente
- Alta escalabilidad
- Difícil garantizar la seguridad y confiabilidad



Computación con “grids” II

Ventajas de GC

- Facilidad de programación vs. supercomputadores
- Desarrollo y depuración de programas en 1 sola máquina
- Computación como un Servicio (CaaS), que se consigue gratis
- Recuperación de ciclos y uso extensivo de recursos

Computación con “grids” III

Desventajas de GC

- Falta de fiabilidad
- Falta de predecibilidad
- Compromiso de la seguridad de la plataforma local
- Problemas con la interoperabilidad de aplicaciones en plataformas heterogéneas
- Costes elevados en software para afrontar cambios de plataforma

Estado del arte de GC

Tipos de proyectos de GC

- Entornos de GC que incluyen todos los servicios y herramientas: Globus Toolkit:
<http://toolkit.globus.org/toolkit/>
- Entornos para *Computación Desinteresada*: Berkeley Open Infrastructure for Network Computing (BOINC):
<https://boinc.berkeley.edu/>
- La tecnología GC está siendo muy utilizada para grandes proyectos de investigación internacionales por varias agencias: NASA, EU Comision, NSF, EEUU Cancer Research Project, WCG, ...

Estado del arte de GC II

Supercomputadores más rápidos actualmente

Cuadro: Supercomputadores más rápidos en la actualidad

Nombre	Fecha referencia	Potencia en TFLOPS
Bitcoin Network	2014	$1,16 \times 10^8$
Folding@home	2013	580 (<i>nativos</i> \equiv 1140 x86)
BOINC	2013	920
MilkyWay@Home	2010	160
SETI@Home	2010	730
Einstein@Home	2010	210
GIMPS	2011	61



Estado del arte de GC II

Principales infraestructuras internacionales

- European Grid Infrastructure (EGI):
<http://www.egi.eu>
- INFN Production Grid <http://www.italiangrid.it>
- Facilidad NAS (Nasa Advanced Supercomputing)
- Open Grid Forum: <http://www.ogf.org>: documentos con siglas OGSA, OGSI y JSDL
- Proyecto XGrid de Apple: <http://www.apple.com/server/macosx/technology/xgrid.html>
(preinstalado en Macs con OS > 10.4)

Bibliografía Fundamental

-  Bass, L., Clements, P., and Kazman, R. (2012).
Software Architecture In Practice.
Addison-Wesley, Boston, Massachussets, third edition.
-  Booch, G. (2008).
Handbook of Software Architecture.
<http://www.booch.com/systems.jsp>.
-  Buschmann, F. and et al. (2004).
Pattern-oriented software architecture, volume I.
Wiley, Chichester, England.
-  Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010).