Arquitectura de Software

M.I. Capel

ETS Ingenierías Informática
y Telecommunicación
Universidad de Granada
Email: manuelcapel@ugr.es

Desarrollo de Software

Índice

Índice

Índice

Índice

Índice

Índice

Arquitectura de Software

- Es una descripción de la estructura del software necesaria para razonar sobre el sistema:
  - elementos del software,
  - relaciones entre elementos,
  - las propiedades de ambos

[Bass et al., 2012]

La Arquitectura de Software de sistema de computación es un conjunto de estructuras de software, que comprenden: unidades de implementación, aspectos dinámicos y la correspondencia con entornos de ejecución, instalación, desarrollo y organización del sistema.

Motivación de las AS

[Bass et al., 2012]

Las arquitecturas software son un artefacto esencial para realizar el diseño de un sistema informático.

1. Facilita la comunicación entre las partes interesadas
2. Documenta *decisiones de diseño tempranas* sobre el sistema final en funcionamiento
3. Modelo conceptualmente *manejable* sobre estructuración y colaboración entre componentes del sistema
4. Las decisiones de diseño aquí son muy importantes para conseguir requisitos críticos del sistema

Importancia de las arquitecturas de software

1. Propiciar/inhibir atributos de calidad
2. Predicción temprana de los atributos de calidad mediante análisis de la arquitectura
3. Facilitar el razonamiento sobre el sistema y sus cambios cuando evolucione
4. Documentación que propicia la comunicación entre las partes
5. Soporte idóneo para decisiones de diseño tempranas (difíciles de cambiar posteriormente)
6. Conjunto de restricciones de implementación del software
7. Dictar la estructura de una organización o viceversa
8. Fundamento del prototipado rápido
9. Facilita razonar sobre el coste y la planificación del proyecto
10. Modelo reutilizable y transferible conforme con el núcleo de una línea de producto
11. Se centra en el ensamblaje de componentes más que en su creación individualizada
12. Canaliza la creatividad de los desarrolladores, reduciendo la complejidad del diseño
13. Base para entrenar a nuevos miembros del equipo de desarrollo

Estructuras arquitectónicas

Idea fundamental

Un estructura de un sistema es arquitectónica si sirve de apoyo al desarrollo de razonamiento acerca del sistema y sus propiedades

Caracterización:

- relación con los *atributos de calidad* del sistema
- no está limitado en cuanto a número
- lo *arquitectónico* ha de ser *ùtil* en el contexto de la ejecución

Todo sistema software ha de poseer una arquitectura

Categorías de estructuras arquitectónicas

Idea fundamental

Un sola estructura no puede reclamar ser la *única arquitectura software* del sistema objetivo

Clases de estructuras importantes en el análisis, diseño y documentación de una arquitectura

- Estructuras que dividen al sistema en unidades de implementación: estructura estática *módulo*
- Estructuras dinámicas: {servicios, infraestructura, relaciones sincronización e interacción}
- Estructuras para la correspondencia con entornos de ejecución, instalación, desarrollo y organización

Módulo vs. componente software

Elementos de una arquitectura software

- Se incluye el *comportamiento* de cada elemento
- Necesidad de documentación si afecta a otros elementos o tiene impacto en la aceptación del sistema completo
- Concepto de *Vista*

Vista

representación de un conjunto coherente de elementos y las relaciones que existen entre ellos

Estructura

conjunto de elementos del sistema, tanto si existen en el software como en el hardware

Diseño arquitectónico e identificación de las estructuras

Decisiones a tomar para obtener un diseño arquitectónico:

1. *estructuras modulares*,
2. *estructuras componente-conector (C&C)*
3. *estructuras de despliegue* o *asignación*

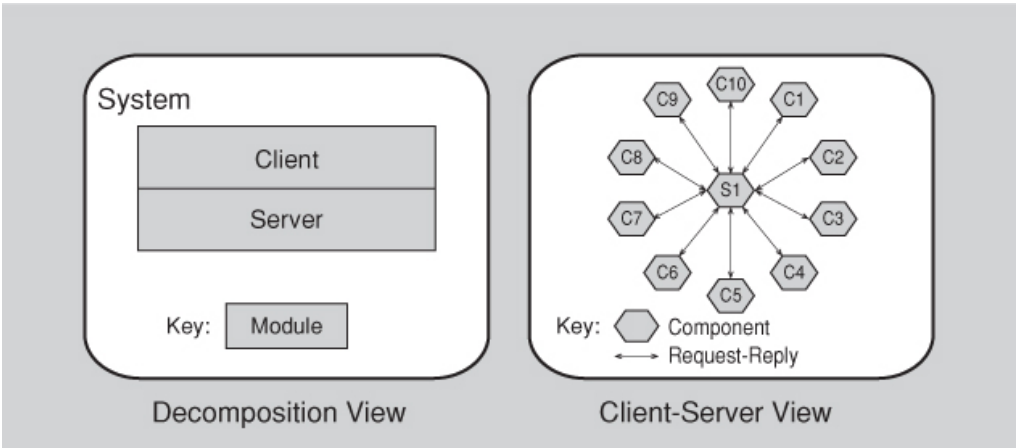
Tabla de estructuras arquitectónicas

	Software Structure	Element Types	Relations	Useful For	Quality Attributes Affected
Module Structures	Decomposition	Module	Is a submodule of	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control	Modifiability
	Uses	Module	Uses (i.e., requires the correct presence of)	Engineering subsets, engineering extensions	"Subsetability," extensibility
	Layers	Layer	Requires the correct presence of, uses the services of, provides abstraction to	Incremental development, implementing systems on top of "virtual machines"	Portability
	Class	Class, object	Is an instance of, shares access methods of	In object-oriented design systems, factoring out commonality; planning extensions of functionality	Modifiability, extensibility
	Data model	Data entity	{one, many}-to-{one, many}, generalizes, specializes	Engineering global data structures for consistency and performance	Modifiability, performance
C&C Structures	Service	Service, ESB, registry, others	Runs concurrently with, may run concurrently with, excludes, precedes, etc.	Scheduling analysis, performance analysis	Interoperability, modifiability
	Concurrency	Processes, threads	Can run in parallel	Identifying locations where resource contention exists, or where threads may fork, join, be created, or be killed	Performance, availability
Allocation Structures	Deployment	Components, hardware elements	Allocated to, migrates to	Performance, availability, security analysis	Performance, security, availability
	Implementation	Modules, file structure	Stored in	Configuration control, integration, test activities	Development efficiency
	Work assignment	Modules, organizational units	Assigned to	Project management, best use of expertise and available resources, management of commonality	Development efficiency

Relación entre estructuras arquitectónicas

Idea fundamental

- Perspectiva diferente, útil para contribuir a comprender el diseño
- Interdependencia de vistas
- Correspondencia 1-a-muchos entre vistas y estructuras



Recomendaciones estructurales

Fundamentadas en conseguir una buena descomposición:

- Modulos bien definidos con responsabilidades asignadas según los principios de la Ingeniería de Software
- Los atributos de calidad se han de poder alcanzar utilizando patrones
- Independencia de productos o herramientas-software; han de poder cambiarse de versión con facilidad
- Separación de los módulos que producen datos de los que los consumen
- No correspondencia 1-a-1 entre módulos y componentes
- Los procesos han de poder reasignarse a procesadores
- Número pequeño de formas de interacción entre componentes
- Número pequeño de áreas de contención

Concepto de Diseño Arquitectónico

[Eden and Kazman, 2003]

Un diseño es una instancia de una arquitectura software tal como un objeto es una instancia de una clase. Por ejemplo, consideremos la arquitectura cliente-servidor, podremos diseñar un sistema software en el cual el concepto de red sea lo más importante de muchas formas distintas utilizando esta arquitectura, tal vez utilizando la plataforma Java actual (JEE) o la de Microsoft (.NET). Así, existe una sola arquitectura pero muchos diseños diferentes que se pueden crear a partir de ella. Por consiguiente, no podemos mezclar los conceptos “arquitectura” y “diseño”, ya que son complementamente diferentes entre sí.

Impacto del diseño en los atributos de calidad del sistema objetivo

Ejemplos de atributos a propiciar:

- *Alto rendimiento*: Controlar: gestión del comportamiento temporal de elementos, uso de recursos compartidos, volumen y frecuencia de comunicación
- *Disponibilidad*: Controlar: relevo de componentes cuando ocurre un fallo en el sistema
- *Comprobabilidad*: Controlar: fácil "testing" de elementos individuales y el comportamiento emergente del grupo
- *Seguridad*:Controlar: envoltorio comportamental de los elementos y comportamiento emergente
- *Interoperabilidad*: Controlar: qué elementos son responsables de comunicaciones externas
- *Usabilidad* : Controlar: aislar los detalles de la interfaz de los componentes de su implementación interna y del resto del sistema

Arquitecturas de software y la actividad denominada *Desarrollo de Software*

Contextos de interés:

1. Ciclo de vida del proyecto:¿Cómo se relaciona una arquitectura con las otras fases de un ciclo de vida de desarrollo de software?
2. Técnico:¿Qué papel juega la arquitectura en el sistema software o en los sistemas de los que form parte?
3. Negocio:¿Cómo afectará la existencia de una arquitectura al entorno de negocio de una organización?
4. Profesional:¿Cuál es el papel de una arquitectura de software dentro de la organización o de un proyecto?

Arquitectura de software dentro del contexto del ciclo de vida del software

Tipos actuales de proceso de desarrollo de sistemas software:

1. *En cascada*: defectos ocultos descubiertos en la fase de comprobación ocasionarían la reimplementación y/o rediseño del software
2. *Iterativo*: desarrollo mediante ciclos cortos (iteraciones), cada uno implica captura de requisitos, diseño parcial del sistema, implmentación y prueba del prototipo (Rational Unified Process)
3. *Ágil*: metodologías incrementales e iterativas de desarrollo (Scrum, XP, Crystal Clear) para entregar un prototipo funcionante del sistema lo antes posible y con frecuencia; se apoyan en el uso de los clientes para depurar y mejorar el software
4. *Desarrollo Dirigido por Modelos*

Arquitectura de software dentro de un contexto técnico

Aspectos técnicos impactados:

1. Inhibir/propiciar que el sistema alcance los atributos de calidad
2. Predecir muchos de los aspectos de calidad del sistema objetivo
3. Facilitar el razonamiento sobre los atributos y sobre las consecuencias de hacer cambios

Figura: El barco de guerra Vasa: ejemplo de mala arquitectura de un sistema que conduce a un resultado catastrófico

Atributos de calidad

Idea fundamental

Ninguna otra cosa tiene mayor importancia en la definición de una arquitectura que conseguir los atributos de calidad a satisfacer por el sistema software objetivo

atributo a propiciar	**aspectos a cuidar** durante el diseño
alto rendimiento	manejo temporal de elementos arquitectónicos
disponibilidad	relevo de componentes entre sí cuando ocurren fallos
comprobabilidad	componentes individuales y comportamiento grupal emergente
seguridad	envoltorio comportamental de los elementos
interoperabilidad	elementos responsables de las comunicaciones externas
usabilidad	aislar detalles interfaz de componentes del resto del sistema

Ejemplo: el navío de guerra Vasa

Motivación del fallo catastrófico del sistema final (The Vasa Museum, Estocolmo, Suecia)

- Se accedió a construir un sistema cuyos requisitos eran contradictorios e imposibles de satisfacer
- La arquitectura del sistema poseía un fallo que convertía en inviable al sistema resultante
- Muy mala gestión de los riesgos del proyecto y de las coacciones del cliente

Lo que el ingeniero naval Hybertsson debió hacer

- Utilizar prácticas de arquitectura de sistemas que, bajo requisitos muy exigentes, hayan probado su correcto funcionamiento
- Evaluar la arquitectura del sistema antes de comenzar la construcción del navío (mitigando los riesgos de sacar un diseño que no tenía ningún precedente)
- Seguir técnicas de desarrollo incremental basadas en la arquitectura del sistema para identificar defectos de diseño antes de que fuera muy tarde para corregirlos

Recomendaciones del proceso de desarrollo

Fundamentadas en conseguir una buena arquitectura del sistema objetivo:

- La integridad conceptual de una arquitectura se consigue si es la obra de 1 solo arquitecto de software
- Se ha de fundamentar en 1 lista priorizada de requisitos (importan más los atributos de calidad que la funcionalidad)
- Documentación utilizando *vistas* (1 vista/conjunto de intereses de una parte)
- Evaluación para determinar si ofrece los atributos de calidad requeridos
- Las arquitecturas han de facilitar una implementación incremental

Patrones

Definición:

composición de elementos arquitectónicos útil para resolver problemas específicos en un contexto, aplicables a diferentes tipos de casos, bien documentados y difundidos

Describen los tipos de elementos y sus interacciones utilizados para resolver un problema

Caracterización por los elementos arquitectónicos que utilizan:

- Patrón con capas
- Patrón Cliente-Servidor
- Patrón en múltiples niveles
- Patrón repositorio
- Centro de *competencia y plataforma*

Patrón Arquitectónico

Clasificación de patrones (2008) [Booch, 2008]

Establece una clasificación atendiendo a características para cumplir con los requisitos de *seguridad*, *rendimiento*, *despliegue* y *almacenamiento*

1. *Control de accesos*: el acceso a determinadas partes de la arquitectura software ha de ser controlado rigurosamente
2. *Concurrencia*: diferentes formas de permitir que los componentes de la aplicación sean concurrentes
3. *Distribución*: la comunicación entre entidades software es muy diversa y afecta al diseño, la ubicación de los componentes ha de ser optimizable (*configurabilidad*)
4. *Persistencia*: la *supervivencia* de los objetos (atributos, estado) entre distintas ejecuciones es algo diseñable Una mala solución puede dañar la eficiencia gravemente

Patrones arquitectónicos– 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.

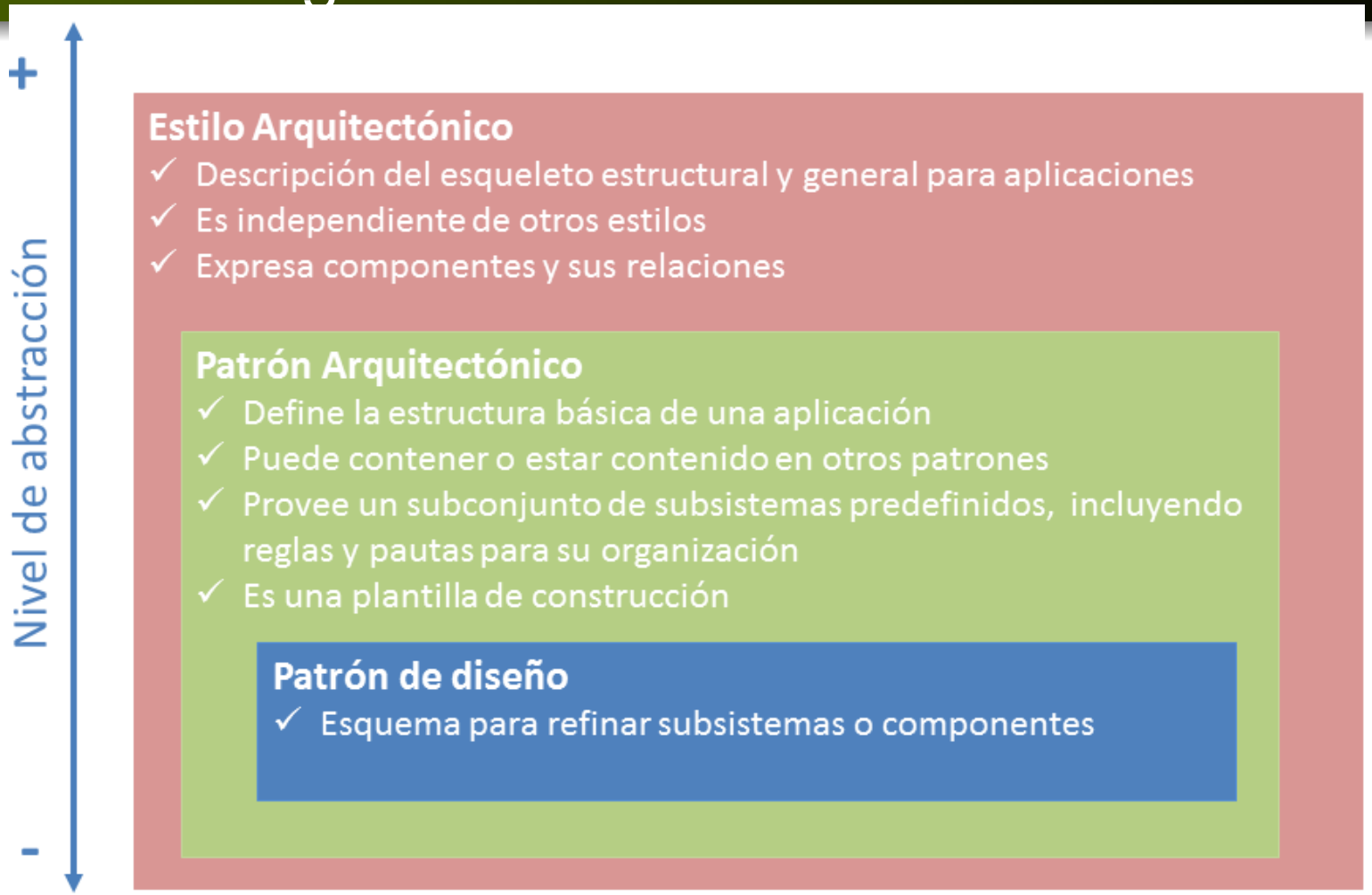
El uso de ciertos mecanismos, como los estilos y patrones arquitectónicos, permite mejorar las características de calidad en el software [Jansen and Bosch, 1999], bien sean éstas observables o no en tiempo de ejecución [Bass et al., 2012].

Patrones arquitectónicos– III

Características generales de los patrones arquitectónicos

- El ámbito del patrón es menos general que el del estilo arquitectónico
- Un patrón impone una regla a la arquitectura de una aplicación, por ejemplo, describiendo cómo gestionar un aspecto de su funcionalidad (p.e.: la concurrencia).
- Los patrones arquitectónicos tienden a abordar problemas de *comportamiento* del software 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



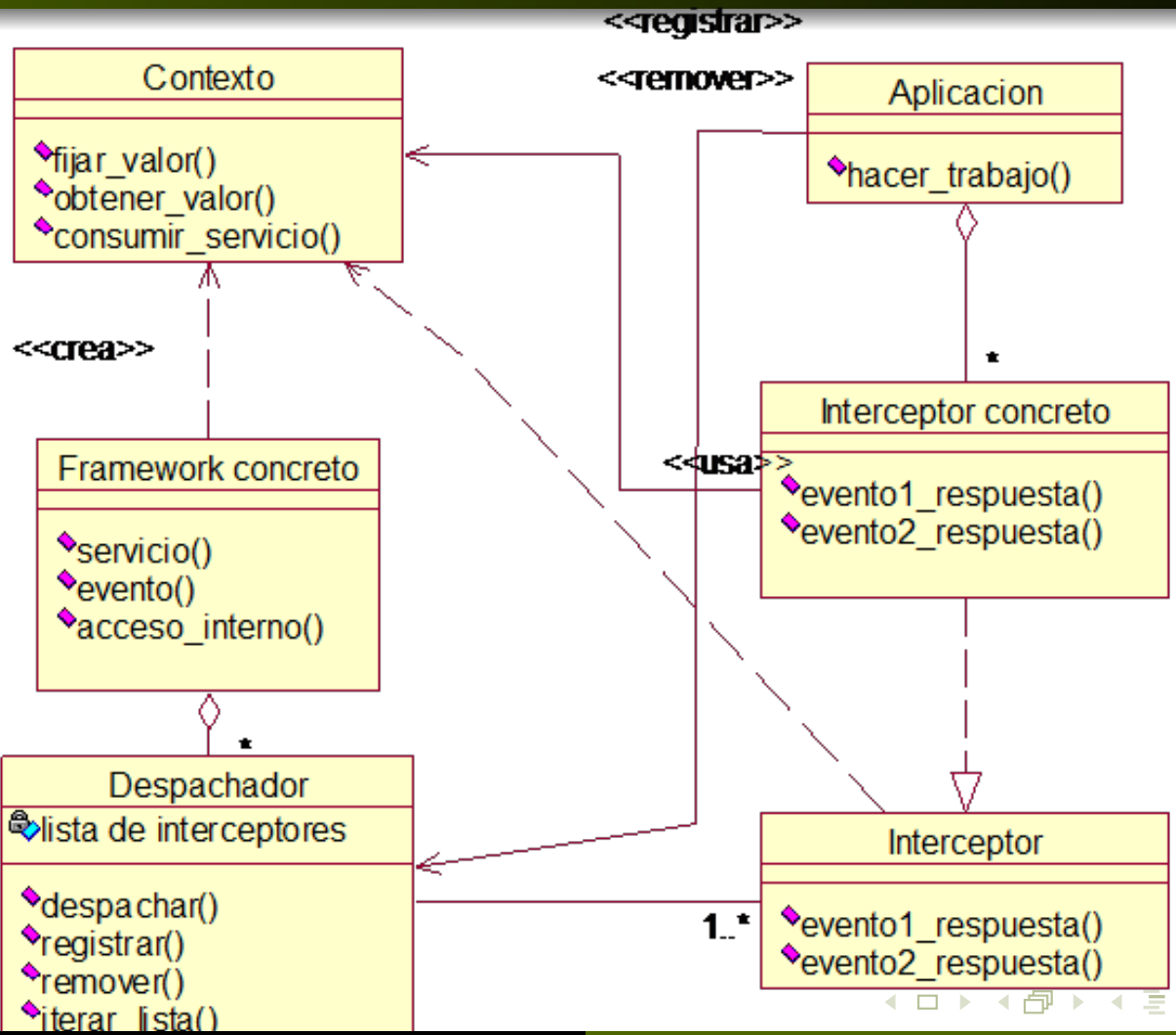
Patrón Interceptor

Descripción general:

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

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, Flex.,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 Fac.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
Degradación 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

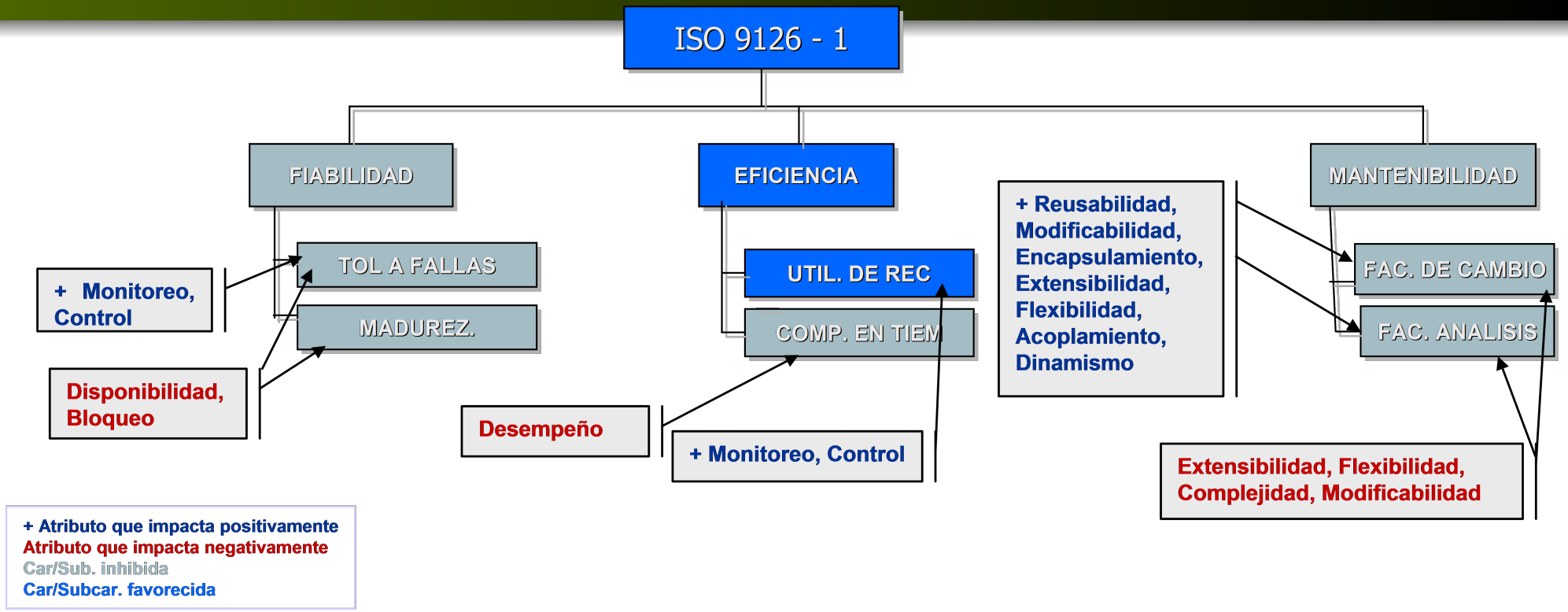


Figura: Características ISO del patrón “Interceptor"

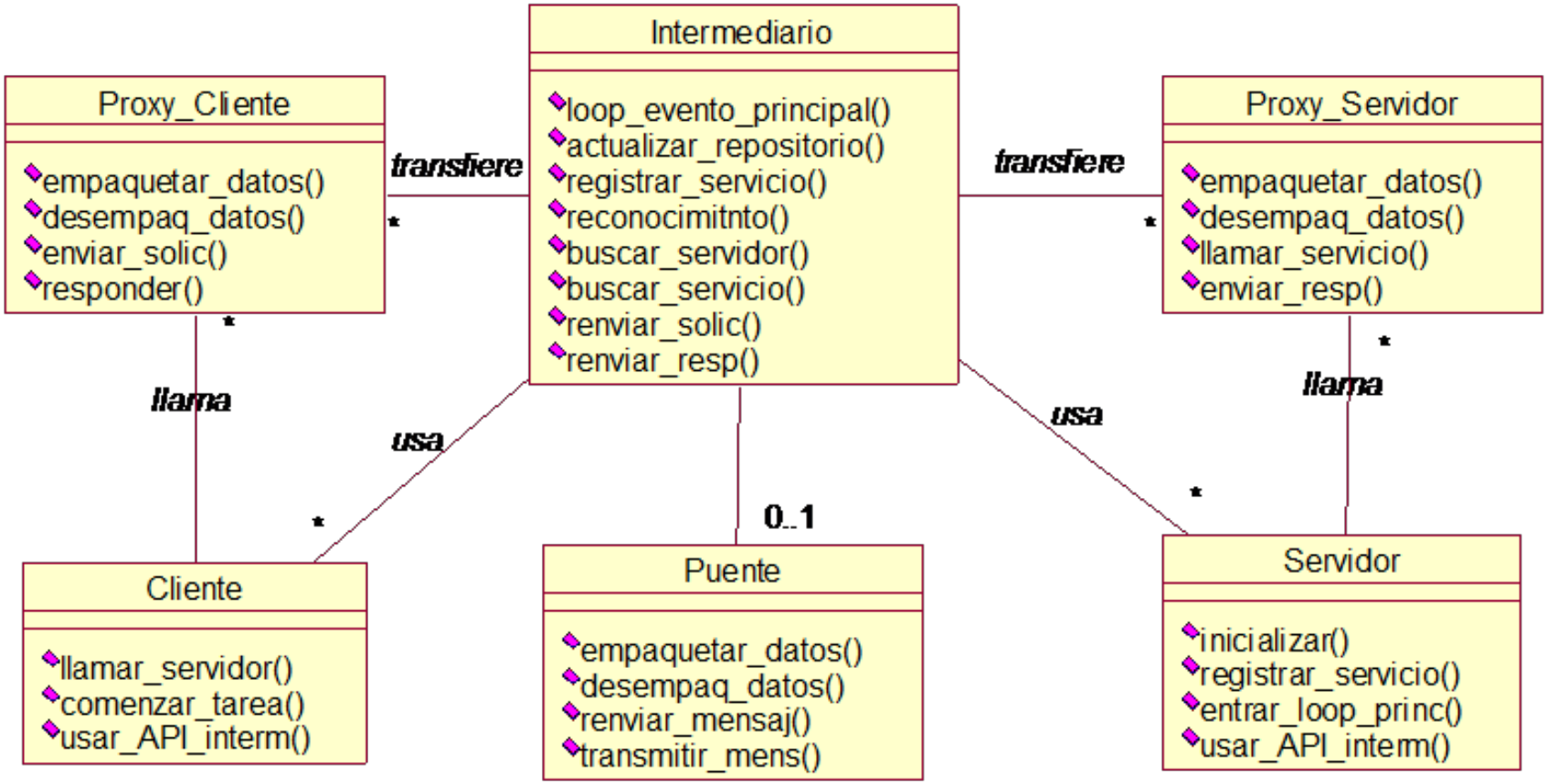
Patrón Broker

Descripción general:

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


Intermediario
loop_evento_principal()
actualizar_repositorio()
registrar_servicio()
reconocimitnto()
buscar_servidor()
buscar_servicio()
renviar_solic()
renviar_resp()
Proxy_Cliente
empaquetar_datos()
desempaq_datos()
enviar_solic()
responder()
Proxy_Servidor
empaquetar_datos()
desempaq_datos()
llamar_servicio()
enviar_resp()
transfiere
transfiere
llama
usa
usa
llama
0..1
Cliente
llamar_servidor()
comenzar_tarea()
usar_API_interm()
Puente
empaquetar_datos()
desempaq_datos()
renviar_mensaj()
transmitir_mens()
Servidor
inicializar()
registrar_servicio()
entrar_loop_princ()
usar_API_interm()

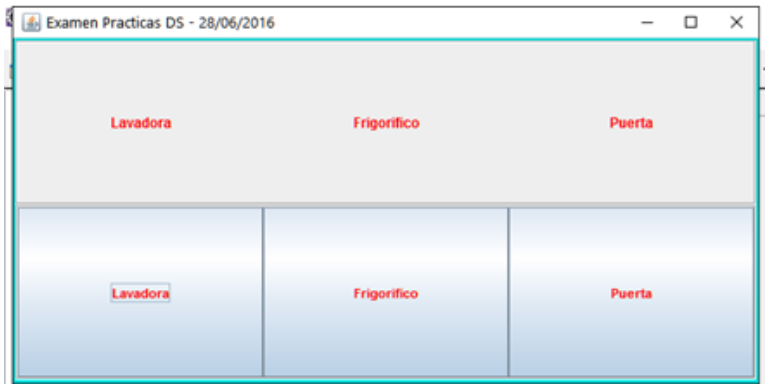
Estructura de clases del patrón Broker

- *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*

Ejemplo de uso del patrón Broker

Ejemplo demostrativo con un sistema E-Home que implementa una red de dispositivos domésticos colaborativos.
El `Broker` se encargará de facilitar la compatibilidad entre el software y los datos del cliente con los controladores de los distintos dispositivos incluidos en esta red.
Los servicios que están situados en un control central estarán conectados a través de una red con el software que se ejecuta en el controlador de los dispositivos.

Ha de ser posible acceder a los dispositivos con facilidad y los servicios no tienen por qué ejecutarse siempre en el mismo controlador, lo cual se consigue a través de `ControladorCentral`.



Ejemplo de uso del patrón Broker-II

Estructura de clases

- `Cliente`: envía una petición de activación de un dispositivo concreto a `Broker`
- `Broker`: accede a `ControlCentral` de la red y envía al servidor los mensajes que ha recibido de `Cliente`
- `ControlCentral`: recibe mensajes de `Broker` y contacta con los distintos dispositivos para configurarlos y activarlos
- `ControladorDispositivo`: realiza acciones específicas sobre el dispositivo del que es responsable

Ejemplo de uso del patrón Broker-III

ControladorDispositivo, además de su constructor, sólo define un método tal como el siguiente:

```
synchronized void realizarAccion(String m){
  System.out.println(m+ " realizara accion");
}
```

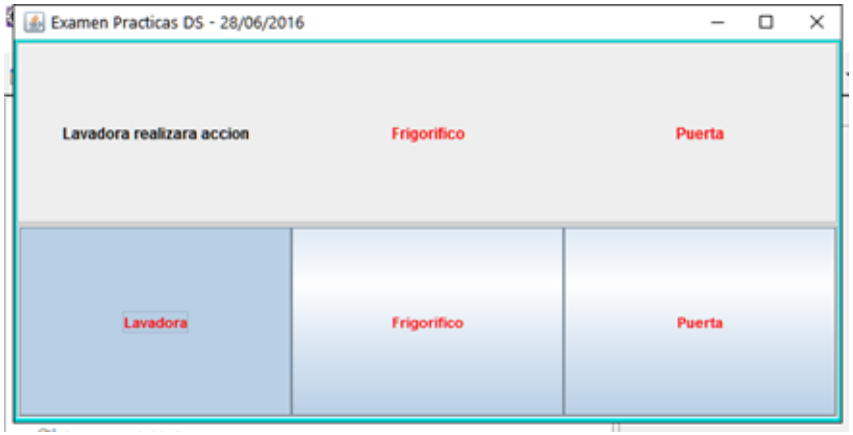


Figura: Aspecto de la interfaz después de pulsar el botón *Lavadora*

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 objetos	Interoperabilidad	Portabilidad Adaptabilidad

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

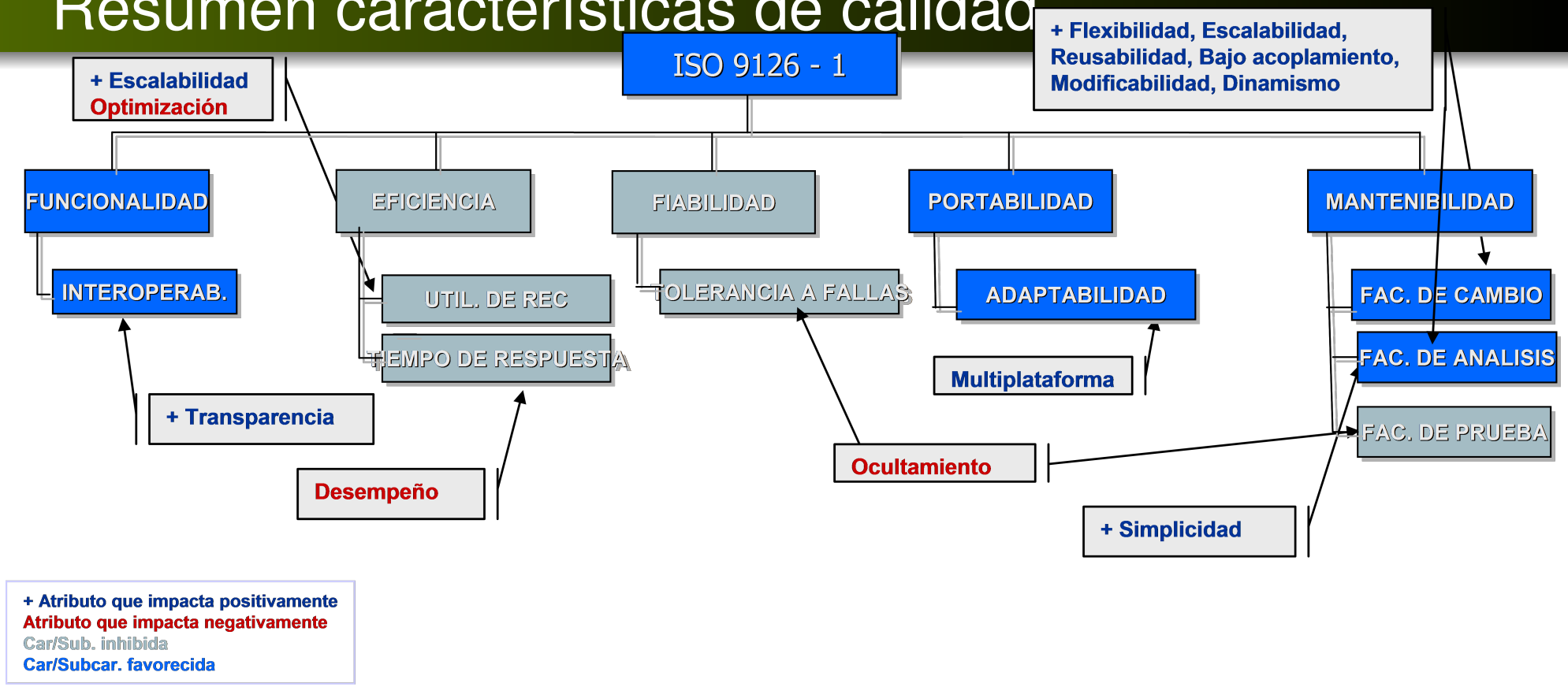


Figura: Características ISO del patrón “Broker"

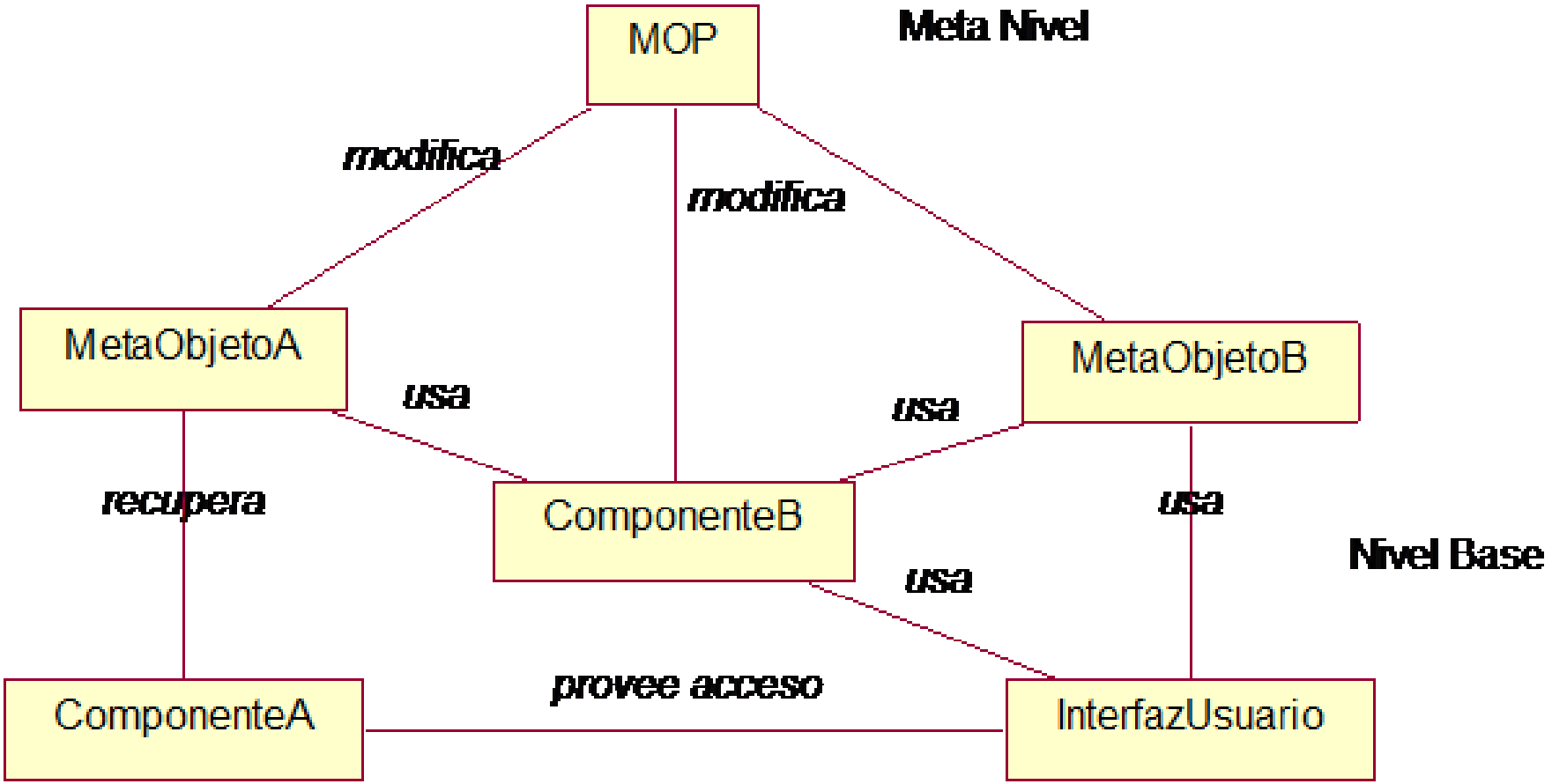
Patrón Reflection

Descripción general

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 métodos

- Contexto: Cualquier sistema que necesite soporte para realizar cambios propios y 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


MOP
Meta Nivel
modifica
modifica
MetaObjetoA
MetaObjetoB
usa
usa
recupera
ComponenteB
usa
usa
Nivel Base
provee acceso
ComponenteA
InterfazUsuario

Estructura de clases del patrón Reflection

- *Meta Nivel*: autoconsciencia de la estructura y funcionamiento del software
- La implementación del *Meta Nivel* utiliza *Meta Objetos*
- *Meta Objetos*: encapsulan y representan información acerca del software
- *Nivel Base*: objetivo y relación con el metanivel
  - Los cambios realizados en el *Meta Nivel* afectan consecuentemente al comportamiento del *Nivel Base*
- Cambios en los metaobjetos y su efecto en los componentes y código del nivel base

Ejemplo de Reflection: *juego de la metamorfosis*

Ejemplo demostrativo del patrón *Reflection* que genera clases y métodos utilizando este patrón de diseño y las facilidades reflectivas de Java.
También incluye una interfaz de 4 botones programada con Swing/Java.

Estructura de la aplicación:

- Clase `Bicho`: métodos para fijar el tipo de bicho, obtener su descripción (aparecerá como una etiqueta en el panel inferior) y métodos para que un bicho se *"mueva"* (si puede hacerlo).

Ejemplo de Reflection: *juego de la metamorfosis*–II

La metamorfosis de un bicho puede estar en tres fases distintas: *Gusano ("worm")*, *Capullo ("cocoon")* y *Mariposa ("butterfly")*.

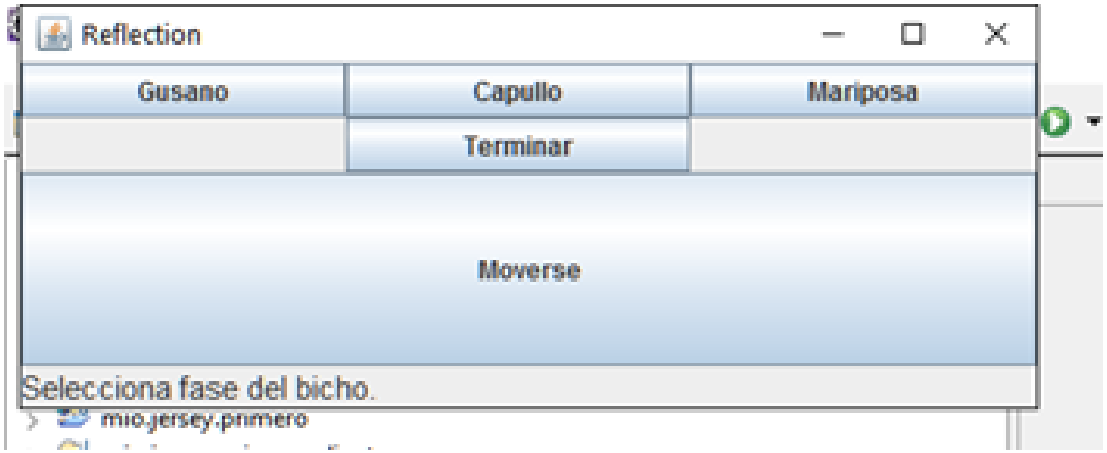


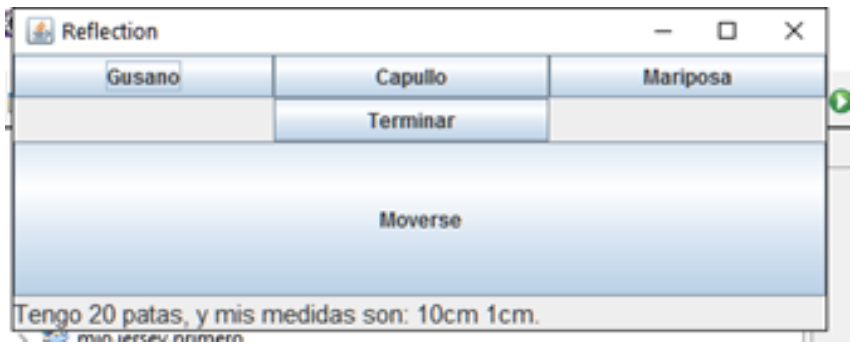
Figura: Aspecto de la interfaz al comenzar la aplicación

Ejemplo de uso del patrón Reflection: *juego de la metamorfosis*–III

```
public class Bicho {
  public String tipo;
  public void setTipo(String t){
    tipo= t;
    }
  public String getTipo(){
    return tipo;
    }
  public String descripcionGusano(int patas, int longitud, int ancho){
        return "Tengo: "+patas+" patas, y mis medidas son: "+longitud+"cm, "+ancho+"
              cm.";
        }
  public String descripcionCapullo(int radio){
        return "Mido: "+radio+"cm de radio.";
        }
  public String descripcionMariposa(int longitud, int ancho){
        return "Mido: "+longitud+"cm de longitud y "+ancho+"cm de ancho.";
        }
...
}
```

Ejemplo de Reflection: *juego de la vida*–IV

```
....
public String mover(int velocidad){
  if (tipo=="Gusano"){
        return "Voy caminando:"+velocidad+"Km/h.";}
   else if (tipo=="Capullo"){
        return "Soy un capullo, no me puedo mover.";}
      else{return "Voy volando:"+velocidad+"Km/h.";}
        }
}
```



Ejemplo de Reflection: *juego de la vida*–V

```
\\Clase Interfaz --parte Reflectiva
Class cls = Class.forName("com.ds_reflection.Bicho");
Object obj = cls.newInstance();
Method metodo;
```

Instalación de botones

```
public void actionPerformed(ActionEvent evt){//Utilizar para: new JButton("Gusano")
Class[] paramString = new Class[1];
Class[] paramGusano = new Class[3];
...
paramString[0]= String.class;
String tipo= "Gusano", salida1;
try {
   metodo = cls.getDeclaredMethod("setTipo", paramString);
   metodo.invoke(obj, tipo);
   metodo = cls.getDeclaredMethod("descripcionGusano", paramGusano);
   Integer[] pam = {20,10,1};
   salida1 = (String) metodo.invoke(obj, pam);
   textField.setText(salida1);
  }
  catch (NoSuchMethodException | SecurityException | IllegalAccessException|
       IllegalArgumentException | InvocationTargetException e) {
                                e.printStackTrace(); }
};//actionPerformed()
```

Ejemplo de Reflection: *juego de la vida*–VI

```
\\Clase Interfaz --parte Reflectiva
Class cls = Class.forName("com.ds_reflection.Bicho");
Object obj = cls.newInstance();
Method metodo;
```

Instalación del botón “Moverse”

```
public void actionPerformed(ActionEvent e){
Class[] paramVelocidad = new Class[1];
paramVelocidad[0] = Integer.TYPE;

...
try {
        metodo = cls.getDeclaredMethod("getTipo", null);
        String tipo  = (String)metodo.invoke(obj, null);
        metodo = cls.getDeclaredMethod("mover", paramVelocidad);
        int velocidad = 1;
        salida2 = (String) metodo.invoke(obj, velocidad);
        textField.setText(salida2);
        }
        catch (NoSuchMethodException | SecurityException | IllegalAccessException|
             IllegalArgumentException | InvocationTargetException e) {
                                e.printStackTrace(); }
};//actionPerformed()
```

Implicaciones en la calidad del patrón Reflection

Beneficios	**Atributos de calidad**	**Características ISO 9126**
Comprobación correctitud desde el nivel de *MetaObjetos*	Correctitud	Funcionalidad Precisión
Ejecución de los cambios desde el nivel de *MetaObjetos*	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–nivel 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 Facilidad de análisis	+	Modificabilidad Extensibilidad Dinamismo
	Facilidad de análisis	-	Complejidad
	Facilidad de prueba		
Funcionalidad	Precisión	+	Correctitud
Eficiencia	Uso de recursos	-	Desempeño Complejidad
Fiabilidad	Madurez	-	Propensión a fallas
	Tolerancia a fallas		

Resumen características de calidad

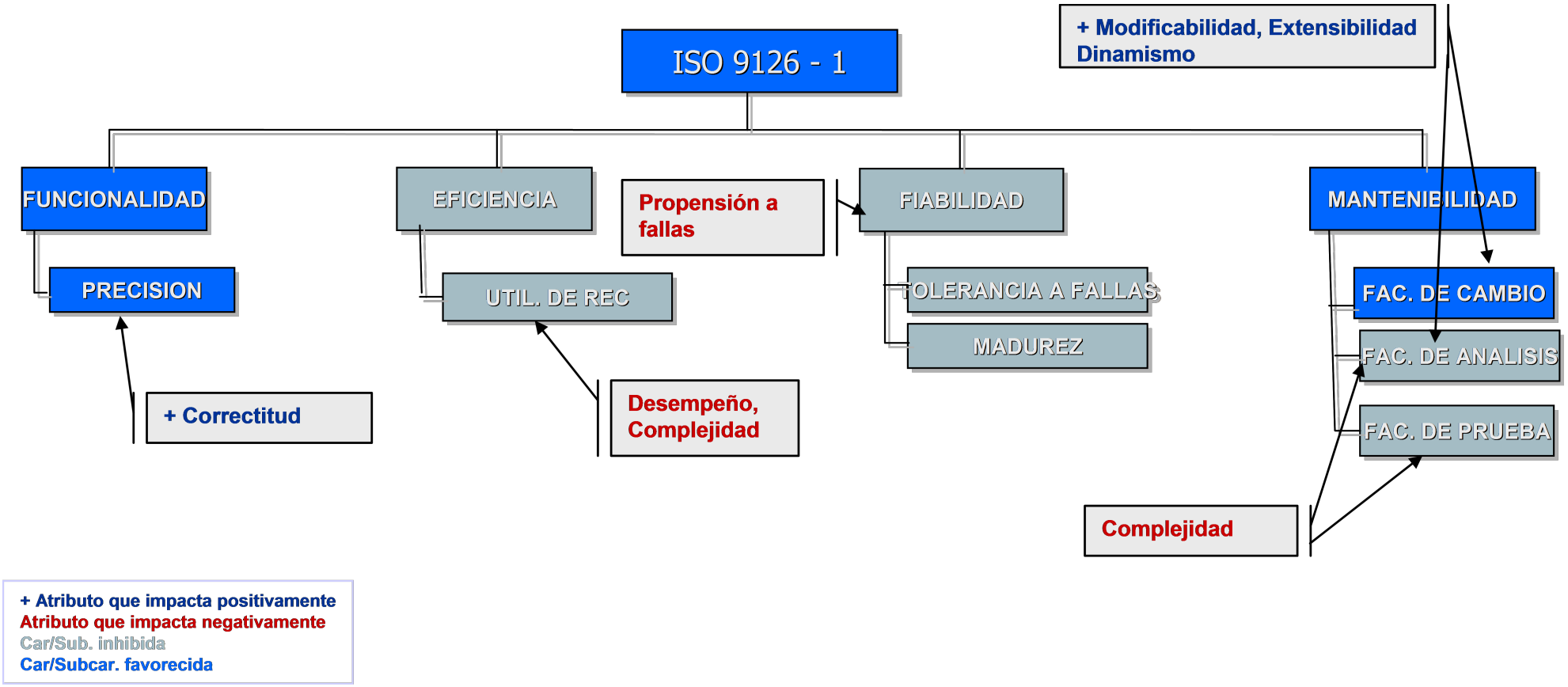
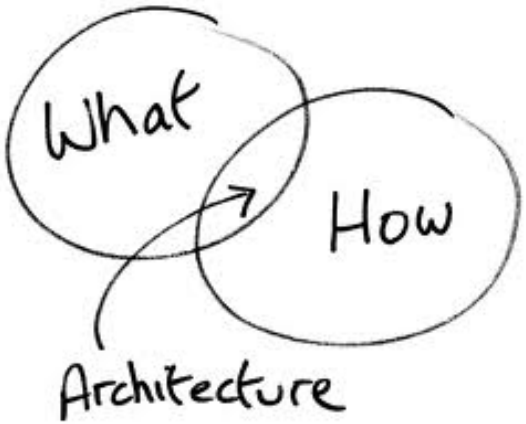


Figura: Características ISO del patrón “Reflection"

Características del proceso de diseño del software



- El diseño consiste en distintos *modelos*, que se obtienen iterativamente tras varias versiones
  - Supone regresar a modelos anteriores y corregirlos
  - El proceso de diseño es aún un proceso *adhoc*
- Se necesita mucho esfuerzo de desarrollo y mantenimiento de estos modelos y, para sistemas pequeños, puede no ser rentable

El proceso de diseño

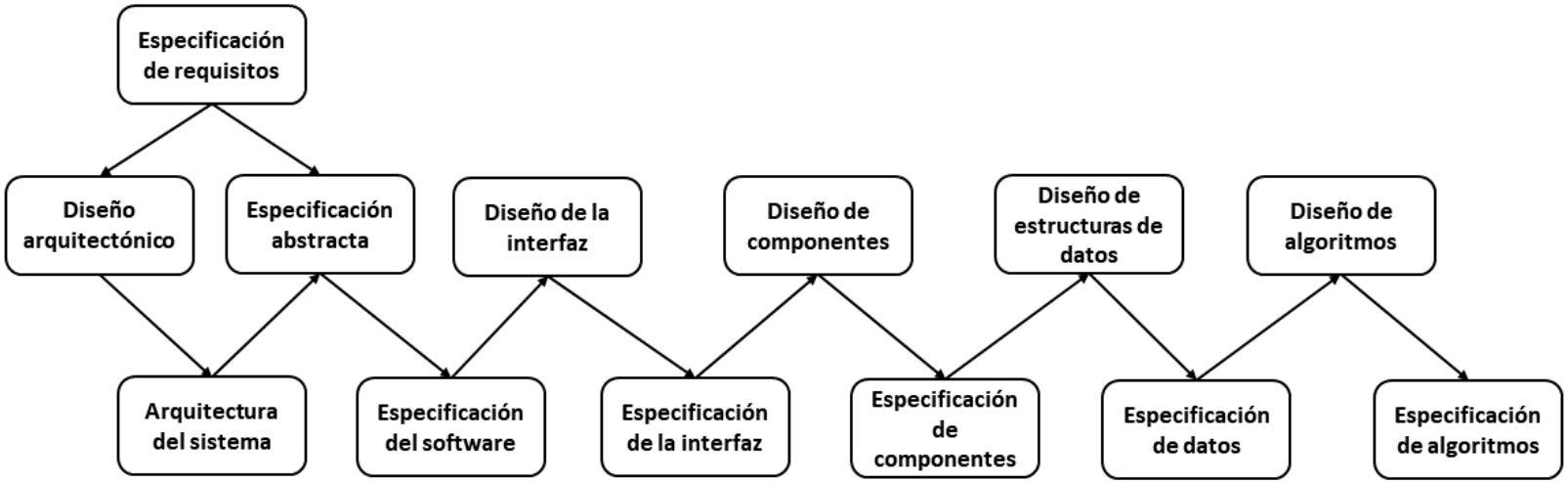


Figura: Proceso de Diseño de Software

- Desarrollo de varios modelos con diferentes niveles de abstracción:
  - Incluye agregar formalidad y detalles durante el desarrollo del diseño
  - Retroalimentación entre actividades

El proceso de diseño – II

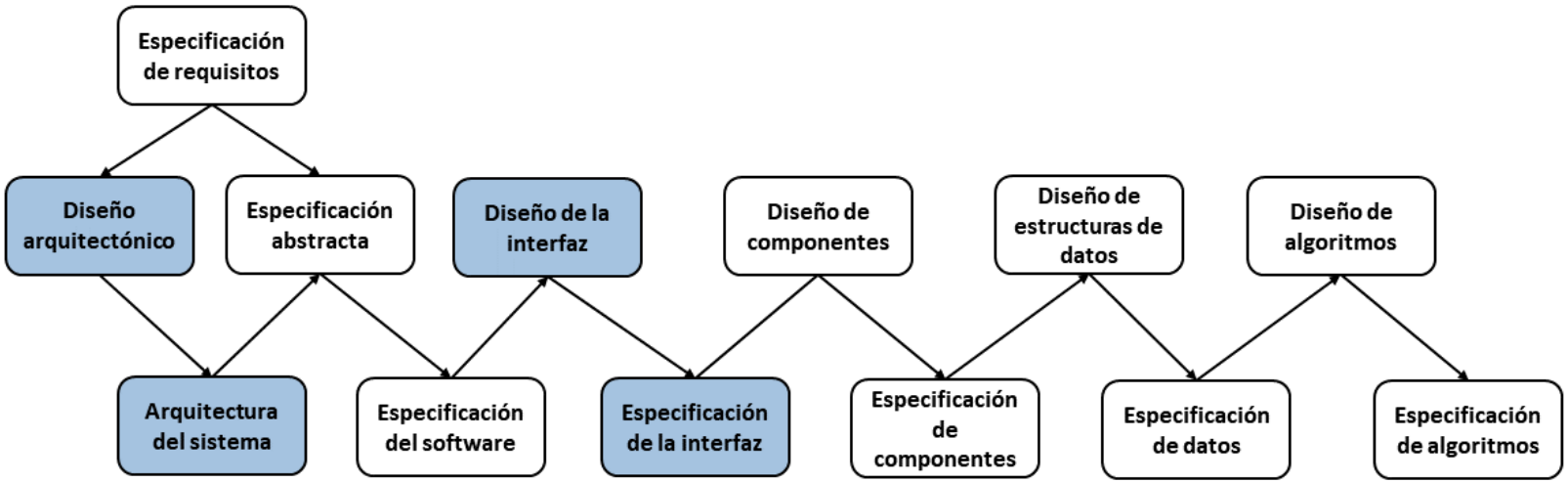


Figura: Proceso de Diseño de Software

- Ubicación dentro del proceso de desarrollo de software
- "Marco de trabajo" para componentes y comunicaciones
- Los modelos del diseño son una herramienta muy útil de comunicación entre distintos grupos de desarrolladores

Requisitos no–funcionales

- El estilo arquitectónico y estructura interna escogidos puede depender de requisitos *no-funcionales*: *rendimiento*, *seguridad*, *fiabilidad*, *disponibilidad* y *mantenibilidad*

Rendimiento

Clave: localizar las operaciones críticas dentro del menor número posible de subsistemas, que mantengan el mínimo acoplamiento posible

Seguridad (control acceso a la información)

Clave: utilizar una arquitectura *estratificada* donde los elementos más críticos se ubiquen en las capas más internas y cuyo acceso esté protegido muy rigurosamente

AS y requisitos no–funcionales II

Seguridad (prevención de daños)

Clave: todas las operaciones sensibles han de ser ubicadas en un solo subsistema o en un número pequeño de estos para conseguir la mayor protección

Disponibilidad

Clave: la arquitectura debe incluir componentes redundantes, tal que se puedan sustituir sin parar la ejecución del sistema

Mantenibilidad

Clave: utilizar componentes pequeños y autocontenidos que puedan ser cambiados con rapidez

Lograr equidad entre requisitos

Conflicto de intereses

- El potenciar un atributo arquitectónico correspondiente a un requisito no-funcional pueden entrar en conflicto con otros requisitos del sistema
  - Ejemplo: utilizar componentes grandes mejora el *rendimiento* pero perjudica la *matenibilidad* del sistema
- Se podría llegar a una solución de compromiso o bien utilizar diferentes *estilos arquitectónicos* para distintas partes del sistema

Diseño arquitectónico

Posterior a la especificación de requerimientos

- Solapamiento entre análisis/especificación requerimientos y el *diseño arquitectónico*
- La especificación de requerimientos no debería incluir ninguna información sobre diseño
- Descomposición arquitectónica y especificación del sistema
- Punto de partida previo a la especificación de subsistemas

Proceso de diseño OO

- Resaltar actividades clave sin atarse a ningún proceso *propietario* como RUP

Pasos a seguir

1. Entender y definir los modos de uso y el contexto del sistema WMS
2. Diseñar la arquitectura de software (AS)
3. Identificar los objetos principales dentro de la AS
4. Desarrollar los modelos de diseño
5. Especificar las interfaces de objetos

Ejemplo inicial de diseño: el caso WMS

Definition (Sistema de Información Meteorológica –WMS)

Un WMS se necesita para generar informes meteorológicos regularmente utilizando datos recogidos de estaciones de observación remotas y sin personal, así como de otras fuentes tales como observadores voluntarios, globos y satélites. Como respuesta de una petición, las estaciones transmiten sus datos al computador de área.

El sistema del computador de área valida los datos recogidos desde diferentes fuentes y los integra. Los datos integrados son archivados y, utilizando los datos desde este archivo y los de una base de datos de informes meteorológicos, se crea un conjunto de informes locales. Los informes se pueden imprimir para su distribución en una impresora de informes de propósito especial o bien se pueden mostrar en pantalla en diferentes formatos.

Contexto del sistema y modelos de uso

- Desarrollar una comprensión de las relaciones entre el software que está siendo diseñado y su entorno externo

Modelos iniciales

- Contexto del sistema
  - Un modelo estático que describe otros sistemas en el entorno. Utiliza subsistemas para mostrar a otros sistemas.
- Modelo del sistema
  - Un modelo dinámico que describe cómo interacciona el sistema con el contexto. Se utilizan *casos de uso* para mostrar las interacciones

Arquitectura software del WMS

Selección de una AS adecuada al ejemplo:

- Arquitectura *estratificada* (por capas)
- Cada capa sólo necesita de la capa inferior para poder realizar su proceso
- Capas identificadas:
  - Recoger Datos
  - Procesar Datos
  - Achivar Datos
  - Mostrar Datos

Arquitectura estratificada del WMS

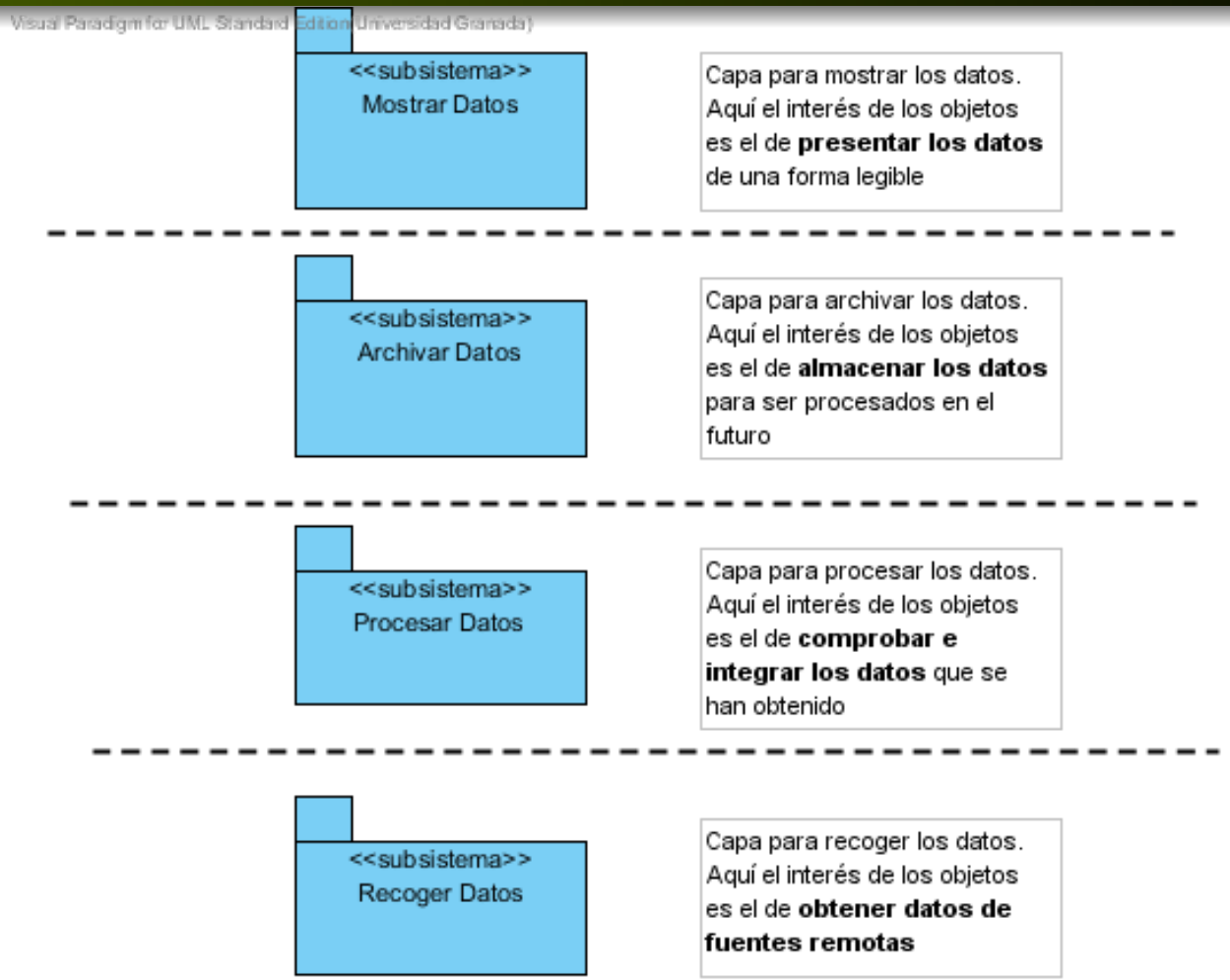


Figura: Arquitectura inicial *en capas* del WMS

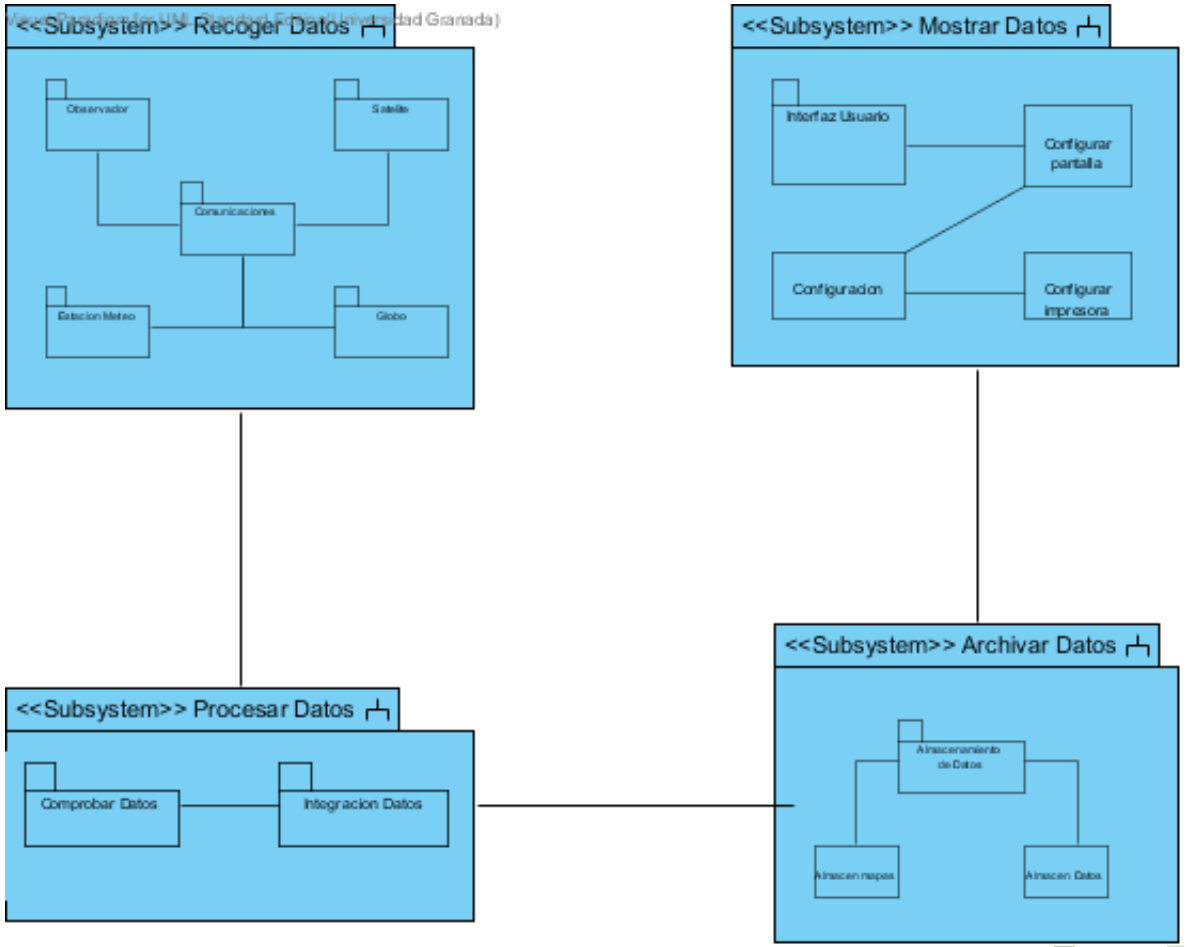
Elementos notacionales: UML

Selección de una notación adecuada para realizar el diseño

- Cada capa ha de incluir a distintos componentes del sistema
- Fácil identificación de subsistemas
- Posibilidad de generar código

Se utilizan *packaqes-UML* porque permiten representar colecciones de objetos y también a otros paquetes

Descomposición en subsistemas



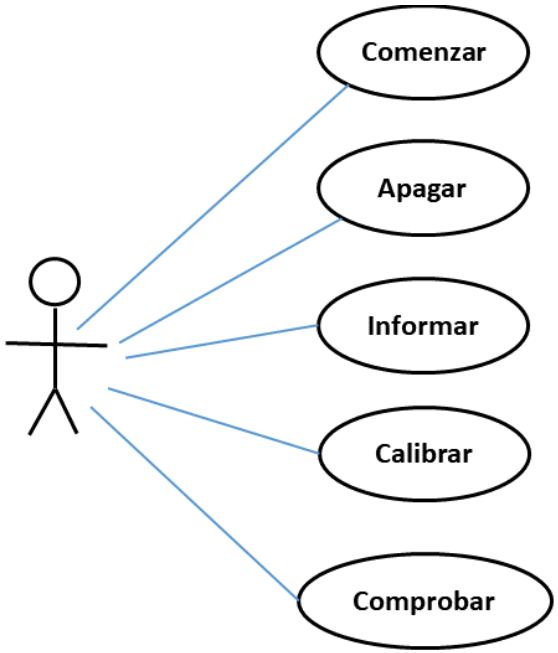
Modelos de uso y contexto del sistema

Modelo de casos de uso: objetivo

Desarrollar una comprensión de las relaciones entre el software que está siendo diseñado y su entorno

- La descripción del contexto del sistema suele ser un modelo estático que describe a otros sistemas
- Sin embargo, mejor utilizar un modelo dinámico para describir cómo interactúa el sistema realmente con su entorno
- Utilizar diagramas de *casos de uso*
- Un caso de uso se utiliza para representar cada interacción con el sistema
- Asociaciones entre casos de uso y descripciones según una plantilla

Casos de uso para el WMS



Sistema WMS- Caso de Uso "Informar"
Datos: la estación envía un resumen de datos meteorológicos que hayan sido recogidos de los instrumentos durante el periodo de recogida del sistema. Los datos enviados son las temperaturas promedio, máxima y mínima en tierra y en el aire; las presiones promedio, máxima y mínima y la velocidad del aire promedio, máxima y mínima, la lluvia caída total y la dirección del viento, muestreada cada 5 min.
Estímulos: el sistema de recogida de datos establece un enlace por *modem* con la estación meteorológica y demanda la transmisión de los datos.
Respuesta: los datos resumidos se envían al sistema de recogida de datos meteorológicos
Comentarios: La estaciones son demandadas para que informen 1 vez cada hora, pero esta frecuencia podría cambiar de una estación a otra y podría cambiar en el futuro.

Diseño Arquitectónico

Líneas-guía para obtenerlo

- Una vez que las interacciones entre sistema y entorno han sido comprendidas, se diseña la arquitectura del sistema
- Componentes *"lógicos"* de la Estación Meteorológica:
  1. *Interfaz*: comunicaciones con las otras partes del sistema
  2. *Recolección*: de datos: gestionar los datos recogidos por los instrumentos y el resumen de los datos meteorológicos antes de transmitirlos
  3. *Instrumentación*: encapsulación de los instrumentos utilizados para recoger los datos en bruto
- Descomponer los modelos hasta que sólo existan 7 entidades de modelado como máximo en un modelo arquitectónico

Arquitectura de la Estación Meteorológica

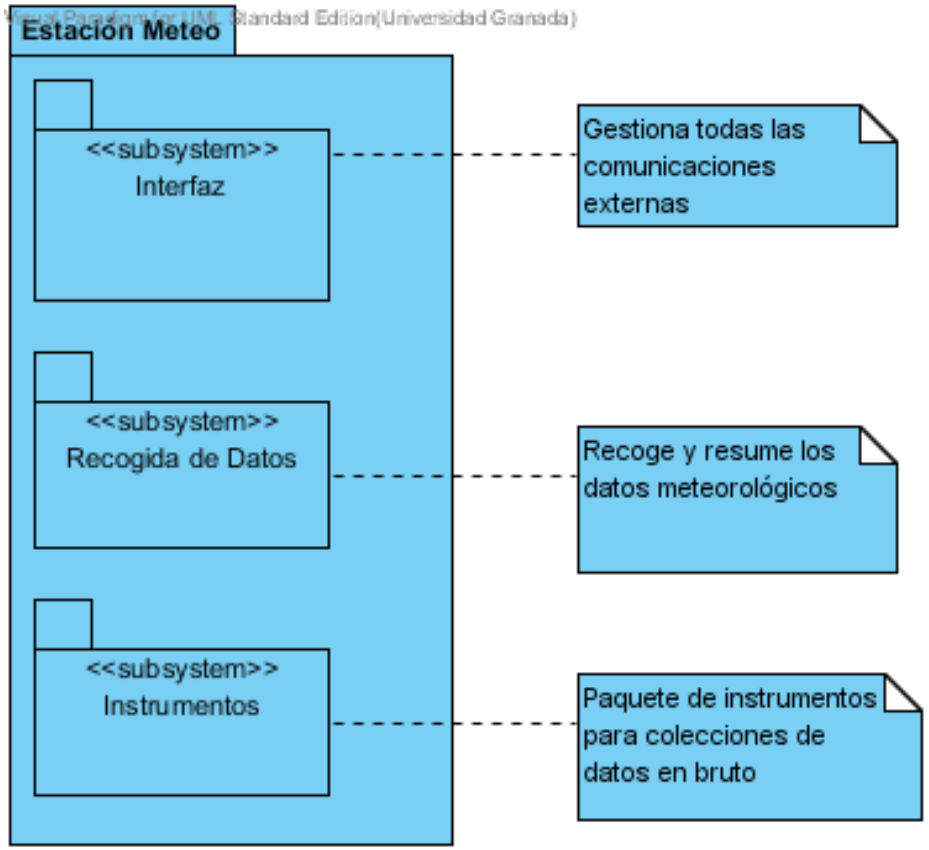


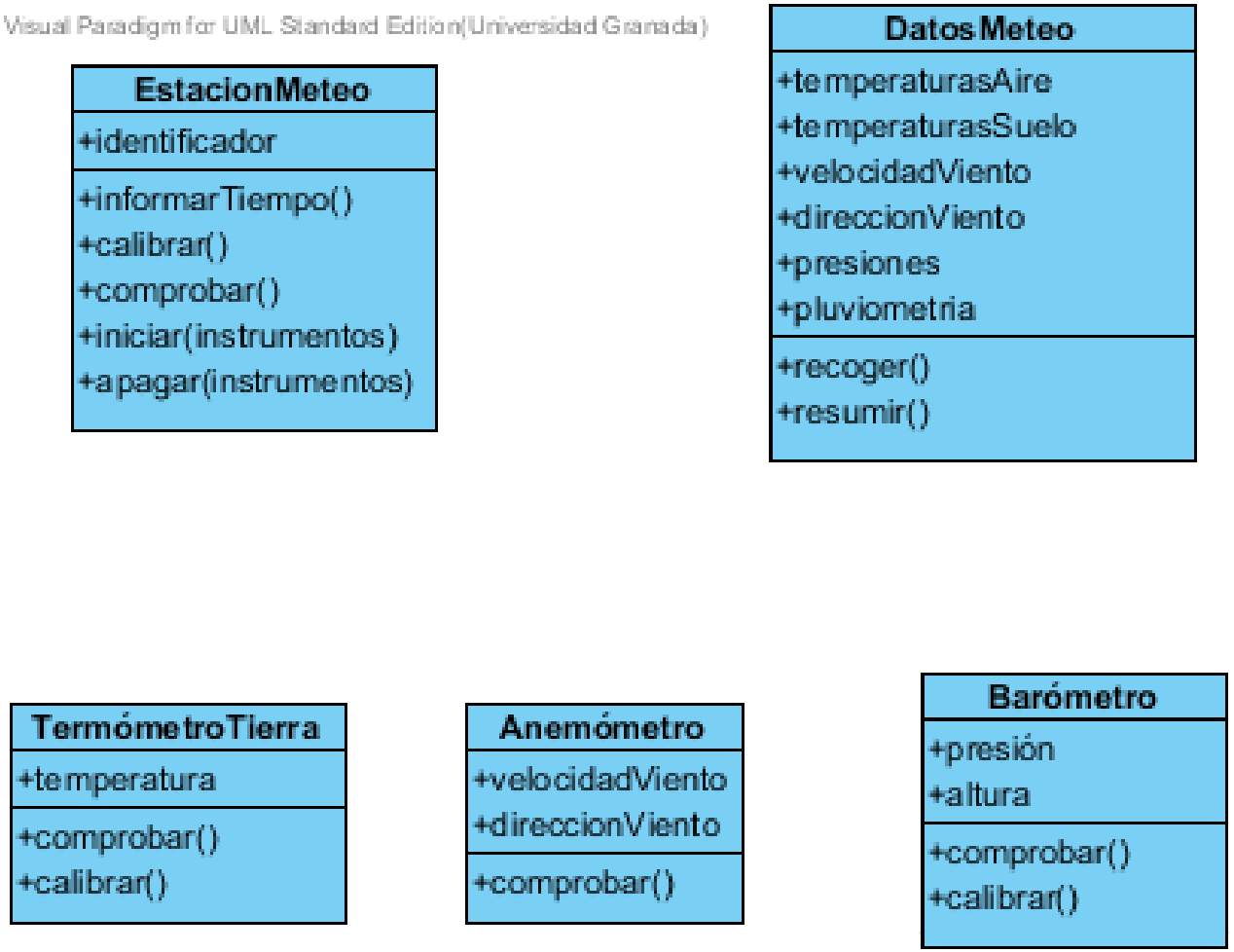
Figura: Arquitectura del sistema

3- Identificación de objetos principales dentro de la AS

¿Cómo diseñar la AS utilizando *objetos*?

- Los objetos *tienden a emerger* durante las primeras etapas del proceso de diseño
- El diseño tiene que ver con la identificación de las *clases*
- Identificación de atributos y operaciones de las clases
- Identificación de objetos de la estación meteorológica:
  - Objetos ("hardware") del dominio de aplicación:
    1. Termómetro Tierra
    2. Anemómetro
    3. Barómetro
  - Objetos procedentes de escenarios de casos de uso:
    1. EstacionMeteo (interfaz básica de la estación con su entorno)
    2. DatosMeteo (encapsula datos resumidos desde los instrumentos)

Clases de Objetos



Caracterización de los objetos encontrados

Determinación del carácter pasivo/activo

- Los objetos *pasivos* ejecutan sus métodos por petición del sistema: cambios en la política de recogida de datos no afecta a las clases que los modelan
- Objetos *activos* incluyen su propio *control*, deciden cuándo realizan las lecturas de los instrumentos: perjudica la interoperabilidad

4- Desarrollar los modelos de diseño

- Los modelos de diseño muestran los objetos, las clases de objetos y las relaciones entre éstas entities:
  1. *Modelos estáticos*: clases, objetos, relaciones de generalización, usa/usado–por, composición
  2. *Modelos dinámicos*: diagramas de secuencia y de estados, entre otros (UML proporciona 12 modelos dinámicos de diagramas)
- Modelos de subsistemas: incluyen componentes y asociaciones (por ejmplo, los diagramas *UML-packages*)
- *Secuencias*: para cada modo de interacción del sistema, la secuencia de llamadas entre objetos que tienen lugar
- *Estados*: comportamiento de un solo objeto en respuesta a los mensajes que puede procesar

Subsistemas de una estación meteorológica

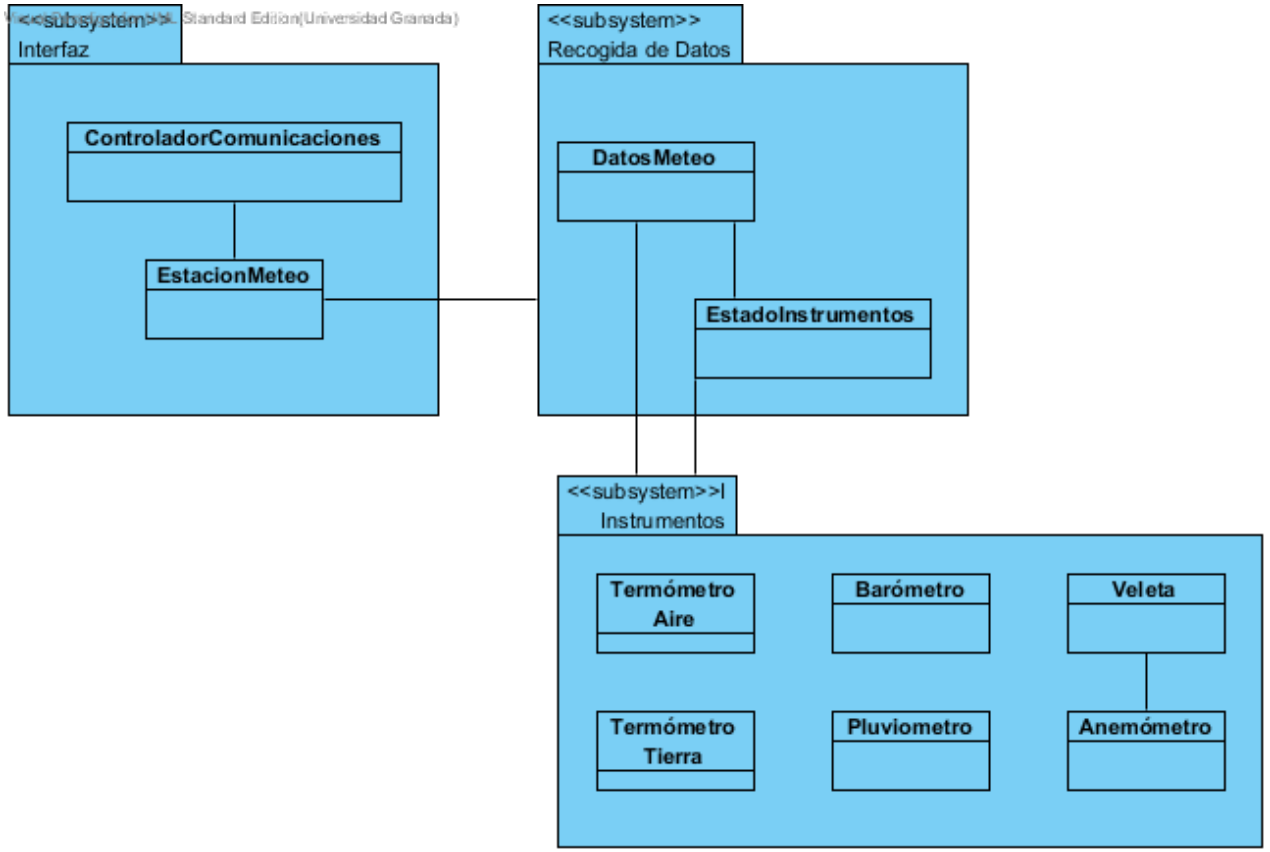


Figura: Subsistemas de la Estación Meteorológica. Se trata de un modelo lógico: la organización de los objetos en el sistema podría ser diferente

Modelos de secuencia de operaciones

Idea fundamental

Estos modelos muestran la secuencia de interacciones entre objetos que tienen lugar en el sistema

Diagramas de secuencia-UML

- Los objetos se alinean horizontalmente en la parte superior;
- El tiempo se representa verticalmente, así los modelos se leen de arriba abajo;
- Las interacciones se representan mediante flechas con etiqueta, diferentes estilos de flecha representan diferentes tipos de interacción
- Un rectángulo delgado en la *línea de vida* de un objeto representa el tiempo en el cual el objeto controla el sistema

Modelos de secuencia de operaciones–II

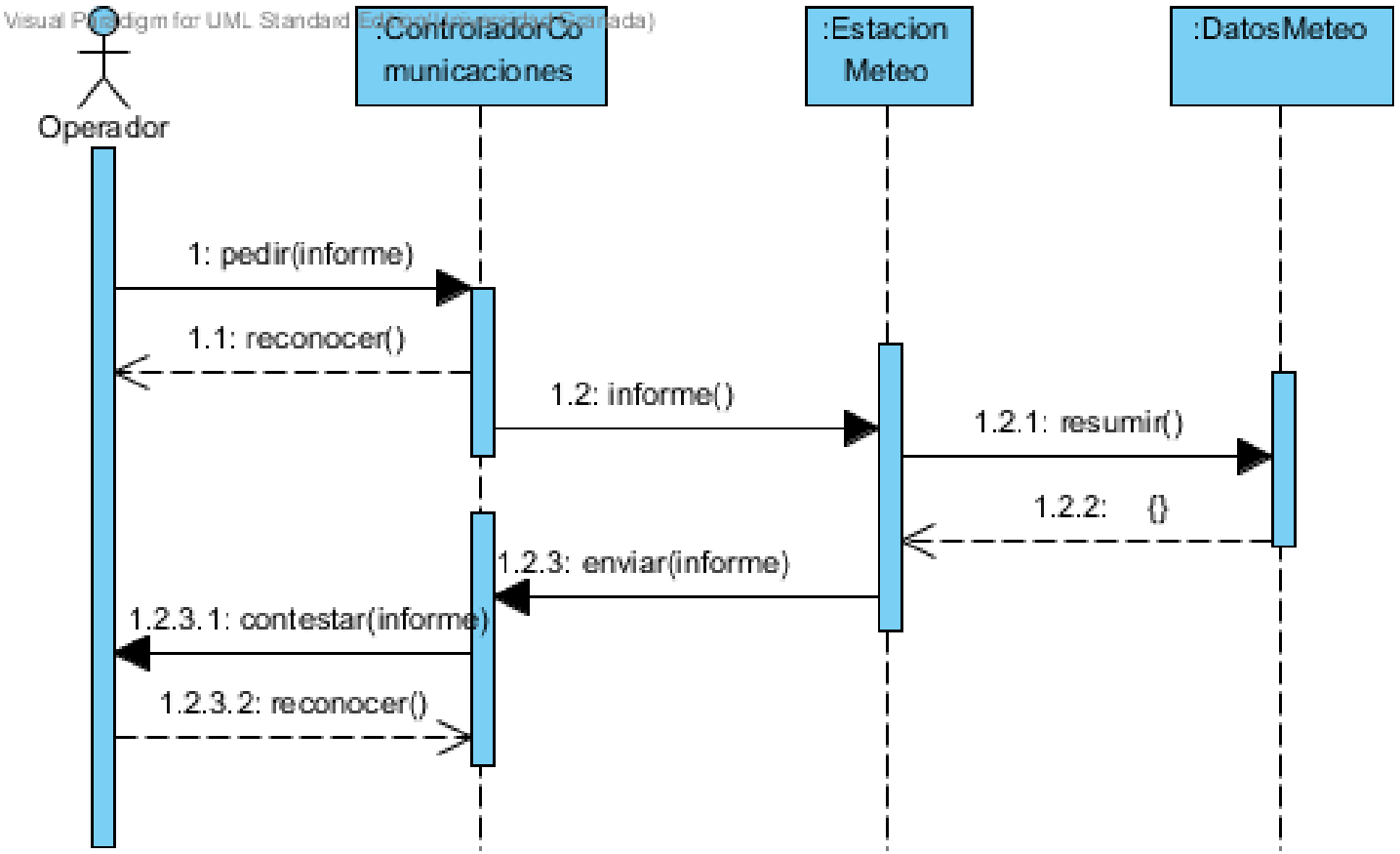


Figura: Recogida datos: secuencia operaciones

Modelo basado en diagramas de estados UML

Idea fundamental

Muestra cómo responden los objetos a distintas peticiones de servicio y las transiciones entre estados disparadas por estas peticiones

Explicación del Statechart

- Si el estado del objeto es Apagar, responde a un mensaje `apagar();`
- En el estado Esperar el objeto espera más mensajes;
- Si es `informarTiempo()`, el sistema pasa al estado Resumir;
- Si es `calibrar()`, el sistema pasa al estado Calibrar;
- Se entra en un estado colectivo cuando se recibe una señal del reloj

Modelo basado en diagramas de estados UML-II

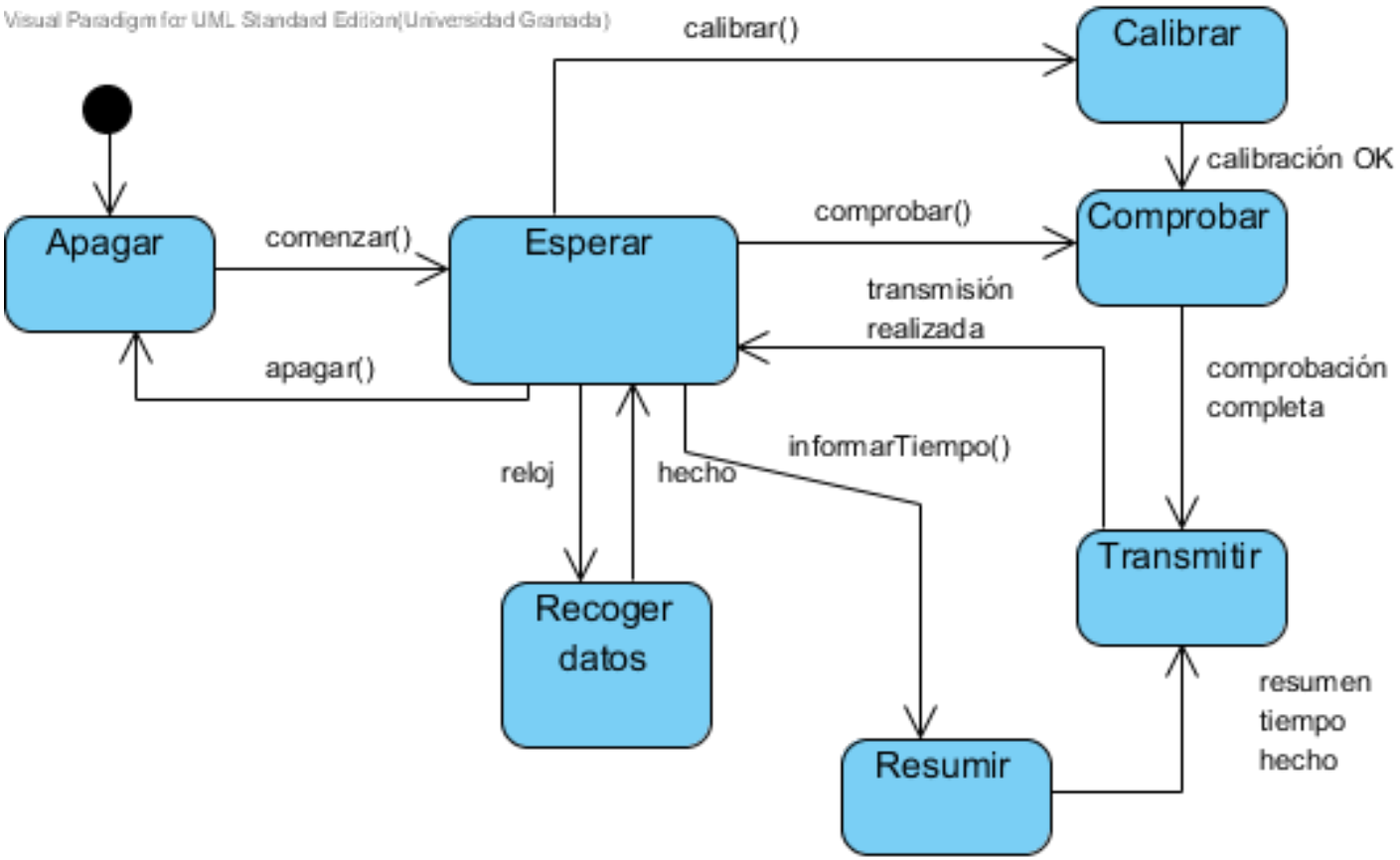


Figura: Estación Meteo: diagrama de estados

5- Especificar las interfaces de objetos

- Especificar las interfaces de los objetos de tal manera que los objetos y los subsistemas se puedan diseñar *en paralelo*
- No incluir detalles de la representación en el diseño
- Sólo proporcionar las operaciones de los objetos necesarias para acceder y actualizar los datos del objeto
- No tiene por qué existir una relación 1:1 entre objetos e interfaces
- Utilizar un lenguaje de programación adecuado (como Java) para definir las interfaces

Interfaz de la Estación Meteorológica

```
interface EstacionMeteo {
public void EstacionMeteo () ;
public void comenzar () ;
public void comenzar (Instrumento i) ;
public void apagar () ;
public void apagar (Instrumento i) ;
public void informarTiempo ( ) ;
public void comprobar () ;
public void comprobar ( Instrumento i ) ;
public void calibrar ( Instrumento i) ;
public int getID () ;
} //EstacionMeteorologica
```

Evolución del diseño

- Ocultar información dentro de los objetos significa que los cambios realizados a un objeto no afectarán a los otros objetos de una forma impredecible
- Suponer que hemos de añadir funcionalidad a las estaciones meteorológicas para monitorizar la polución. Se trata de tomar muestras del aire y calcular la cantidad presente de distintos contaminantes en la atmósfera
- Las lecturas de polución se transmitirán con los datos meteorológicos

Cambios necesarios

- Añadir una clase denominada `CalidadAire` como integrante de la estación meteorológica
- Añadir una operación `informarCalidadAire()` a la clase `EstacionMeteo`. Modificar el software de control para recoger lecturas de polución
- Añadir objetos que representen a instrumentos de medida de la polución

Monitorización de la polución

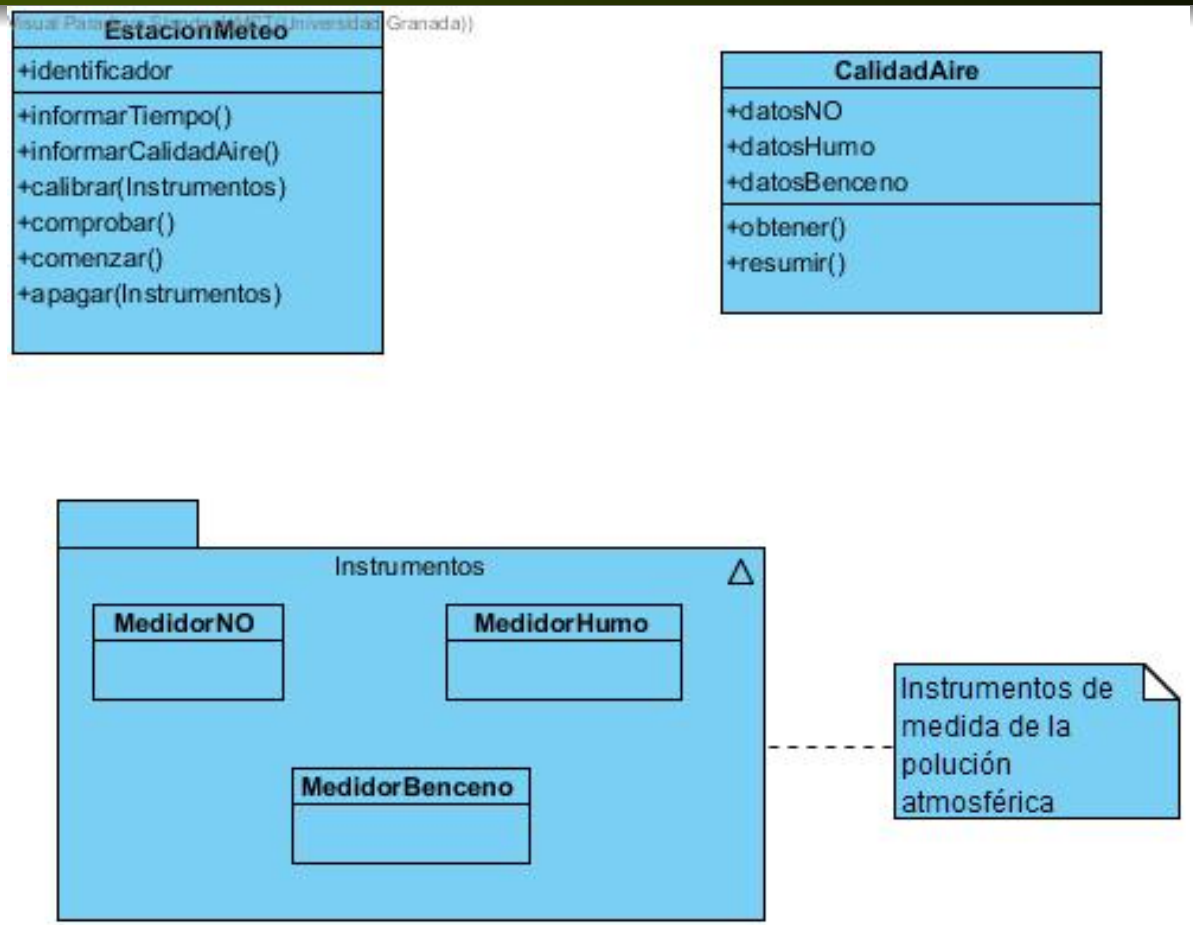


Figura: Modificación de la arquitectura de la Estación Meteorológica

Lenguajes de Descripción Arquitectónicos

ISO/IEC/IEEE 42010

- Guiar en la construcción y mantenimiento del sistema
- Ayudar a planear los costes y evolución del sistema
- Servir como un medio para el análisis, evaluación o comparación de arquitecturas software
- Facilitar la comunicación entre las partes interesadas en las arquitecturas y los sistemas
- Documentar el conocimiento arquitectónico más allá del ámbito de los proyectos individuales
- Capturar idiomas arquitectónicos reutilizables (tales como estilos arquitectónicos y patrones)

ADL

Concepto

- Resuelven el problema de la representación formal de una arquitectura software desde un punto de vista sintáctico y semántico [Bass et al., 2012] [Clements and et al., 2010]
- Un ADL permite representar, analizar y especificar una AS
- Pueden ser descriptivo–formales o semiformales, gráficos, o ambas cosas
- Algunos han sido sólo definidos formalmente y otros cuentas con herramientas que los soportan

ADL

Actuales

ACME (Carnegie-Mellon), CODE, UNICON, Wrigth, RAPIDE, Darwin (Imperial College), xADL (University of California, Irvine–UCI), DAOP-ADL (Universidad de Málaga) y ByADL (Universidad de L'Aquila)

Características

- Los primeros ADLs hacían más énfasis en el modelado de componentes, conectores y configuraciones.
- Los ADLs actuales (ArchiMate, SysML, etc.) tienden a ser lenguajes descriptivos de un espectro mucho más amplio.
- Se pueden utilizar lenguajes de propósito general como UML como ADLs. así como para modelar procesos de negocio y similares.

Requerimientos que han de satisfacer los ADLs

[Bass 1998][Luckhan y Vera 1995][[M. Shaw, 1996]]

- Capacidad de representación de componentes y elementos de análisis arquitectónico (conectores, interfaces, etc.)
- Proporcionar capacidad de análisis con el soporte de herramientas software
- Integridad comunicacional
- Soporte a las tareas de creación, refinamiento y validación de una AS

Requerimientos que han de satisfacer los ADLs

- Capacidad para representar la mayoría de los patrones arquitectónicos conocidos, directa o indirectamente
- Capacidad de proveer vistas del sistema que expresen información arquitectónica
- Soportar la especificación de familias de implementaciones que satisfacen una arquitectura común
- Capacidad de análisis basada, o bien la capacidad para la rápida generación de prototipos

Lenguajes de descripción de arquitecturas

ADLs actuales

- **Rapide**: se basa en la noción matemática de *power sets* y posee estructuras de programación muy potentes.
- **UniCon**: un ADL pensado para ayudar a los diseñadores a definir AS utilizando abstracciones identificadas.
- **Aesop**: intenta dar solución al problema de la reutilización de estilos arquitectónicos.
- **Wright**: se trata de un lenguaje formal que incluye *componentes* con *puertos*, *conectores* con *roles* ... y el *pegamento* para ligarlos.
- **ACME**: un ADL de segunda generación, que intenta hacer cooperar a los ADLs.
- **UML**: como ADL incluye muchos elementos de modelado arquitectónico pero le falta una definición formal.

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`

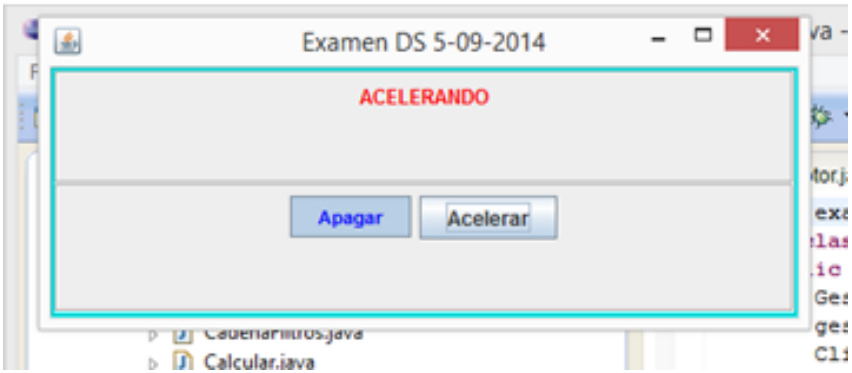


Tabla: Se muestran 2 instantes de la ejecución de la interfaz: (a) inicialmente; (b) después de pulsar el botón “Encender”

Ejemplo de ADE: SCACV con 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.ActionEvent evt){
                      BotonAcelerarActionPerformed(evt)}});
subpanel.add(BotonAcelerar);
BotonAcelerar.setBounds(10,10,90,23);
```

Ejemplo de ADE: SCACV con Swing II

```
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"); }}
synchronized private void BotonAcelerarActionPerformed(java.awt.event.ActionEvent evt
      ){
     if (BotonEncender.isSelected()){
          EtiqMostrarEstado.setForeground(Color.red);
     EtiqMostrarEstado.setText("ACELERANDO");}}
```

Concepto de Servicio

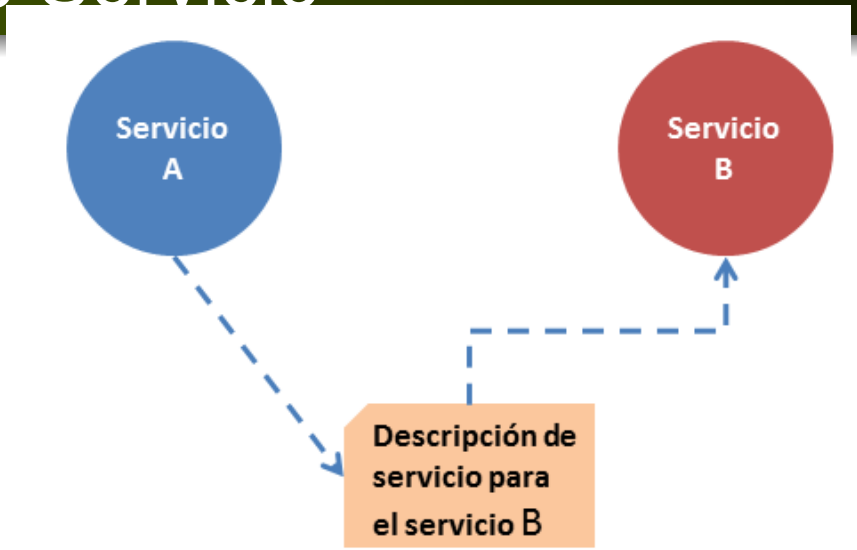


Figura: Acceso entre servicios

Servicio

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

Concepto de Servicio II

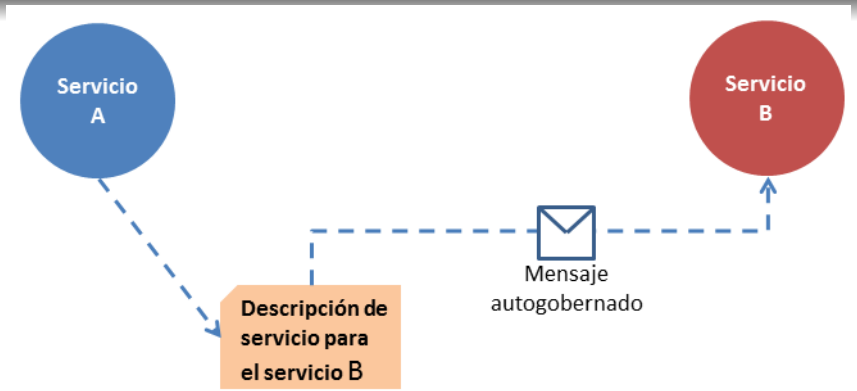


Figura: Acceso entre servicios

- 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*

Motivación para utilizar un SOA

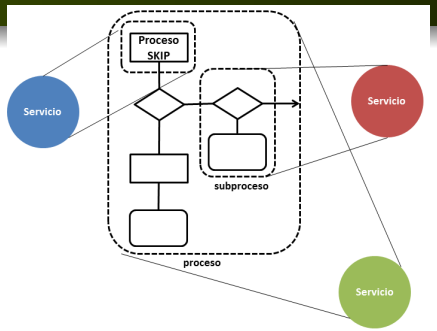
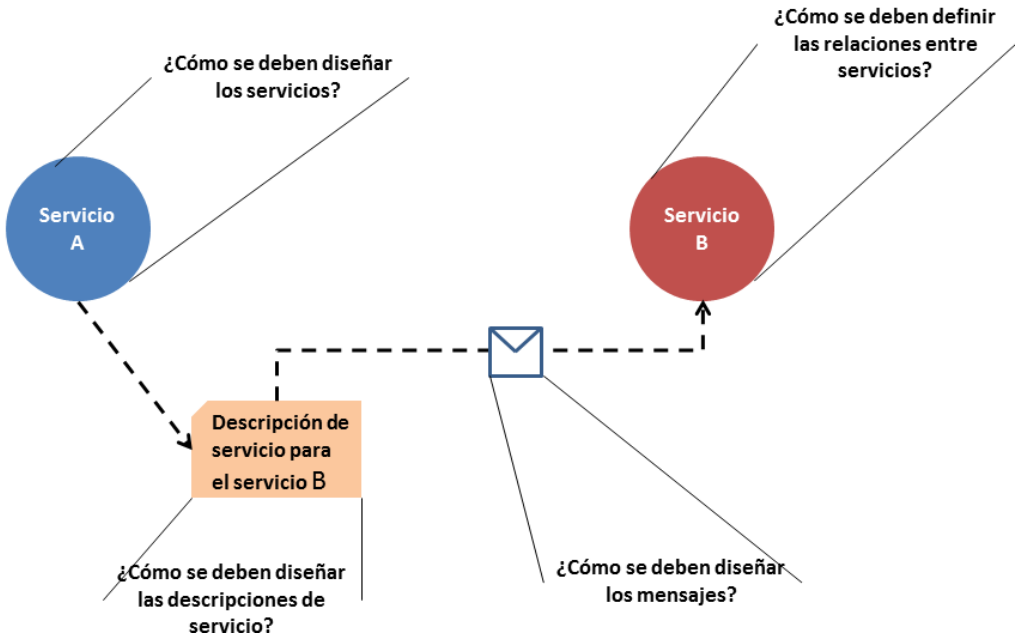


Figura: Aplicación con varios servicios

- Facilidad para la combinación de módulos de *funcionalidad rica* en el desarrollo de aplicaciones
- Minimizar costes: coordinación entre componentes ya existentes
- Interacciones a *nivel de servicios*, no a *nivel de módulos*
- Conseguir implementar el concepto de *pay per service* en el mercado de componentes software
- Repartir costes entre las *partes interesadas* (stakeholders) en un determinado producto software
- Promover el concepto de *orientación a servicios* en la industria de desarrollo de software

Arquitecturas orientadas a servicios



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

Principios de Orientación a Servicios

i. Bajo acoplamiento
ii. Contrato de servicio
iii. Autonomía
iv. Abstracción
v. Reusabilidad
vi. Composicionalidad
vii. Ausencia de estado
viii. Facilidad de descubrimiento

Definiciones de SOA

IBM

"Un modelo de componentes que interrelaciona las diferentes unidades funcionales de las aplicaciones, denominadas servicios, a través de interfaces y contratos bien definidos entre esos servicios. La interfaz se define de forma neutral, y debería ser independiente de la plataforma hardware, del sistema operativo y de los lenguajes de programación utilizados. Esto permite a los servicios, construidos sobre sistemas heterogéneos, interactuar entre ellos de una manera uniforme y universal."

Definiciones de SOA-II

Microsoft

"La Arquitectura SOA establece un marco de diseño para la integración de aplicaciones independientes de manera que desde la red pueda accederse a sus funcionalidades, las cuales se ofrecen como servicios."

Ideas fundamentales acerca de un SOA

Un SOA no es un *software concreto*

sino una metodología, una guía de trabajo y marco conceptual

¿Qué ha de contener?

Lógica de negocio y los datos de todos los sistemas informáticos de las empresas, aprovechando las red y determinados tipos de servicios en la Web

Definiciones de SOA-III

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"

Encapsulación de la “lógica de negocio”

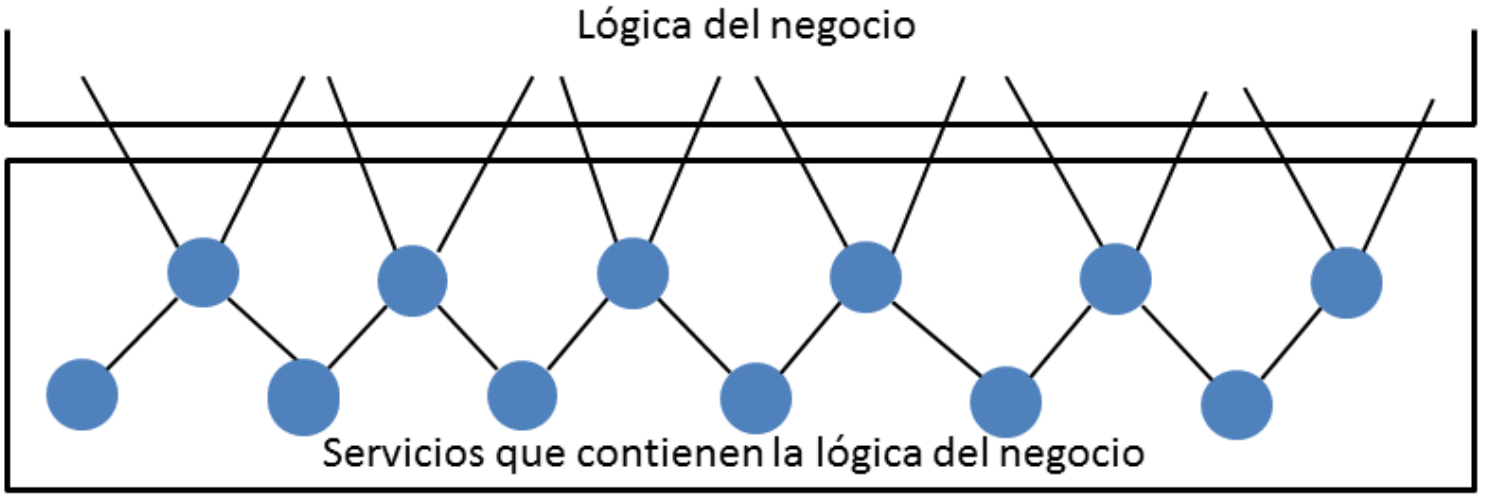


Figura: Capa de servicios que encapsulan la lógica de negocio de una aplicación

Encapsulación de la “lógica de negocio” II

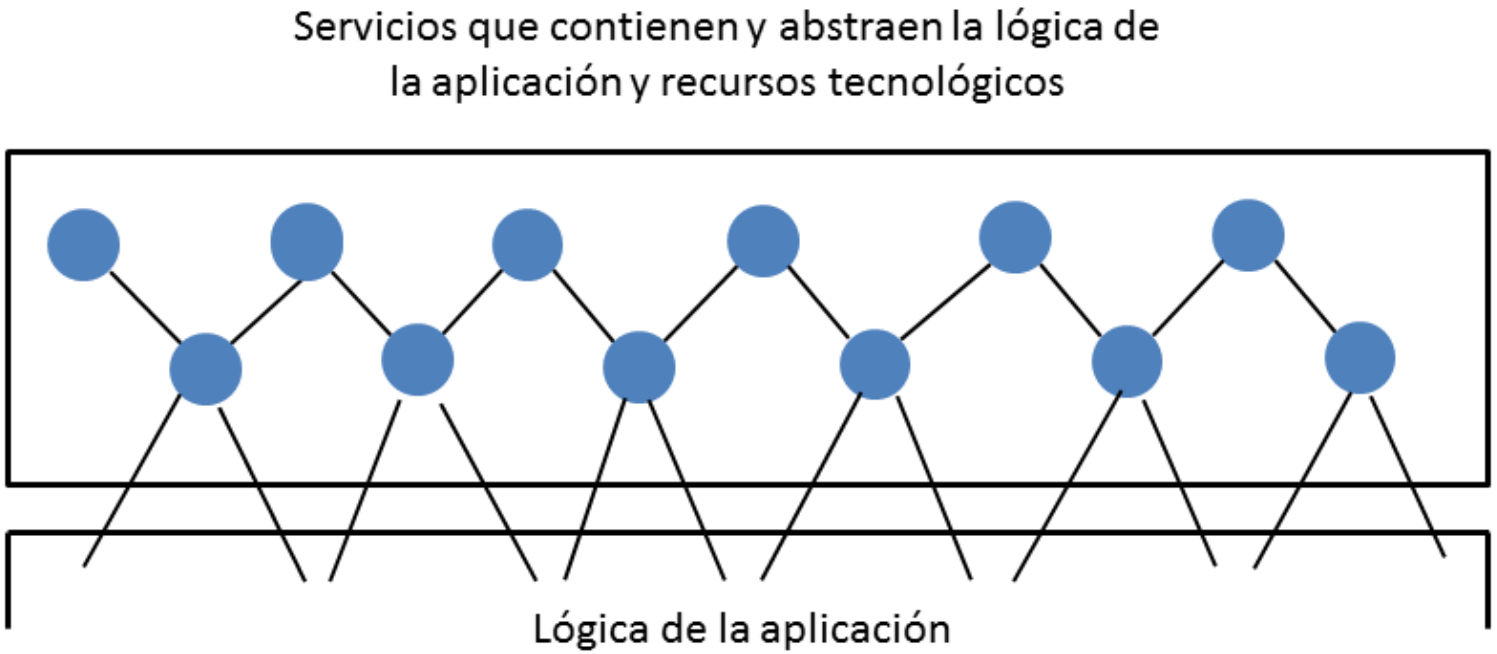


Figura: Abstracción de una lógica de negocio representada con tecnología privativa

Encapsulación de la “lógica de negocio” III

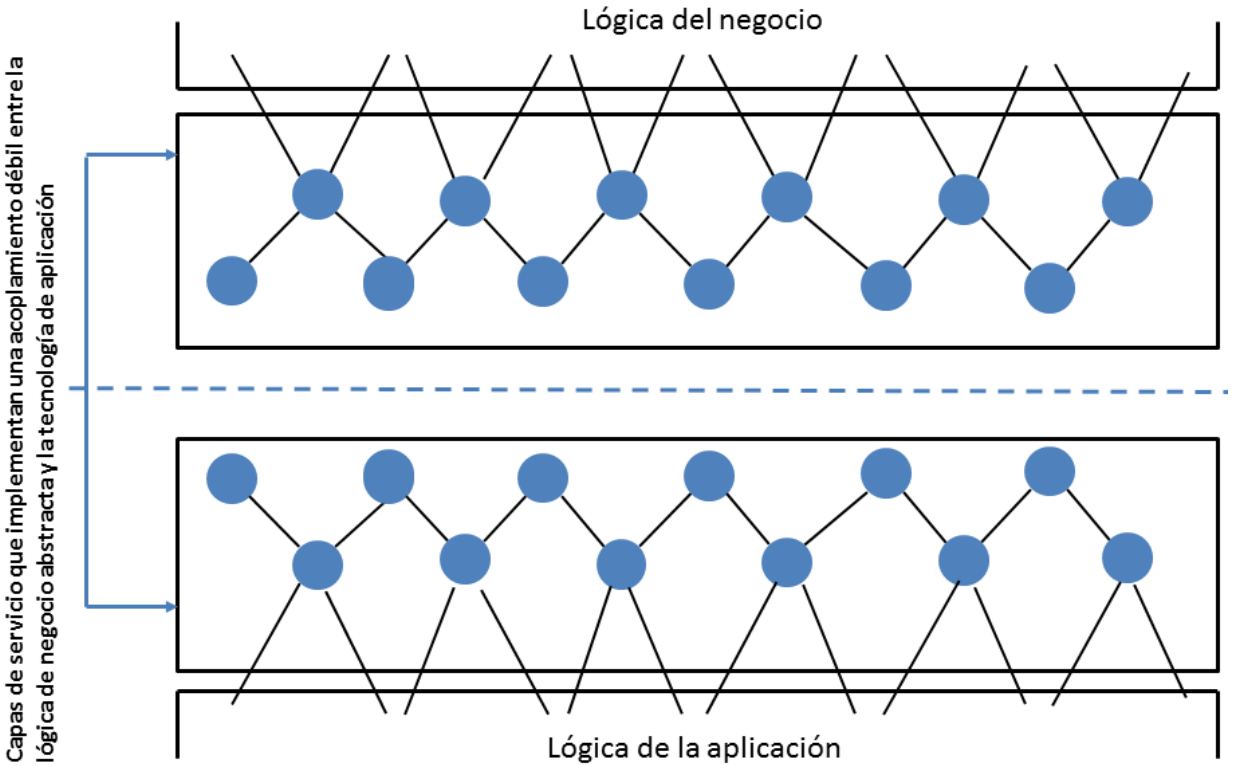


Figura: Abstracción de lógica de negocio en capas para implantar un SOA

Objetivos de un SOA

- Modelar la denominada *lógica de negocio* de una empresa como servicios: se puede expresar la *capa de negocio* mediante la orquestación de servicios
- Crear una capa de servicios que ofrezca la funcionalidad de la capa de aplicación, independientemente de la tecnología que la soporte
- Minimizar las dependencias entre la capa de negocio y la capa de aplicación para desacoplar el negocio de la tecnología
- Reutilizar los servicios de negocio creados en la organización
- Reducir el impacto de la evolución de la tecnología en las aplicaciones de negocio y mejorar la flexibilidad y agilidad de los sistemas
- Desacoplamiento, estandarización e interoperabilidad entre capas de aplicaciones (conseguir *agilidad tecnológica*); mejorar productividad y deslocalización de servicios

Implementación de *agilidad* tecnológica en una organización

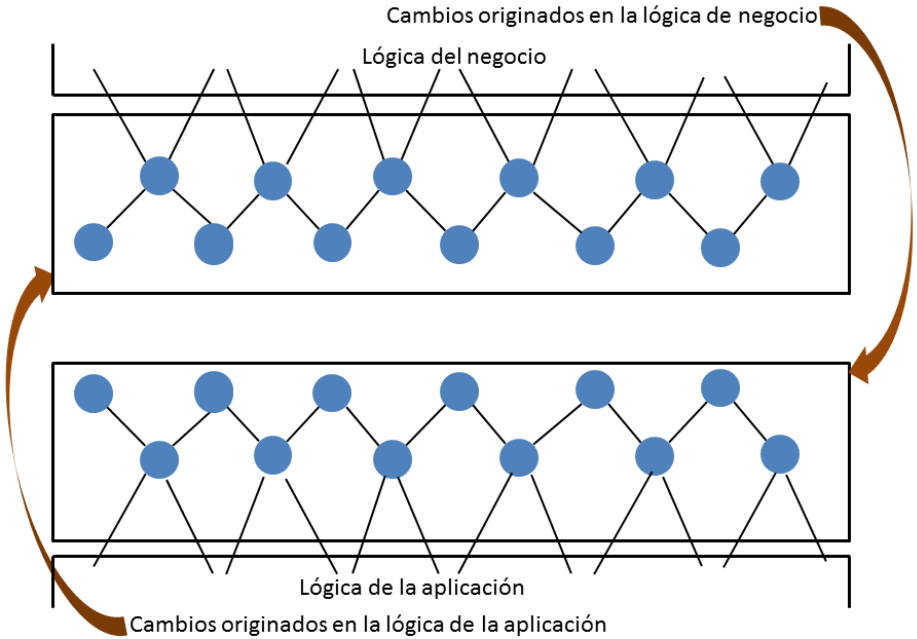
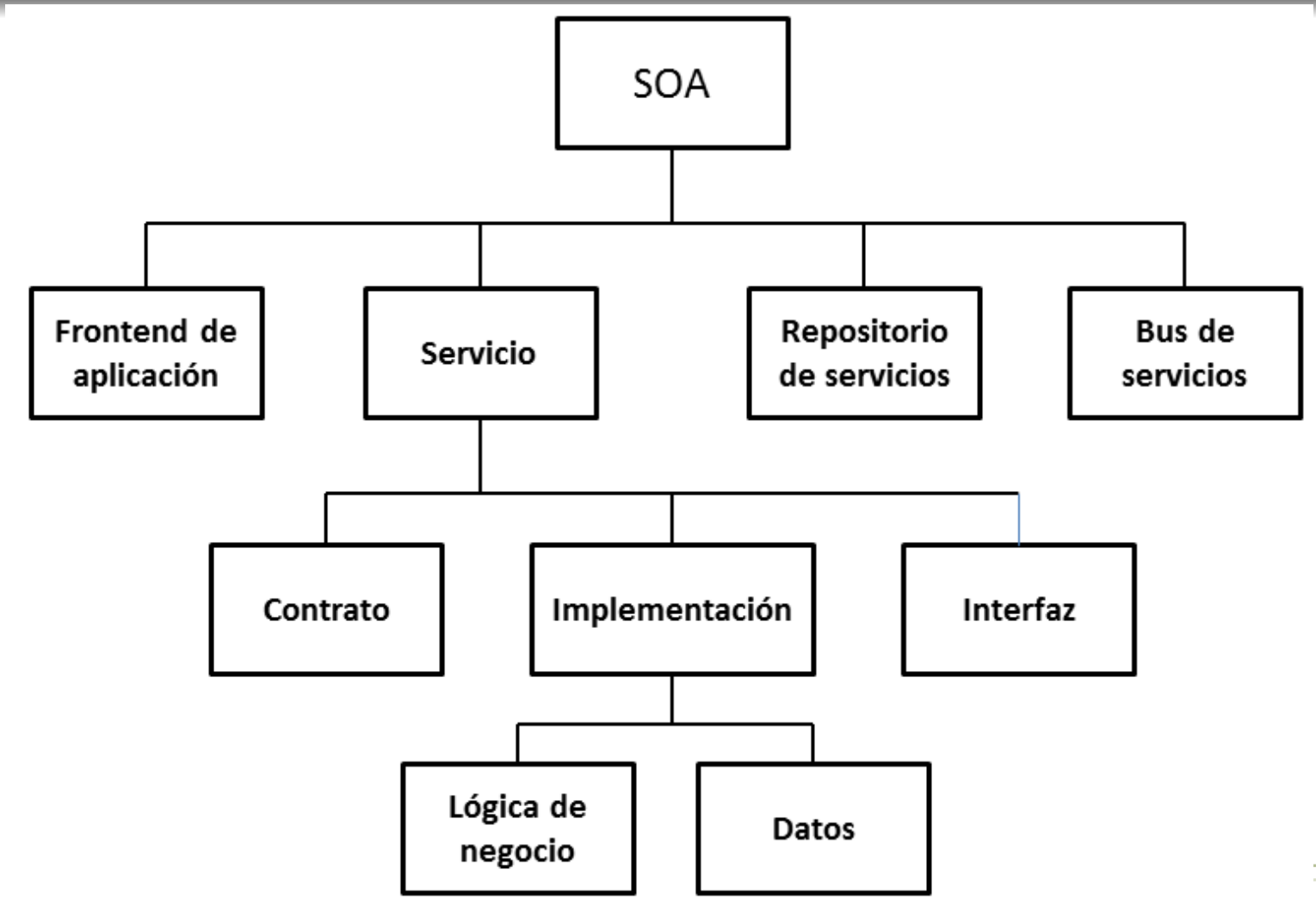


Figura: Facilidad de cambios mediante acoplamiento débil entre negocio y tecnología de aplicaciones

Elementos de un SOA



Diferencias entre SOA y *arquitecturas orientadas a objetos*

- Un SOA está basado en *componentes software*, no en objetos
- Está soportado por *estándares abiertos*: XML, SOAP, WSDL, UDDI, REST, GraphQL, ...
- Máxima interoperabilidad entre servicios gracias al uso de estándares
- Extensibilidad independiente de los servicios, cosa que no tienen los objetos
- Desarrollo ágil de aplicaciones
- Facilidades evolutivas del software

SOA y Servicios Web

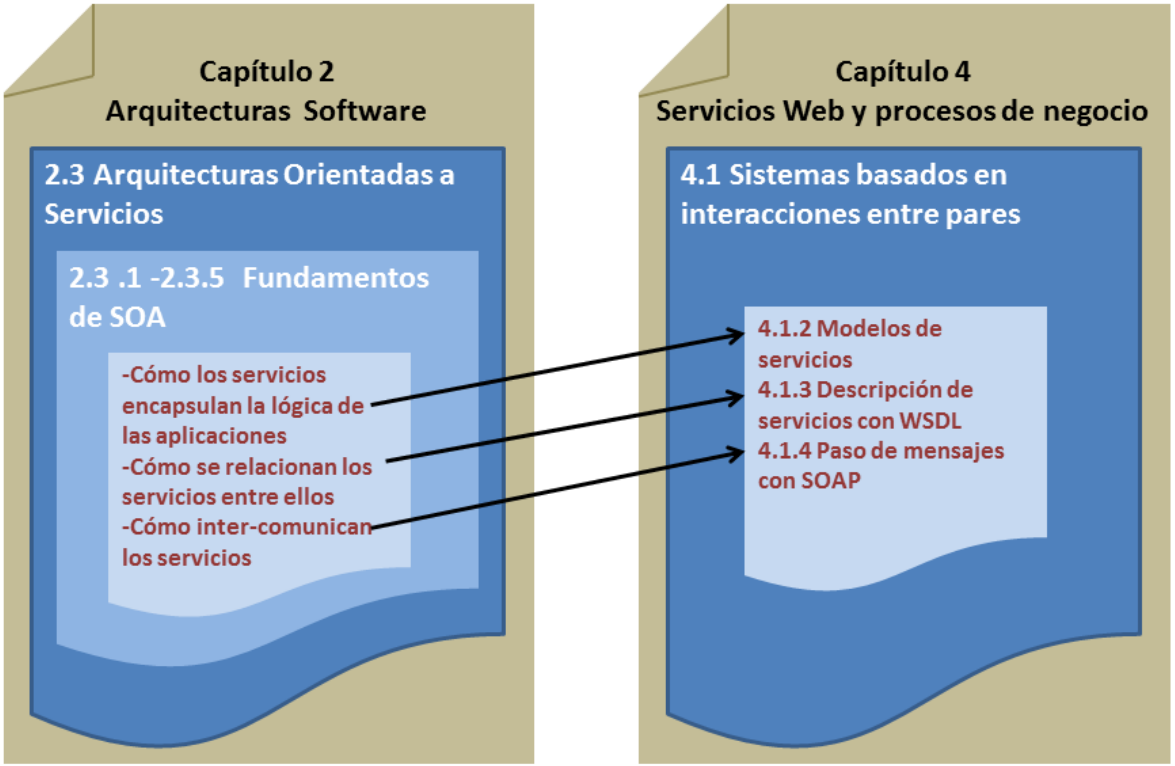


Figura: Relación estructural entre SOA y Servicios Web (“Desarrollo de software y sistemas basados en componentes y servicios”, M.I.Capel(2017))

Servicios Web

Los Servicios Web son servicios implementados utilizando estándares como SOAP, WSDL y UDDI para ser accedidos a través de una red, preferiblemente Internet, y ejecutados de manera remota.

Protocolo estándar

Para esto es necesario un protocolo de comunicación común para todos estos elementos de la arquitectura, el protocolo estándar es el Service Oriented Architecture Protocol (SOAP).

Lenguaje de descripción de servicios

Web Service Description Language (WSDL).

Descubrimiento de servicios disponibles

Los servicios deben ser clasificados por categorías basadas en los servicios que ofrecen y la forma en que deben ser invocados, el mecanismo estándar es Universal Description Discovery and Integration (UDDI).

Extensión de la definición de SOA actualmente

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".

SOA y modelado de procesos de negocio

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?
- Sobrecarda 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

Implementación de SW

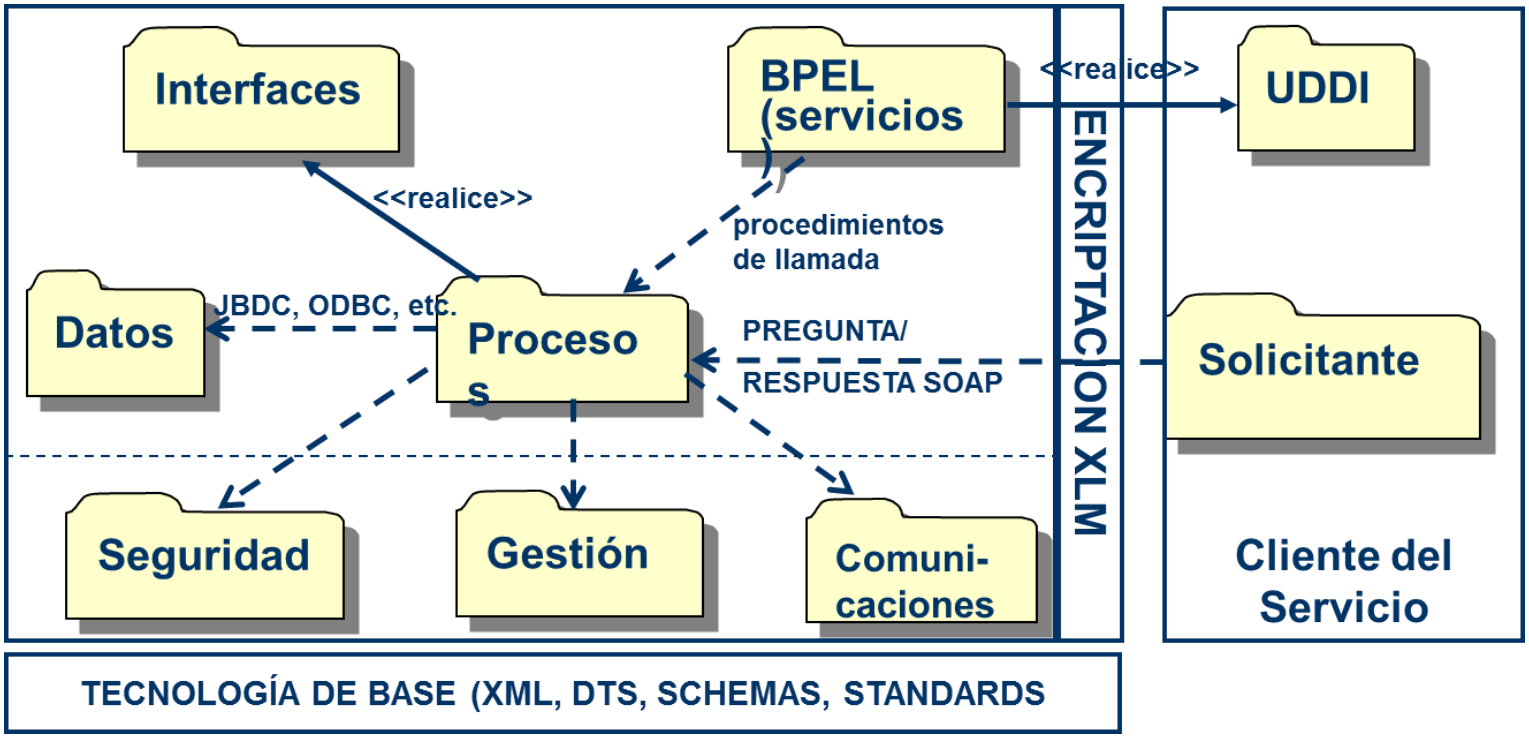


Figura: Implementación de una arquitectura para Servicios–Web

Marco de implementación de SW

Caracterización Servicios Web *de base*

- Ha de ser *abstracto*:
  - definido por organización de estándares (ISO, ...)
  - implementado en todas las plataformas
- Bloques constructivos, descripciones de servicios y de mensajes, basados en WSDL
- Componente de mensajería basado en SOAP y sus conceptos
- Registro de descripción de servicios, a veces se incluye descubrimiento de servicios, basado en UDDI
- Arquitectura software bien definida con mensajería, patrones y composición

Tecnologías

XML

El lenguaje de marcas XML se basa en la combinación de texto junto con información que describe ese texto. En XML se utilizan las etiquetas (tags) para describir bloques de texto. Fue diseñado para compartir fácilmente las estructuras de datos a través de la red.

SOAP

Simple Object Access Protocol es un protocolo estándar bajo el auspicio de la W3C que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML

Tecnologías – II

WSDL

Web Service Description Language es un lenguaje basado en XML utilizado para describir los Servicios Web. Este lenguaje define a los servicios como colecciones de puertos, donde cada puerto indica una función del servicio que está siendo descrito. Así, cualquier usuario de un servicio puede leer el WSDL y saber qué funciones se pueden llamar utilizando SOAP.

UDDI

Universal Description, Discovery and Integration es un registro basado en XML para listar servicios en Internet. Está diseñado para ser interrogado con mensajes SOAP y proveer acceso a los WSDL de los servicios que se encuentran listados.

Tecnologías – III

BPEL

Business Process Execution Language es un lenguaje ejecutable de procesamiento de negocios basado en XML. BPEL nació como la combinación de WSFL (IBM) y XLANG (Microsoft) para crear un estándar mundial de ejecución de procesos de negocio. Este lenguaje se utiliza para orquestar la ejecución de un proceso, llamando a los servicios que va necesitando.

Clasificación de los Servicios

Atendiendo a los roles que puede asumir un Servicio Web

- Proveedor de servicios
- Solicitante de servicio
- Intermediario
- Emisor inicial y receptor final

Despliegue de un servicio Web desarrollado

Partes

- Métodos HTTP y arquitecturas REST
- Arquitectura de Java para Ligadura XML
- Java Specification Request (JAX-RS)
- Anotaciones

Métodos HTTP y arquitecturas REST

- El estilo arquitectónico REST
  - Entidades REST: recursos: soporte, identificación y acceso
  - Interfaz de acceso
  - Negociación de contenidos
- Métodos HTTP
  - GET define un acceso de lectura al recurso
  - PUT crea un nuevo recurso
  - DELETE elimina recursos
  - POST actualiza un recurso existente o crea uno nuevo

Métodos HTTP y arquitecturas REST – II

Servicios Web 'RESTful'

- Definición de la URL base para los servicios:
  `UriBuilder.fromUri("http://localhost:8080/com.mio.jersey.first").build();`
- Los tipos MIME soportados:

```
servicio.accept(MediaType.TEXT_XML).get(String.class));
servicio.accept(MediaType.APPLICATION_XML).get(String.class));
servicio.accept(MediaType.APPLICATION_JSON).get(String.class));
```

- Las operaciones de REST que vayan a ser soportadas.

Arquitectura de Java para Ligadura XML

JAXB

- Arquitectura Java para XML
- Conversión de POJOs a la notación XML
- Java “mappings”: POJOs $<--->$ XML con APIs específicas
- Especificación estándar de JABX susceptible de varias implementaciones comerciales

Arquitectura de Java para Ligadura XML –II

Anotaciones de JAXB

`@XmlRootElement(namespace = "espacionombre")`	raíz de un "árbol XML"
`@XmlType(propOrder = "campo2", "campo1",.. )`	orden escritura campos en el XML
`@XmlElement(name = "nuevoNombre")`	El elemento XML que será usado (*)

Nota(*): Sólo necesita ser utilizado si el nombre es diferente del nombre que tiene en JavaBeans

Java Specification Request (JAX-RS)

Java Specification Request (JSR) 311

- JAX-RS: Soporte REST para aplicaciones y servicios
- Utiliza anotaciones para seleccionar la "parte REST" de las aplicaciones de Java
- Implementación–referencia de JSR 311 se llama *Jersey*
  - Sirve para implementar servicios Web RESTful dentro de un contenedor de servlets (Tomcat, por ejmplo)
  - URL de base del servlet: `http://localhost:8080/nombre-a-mostrar/patron-url/camino-para-el-resto-de-la-clase`
- JAX-RS apoya la creación de XML y JSON a través de la Arquitectura Java para ligadura con XML (JAXB).

Anotaciones

Anotaciones más importantes de JAX–RS

- `@PATH(mi_camino)`
- `@POST`
- `@GET`
- `@PUT`
- `@DELETE`
- `@Produces(TiposMedia.TEXT_PLAIN[Más tipos])`

  $\in \{$“*application/xml*”, “*application/json*”, “*application_plain*”$\}$
- `@Consumes(tipo, [más tipos])`
- `@PathParam`

Composición de servicios

Idea fundamental

Es el proceso de agregar múltiples servicios en uno solo para realizar funciones más complejas.

Formas fundamentales de composición de los SW

- Orquestación
- Coreografía

Orquestación de SW

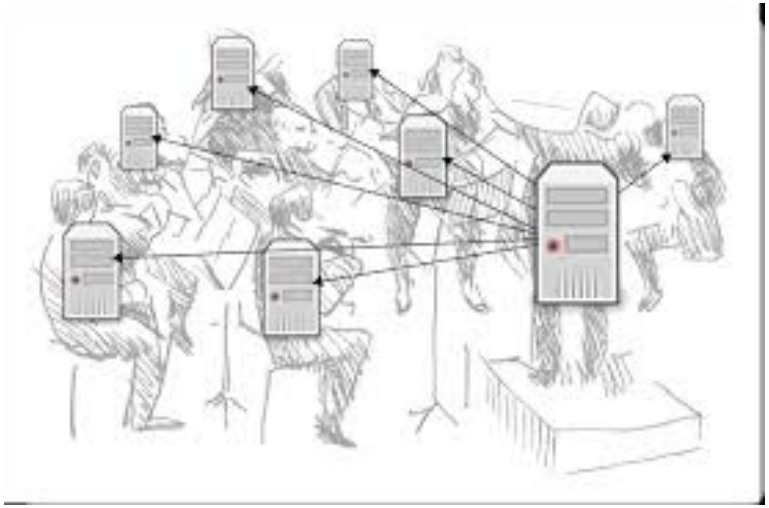


Figura: Orquestación de SW

Concepto

Parte centralizada y bien definida de la *lógica de flujo de trabajos* de un sistema de información de negocios que facilita la *interoperabilidad* entre 2 ó más aplicaciones diferentes.

Orquestación de SW–2

Objetivos

- Hacer posible la fusión de grandes procesos de negocio sin tener que reconstruir las aplicaciones.
- Se consigue la colaboración de las aplicaciones mediante la introducción de una nueva *lógica de flujo de trabajos*.
- Sirve para abstraer de lógica de flujo de trabajos de la solución.
- La orquestación es más importante en entornos de desarrollo orientados a servicios
  - La lógica del proceso de negocio se ha de conseguir expresar sólo mediante servicios.

Orquestación como servicio

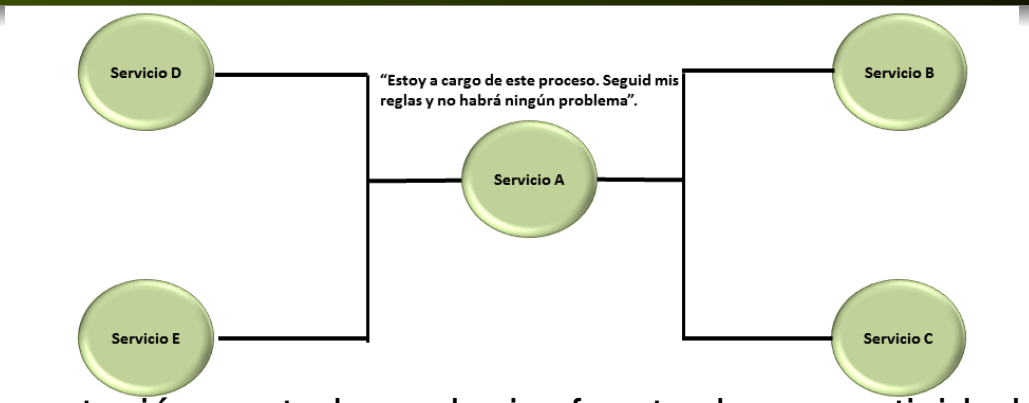


Figura: Una orquestación controla cualquier faceta de una actividad compleja en procesos de negocio.

- Federación de empresas y la orientación a servicios
- El diseño de servicios fomenta la *interoperabilidad*
- Siendo un servicio más, se consigue extender un sistema sin afectar a la interoperabilidad de los componentes
- La lógica incluida en una orquestación permite normalizar la representación de toda una empresa como modelos

Web Services Business Process Execution Language (WS-BPEL)

Características de WS-BPEL/BPEL4WS/BPEL

- Extensión clave para poder obtener WS de segunda generación
- Utiliza conceptos y terminología de WS-* para el modelado de procesos de negocio
- Estándar actualmente vigente: WS-BPEL 2.0 (ver www.oasis-open.org)

Protocolos de negocio y definición de procesos

Características de una orquestación

- La lógica de flujo de trabajos: eventos, condiciones y reglas de negocio
- Se define un protocolo: cómo interactúan los participantes de la orquestación para realizar una tarea de negocio
- La lógica de flujo de trabajos de una orquestación se encuentra contenida dentro de una definición de proceso

Servicios de procesos y entre asociados

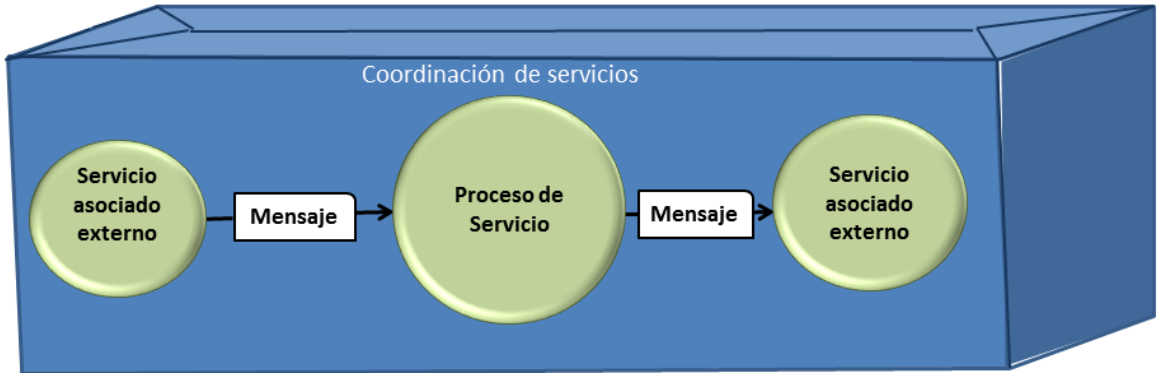


Figura: Un *proceso de servicio* después de ser inicialmente llamado por un servicio asociado.

Semántica de una orquestación

- Los procesos participantes se representan como servicios
- El *proceso de servicios* pueden asociarse y/o ser llamados externamente

Descripción del flujo de trabajos con WS-BPEL

Idea fundamental de la notación BPEL

- Descomponer la lógica de flujo de trabajos en una serie de acciones básicas pre-establecidas
- Identificar acciones fundamentales de flujo de trabajos que se puedan *ensamblar* utilizando la lógica de actividades estructuradas: secuencia, interruptor, repetición, secuencia, etc.
- Es importante conseguir que el orden de ejecución de las actividades no sea ambiguo y esté predefinido

Descripción del flujo de trabajos con WS-BPEL II

Orquestación y sincronización

- Un flujo de trabajos no termina hasta que todas las actividades que contiene han terminado su procesamiento
- Se consigue, por tanto, una lógica de orquestación/ sincronización global basada en los flujos individuales
- El principio de modularidad entre flujos se consigue con la definición de *enlaces*: dependencias formales entre actividades de los flujos
  - Un actividad no puede completarse hasta que se cumplan los requisitos de sus enlaces salientes
  - Antes de comenzar cualquier actividad han de cumplirse los contenidos de los enlaces entrantes

Orquestación y SOA

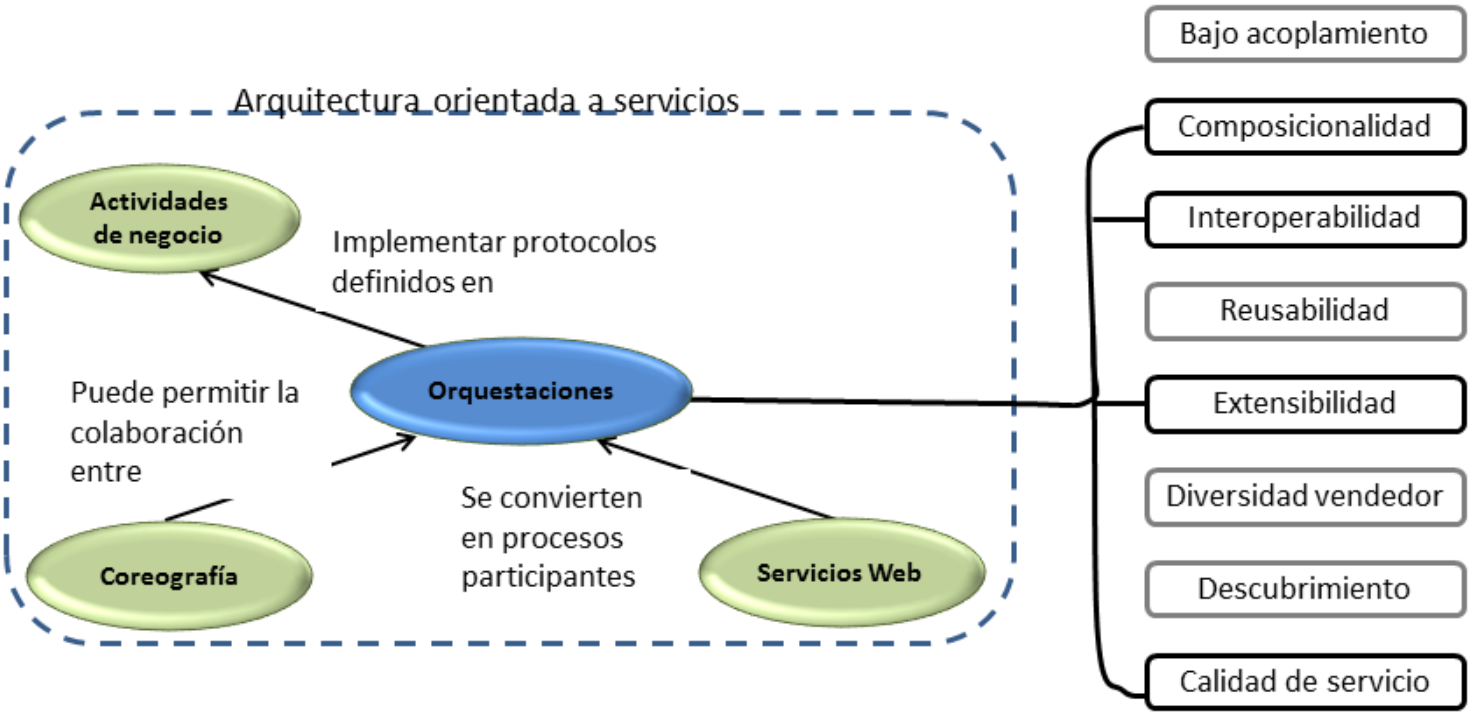


Figura: Orquestación relacionada con otras partes de un SOA.

Orquestación y SOA–2

Principios de automatización de empresas

- La lógica de proceso de negocio está centralizada
- Las orquestaciones propician entornos de aplicaciones orientados a servicios extensibles y adaptativos

Beneficios de utilizar orquestación en SOA

- Modificación centralizada de la lógica de flujo de trabajos
- Facilitar la fusión de procesos de negocio.
- Definición de SOAs a gran escala respalda la evolución hacia una empresa federada.
- Middleware permite integrar motores de orquestación en entornos de aplicaciones orientadas a servicios.

Coreografía de SW

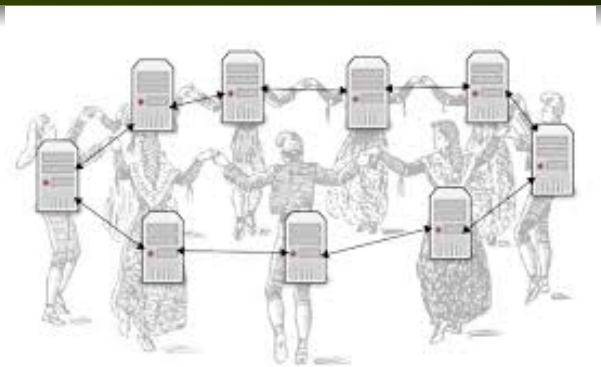


Figura: Coreografía de SW

Concepto

Se trata de una parte bien definida de la *lógica de negocio* que facilita la *interoperabilidad* de servicios cuando la colaboración se extiende más allá de los límites de una organización.
Se pueden ver como patrones de colaboración e interoperabilidad universal para la realización de tareas de negocio comunes a organizaciones.

Coreografías

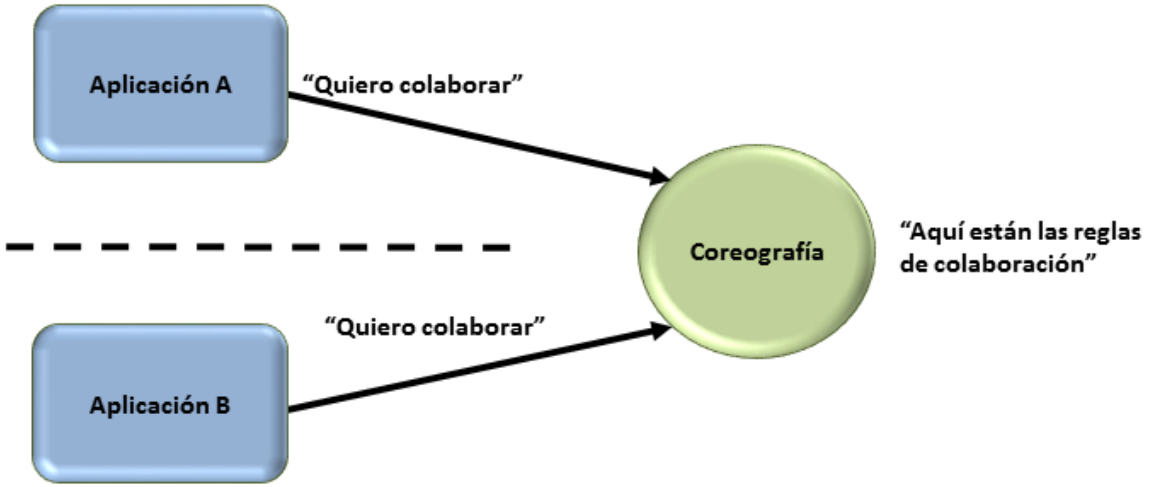


Figura: Facilitan la colaboración entre participantes.

Motivación

Se necesita "coreografiar" múltiples servicios de diferentes organizaciones que necesitan trabajar juntos para lograr un objetivo común en el desempeño de una tarea de negocio.

Coreografía como servicio

Características de una coreografía

- Sirven para intercambio de *mensajes públicos*: colaboración entre SW que pertenecen a diferentes organizaciones.
- A diferencia de las orquestaciones, ninguna organización controla la *lógica de colaboración*.
- Se podrían utilizar también para establecer colaboración entre aplicaciones de una misma organización.
- *Acciones* dentro de una coreografía se estructuran como secuencias de intercambios de mensajes entre SW.

Coreografía como servicio – II

Canales

- Un intercambio de mensajes se define como una relación entre pares de roles, cada uno asumido por un SW.
- Una acción de coreografía viene definida por un conjunto de pares de roles de SW.
- Un canal define la representación del intercambio de mensajes entre los miembros de un par de roles.
- La información sobre un canal se puede pasar en un mensaje para conseguir *descubrimiento de servicios*.
- Las *unidades de trabajo* de una coreografía incluyen interacciones entre SW a las que imponen condiciones para conseguir completarse con éxito.

Descripción de coreografías con WS-CDL

Web Services Choreography Description Language

- Lenguaje basado en XML que describe la colaboración entre pares mediante la definición comportamientos comunes y observables de cada participante.
- La especificación del lenguaje se debe a W3C (W3C Web Services Choreography Working Group, cerrada en julio de 2009), ver: http://www.w3.org/TR/ws-cdl-10/
- Actualmente existen solapamientos con el lenguaje WS-BPEL para orquestaciones.
- La especificación del lenguaje fomenta el descubrimiento dinámico de servicios y propicia la colaboración entre muchos participantes a gran escala.

Reusabilidad, composicionalidad y modularidad

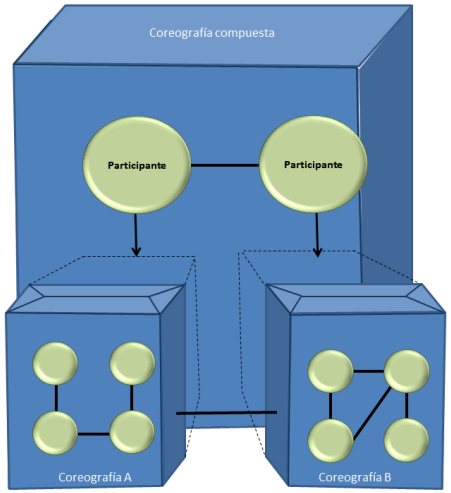


Figura: Una coreografía compuesta por 2 coreografías más pequeñas.

- Se diseñan de forma *reutilizable* para ser aplicadas.
- Ensamblado de coreografías mediante un servicio de importación del marco de trabajo.
- Se pueden estructurar en módulos que incluyen tareas para ser usados por una jerarquía de coreografías.

Coreografía y SOA

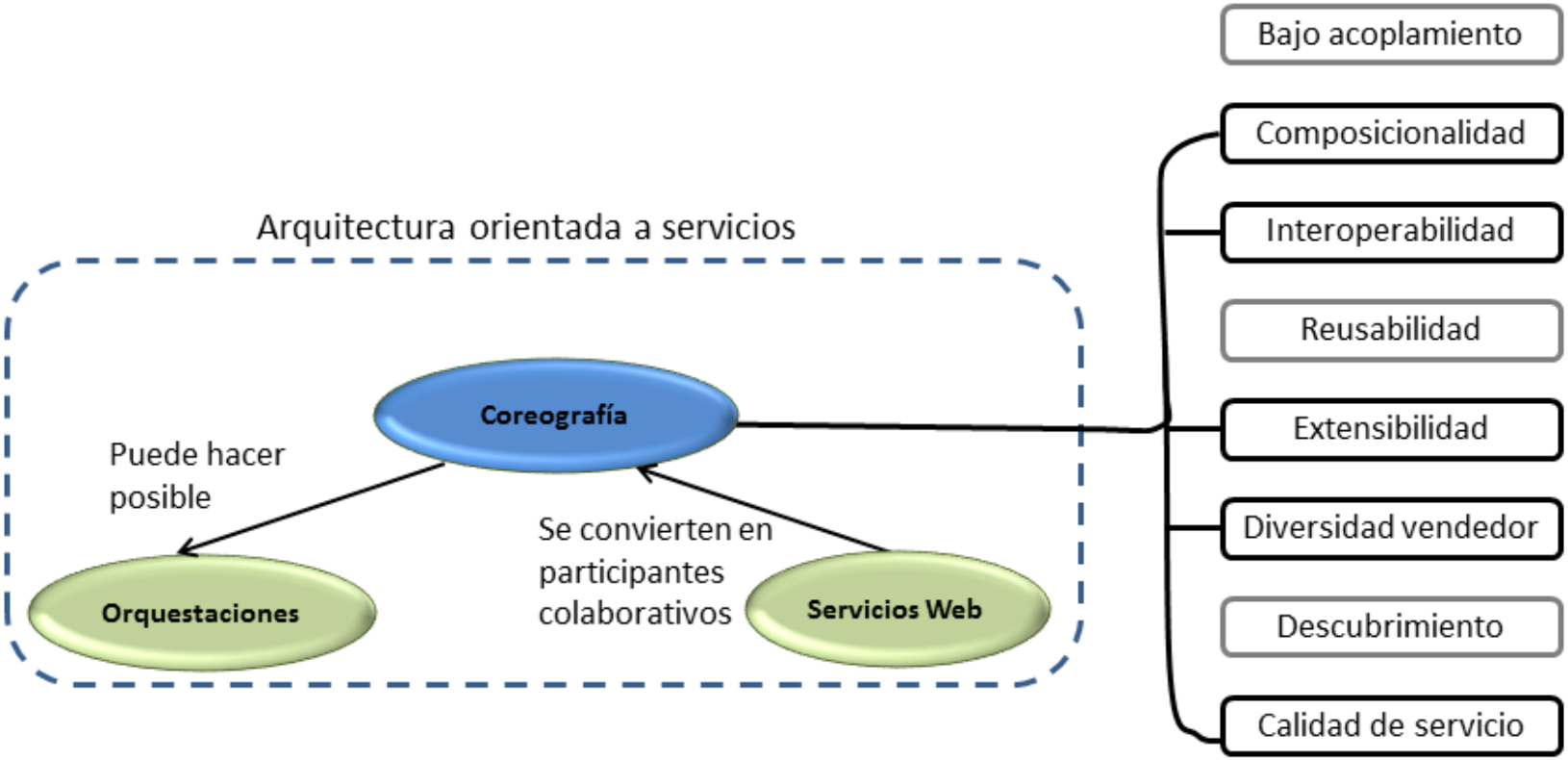


Figura: Coreografía relacionada con otras partes de un SOA.

Coreografías y SOA

Conceptos fundamentales

- 2 SW pertenecientes a varias organizaciones usan una coreografía para conseguir realizar tareas más complejas.
- Permiten ampliar o modificar dinámicamente los procesos de negocio participantes.
- Ayudan a configurar SOA complejos que transcienden las fronteras de una organización.
- Respaldan la composicionalidad, reutilización y extensibilidad de SW.
- Descubrimiento de SW y el diseño y desempeño ágil de tareas de negocio dentro de una organización.

Orquestación vs. coreografía

Coincidencias y diferencias

- Una orquestación se podría entender como la aplicación específica de una coreografía a un determinado negocio
- Tanto las orquestaciones como las coreografías se utilizan para patrones complejos de intercambio de mensajes
- Un orquestación normalmente representa el flujo de trabajos de una organización:
  - La organización posee y controla la lógica de colaboración
- La lógica de control de una coreografía no suele ser propiedad de una sola organización:
  - el patrón de intercambio que se propone se utiliza para la colaboración entre SW de distintas organizaciones

Orquestación vs. coreografía–2

Ventajas

- Son menos complejas que las coreografías
- Actualmente existen herramientas software (BPEL/ODE) industriales que las soportan

Inconvenientes

- La lógica de las orquestaciones suele ser *propietaria*
- No son tan *colaborativas* como las coreografías
- El lenguaje estándar para coreografías está parado de desde 2005
- No se conocen implementaciones de interés industrial de herramientas software para WS-CDL

Historia de BPEL4WS y WS_BPEL

- Antecedentes: Web Services Flow Language (WSFL) de IBM, especificación XLANG de Microsoft
- Especificación de BPEL4WS 1.0 (Julio, 2002), promovido por IBM, Microsoft y BEA Systems
- BPEL4WS 1.1 (Mayo, 2003) con SAP y Siebel Systems
- Aparición de *motores de orquestación* conformes con BPEL4WS
- Submisión al Comité Técnico de OASIS
- Estándar abierto y oficial de OASIS, que le da un nuevo nombre: WS-BPEL 2.0
- Especificación: `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html` (Abril, 2007)

Bibliografía I


Bass, L., Clements, P., and Kazman, R. (2012).
Software Architecture In Practice.
Third edition. Addison–Wesley, Boston, Masssachussets.


Booch, G. (2008).
Handbook of Software Architecture.
http://www.booch.com/systems.jsp.


Clements, P. and et al. (2010).
Documenting Software Architectures: Views and Beyond, volume I.
Addison-Wesley, Boston, Massachussets.


Eden, A. and Kazman, R. (2003).
Architecture Design Implementation:On the Distinction Between Architectural Design and Detailed Design, volume I.


Jansen, A. and Bosch, J. (1999).
Software architecture as a set of architectural design decisions.
In 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). John Wiley.


M. Shaw, D. G. (1996).
Software architecture: perspectives on an emerging discipline.
Prentice-Hall.