Sistemas Distribuidos e Internet

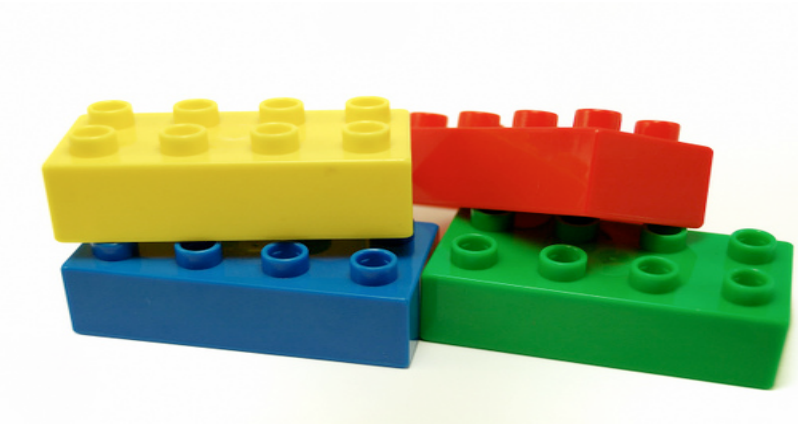
Tema 1

Introducción a Patrones para la Web

Índice

- Introducción
- MVC
- Capas
- Fachada
- Factoría
- DAO y DTO

Introducción



Df. de Patrón y tipos

- Df. Un Patrón es la repetición de las mejores prácticas de lo que funciona en cualquier dominio
- Tipos de patrones:
  - **Arquitectónicos**: Relacionados con el diseño a gran escala y de granularidad gruesa. Ejemplo: El patrón Capas.
  - **Diseño**: Relacionados con el diseño de objetos y *frameworks* de pequeña y mediana escala. Ejemplo: El patrón *Fachada.*
  - **Estilos**: Soluciones de diseño de bajo nivel orientadas a la implementación o al lenguaje. Ejemplo: El patrón *Singleton.*

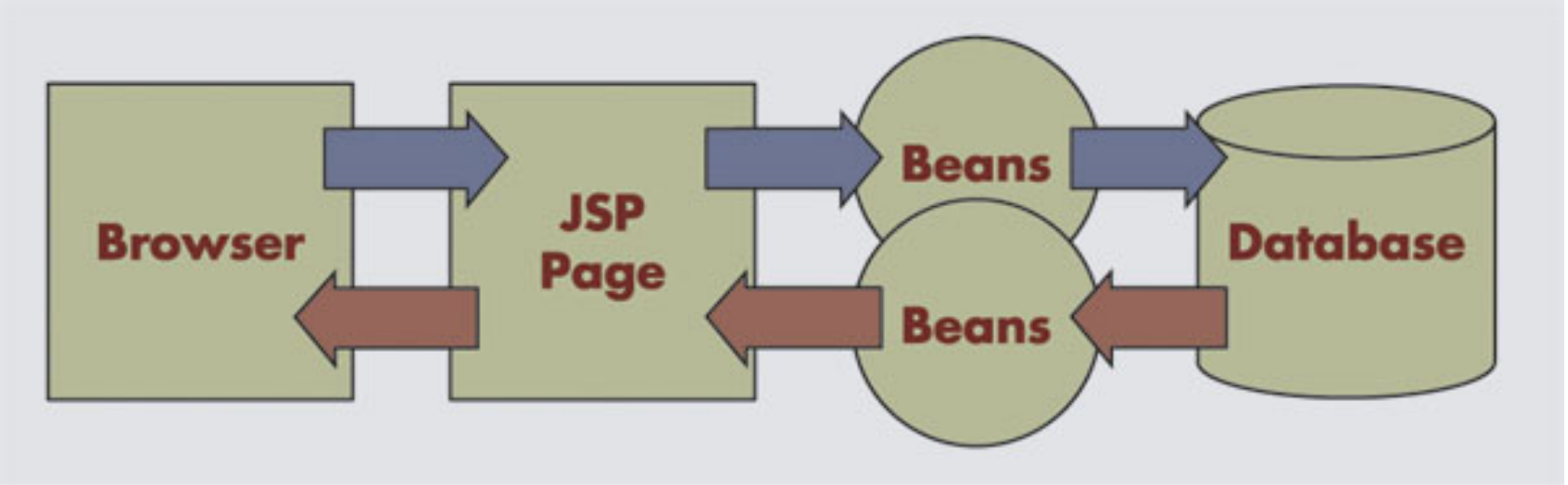
Patrones arquitectónicos y de diseño

- En Pattern of Enterprise Application Architecture [Fowler03] los Patrones arquitectónicos se clasifican en:
  - Domain Logic Pattern.
  - Mapping to Relational Databases.
  - Web Presentation Patterns: MVC, Page Controller..
  - Session State Patterns
- Patrones de diseño. Un patrón de arquitectura puede contener múltiples patrones de diseño [GOF94]. Por ejemplo en una arquitectura MVC se suelen emplear los siguientes patrones de diseño:
  - Creacionales (Factory, Protoype, …)
  - Estructurales (Facade, Adapter, …)
  - Comportamiento (Command, Interpreter, …)

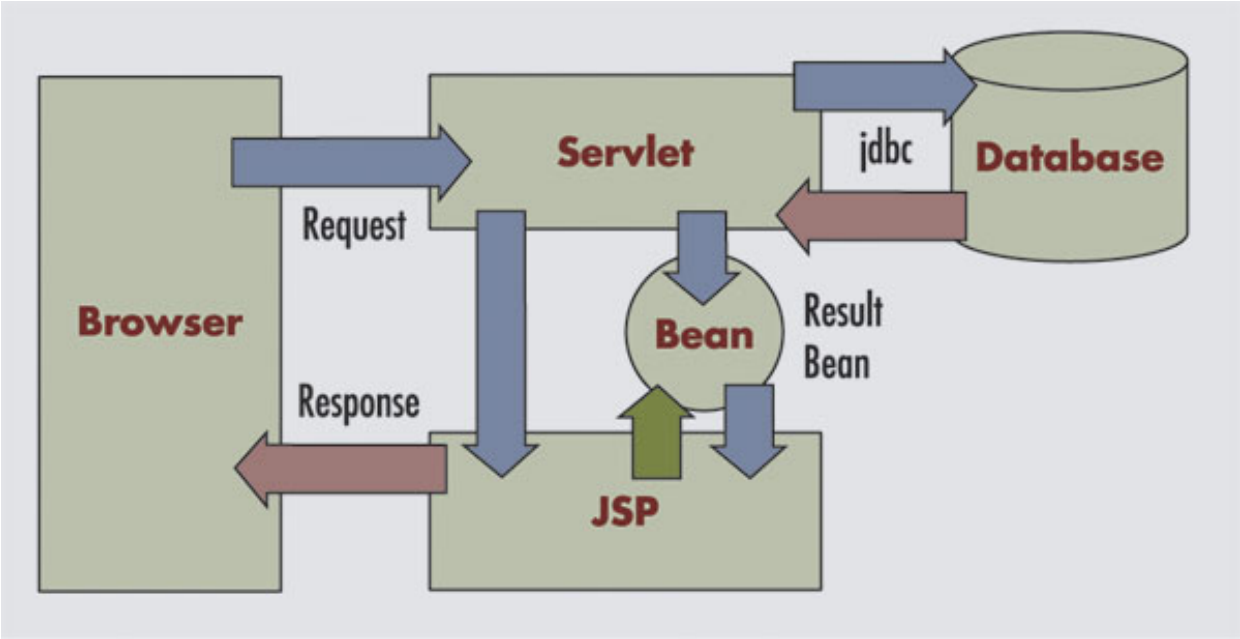
Modelos de desarrollo de aplicaciones Web en JEE (Servlets y JSPs)

- Los servlets son buenos ejecutando lógica de negocio, pero no son tan buenos presentando información
- JSPs son muy buenos presentando pero pésimos introduciendo lógica programática en ellos
- La combinación Servlet/JSPs era lo más común en el desarrollo de aplicaciones web antes de la aparición de los frameworks MVC (Structs, JSF o Spring)
- Dos arquitecturas:
  - Model-1.5: JSPs para presentación y control y JavaBeans para la lógica
  - Model-2: Model-View-Controller = JavaBeans-JSPs-Servlets
    - MVC es tan común que se han desarrollado varias infraestructuras en torno a este patrón de diseño:
      - Apache Struts
      - Java Server Faces
      - Spring

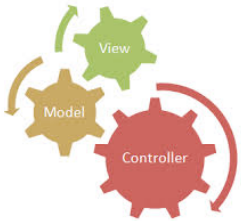
Arquitectura Model 1.5

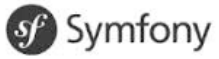


Arquitectura Model 2



MVC

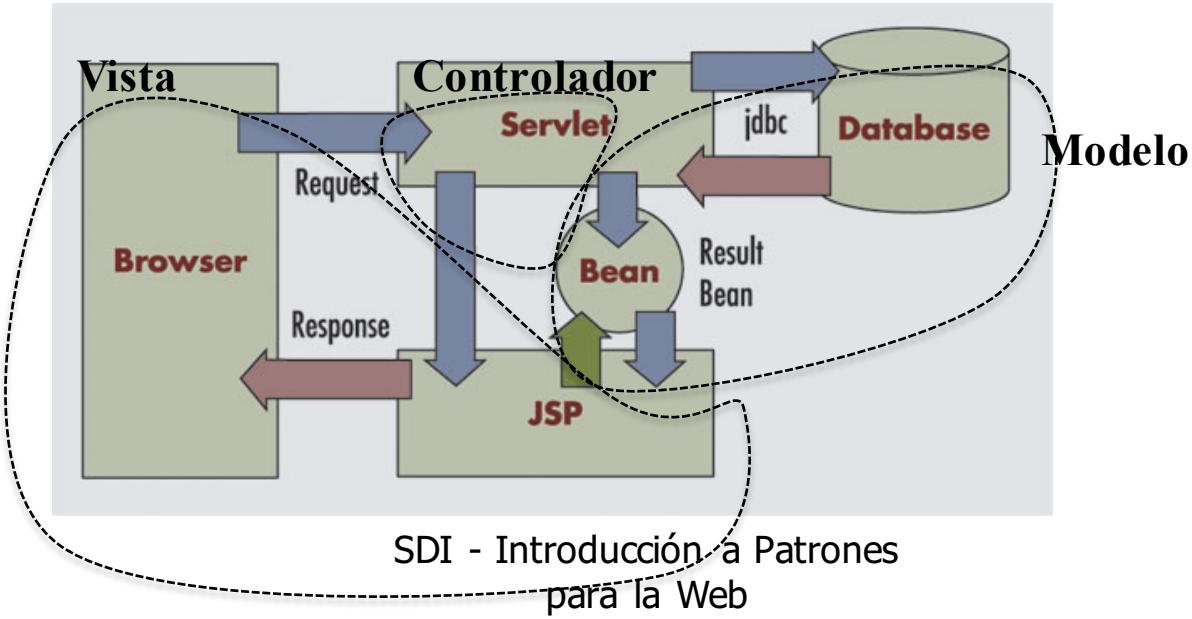




SDI - Introducción a Patrones para la Web

Patrón MVC

- Desarrollado por *Trygve Reenskau* para la plataforma SmallTalk a finales de los 70s.
- Evolución del modelo 1.5 (sin controlador)
- Roles en el patrón Arquitectónico MVC.
  - Controlador: Navegación/Servlet
  - Modelo (Negocio y Datos): Servlet/Beans
  - Presentación: JSPs



MVC: Descripción de roles

- **Modelo**: Representación específica del dominio del problema. Este rol también incluiría la capa de datos. La lógica de dominio añade significado a los datos; por ejemplo, calculando totales, impuestos o portes en un carrito de la compra, no permitiendo stocks negativos, ...
- **Vista**: Presenta el modelo en un formato adecuado para interactuar, usualmente un elemento de interfaz de usuario.
- **Controlador**: Código navegacional. Recibe eventos, usualmente acciones del usuario, invoca al modelo y a la vista.
- Acoplamiento de capas
  - La separación de la Vista y el Modelo es una de las claves del buen diseño de software (Fawler02).
  - Buen look&feel vs políticas de negocio/acceso a datos. Librerías y especialistas muy diferentes.
  - Lograr diferentes vistas para el mismo modelo
  - Los objetos No-visuales son más fáciles de probar.

Modelo MVC I

- El Controlador (Controller)
  - Servlet central recibe peticiones, procesa URL recibida y delega procesamiento a JavaBeans
  - Servlet guarda resultado de procesamiento realizado por JavaBeans en el contexto de la petición, la sesión o la aplicación
  - Servlet transfiere control a un JSP que lleva a cabo la presentación de resultados

Modelo MVC II

- El Modelo (Model)
  - JavaBeans (o EJBs para aplicaciones más escalables) juegan el rol de modelo:
    - Algunos beans ejecutan lógica
    - Otros guardan datos
  - Normalmente:
    1. Servlet controlador invoca un método en bean lógico y éste devuelve un bean de datos
    2. Autor de JSP tiene acceso a bean de datos

Modelo MVC III

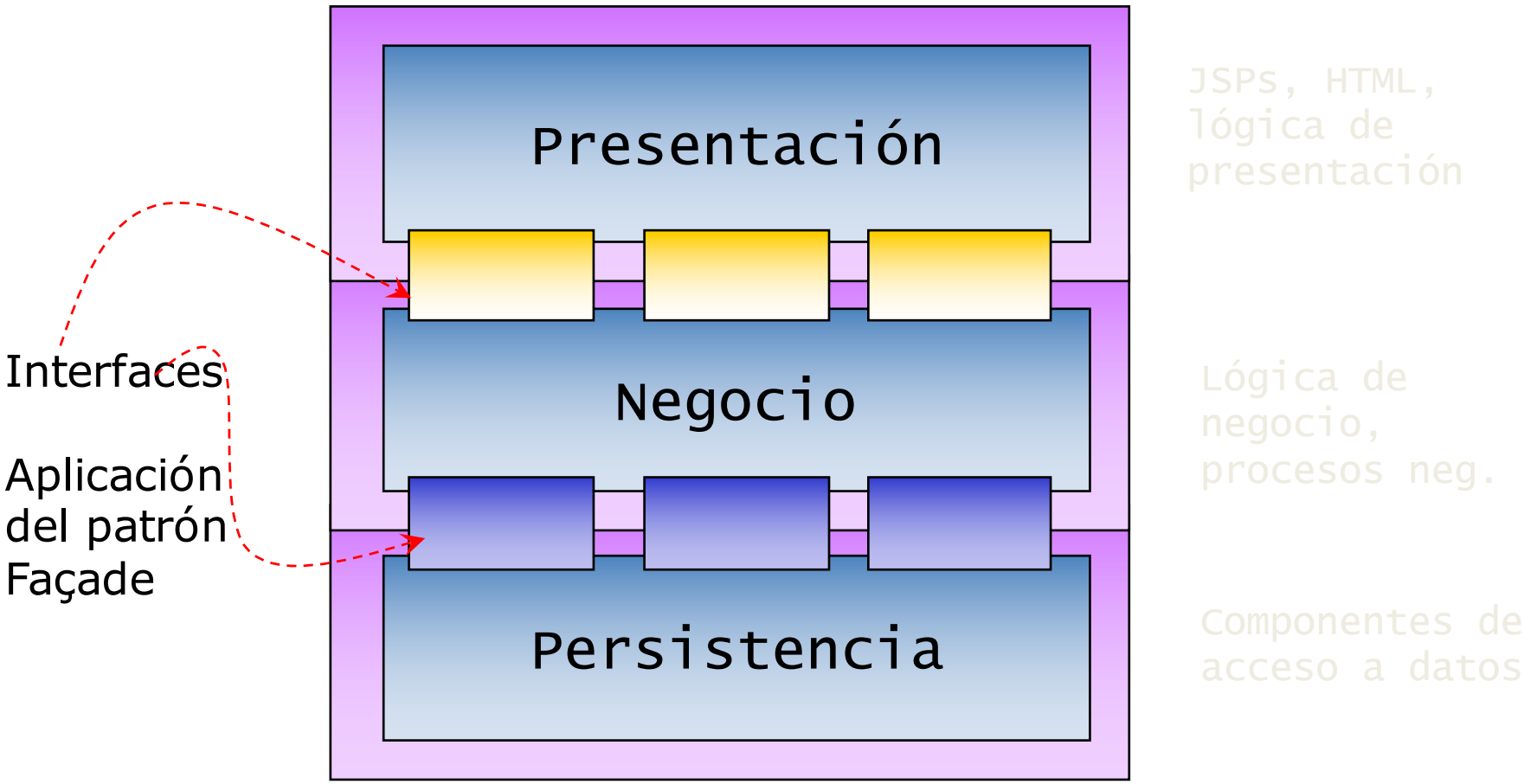
- La Vista (View)
  - Rol ejecutado por JSPs
  - Servlet Controlador transfiere control al JSP después de haber guardado en un contexto el resultado en forma de un bean de datos
  - JSP usa `jsp:useBean` y `jsp:getProperty` para recuperar datos y formatear respuesta en HTML o XML

Modelo MVC IV

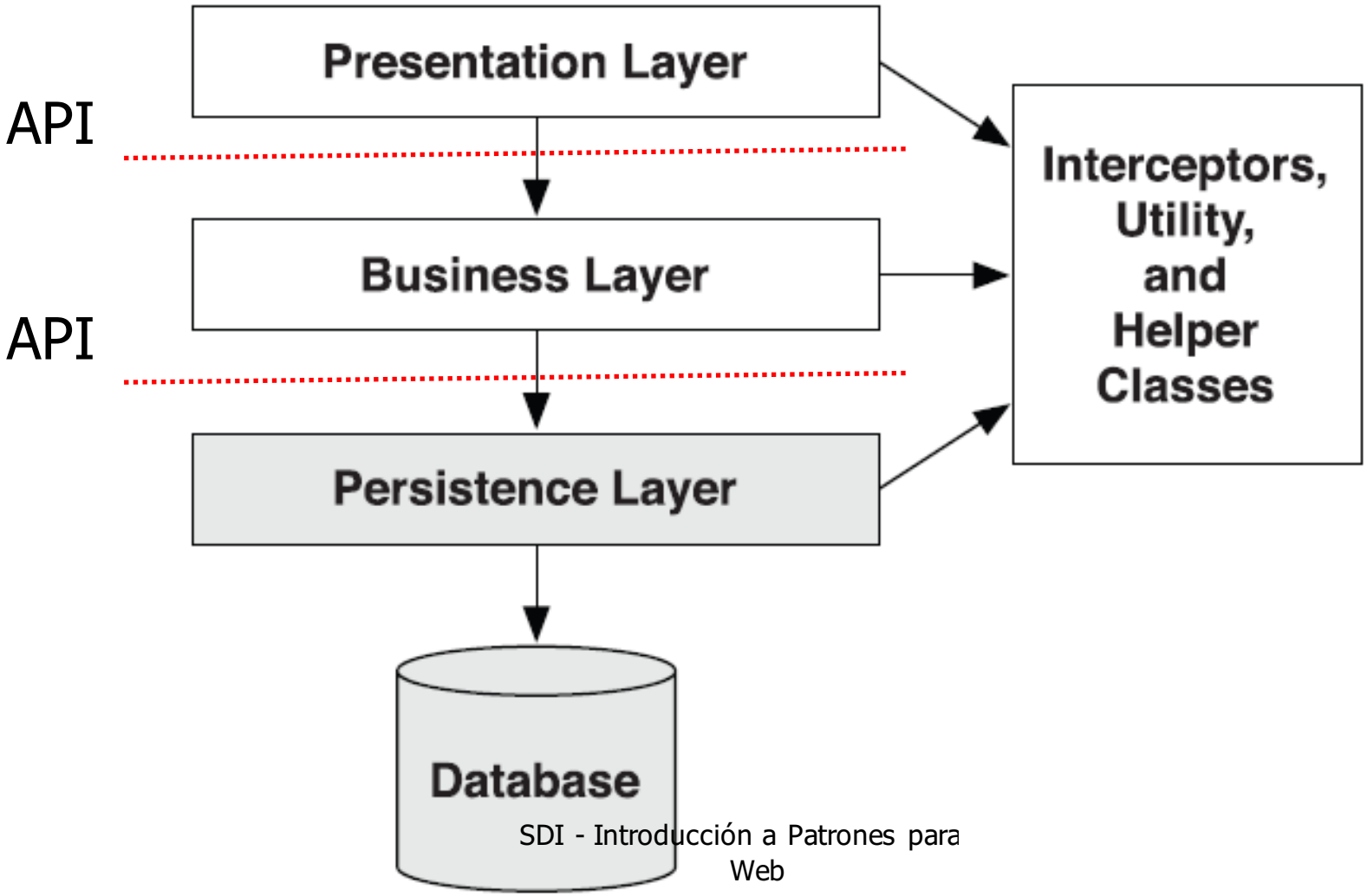
- En resumen:
  - Los beans o EJBs ejecutan la lógica de negocio y guardan los resultados
  - Los JSPs proveen la información formateada
  - Los servlets coordinan/controlan la ejecución de los beans y los JSPs

Patrón N-Capas

Patrón Capas (Modelo de Brown n-capas)

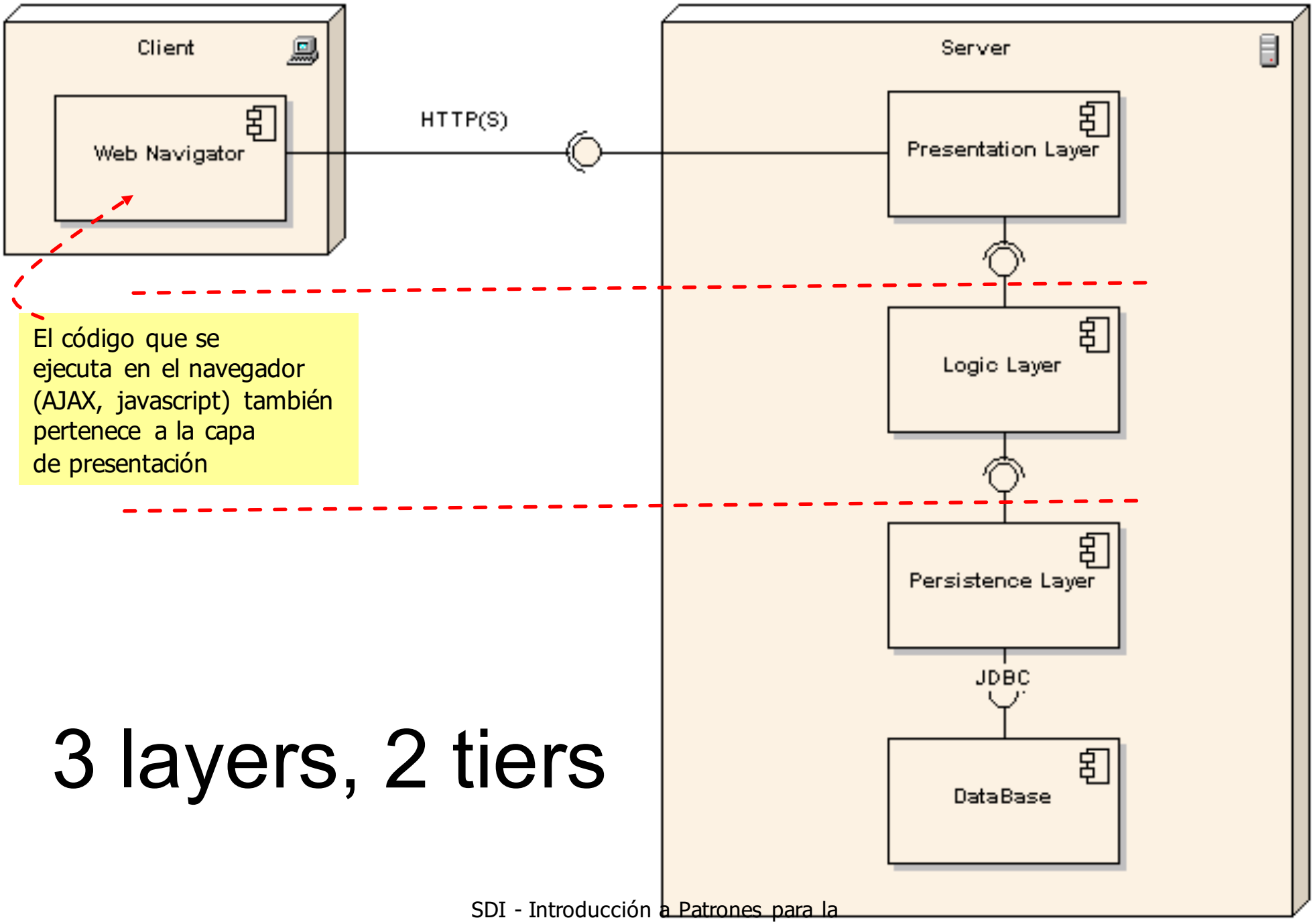


Arquitectura en capas



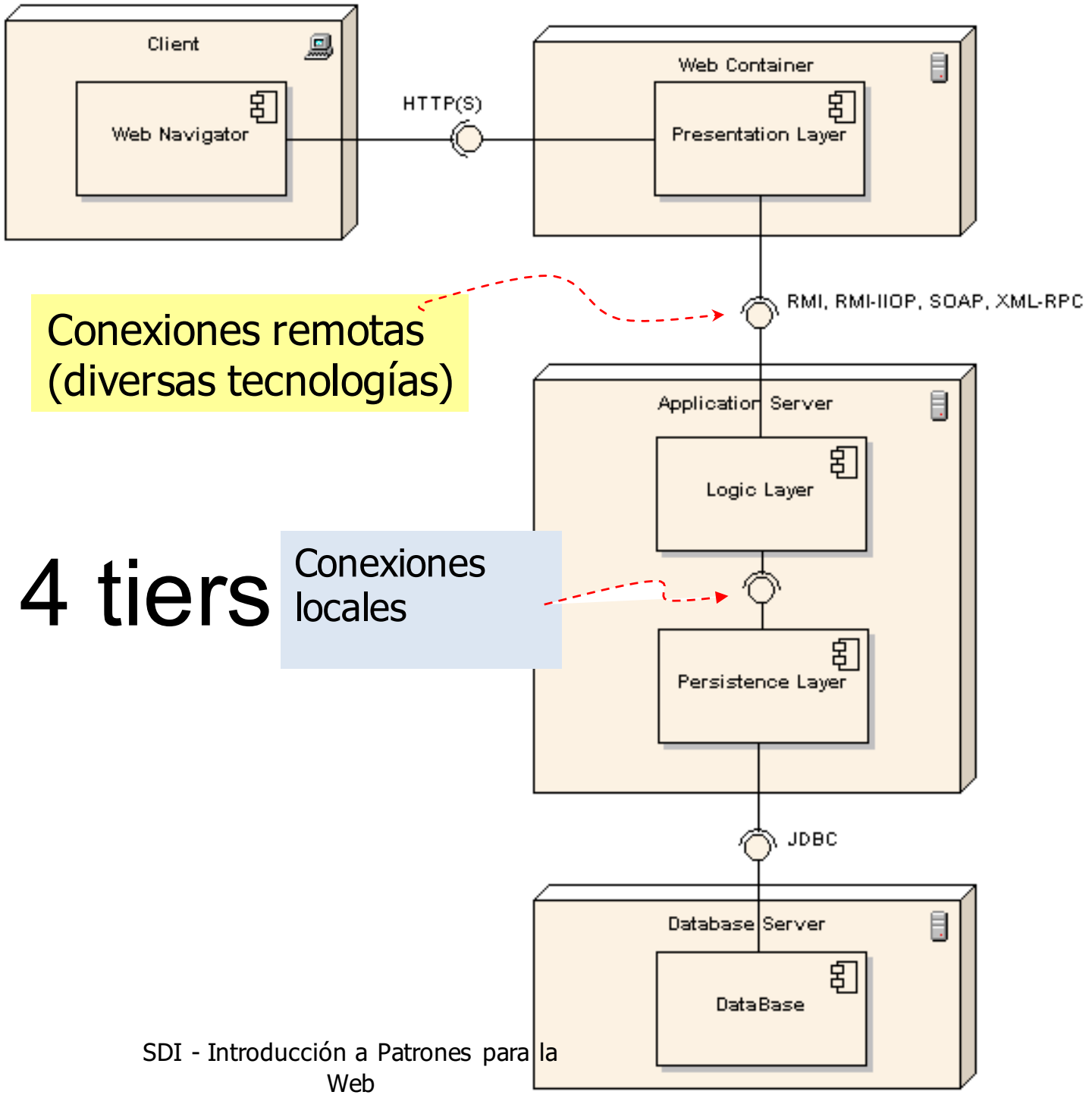
Layers y Tiers

- Layer: capa arquitectónica de la aplicación software
  - Presentación, lógica, persistencia
- Tier: capa física de la arquitectura de despliege del hardware
  - Máquinas: Servidor web, servidor de aplicaciones, servidor de base de datos
- Las "layers" se despliegan sobre las "tiers"



3 layers, 2 tiers

3 layers, 4 tiers

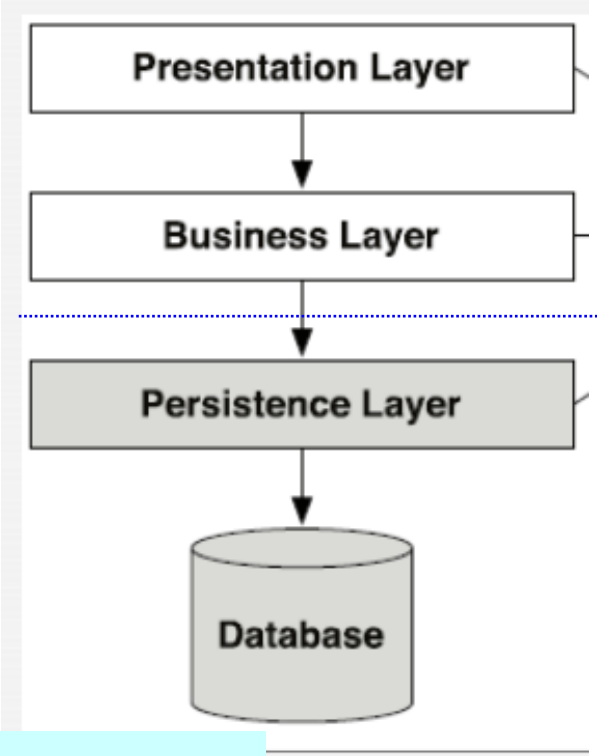


Conexiones remotas (diversas tecnologías)

Conexiones locales

Arquitectura en capas

- Las capas se comunican a través de interfaces
  - Las implementaciones están ocultas al exterior
  - Una factoría sirve una implementación para cada interfaz
  - La capa superior se comunica con la inferior, no al revés
  - Las capas no se pueden saltar



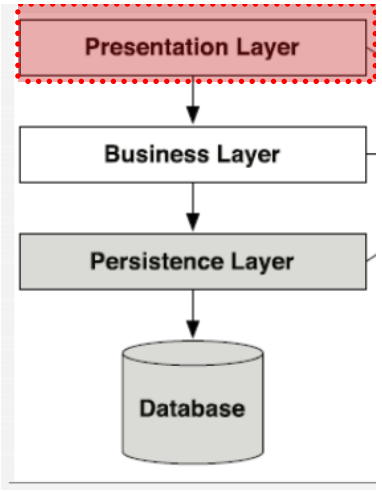
Las capas, hechas así, pueden ser modificadas independientemente

Arquitectura en capas: patrones

Presentación	Negocio	Persistencia
MVC	Fachada Factoría	DAO DTO Factoría Active Record

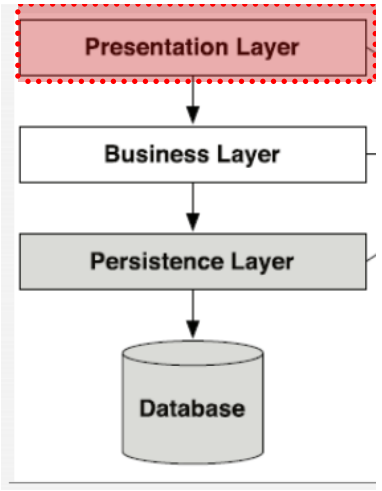
Patrón N-Capas
Capa de Presentación

Capa de presentación



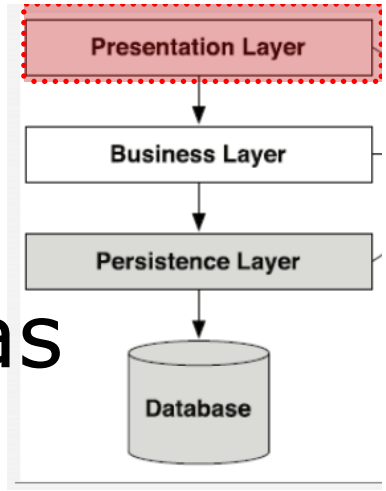
- Resuelve la interacción con el usuario
  - Mostrar datos, formatearlos, ordenarlos
  - Solicitar datos, validarlos
    - Incluye algo de lógica (pero de presentación)
  - Internacionalización (i18N)
  - Informar de los errores lógicos y de ejecución (errores internos)

Capa de presentación



- Controlar la navegación entre pantallas
- Algunas reglas de negocio pueden ser responsabilidad de esta capa
  - Presentar estos datos así y los otros asá…
  - Ocultar/deshabilitar determinado dato/control si se da tal circunstancia…

Capa de presentación



- Puede estar dividida en subcapas
  - Parte en el servidor (p.e. servidor web)
  - Parte en el cliente (p.e. navegador, AJAX)
- Patrones habituales:
  - MVC → Struts Filter/Servlet Faces
  - ServiceLocator o Factory → desacopla la implementación del servicio

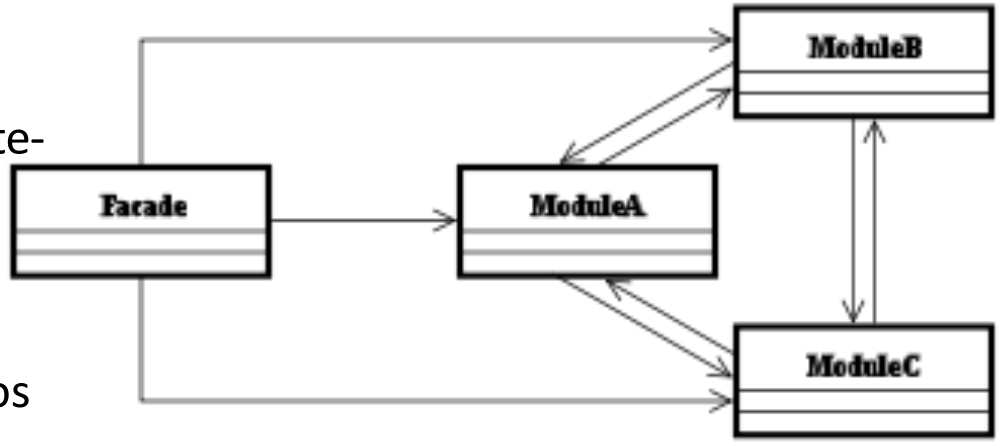
Acceso a lógica desde presentación: ejemplo

```
public class ViewMatesByEnrollmentAction
    extends ActionSupport
    implements ServletRequestAware {

    private HttpServletRequest request;
    private Long enrollmentId;

    public void setServletRequest(HttpServletRequest httpServletRequest) {
        this.request = httpServletRequest;
    }

    public String execute() throws Exception {
        if (request.getSession().getAttribute("user") == null) {
            return LOGIN;
        }
        StudentService ss = ServiceLocator.getStudentService();

        request.setAttribute("matesCollection",
                ss.findMatesByEnrollment(enrollmentId));

        return SUCCESS;
    }
}
```

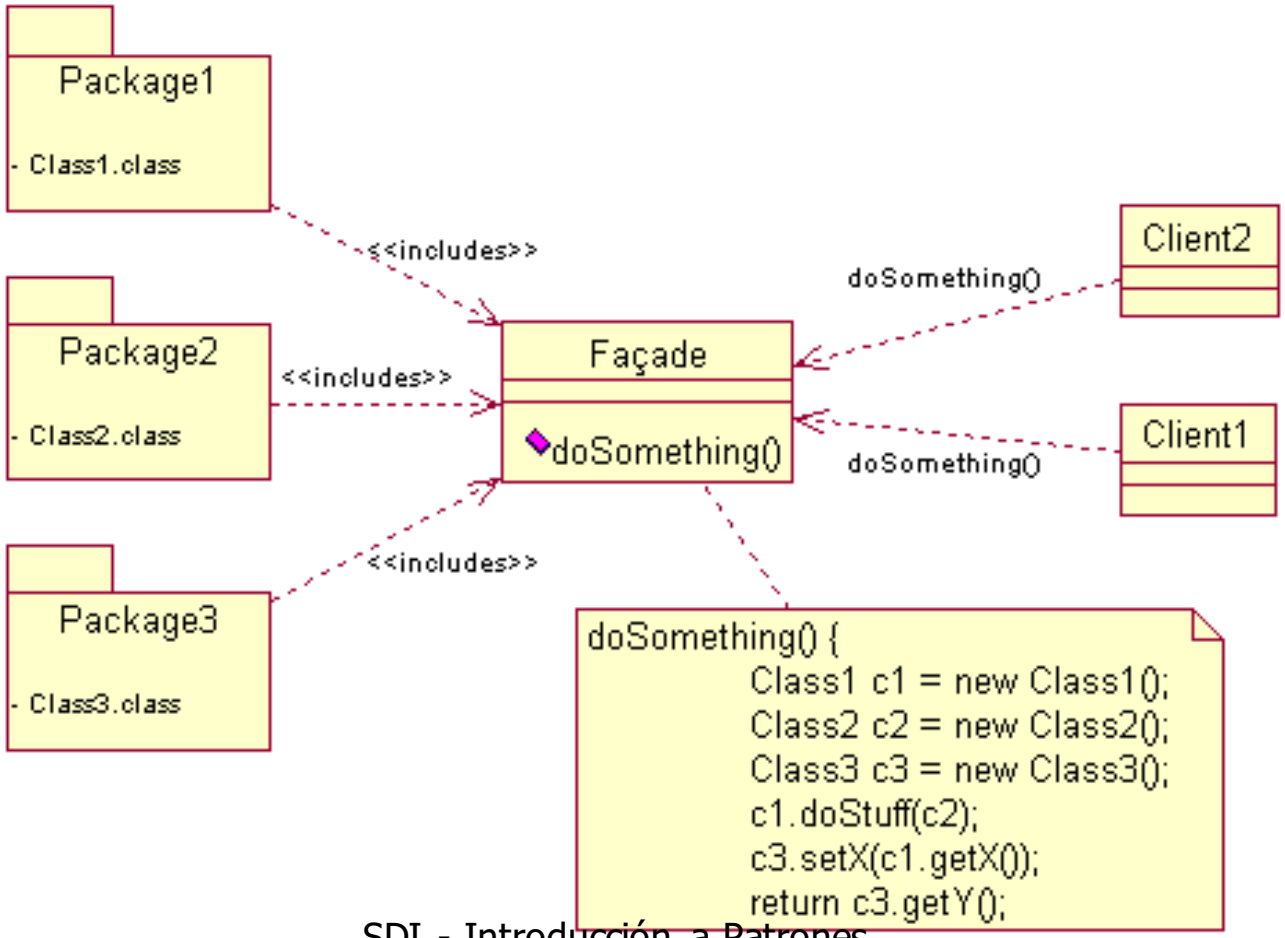
Patrón Fachada

- Df. Interfaz único y simplificado de los servicios más generales de un subsistema
- Cuando se usa:
  - Se busca un interfaz simple para un subsistema complejo
  - Hay muchas dependencias entre clientes y clases que implementan una abstracción
  - Se desea obtener una división en capas de nuestros subsistemas

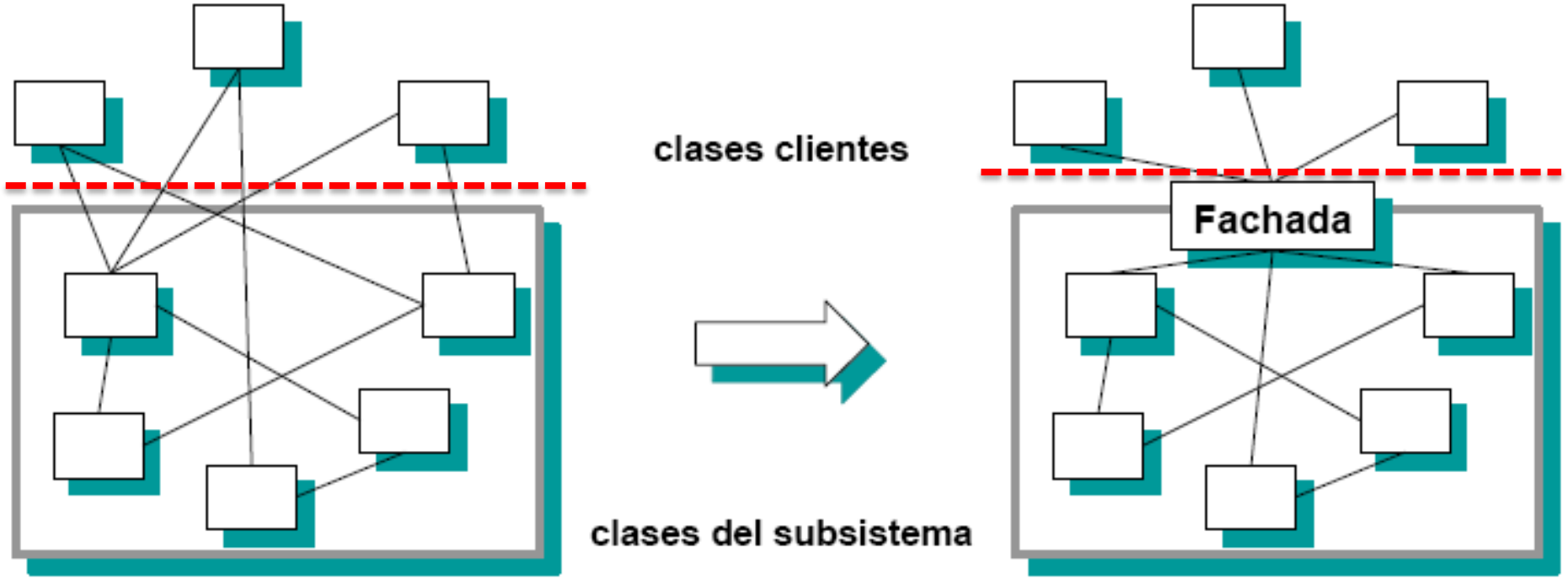
- Como se usa:
  - Reducción del acoplamiento cliente-subsistema (alternativa a la herencia).
  - Clases del subsistema públicas o privadas. No todos los lenguajes los soportan.



Capa de lógica: patrón fachada (*facade*)

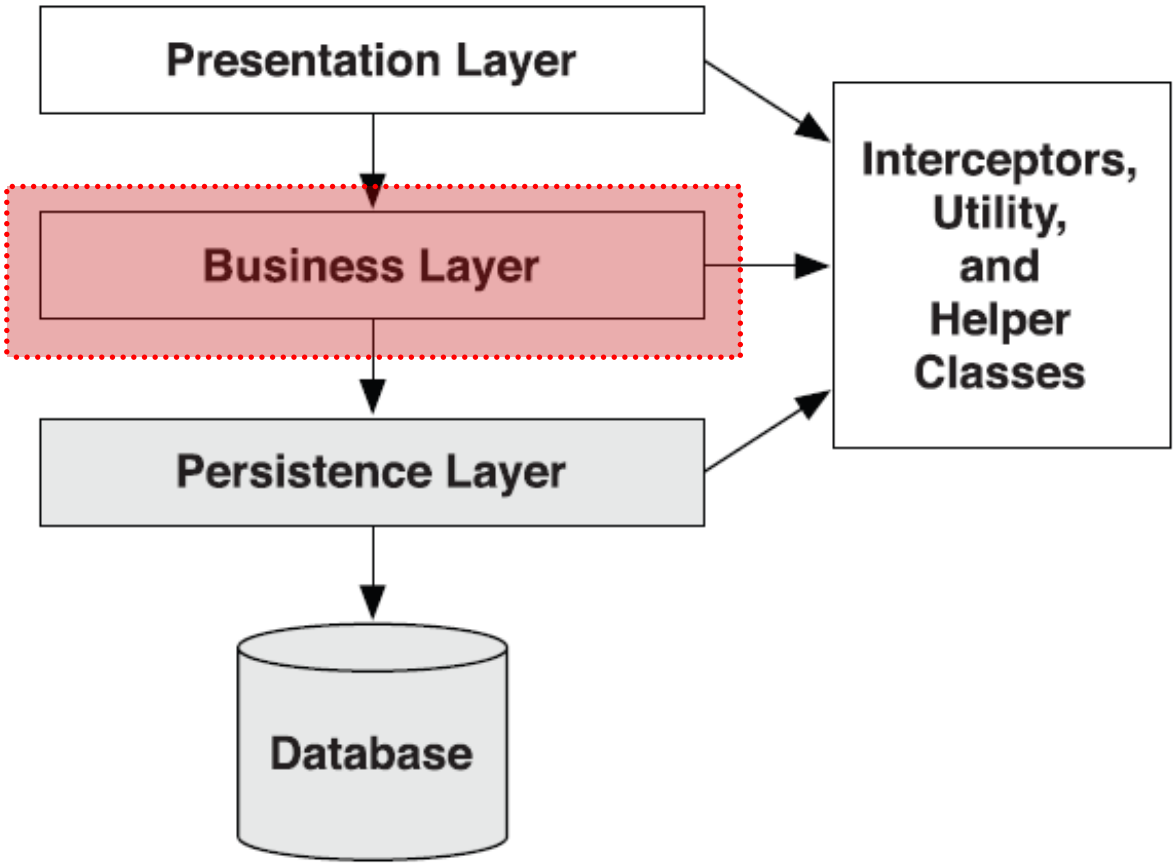


Desacomplamiento de capas



Patrón N-Capas
Capa de Negocio

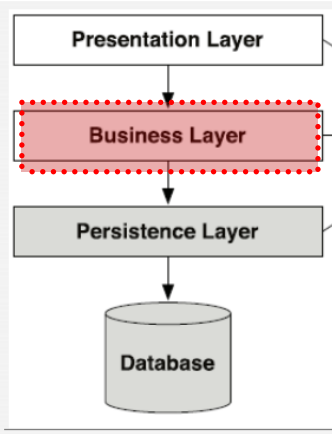
Capa de Negocio



Interfaz de negocio: ejemplo

```
public interface ProfesorServices {
    void calificarAlumno(Long matriculaId, Nota nota) throws BusinessException;
    void calificarCurso(Map<Long, Nota> tablaNotas) throws BusinessException;
    Collection<Nota> findNotasMatricula(Long matriculaId) throws BusinessException;

    Asignatura findAsignaturaById(Long asignaturaId) throws BusinessException;
    Collection<Matricula> findMatriculas(Long asignaturaId) throws BusinessException;
    Collection<Asignatura> findAllAsignaturas() throws BusinessException;
    Matricula findMatriculaById(Long matriculaId) throws BusinessException;
}
```

Capa de negocio: implementación



En esta capa no se debería meter ninguna dependencia de tecnología de infraestructura

- Debería poderse ejecutar fuera de cualquier entorno (para testear)
- La persistencia suele ser la principal dependencia. La capa DAO la evita

Capa de negocio: patrón factoría (*factory*)

- Una factoría es un objeto encargado de la creación de otros objetos
- Utilizados en las ocasiones en las que hacerse con un objeto implica algo más complejo que crearlo
  - Crear la clase del objeto dinámicamente
  - Obtenerlo de un "pool" de objetos
  - Realizar una configuración compleja del mismo
  - Etc.
- El cliente no conoce el tipo concreto del objeto a crear
  - Sólo los conoce a través de su interfaz

Factoría de un servicio: ejemplo 1/3

```
package com.tew.business;

public interface ServicesFactory {

        AlumnosService createAlumnosService();
}

package impl.tew.business;

import com.tew.business.AlumnosService;
import com.tew.business.ServicesFactory;

public class SimpleServicesFactory implements ServicesFactory {

        @Override
        public AlumnosService createAlumnosService() {
                return new SimpleAlumnosService();
        }

}
```

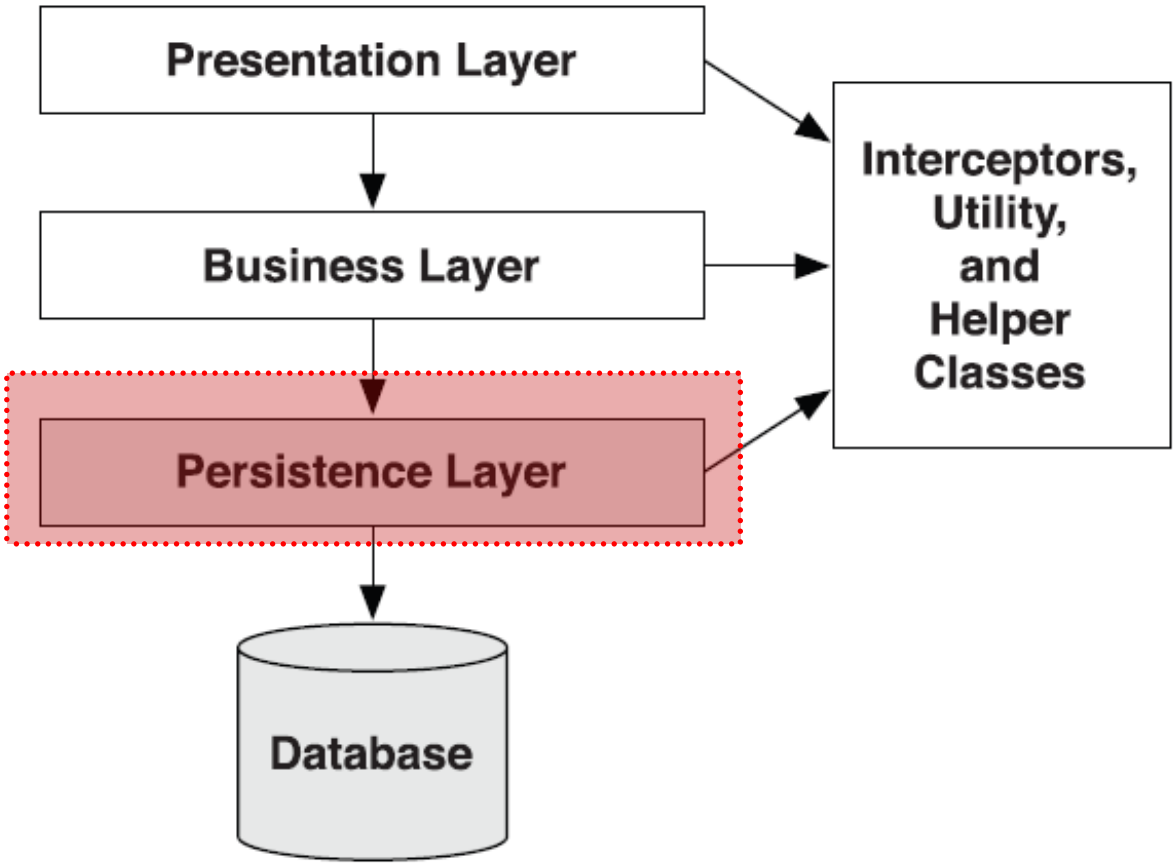
Factoría de un servicio: ejemplo 2/3

```
package impl.sdi.business;

import impl.sdi.business.classes.AlumnosListado;
import impl.sdi.business.classes.AlumnosOtroProcesoDeNegocio;
import java.util.Set;
import impl.sdi.business.AlumnosService;
import impl.sdi.model.Alumno;
/**
 * Clase de implementación (una de las posibles) del interfaz de la fachada de
 * servicios
 */
public class SimpleAlumnosService implements AlumnosService {
        @Override
        public Set<Alumno> getAlumnos() throws Exception {
                return new AlumnosListado().getAlumnos();
        }
        @Override
        public void proceso(Set<Alumno> alumnos) throws Exception {
                new AlumnosOtroProcesoDeNegocio().procesar(alumnos);
        }
}
```

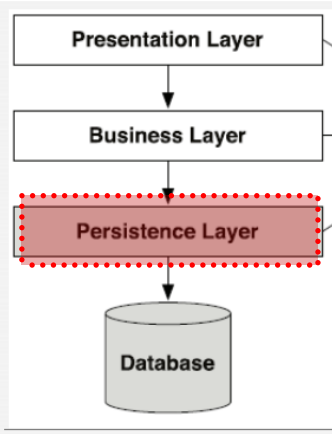
Factoría de un servicio: ejemplo 3/3

```
package impl.sdi.infrastructure;

import impl.sdi.business.SimpleServicesFactory;
import impl.sdi.persistence.SimplePersistenceFactory;

import impl.sdi.business.ServicesFactory;
import impl.sdi.persistence.PersistenceFactory;

/**
 * Esta clase es la que realemente relaciona las interfaces de las capas
 * con sus implementaciones finales. Si se deben hacer cambios de implementación
 * en algunas de las capas habrá que retocar esta clase.
 *
*/
public class Factories {

        public static ServicesFactory services = new SimpleServicesFactory();
        public static PersistenceFactory persistence = new SimplePersistenceFactory(); }
```

Capa de persistencia

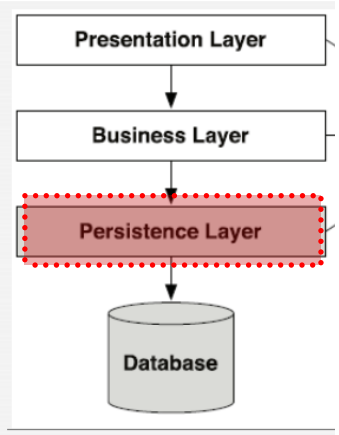


Capa de persistencia



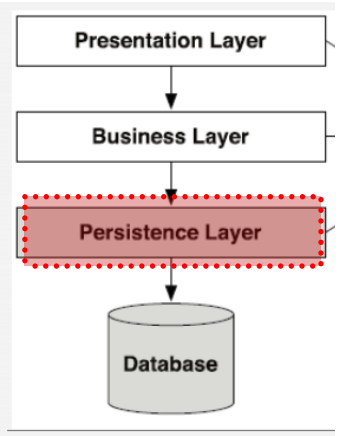
- Ofrece interfaz a la capa superior
- Las distintas implementaciones de la persistencia no deben ser perceptibles por la capa de lógica → independencia
- Uso de patrones DTO/DAO/Active Record
  - Con frecuencia se usa
    - Un DAO para cada entidad del modelo (interface abstracto)
    - Un DTO para cada fila de la entidad o fusión de de diferentes entidades.
    - Active Record es una simplificación de DAO. La funcionalidad CRUD va directamente en el Objeto de Dominio. (más simple pero menos flexible que DAO).
  - Obtenidos a través de una factoría

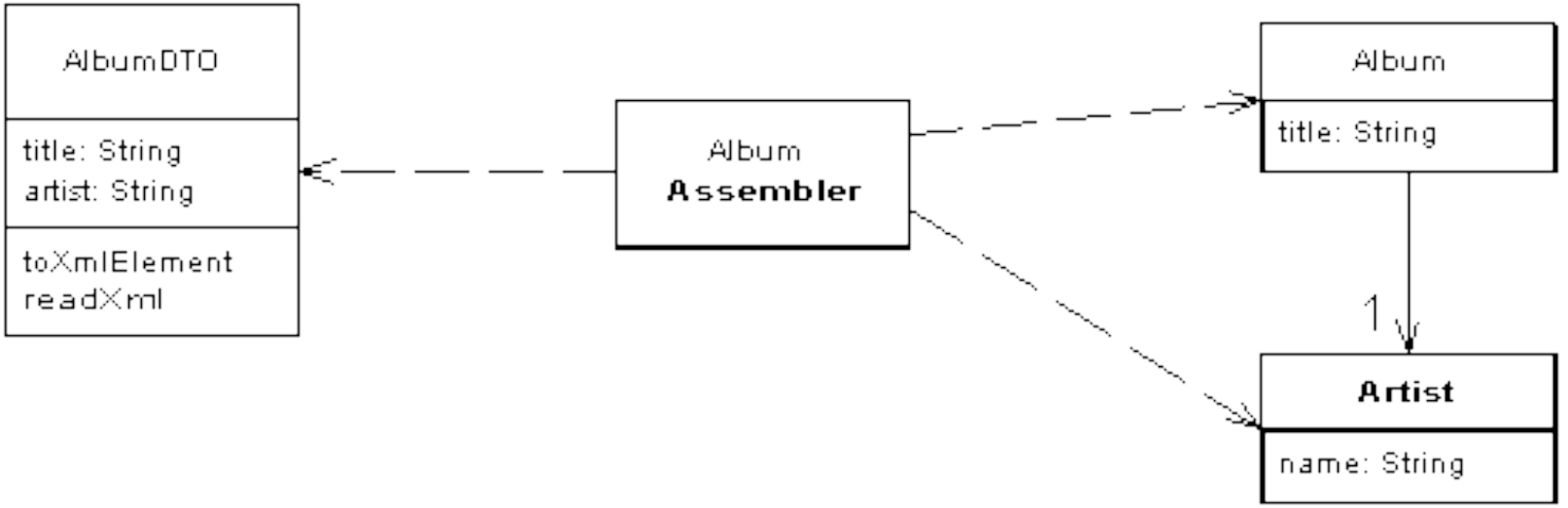
Patrón *Data Transfer Object (DTO*



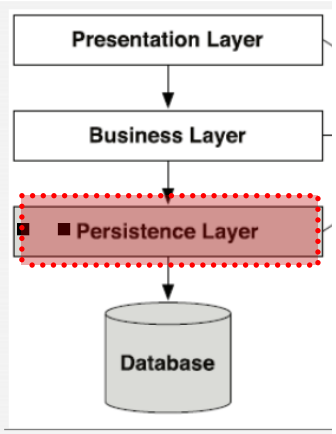
- Utilizado para transferir datos ent subsistemas
  - Para reducir el número de llamadas a método
  - Su único comportamiento viene dado básicamente por *getters* y *setters*
- Se utilizan a menudo en combinación con objetos DAO (persistencia) para obtener datos de una base de datos

Capa de persistencia: patrón *Data Transfer Object (DTO)*



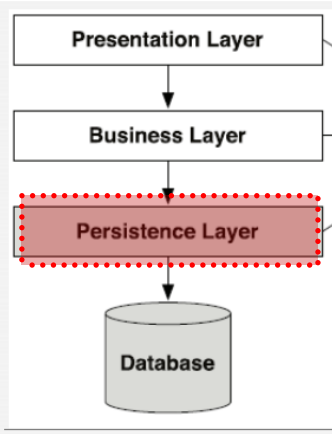


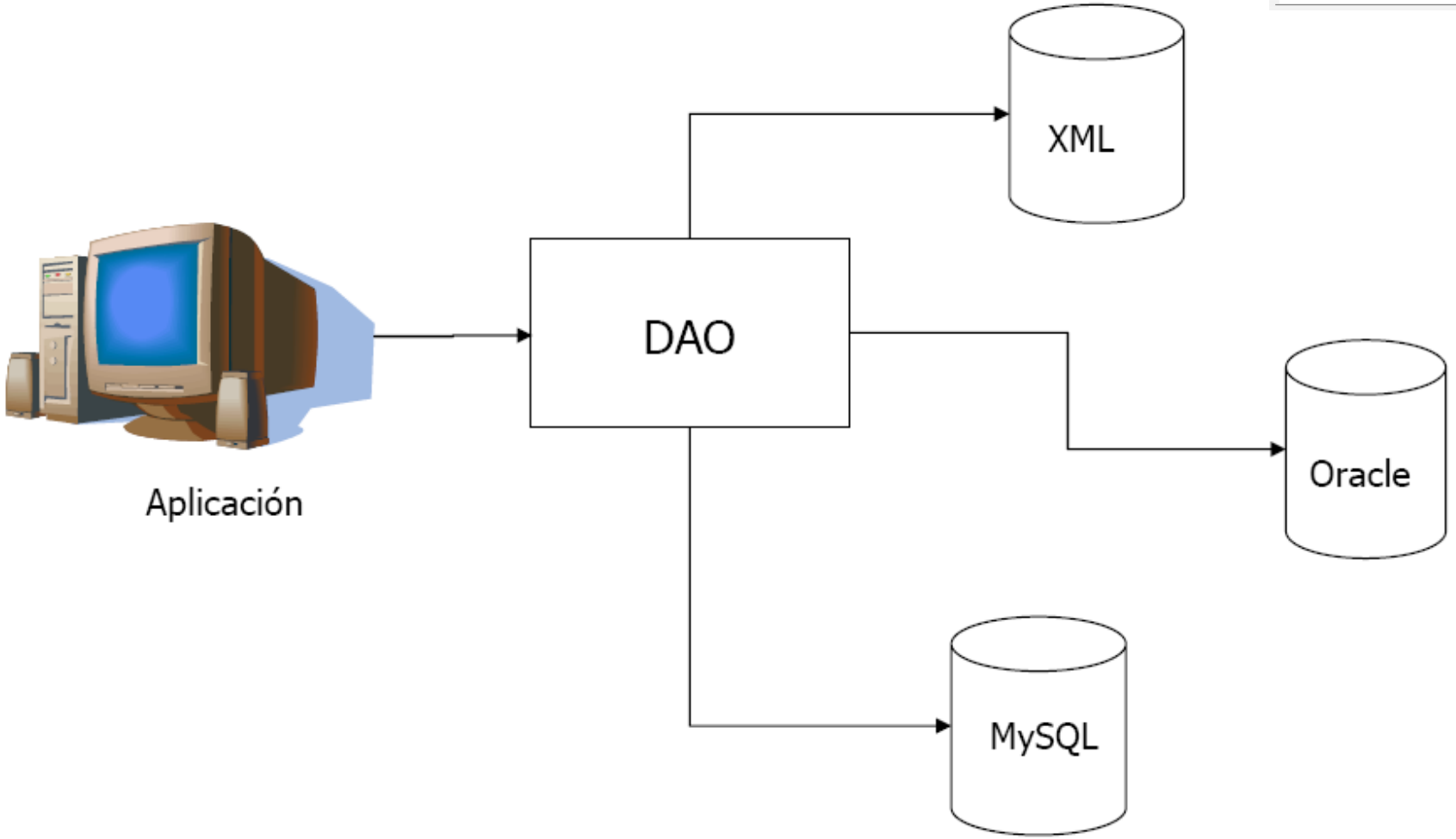
Patrón DAO: problemas si...



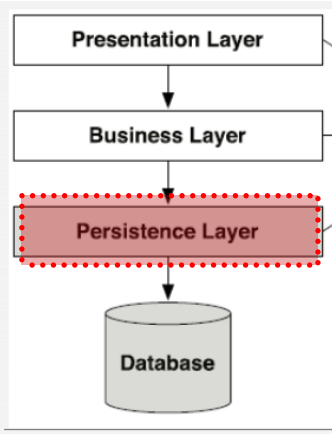
- …Se necesita independencia del sistema de persistencia
  - BDD relacional, BDD orientada a objetos, ficheros, XML, BDD XML, serialización, …
- … Se debe acceder a varios sistemas desde la misma aplicación:
  - Y tienen APIs muy diferentes (o ligeramente)

DAO: solución



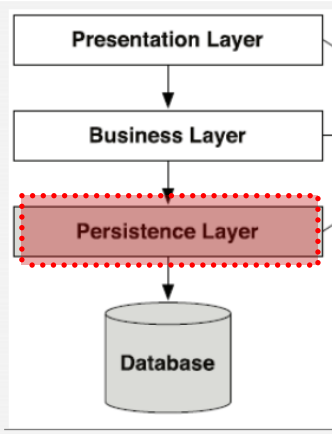


DAO



- DAO →Data Access Object
- DAO proporciona una interfaz única de acceso a los datos, de forma independiente a dónde se hallen almacenados.
- Independiza la lógica de negocio del acceso a los datos.
- Ofrece operaciones CRUD para cada objeto persistente del dominio

Modelo DAO



Clase de la capa de persistencia

BusinessObject

uses

DataAccessObject

encapsulates

DataSource

obtains/modifies

creates/uses

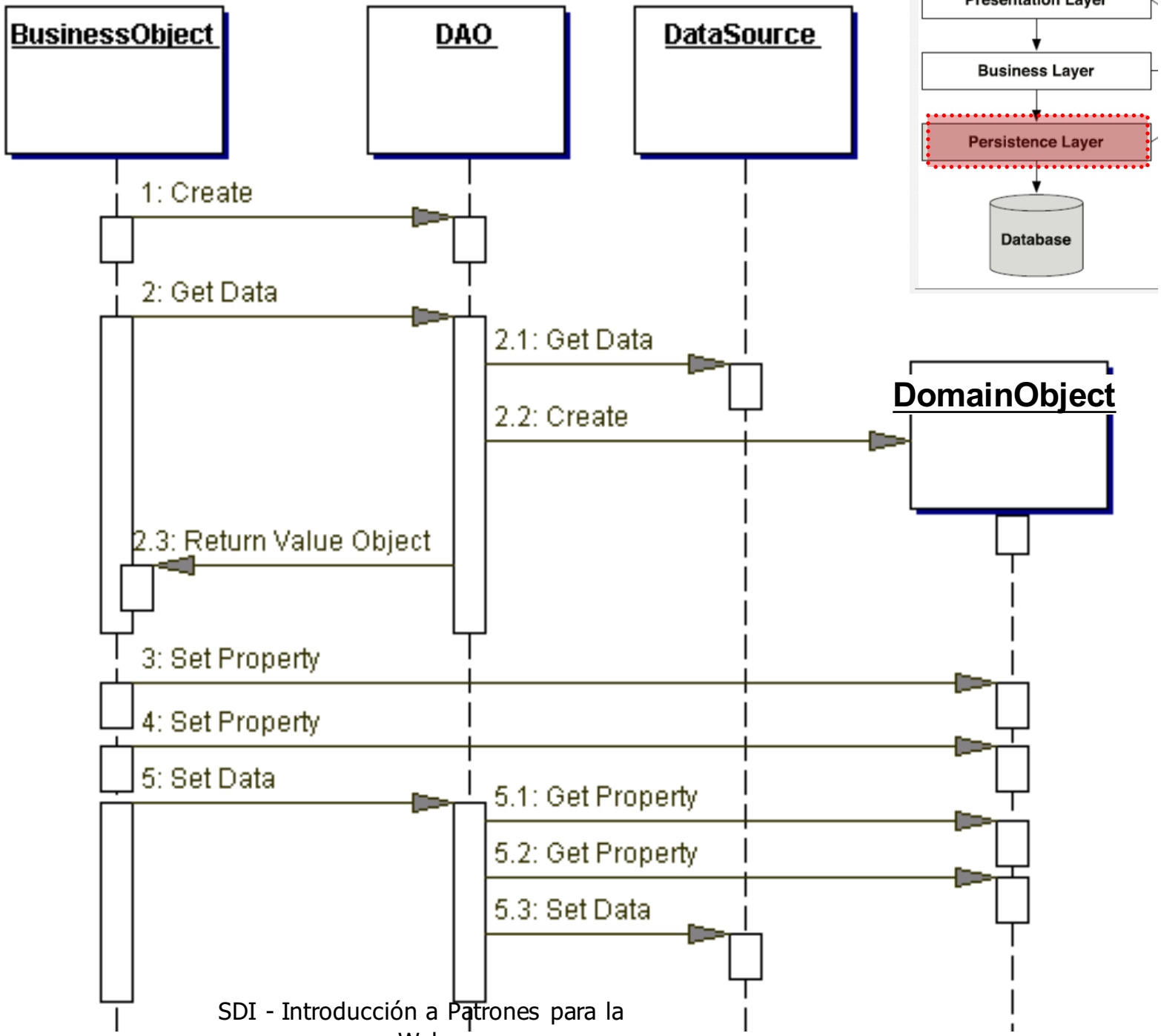
Clase de la capa de negocio

Clase del API de acceso al soporte persistente (ej. JDBC, ADO.NET)

DomainObject

Clase del modelo de dominio

Modelo DAO: interacción



Interfaces DAO: ejemplo

```java
public interface GenericDao<T> {
    void save(T t);
    T update(T t);
    void delete(T t);

    T findById(Long id);
    Collection<T> findAll();
}
```

Métodos CRUD básicos

Métodos CRUD específicos para cada entidad del modelo

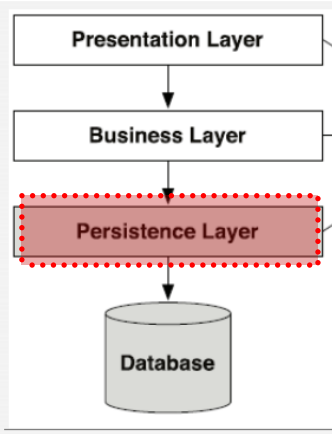
```java
public interface StudentDao extends GenericDao<Student>{
    Collection<Student> findByNameSurname(String name, String surname);
    Collection<Student> findByEnrollmentId(Long enrollmentId);
    Collection<Student> findMatesByNameSurnameCourse(String name,
            String surname, String course);
    Set<Student> findUnenrolledAfterDate(Date date);
}
```

```
public class ManageAuction {
    ItemDAO itemDAO = new ItemDAO();
    PaymentDAO paymentDAO = new PaymentDAO();

    public void endAuction(Item item) {
        // Reattach item
        itemDAO.merge(item);

        // Set winning bid
        Bid winningBid = itemDAO.getMaxBid( item.getId() );
        item.setSuccessfulBid(winningBid);
        item.setBuyer( winningBid.getBidder() );

        // Charge seller
        Payment payment = new Payment(item);
        paymentDAO.persist(payment);

        // Notify seller and winner
        ...
        }
    ...
}
```

Código que resuelve lógica de negocio →

No tiene dependencias de persistencia

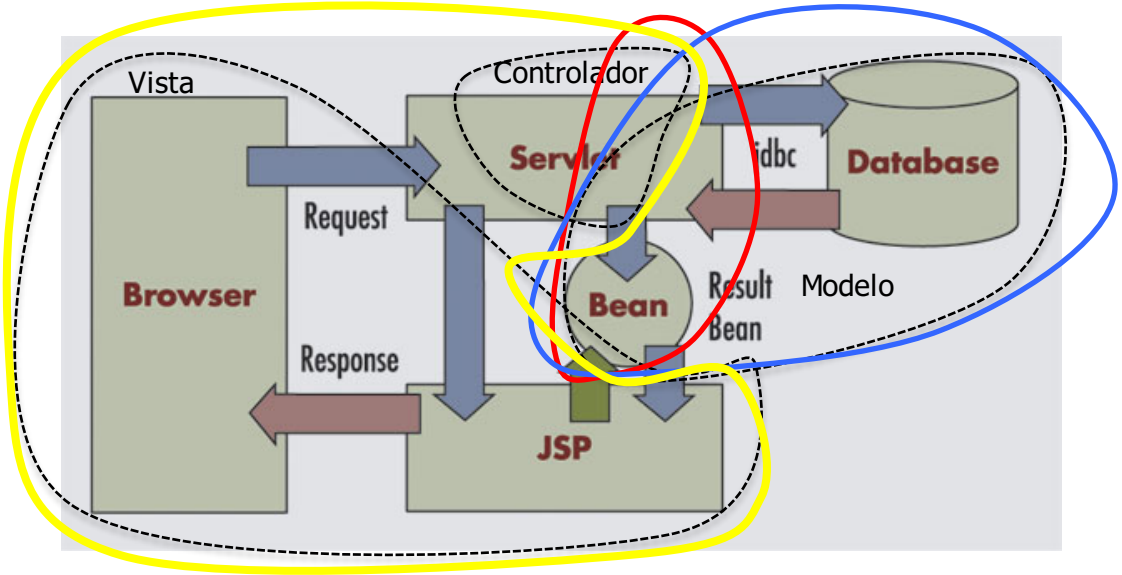
Posibles alternativas de implementación en JEE



- Clases java manejando SQL
  - DAO con JDBC y SQL
- Framework de persistencia mapeo O/R
  - Hibernate, TopLink → EJB 3.0 JPA
- Conjunto de conectores a otros sistemas BackEnd
  - Ej. JCO o Business Connector para el acceso a SAP y BIW.
  - Integración con sistemas LEGACY
- Soluciones Híbridas de las anteriores.
- Generación de código JDBC

Acoplamiento entre MVC y N-Capas

MVC



N-CAPAS

Correspondencia de capas

MVC	N-Capas
Vista	Presentación
Controlador	Presentación
Modelo	Negocio/Persistencia

Referencias

- URLs
  - http://jakarta.apache.org/Struts
  - http://theserverside.com
- Libros
  - *Programming Jakarta Struts* de O'Reilly
  - *Mastering Tomcat Development* de WILEY
  - *Java Server Programming J2EE Edition* de Wrox
  - Marty Hall, Java Core Servlets
  - GOF94
  - Fowler93