Tema 13 –
Patrones arquitectónicos de capas

Ingeniería del Software

Rubén Fuentes Fernández
Dep. Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense Madrid

Trabajando con Antonio Navarro, Juan Pavón y Pablo Gervás

Contenidos (1/2)

- Introducción
  - Patrones y arquitecturas
  - Sistemas de información
- Patrones de diseño básicos
  - Modelo-Vista-Controlador
  - Fachada
  - Factoría abstracta
  - *Singleton*
- Arquitecturas
  - Arquitectura de una capa
  - Arquitectura de dos capas
  - Arquitectura multicapa

Contenidos (2/2)

- Patrones de diseño en arquitectura multicapa
  - *Controlador frontal*
  - *Controlador de aplicación*
  - Transferencia
  - *Data Access Object*
  - *Delegado del negocio*
  - Servicio de aplicación
  - *Objeto del negocio*
  - *Almacén del dominio*

Los patrones en gris no es necesario aplicarlos en la práctica.

Patrón

- "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" [Alexander et al., 1977]
  - A propósito de arquitectura, ciudades y urbanismo.
  - Fuerte impacto en la comunidad de Ingeniería del Software y, en particular, en la Orientación a Objetos (OO).
    - Patrones de diseño
    - Patrones arquitectónicos o arquitecturas

Patrón de diseño

- Los *patrones de diseño*:
  - Son soluciones simples y elegantes a problemas específicos del diseño de software.
  - Representan soluciones que han sido desarrolladas y evaluadas, y han ido evolucionando a través del tiempo.
    - Encapsulan el conocimiento de diseñadores experimentados.
- El *patrón de diseño* describe múltiples aspectos de su solución:
  - Aspectos clave del diseño
  - Requisitos de aplicabilidad → contexto
  - Ventajas e inconvenientes → compromisos (*tradeoffs*)
  - Compromisos que implica entre distintas características

Patrón de diseño OO

- Los *patrones de diseño OO* describen su solución mediante:
  - Las clases e instancias participantes
  - Los roles y colaboraciones de dichas clases e instancias
  - La distribución de responsabilidades
- No consideran hasta los últimos detalles de la solución.
  - Ej. estructuras de datos subyacentes, les basta con las interfaces y sus restricciones
- Han de ser adaptados para dominios y problemas específicos.
  - Siempre dentro de los válidos según el contexto de aplicación.

Arquitectura

- La *arquitectura* es la organización fundamental de un sistema, expresada en sus componentes, sus relaciones entre ellos, y en el entorno y principios que guían su diseño y evolución [IEEE, 2000].
  - La arquitectura no se ocupa de proporcionar buenos diseños a nivel de componentes individuales.
  - De ello se ocupan disciplinas como:
    - Interacción persona-computadora
    - Programación
    - Patrones de diseño
    - Bases de datos
- Las *arquitecturas* incluyen frecuentemente múltiples *patrones de diseño*.

Sistemas de información

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

Desarrollo de sistemas de información

- Los sistemas de información tienen una gran relevancia en informática.
- Su desarrollo ha de contemplar múltiples aspectos [ACM, 2012]:
  - Modelos y principios
  - Gestión de bases de datos
  - Almacenamiento y recuperación de información
  - Aplicaciones de sistemas de información
  - Interfaces y presentación de la información

Aplicaciones empresariales

- Una *aplicación empresarial* es un sistema de información con las siguientes características:
  - Maneja una gran cantidad de datos persistentes.
  - Estos datos son accedidos concurrentemente.
  - Hay una gran cantidad de lógica del negocio.
    - Representa la funcionalidad de la aplicación.
  - El acceso se produce a través de elaboradas interfaces de usuario.
  - Suelen tener necesidades de integración con otras aplicaciones empresariales de arquitectura heterogénea.

Arquitecturas para sistemas de información

- La descripción de arquitecturas se centrará en *arquitecturas multicapa*.
  - También llamadas últimamente *patrones arquitectónicos de aplicaciones empresariales*.
- Son arquitecturas especialmente indicadas para:
  - sistemas de información, sobre todo aplicaciones empresariales,
  - que trabajan sobre la web.
- Lo visto es aplicable también a otros tipos de sistemas.

PATRONES DE DISEÑO BÁSICOS

Patrones de diseño básicos

- Son una serie de patrones de uso común en la mayor parte de las arquitecturas multicapa.
  - Se usan para cubrir diferentes funcionalidades en las mismas.
- En este sección veremos los siguientes patrones:
  - Modelo-Vista-Controlador
  - Fachada
  - Factoría abstracta
  - *Singleton*

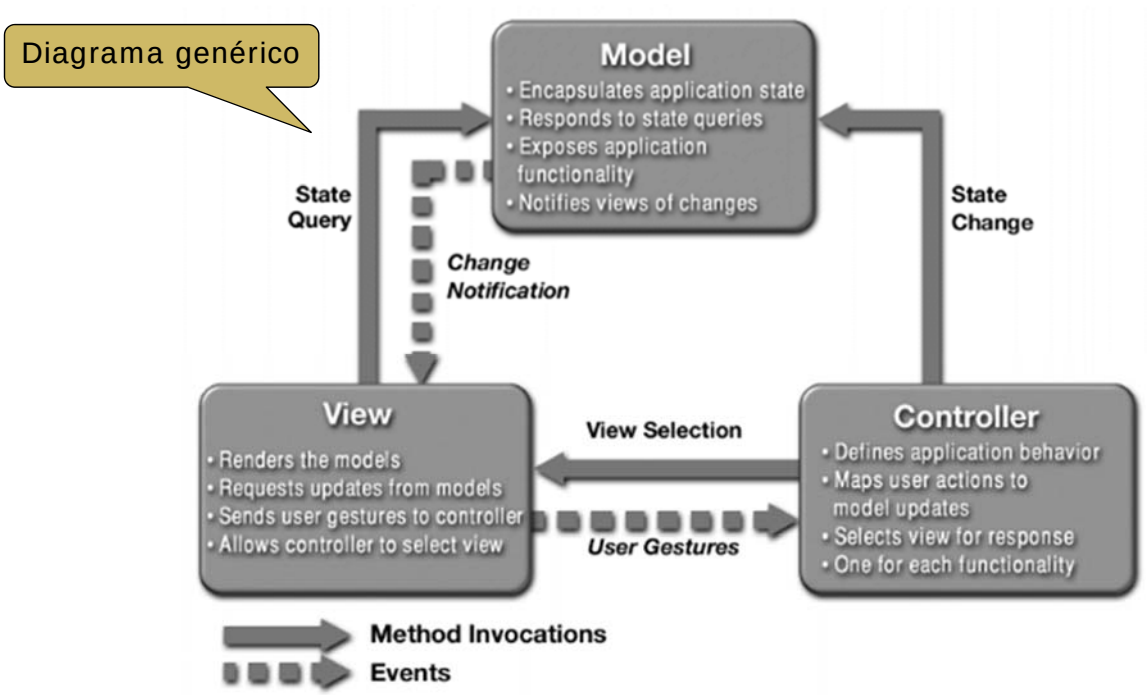
Patrones de diseño

MODELO-VISTA-CONTROLADOR

Modelo-Vista-Controlador

- El patrón de diseño *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*.
- El MVC se considera frecuentemente tanto *patrón de diseño* como *arquitectura*.
  - El MVC es *patrón de diseño* porque puede ofrecer una solución organizativa de una parte interactiva de una aplicación.
  - Esta solución se extiende frecuentemente a todas las partes interactivas de una aplicación y se habla de *arquitectura*.

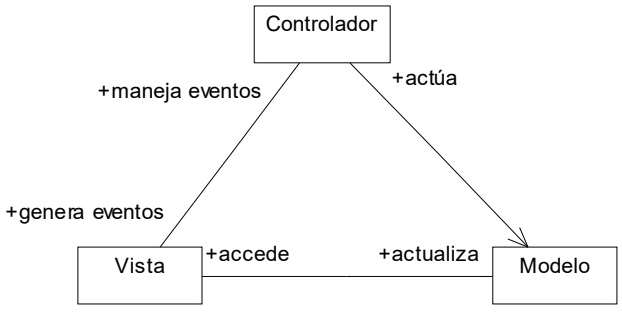
MVC: diagrama de bloques



MVC: variantes

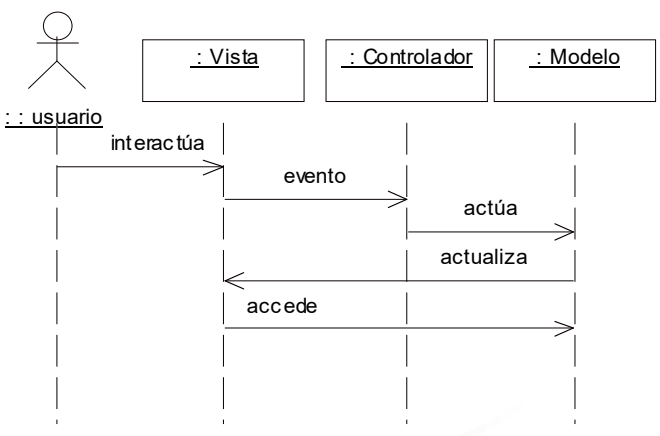
- Sobre el modelo:
  - Modelo activo
    - El modelo es responsable de disparar la actualización de las vistas.
  - Modelo pasivo
    - El modelo sólo responde invocaciones.
    - El controlador es responsable de disparar la actualización de las vistas.
- Sobre el controlador:
  - Un controlador por evento
  - Uno por conjuntos de funcionalidades / estímulos
  - Uno por aplicación

MVC: modelo activo

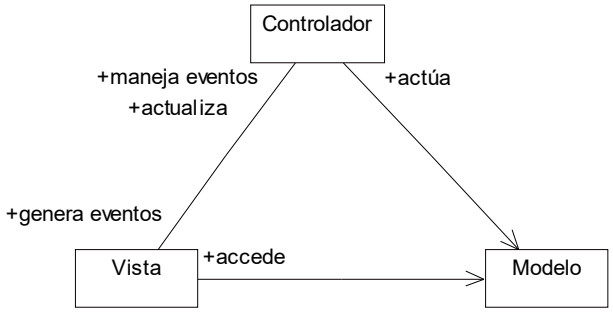


Participantes

Interacción

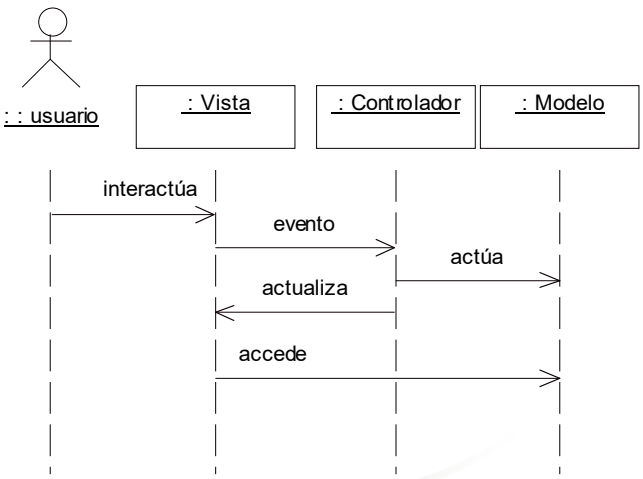


MVC: modelo pasivo



Participantes

Interacción



Ejemplo: MVC caso 1 – interfaz *vista*

- Modelo activo sin utilizar `java.util.Observer`.

```
public interface IVista {
   void actualizar(Object actualizado);
}
```

Una vista se caracteriza por implementar la interfaz *Ivista*.

El método *actualizar* permite notificar a la vista cambios en el modelo.

Ejemplo: MVC caso 1 – controlador

```
class Controlador implements ActionListener{
   ...
   public void actionPerformed (ActionEvent e)
   {
       modelo.sumar();
   }
}
```

El controlador se ocupa de actualizar el modelo según los cambios (eventos) que le indican las vistas.

En Java, los controladores suelen implementar una interfaz de tipo *Listener*, que incluye métodos a los que las vistas invocan cuando se produce el evento, por ejemplo *actionPerformed*.

El controlador es avisado desde las vistas cuando se produce algún cambio mediante este método.

Ejemplo: MVC caso 1 – modelo

```
class Modelo {
   int valor;
   IVista vista;
   ...
   void sumar() {
      valor++;
       vista.actualizar(this);
   }
   int obtenerValor(){
      return valor;
   }
}
```

El modelo implementa métodos para actualizar la representación de la información.

El *modelo activo* mantiene una lista de vistas a las que avisar. Es posible que distintas vistas sean avisadas de eventos diferentes. Se dice que las vistas están *suscritas* a determinados eventos.

Como parte de la actualización de la información, el *modelo activo* notifica a las vistas relevantes el cambio que se ha producido.

Ejemplo: MVC caso 1 – vista (1/2)

```
class Vista extends JFrame implements IVista {
   JTextField valor;
   JButton sumar;

   public Vista(Modelo modelo) {
       ...
       sumar= new JButton ("+");
       ActionListener controlador =
                          new Controlador(modelo);
       sumar.addActionListener(controlador);
   }
```

Vinculación de un cambio en la vista (en uno de sus botones) al controlador: cuando se produzca el evento, la vista avisará al controlador y le pasará la información sobre el cambio.

Creación del controlador y vinculación al modelo.

Ejemplo: MVC caso 1 – vista (2/2)

```
   public void actualizar(Object o){
       Modelo modelo = (Modelo) o;
       Integer i = new
          Integer(modelo.obtenerValor());
       valor.setText(i.toString());  }
       ...
    }
...
}
```

El método *actualizar* se invoca con información sobre el cambio al que debe responder la vista. El parámetro no tiene porque ser el modelo.

El método *actualizar* cambia la vista según el cambio indicado.

En el MVC con *modelo activo*, el modelo invoca al método *actualizar*.

Ejemplo: MVC caso 1 – comentarios

- En este ejemplo:
  - La vista que envía los eventos al controlador es la misma que recibe las actualizaciones del modelo/controlador.
  - Coinciden el controlador de eventos de interfaz y el controlador de eventos del negocio.
- En general, esto no tiene porque ser así.
  - La vista avisada de los cambios puede ser diferente de la que los originó.
  - Se pueden tener múltiples controladores para distintos aspectos de la aplicación.
    - Ej. en aplicaciones web

Ejemplo: MVC caso 2 – interfaz *vista*

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

   void actualizar(int evento, Object datos);
}
```

Como *IVista* del caso 1.

Ejemplo: MVC caso 2 – controlador (1/2)

```
public class Controlador {
  ...
  private Biblioteca biblioteca;
  private IGUI gui;

  // implementación naif de una tabla de controlador
  public void accion(int evento, Object datos) {
    switch (evento){
      case EventoNegocio.INSERTAR_USUARIO: {... }
      ...
```

El modelo y la vista gestionados por el controlador

Ejemplo: MVC caso 2 – controlador (2/2)

```
  public void accion(int evento, Object datos) {
    switch (evento){
      ...
      case EventoNegocio.BAJA_USUARIO: {
        Integer id= (Integer) datos;
        Boolean resultado =
  biblioteca.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;
  }
```

Indicación del dato afectado más que del modelo

Con el modelo pasivo, el controlador gestiona los avisos a las vistas a través del método *actualizar*.

Ejemplo: MVC caso 2 – evento

```
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;
}
```


Ejemplo: MVC caso 2 – vista (1/4)

```
public class GUIBiblioteca extends JFrame implements
  IGUI  {

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


Ejemplo: MVC caso 2 – vista (2/4)

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

El botón *aceptar* será la parte de la vista que genere los eventos que se notifican al controlador.

Ejemplo: MVC caso 2 – vista (3/4)

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

Al pulsar el botón *aceptar* se avisa al controlador (Java *ActionListener*) invocando su método correspondiente (Java *actionPerformed*).

El método *accion* está en la diapositiva del controlador.

El método *getInstancia* es estático de la clase *Controlador*. Corresponde a la implementación de un patrón *singleton* que veremos más adelante.

Ejemplo: MVC caso 2 – vista (4/4)

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

La vista reacciona a diferentes tipos de eventos (parámetro *evento*).

Ventajas e inconvenientes

- 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 y lógica.
- Inconvenientes
  - Complejidad.

Patrones relacionados

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

Patrones de diseño

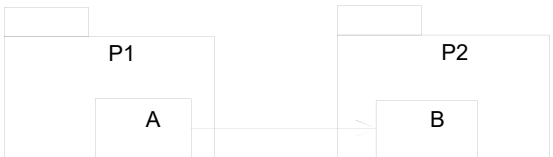
FACHADA

Fachada

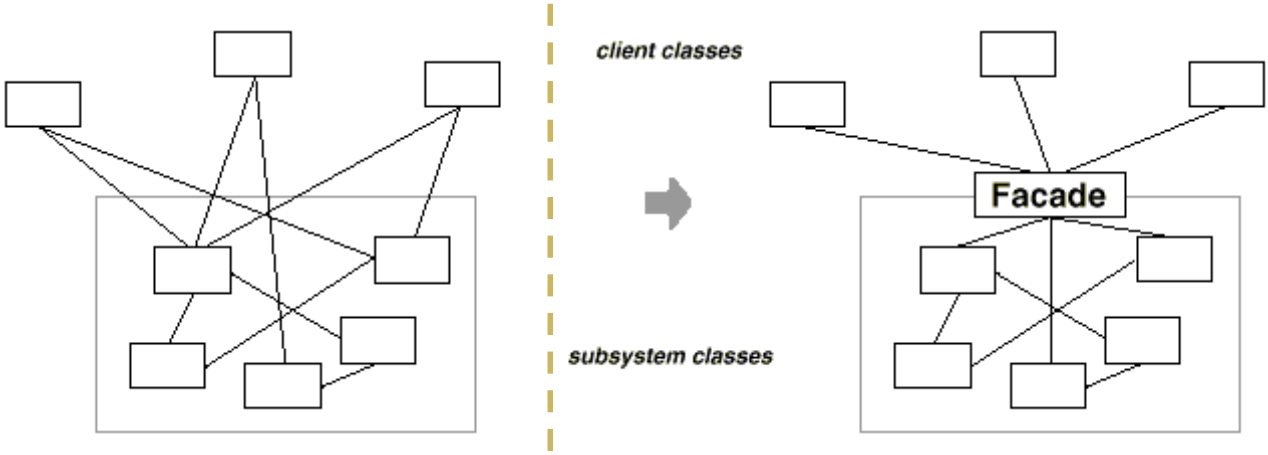
- El patrón de diseño *fachada* (*façade*) [Gamma et al., 2008] proporciona una interfaz unificada para un conjunto de interfaces de un subsistema.
  - Define una interfaz de alto nivel para que el subsistema sea más fácil de utilizar.
  - Cuando hablamos de subsistemas, nos referimos, en general, a los *subsistemas de diseño*.
- Los *subsistemas de diseño* deben estar formados por subsistemas cohesivos con bajo acoplamiento.
  - Las interfaces evitan el acoplamiento.
  - Los subsistemas de diseño se plasman como paquetes.
- Ahora, no solamente utilizamos interfaces, sino además una interfaz de acceso a interfaces y/o clases → la *fachada*.
  - Cada subsistema debe implementar sus responsabilidades

Dependencias entre paquetes

- Subsistemas de diseño



Fachada: diagrama de clases



[Gamma et al., 2008]

Subsistema original

Subsistema con *fachada*

Ejemplo: fachada – interfaz

Interfaz *fachada* con los servicios mostrados por el subsistema.

```
public interface Biblioteca {
  public Integer insertaUsuario(TUsuario usuario);
  public Boolean daDeBajaUsuario(Integer id);
  public TDAOUsuario obtenUsuario(Integer id);
  public Integer insertaPublicacion(TPublicacion
  publicacion);
  public Boolean daDeBajaPublicacion(Integer id);
  public TDAOPublicacion obtenPublicacion(Integer
  id);
  public TPrestamo prestamo(TPrestamo tPrestamo);
  public Boolean devolucion(Integer ejemplar);
}
```


Ejemplo: fachada – subsistema

```
public class BibliotecaImp implements Biblioteca {
  public Integer insertaUsuario(TUsuario usuario)
  { ... }

  public Boolean daDeBajaUsuario(Integer id)
  {   //nótese que no se ha hecho explícito
      //el acceso a estos servicios por parte de la
      //Biblioteca
      serviciosUsuario.daDeBajaUsuario(id);
  }
  ...
}
```

El subsistema implementa las responsabilidades descritas por su fachada. Aquí mediante una clase que invoca los servicios de otros componentes del subsistema como *serviciosUsuario*.

Ventajas e inconvenientes

- Ventajas:
  - Oculta a los clientes los componentes del subsistema.
    - Reduce el número de objetos con los que tratan los clientes.
    - De esta forma el subsistema es más fácil de utilizar.
  - Promueve un débil acoplamiento entre el subsistema y los clientes.
    - Al introducir la fachada podemos modificar los componentes del subsistema sin afectar a los clientes.
    - Esto permite implementaciones independientes de los subsistemas.
  - No impide que las aplicaciones utilicen las clases del subsistema en caso necesario.
- Inconvenientes:
  - Al incluir nuevas operaciones en el subsistema, hay que actualizar la fachada.

Patrones relacionados

- Si la dependencia entre subsistemas de diseño (paquetes) es una asociación navegada, la responsabilidad de crear los elementos accedidos no debe recaer sobre el objeto que los accede.
  - El objeto accedido ya ha de estar creado y se pasa al que lo accede como parámetro.
  - La creación del objeto accedido se delega en otro objeto con el patrón *factoría abstracta*.

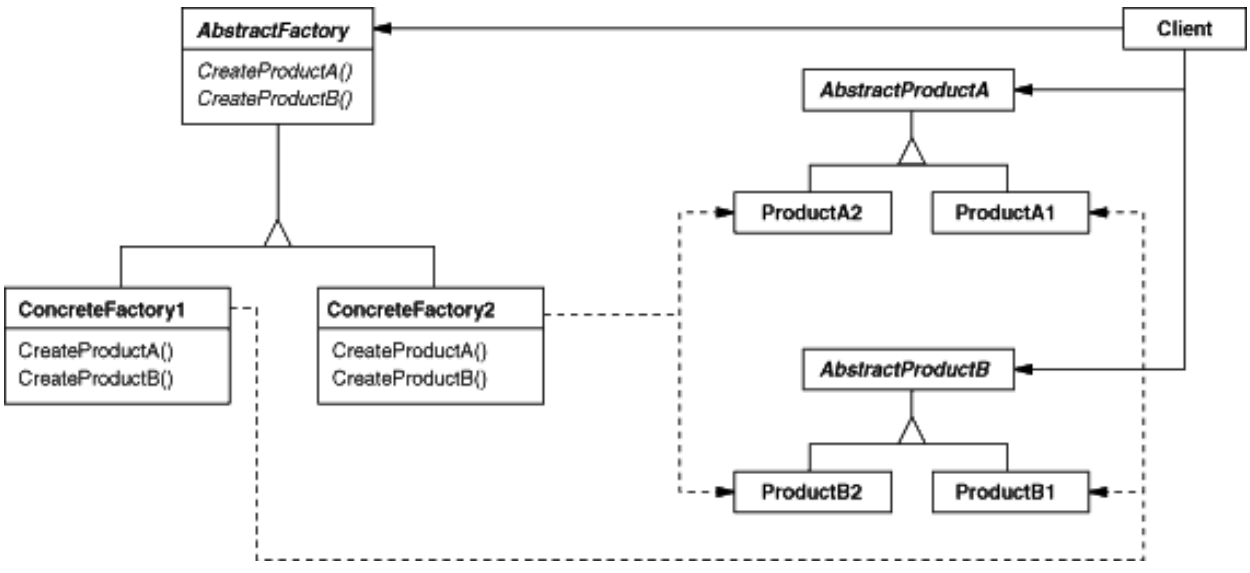
Patrones de diseño

FACTORÍA ABSTRACTA

Factoría abstracta

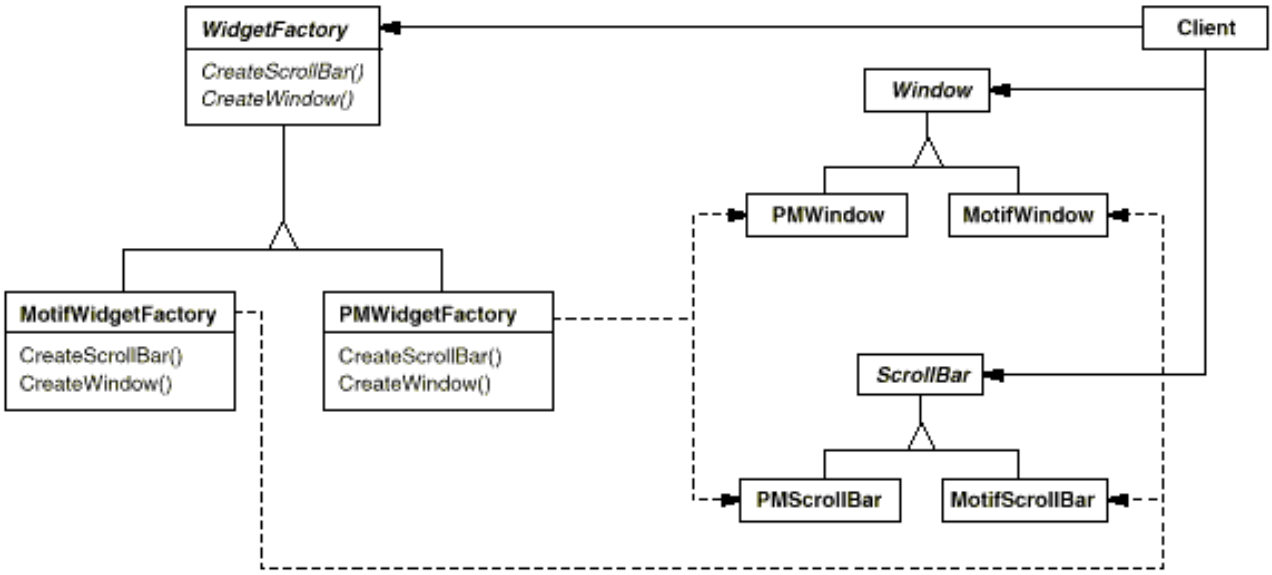
- El patrón de diseño *factoría abstracta* [Gamma et al., 2008] proporciona una interfaz general para crear familias de objetos.
  - Se trata de objetos relacionados o que dependen entre sí.
  - Se busca no tener que especificar sus clases concretas.
    - Desliga la creación de nuevos objetos de la clase que se refiere a dichos objetos a través de la interfaz que implementan.

Factoría abstracta: diagrama de clases



[Gamma et al., 2008]

Ejemplo: factoría abstracta para interfaz gráfica



[Gamma et al., 2008]

Ejemplo: factoría abstracta

```
public class ServiciosUsuarioImp implements
  ServiciosUsuario {

  //forzaría a tener estado. Se podría quitar y usar
  //la factoría cada vez que se necesitase usar el
  //daoUsuario
  protected DAOUsuario daoUsuario;

  public ServiciosUsuarioImp(FactoriaDAOUsuario
  factoria)
  {
      daoUsuario= factoria.generaDAOUsuario();
  }
...
}
```

El cliente *ServicioUsuarioImp* sólo necesita objetos que cumplan con la interfaz *DAOUsuario* (los *AbstractProduct*). Se los proporciona un objeto que implementa la interfaz *FactoriaDAOUsuario* (la *AbstractFactory*).

Ejemplo: factoría abstracta

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

La interfaz *FactoriaDAOUsuario* describe la *AbstractFactory*.

```
public class FactoriaDAOUsuarioImp implements
  FactoriaDAOUsuario {

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

La clase *FactoriaDAOUsuarioImp* (una *ConcreteFactory*) implementa la interfaz *FactoriaDAOUsuario* (que describe la *AbstractFactory*).

Ejemplo: factoría abstracta – comentarios

- En el ejemplo anterior, la factoría abstracta se crea fuera del cliente, y se pasa a éste como parámetro.
- La factoría se puede crear de varias maneras:
  - En el programa principal como un objeto más.
  - Una opción mejor sería considerar a la factoría un objeto *singleton*.

Ventajas e inconvenientes

- Ventajas:
  - Aísla las clases concretas que se crean en una aplicación y se manejan a través de las 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 relacionados

- El patrón *singleton* se usa frecuentemente para la creación de los objetos *factoría*.

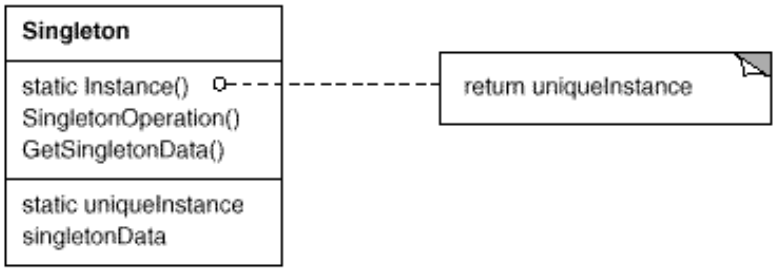
Patrones de diseño

SINGLETON

Singleton

- El patrón de diseño *singleton* [Gamma et al., 2008] garantiza que sólo hay una instancia de una clase, proporcionando un único punto de acceso a ella.
- La instancia puede ser redefinida mediante herencia.
  - Los clientes deben ser capaces de utilizar estas subclases sin modificar su propio código.

Singleton: diagrama de clases



[Gamma et al., 2008]

Ejemplo: *singleton* (1/2)

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

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

  public abstract DAOUsuario generaDAOUsuario();
}
```

Instancia única (*singleton*) de la factoría.

La instancia se crea la primera vez que se necesita. Luego siempre se devuelve la misma instancia.

La constructora de la clase del *singleton* es privada.

Ejemplo: *singleton* (2/2)

```
public class FactoriaDAOUsuarioImp extends
  FactoriaDAOUsuario {

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

Redefinición de la instancia ocultada por el mecanismo de herencia.

Ejemplo: *singleton* – comentarios

- Aquí el *singleton* siempre crea la misma clase de factoría.
  - Si los clientes quieren obtener otra implementación de la factoría, deberá cambiarse el código de ésta a nivel paquete.
- Hay opciones más razonables.
  - El 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
- Para evitar problemas de creación y/o carga en entornos concurrentes, el método estático que devuelve la instancia debe garantizar el acceso concurrente.
  - Ej. `synchronized` en Java

Ventajas e inconvenientes

- 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.
    - No tiene que ser necesariamente una sola instancia.
    - Ej. *pools* de componentes.
  - Más flexibles que las operaciones de clase estáticas.
    - Las instancias son objetos completos.

Patrones relacionados

- Se puede usar en cualquier patrón que requiera limitar el número de instancias de una clase determinada.

ARQUITECTURAS

Arquitecturas de capas

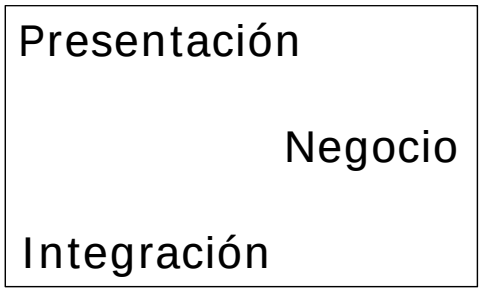
- Las arquitecturas de capas en sistemas de información distinguen componentes de tres tipos:
  - Presentación
    - Gestionan las interfaces con el usuario.
  - Negocio
    - Implementan las reglas de gestión de datos y servicios de la aplicación.
  - Integración
    - Comunican a los otros componentes con recursos y sistemas externos.
- Dependiendo de cómo se organizan estos componentes se habla de arquitecturas de una, dos y tres capas.

Capas físicas y lógicas

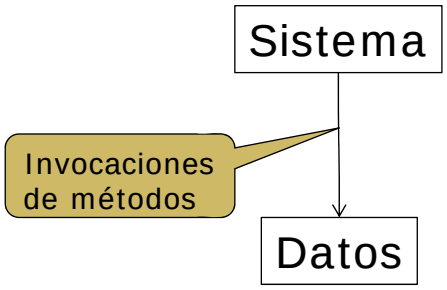
- Nótese que las capas de estas arquitecturas son *capas lógicas*.
  - Relativas a la organización de los componentes del sistema.
- Las *capas físicas* se refieren al despliegue en nodos de los componentes según sus tipos / capas.
  - Ej. la capa de presentación web y la lógica del negocio podrían estar en la misma máquina o en máquinas distintas.

Arquitectura de una capa

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

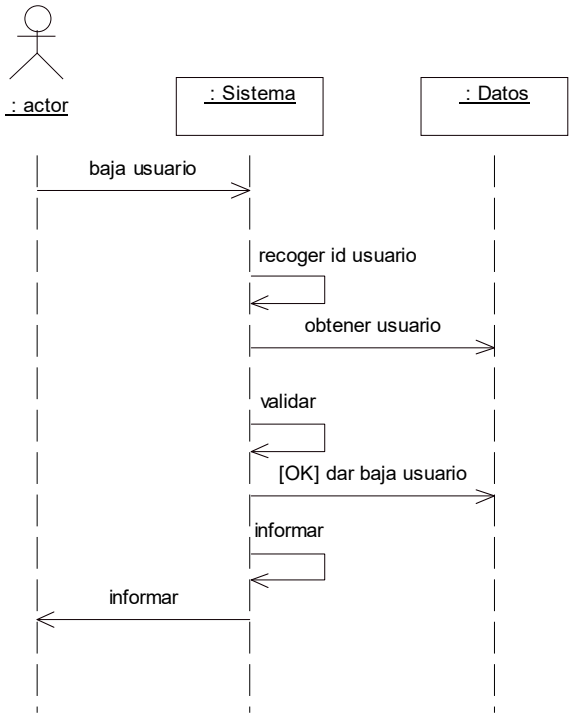


Esquema de capas



Clases del sistema

Arquitectura de una capa: comportamiento



Ventajas e inconvenientes

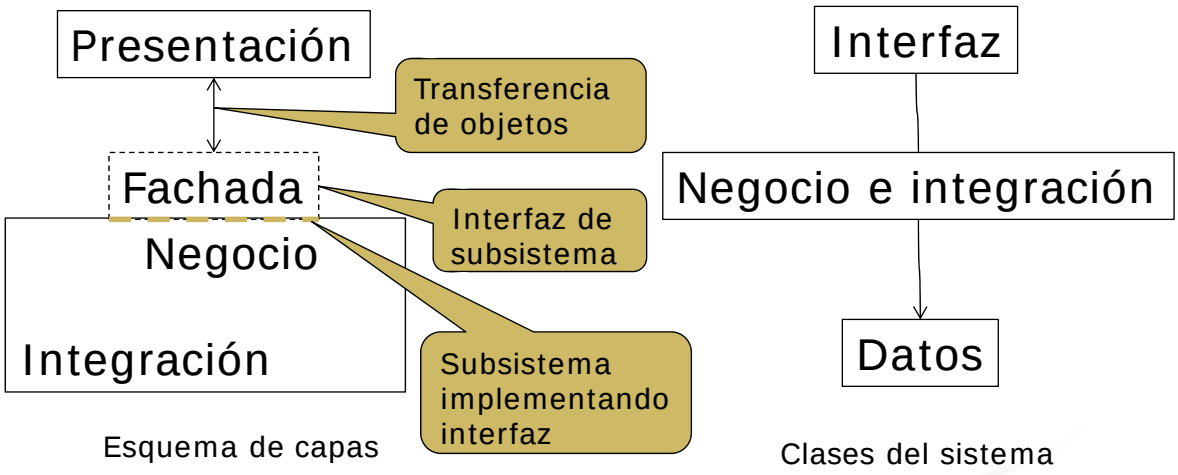
- Ventajas:
  - Sencillez conceptual.
- Inconvenientes:
  - No se puede modificar ni la interfaz de usuario, ni la lógica del negocio ni la representación de los datos sin afectar a las demás capas.
  - Complicación fáctica.

Patrones relacionados

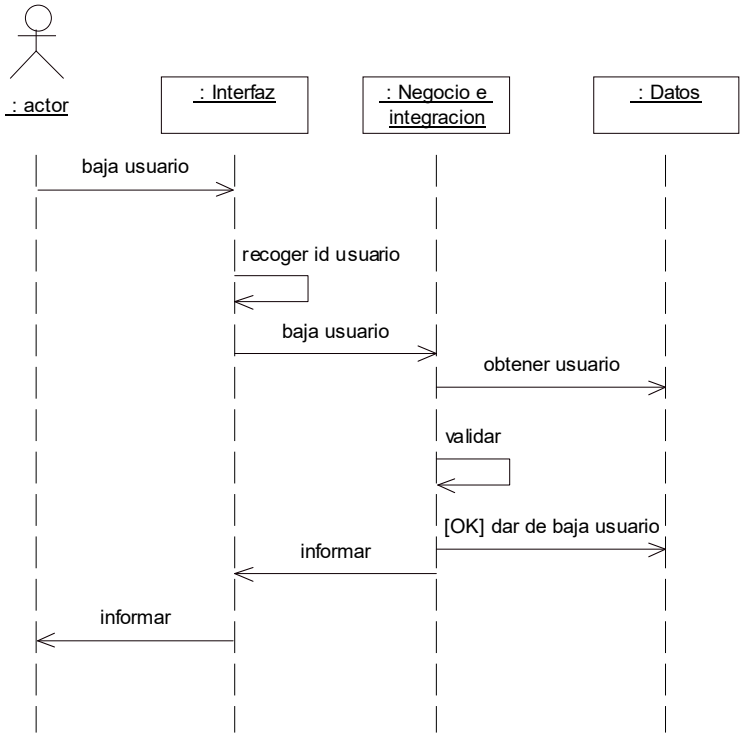
- No hay ninguno a priori, aunque pueden aplicarse a elección del arquitecto.

Arquitectura de dos capas

- 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: comportamiento



Ventajas e inconvenientes

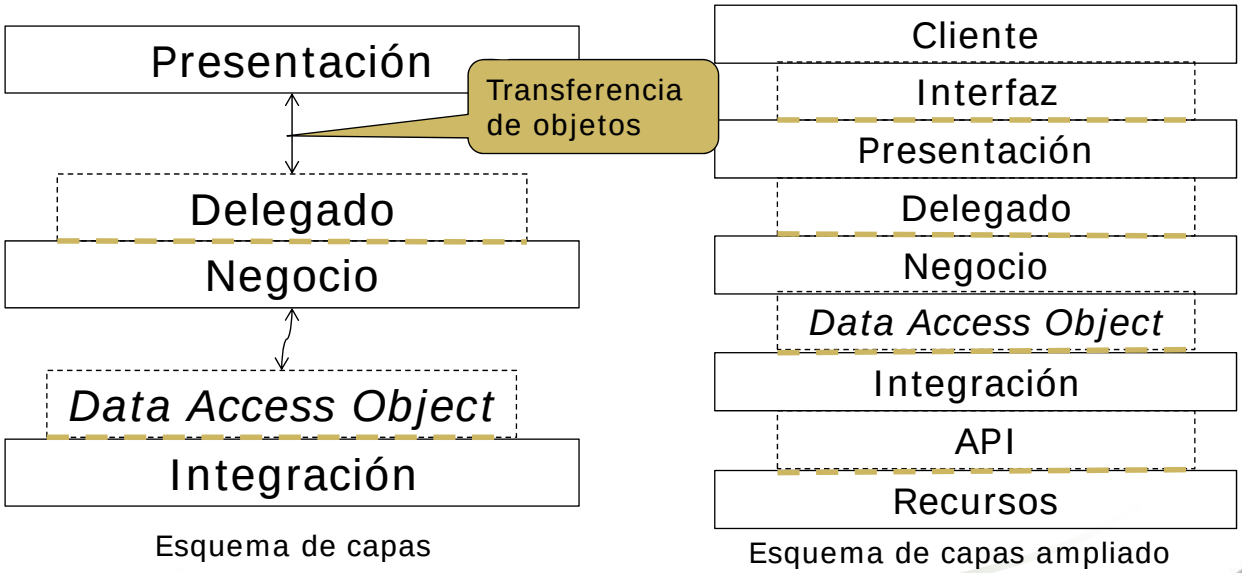
- Ventajas:
  - Permite cambios en la interfaz de usuario o en el resto del sistema sin interferencias mutuas.
  - Simplicidad fáctica.
- Inconvenientes:
  - Mayor complicación arquitectónica que la arquitectura de una capa.
  - No se puede modificar la lógica del negocio o la representación de los datos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

Patrones relacionados

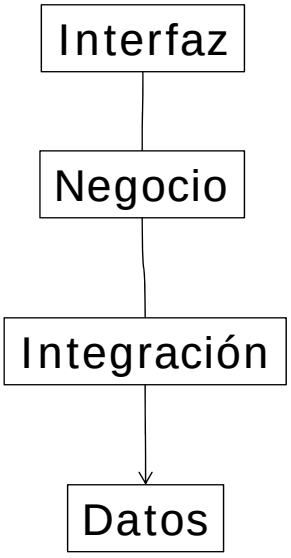
- Aunque no es estrictamente necesario, suele implementarse con un MVC.
- Se pueden aplicar otros patrones a elección del arquitecto.

Arquitectura multicapa

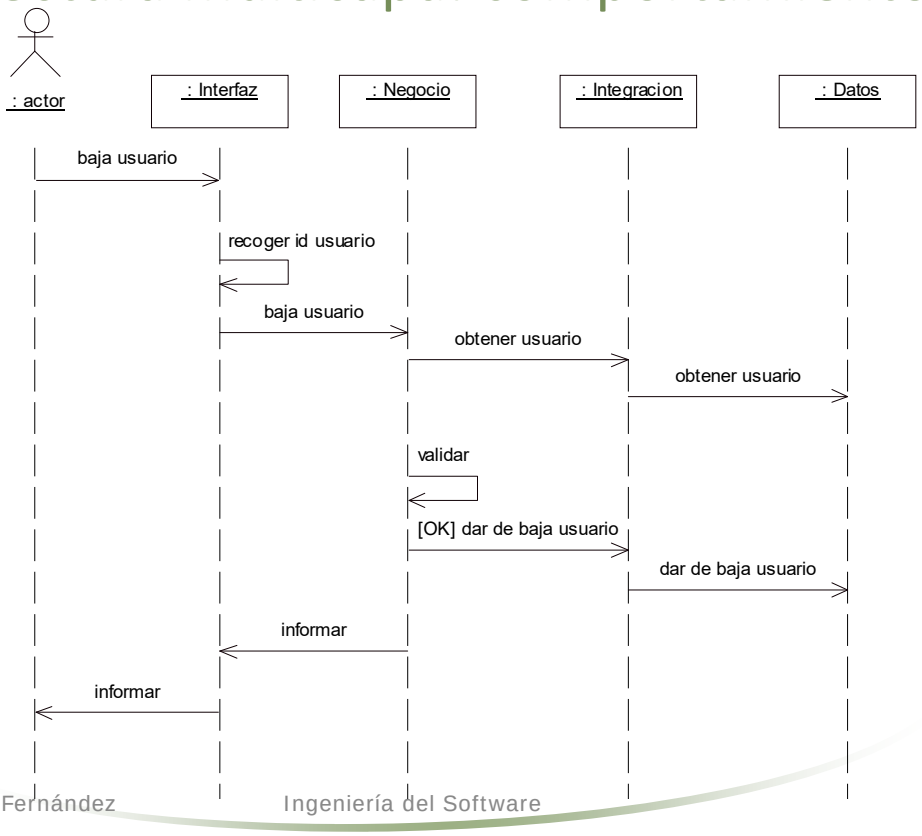
- La arquitectura de tres capas o multicapa considera capas diferentes para presentación, negocio e integración.



Arquitectura multicapa: clases del sistema



Arquitectura multicapa: comportamiento



Ventajas e inconvenientes

- Ventajas:
  - Se puede modificar cualquier capa sin afectar a las demás.
  - ¿Simplicidad fáctica?
  - Integración y reusabilidad — Encapsulación
  - Distribución — Particionamiento
  - Escalabilidad
  - Mejora del rendimiento — Mejora de la fiabilidad
  - Manejabilidad — Incremento en la consistencia y flexibilidad
  - Soporte para múltiples clientes
  - Desarrollo independiente — Desarrollo rápido
  - Empaquetamiento — Configurabilidad
- Inconvenientes:
  - Mayor complejidad arquitectónica
  - Posible pérdida de rendimiento y escalabilidad
  - Riesgos de seguridad
  - Gestión de componentes

Algunas propiedades pueden ser ventajas o inconvenientes, dependiendo del diseño final.

Patrones relacionados

- Capa de presentación
  - Controlador frontal
  - Controlador de aplicación
- Capa de negocio
  - Transferencia
  - Servicio de aplicación
  - Delegado del negocio
  - Objeto del negocio
- Capa de integración
  - *Data Access Object*
  - Almacén del dominio
- Se pueden aplicar otros patrones a elección del arquitecto.

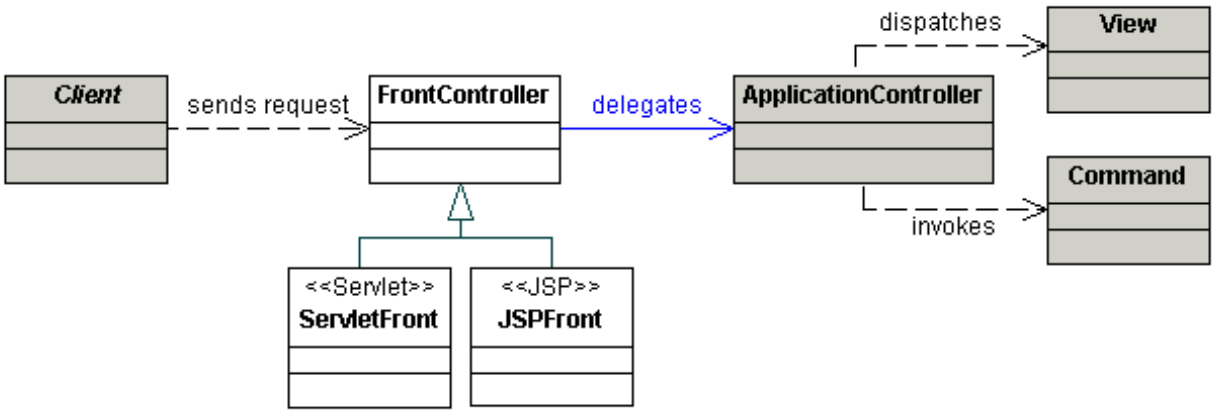
Patrones de arquitectura multicapa

CONTROLADOR FRONTAL

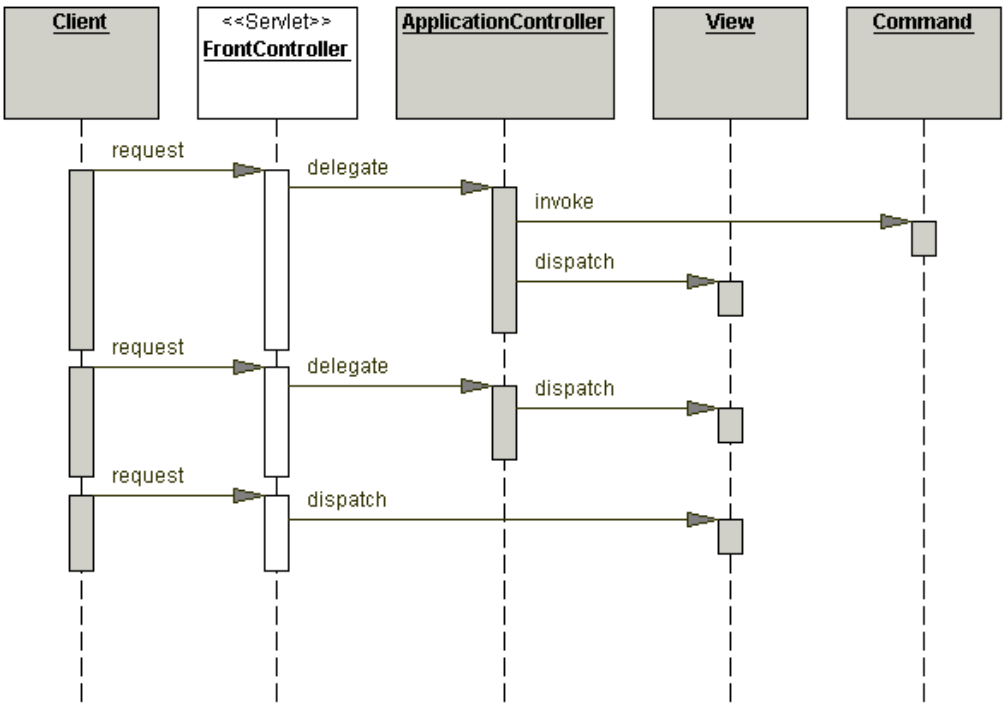
Controlador frontal

- El patrón *controlador frontal* (*front controller*) persigue proporcionar un punto de acceso para el manejo de las peticiones de la capa de presentación.
  - Se tiene un punto inicial de contacto para gestionar las peticiones.
  - Ello permite centralizar la lógica de control y las actividades de manejo de peticiones.
- 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.

Controlador frontal: estructura



Controlador frontal: interacción



Ejemplo: controlador frontal – acceso

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

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

Posibles peticiones a una aplicación web. Este componente de la capa de presentación es el *controlador frontal*. Gestiona las peticiones a través de un método común *processRequest*. En el método se producirá la delegación a un *controlador de aplicación*.

Ejemplo: controlador frontal – gestión interna (1/3)

```
 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);
...
```

Creación de parámetros (*RequestContext*) para el procesamiento de la petición web en el *controlador de aplicación*.

Ejemplo: controlador frontal – gestión interna (2/3)

```
    ApplicationController applicationController =
      new ApplicationControllerImpl();
    ResponseContext responseContext =
      applicationController.handleRequest(
        requestContext);
    applicationController.handleResponse(
        requestContext, responseContext);
  } catch (Exception e) {
```

Las condiciones de error excepcionales se procesan aparte.

El *controlador frontal* crea e invoca un controlador de aplicación (*ApplicationController*) para gestionar la petición. Éste procesa la petición (*requestContext*) y genera la respuesta apropiada (*responseContext*).

Ejemplo: controlador frontal – gestión interna (3/3)

```
    LogManager.logMessage("FrontController:exception
      : " + e.getMessage());
    request.setAttribute(resource.getMessageAttr(),
      "Exception occurred : " + e.getMessage());
    page = resource.getErrorPage(e);
    dispatch(request, response, page);
  }
}
```

Gestión de excepciones a través de un método común *dispatch*.

Ejemplo: controlador frontal – errores

```
//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);
}

}
```

Método de visualización de errores.

Ventajas e inconvenientes

- 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.

Patrones relacionados

- Ver el patrón *controlador de aplicación*.

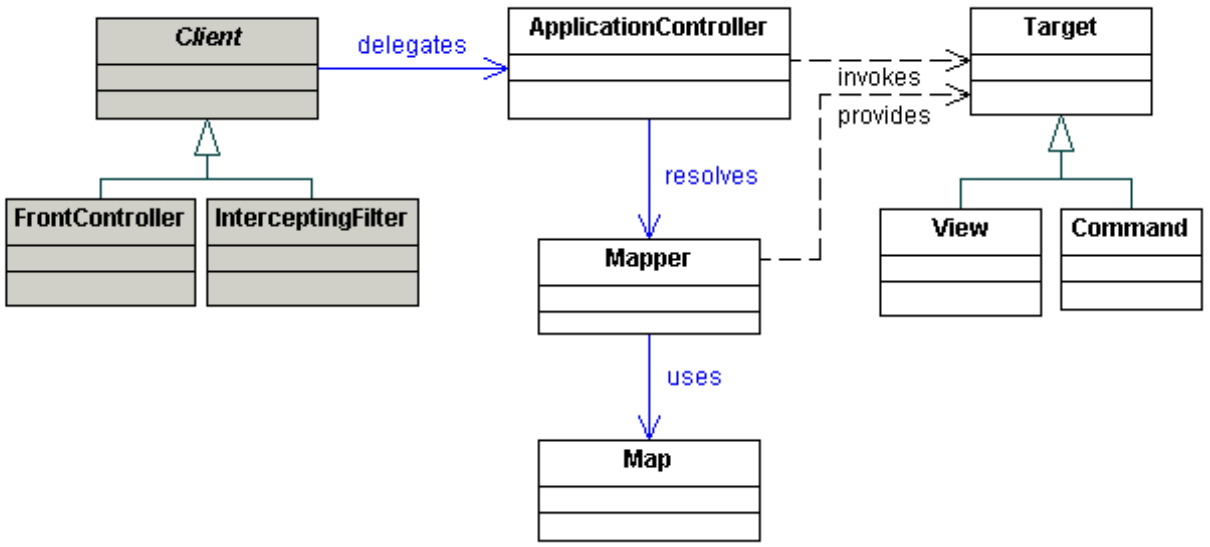
Patrones de arquitectura multicapa

CONTROLADOR DE APLICACIÓN

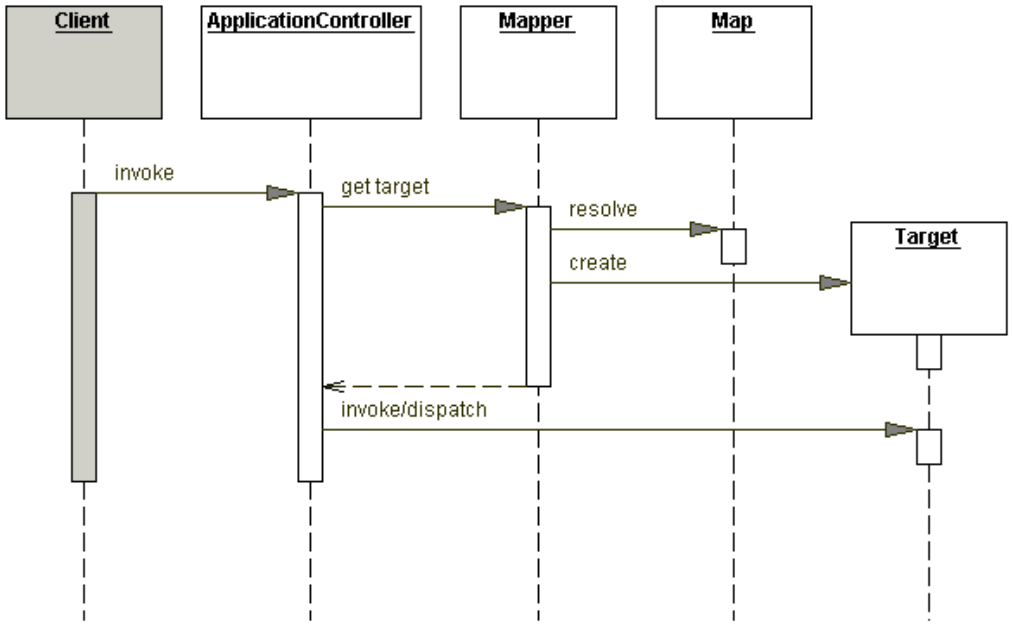
Controlador de aplicación

- El patrón *controlador de aplicación* (*application controller*) persigue centralizar y modularizar la gestión de acciones y vistas.
  - Centraliza la recuperación e invocación de componentes de procesamiento de las peticiones, tales como comandos y vistas.
- Motivación:
  - Se desea reutilizar el código de gestión de vistas y acciones.
  - Se desea mejorar la extensibilidad del manejo de peticiones.
    - Ej. añadir casos de uso a una aplicación incrementalmente.
  - Se desea mejorar la modularidad del código y la mantenibilidad.
    - Facilitando la extensión de la aplicación y la prueba del código de manejo de peticiones independiente de aspectos concretos del entorno.
    - Típicamente, abstrayéndose del contenedor web.

Controlador de aplicación: estructura



Controlador de aplicación: interacción



Ejemplo: controlador de aplicación – interfaz

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

La interfaz ofrece métodos para procesar diferentes tipos de peticiones.

Viene del ejemplo anterior

Ejemplo: controlador de aplicación – clase (1/2)

```
class WebApplicationController implements
 ApplicationController {

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

Procesamiento de la petición web en el *controlador de aplicación*.

Ejemplo: controlador de aplicación – clase (2/2)

```
     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;
 }
```

El *mapper* (*commandFactory*) devuelve la indicación del *target* (*Command*) capaz de procesar la petición (*commandName*).

Aquí el controlador de aplicación usa un objeto adicional (*CommandProcessor*) para invocar el target (*Command*) indicado.

Ventajas e inconvenientes

- 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.

Patrones relacionados

- Ver el patrón *controlador frontal*.

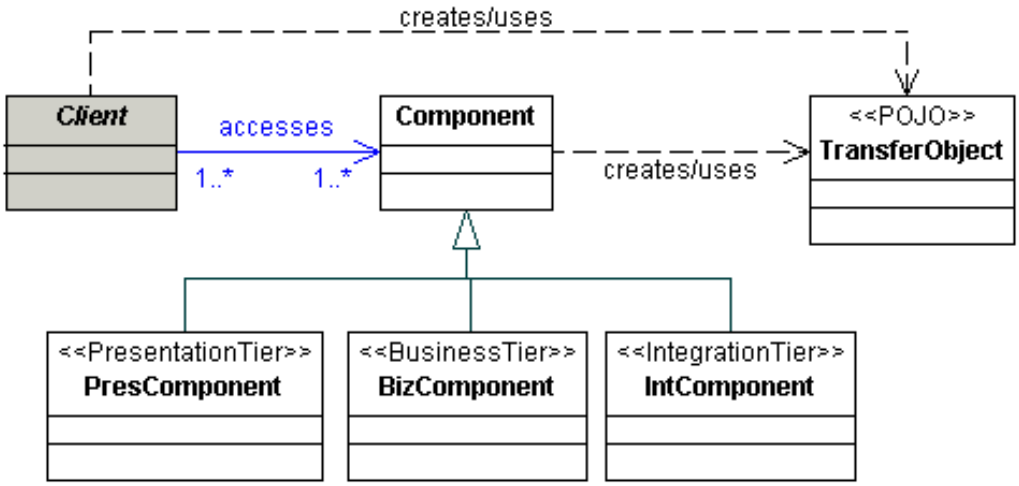
Patrones de arquitectura multicapa

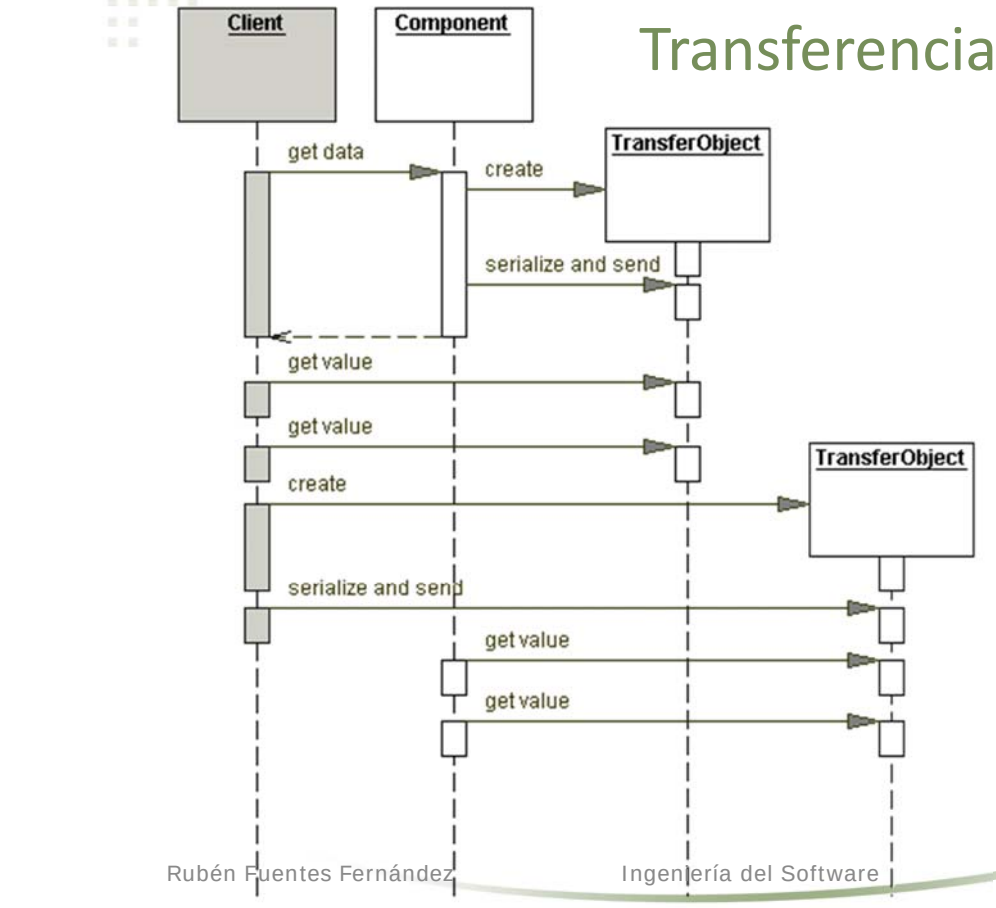
TRANSFERENCIA

Transferencia

- El patrón *transferencia* (*transfer*) persigue independizar el intercambio de datos entre capas.
  - No se desea conocer la representación interna de una entidad dentro de una capa.
  - Al ser un mecanismo de comunicación entre capas, son objetos serializables.
- Motivación:
  - Se quiere que la representación de las entidades dentro de cada capa sea transparente a las demás.
    - Indispensable para independizar efectivamente las capas.
    - Ej. si se accede a bases de datos relacionales, los clientes deberían abstraerse de la existencia de columnas en los datos y trabajar sólo con objetos.

Transferencia: estructura





Transferencia: interacción

Ejemplo: transferencia – clase

```
public TransferLibro {
 //atributos de libro
 public String autor;
 ...
 //accesores y mutadores de libro
 public String getAutor() {
   return autor; }
 ...
 public void setAutor(String autor) {
   this.autor= autor; }
 ...
```

El objeto transferencia encapsula los datos y es a través del que se maneja esta información en las capas.

Ejemplo: transferencia – carga

```
public DAOEjemplaresImp implements DAOEjemplares {

 public TransferLibro obtenerLibro(String id) {
   //código acceso a la base de datos
  TransferLibro libro= new TransferLibro(autor,...);
  return libro;
 }
 ...
}
```

El acceso a la representación interna de los datos en el sistema de gestión de datos se delega en DAOs. Los DAOs transfieren información a y desde los objetos transferencia con el sistema de gestión de datos. El resto del sistema usa los objetos transferencia.

Ventajas e inconvenientes

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

Patrones relacionados

- Ver el patrón *Data Access Object*.

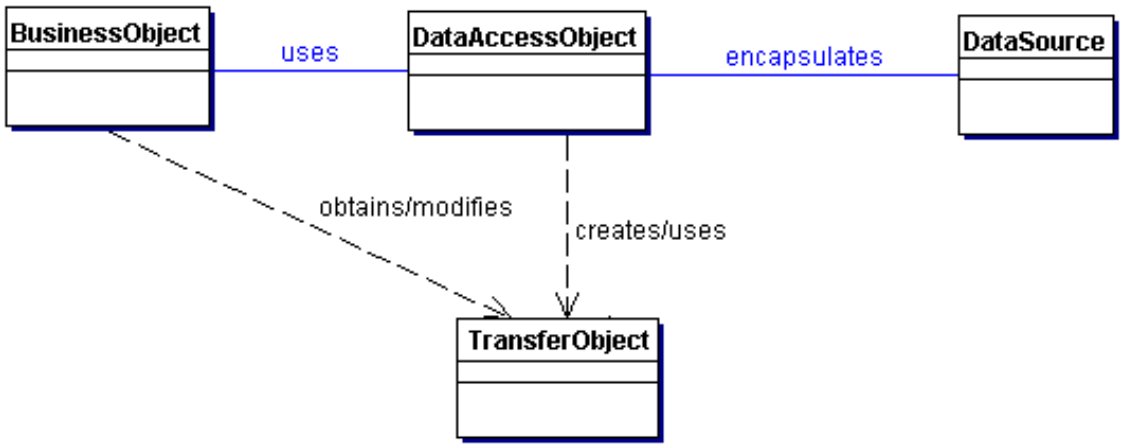
Patrones de arquitectura multicapa

DAO

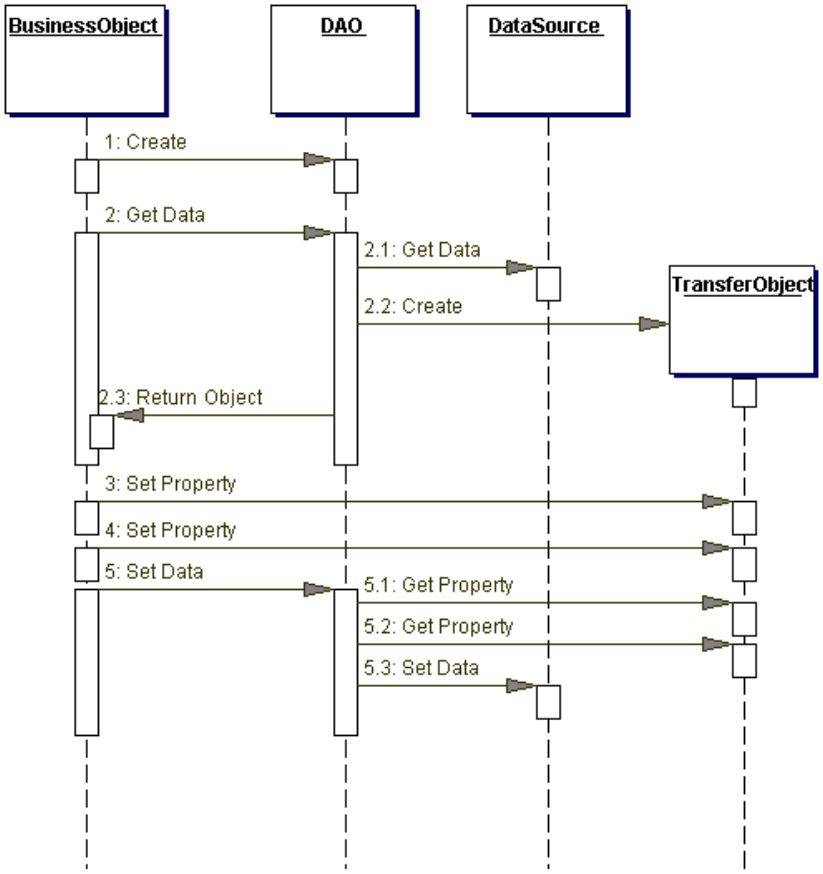
DAO

- El patrón *DAO* (*Data Access Object*) permite acceder a la capa de datos (recursos en general), proporcionando representaciones orientadas a objetos a sus clientes.
  - Se quiere independizar la representación y acceso a los datos de su procesamiento.
  - Las representaciones orientadas a objetos son patrones *transferencia*.
- Motivación:
  - Los sistemas de información (y muchos programas) guardan datos.
  - Estos datos suelen tener estructura, la cual queda plasmada en un sistema de representación.
    - Ej. relacional, XML
  - Manejar estos datos fuerza a:
    - Conocer los mecanismos de acceso del sistema de gestión de datos.
      - Ej. base de datos y sistema operativo
    - Conocer la representación de los datos en el sistema de gestión de datos.
      - Ej. columnas, elementos o bytes.
  - Un cliente de la capa de negocio debería ser independiente de estas cuestiones.
    - 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.

DAO: estructura



DAO: interacción



Ejemplo: DAO – interfaz

```
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);
}
```


Ejemplo: DAO – clase

```
public class DAOUsuarioImp implements DAOUsuario {
 ...
 public Boolean daDeBajaUsuario(Integer idInteger) {
   boolean resultado= true;
   int id= idInteger.intValue();
   //conexión con la base de datos
   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);
 }
...
```

Prepara el acceso a las tablas de la base de datos y ejecuta la operación. Devuelve el resultado al cliente.

Ejemplo: DAO – comentarios

- Aunque el ejemplo lo 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.

Ventajas e inconvenientes

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

Patrones relacionados

- Ver patrón *transferencia*.

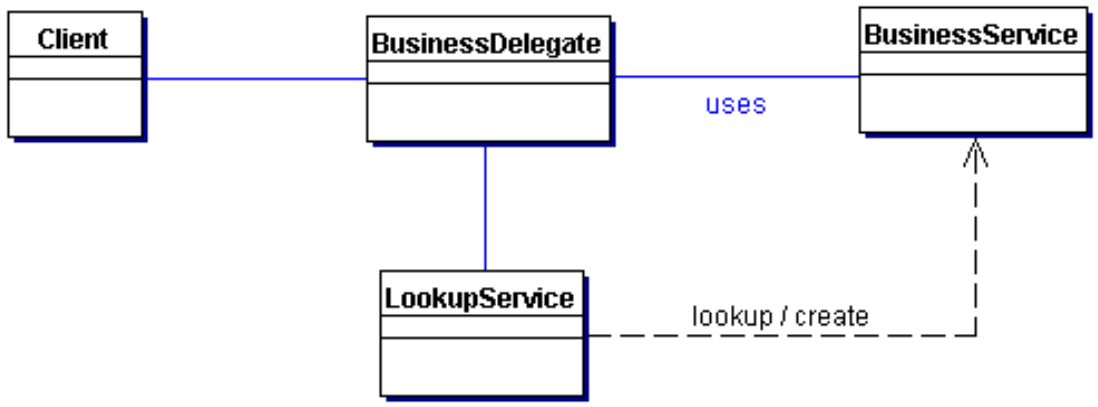
Patrones de arquitectura multicapa

DELEGADO DEL NEGOCIO

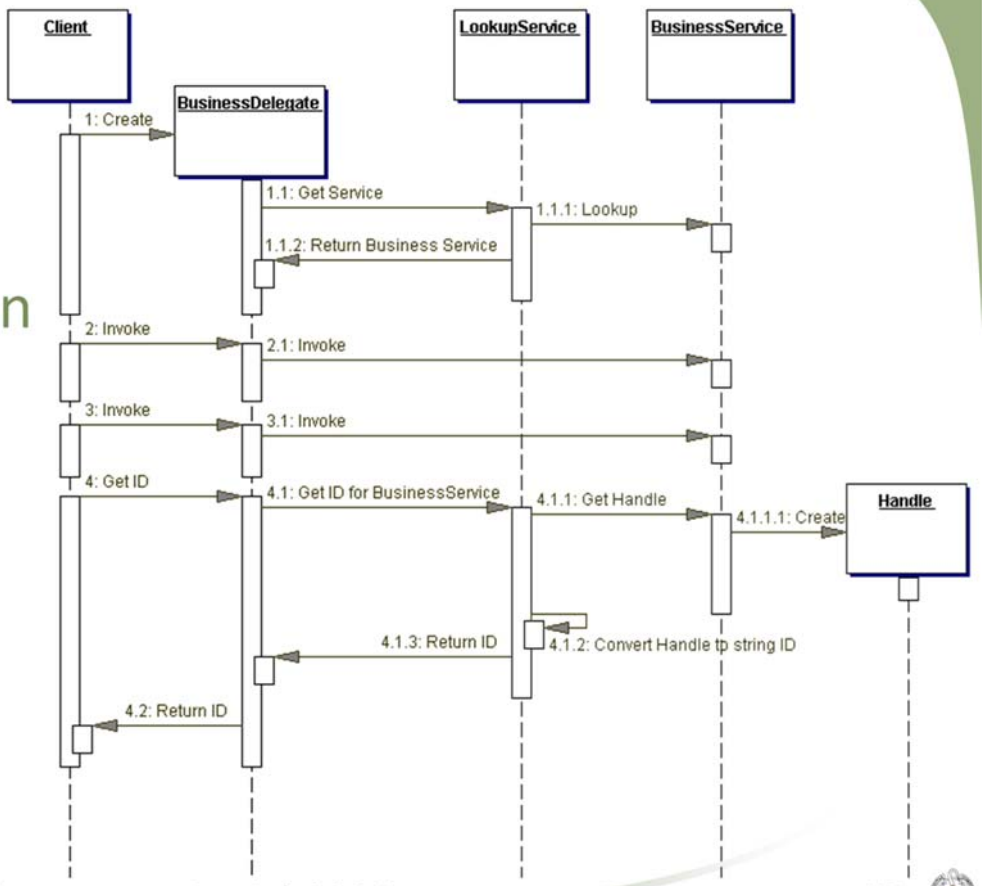
Delegado del negocio

- El patrón *delegado del negocio* (*business delegate*) permite ocultar a los clientes los detalles del acceso a componentes distribuidos.
  - Así los clientes se pueden centrar en la implementación de los procesos propios del negocio.
- Motivación:
  - En una aplicación multicapa es frecuente implementar la capa de negocio con componentes distribuidos.
  - Los clientes de la capa de negocio se ven obligados así a tratar con detalles de conexión y acceso al servidor de aplicaciones.
    - Ej. de estos clientes son los componentes de la capa de presentación.
  - El patrón delegado se encarga de estos detalles haciéndolos transparentes a los clientes.

Delegado del negocio: estructura



Delegado del negocio: interacción



Ejemplo: delegado del negocio – interfaz

```
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());
     }
 }
```

El delegado del negocio accede al servicio de búsqueda (*LookupService*).

Realiza la búsqueda (*lookup*) del servicio de negocio (*BusinessService*) requerido. Aquí se busca la *fachada* (*StockList*) de la aplicación de gestión de acciones (*StockTO*).

Ejemplo: delegado del negocio – interfaz

```
...
//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());
 }
}
...
```

Las operaciones sobre los objetos *transferencia* (*StockTO*) se delegan en la *fachada*.

Ejemplo: delegado del negocio – interfaz

```
...
//el delegado en un singleton
 public static StockListDelegate getInstance()
   throws StockListException {
   if (stockListDelegate == null)
     stockListDelegate = new StockListDelegate();
   return stockListDelegate;
 }
}
```

El delegado de negocio se implementa como un *singleton*. Provee un servicio concreto que no es necesario (ni conveniente) replicar en múltiples objetos.

Ejemplo: delegado del negocio – interfaz

```
//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 addAnalyst(AnalystTO analystTO);
 public void addStock(StockTo stockTO);
}
```

El delegado obtiene de la búsqueda (*lookup*) una interfaz de acceso remota. Ella indica que el servicio se puede manejar distribuidamente, y proporciona los métodos para su uso. En J2EE hay otra interfaz llamada *Home* que permite la búsqueda, creación y destrucción de componentes.

Ejemplo: delegado del negocio – interfaz

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

El EJB de sesión sin estado implementa la interfaz remota.

Ventajas e inconvenientes

- Ventajas:
  - Oculta detalles.
  - Independiza capas.
- Inconvenientes:
  - Introduce un nivel más de indirección.

Patrones relacionados

- El patrón delegado del negocio se introduce por motivos de completitud con respecto a la descripción de aplicaciones de tres capas.
  - Estas aplicaciones suelen estar ligadas a componentes distribuidos.
- El patrón *fachada* tiene una función de abstracción similar.
  - Aunque su funcionalidad es distinta.
    - El delegado del negocio abstrae detalles de conexión y acceso.
    - La fachada abstrae detalles sobre componentes involucrados.

Las aplicaciones desarrolladas en esta asignatura no tienen porque utilizar componentes distribuidos. Ej. EJBs
El patrón fachada encapsulará todos los detalles de implementación a los clientes del negocio.

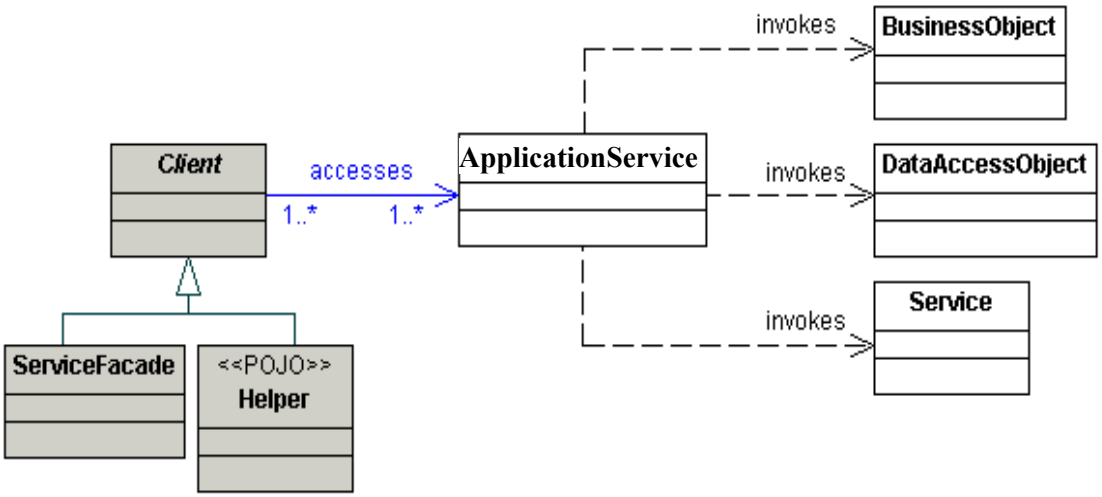
Patrones de arquitectura multicapa

SERVICIO DE APLICACIÓN

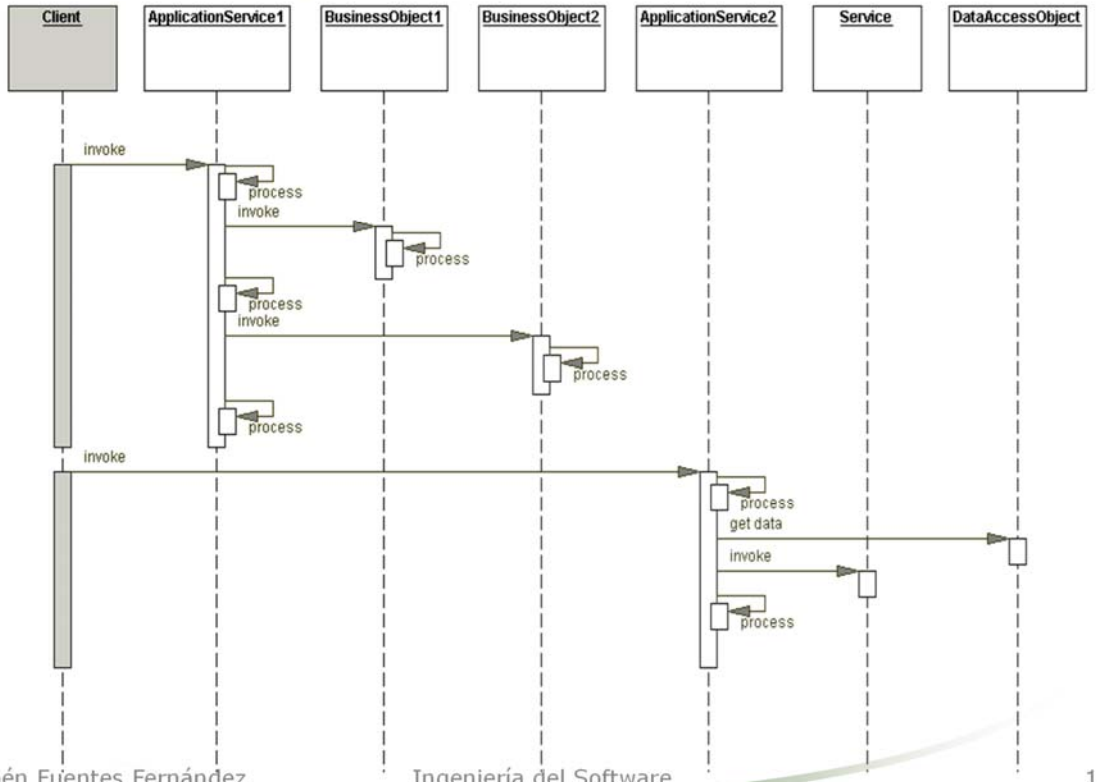
Servicio de aplicación

- El patrón *servicio de aplicación* (*application service*) permite centralizar la lógica de negocio.
  - Se usa para representar una lógica del negocio que actúa sobre distintos servicios u objetos del negocio.
  - Agrupa funcionalidades relacionadas.
  - Encapsula lógica no representada por objetos del negocio.
- 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 aplicación*.

Servicio de aplicación: estructura



Servicio de aplicación: interacción



Ejemplo: servicio de aplicación – interfaz

```
public interface Biblioteca {
 public Integer insertaUsuario(TUsuario usuario);
 public Boolean daDeBajaUsuario(Integer id);
 public TDAOUsuario obtenUsuario(Integer id);
 public Integer insertaPublicacion(
   TPublicacion publicacion);
 public Boolean daDeBajaPublicacion(Integer id);
 public TDAOPublicacion obtenPublicacion(
   Integer id);
 public TPrestamo prestamo(TPrestamo tPrestamo);
 public Boolean devolucion(Integer ejemplar);
}
```

Interfaz pública de una fachada

Ejemplo: servicio de aplicación – clase

```
public class BibliotecaImp implements Biblioteca {
 ...
 public TPrestamo prestamo(TPrestamo tPrestamo) {
   //nótese que no se ha hecho explícito
   //el acceso a estos servicios por parte de la
   //Biblioteca
   serviciosPrestamo.prestamo(tPrestamo);
 }
 ...
}
```

La clase que implementa la fachada accede al servicio de aplicación para prestar el servicio solicitado.

Ejemplo: servicio de aplicación – interfaz

```
package logica.serviciosPrestamo;

import transferenciaCliente.prestamo.TPrestamo;


public interface ServiciosPrestamo {
 public TPrestamo prestamo(TPrestamo tPrestamo);
 public Boolean devolucion(Integer ejemplar);
}
```

Servicio de aplicación

Ejemplo: servicio de aplicación – clase

```
package logica.serviciosPrestamo;
...
public class ServiciosPrestamoImp implements
 ServiciosPrestamo {
 public TPrestamo prestamo(TPrestamo tPrestamo) {
   ...
 }

 public Boolean devolucion(Integer ejemplar) {
 ...
 }
}
```

El servicio de aplicación accede a los diferentes componentes para implementar el servicio con la lógica necesaria.

Ventajas e inconvenientes

- 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.

Patrones relacionados

- Los servicios de la aplicación pueden colaborar entre sí para obtener objetos del negocio (y por extensión, los datos).
  - Ej. en [Alur et al., 2003]
  - Sin embargo esta aproximación puede complicar las validaciones de consistencia de los datos en un entorno multiusuario.
    - EJB facilita no obstante este tipo de control.
- Los servicios de aplicación no suelen tener atributos para hacerlos más ligeros.
  - Los objetos que tienen atributos y operaciones del negocio se implementan con los patrones de *objeto del negocio*.

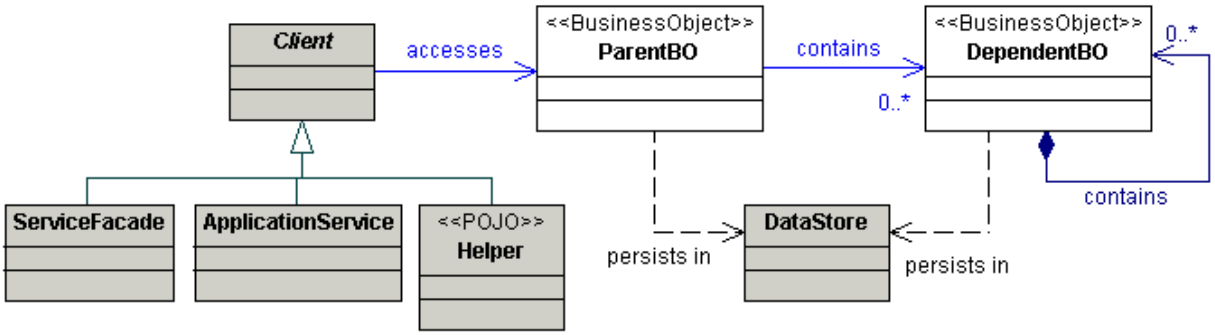
Patrones de arquitectura multicapa

OBJETO DEL NEGOCIO

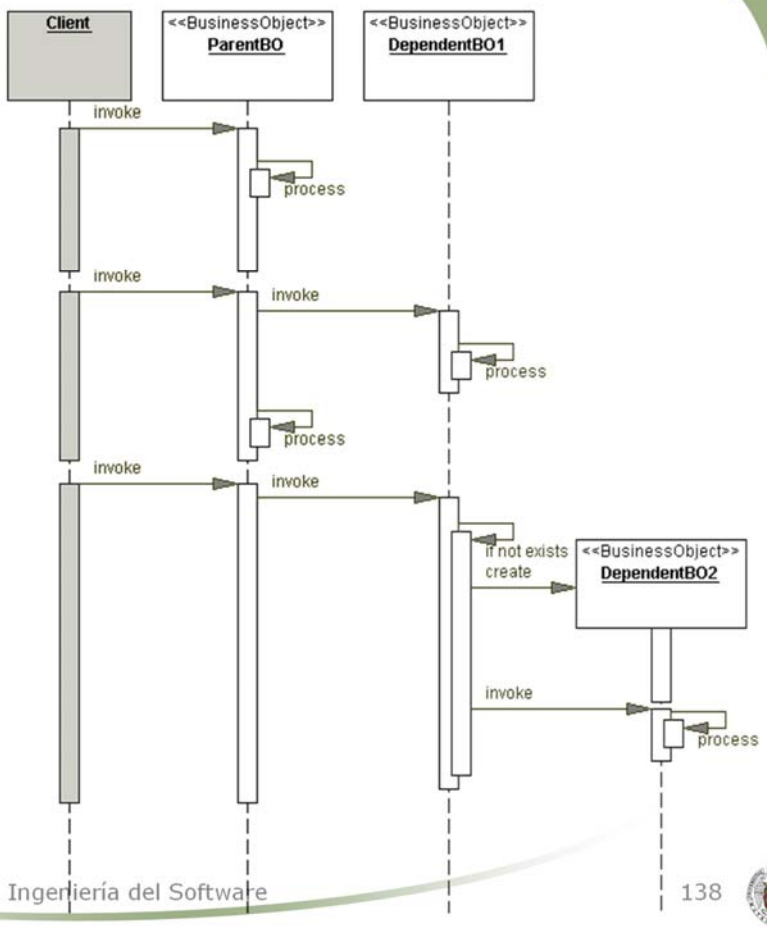
Objeto del negocio

- El patrón *objeto del negocio* (*business object*) permite representar la lógica del negocio y el modelo del dominio en términos orientados a objetos.
  - Se usa cuando se dispone de un modelo conceptual con reglas de validación y lógica del negocio avanzadas.
  - Se desea separar la lógica del negocio del resto de la aplicación.
  - Se busca centralizar la lógica del negocio.
  - Se persigue incrementar la reusabilidad del código.
- 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 (ej. *ServiciosUsuarioImp*) podría acceder directamente a un DAO.
  - 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.

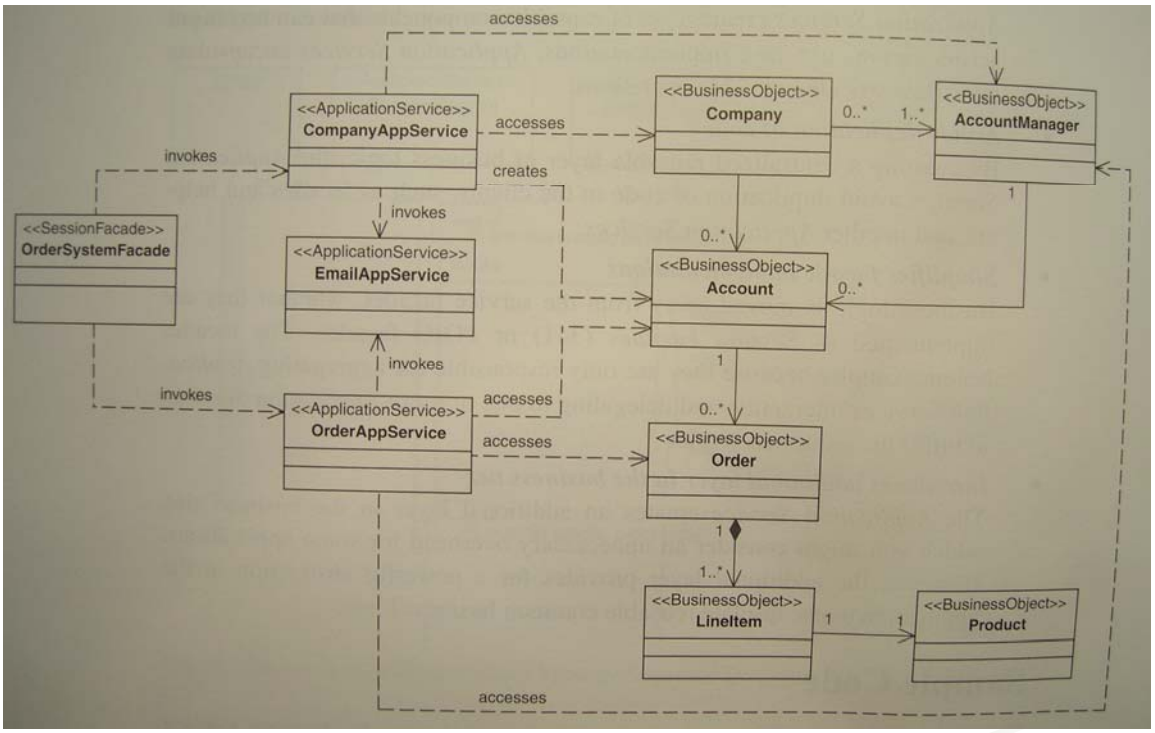
Objeto del negocio: estructura



Objeto del negocio: interacción



Ejemplo: objeto de negocio



Ejemplo: objeto del negocio – cliente (1/2)

```
package logica.serviciosPrestamo.imp;
...
public class ServiciosPrestamoImp implements
 ServiciosPrestamo {

 public TPrestamo prestamo(TPrestamo tPrestamo) {
   int resultado;
   DAOEjemplar daoEjemplar =
     FactoriaPersistencia.getInstancia(
     ).generaDAOEjemplar();
   TEjemplar tEjemplar = daoEjemplar.obtenEjemplar(
     new Integer(tPrestamo.getEjemplar()));
```

Se recuperan los datos para el funcionamiento del objeto de negocio.

Ejemplo: objeto del negocio – cliente (2/2)

```
   if (tEjemplar == null)
     resultado = logica.serviciosPrestamo.
       EventosPrestamo.
       OK_EJEMPLAR_PRESTADO_O_INEXISTENTE;
   else {
     Ejemplar ejemplar= new Ejemplar(tEjemplar);

     if (!ejemplar.sePuedePrestar())
      resultado = logica.serviciosPrestamo.
       EventosPrestamo.
         OK_EJEMPLAR_PRESTADO_O_INEXISTENTE;
     else { ... }
   }
...
}
```

El objeto de negocio (*Ejemplar*) encapsula la lógica sobre los préstamos, aquí con el método *sePuedePrestar*.

Ejemplo: objeto del negocio – clase (1/2)

```
package logica.serviciosPrestamo.imp;
...
//clase que valida si un ejemplar se puede prestar
public class Ejemplar {

 protected int id;
 protected int publicacion;
 protected boolean prestado;
 protected boolean activo;
```

Atributos con información para los servicios en general y la validación en particular.

Ejemplo: objeto del negocio – clase (2/2)

```
public Ejemplar(TEjemplar tEjemplar) {
  this.id= tEjemplar.id;
  this.publicacion= tEjemplar.publicacion;
  this.prestado= tEjemplar.prestado;
  this.activo= tEjemplar.activo;
}

public boolean sePuedePrestar() {
  return !prestado && activo;
}

}
```

Servicio (aquí método) con la lógica de negocio.

Ejemplo: servicio de aplicación – comentarios

- En el ejemplo anterior, la lógica asociada al *objeto del negocio* es trivial.
- Además, la creación del *objeto del negocio* a partir del objeto *transferencia* no es demasiado ortodoxa.
  - Sería mejor relacionar los *objetos del negocio* con su persistencia a través de un *almacén del dominio*.

Ventajas e inconvenientes

- Ventajas
  - Promueve una aproximación orientada a objetos en la implementación del modelo del negocio.
  - Centraliza el comportamiento del negocio, promoviendo la reusabilidad.
  - Evita la duplicación de código.
- Inconvenientes
  - Añade una capa de indirección.
  - Puede producir objetos “inflados” de funcionalidad.
  - Persistencia de dichos objetos del negocio.

Patrones relacionados

- El patrón *almacén del dominio* se suele utilizar para relacionar los *objetos del dominio* con su persistencia.

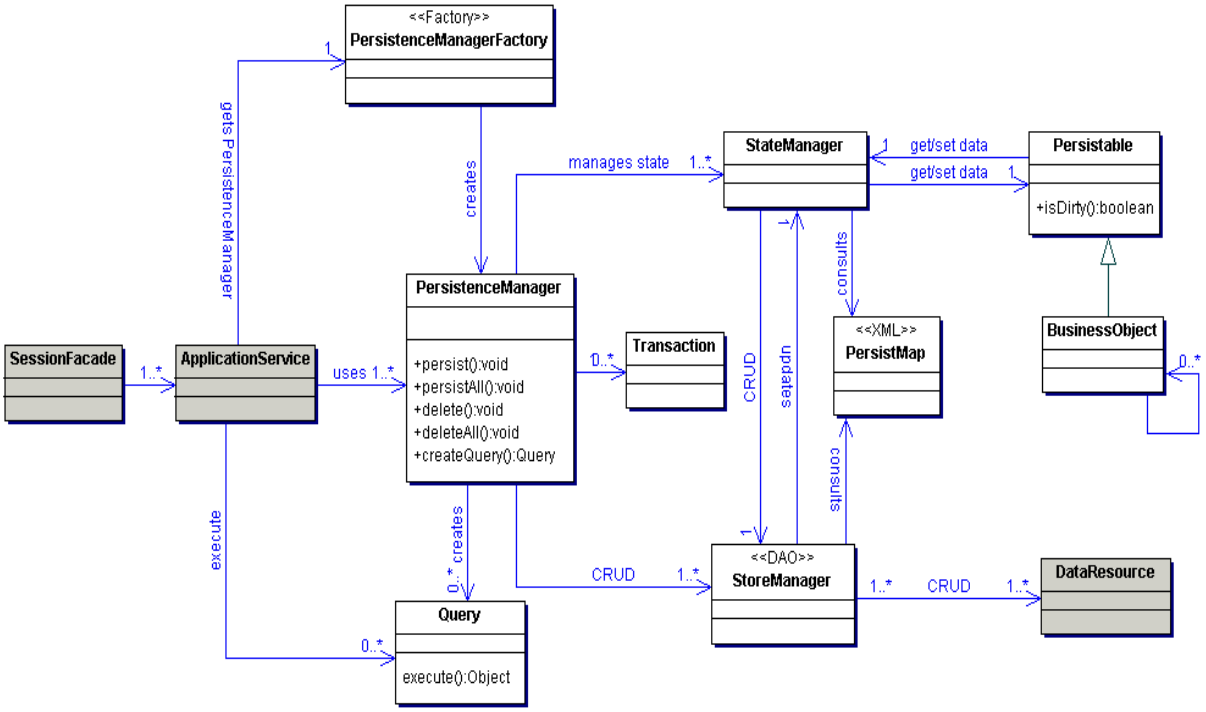
Patrones de arquitectura multicapa

ALMACÉN DEL DOMINIO

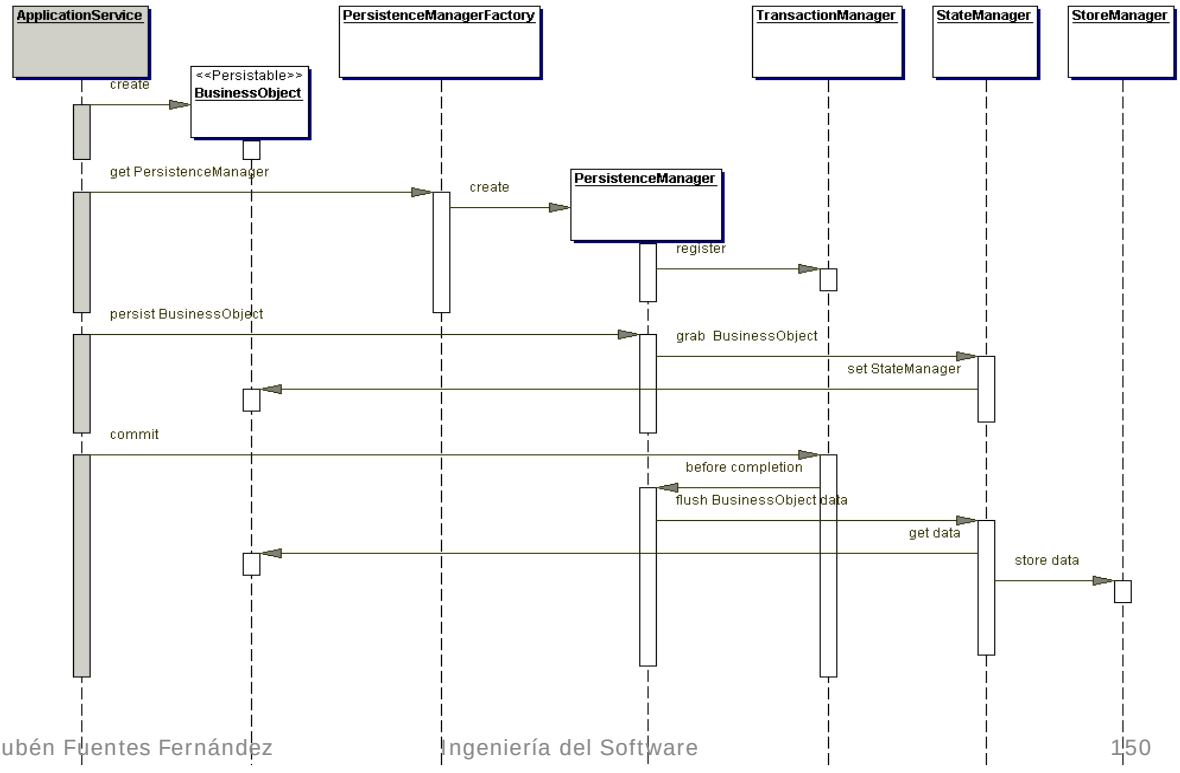
Almacén del dominio

- El patrón almacén del dominio (*domain store*) permite encapsular la persistencia del modelo de objetos.
  - Se busca omitir detalles de persistencia en los objetos del negocio.
  - El modelo de objetos utiliza herencia y relaciones complejas.
  - Referido a J2EE:
    - No se desea utilizar EJBs de entidad.
    - La aplicación podría ejecutarse en un contenedor web.
- Motivación:
  - Muchas veces se dispone de un modelo de objetos del negocio muy rico.
  - El simple uso de patrones *transferencia* no es suficiente.
  - Se desea dar persistencia al modelo de objetos de forma independiente a éste.

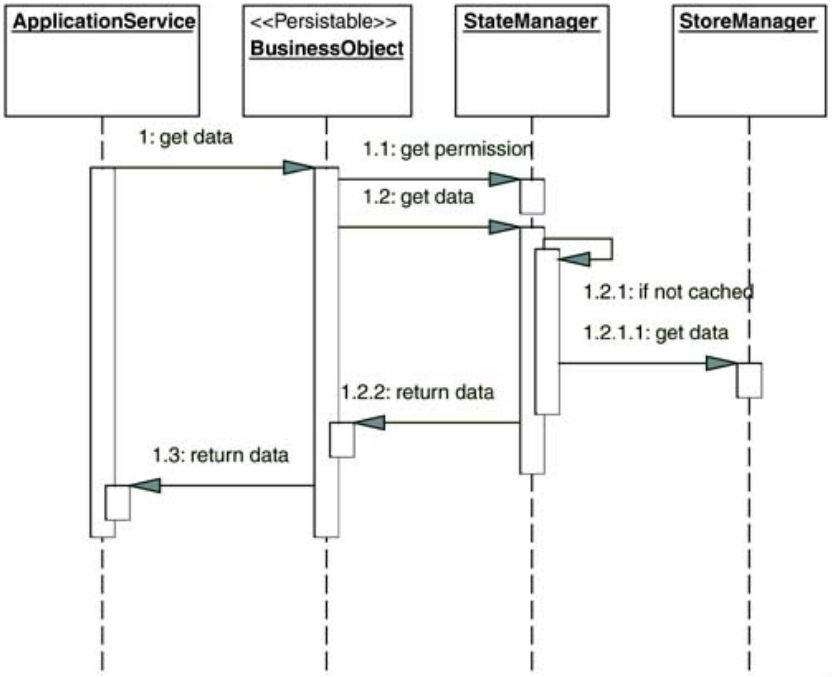
Almacén del dominio: estructura



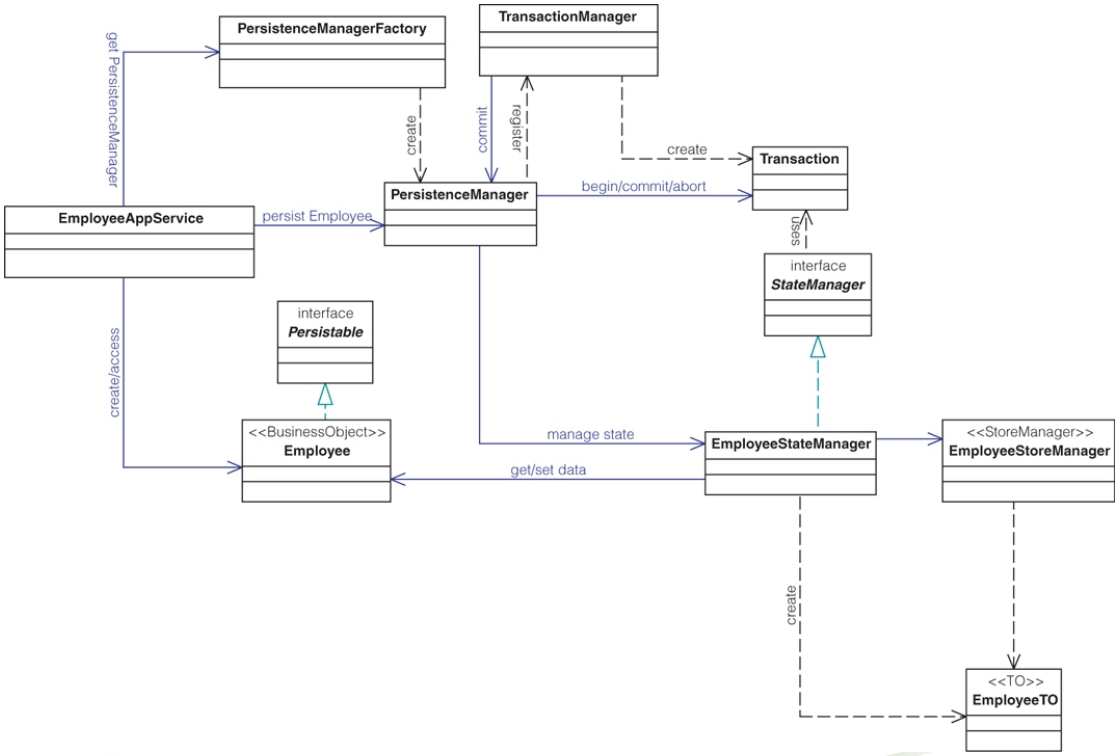
Almacén del dominio: interacción – persistencia de un objeto de negocio



Almacén del dominio: interacción – recuperación de atributos de objeto de negocio



Ejemplo: almacén del dominio



Ventajas e inconvenientes

- Ventajas:
  - Independiza el modelo de objetos de su persistencia.
  - Es capaz de construir el grafo de dependencias entre objetos bajo demanda.
  - Control de transacciones.
- Inconvenientes:
  - Número de objetos.
  - Complejidad.

Patrones relacionados

- El *almacén de dominio* es la combinación de múltiples patrones:
  - *Unit of work*
  - *Query object*
  - *Data mapper*
  - *Table data gateway*
  - *Dependent mapping*
  - *Domain model*
  - *Data transfer object*
  - *Identity map*
  - *Lazy load*

CONCLUSIONES

Conclusiones: generales

- Los sistemas de información son muy relevantes hoy en día.
  - En particular, las aplicaciones empresariales por su tamaño en cuanto a funcionalidad, volumen de datos, distribución y evolución.
- Los patrones vistos proporcionan pautas y estructuras reutilizables para el diseño arquitectónico de sistemas de información.
  - Se centran en los componentes participantes y sus interacciones.
  - No dan detalles internos de los componentes.
- Aunque vistos en el contexto de los sistemas de información, se trata de técnicas útiles para cualquier aplicación software.

Conclusiones: patrones básicos

- 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
- *Singleton* → control del número de instancias disponibles + acceso global sin necesidad de creación + redefinición de operaciones no estáticas

Conclusiones: patrones multicapa

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

Conclusiones: arquitecturas de capas

- Arquitectura de una capa → ¿sencilla? e inmantenible
- Arquitectura de dos capas → sencilla y mantenible a nivel de cambios de interfaz
- Arquitectura multicapa → ¿sencilla? y mantenible a nivel de cambios en cualquier capa

Glosario

- ACM = *Association for Computing Machinery*
- DAO = *Data Access Object*
- EAI = *Enterpise Application Integration*
- EE = *Enterprise Edition*
- EJB = *Enterprise JavaBean*
- IEC = *International Electrotechnical Commission*
- IEEE = *Institute of Electrical and Electronics Engineers*
- ISO = Organización Internacional para la Estandarización
- J2EE = *Java 2 Platform, EE*
- MVC = Modelo-Vista-Controlador
- OO = Orientación a Objetos, Orientado a Objetos
- POJO = *Plain Old Java Object*

Referencias (1/2)

- ACM: The 1998 ACM Computing Classification System. ACM, 2012. Disponible en: http://www.acm.org/about/class/1998, accedido el 20/03/2012.
- C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: A Pattern Language. Oxford University Press, 1977.
- D. Alur, J. Crupi, D. Malks: Core J2EE Patterns – Best Practices and Design Strategies, 2nd Edition. Prentice-Hall PTR, 2003.
- M. Fowler: Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2003.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Patrones de Diseño – Elementos de software orientado a objetos reutilizables. Addison-Wesley, 2008.

Referencias (2/2)

- Software Engineering Standards Committe of the IEEE Computer Society: IEEE-Std-1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE, 2000.
  - ISO/IEC 42010
- M. Juric, R. Nagappan, R. Leander, S.J. Basha: Professional J2EE EAI. Wrox Press, 2001.
- K. Mukhar, C. Zelenak: Beginning Java EE 5 Platform – From Novice to Professional. Apress, 2006.
- S. Stelting, O. Maassen: Patrones de diseño aplicados a Java. Pearson Educación, 2003.