1. La arquitectura multicapa

Índice

- Referencias
- Introducción
- Patrones auxiliares
  - Introducción
  - MVC: Modelo Vista Controlador.
  - Factoría abstracta.
  - Singleton.

Índice

- Arquitectura de una capa
  - Características
  - Ventajas e inconvenientes
- Arquitectura de dos capas
  - Características
  - Ventajas e inconvenientes
  - Patrones relacionados

Índice

- Arquitectura multicapa
  - Características
  - Ventajas e inconvenientes
  - Patrones relacionados
  - Aplicaciones web
  - Soluciones aplicables en base a los requisitos
- Patrón controlador frontal
- Patrón controlador de aplicación

Índice

- Patrón transferencia
- Patrón *Data Access Object*
- Patrón servicio de aplicación
- Patrón objeto del negocio
- Concurrencia en persistencia
- Patrón almacén del dominio
- Patrón delegado del negocio
- Conclusiones

Referencias

- Alur, D., Malks, D., Crupi. J. *Core J2EE Design Patterns: Best Practices and Design Strategies. 2nd Edition.* Prentice Hall, 2003.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizables.* Addison-Wesley, 2006

Referencias

- Mukhar, K., Zelenak, C. *Beginning Java EE 5. From Novice to Professiona*l. Apress, 2006
- Stelting, S., Maassen, O. *Patrones de diseño aplicados a Java*. Pearson Educación, 2003

Referencias

- Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003
- Juric, M.B., Basha, S.J., Leander, R., Nagappan, R. *Professional J2EE EAI*. Wrox Press, 2001

Introducción

- En este tema veremos los fundamentos de la *arquitectura multicapa*
- En particular, la enfocaremos en sistemas de información
- Lo visto es aplicable a otros tipos de sistemas
- Aunque es en sistemas de información donde cobra una especial relevancia

Introducción

- No nos preocuparemos de obtener buenos diseños a nivel componentes de cada capa
- De ello se ocupan disciplinas como:
  - Interacción persona-computadora
  - Programación
  - Patrones de diseño
  - Bases de datos

Introducción

- Nosotros nos preocuparemos de dar un buen sistema de información desde el punto de vista *arquitectónico*
- *Arquitectura** es la organización fundamental de un sistema, expresado en sus componentes, sus relaciones entre ellos, y en el entorno y principios que guían su diseño y evolución

*IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems

Introducción

- En particular veremos la arquitectura multicapa y sus patrones fundamentales
- Aunque son patrones extraídos de la ingeniería web*, son también aplicables a aplicaciones no web

*http://www.corej2eepatterns.com/index.htm

Introducción

- Un *sistema de información* es un sistema, manual o automático formado por personas, máquinas y/o métodos organizados para recopilar, procesar, transmitir y diseminar *datos* que representan *información* del usuario*
- Un *sistema de información* es un sistema que recopila y guarda información

*http://www.wikipedia.org/

Introducción

- *Dato* es una declaración aceptada como valor nóminal (p.e. 100)
- *Información* es una colección de datos procesados que tiene un significado adicional (p.e. 100º)
- *Conocimiento* es información de la que se es consciente, se entiende y puede ser utilizada para un propósito (p.e. el agua hierve a 100º)

Introducción

- Los sistemas de información tienen una gran relevancia en informática
- Según la ACM* incluyen:
  - Modelos y principios
  - Gestión de bases de datos
  - Almacenamiento y recuperación de información
  - Aplicaciones de sistemas de información
  - Interfaces y presentación de la información

*http://www.acm.org/class/1998/

Introducción

- Por eso, veremos los principales patrones de la arquitectura multicapa en sistemas de información
- Quizás una denominación más actual puede ser *patrones de arquitectura de aplicaciones empresariales*

Introducción

- Las características de las aplicaciones empresariales son:
  - Manejan una gran cantidad de datos persistentes
  - Estos datos son accedidos concurrentemente
  - Hay una gran cantidad de lógica del negocio, que representa la funcionalidad de la aplicación

Introducción

- El acceso se produce a través de elaboradas interfaces de usuario
- Suelen tener necesidades de integración con otras aplicaciones empresariales de arquitectura heterogénea

Introducción

- Según Christopher Alexander, "un *patrón* describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema de tal modo que se puede aplicar esta solución un millón de veces, sin hacer lo mismo dos veces"

Introducción

- Aunque Alexander se refería a patrones en ciudades y edificios, lo que dice también es válido para patrones de diseño OO
- Podemos decir que los patrones de diseño:
  - Son soluciones simples y elegantes a problemas específicos del diseño de software OO.
  - Representan soluciones que han sido desarrolladas y han ido evolucionando a través del tiempo.

Introducción

- Los patrones de diseño no tienen en cuenta cuestiones tales como:
  - Estructuras de datos.
  - Diseños específicos de un dominio.
- Son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto

Introducción

- Cada patrón de diseño identifica:
  - Las clases e instancias participantes.
  - Los roles y colaboraciones de dichas clases e instancias.
  - La distribución de responsabilidades

Introducción

- Algunas fuentes de patrones:
  - GRASP*, de Craig Larman.
  - Los patrones *Gang of Four* (GoF), de Eric Gamma et al.
  - *Core J2EE patterns*, de Alur et al.
  - *Patterns of Enterprise Application Architecture*, de Fowler et al.
  - *SOA Design Patterns*, de Thomas Erl.

**General Responsibility Assignment Software Patterns*

Patrones auxiliares
Introducción

- Son patrones de propósito general que nosotros usaremos más adelante en este tema y en el proyecto
- De ahí el calificativo *auxiliares*
- Son:
  - Modelo-Vista-Controlador
  - Fachada
  - Factoría abstracta
  - Singleton

Patrones auxiliares
MVC

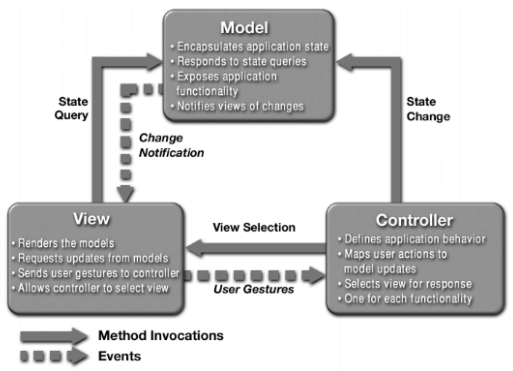
- El patrón/arquitectura *Modelo Vista Controlador MVC* divide una aplicación interactiva en tres componentes:

  - El *modelo* contiene la funcionalidad básica y los datos.

  - Las *vistas* muestran/recogen información al/del usuario.

  - Los *controladores* median entre vistas y modelo

Patrones auxiliares
MVC



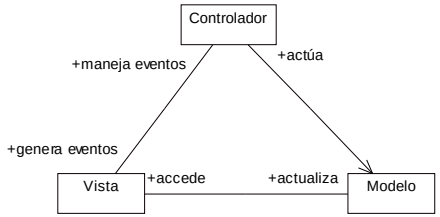
Patrón MVC

Patrones auxiliares
MVC

- El patrón MVC tiene dos variantes:
  - Modelo activo
  - Modelo pasivo

Patrones auxiliares
MVC

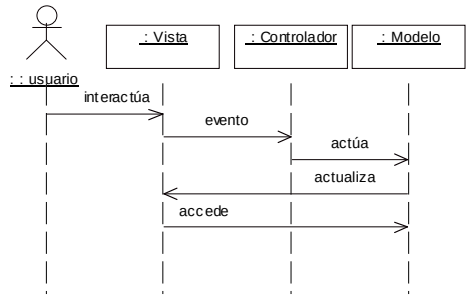
- Participantes en MVC. Modelo activo:



Participantes en MVC. Modelo activo

Patrones auxiliares
MVC

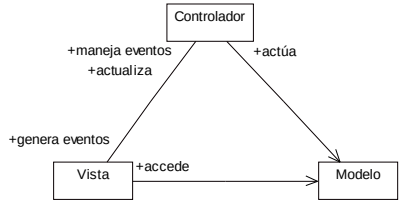
- Interacción en MVC. Modelo activo:



Interacción en MVC. Modelo activo

Patrones auxiliares
MVC

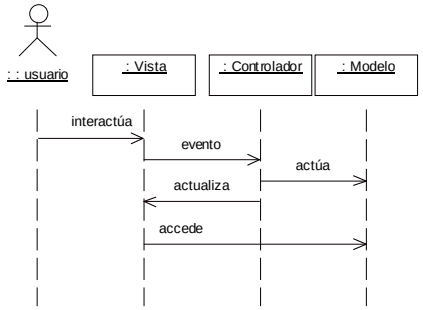
- Participantes en MVC. Modelo pasivo:



Participantes en MVC. Modelo pasivo

Patrones auxiliares
MVC

- Interacción en MVC. Modelo pasivo:



Interacción en MVC. Modelo pasivo

Patrones auxiliares
MVC

- Ventajas:

  - Modelo independiente de la representación de la salida y del comportamiento de la entrada.

  - Puede haber múltiples vistas para un mismo modelo.

  - Cambios independientes en interfaz/lógica.

- Inconvenientes

  - Complejidad

Patrones auxiliares
MVC

- Respecto al número de controladores, puede haber:
  - Uno por evento.
  - Uno por conjuntos de funcionalidades/estímulos.
  - Uno por aplicación.

Patrones auxiliares
MVC

- Código de ejemplo*

```
public class GUIBajaUsuario extends JFrame {
.............
  public GUIBajaUsuario()
  {
      setTitle("Baja usuario");
      JPanel panel= new JPanel();
      JLabel eId= new JLabel("Id:");
      final JTextField cId= new JTextField(20);
      JButton aceptar= new JButton("Aceptar");
      JButton cancelar= new JButton("Cancelar");
```

*Controlador único, modelo pasivo

Patrones auxiliares
MVC

```
panel.add(eId);
panel.add(cId);
panel.add(aceptar);
panel.add(cancelar);
getContentPane().add(panel);

pack();
```


Patrones auxiliares
MVC

```
aceptar.addActionListener(new ActionListener()
  { public void actionPerformed(ActionEvent e)
      {   setVisible(false);
          Integer id= new Integer(cId.getText());
          Controlador.getInstancia().
          accion(EventoNegocio.BAJA_USUARIO, id);
      }

  });
.....................................
}
```

Patrones auxiliares
MVC

```
public  class EventoNegocio {

   public static final int INSERTAR_USUARIO= 101;
   public static final int BAJA_USUARIO= 102;
   public static final int MOSTRAR_USUARIO= 103;
   public static final int INSERTAR_PUBLICACION= 201;
   public static final int BAJA_PUBLICACION= 202;
   public static final int MOSTRAR_PUBLICACION= 203;
   public static final int PRESTAMO= 301;
   public static final int DEVOLUCION= 302;
}
```


Patrones auxiliares
MVC

```
public class Controlador {
   ……..
   private SAUsuario saUsuario;
   private IGUI gui;
```


Patrones auxiliares
MVC

```
//implementación naif de una tabla de controlador
public void accion(int evento, Object datos)
{ switch (evento){
   case EventoNegocio.INSERTAR_USUARIO: {  …… }
   case EventoNegocio.BAJA_USUARIO: {
    Integer id= (Integer) datos;
    Boolean resultado= saUsuario.daDeBajaUsuario(id);
    if (resultado.booleanValue())
  gui.actualizar(EventoGUI.BAJA_USUARIO, resultado);
    else
  gui.actualizar(EventoGUI.USUARIO_INEXISTENTE_O_CON_
                        PRESTAMOS_O_NO_ACTIVO, null);
  break; }
……..
}
```


Patrones auxiliares
MVC

```
public interface IGUI {
 // no utilizamos java.util.Observer
 //porque obliga a que los datos sean observable

 void actualizar(int evento, Object datos);

}
```

Patrones auxiliares
MVC

```
public class GUIBiblioteca extends JFrame
  implements IGUI  {

  private static GUIBiblioteca guiBiblioteca;
  private IGUIUsuario guiUsuario;
  private IGUIPublicacion guiPublicacion;
  private IGUIPrestamo guiPrestamo;
  private Controlador controlador;
```


Patrones auxiliares
MVC

```
public void actualizar(int evento, Object datos)
  {   switch (evento)
      {
  case EventoGUI.MOSTRAR_GUI_BIBLIOTECA:
  { setVisible(true); break; }
  case EventoGUI.OCULTAR_GUI_BIBLIOTECA: {
  setVisible(false); break; }
.............
```


Patrones auxiliares
MVC

```
case EventoGUI.BAJA_USUARIO:
{ JOptionPane.showMessageDialog(null,
                      "Usuario eliminado");

  setVisible(true);
  break; }

.................
}

}
```


Patrones auxiliares
MVC

- Comentarios:

  - El patrón MVC es un caso particular del patrón *observador*.

  - Las vistas en MVC se pueden anidar. De esta forma una vista sería un caso particular del patrón *compuesto*.

Patrones auxiliares
MVC

- Enlaces *básicos* sobre MVC:

  http://www.enode.com/x/markup/tutorial/mvc.html
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesMVC.asp

  http://citeseer.ist.psu.edu/krasner88description.html*

- Enlaces *avanzados* sobre MVC:

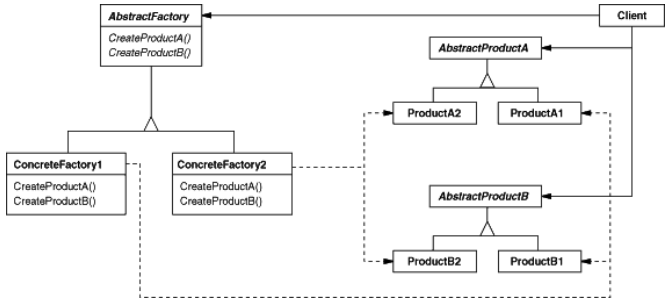
  http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/

*Versión previa de: Krasner, G.E., Pope, S.T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *JOOP* August/September 1988

Patrones auxiliares
Factoría abstracta

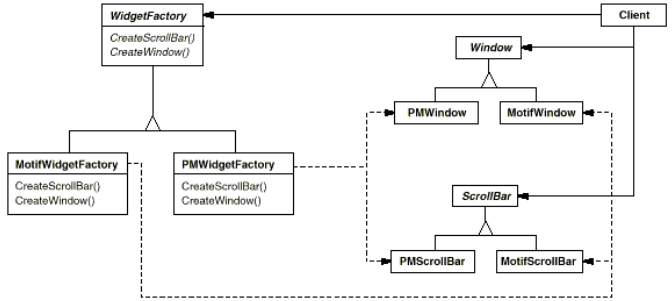
- El patrón *factoría abstracta* proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas
- Desliga la creación de nuevos objetos de la clase que se refiere a dichos objetos a través de la interfaz que implementan

Patrones auxiliares
Factoría abstracta



Patrón factoría abstracta

Patrones auxiliares
Factoría abstracta



Factoria abstracta para un conjunto de elementos de IGU

Patrones auxiliares
Factoría abstracta

- Ventajas:

  - Aísla las clases concretas que se crean en una aplicación y se manejan a través de los interfaces que implementan.

  - Facilita el intercambio de familias de productos.

  - Promueve la consistencia entre productos.

- Inconvenientes:

  - Nuevos tipos de productos provocan la redefinición de la factoría.

Patrones auxiliares
Factoría abstracta

- Código de ejemplo

```
public class ServiciosUsuarioImp implements
  ServiciosUsuario {


  public ServiciosUsuarioImp(FactoriaDAOUsuario
  factoria)
  {   DAOUsuario daoUsuario;

      daoUsuario= factoria.generaDAOUsuario();
  }
.....}
```


Patrones auxiliares
Factoría abstracta

```
public interface FactoriaDAOUsuario {
      public DAOUsuario generaDAOUsuario();
  }

public class FactoriaDAOUsuarioImp implements
  FactoriaDAOUsuario {

  public DAOUsuario generaDAOUsuario()
  {
      return new DAOUsuarioImp();
  }
}
```


Patrones auxiliares
Factoría abstracta

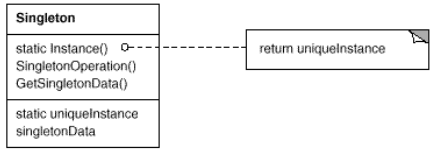
- Comentarios
  - En el ejemplo anterior, la factoría abstracta se crea fuera del cliente, y se pasa a éste como parámetro
  - ¿Quién crea la factoría? Se puede crear *manualmente* en el programa principal
  - Una opción mejor sería considerar a la factoria objeto *singleton*

Patrones auxiliares
Singleton

- El patrón *singleton* garantiza que sólo hay una instancia de una clase, proporcionando un único punto de acceso a ella
- Esta instancia podría ser redefinida mediante herencia
- Los clientes deberían ser capaces de utilizar estas subclases sin modificar su código

Patrones auxiliares
Singleton

- Estructura



Estructura del patrón Singleton

Patrones auxiliares
Singleton

- Ventajas

  - Acceso controlado a la única instancia.
  - Espacio de nombres reducido.
  - Permite el refinamiento de operaciones y la representación.
  - Permite un número variable de instancias.
  - Más flexibles que las operaciones de clase estáticas.

Patrones auxiliares
Singleton

- Código de ejemplo

```
public abstract class FactoriaDAOUsuario {
  private static FactoriaDAOUsuario factoria;

  public static FactoriaDAOUsuario
  obtenerInstancia()
  { if (factoria == null) factoria = new
      FactoriaDAOUsuarioImp();
    return factoria;
  }

  public abstract DAOUsuario generaDAOUsuario();
}
```

Patrones auxiliares
Singleton

```
public class FactoriaDAOUsuarioImp extends
  FactoriaDAO {

  public abstract DAOUsuario generaDAOUsuario()
  {
      return new DAOUsuarioImp();
  }
}
```


Patrones auxiliares
Singleton

- Nota:
  - En el ejemplo anterior, el singleton siempre crea la misma clase de factoría
  - Por lo tanto, si los clientes quieren obtener otra implementación de la factoría, debería cambiarse el código de ésta a nivel paquete
  - Hay opciones más razonables

Patrones auxiliares
Singleton

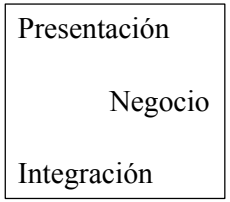
  - Su método de generación lee de un archivo la clase concreta que implementa a dicha factoría y que debe generar, la carga dinámicamente y se la devuelve al cliente

    http://developer.classpath.org/doc/javax/xml/parsers/DocumentBuilderFactory-source.html

- Nota: además, para evitar problemas de creación y/o carga, en entornos concurrentes, el método estático que devuelve la instancia debe garantizar el acceso concurrente (e.g. `synchronized` en Java)

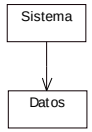
Arquitectura de una capa
Características

- La arquitectura de una capa no divide al sistema en presentación, negocio e integración



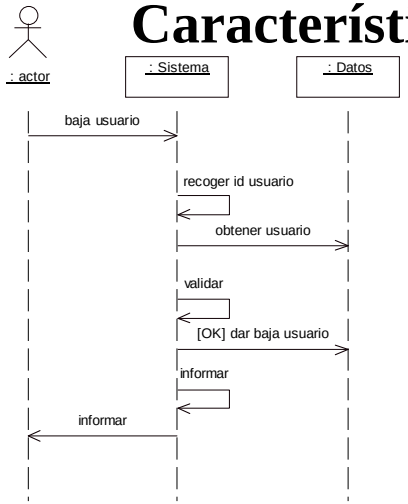
Arquitectura de una capa

Arquitectura de una capa
Características



Clases del sistema

Arquitectura de una capa
Características



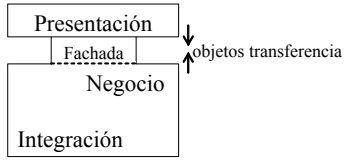
Comportamiento del sistema

Arquitectura de una capa
Ventajas e inconvenientes

- Ventajas
  - Sencillez conceptual
- Inconvenientes
  - No se puede modificar ni la interfaz de usuario, ni la lógica del negocio ni la representación de los datos sin afectar a las demás capas
  - Complicación fáctica

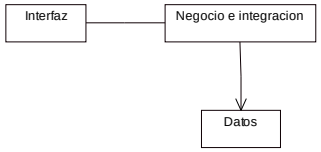
Arquitectura de dos capas
Características

- La arquitectura de dos capas diferencia entre la capa de presentación y el resto del sistema
- No diferencia negocio de integración



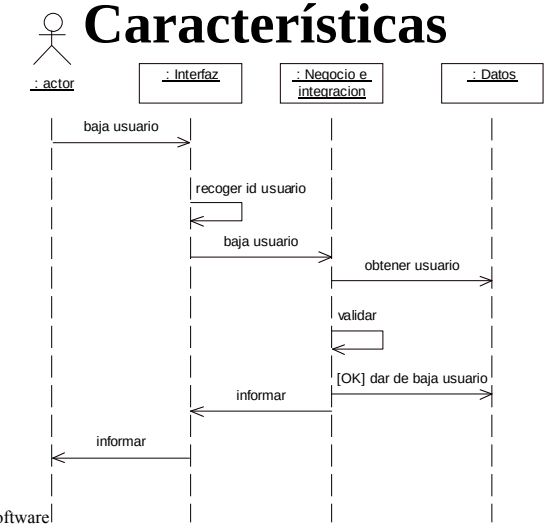
Arquitectura de dos capas

Arquitectura de dos capas
Características



Clases del sistema

Arquitectura de dos capas
Características



Comportamiento del sistema

Arquitectura de dos capas
Ventajas e inconvenientes

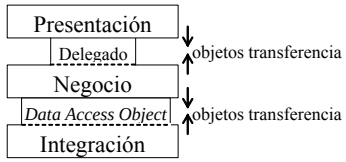
- Ventajas
  - Permite cambios en el interfaz de usuario o en el resto del sistema sin interferencias mutuas
  - Simplicidad fáctica
- Inconvenientes
  - Mayor complicación arquitectónica que la arquitectura de una capa
  - No se puede modificar la lógica del negocio o la representación de los datos sin interferencias mutuas

Arquitectura de dos capas
Patrones relacionados

- Aunque no es estrictamente necesario, suele utilizarse:
  - MVC

Arquitectura multicapa Características

- La arquitectura multicapa considera una capa de presentación, otra de negocio, y otra de integración



Arquitectura multicapa

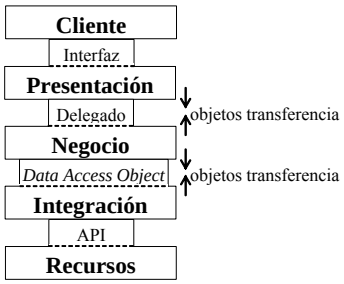
Arquitectura multicapa Características

- La *capa de presentación* encapsula toda la lógica de presentación necesaria para dar servicio a los clientes que acceden al sistema
- La *capa de negocio* proporciona los servicios del sistema
- La *capa de integración* es responsable de la comunicación con recursos y sistemas externos

Arquitectura multicapa Características

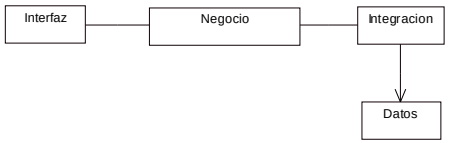
- En realidad, la arquitectura es de *cinco capas*, ya que incluye las capas de clientes y recursos
- La *capa de clientes* representa a todos los dispositivos o clientes del sistema que acceden al mismo. Está sobre la capa de presentación
- La *capa de recursos* contiene los datos del negocio y recursos externos. Está bajo la capa de integración

Arquitectura multicapa Características



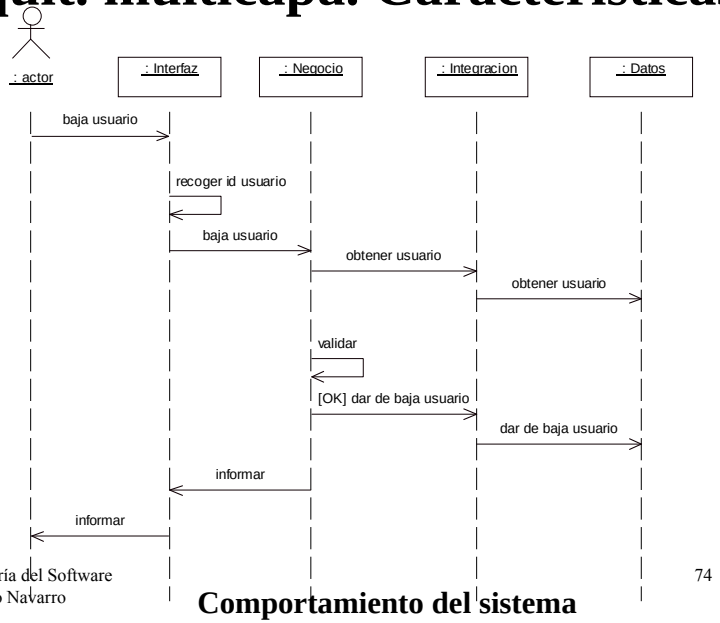
Arquitectura multicapa

Arquitectura multicapa Características



Clases del sistema

Arquit. multicapa. Características



Comportamiento del sistema

Arquitectura multicapa Características

- Nótese que estas son capas *lógicas*
- Otra cosa son las capas *físicas*
- Así, la capa de presentación web y la lógica del negocio podrían estar en la misma máquina o en máquinas distintas

Arquitectura multicapa Características

- Ventajas
  - Se puede modificar cualquier capa sin afectar a a las demás
  - ¿Simplicidad fáctica?
- Inconvenientes
  - Mayor complejidad arquitectónica

Arquitectura multicapa
Características

- Ventajas:
  - Integración y reusabilidad
  - Encapsulación
  - Distribución
  - Particionamiento
  - Escalabilidad
  - Mejora del rendimiento
  - Mejora de la fiabilidad

Arquitectura multicapa
Características

- Manejabilidad
- Incremento en la consistencia y flexibilidad
- Soporte para múltiples clientes
- Desarrollo independiente
- Desarrollo rápido
- Empaquetamiento
- Configurabilidad

Arquitectura multicapa
Características

- Inconvenientes:
  - Posible pérdida de rendimiento y escalibilidad
  - Riesgos de seguridad
  - Gestión de componentes

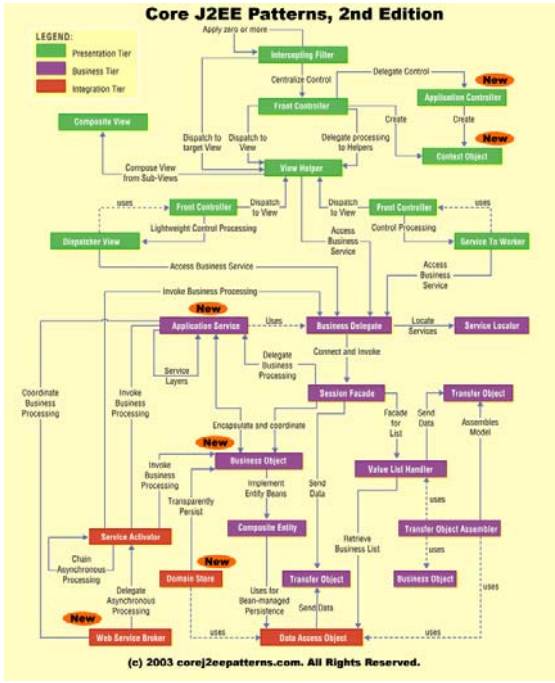
Arquitectura multicapa
Patrones relacionados

- Patrones relacionados:
  - Controlador frontal (capa de presentación)
  - Controlador de aplicación (capa de presentación)
  - Transferencia (capa de negocio)
  - *Data Access Object* (DAO) (capa de integración)
  - Servicio de aplicación (capa de negocio)
  - Delegado del negocio (capa de negocio)

Arquitectura multicapa
Patrones relacionados

– Objeto del negocio (capa de negocio)
– Almacén del dominio (capa de integración)

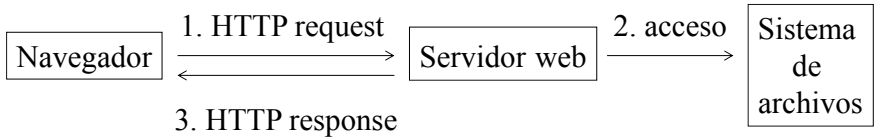
Catálogo de patrones multicapa



Arquitectura multicapa
Aplicaciones web

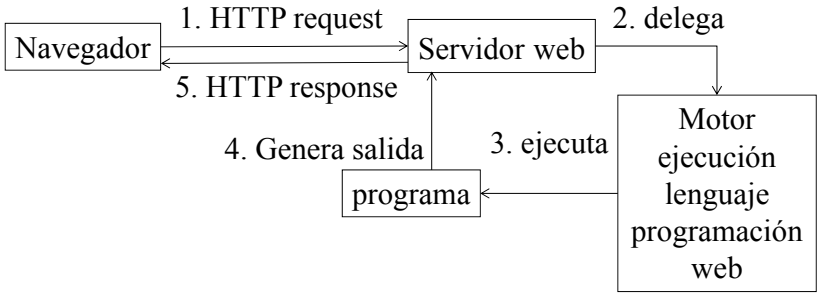
- Un servidor web utiliza el protocolo HTTP para recibir y enviar información a través de *requests/responses* (peticiones/respuestas)

Arquitectura multicapa
Aplicaciones web



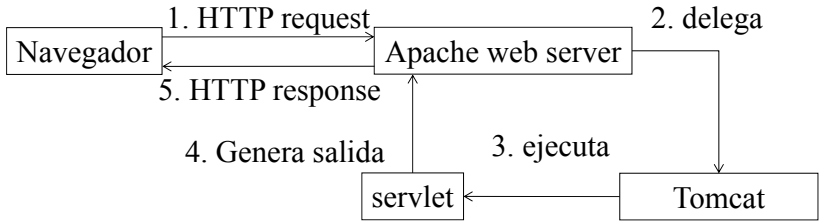
Esquema de funcionamiento de un servidor web

Arquitectura multicapa Aplicaciones web



Esquema de funcionamiento de un servidor web extendido para ejecutar código de propósito general

Arquitectura multicapa Aplicaciones web



Ejemplo concreto de servidor y motor de ejecución

Arquitectura multicapa Aplicaciones web

- Ejemplo*:


```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

*http://www.roseindia.net/servlets/HelloWorld.shtml

Arquitectura multicapa Aplicaciones web

```
public class HelloWorld extends HttpServlet{
  public void doGet(HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException,IOException{
  response.setContentType("text/html");
  PrintWriter pw = response.getWriter();
  pw.println("<html>");
  pw.println("<head><title>Hello
World</title></title>");
  pw.println("<body>");
  pw.println("<h1>Hello World</h1>");
  pw.println("</body></html>");
  }
}
```

Arquitectura multicapa
Aplicaciones web

```
<web-app>
 <servlet>
  <servlet-name>Hello</servlet-name>
  <servlet-class>HelloWorld</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>Hello</servlet-name>
 <url-pattern>/HelloWorld</url-pattern>
 </servlet-mapping>
</web-app>
```


Tipo de aplicación	Solución	Uso de marcos	Ejemplos implementación J2EE		
			Pres.	Negocio	Integración
Empresarial	Arquitectura multicapa	No	JSP	SAs POJOs + transfers	DAOs POJOs
		Sí	JSF	SAs POJOs + entidades	JPA
+ Lógica distribuida	+ RPC (Remote Procedure Call)	No	JSP	SAs POJOs + RMI + transfers	DAOs POJOs
		Sí	JSF	EJBs de sesión + entidades	JPA
+ Plataformas heterogéneas	+ SOA	No	JSP	SAs POJOs + transfers + JAX WS / JAX RS	DAOs POJOs
		Sí	JSF	EJBs de sesión JAX WS / JAX RS + entidades	JPA

Solución aplicable en base a los requisitos de aplicación

Patrón controlador frontal

- Propósito
  - Proporciona un punto de acceso para el manejo de las peticiones de la capa de presentación
- También conocido como
  - *Front controller*

Patrón controlador frontal

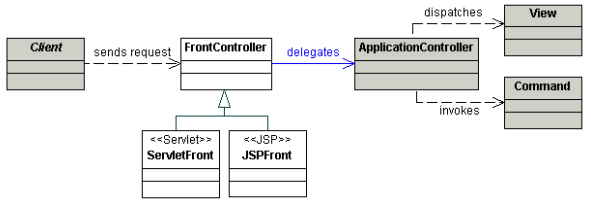
- Motivación
  - Se desea evitar lógica de control duplicada
  - Se desea aplicar una lógica común a distintas peticiones
  - Se desea separar la lógica de procesamiento del sistema de la vista
  - Se desea tener puntos de acceso centralizado y controlado al sistema

Patrón controlador frontal

- Debe aplicarse cuando
  - Se quiera tener un punto inicial de contacto para manejear las peticiones, centralizando la lógica de control y manejando las actividades de manejo de peticiones

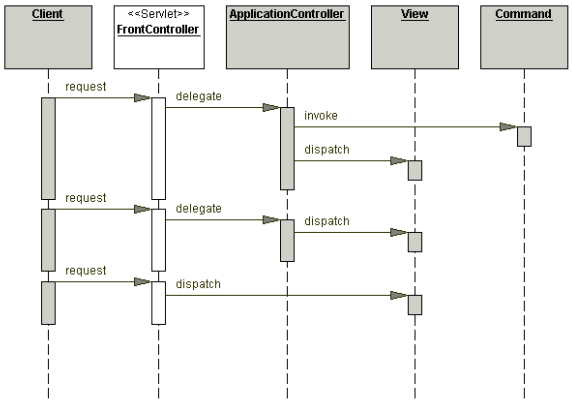
Patrón controlador frontal

- Estructura



Estructura del patrón controlador frontal

Patrón controlador frontal



Interacción de objetos relacionados por el controlador frontal

Patrón controlador frontal

- Consecuencias
  - Ventajas:
    - Centraliza el control
    - Mejora la gestión de la aplicación
    - Mejora la reutilización
    - Mejora la separación de roles
  - Inconvenientes
    - En aplicaciones grandes puede llegar a crecer mucho

Patrón controlador frontal

- Código de ejemplo

```
public class FrontController extends HttpServlet
{
 protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws
ServletException, java.io.IOException {

      processRequest(request, response);

    }
```


Patrón controlador frontal

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException
{

      processRequest(request, response);
    }
```


Patrón controlador frontal

```
protected void processRequest(HttpServletRequest
request, HttpServletResponse response) throws
ServletException, java.io.IOException {
      String page;
      ApplicationResources resource =
ApplicationResources.getInstance();
      try {
        RequestContext requestContext =
            new RequestContext(request, response);
```


Patrón controlador frontal

```
ApplicationController applicationController = new
            ApplicationControllerImpl();
ResponseContext responseContext =
applicationController.handleRequest(requestContext);
applicationController.handleResponse(
             requestContext, responseContext);
      } catch (Exception e) {
LogManager.logMessage("FrontController:exception : " +
            e.getMessage());
request.setAttribute(resource.getMessageAttr(),
            "Exception occurred : " + e.getMessage());
        page = resource.getErrorPage(e);
```

Patrón controlador frontal

```
dispatch(request, response, page);
      }
    }
//sólo se utiliza esta función si hay error
protected void dispatch(HttpServletRequest request,
HttpServletResponse response, String page)
throws javax.servlet.ServletException, java.io.IOException
{
 RequestDispatcher dispatcher = this.getServletContext().
          getRequestDispatcher(page);
  dispatcher.forward(request, response);
    }
}
```


Patrón controlador de aplicación

- Propósito
  - Se desea centralizar y modularizar la gestión de acciones y de vistas
- También conocido como
  - *Application controller*

Patrón controlador de aplicación

- Motivación
  - Se desea reutilizar el código de gestión de vistas y acciones
  - Se desea mejorar la extensibilidad de el manejo de peticiones (p.e. añadir casos de uso a una aplicación incrementalmente)

Patrón controlador de aplicación

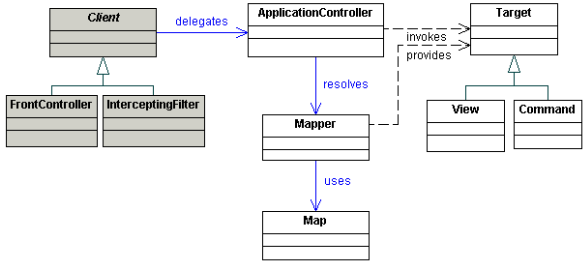
- Se desea mejorar la modularidad del código y la mantenibilidad, facilitando al extensión de la aplicación y la prueba del código de manejo de peticiones de manera independiente del contenedor web

Patrón controlador de aplicación

- Debe aplicarse cuando
  - Se quiera centralizar la recuperación e invocación de componentes de procesamiento de las peticiones, tales como comandos y vistas

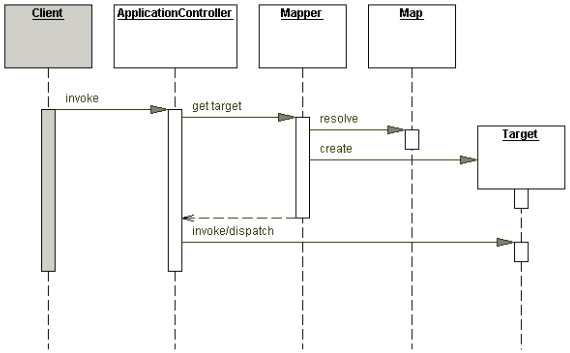
Patrón controlador de aplicación

- Estructura



Estructura del patrón controlador de aplicación

Patrón controlador de aplicación



Interacción entre objetos relacionados por controlador de aplicación

Patrón controlador de aplicación

- Consecuencias
  - Ventajas
    - Mejora la modularidad
    - Mejora la reutilización
    - Mejora la extensibilidad
  - Inconvenientes
    - Aumenta el número de objetos involucrados
    - En aplicaciones grandes puede llegar a crecer mucho

Patrón controlador de aplicación

- Código de ejemplo

```
interface ApplicationController {
ResponseContext handleRequest(RequestContext
requestContext);
void  handleResponse(RequestContext requestContext,
ResponseContext responseContext);
}
```


Patrón controlador de aplicación

```
class WebApplicationController implements
ApplicationController {

public ResponseContext handleRequest(RequestContext
requestContext) {
 ResponseContext responseContext = null;
 try {
      String commandName =
requestContext.getCommandName();
```


Patrón controlador de aplicación

```
CommandFactory commandFactory =
CommandFactory.getInstance();
Command command =
commandFactory.getCommand(commandName);
CommandProcessor commandProcessor = new
CommandProcessor();
responseContext = commandProcessor.invoke(command,
            requestContext);
      } catch (java.lang.InstantiationException e) {
      } catch (java.lang.IllegalAccessException e) {
      }
      return responseContext; }
}
```


Patrón transferencia

- Propósito
  - Independizar el intercambio de datos entre capas
- También conocido como
  - *Transfer*

Patrón transferencia

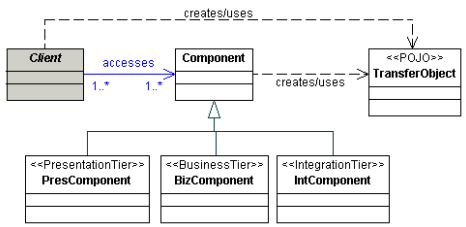
- Motivación
  - Si queremos independizar las capas, éstas no pueden tener conocimiento de la representación de las entidades de nuestro sistema dentro de cada capa
  - Por ejemplo, si accedemos a bases de datos relacionales, los clientes deberían abstrae de la existencia de *columnas* en los datos

Patrón transferencia

- Debe aplicarse cuando
  - No se desee conocer la representación interna de una entidad dentro de una capa
- Nota
  - Al ser un mecanismo de comunicación entre capas, son objetos serializables

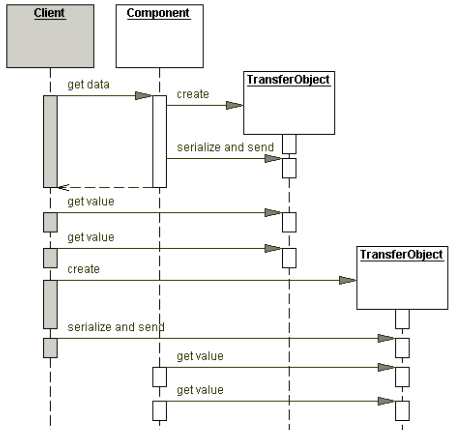
Patrón tranferencia

- Estructura



Estructura del patrón transferencia

Patrón transferencia



Interacción entre objetos relacionados por el patrón transferencia

Patrón transferencia

- Consecuencias
  - Ventajas
    - Ayuda a independizar capas
  - Inconvenientes
    - Aumenta significativamente el número de objetos del sistema

Patrón transferencia

- Código de ejemplo

```
public TransferLibro {
//atributos de libro
public String titulo;
.........
//accesores y mutadores de libro
public String getTitulo()
{ return titulo; }
.........
public void setTitulo(String titulo)
{ this.titulo= titulo; }
.........
```


Patrón transferencia

```
public DAOEjemplaresImp implements DAOEjemplares {

  public TransferLibro obtenerLibro(String id)
  {
      //código acceso a la base de datos

      TransferLibro libro= new TranserLibro(titulo, …);

      return libro;
  }
  ..............
}
```


Patrón DAO

- Propósito
  - Permite acceder a la capa de datos (recursos, en general), proporcionando representaciones orientadas a objetos (e.g. objetos transferencia) a sus clientes
- También conocido como
  - *Data access object*
  - Objeto de acceso a datos

Patrón DAO

- Motivación
  - Los sistemas de información (y muchos programas) guardan datos del usuario
  - Estos datos suelen tener estructura, la cual queda plasmada en un sistema de representación (p.e., relacional, XML)

Patrón DAO

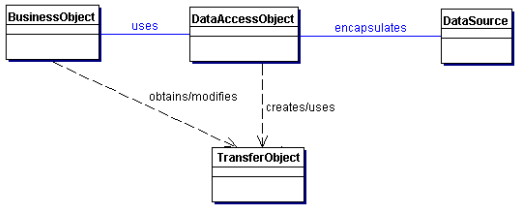
  - Manejar estos datos fuerza a:
    - Conocer los mecanismos de acceso del sistema de gestión de datos (p.e., base de datos, sistema operativo, etc.)
    - Conocer la representación de los datos en el sistema de gestión de datos (p.e., columnas, elementos, bytes, etc.)
  - Un cliente de la capa de negocio debería ser independiente de estas cuestiones

Patrón DAO

  - Así, se podría cambiar la capa de datos, sin afectar a la capa de negocio. Solamente habría que actualizar la capa de integración, más ligera que la de negocio
- Debe aplicarse cuando
  - Se quiera independizar la representación y acceso a los datos de su procesamiento

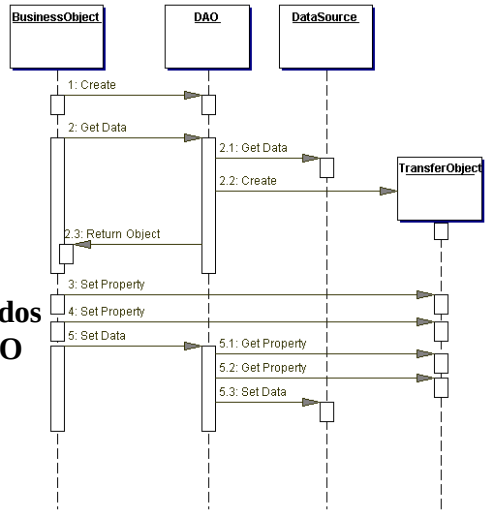
Patrón DAO

- Estructura



Estructrura del patrón DAO

Patrón DAO



Interacción entre objetos relacionados por el patrón DAO

Patrón DAO

- Consecuencias
  - Ventajas:
    - Independiza el tratamiento de los datos de su acceso y estructura
    - Permite independizar la capa de negocio de la de datos
  - Inconvenientes
    - Aumenta el número de objetos del sistema

Patrón DAO

- Código de ejemplo

```
public interface DAOUsuario {
  //podría haberse considerado un DAO para cada
  //operación
  public Integer insertaUsuario(TUsuario tUsuario);
  public Boolean daDeBajaUsuario(Integer id);
  public TUsuario obtenUsuario(Integer id);
  public Boolean modificaUsuario(TUsuario tUsuario);

}
```


Patrón DAO

```
public class DAOUsuarioImp implements DAOUsuario {
……
  public Boolean daDeBajaUsuario(Integer idInteger)
  {
      boolean resultado= true;
      int id= idInteger.intValue();

      //conexión con la base de datos
```

Patrón DAO

```
        Sring plantilla= "UPDATE usuario SET
                         activo=false WHERE id=?";
        PreparedStatement pstmt=
                  con.prepareStatement(plantilla);

        pstmt.setInt(1, id);
        resultado= (pstmt.executeUpdate() > 0);

        //cerrar conexión y tratar excepciones

        return new Boolean(resultado);
    }
.........
}
```


Patrón DAO

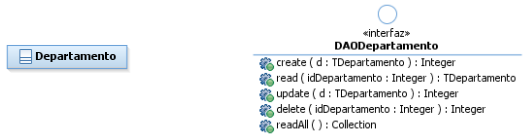
- Aunque en estas transparencias se obvia, es fundamental que los DAOs capturen y lancen las excepciones correspondientes al acceder a los recursos externos
- Así, la capa de negocio sabrá qué ha sucedido si ha habido algún tipo de fallo en dicho acceso

Patrón DAO

- NOTA
  - Aunque, por lo general, los DAOs sólo debería tener las operaciones CRUD (Create, Read, Update y Delete), es posible que en una arquitectura multicapa sin objetos del negocio, necesitemos enriquecer a los DAOs para facilitar la gestión de las relacione 1..n y m..n

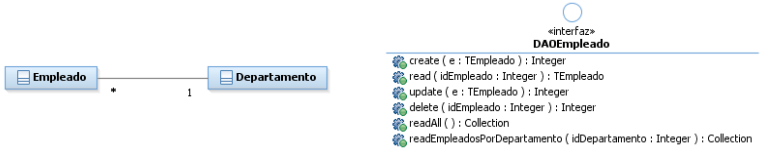
Patrón DAO

- Para una clase:



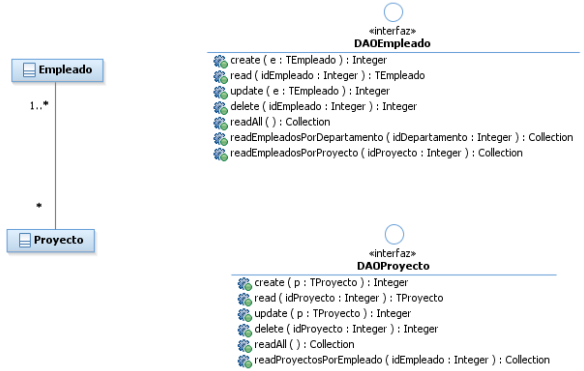
Patrón DAO

– Para una clase, extremo N de una relación 1..N



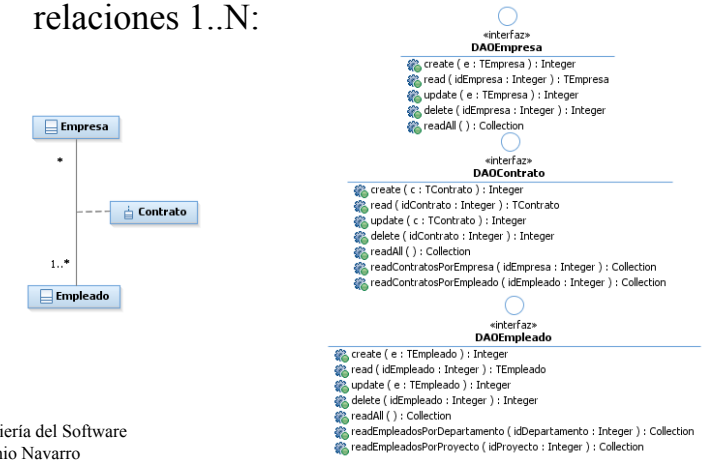
Patrón DAO

– Para dos clases extremos de una relación M..N



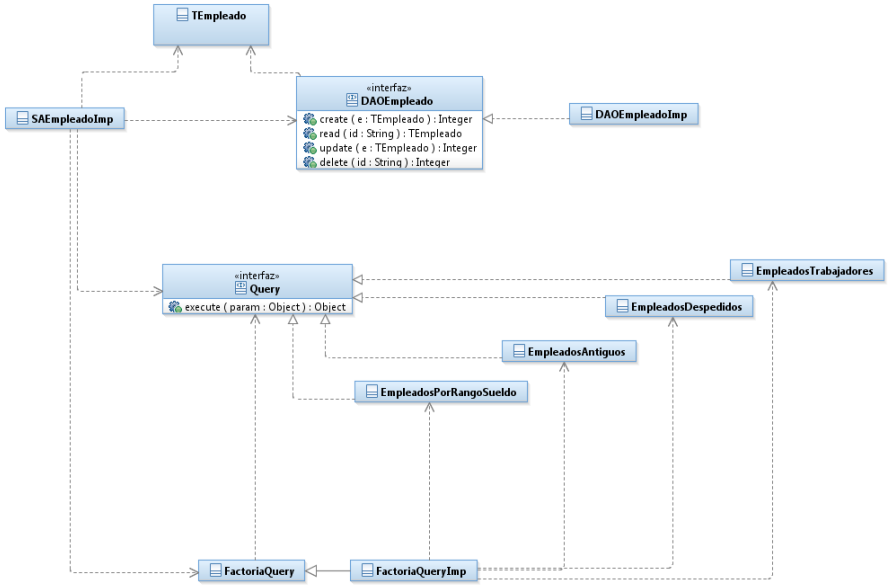
Patrón DAO

– En el caso de una clase asociación, es como dos relaciones 1..N:



Patrón DAO

- Hay veces que se debe seleccionar un conjunto de elementos que se corresponde con una query muy compleja
- En JPA tenemos JPA QL
- Una opción podría ser incluir estas querys como funciones del DAO
- Otra podría ser tenerlas como objetos querys, desvinculadas de los DAOS



Querys externas a los DAOs

Patrón servicio de aplicacion

- Propósito
  - Centraliza lógica del negocio.
- También conocido como:
  - *Application service*

Patrón servicio de aplicacion

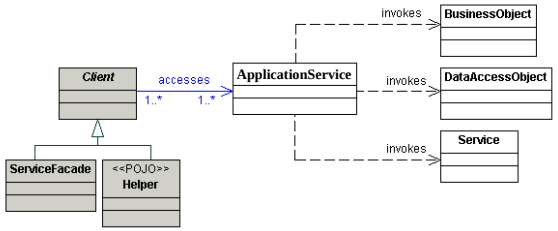
- Motivación
  - En una arquitectura multicapa, la lógica del negocio debe estar en algún sitio.
  - Dejarla en los clientes, vulneraría una estructura multicapa.
  - Incluirla en la fachada corrompería su naturaleza
  - Por eso la incluimos en servicios de la aplicación.

Patrón servicio de aplicacion

- Debe aplicarse cuando
  - Se quiera representar una lógica del negocio que actúe sobre distintos servicios u *objetos del negocio*.
  - Se quiera agrupar funcionalidades relacionadas.
  - Se quiera encapsular lógica no representada por objetos del negocio.

Patrón servicio de aplicacion

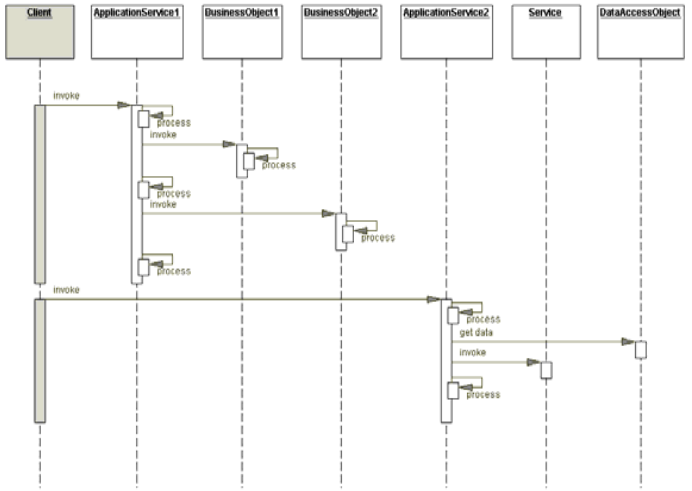
- Estructura



Estructura del patrón servicio de aplicación

Patrón servicio de aplicacion

Interacción entre objetos relacionados por el patrón servicio de aplicación



Patrón servicio de aplicacion

- Consecuencias
  - Ventajas
    - Centraliza lógica del negocio.
    - Mejora la reusabilidad del código.
    - Evita duplicación de código.
    - Simplifica la implementación de fachadas
  - Inconvenientes
    - Introduce un nivel más de indirección

Patrón servicio de aplicacion

- Código de ejemplo:

```
package logica.serviciosPrestamo;

import transferenciaCliente.prestamo.TPrestamo;


public interface ServiciosPrestamo {
  public TPrestamo prestamo(TPrestamo tPrestamo);

  public Boolean devolucion(Integer ejemplar);
}
```

Patrón servicio de aplicacion

```
package logica.serviciosPrestamo;
……….

public class ServiciosPrestamoImp implements
   Servicios Prestamo {
   public TPrestamo prestamo(TPrestamo tPrestamo)
   { …… }

   public Boolean devolucion(Integer ejemplar)
   { …… }


}
```


Patrón servicio de aplicacion

- Nota
  - Aunque en el ejemplo del libro *Core J2EE Patterns*, los servicios de la aplicación colaboran entre ellos para obtener objetos del negocio (y por extensión, los datos), esta aproximación podría complicar las validaciones de consistencia de los datos en un entorno multiusuario no EJB
  - En cualquier caso, nótese que el servicio de aplicación invocado en el ejemplo (t154), tiene toda la pinta de no acceder a información persistente

Patrón servicio de aplicacion

- Nota
  - Los servicios de aplicación no suelen tener atributos para hacerlos *más ligeros*
  - Entonces, ¿dónde están los objetos que tienen atributos y operaciones del negocio?
  - Estos objetos son los *objetos del negocio*

Patrón objeto del negocio

- Propósito
  - Representar la lógica del negocio y el modelo del dominio en términos orientados a objetos
- También conocido como:
  - *Business object*

Patrón objeto del negocio

- Motivación
  - Cuando la lógica del negocio es poca o inexistente, las aplicaciones pueden permitir a los clientes acceder directamente a la capa de datos.
  - Así, un componente de la capa de negocio (e.g. `ServiciosUsuarioImp`) podría acceder directamente a un DAO.

Patrón objeto del negocio

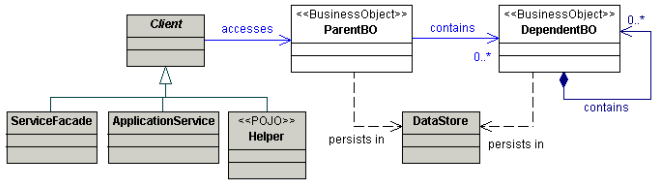
  - Sin embargo, si en el cliente hay una gran cantidad de procesos computacionales asociados a los datos, dichos procesos deberían encapsularse en un objeto que representase un objeto del negocio

Patrón objeto del negocio

- Debe aplicarse cuando:
  - Se disponga de un modelo conceptual con reglas de validación y lógica del negocio avanzadas.
  - Se desee separar la lógica del negocio del resto de la aplicación.
  - Se desee centralizar la lógica del negocio
  - Se desee incrementar la reusabilidad del código.

Patrón objeto del negocio

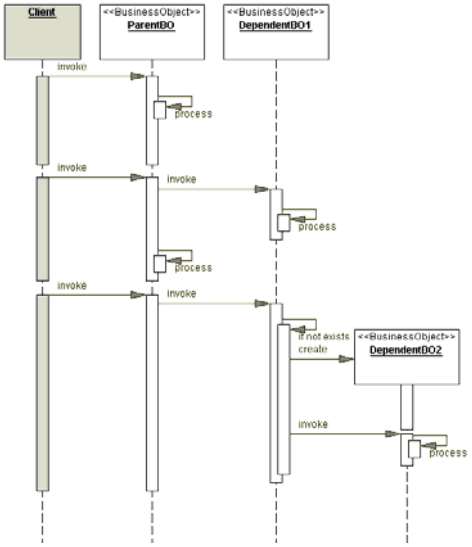
- Estructura

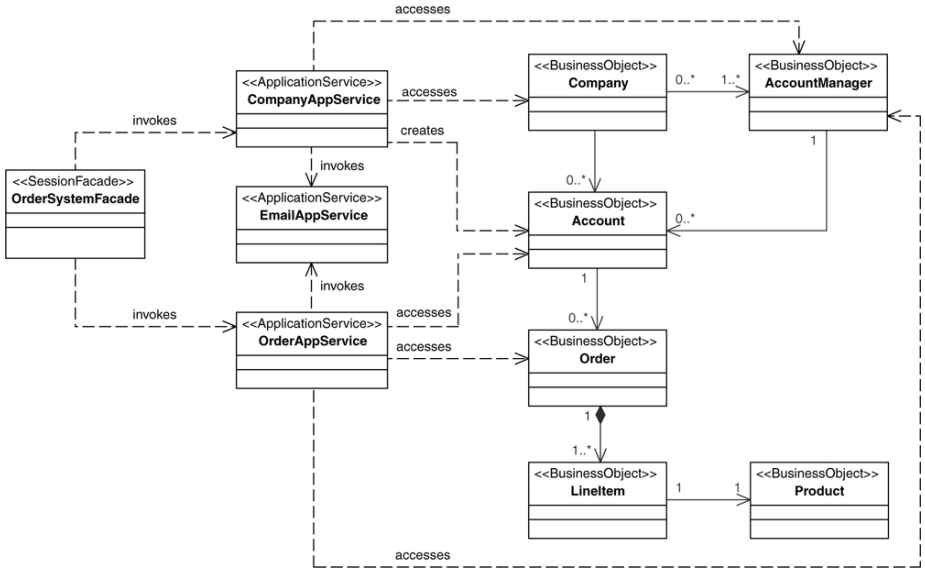


Estructura del patrón objeto del negocio

Patrón objeto del negocio

Interacción entre objetos relacionados por el patrón objeto del negocio





Relación entre objetos del negocio y servicios de aplicación

Patrón objeto del negocio

- Consecuencias
  - Ventajas
    - Promueve una aproximación orientada a objetos en la implementación del modelo del negocio.
    - Centraliza el comportamiento del negocio, promoviendo la reutilizabilidad.
    - Evita la duplicación de código

Patrón objeto del negocio

- Inconvenientes
  - Añade una capa de indirección.
  - Puede producir objetos "inflados" de funcionalidad.
  - Persistencia de dichos objetos del negocio.

Patrón objeto del negocio

- Código de ejemplo

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

    @ManyToOne
    private Department department;
.....
}
```

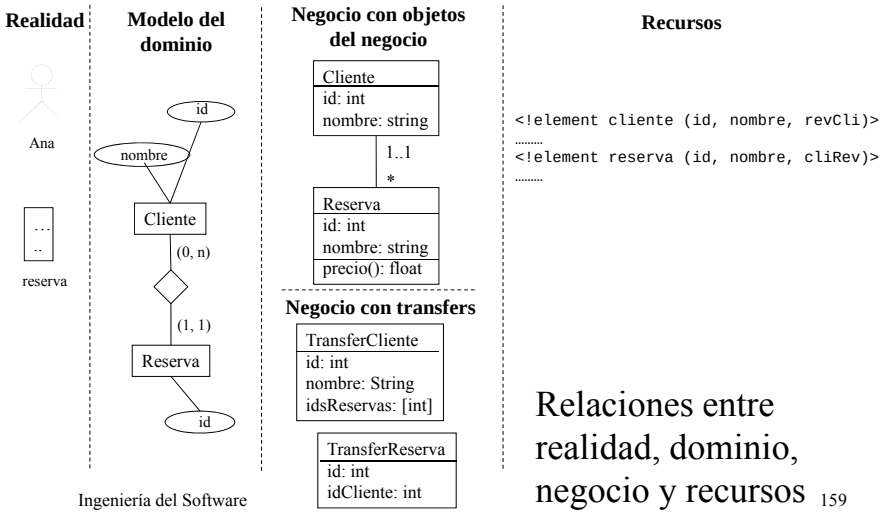

Patrón objeto del negocio

```
@Entity
public class Department {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;

    public int getId() {
        return id;     }

    public void setId(int id) {
        this.id = id;         }
.....
}
```


Nota

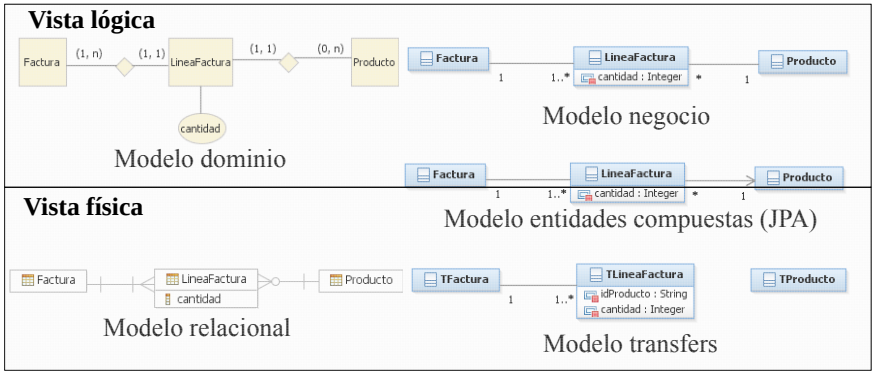


Relaciones entre realidad, dominio, negocio y recursos

Patrón objeto del negocio

- Hay una equivalencia bastante directa entre:
  - Modelo dominio y modelo negocio
  - Tablas y transfers
  - Las entidades compuestas serían algo así como una visión intermedia entre el modelo del dominio y el modelo propuesto por los transfers

Patrón objeto del negocio



Equivalencias entre distintos modelos

Concurrencia en persistencia Introducción

- La concurrencia es uno de los aspectos más complejos en desarrollo de software
- Aparece cuando se tienen distintos procesos o hebras manipulando los mismos datos
- En aplicaciones empresariales lo resolvemos con los transaction managers

Concurrencia en persistencia Problemas

- *Pérdida de actualizaciones* en la BBDD*:*
  - La hebra A toma el control y lee el empleado 44
  - La hebra B toma el control, despide al empleado 44 (active=false) y termina
  - La hebra A sube el sueldo al empleado 44 y termina
  - El empleado 44 no es despedido (suponemos un update de todo el empleado) y le suben el sueldo

Concurrencia en persistencia Problemas

- *Lectura inconsistente* en memoria:
  - La hebra A toma el control y lee la tabla proyectos
  - La hebra B toma el control da de baja al proyecto 27, despide a sus empleados y termina
  - La hebra A toma el control y lee la tabla de empleados
  - La hebra A va a tener el proyecto 27 con todos sus empleados despedidos

Concurrencia en persistencia Problemas

- Ambos problemas son un ejemplo de fallo en la *corrección*, y se debe al acceso concurrente a los mismos datos
- Pueden resolverse eliminando la concurrencia, pero eso nos llevaría a un problema de *viveza*: la cantidad de actividad concurrente que puede llevarse a cabo simultáneamente

Concurrencia en persistencia Problemas

- Corrección y viveza son atributos en tensión que tendrán que equilibrarse en cada aplicación

Concurrencia en persistencia Contextos de ejecución

- En una aplicación empresarial hay dos contextos de ejecución importantes: la petición (request) y la sesión
- Una *petición* es una única llamada del mundo exterior a nuestra aplicación que puede ser contestada con una respuesta (response)

Concurrencia en persistencia Contextos de ejecución

- Una *sesión* es una interacción de larga duración entre un cliente y un servidor. Normalmente incluirá varios ciclos petición/respuesta
- Normalmente la sesión tiene lugar dentro de la misma *hebra* de ejecución, que es un mecanismo de ejecución concurrente más ligero que el *proceso*

Concurrencia en persistencia
Contextos de ejecución

- En lo referente a datos, hay un contexto importante: la *transacción*, que agrupa acciones contra la base de datos:
  - *Transacción del sistema*: acciones de la aplicación contra la base de datos
  - *Transacción de negocio*: acciones del usuario contra la aplicación

Concurrencia en persistencia
Aislamiento e inmutabilidad

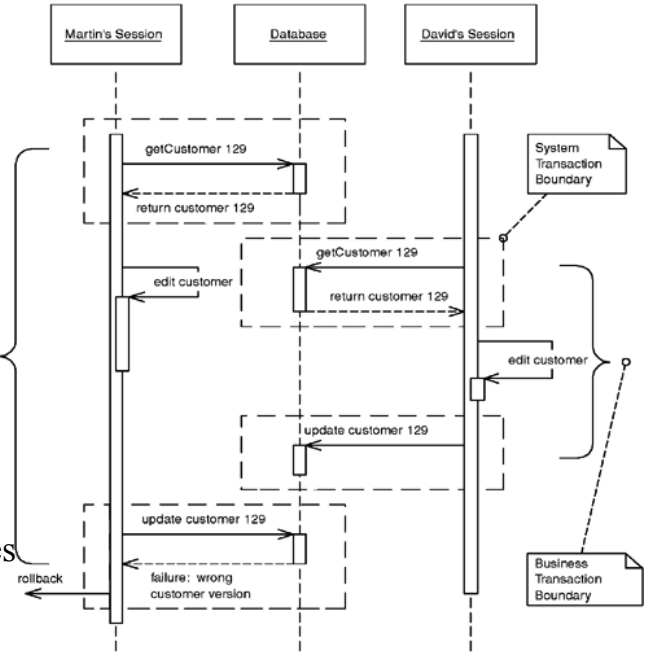
- El *aislamiento* divide los datos de tal forma que cualquier elemento suyo sólo pueda ser accedido por un agente activo (i.e. proceso o hebra)
- La *inmutabilidad* fuerza a que un dato no pueda ser cambiado, evitando así problemas de concurrencia

Concurrencia en persistencia
Aislamiento e inmutabilidad

- Una forma sencilla de inmutabilidad es permitir accesos de sólo lectura: los datos no son inmutables, pero los clientes sólo pueden leerlos

Concurrencia en persistencia
Control optimista y pesimista

- ¿Qué hacer cuando tenemos datos mutables que no pueden ser aislados?

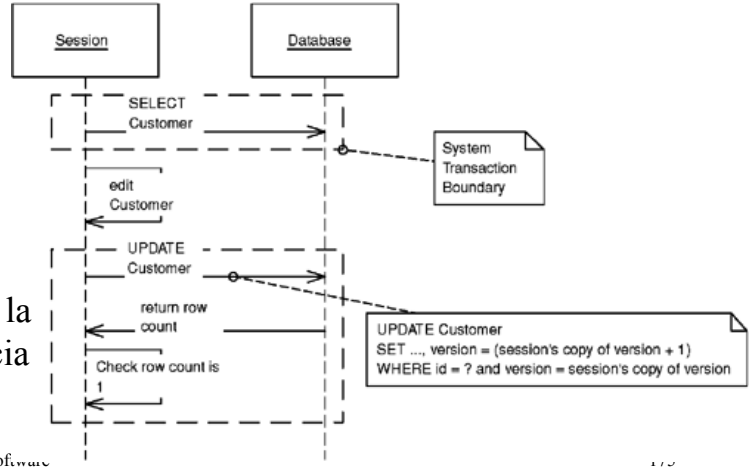


Problemas de acceso y modificaciones concurrentes

Concurrencia en persistencia Control optimista y pesimista

- Podemos hacer un control de la concurrencia optimista y pesimista
- El control de la concurrencia *optimista* permite cambios sin control, pero garantiza que el cambio se realiza sobre el objeto original (p.e. con un código de versión)

Concurrencia en persistencia Control optimista y pesimista



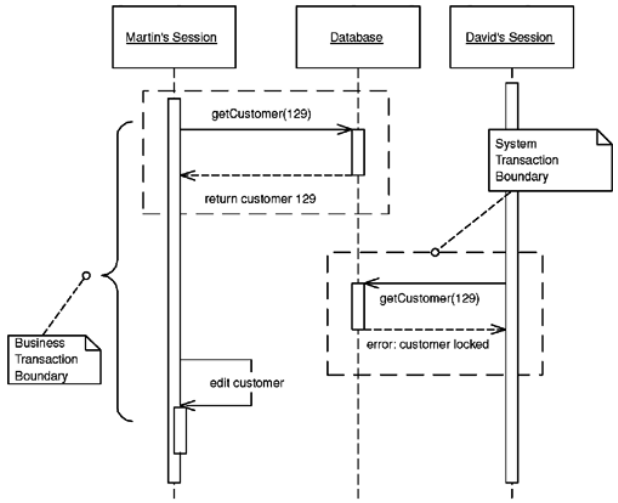
Control de la concurrencia optimista

Concurrencia en persistencia Control optimista y pesimista

- El control de la concurrencia *pesimista* bloquea los datos, evitando modificaciones ajenas al agente que los bloqueo

Concurrencia en persistencia
Control optimista y pesimista

Control pesimista de la concurrencia



Concurrencia en persistencia
Control optimista y pesimista

- A nivel integración, se puede implementar con:
  - `LOCK TABLES … READ`: bloquea escrituras
  - `LOCK TABLES … WRITE`: bloquea lecturas
  - `SELECT … FOR UPDATE`: bloquea escritura de filas
  - `SELECT … LOCK IN SHARE MODE`: bloquea escritura de filas (no ISO SQL)
- Podríamos decir que un control optimista detecta conflictos, y un control pesimista los evita

Concurrencia en persistencia
Control optimista y pesimista

- Nota:
  - Si la hebra 1 hace `LOCK TABLES … READ` , la hebra 2 puede hacer `SELECT`, pero se bloquea si hace `SELECT … FOR UPDATE`
  - Si la hebra 1 hace `LOCK TABLES … WRITE`, la hebra 2 se bloquea si hace `SELECT`
  - Si la hebra 1 hace `SELECT … FOR UPDATE`, la hebra 2 puede hacer `SELECT`, pero se bloquea si hace `SELECT … FOR UPDATE`

Concurrencia en persistencia
Control optimista y pesimista

- Si la hebra 1 hace `SELECT … LOCK IN SHARE MODE`, la hebra 2 puede hacer `SELECT … LOCK IN SHARE MODE`, pero si bloquea si hace `SELECT … FOR UPDATE`

Concurrencia en persistencia
Control optimista y pesimista

- El control pesimista reduce la concurrencia y puede producir bloqueos
  - La hebra A lee la tabla 1 y la bloquea
  - La hebra B lee la tabla 2 y la bloquea
  - La hebra A intenta leer la tabla 2 y no puede
  - La hebra B intenta leer la tabla 1 y no puede

Concurrencia en persistencia
Control optimista y pesimista

- El control optimista puede producir conflictos problemáticos:
  - La hebra A hace una venta de los productos 1, 2 y 3
  - Según va leyendo los productos, otras tres hebras (X, Y, Z) que hacen venta también leen simultáneamente esos productos (1, 2 y 3)
  - La hebra A modifica los tres productos
  - Las otras tres hebras encuentran problemas y tienen que abortar todas sus ventas en curso

Concurrencia en persistencia
Control optimista y pesimista

- Si los conflictos son poco probables o de poca consecuencia, es mejor un control optimista
- En otro caso, es mejor el pesimista

Concurrencia en persistencia
Control optimista y pesimista

- Otro problema es el de las *lecturas inconsistentes*
- El control pesimista puede extenderse a la lectura, para evitarla
- El control optimista fuerza a que exista un marcador de versión para datos compartidos, y a contrastarlo

Concurrencia en persistencia Control optimista y pesimista

- Otra forma de solucionarlo es que el almacen de datos permita *lecturas temporales*, donde los datos van *marcados* con algún tipo de marca temporal o etiqueta inmutable
- La base de datos devuelve el resto de datos tal y como era en el momento de la primera lectura. No es muy común

Concurrencia en persistencia Transacciones

- La principal herramienta para controlar la concurrencia en aplicaciones empresariales son las transacciones
- Una transacción es una agrupación de acciones que debe tener las propiedades *ACID:*

Concurrencia en persistencia Transacciones

- *Atomicity*: cada paso en la secuencia de acciones dentro de una transacción debe completarse con éxito o debe echarse para atrás
- *Consistency*: los recursos de un sistema deben estar en un estado consistente, no corrupto al principio y al final de la transacción
- *Isolation*: el resultado de una transacción no debe ser visible a ninguna otra transacción abierta hasta que la transacción termine con éxito

Concurrencia en persistencia Transacciones

- *Durability*: cualquier resultado de una transacción comprometida debe ser permanente, es decir, debe sobrevivir a un accidente de cualquier tipo
- En general, puede haber más recursos que bases de datos participando en una transacción (colas de mensajes, periféricos, etc.)

Concurrencia en persistencia Transacciones

- Así un *recurso transaccional* es cualquier cosa que utiliza transacciones para controlar la concurrencia (e.g. un SGBDR)
- Hay varios tipos de transacciones:
  - *Transacción larga*: una transacción que se expande a través de múltiples peticiones
  - *Transacción petición*: una transacción que sólo dura durante el procesamiento de la petición

Concurrencia en persistencia Transacciones

  - *Transacción de última hora*: es una transacción de petición que hace todas las lecturas posibles fuera de ella y que la abre para las actualizaciones
- Cuando se hace una transacción se hacen bloqueos y se produce una *escalada de bloqueos*, donde hay múltiples bloqueos de recursos, dificultando el procesamiento

Concurrencia en persistencia Transacciones

- También puede interesar reducir el aislamiento de la transacción para mejorar la viveza
- Si hay pleno aislamiento las transacciones son *serializables*: las transacciones se ejecutan como si no hubiera otras transacciones en marcha

Concurrencia en persistencia Transacciones

- Los problemas asociados al aislamiento son:
  - *Lecturas sucias*: las lecturas de una transacción se pueden ver afectadas por modificaciones no comprometidas de otra
    - Ejemplo: aerolíneas
      - T1 busca asientos libres. Sólo hay uno
      - T1 reserva ese asiento
      - T2 busca asientos libres: no hay
      - T1 intenta hacer el cobro, no hay dinero, y hace rollback
      - No hemos vendido el asiento a nadie

Concurrencia en persistencia
Transacciones

– *Lecturas irrepetibles*: las modificaciones o borrados comprometidos en una transacción pueden ser visibles en otras:
  • Ejemplo: empresa, si hay más de dos proyectos retrasados rebaja el sueldo 5% a empleados de los proyectos y un 10% al responsable del proyecto
    – T1 selecciona los proyectos retrasados: son el 1, 2 y 3
    – T1 baja el 5% a los empleados de esos proyectos
    – T2 da de baja el proyecto 3 y hace commit
    – T1 selecciona los responsables de proyectos retrasados: son el 1 y 2
    – T1 baja 10% a los responsables de los proyectos 1 y 2, pero no el del 3

Concurrencia en persistencia
Transacciones

– *Lecturas fantasma*: las inserciones comprometidas en una transacción pueden visibles a otras en las que el predicado de selección se ve afectado:
  • Ejemplo: empresa, si hay más de dos proyectos retrasados rebaja el sueldo 5% a empleados de los proyectos y un 10% al responsable del proyecto
    – T1 selecciona los proyectos retrasados: son el 1, 2 y 3
    – T1 baja el 5% a los empleados de esos proyectos
    – T2 inserta el proyecto 4, atrasado, y hace commit
    – T1 selecciona los responsables proyectos retrasados: son el 1, 2, 3 y 4
    – T1 baja 10% a los responsables de los proyectos 1, 2, 3 y 4, pero no el sueldo a los empleados del proyecto 4

Concurrencia en persistencia
Transacciones

- En base a esto, los niveles de aislamiento permiten:

	lecturas sucias	**lecturas no repetibles**	**lecturas fantasma**
read uncommitted	posible	posible	posible
read commited	imposible	posible	posible
repeatable read	imposible	imposible	posible
serializable	imposible	imposible	imposible

Niveles de aislamiento

Concurrencia en persistencia
Transacciones

– Cabe recordar que los `update` y `delete` imponen bloqueos
  • Así incluso en un read uncomitted si:
    – T1 hace un `update/delete` de una fila de una tabla
    – T2 intenta hacer un `update/delete` de la misma fila se queda bloqueada hasta que T1 hace un `commit`

Concurrencia en persistencia Transacciones

- En cualquier caso, los niveles de aislamiento son bastante dependientes de los SGBDs
- Así, MySQL 5.1 InnoDB con aislamiento repeatable read no ve las filas fantasma
  - Como dice el manual, un serializable funciona como un repeteable read con shared lock*

*http://dev.mysql.com/doc/refman/5.1/en/set-transaction.html

Concurrencia en persistencia Transacciones

- En un repeatable read:
  - T1 accede a una cuenta con 100 euros
  - T2 accede a la misma cuenta, comprueba el saldo, hace una transferencia de 100 euros, modifica el saldo a 0 y hace commit
  - T1 saca los 100 euros de la cuenta, modifica el saldo a 0 y hace commit

Concurrencia en persistencia Transacciones

- Un serializable es más restrictivo:
  - T1 accede a una cuenta con 100 euros
  - T2 accede a la misma cuenta, comprueba el saldo, hace una transferencia de 100 euros, modifica el saldo a 0 y no puede porque se queda esperando a que termine la transacción T1
  - T1 saca los 100 euros de la cuenta, modifica el saldo a 0 y no puede porque se detecta un deadlock. T1 hace rollback
  - T2 recupera el control modifica el saldo a 0 y hace commit

Concurrencia en persistencia Transacciones

- Sin embargo, repeteable read es lo más usual
- En este escenario es necesario:
  - Estrategia optimista:
    - Comprobar que el número de versión de cuenta no ha variado
  - Estrategia pesimista:
    - Hacer `select … for update`
    - Bloquear la tabla (muy restrictivo)

Concurrencia en persistencia Transacciones

- Hay otro tipo de transacciones:
  - *De sistema*: se llevan a cabo teniendo en mente un sistema concreto (e.g. una venta donde la validación de cliente y stock se hace al cerrarla)
  - *De negocio*: se llevan a cabo teniendo en cuenta una regla de negocio (e.g. una venta donde la validación de cliente y stock se hace al añadir al carrito)

Concurrencia en persistencia Transacciones

- Al final, las transacciones de negocio suelen ser largas (más de una petición) y las del sistema cortas (una única petición)
- Se suele simular una de negocio con varias de sistemas, lo que puede generar el problema de la *concurrencia offfline* ya que las del negocio también deberían ser ACID:

Concurrencia en persistencia Transacciones

- La atomicidad y durabilidad son sencillas (el commit/rollback se hace en cerrar venta)
- Lo difícil de garantizar es:
  - El aislamiento (no puedo garantizar que el cliente siga activo a la hora de cerrar la venta)
  - Esto genera fallos de inconsistencia (puedo hacer una venta a un cliente dado de baja)

Concurrencia en persistencia Transacciones

- Por último, cabe destacar que una transacción de negocio puede afectar a diversos recursos transaccionales distribuidos: tenemos entonces una *transacción distribuida*
- Lo único que cabe en estas situaciones es el *commit de dos fases*:
  - Petición de commit
  - Commit

Concurrencia en persistencia
Concurrencia en el servidor

- Otro problema es la *concurrencia en el servidores web y de aplicaciones*
- No afecta directamente al almacén persistente, pero sí a los datos asociados a las sesiones y/o peticiones

Concurrencia en persistencia
Concurrencia en el servidor

- Hay tres opciones:
  - *Proceso por sesión*: cada sesión es un proceso. Hay aislamiento entre sesiones, pero es muy costoso
  - *Proceso por petición*: cada petición es un proceso. Hay aislamiento entre peticiones, pero sigue siendo costoso
  - *Hebra por petición*: cada petición está gestionada por una hebra. No hay aislamiento entre peticiones, pero es menos costoso

Concurrencia en persistencia
Concurrencia en el servidor

- Lo normal es que sea hebra por petición
- Esto fuerza a que los singleton estén preparados preparados para trabajar en entornos concurrentes

Almacén del dominio

- Propósito
  - Se desea separar la persistencia del modelo de objetos
- También conocido como:
  - Domain store
  - Unit of work + Query object + Data mapper + Table data gateway + Dependent mapping + Domain model + Data transfer object + Identity map + Lazy load

Almacén del dominio

- Motivación
  - Muchos sistemas tienen un modelo de objetos complejo que requiere sofisticadas estrategias de persistencia
  - Estas estrategias deberían ser independientes de los objetos del negocio, para no acoplarlos con un almacén concreto

Almacén del dominio

  - Así, deben resolverse cuatro problemas simultáneos:
    - Persistencia
    - Carga dinámica
    - Gestión de transacciones
    - Concurrencia

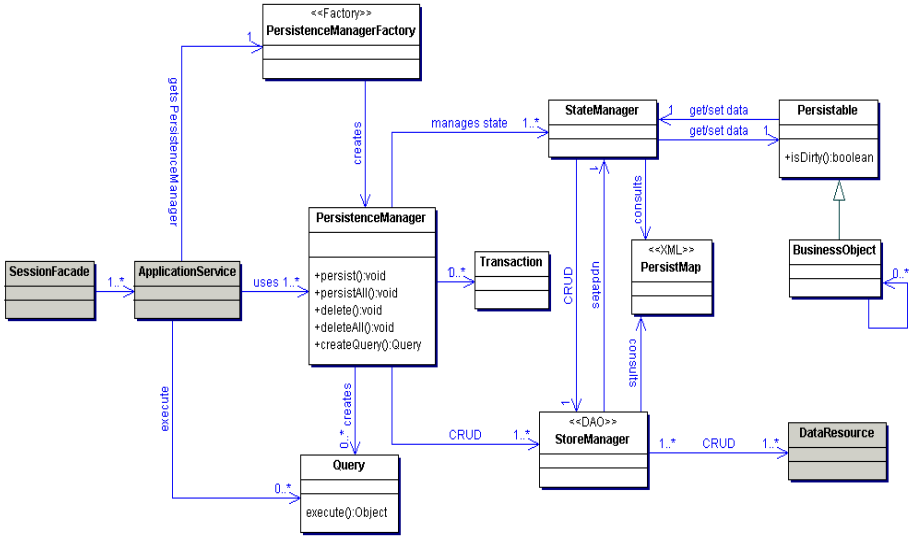
Almacén del dominio

- Contexto
  - Se desea omitir detalles de persistencia en los objetos del negocio
  - La aplicación podría ejecutarse en un contenedor web
  - El modelo de objetos utiliza herencia y relaciones compleja

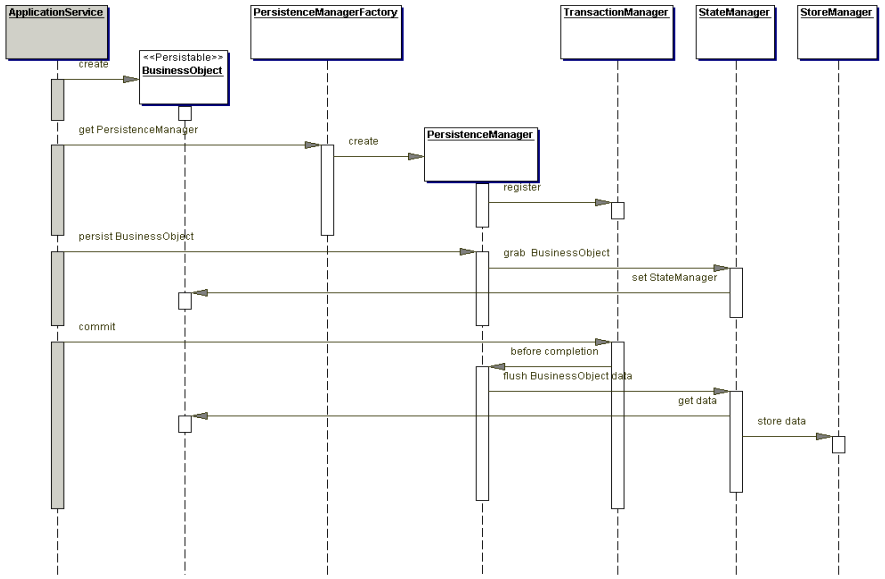
Almacén del dominio

- Solución
  - Utilizar un almacén del dominio para persistir de manera transparente un modelo de objetos

- Descripción

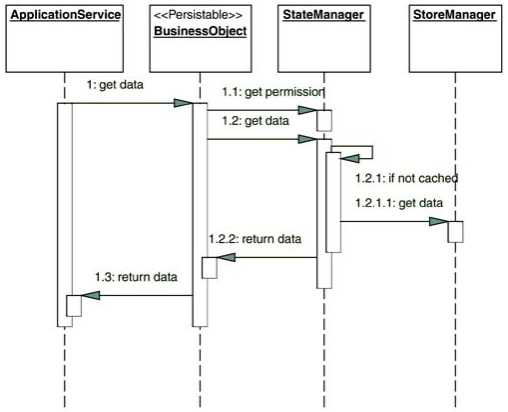


Estructura del patrón almacén del dominio

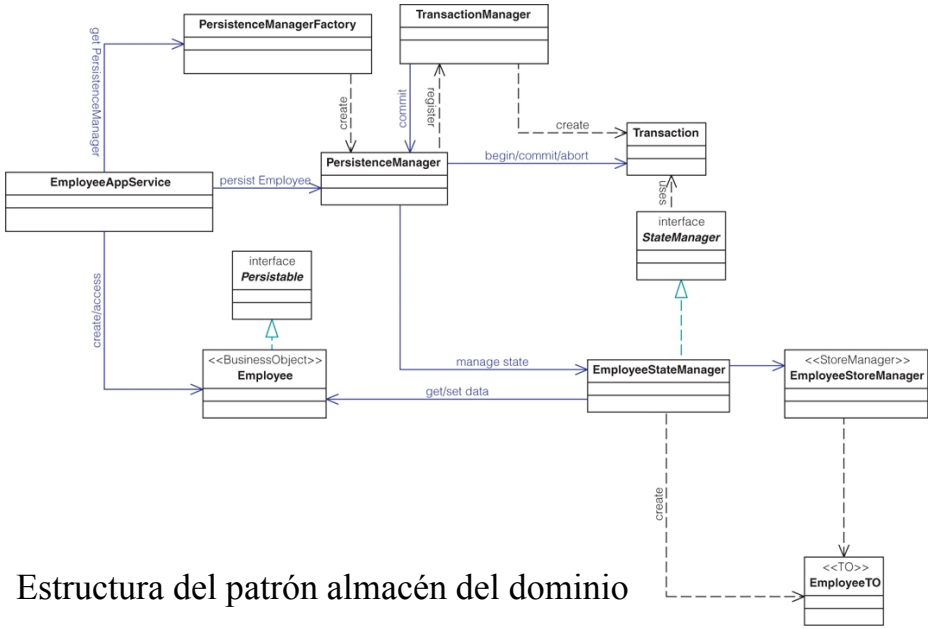


Persistencia de un objeto del negocio

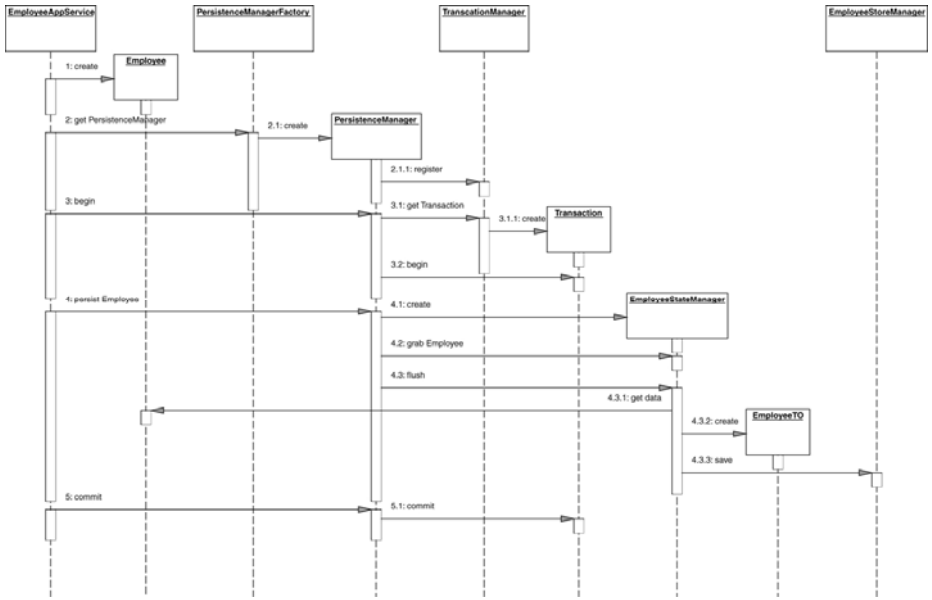
Almacén del dominio



Acceso a atributos simples de un objeto del negocio



Estructura del patrón almacén del dominio



Interacción en el patrón almacén del dominio

Almacén del dominio

- Consecuencias
  - Ventajas:
    - Resuelve:
      - Persistencia
      - Carga dinámica
      - Transacciones
      - Concurrencia
    - Mejora el entendimiento de los marcos de persistencia
    - Mejora la prueba del modelo de objetos persistente
    - Separa el modelo de objetos de negocio de lógica de persistencia

Almacén del dominio

  - Inconvenientes:
    - Crear un marco de persistencia a medida es una tarea compleja
    - La carga y almacenamiento de un árbol de objetos requiere técnicas de optimización
    - Un marco de persistencia en toda regla podría ser excesivo para un modelo de objetos pequeño

Almacén del dominio

- Ejemplo

```
import javax.transaction.*;
public class EmployeeApplicationService {
 public String createEmployee(String lastName,
       String firstName, String ss, float salary,
           String jobClassification, String geography) {
       String id = null;
       String divisionId = null;
       // Create new id
       divisionId =
             getDivisionId(jobClassification, geography);
```

Almacén del dominio

```
// Create Employe
       Employee e = new Employee(id, lastName, firstName,
                                 ss, salary, divisionId);
       PersistenceManagerFactory factory =
              PersistenceManagerFactory.getInstance();
       PersistenceManager manager =
                           factory.getPersistenceManager();
              try {
                     manager.begin();
                     e = (Employee) manager.persistNew(e);
                     manager.commit();
              } catch (SystemException e1) {
```


Almacén del dominio

```
                     } catch (NotSupportedException e1) {
                     } catch (HeuristicRollbackException e1) {
                     } catch (RollbackException e1) {
                     } catch (HeuristicMixedException e1) {
                     }
                     return id;
              }
```


Almacén del dominio

```
public void setEmployeeSalary(String id, float salary)
{
       PersistenceManagerFactory factory =
              PersistenceManagerFactory.getInstance();
       PersistenceManager manager =
                     factory.getPersistenceManager();
       Employee e = manager.getEmployee(id);

       if (e != null) {
                     e.setSalary(salary);
              }
```


Almacén del dominio

```
       try {
                     manager.begin();
                     e = (Employee) manager.persist(e);
                     manager.commit();
              } catch (SystemException e1) {
              } catch (NotSupportedException e1) {
              } catch (HeuristicRollbackException e1) {
              } catch (RollbackException e1) {
              } catch (HeuristicMixedException e1) {
              }
       }
}
```

Almacén del dominio

```
public interface Persistable { }

public class Employee implements Persistable {
       protected String id;
       protected String firstName;
       protected String lastName;
       protected String ss;
       protected float salary;
       protected String divisionId;

       public Employee(String id) {
              this.id = id;
       }
```


Almacén del dominio

```
public Employee(String id, String lastName,
       String firstName, String ss, float salary,
                           String divisionId) {
              this.firstName = firstName;
              this.lastName = lastName;
              this.firstName = firstName;
              this.ss = ss;
              this.salary = salary;
              this.divisionId = divisionId;
       }
```


Almacén del dominio

```
       public void setId(String id) {
              this.id = id; }
       public void setFirstName(String firstName) {
              this.firstName = firstName; }
       public void setLastName(String lastName) {
              this.lastName = lastName; }
       public void setSalary(float salary) {
              this.salary = salary;      }
       public void setDivisionId(String divisionId) {
              this.divisionId = divisionId; }
       public void setSS(String ss) {
              this.ss = ss; }
. . . }
```


Almacén del dominio

```
public class EmployeeStateManager implements
StateManager {
       private final int ROW_LEVEL_CACHING = 1;
       private final int FIELD_LEVEL_CACHING = 2;
       int cachingType = ROW_LEVEL_CACHING;
       boolean isNew;
       private Employee employee;
       private PersistenceManager pm;
```

Almacén del dominio

```
public EmployeeStateManager(PersistenceManager pm,
                    Employee employee, boolean isNew )
{
       this.pm = pm;
       this.employee = employee;
       this.isNew = isNew;
}
```


Almacén del dominio

```
public void flush() {
if (pm.isDirty(employee)) {
       EmployeeTO to = new EmployeeTO(employee.id,
              employee.lastName, employee.firstName,
              employee.ss, employee.salary,
              employee.divisionId);
              EmployeeStoreManager storeManager =
       new EmployeeStoreManager();
       if (isNew) {  storeManager.storeNew(to);
                     isNew = false;
                   } else { storeManager.update(to);  }
       pm.resetDirty(employee);
       }
}
```


Almacén del dominio

```
public void load() {
       EmployeeStoreManager storeManager =
                                  new EmployeeStoreManager();
       EmployeeTO to = storeManager.load(employee.id);
       updateEmployee(to);
}
```


Almacén del dominio

```
public void load(int field) {
if (fieldNeedsReloading(field)) {
       EmployeeStoreManager storeManager =
       new EmployeeStoreManager();
       if (cachingType == FIELD_LEVEL_CACHING) {
              Object o =
       storeManager.loadField(employee.id, field);
               updateEmployee( field, o );
       } else { EmployeeTO to =
                     storeManager.load(employee.id);
                updateEmployee(to);
              }
       }
}
```

Almacén del dominio

```
private boolean fieldNeedsReloading(int field) {
// Caching and valid data rule apply here
// data can be cached at the field or the row level
switch (field) {
  case EmployeeStateDelegate.LAST_NAME:
            if (employee.lastName == null) return true;
                        break;
   case EmployeeStateDelegate.FIRST_NAME:
            if (employee.firstName == null) return true;
                        break;
```


Almacén del dominio

```
case EmployeeStateDelegate.DIVISION_ID:
                    String did = employee.divisionId;
             if (did == null || did.indexOf("99-") == -1)
                    return true;
                    break;
case EmployeeStateDelegate.SS:
            if (employee.ss == null) return true;
                        break;
case EmployeeStateDelegate.SALARY:
             if (employee.salary == 0.0) return true;
             break;
             }
return false;       }
```


Almacén del dominio

```
private void updateEmployee(EmployeeTO to) {
             employee.id = to.id;
             employee.lastName = to.lastName;
             employee.firstName = to.firstName;
             employee.ss = to.ss;
             employee.salary = to.salary;
             employee.divisionId = to.divisionId;
             isNew = false; }
public boolean needsLoading() {
             if (pm.needLoading(employee)) return true;
             else return false;
       }
}
```


Almacén del dominio

```
public class EmployeeStateDelegate extends Employee {
     static final int LAST_NAME = 1;
     static final int FIRST_NAME = 2;
     static final int SS = 3;
     static final int SALARY = 4;
     static final int DIVISION_ID = 5;

     private EmployeeStateManager stateManager;
```

Almacén del dominio

```
    public EmployeeStateDelegate(String id, String
lastName, String firstName, String ss, float salary,
                          String divisionId) {
super(id, lastName, firstName, ss, salary, divisionId);
  }
    public EmployeeStateDelegate(Employee e) {
      super(e.id, e.lastName, e.firstName,
           e.ss, e.salary, e.divisionId);
    }

    public EmployeeStateDelegate(String employeeId) {
      super(employeeId);
    }
```


Almacén del dominio

```
public EmployeeStateDelegate(String employeeId,
      EmployeeStateManager stateManager) {
   super(employeeId);
   this.stateManager = stateManager;
 }
 public void setStateManager(
      EmployeeStateManager stateManager) {
   this.stateManager = stateManager;
 }
 public String getFirstName() {
   stateManager.load(FIRST_NAME);
   return firstName;
 }
```


Almacén del dominio

```
    public String getDivisionId() {
      stateManager.load(DIVISION_ID);
      return divisionId;
    }
    public String getLastName() {
      stateManager.load(LAST_NAME);
      return lastName;
    }
    public String getSS() {
      stateManager.load(SS);
      return ss;
    }
```


Almacén del dominio

```
public float getSalary() {
    stateManager.load(SALARY);
    return salary;
  }

  public EmployeeStateManager getStateManager() {
    return stateManager;
  }
}
```

Almacén del dominio

```
public class EmployeeTO {
       public String id;
       public String lastName;
       public String firstName;
       public String ss;
       public float   salary;
       public String divisionId;
```


Almacén del dominio

```
public EmployeeTO(String id, String lastName,
                  String firstName, String ss,
                  float salary, String divisionId ) {
              this.id = id;
              this.lastName = lastName;
              this.firstName = firstName;
              this.ss = ss;
              this.salary = salary;
              this.divisionId = divisionId;
       }
}
```


Almacén del dominio

```
public class EmployeeStoreManager {
public void storeNew(EmployeeTO to) {
String sql = "Insert into Employee( id, last_name," +
             " first_name, ss, salary, division_id ) " +
             " values( '?', '?', '?', '?', '?', '?' )";
             . . .
      }

public void update(EmployeeTO to) {
String sql = "Update Employee set last_name = '?'," +
             " first_name = '?', salary = '?'," +
             " division_id = '?' where id = '?'";
             . . . }
```


Almacén del dominio

```
public void delete(String empId) {
String sql = "Delete from Employee where id = '?'";
             . . .
      }

public EmployeeTO load(String empId) {
      . . .
      }
}
```

Almacén del dominio

```
public class PersistenceManagerFactory {
static private PersistenceManagerFactory me = null;
public synchronized static PersistenceManagerFactory
getInstance() {
       if (me == null) {
                    me = new PersistenceManagerFactory();
       }
       return me;
       }
private PersistenceManagerFactory() { }
public PersistenceManager getPersistenceManager() {
              return new PersistenceManager();
       }
}
```


Almacén del dominio

```
import javax.transaction.*;
import java.util.HashSet;
import java.util.Iterator;

public class PersistenceManager {
       HashSet stateManagers = new HashSet();
       TransactionManager tm;
       Transaction txn;

       public PersistenceManager() {
              tm = TransactionManager.getInstance();
              tm.register(this);
       }
```


Almacén del dominio

```
public Persistable persistNew(Persistable o) {
       if (o instanceof Employee) {
              return setupEmployee(
new EmployeeStateDelegate((Employee)o), true);
              }
       return o;
       }

public Persistable persist(Persistable o) {
// Must already be an EmployeeStateDelegate
if (o instanceof Employee) {
       EmployeeStateDelegate esd =(EmployeeStateDelegate) o;
                                    return esd;       }
return o; }
```


Almacén del dominio

```
public void commit() throws SystemException,
NotSupportedException, HeuristicRollbackException,
RollbackException, HeuristicMixedException {
       if (txn == null) {
                    throw new SystemException(
                    Must call Transaction.begin() before" +
                                  " Transaction.commit()"); }
       Iterator i = stateManagers.iterator();
       while (i.hasNext()) {
              Object o = i.next();
              StateManager stateManager = (StateManager) o;
              stateManager.flush();
              }
              txn.commit();
              txn = null;  }
```

Almacén del dominio

```
public void begin()throws SystemException,
NotSupportedException {
            txn = tm.getTransaction();
            txn.begin();
      }

public Employee getEmployee(String employeeId) {
      EmployeeStateDelegate esd =
            new EmployeeStateDelegate(employeeId);
            setupEmployee(esd, false);
            return esd;
      }
```


Almacén del dominio

```
private EmployeeStateDelegate setupEmployee(
            EmployeeStateDelegate esd, boolean isNew)
{
      EmployeeStateManager stateManager =
      new EmployeeStateManager(this, esd, isNew);
      stateManagers.add(stateManager);
      esd.setStateManager(stateManager);
      return esd;
      }

public void setDirty(Persistable o) {
            // set dirty marker to true
      }
```


Almacén del dominio

```
public void resetDirty(Persistable o) {
            // reset dirty marker to false
      }

public boolean isDirty(Persistable o) {
            // check if object is dirty
            return true;
      }

public boolean needLoading(Persistable o) {
            // check if needs to be loaded
            return true;
      } }
```


Almacén del dominio

```
import javax.transaction.*;
import java.util.Iterator;
import java.util.LinkedList;
public class TransactionManager {
      static TransactionManager me = null;
      private LinkedList persistenceManagers = new
LinkedList();
      class PManager {
            Thread thread;
            PersistenceManager manager;
```

Almacén del dominio

```
PManager(Thread thread, PersistenceManager manager) {
        this.thread = thread;
        this.manager = manager;
    }
    boolean equals(Thread thread,
    PersistenceManager manager) {
        if (this.thread == thread &&
                this.manager == manager) {
            return true;
        }
        return false;
    }
}
```


Almacén del dominio

```
public synchronized static TransactionManager getInstance()
{
        if (me == null) {
            me = new TransactionManager();
        }
        return me;
    }

private TransactionManager() { }

public Transaction getTransaction() {
        return new Transaction();
    }
```


Almacén del dominio

```
public void register(PersistenceManager manager) {
        . . . }

public void notifyCommit(Thread t)
        throws SystemException,
            HeuristicRollbackException,
            NotSupportedException,
            RollbackException,
            HeuristicMixedException {
    Iterator i = persistenceManagers.iterator();
    while (i.hasNext()) {
        . . .
        pm.manager.commit();
    }   }   }
```


Almacén del dominio

```
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.transaction.*;

public class Transaction {
    UserTransaction txn;

public void setSessionContext( SessionContext ctx ) {
        ctx.getUserTransaction();
    }
```

Almacén del dominio

```
public Transaction() {
            InitialContext ic = null;
            try {
                  ic = new InitialContext();
      txn =
(UserTransaction)ic.lookup("java:comp/UserTransaction")
;
            } catch (NamingException e) {
            }     }

public void begin() throws SystemException,
                  NotSupportedException {
            txn.begin(); }
```


Almacén del dominio

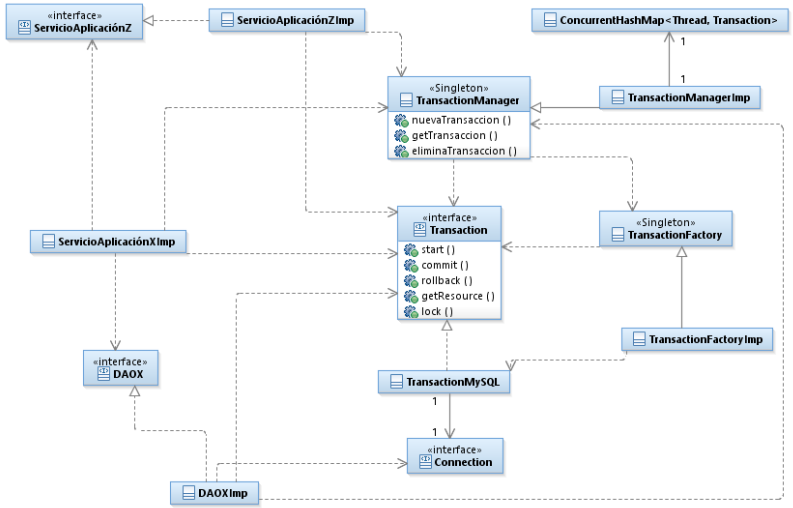
```
public void commit()
                  throws SystemException,
                        HeuristicRollbackException,
                        RollbackException,
                        HeuristicMixedException {
            txn.commit();
      }

public void rollback() throws SystemException {
            txn.rollback();
      }
}
```


Almacén del dominio

- Nota
  - Podemos utilizar un transaction manager aún en el caso de que sólo utilicemos DAOs y transfers, es decir, sin objetos del negocio

Almacén del dominio



Gestión de transacciones sin objetos del negocio

Almacén del dominio

- En el caso de que un servicio de aplicación que ha iniciado una transacción llame a otro que también la inicie, podemos:
  - Lanzar una excepción
  - Tener dos distintas, lo que forzaría a poder guardar más de una transacción por hebra y relacionarla con su servicio de aplicación
  - Usar la misma, para lo cual la transacción debería llevar un contador de starts iniciados, y un booleano que le indicase si todas han hecho commit

Almacén del dominio

- Con respecto al DAO, si es invocado, y no hay transacción en marcha, podría:
  - Lanzar una excepción
  - Crear una conexión propia, usarla y cerrarla
    - Las conexiones en el contexto de una transacción podrían hacer `SELECT … FOR UPDATE`
    - Las conexiones sin transacción podrían hacer simplemente `SELECT`
    - Todo esto supuesto que no se hayan bloqueado las tablas al principio de la transacción, en cuyo caso sólo se haría `SELECT`

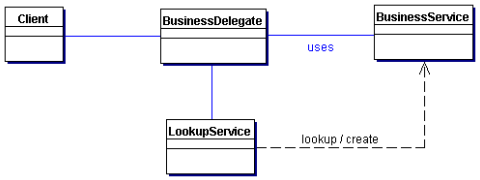
Patrón delegado del negocio

- Propósito
  - Evita que los clientes tengan que tratar con detalles de acceso a componentes distribuidos en una aplicación multicapa
- También conocido como:
  - *Business delegate*

Patrón delegado del negocio

- Motivación
  - Cuando se implementa la capa de negocio con componentes distribuidos los clientes de dicha capa (p.e. la capa de presentación) tienen que tratar con detalles de conexión y acceso al servidor de aplicaciones
  - El patrón delegado se encarga de estos detalles, permitiendo que los clientes se abstraigan de la implementación del negocio

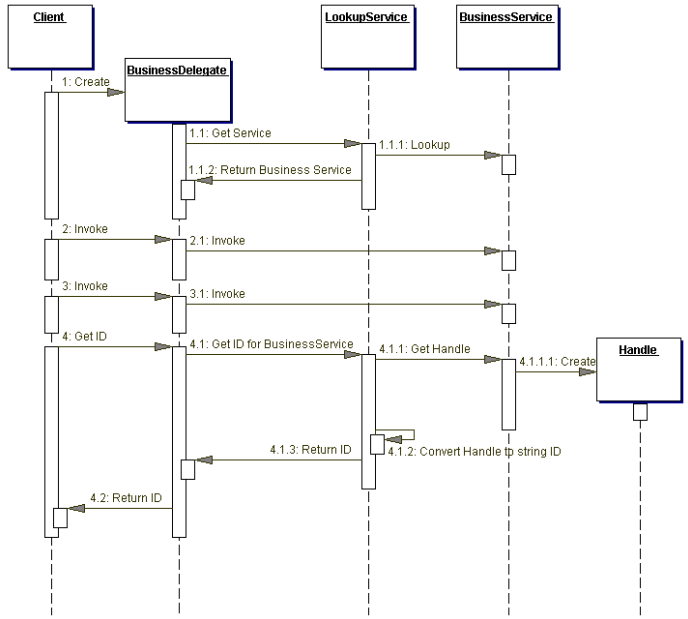
Patrón delegado del negocio



Estructura del patrón delegado

P. delegado del negocio

Interacción entre objetos relacionados por el patrón delegado



Patrón delegado del negocio

- Consecuencias
  - Ventajas
    - Oculta detalles
    - Independiza capas
  - Inconvenientes
    - Introduce otra nivel más de indirección

Patrón delegado del negocio

- Notas
  - Las aplicaciones desarrolladas en esta asignatura no tienen porque utilizar componentes distribuidos (p.e., *Enterprise Java Beans*, EJBs*).
  - El patrón delegado del negocio se introduce por motivos de completitud con respecto a la descripción de aplicaciones de tres capas, las cuales suelen estar ligadas a componentes distribuidos

*http://java.sun.com/products/ejb/

Patrón delegado del negocio

– En el contexto de esta asignatura, en lugar de delegados del negocio podemos utilizar fachadas

– Aunque su funcionalidad es distinta (abstraer detalles de conexión y acceso vs. abstraer detalles sobre componentes involucrados), el patrón fachada encapsulará todos los detalles de implementación a los clientes del negocio

Patrón delegado del negocio

- Código de ejemplo

```
public class StockListDelegate {
  private StockList stockList;

  //accede al objeto fachada de la aplicación
  private StockListDelegate() throws StockListException
  {
      try { InitialContext ctx= new InitialContext();
            stockList=
  (StockList) ctx.lookup(StockList.class.getName());
           } catch(Exception e) {
             throw
            new StockListException(e.getMessage());
            }
  }
```


Patrón delegado del negocio

```
.....
//stock es un objeto transferencia para las acciones
public void addStock(StockTO stock) throws
  StockListException {

  //delega en la fachada
  try { stockList.add(stock); }
  catch (Exception re) {
      throw new StockListException (re.getMessage());
      }
}
.....
```


Patrón delegado del negocio

```
.....
//el delegado en un singleton
public static StockListDelegate getInstance()
  throws StockListException {

  if (stockListDelegate == null)
      stockListDelegate= new StockListDelegate();

  return stockListDelegate;
}
}
```

Patrón delegado del negocio

```
//interfaz remoto de la fachada
@Remote
public interface StockList {

  public List getStockRatings();
  public List getAllAnalysts();
  public List getUnratedStocks();
  public void addStockRating(StockTO stockTO);
  public void addAnalystAnalystTO analystTO);
  public void addStock(StockTo stockTO);
}
```


Patrón delegado del negocio

```
//implementación de la fachada como EJB de sesión
//sin estado
@Stateless
public class StockListBean implements StockList
{
   ..........................................

}
```


Nota

- Doble estructura de paquetes:
  - Capas
  - Módulos

	presentación	negocio	integración
usuarios			
ejemplares			
préstamos			
búsquedas			

- Subsistemas de diseño/paquetes de código: dirigidos por capas, con módulos replicados
  - presentacion
    - usuarios
    - ejemplares
    - prestamos
    - busquedas
  - negocio
    - usuarios
    - ejemplares
    - prestamos
    - busquedas
  - integracion
    - usuarios
    - ejemplares
    - prestamos
    - busquedas

Conclusiones

- Los sistemas de información son bastante relevantes hoy en día
- Este tema se centra en el diseño arquitectónico de sistemas de información/aplicaciones empresariales
- No entra en detalles internos de cada componente
- Técnicas útiles para cualquier aplicación software

Conclusiones

- Patrones: estructuras reutilizables
- MVC: arquitectura de presentación mantenible
- Factoría abstracta: cliente independiente de la implementación de interfaces
- Fachada: punto de acceso a un conjunto de operaciones separadas en varios objetos

Conclusiones

- Singleton: acceso global sin necesidad de creación + redefinición de operaciones no estáticas
- Arquitectura de una capa: ¿sencillo? e inmantenible
- Arquitectura de dos capas: sencillo y mantenible a nivel cambios de interfaz
- Arquitectura multicapa: ¿sencillo? y mantenible a nivel cambios en cualquier capa

Conclusiones

- Transfer: envío de datos entre capas
- DAO: independencia entre negocio y datos
- Delegado del negocio: independencia entre clientes y plataformas
- Servicio de la aplicación: lógica del negocio

Conclusiones

- Objeto del negocio: modelo de objetos en arquitectura multicapa
- Almacén del dominio: persistencia independiente de los objetos del negocio