14. La arquitectura multicapa

Índice

- Referencias
- Introducción
- Patrones auxiliares
  - Introducción
  - Observador
  - 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
- Patrón controlador frontal
- Patrón controlador de aplicación

Índice

- Patrón transferencia
- Patrón *Data Access Object*
- Patrón servicio de aplicación
- Patrón *Transfer Object Assembler*
- Patrón objeto del negocio
- 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*

Introducción

- Nota: Los patrones extraídos del libro de GoF están en notación OMT, similar a UML, pero distinta

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:
  - Observador
  - Modelo-Vista-Controlador
  - Factoría abstracta
  - Singleton

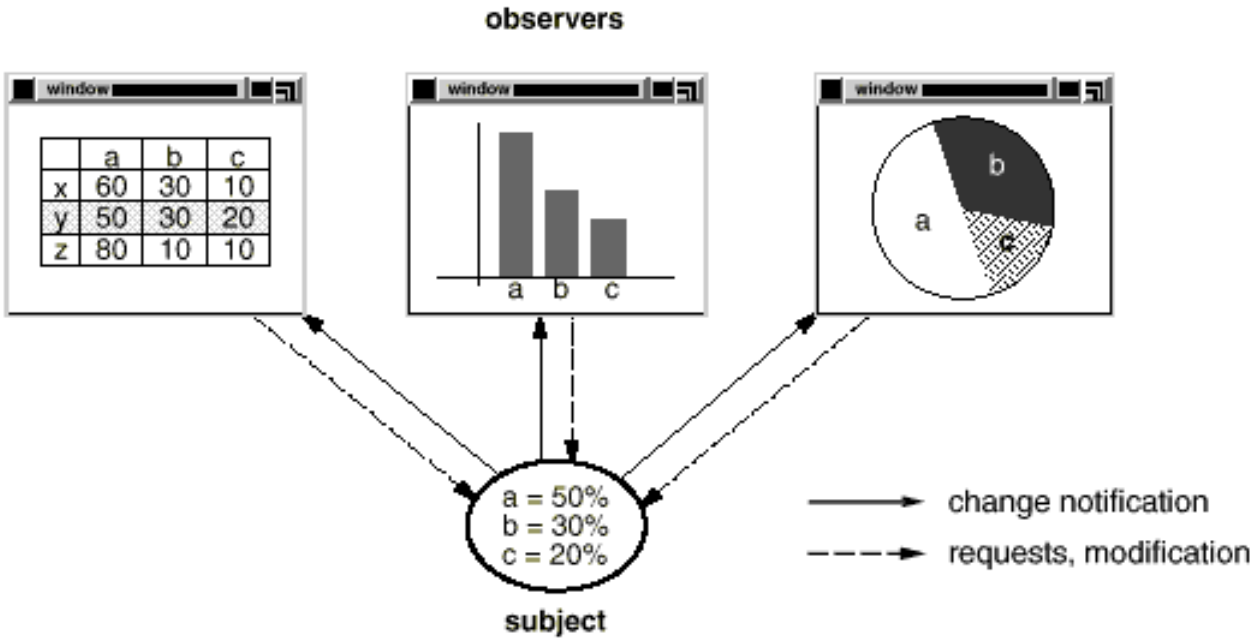
Patrones auxiliares
Observador

- Propósito
  - Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.
- También conocido como
  - Observador.
  - Dependents (dependientes).
  - Publish-Suscribe (publicar-suscribir).

Patrones auxiliares
Observador

- Motivación
  - Si dividimos un sistema en una colección de clases cooperantes debemos mantener la consistencia entre estados relacionados.
  - Esta consistencia no debe lograrse pagando un fuerte acoplamiento.
  - Por ejemplo, en las interfaces de usuario.

Patrones auxiliares
Observador



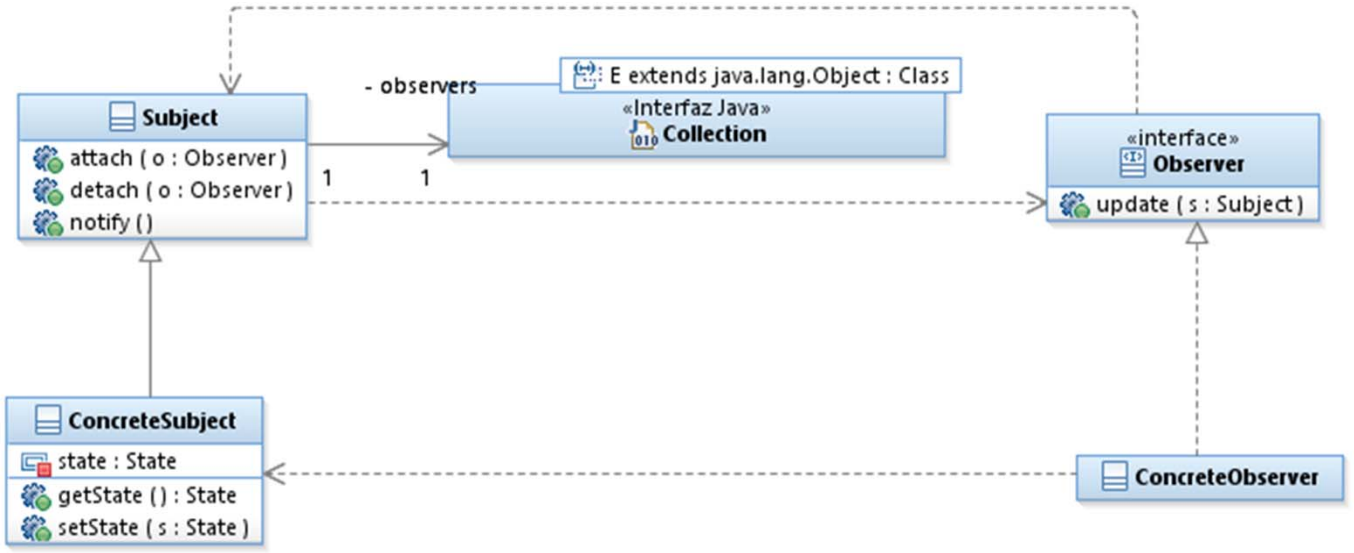
Interfaces de usuario como observers

Patrones auxiliares
Observador

- El patrón Observer debe aplicarse cuando
  - Una abstracción tiene dos aspectos y uno depende del otro.
  - Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
  - Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos.

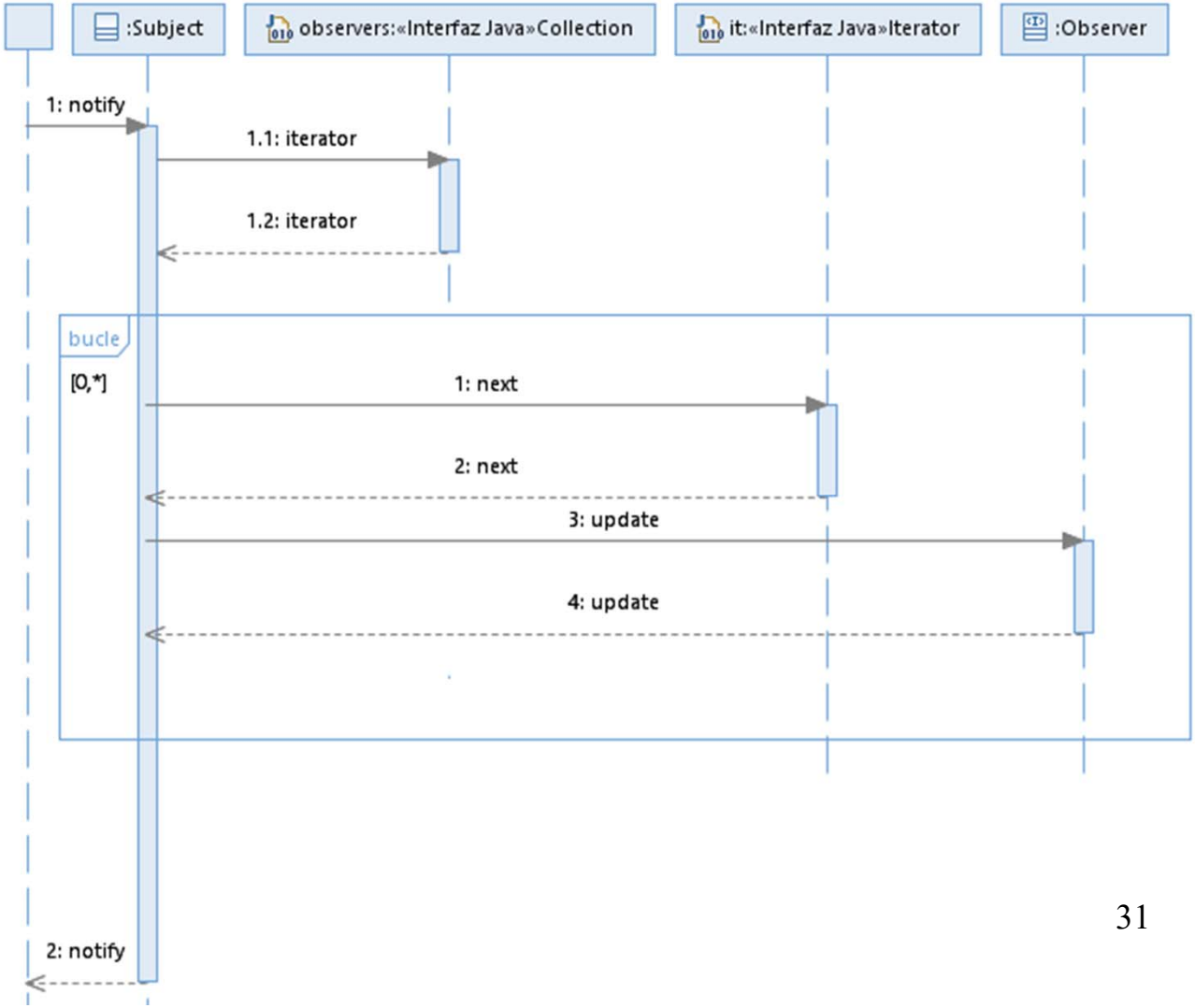
Patrones auxiliares
Observador

- Descripción abstracta



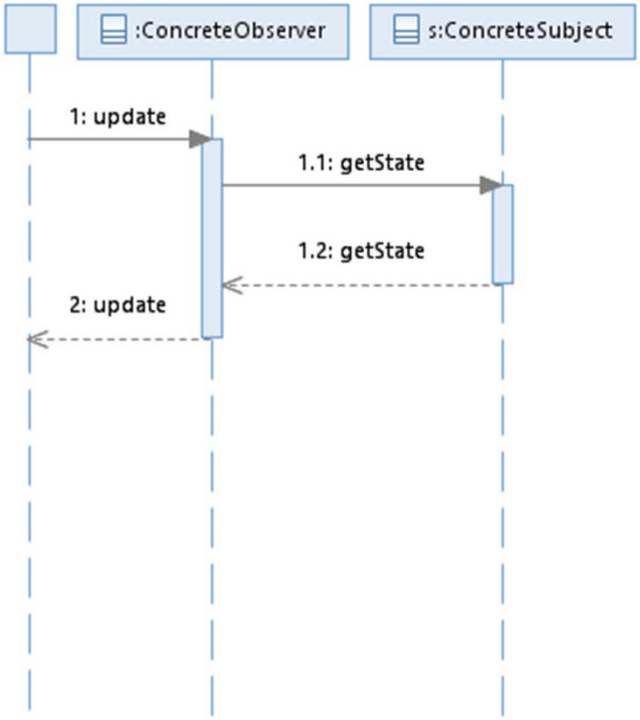
Estructura del patrón Observer

Patrones auxiliares
Observador



Interacción
en Observer

Patrones auxiliares
Observador



Interacción en Observer

Patrones auxiliares
Observador

- Consecuencias
  - Ventajas
    - Permite modificar objetos y observadores de manera independiente.
    - Acoplamiento abstracto entre sujeto y observador.
    - Capacidad de comunicación mediante difusión.
  - Inconvenientes
    - Actualizaciones inesperadas.
    - Protocolo de actualización simple.

Patrones auxiliares
Observador

- Código de ejemplo

```
public interface Observer {
  public void update(Observable o, Object arg);
};
```

Patrones auxiliares
Observador

```
public class Observable {
  public void addObserver(Observer o) {...}
  protected void clearChanged() {...}
  public int countObservers() {...}
  public void deleteObserver(Observer o) {...}
  public void deleteObservers() {...}
  public boolean hasChanged() {...}
  public void notifyObservers() {...}
  public void notifyObservers(Object arg) {...}
  public protected void setChanged() {...}
...
};
```

Patrones auxiliares
Observador

```
//versión naif de un controlador
class Controlador implements ActionListener{
  Modelo modelo;

  public Controlador(Modelo modeloP)
  { modelo= modeloP; }

  public void actionPerformed (ActionEvent e)
  { modelo.sumar(); }

};
```

Patrones auxiliares
Observador

```
class Modelo extends Observable {
   int valor;

   Modelo()
   { valor= 0; }

   void sumar()
   { valor++;
     notifyObservers(); //notify le pasa el objeto
   }
   int obtenerValor()
   { return valor; } };
```

Patrones auxiliares
Observador

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

   public Vista(Modelo modelo) {
   // crea e inicializa sus elementos
   ActionListener controlador= new Controlador(modelo);
   sumar.addActionListener(controlador);
   // termina de configurarse
   }
```

Patrones auxiliares
Observador

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

  public void activar()
  { setVisible(true); }
};
```

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.

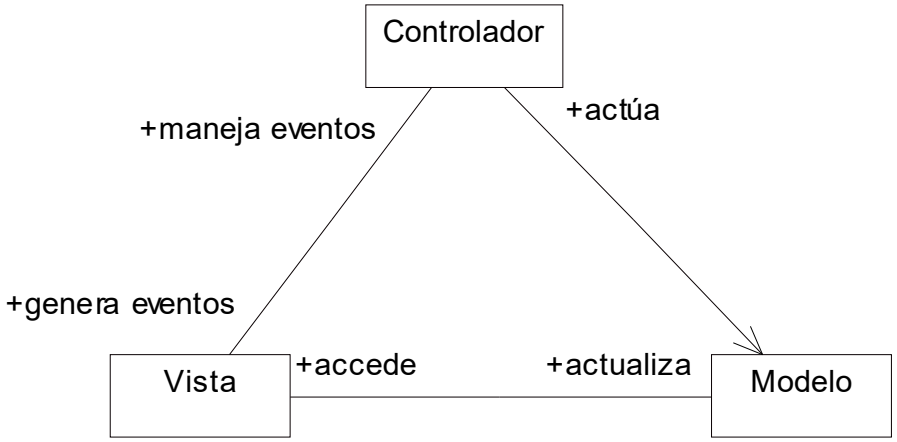
  - Los *controladores* median entre vistas y modelo

Patrones auxiliares
MVC

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

Patrones auxiliares
MVC

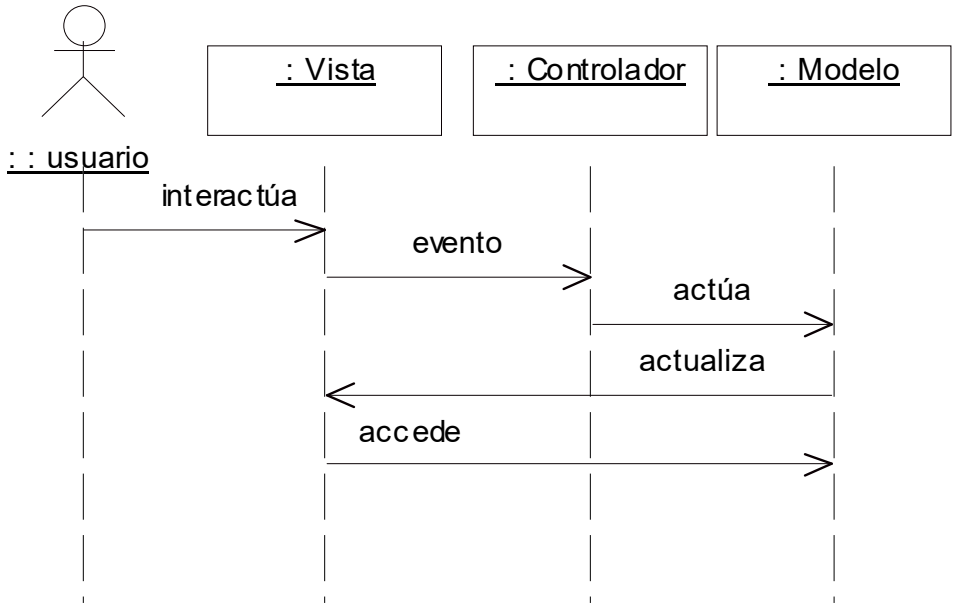
- Participantes en MVC. Modelo activo:



Participantes en MVC. Modelo activo

Patrones auxiliares
MVC

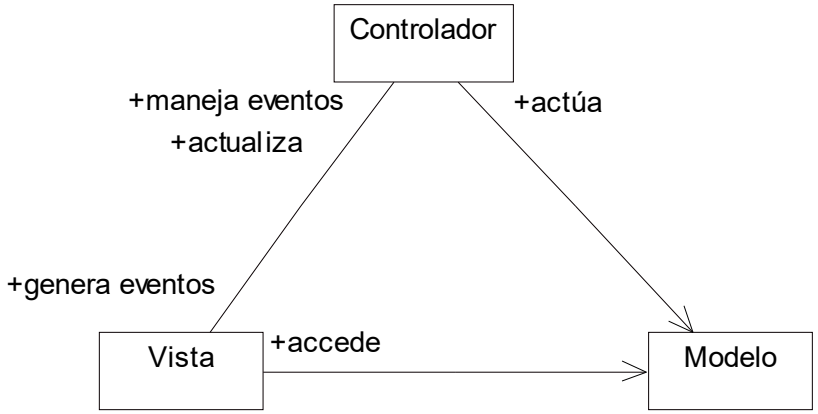
- Interacción en MVC. Modelo activo:



Interacción en MVC. Modelo activo

Patrones auxiliares
MVC

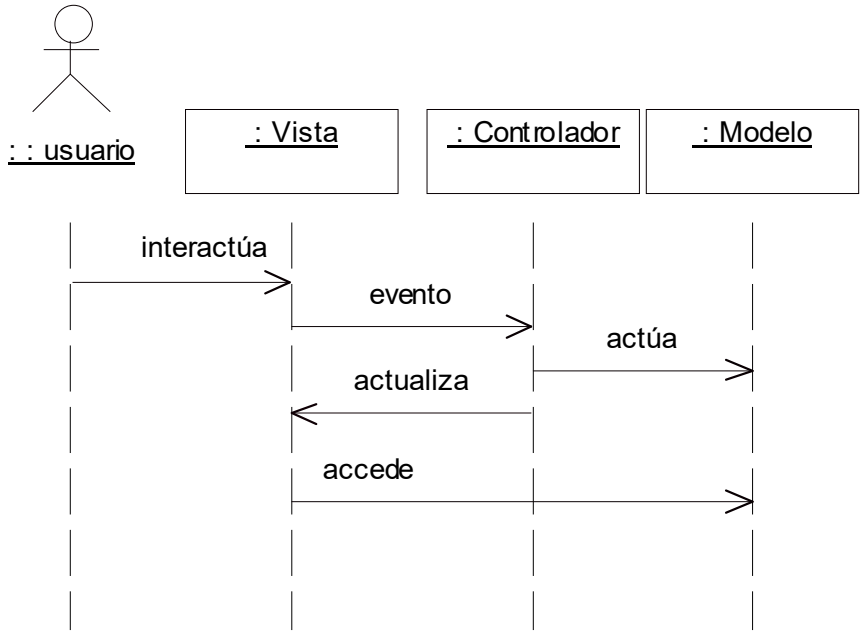
- Participantes en MVC. Modelo pasivo:



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

- Código de ejemplo*:

```
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);
       } ……}
```

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

Patrones auxiliares
MVC

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

Patrones auxiliares
MVC

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

Patrones auxiliares
MVC

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

Patrones auxiliares
MVC

```
class Vista extends JFrame implements IVista {
.........................
public void actualizar (Object o){
   Modelo modelo= (Modelo) o;
   Integer i= new Integer(modelo.obtenerValor());
   valor.setText(i.toString());     }
.........................
}
```

Patrones auxiliares
MVC

- En el ejemplo anterior:
  - 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 es poco razonable (e.g. aplicaciones web)

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 GUIAltaUsuario extends JFrame {
.....
   public GUIAltaUsuario()
   {   setTitle(“Alta usuario");
       JPanel panel= new JPanel();
       JLabel lNombre= new JLabel(“Nombre:");
       final JTextField tNombre= new JTextField(20);
       JLabel lEMail= new JLabel(“e-mail:");
       final JTextField tNombre= new JTextField(20);
       final JTextField tEMail= new JTextField(20);
       JButton aceptar= new JButton("Aceptar");
       JButton cancelar= new JButton("Cancelar");
```

*Controlador único, modelo pasivo

Patrones auxiliares
MVC

```
panel.add(lNombre);
panel.add(tNombre);
panel.add(lEMail);
panel.add(tEmail);
panel.add(aceptar);
panel.add(cancelar);
getContentPane().add(panel);

pack();
```

Patrones auxiliares
MVC

```
aceptar.addActionListener(new ActionListener()
  { public void actionPerformed(ActionEvent e)
      {   setVisible(false);
          String nombre= tNombre.getText();
          String eMail= tEMail.getText();
          TUsuario tU= new TUsuario(nombre , eMail);
          Controlador.getInstancia().
          accion(Eventos.ALTA_USUARIO, tU);
      }
                                                  });
.....
}
```

Patrones auxiliares
MVC

```
public  class Evento {

   public static final int ALTA_USUARIO= 101;
   public static final int BAJA_USUARIO= 102;
   public static final int MOSTRAR_USUARIO= 103;
   ……
   public static final RES_ALTA_USUARIO_OK= 401;
   public static final RES_ALTA_USUARIO_KO= 402;
   ……
}
```

Patrones auxiliares
MVC

```
public class Controlador {
   ……
   //esta es una opción de acceso del controlador
   //a los servicios y a la GUI
   //puede haber otras más avanzadas
   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 Evento.ALTA_USUARIO: {
      TUsuario tUsuario= (TUsuario) datos;
      int res= saUsuario.alta(tUsuario);
      if (res>0)
      gui.actualizar(Evento.RES_ALTA_USUARIO_OK,
            new Integer(res));
      else
      gui.actualizar(Evento.RES_ALTA_USUARIO_KO, null);
      break; }
   case Evento.BAJA_USUARIO: { …… }
.....
}
```

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 Evento.MOSTRAR_GUI_BIBLIOTECA:
  { setVisible(true); break; }
  case Evento.OCULTAR_GUI_BIBLIOTECA: {
  setVisible(false); break; }
```

.....

Patrones auxiliares
MVC

```
case EventoGUI.RES_ALTA_USUARIO_OK:
{ Integer id= (Integer) datos;
  JOptionPane.showMessageDialog(null,
      "Usuario creado con ID: "+id.intValue());
  setVisible(true);
  break; }

case EventoGUI.RES_ALTA_USUARIO_KO:
{ JOptionPane.showMessageDialog(null,
      "No se pudo crear al usuario");
  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

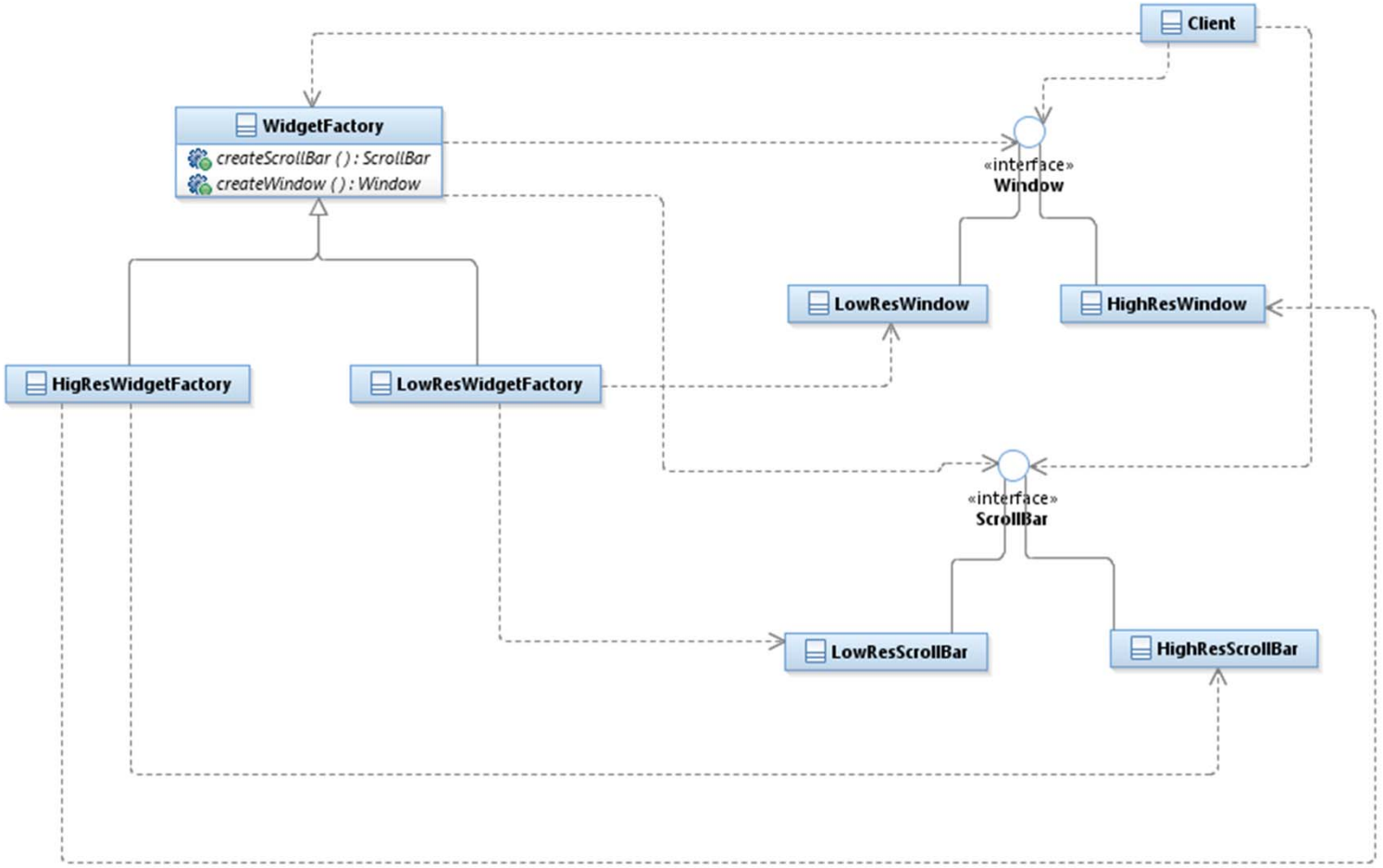
Patrones auxiliares
Factoría abstracta

- Propósito
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas
- También conocido como
  - Fábrica Abstracta
  - Kit

Patrones auxiliares
Factoría abstracta

- Motivación
  - Supongamos que deseamos tener una interfaz de usuario independiente de los objetos concretos que la componen.
  - Si la aplicación crea instancias de clases o útiles específicos de la interfaz de usuario será difícil cambiar ésta más tarde.

Patrones auxiliares
Factoría abstracta



Ejemplo de Abstract Factory

Patrones auxiliares
Factoría abstracta

- Debemos aplicar el patrón cuando:

  - Un sistema debe ser independiente de cómo se crean, componen y representan sus productos.

  - Un sistema debe ser configurado con una familia de productos de entre varias.

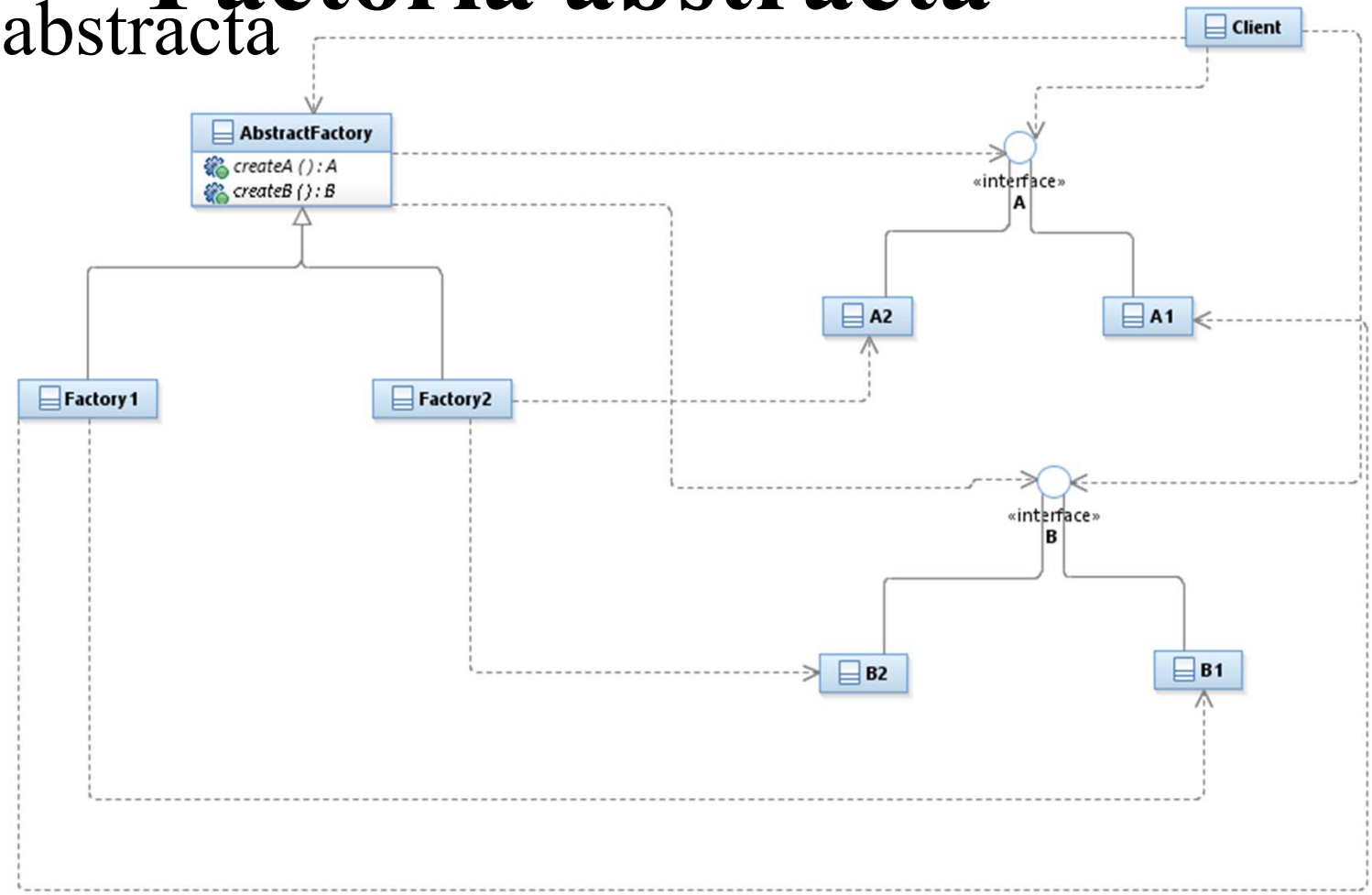
  - Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.

Patrones auxiliares
Factoría abstracta

- Quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones.

Patrones auxiliares
Factoría abstracta

- D. abstracta



Estructura de Abstract Factory

Patrones auxiliares
Factoría abstracta

- Consecuencias de Abstract Factory:
  - Ventajas:
    - Aísla las clases concretas de sus clientes.
    - Facilita el intercambio de familias de productos.
    - Promueve la consistencia entre productos.
  - Inconvenientes:
    - Es difícil dar cabida a nuevos tipos de productos, ya que hay que modificar FabricaAbstracta.

Patrones auxiliares
Factoría abstracta

- Cógido de ejemplo

  - Supongamos que deseamos construir un juego de laberintos.

  - Deseamos que los laberintos que construyamos no dependan de los objetos (e.g. pared) concretos que lo componen.

  - Así podemos tener distintos niveles, para los mismos escenarios.

Patrones auxiliares
Factoría abstracta

```
public interface FabricaDeLaberintos {
      public Laberinto hacerLaberinto();
      public Pared hacerPared();
      public Habitacion hacerHabitacion();
      public Puerta hacerPuerta();
};
```

Patrones auxiliares
Factoría abstracta

```
public class JuegoDelLaberinto {
.........
Laberinto crearLaberinto(FabricaDeLaberinto fabrica)
{
  Laberinto l= fabrica.hacerLaberinto();
  Habitacion h1= fabrica.hacerHabitacion();
  Habitacion h2= fabrica.hacerHabitacion();
  Puerta p= fabrica.hacerPuerta(h1, h2);

  l.anadirHabitacion(h1);
  l.anadirHabitacion(h2);
```

Patrones auxiliares
Factoría abstracta

```
h1.establecerLado(Norte, fabrica.hacerPared());
h1.establecerLado(Este, p);
h1.establecerLado(Sur, fabrica.hacerPared());
h1.establecerLado(Oeste, fabrica.hacerPared());

h2.establecerLado(Norte, fabrica.hacerPared());
h2.establecerLado(Este, fabrica.hacerPared());
h2.establecerLado(Sur, fabrica.hacerPared());
h2.establecerLado(Oeste, p)

return l; }
```

Patrones auxiliares
Factoría abstracta

```
class FabricaDeLaberintosEncantados implements
  FabricaDeLaberintos {
  .......
  Habitacion hacerHabitacion(int n)
  { return new HabitacionEncantada(n); }

  Puerta hacerPuerta(Habitacion h1, Habitacion h2)
  { return new PuertaEncantada(h1, h2); }
  ........
};
```

Patrones auxiliares
Factoría abstracta

```
class FabricaDeLaberintosExplosivos implements
  FabricaDeLaberintos {
  .......
  Habitacion hacerHabitacion(int n)
  { return new HabitacionExplosiva(n); }

  Puerta hacerPuerta(Habitacion h1, Habitacion
  h2)
  { return new PuertaExplosiva(h1, h2); }
  ........
};
```

Patrones auxiliares
Factoría abstracta

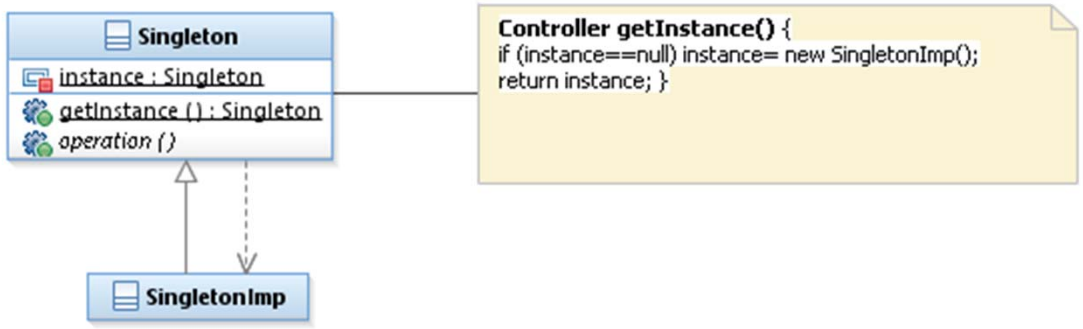
```
JuegoDelLaberinto juego= new JuegoDelLaberinto();
FabricaDeLaberintosExplosivos fabrica = new
  FabricaDeLaberintosExplosivos();

juego.crearLaberinto(fabrica);
```

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

- Descripción abstracta



Estructura y comportamiento 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 FactoriaIntegración {
   private static FactoriaIntegración instancia;

   public static FactoriaIntegracion obtenerInstancia()
   { if (instancia== null)
       instancia = new FactoriaIntegracionImp();
     return instancia;
   }
   public abstract DAOUsuario generaDAOUsuario();
   public abstract DAOLibro generaDAOLibro();
   public abstract DAOEjemplar generaDAOEjemplar();
   public abstract DAOPrestamo generaDAOPrestamo();
}
```

Patrones auxiliares
Singleton

```
public class FactoriaIntegracionImp extends
   FactoriaIntegracion {
   public abstract DAOUsuario generaDAOUsuario()
   { return new DAOUsuarioImp(); }

   public abstract DAOLibro generaDAOLibro()
   { return new DAOLibroImp(); }

   public abstract DAOEjemplar generaDAOEjemplar()
   { return new DAOEjemplarImp(); }

   public abstract DAOPrestamo generaDAOPrestamo()
   { return new DAOPrestamoImp(); }
}
```

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

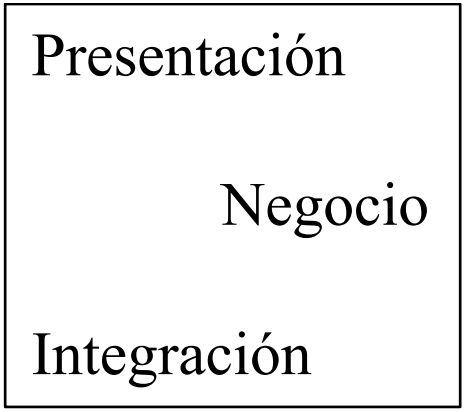
  - Su método de generación lee de un archivo la clase concreta que implementa a dicha factoría y que debe generar, la carga dinámicamente y se la devuelve al cliente

    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)

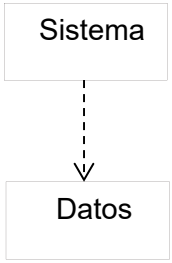
Arquitectura de una capa Características

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



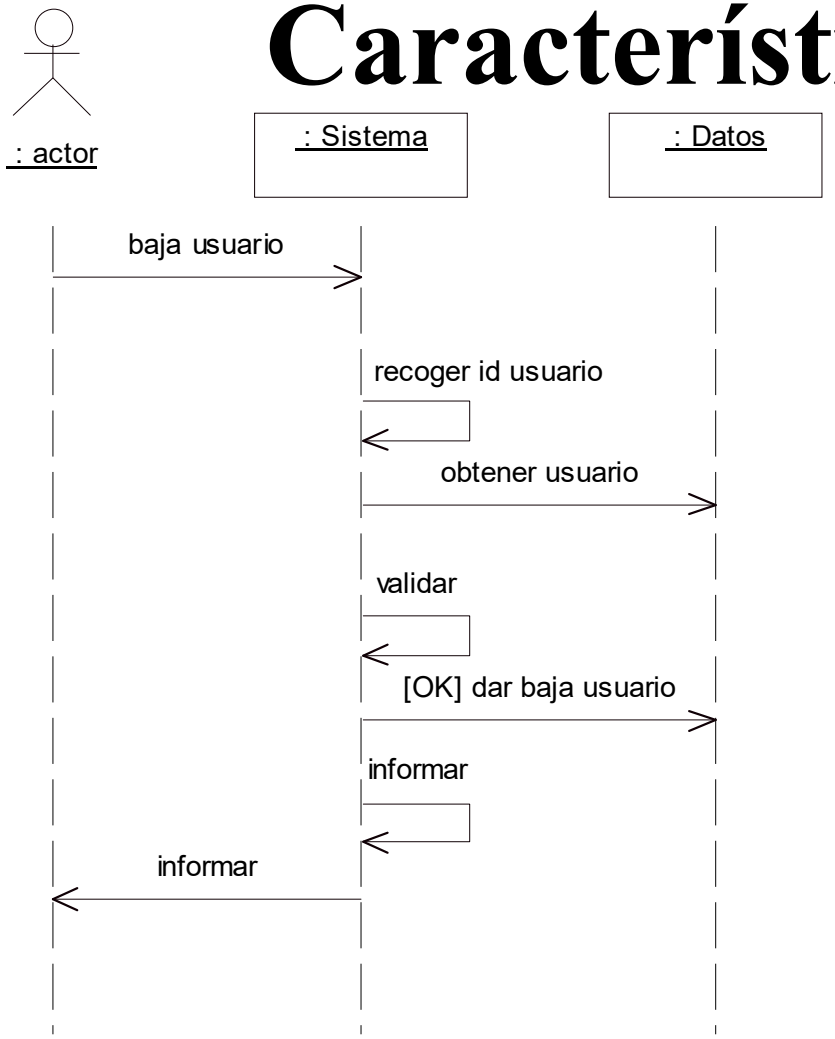
Arquitectura de una capa

Arquitectura de una capa
Características



Clases del sistema

Arquitectura de una capa
Características



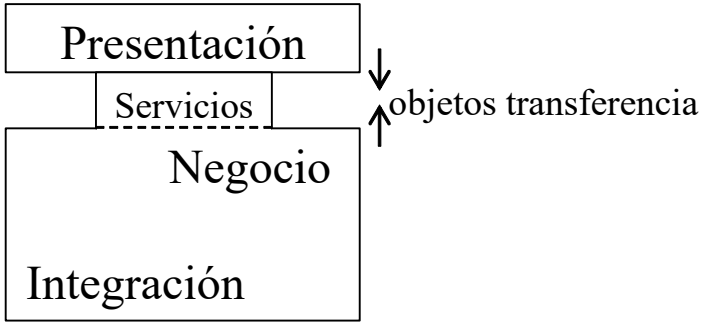
Comportamiento del sistema

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

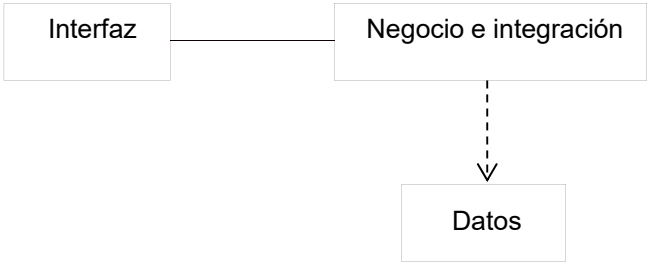
Arquitectura de dos capas Características

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



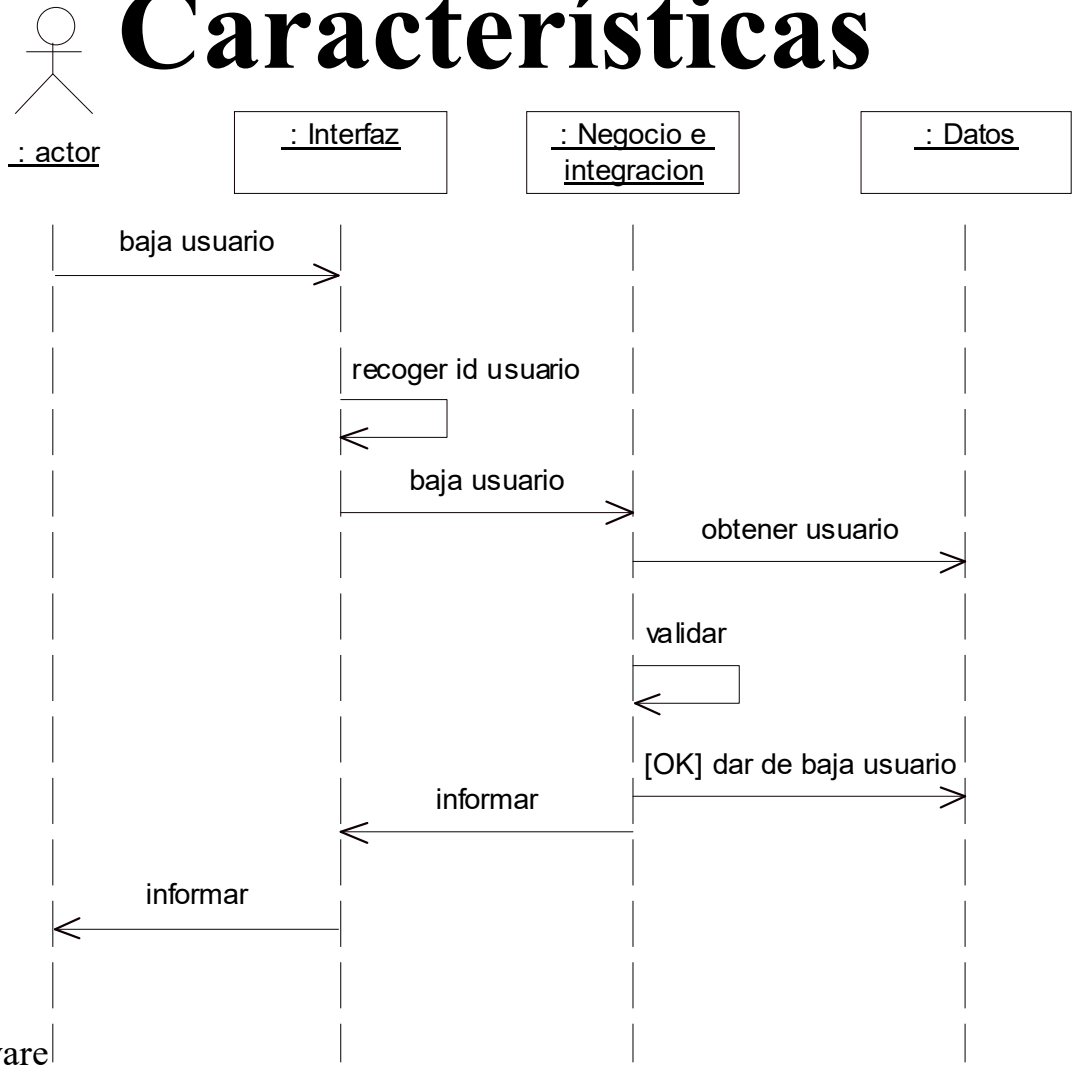
Arquitectura de dos capas

Arquitectura de dos capas
Características



Clases del sistema

Arquitectura de dos capas
Características



Comportamiento del sistema

Arquitectura de dos capas Ventajas e inconvenientes

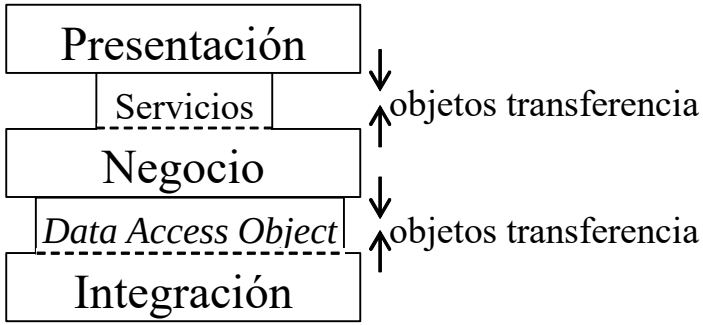
- Ventajas
  - Permite cambios en el interfaz de usuario o en el resto del sistema sin interferencias mutuas
  - Simplicidad fáctica
- Inconvenientes
  - Mayor complicación arquitectónica que la arquitectura de una capa
  - No se puede modificar la lógica del negocio o la representación de los datos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

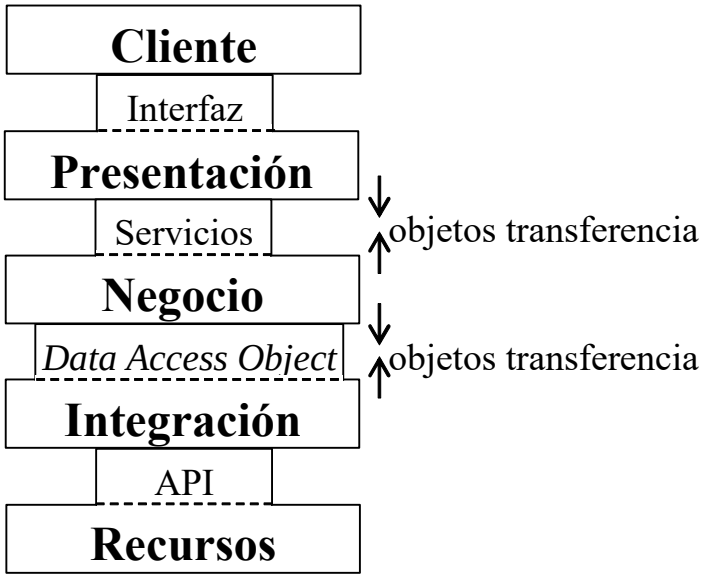
Arquitectura multicapa
Características

- 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

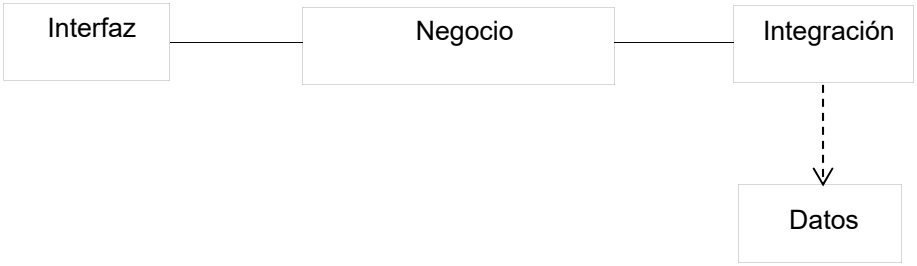
- En realidad, la arquitectura es de *cinco capas*, ya que incluye las capas de clientes y recursos
- La *capa de clientes* representa a todos los dispositivos o clientes del sistema que acceden al mismo. Está sobre la capa de presentación
- La *capa de recursos* contiene los datos del negocio y recursos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

Arquitectura multicapa
Características



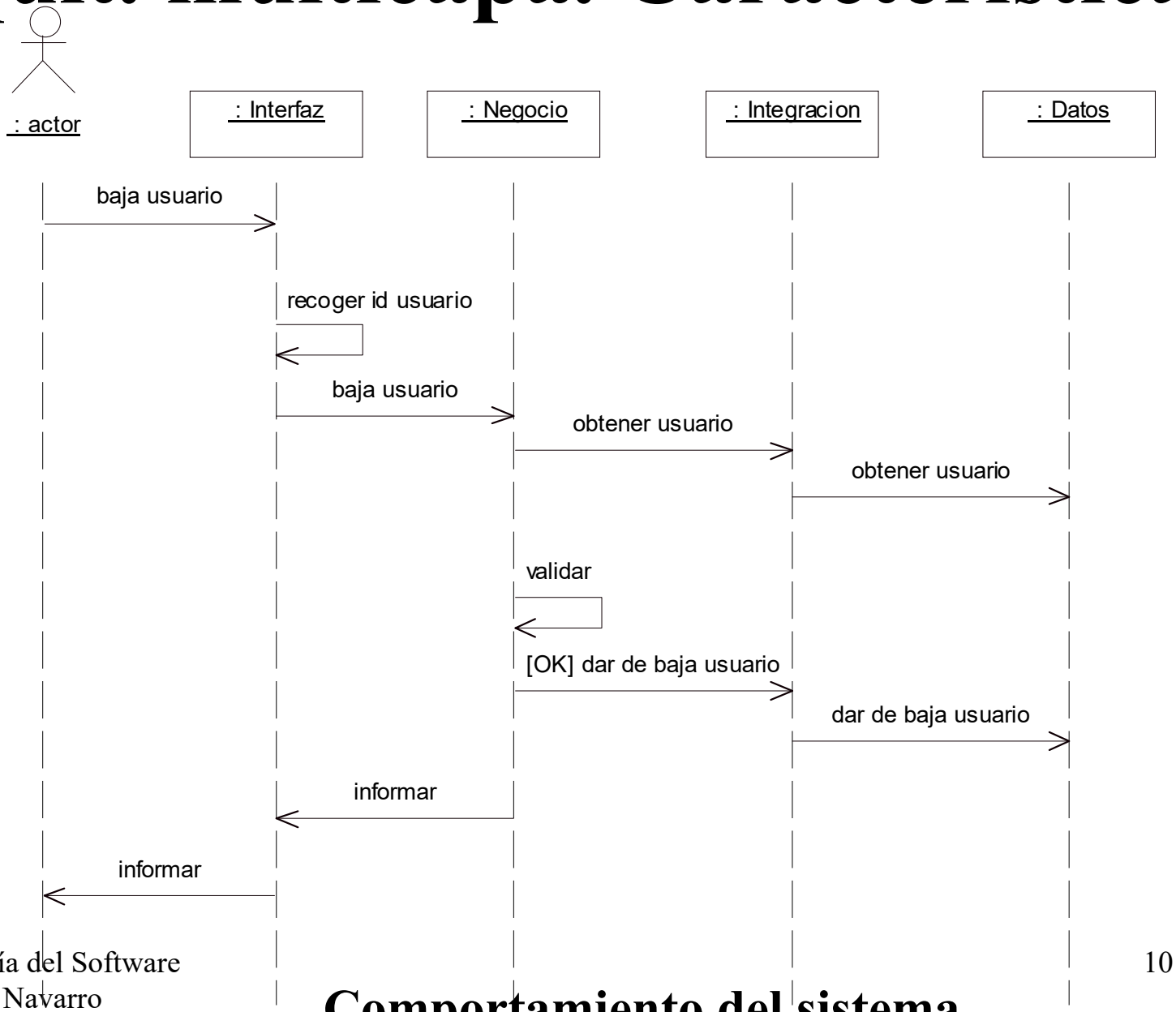
Arquitectura multicapa

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:
  - Presentación
    - Controlador frontal
    - Controlador de aplicación
  - Negocio
    - Transferencia
    - Servicio de aplicación
    - *Transfer Object Assembler*
    - Delegado del negocio
    - Objeto del negocio

Arquitectura multicapa
Patrones relacionados

- Integración
  - *Data Access Object* (DAO)
  - Almacén del dominio

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

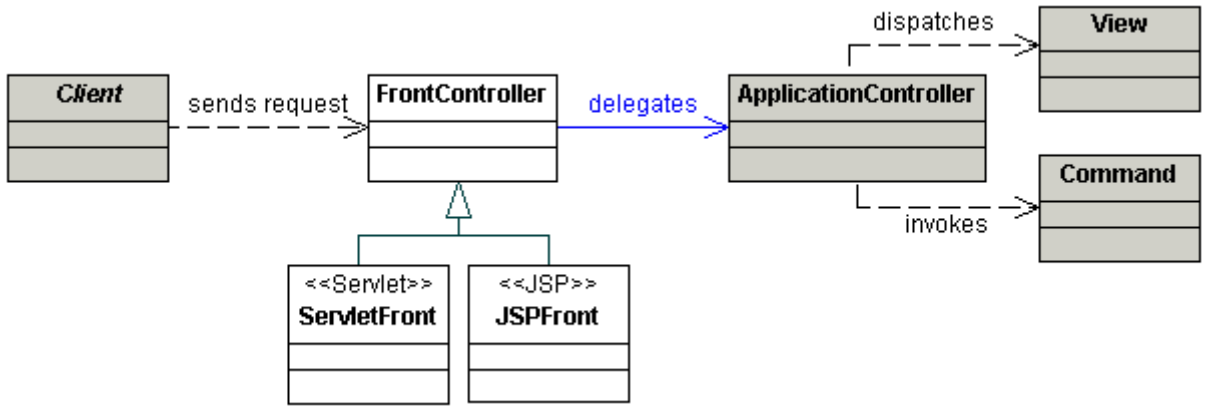
- Motivación
  - Se desea evitar lógica de control duplicada
  - Se desea aplicar una lógica común a distintas peticiones
  - Se desea separar la lógica de procesamiento del sistema de la vista
  - Se desea tener puntos de acceso centralizado y controlado al sistema

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

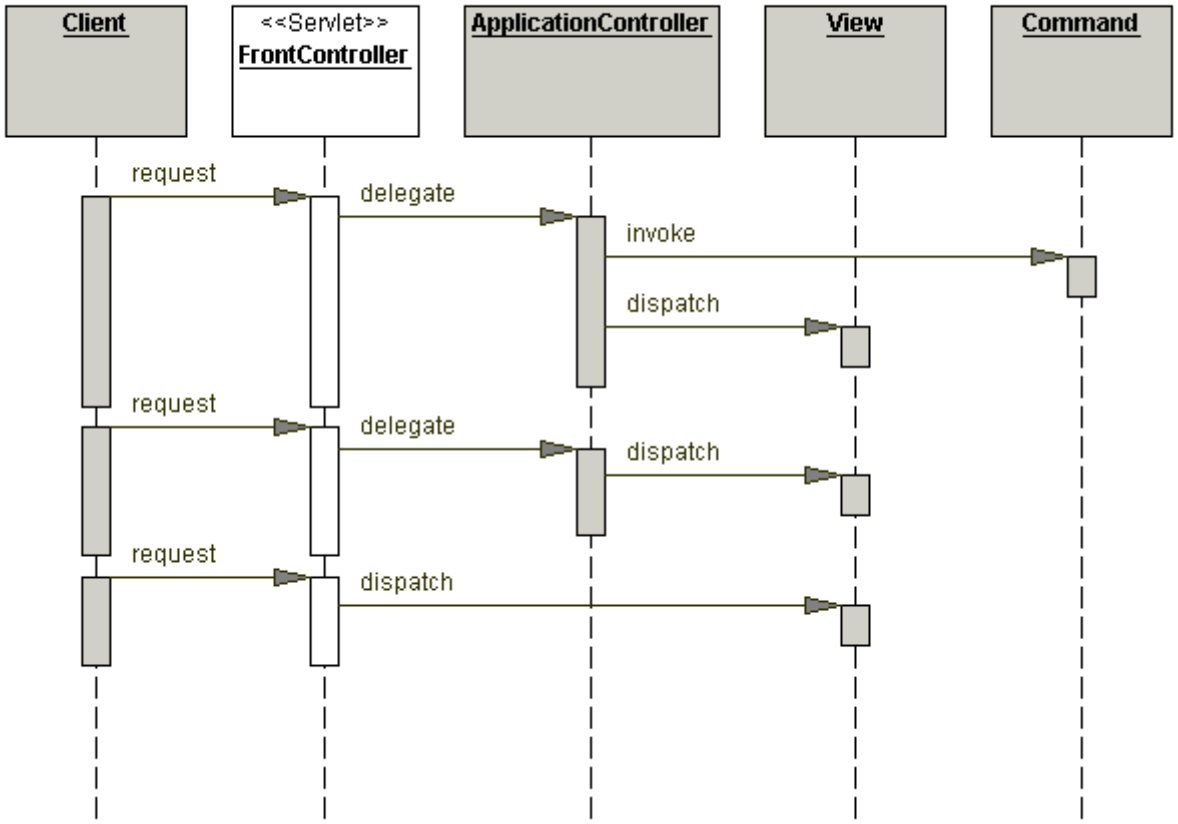
Patrón controlador frontal

- 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

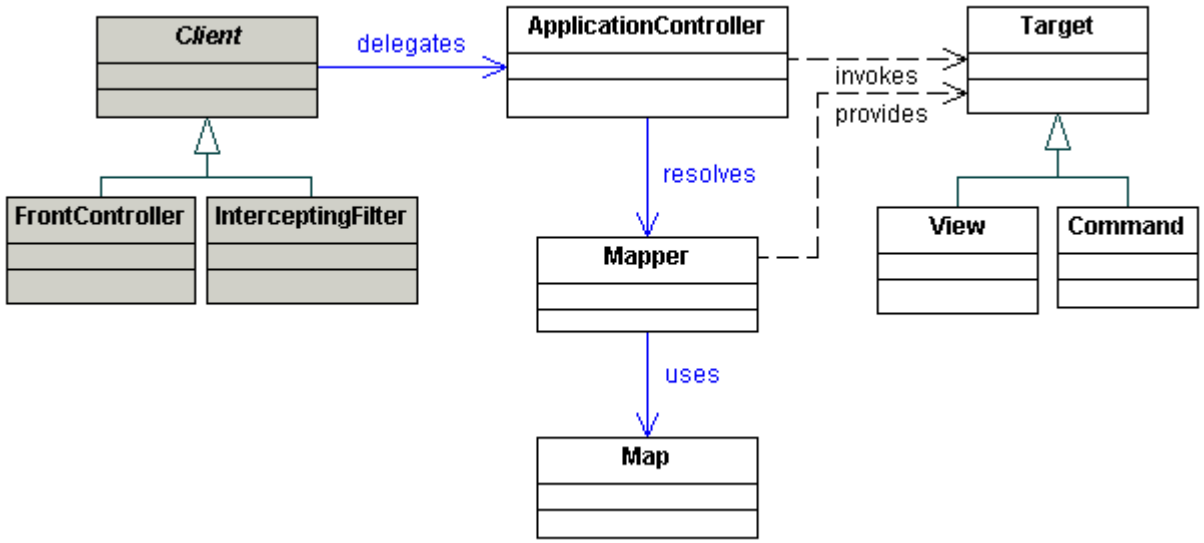
- Se desea mejorar la modularidad del código y la mantenibilidad, facilitando al extensión de la aplicación y la prueba del código de manejo de peticiones de manera independiente del contenedor web

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

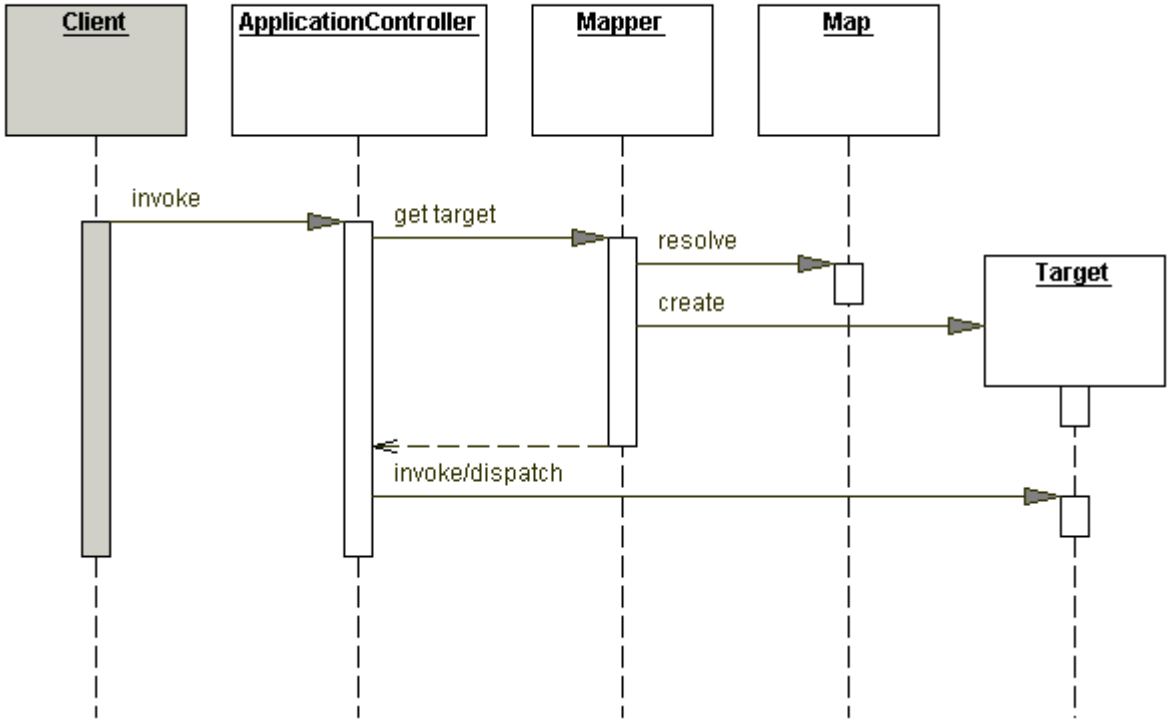
Patrón controlador de aplicación

- 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

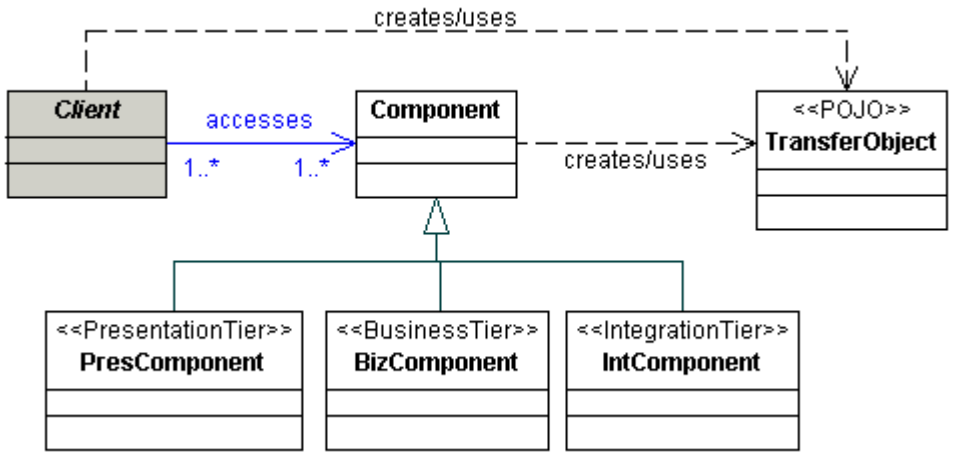
- Motivación
  - Si queremos independizar las capas, éstas no pueden tener conocimiento de la representación de las entidades de nuestro sistema dentro de cada capa
  - Por ejemplo, si accedemos a bases de datos relacionales, los clientes deberían abstrae de la existencia de *columnas* en los datos

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

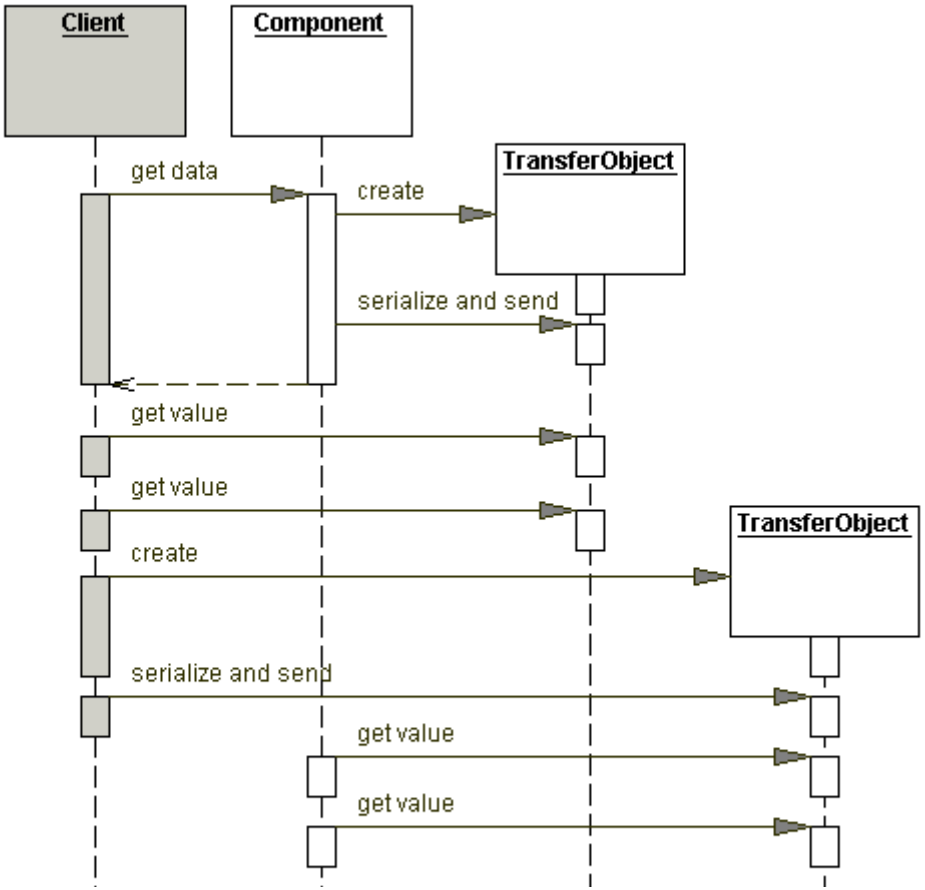
Patrón tranferencia

- 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 TUsuario implements Serializable {
   public int id;
   public String nombre;
   public String eMail;
   public boolean activo;

   public TUsuario(String nombre, String eMail)
   {  this.id=0; this.nombre= nombre;
      this.eMail= eMail; this.activo= true; }

   public TUsuario(int id, String nombre, String
                     eMail, boolean activo)
   {  this.id= id; this.nombre= nombre;
      this.eMail= eMail; this.activo= activo; }
```

Patrón transferencia

```
public int getId()
{ return id; }

public String getNombre()
{ return nombre; }

public String getEMail()
{ return eMail; }

public boolean getActivo()
{ return activo; }
```

Patrón transferencia

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

   public void setNombre(String nombre)
   { this.nombre= nombre; }

   public void setEMail(String eMail)
   { this.eMail= eMail; }

   public void setActivo(boolean activo)
   { this.activo= activo; }

}
```

Patrón transferencia

```
public DAOUsuarioImp implements DAOUsuario {

   public TUsuario read (int id)
   {
       //código acceso a la base de datos

       TUsuario usuario=
             new TUsuario(id, nombre, eMail, activo);

       return usuario;
   }
   ..............
}
```

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

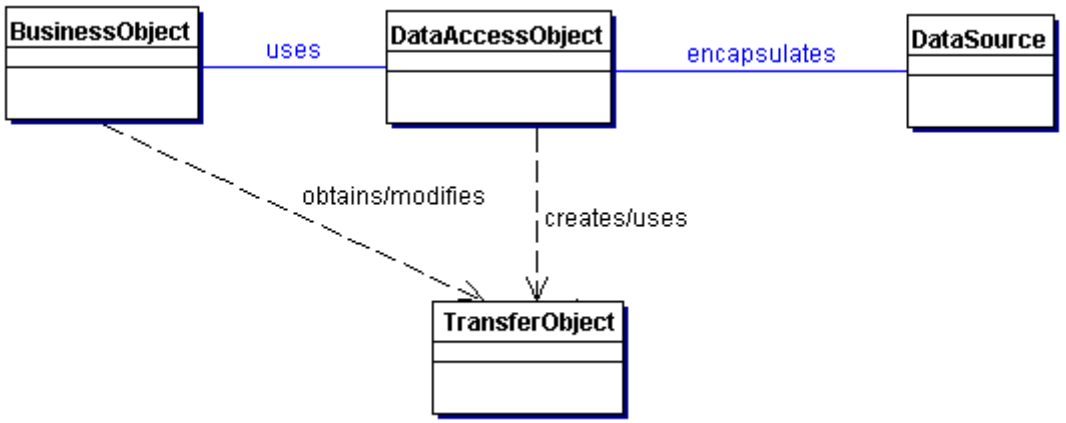
- Manejar estos datos fuerza a:
  - Conocer los mecanismos de acceso del sistema de gestión de datos (p.e., base de datos, sistema operativo, etc.)
  - Conocer la representación de los datos en el sistema de gestión de datos (p.e., columnas, elementos, bytes, etc.)
- Un cliente de la capa de negocio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

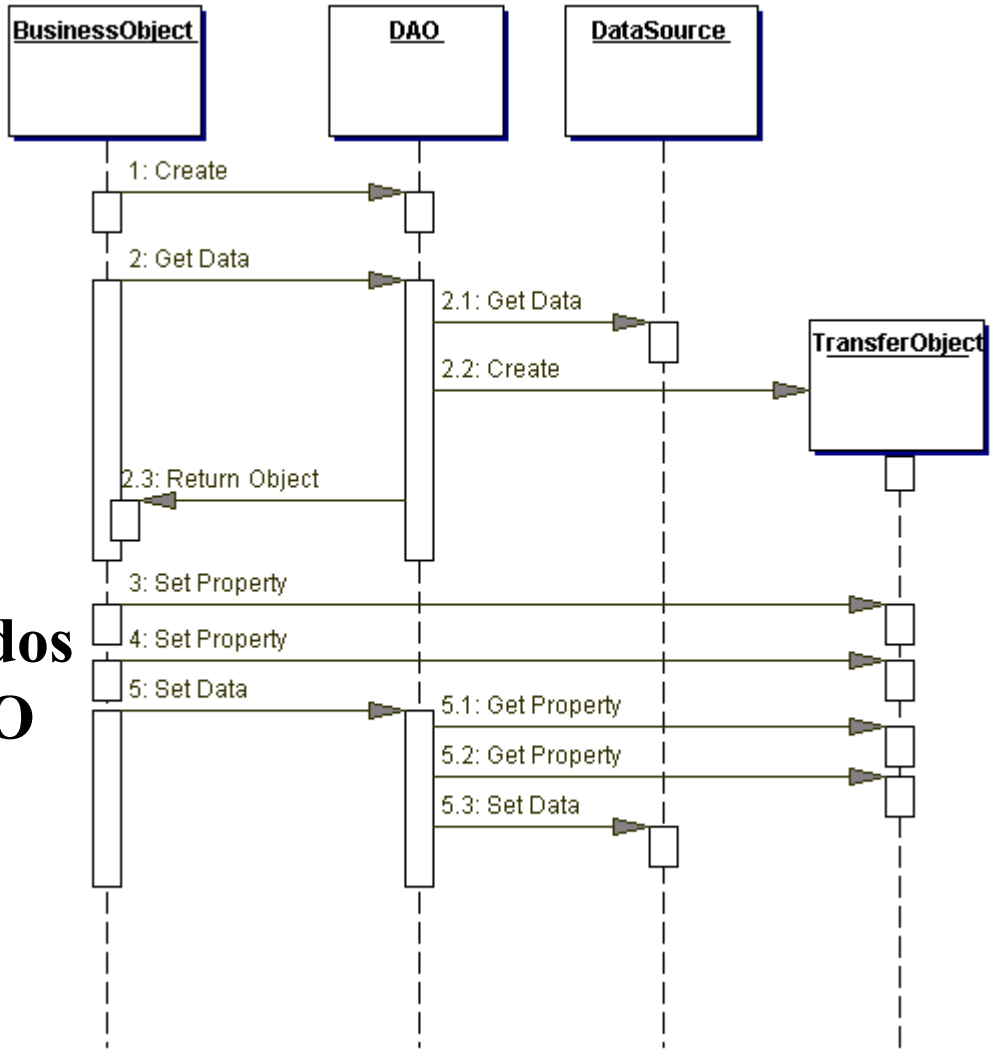
- Estructura



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 {
   public Integer create(TUsuario tUsuario);
   public TUsuario read(Integer id);
   public Collection<TUsuario> readAll();
   public TUsuario readByName(String nombre);
   public Integer update(TUsuario tUsuario);
   public Integer delete (Integer id);

}
```

Patrón DAO

```
public class DAOUsuarioImp implements DAOUsuario {
……
   public int create(TUsuario tUsuario)
   {
       int id= -1;
       //conexión con la base de datos
       PreparedStatement ps;

        ps = conexion.prepareStatement(“INSERT INTO
   usuario (nombre, eMail, activo) VALUES (?,?,?)");
       ps.setString(1, tUsuario.getNombre());
       ps.setString(2, tUsuario.getEMail());
       ps.setBoolean(3, tUsuario.getActivo());
       ps.execute();
```

Patrón DAO

```
 ps = conexion.prepareStatement("SELECT LAST_INSERT_ID()");
 ResultSet rs = ps.executeQuery();
 if (rs.next()) id=rs.getInt("LAST_INSERT_ID()"));


//cerrar conexión y tratar excepciones

 return id;
 }
 ……
 }
```

Patrón DAO

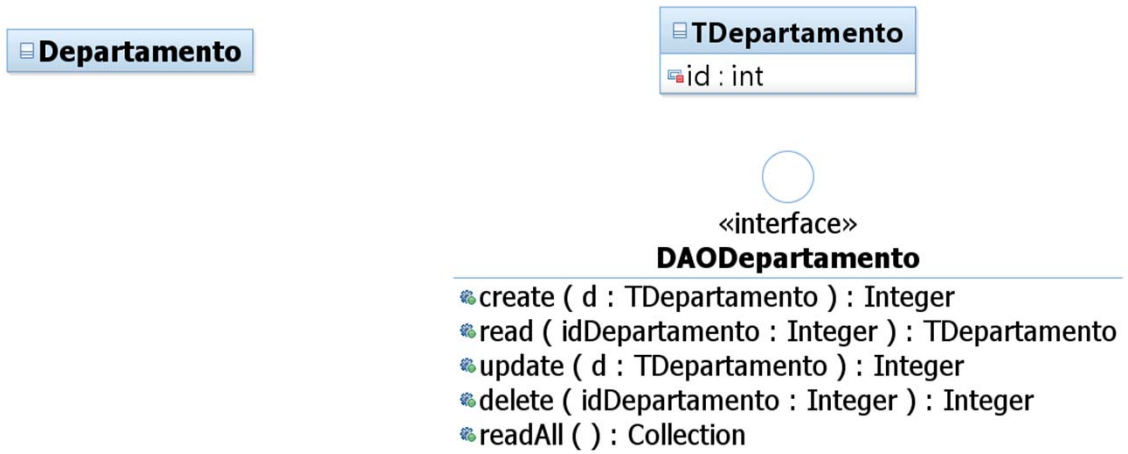
- Aunque en estas transparencias se obvia, es fundamental que los DAOs capturen y lancen las excepciones correspondientes al acceder a los recursos externos
- Así, la capa de negocio sabrá qué ha sucedido si ha habido algún tipo de fallo en dicho acceso

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 relaciones 1..n y m..n

Patrón DAO

– Para una clase:

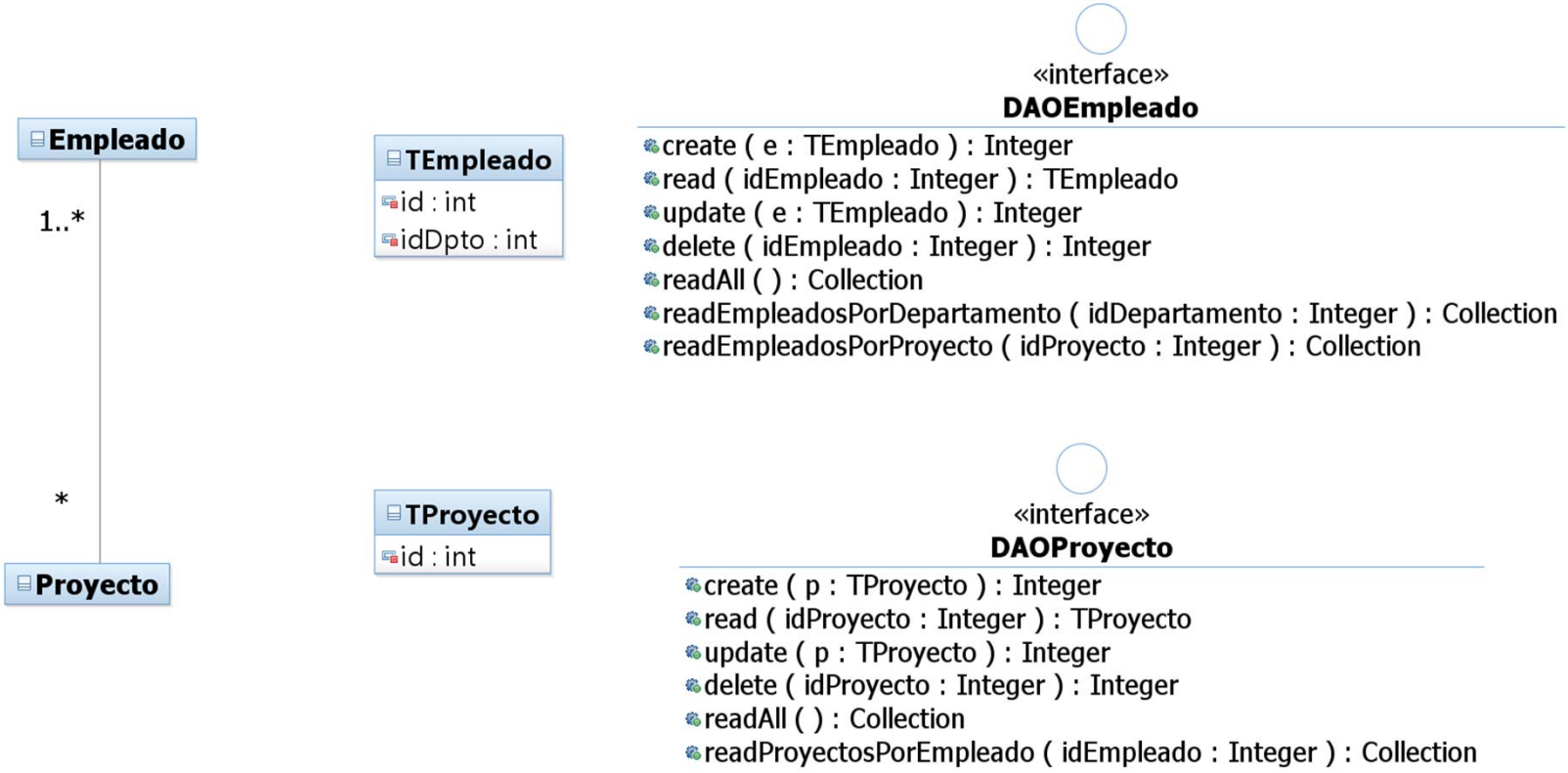


Patrón DAO

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

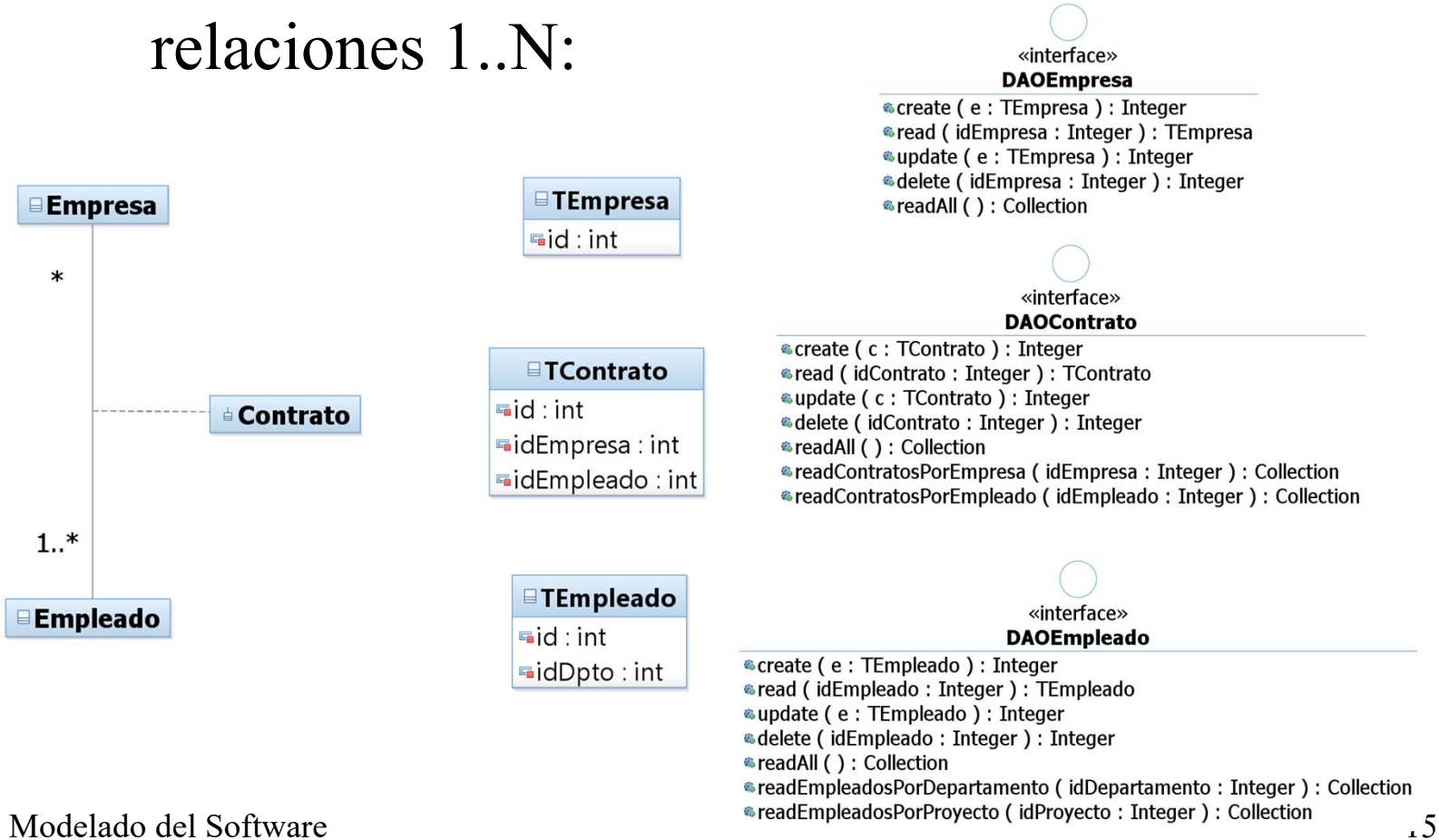
Patrón DAO

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



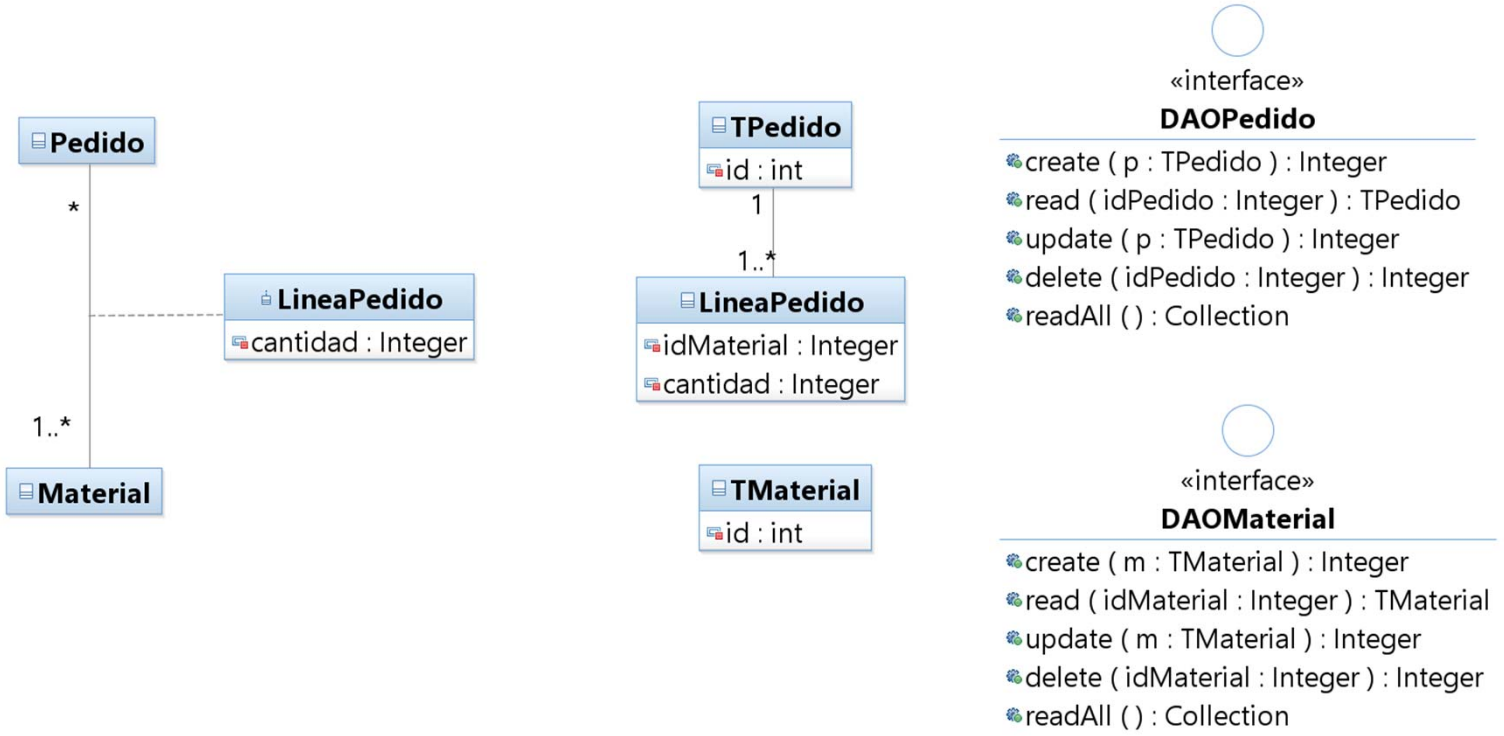
Patrón DAO

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



Patrón DAO

– Salvo que pensemos que la clase intermedia no tiene entidad suficiente (i.e. no tiene ID propio)

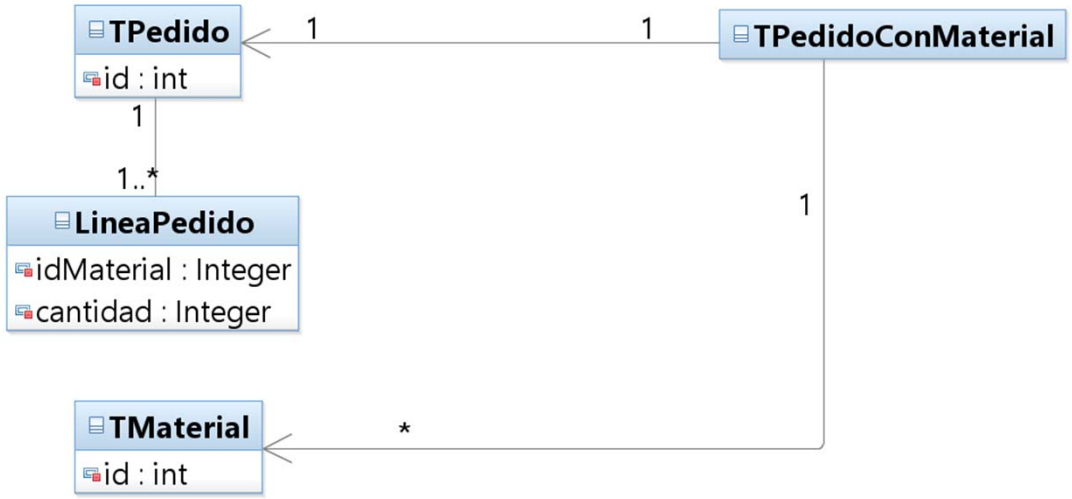


Patrón DAO

- En este último caso, si las líneas de pedido, a pesar de no ser accedidas fuera del transfer pedido, sí que son actualizadas con frecuencia (p.e. por cambios en los pedidos), quizás si convendría darles DAO, pero no servicio de aplicación

Patrón DAO

- Si con el pedido, quisiéramos sacar todos los datos del material, necesitaríamos un TOA (e.g. una operación del SA)

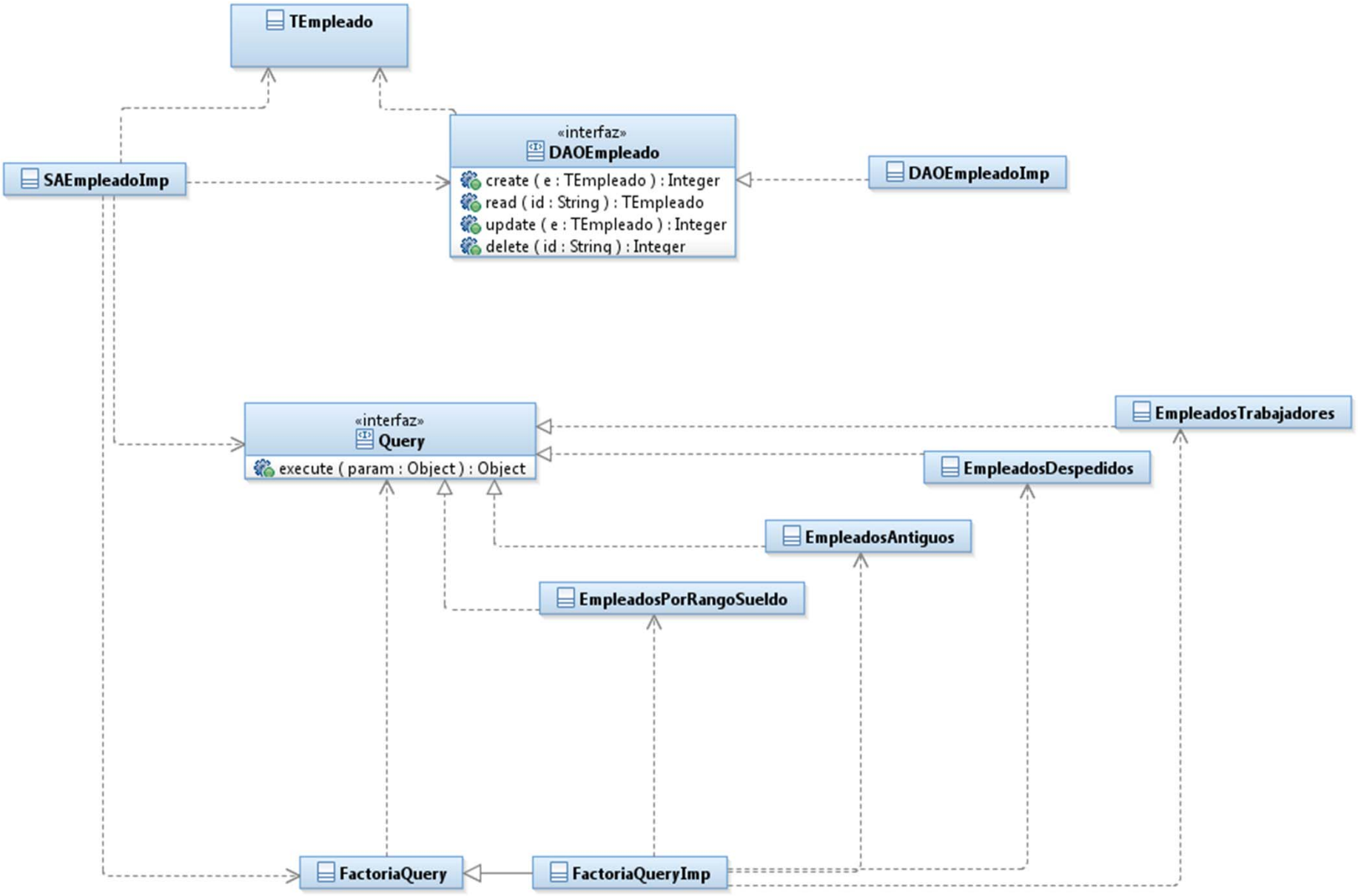


Patrón DAO

– Martin Fowler es más pragmático y devolvería una red de transfers creada bajo demanda de la unión de casos de usos

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 TOA

- Propósito
  - Se desea obtener un modelo de aplicación que agregue objetos transferencia de distintos componentes de negocio
- También conocido como
  - Transfer Object Assembler
  - Ensamblador de objetos transferencia

Patrón TOA

- Motivación
  - Los clientes de aplicación necesitan obtener datos de negocio o un *modelo de aplicación* de la capa de negocio, bien para presentarlo, bien para hacer un procesamiento intermedio
  - Un modelo de aplicación representa datos de negocio encapsulados por componentes de negocio en la capa de negocio

Patrón TOA

- Cuando los clientes necesitan los datos del modelo de aplicación deben localizar, acceder y obtener diferentes partes del modelo de diferentes fuentes, tales como objetos del negocio, DAOs, servicios de aplicación y otros
- Esta aproximación
  - Acopla a los clientes con diversos componentes de la aplicación
  - Puede provocar la duplicación de código en distintos clientes accediendo a los mismos datos

Patrón TOA

- Contexto
  - Se desea encapsular lógica de negocio de forma centralizada, evitando su implementación en el cliente
  - Se desea minimizar las llamadas a objetos remotos al construir una representación de datos de modelos de objetos de la capa de negocio

Patrón TOA

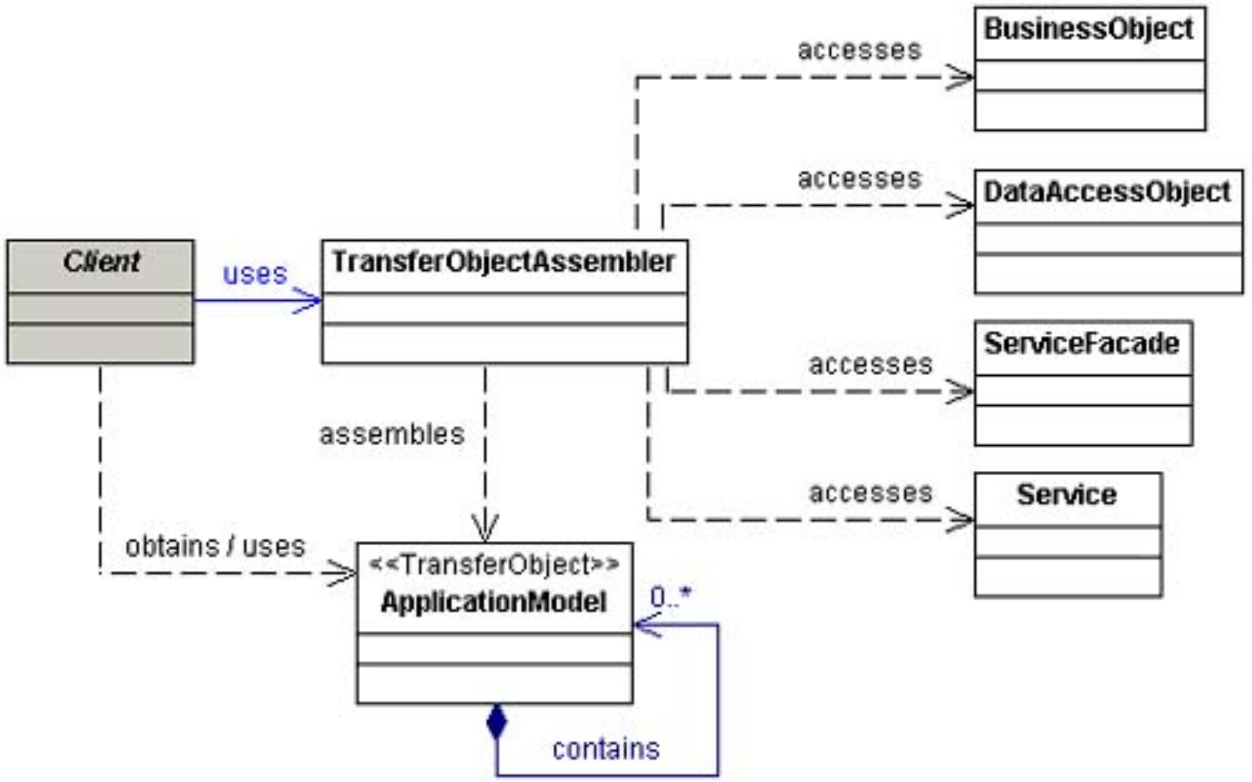
- Se desea crear un modelo complejo para dárselo al cliente en presentación
- Se desea que los clientes sean independientes de la complejidad del modelo de implementación, reduciendo el acoplamiento entre el cliente y los componentes de negocio

Patrón TOA

- Solución
  - Utilizar un TOA para construir un modelo de aplicación como un objeto transferencia compuesto
  - El TOA agrega distintos objetos transferencia provenientes de diversos componentes y lo devuelve al cliente

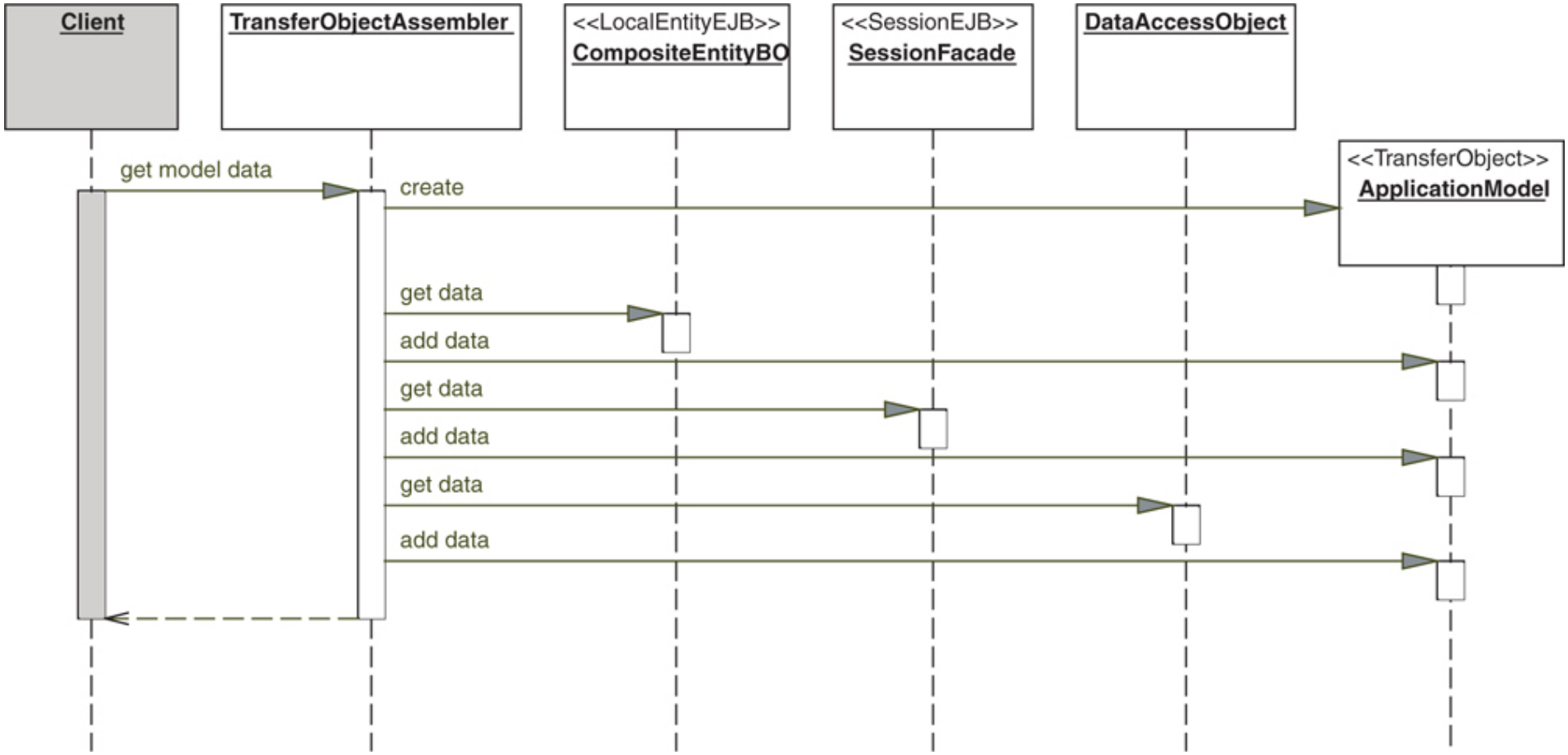
Patrón TOA

- Descripción



Estructura del patrón TOA

Patrón TOA



Interacción en el patrón TOA

Patrón TOA

- Consecuencias
  - Ventajas
    - Separa la lógica de negocio, simplifica la lógica de cliente
    - Reduce el acoplamiento entre clientes y el modelo de aplicación
    - Mejora el rendimiento de la red
    - Mejora el rendimiento del cliente
  - Inconvenientes
    - Puede introducir datos desactualizados

Patrón TOA

- Ejemplo

```
public class TBestOfShop {
 protected TCustomer bestCustomer;
 protected TProduct bestProduct;
 protected TBrand bestBrand;
……..
}
```

Patrón TOA

```
public class ShopTOA {

 TBestOfShop bestOfShop()
{  ……..
    TCustomer customer= CustomerAS.best();
    TProduct product= ProductAS.best();
    TBrand brand= BrandAS.best();

   return new TBestOfShop(customer, product, brand); }

 …… }
```

Patrón servicio de aplicacion

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

Patrón servicio de aplicacion

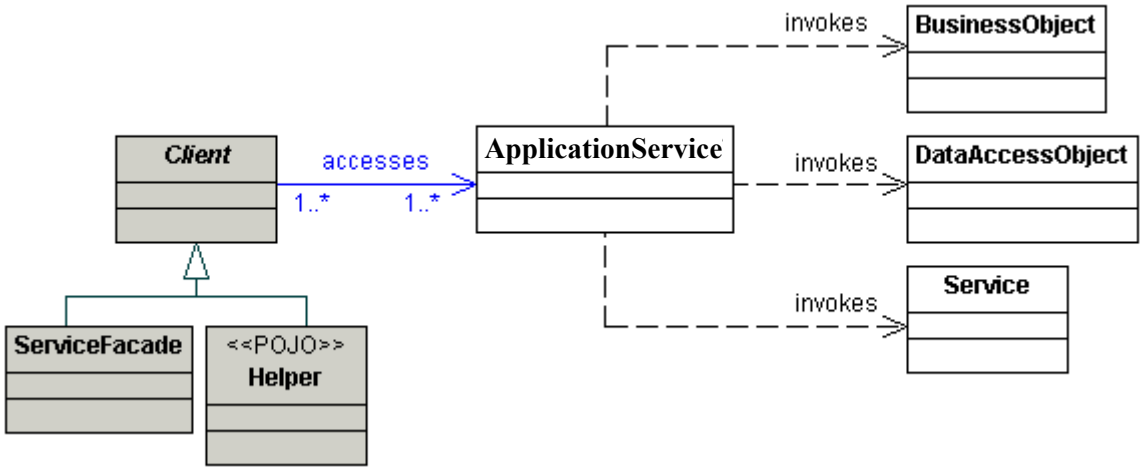
- Motivación
  - En una arquitectura multicapa, la lógica del negocio debe estar en algún sitio.
  - Ponerla en el controlador acoplaría presentación y negocio
  - Ponerla en el DAO acoplaría negocio e integración
  - Por eso la incluimos en los servicios de la aplicación.

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

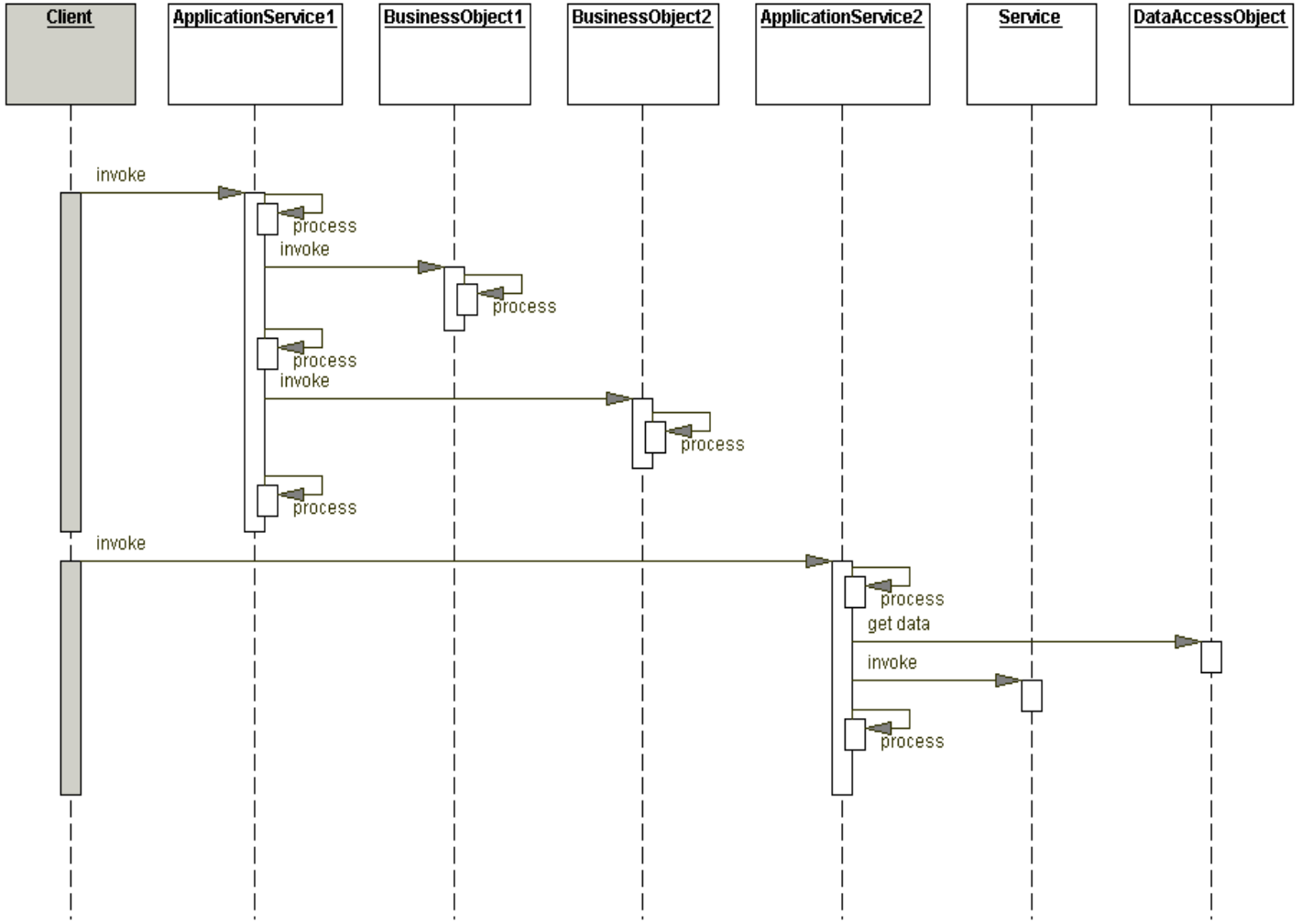
- Estructura



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:

```
public interface SAUsuario {
   public Integer create(TUsuario tUsuario);
   public TUsuario read(Integer id);
   public Collection<TUsuario> readAll();
   public Integer update(TUsuario tUsuario);
   public Integer delete (Integer id);

}
```

Patrón servicio de aplicacion

```
public class SAUsuarioImp implements SAUsuario {
   public Integer create(TUsuario tUsuario)
   {   int id= -1;
       DAOUsuario daoUsuario;
       if (tUsuario!=null)
       { //acceso a la implementación del DAO
          TUsuario leido=
             daoUsuario.readByName(tUsuario.getNombre());
          if (leido==null) id= daoUsuario.create(tUsuario);
        }
       return id;
    }
……..……}
```

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 (t174), 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

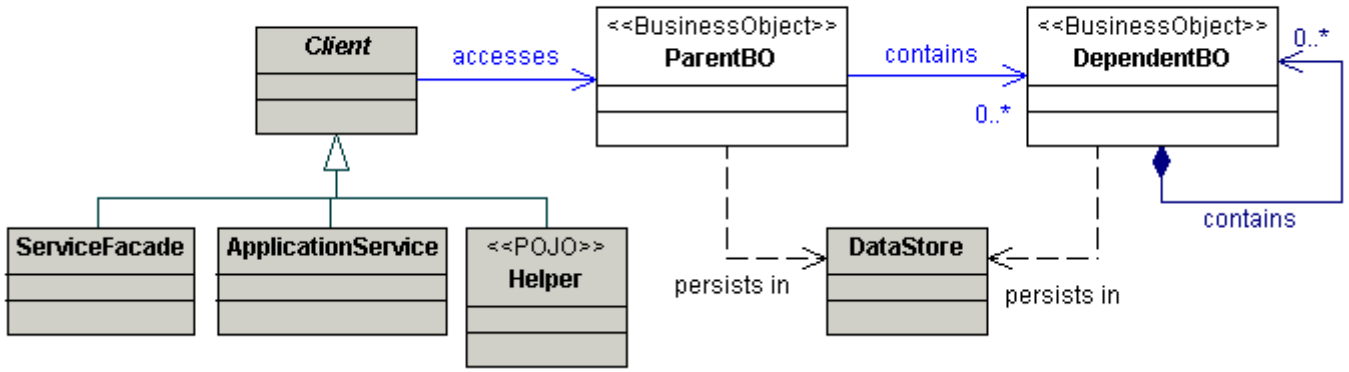
– Sin embargo, si en el cliente hay una gran cantidad de procesos computacionales asociados a los datos, dichos procesos deberían encapsularse en un objeto que representase un objeto del negocio

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

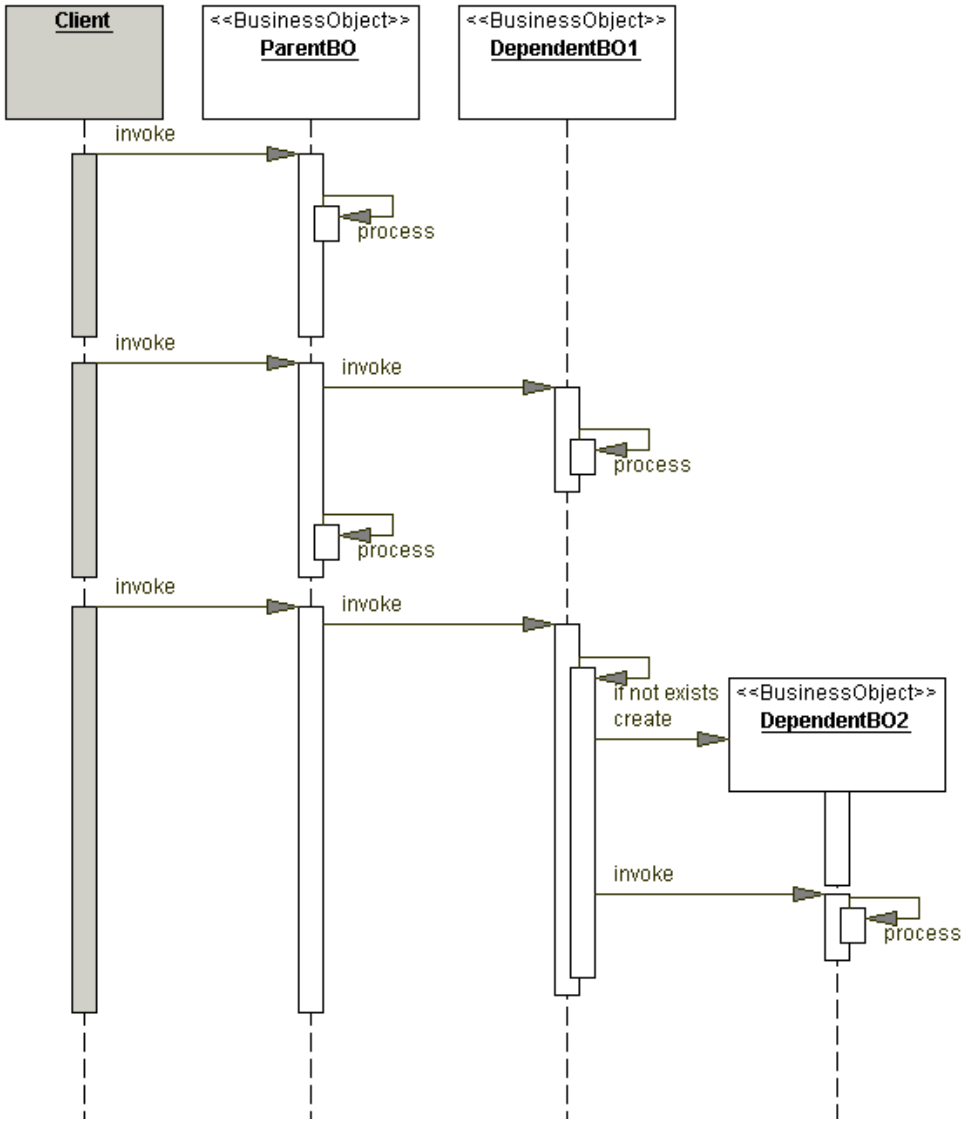
- Estructura

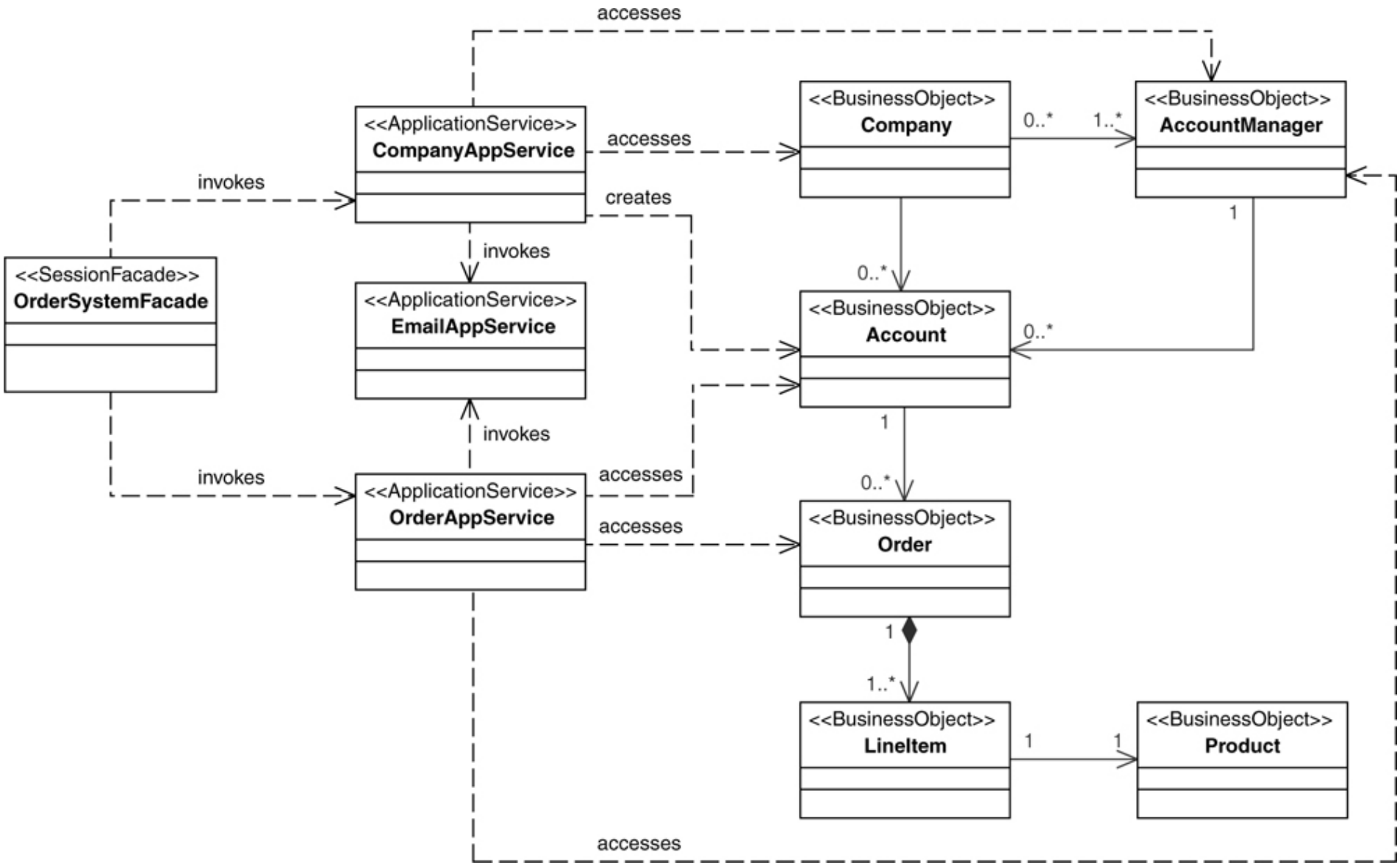


Estructura del patrón objeto del negocio

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

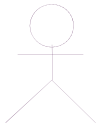
```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;
    private long salary;

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

Patrón objeto del negocio

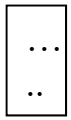
```
@Entity
public class Department {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String name;

    public int getId() {
        return id;     }

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

Nota

Realidad

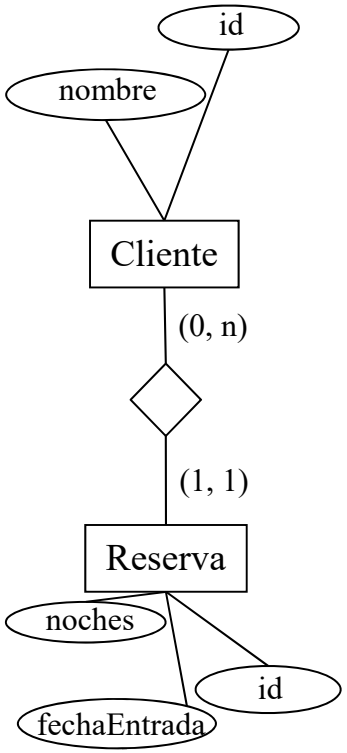


Ana

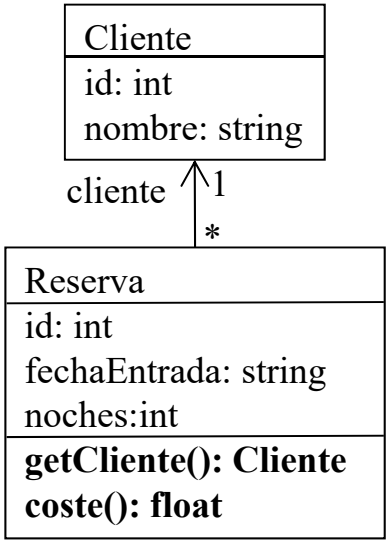


reserva

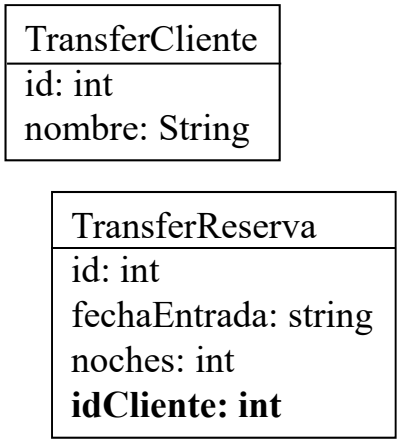
Modelo del dominio



Negocio con objetos del negocio



Negocio con transfers



Recursos

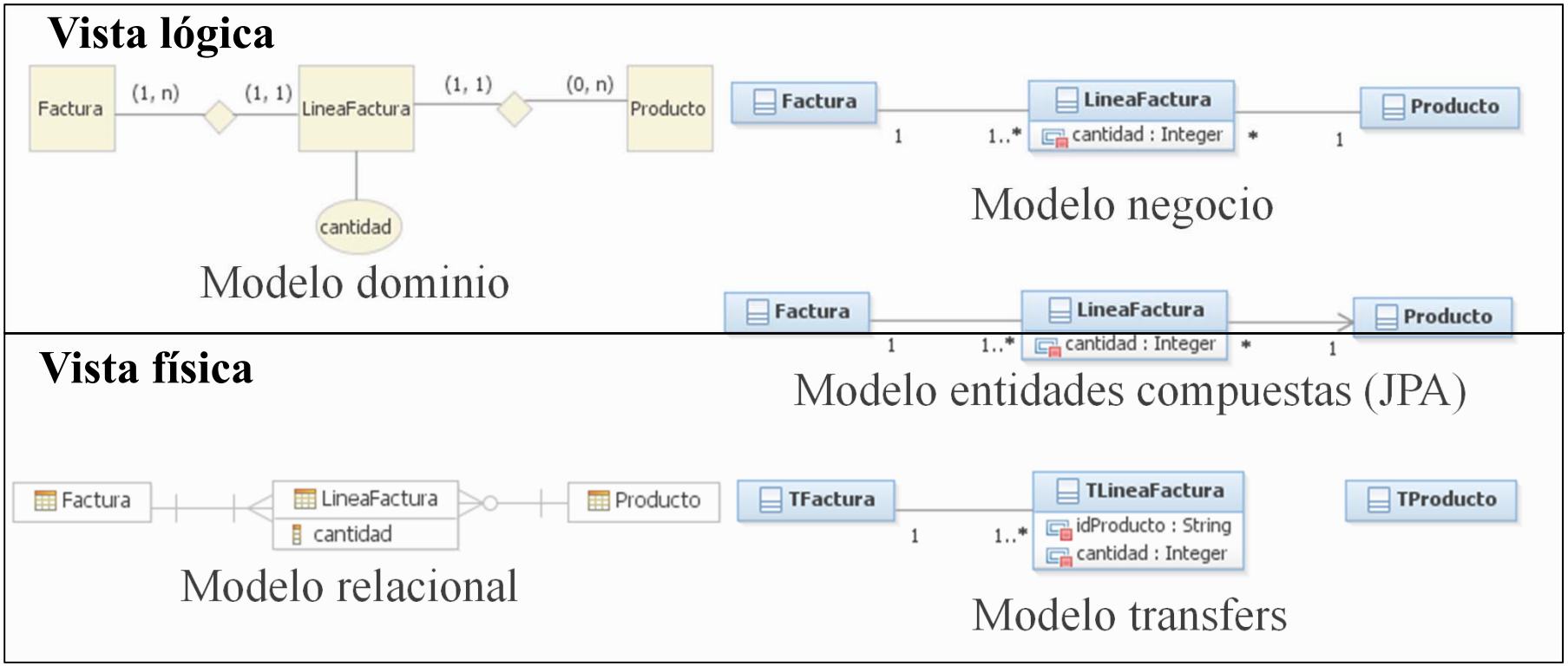
```
<!element cliente (id, nombre, revCli)>
.....
<!element reserva (id, nombre, cliRev)>
.....
```

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

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

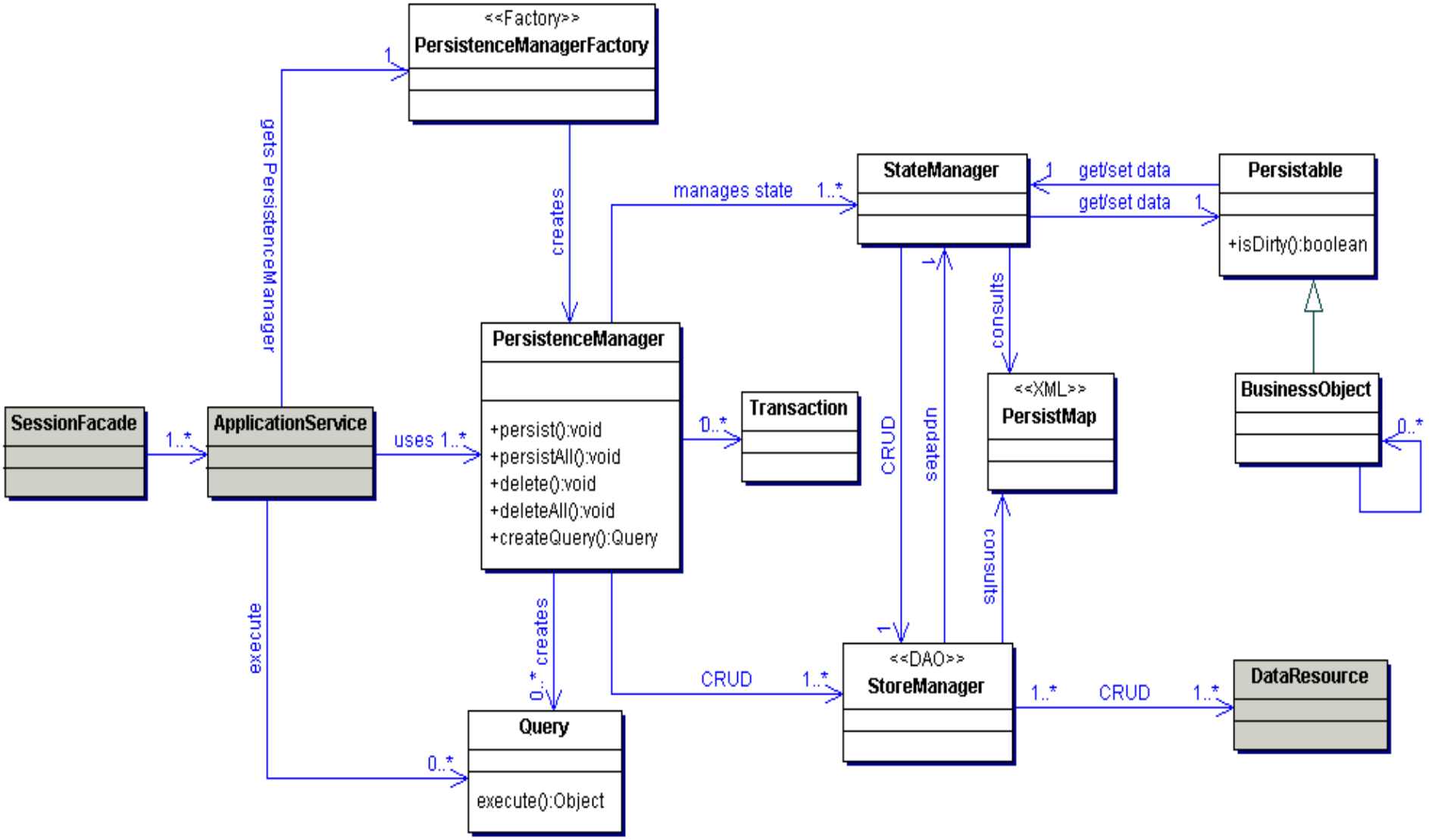
Almacén del dominio

- 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

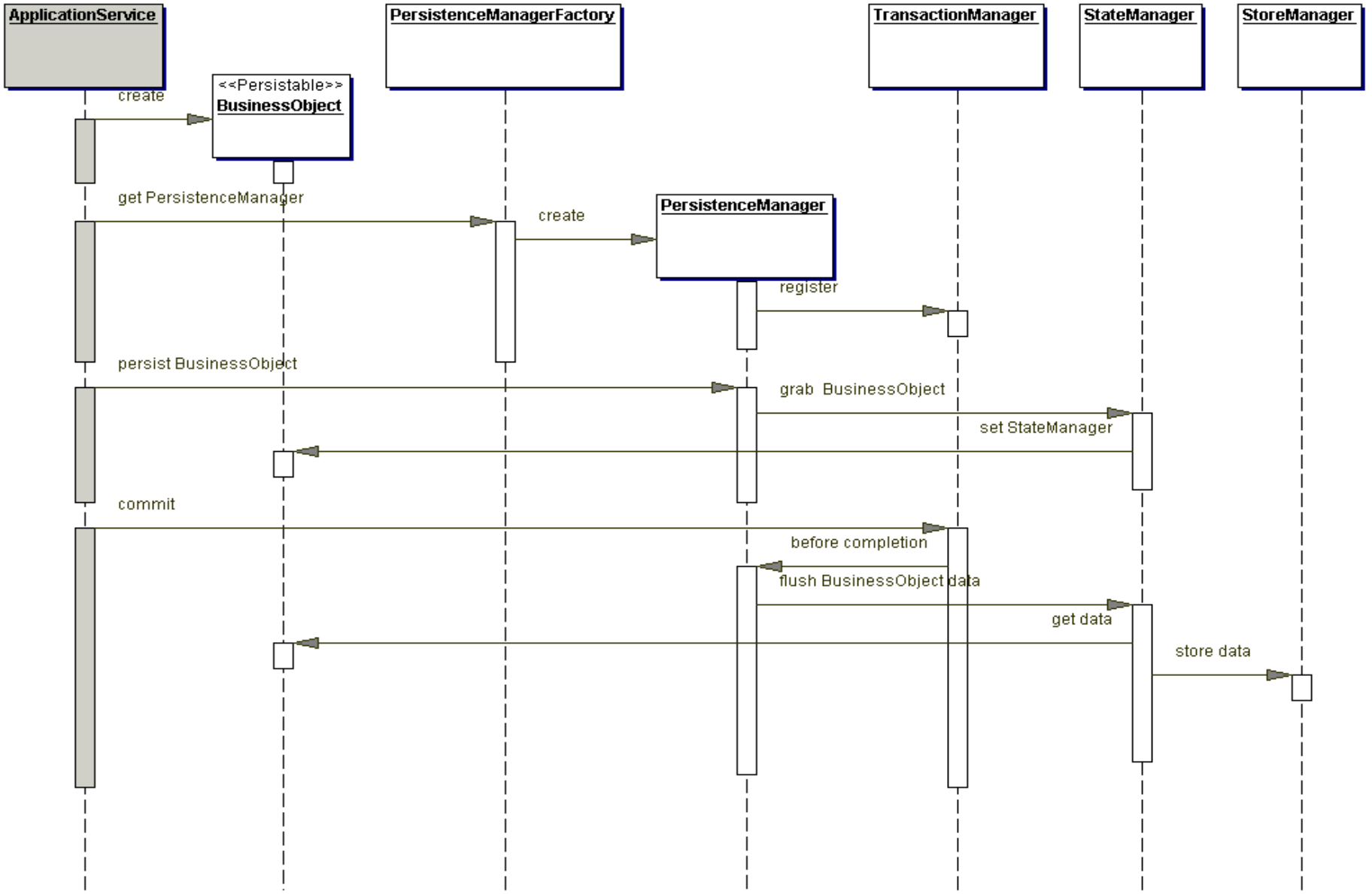
Almacén del dominio

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

- Descripción

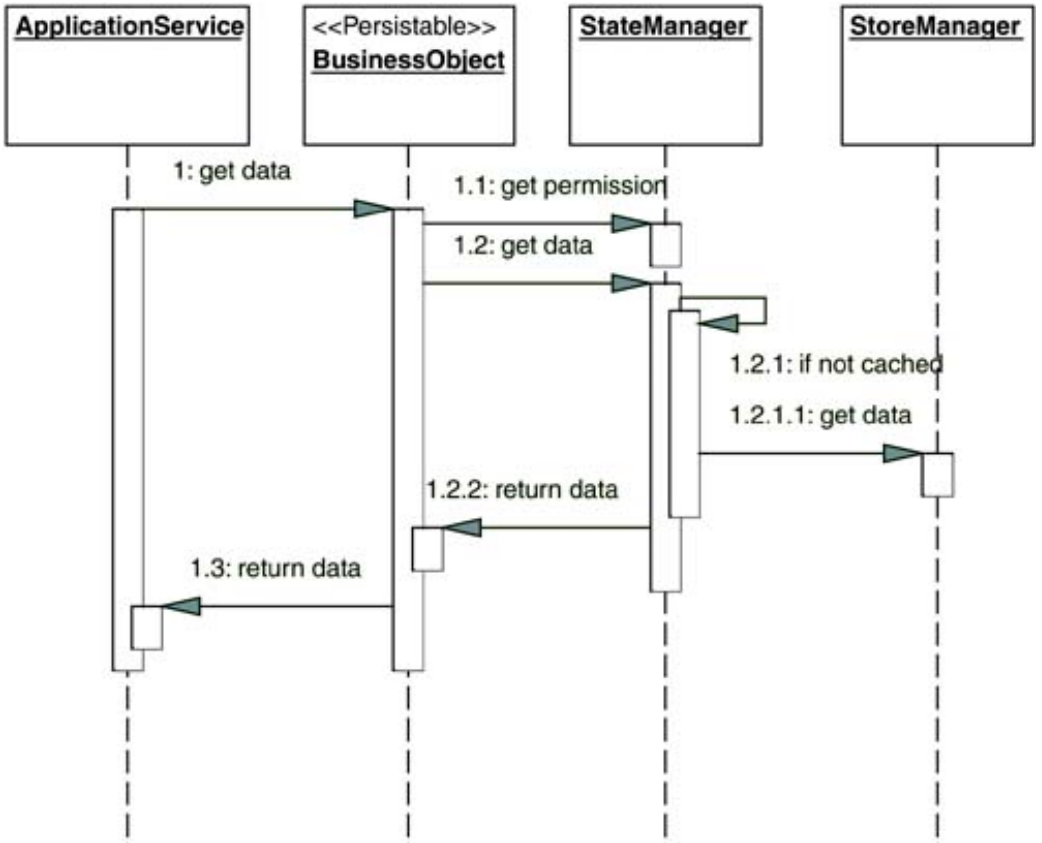


Estructura del patrón almacén del dominio

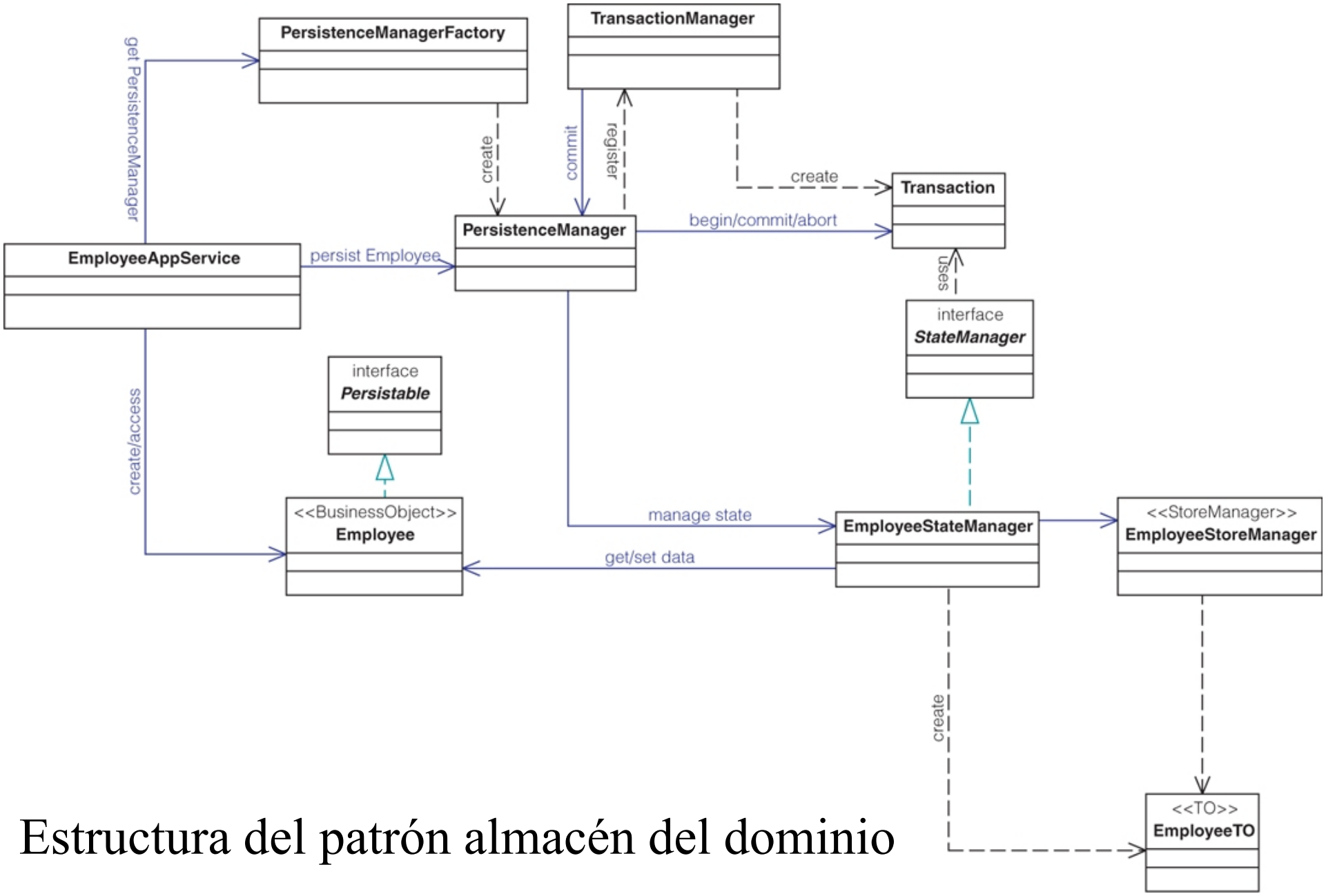


Persistencia de un objeto del negocio

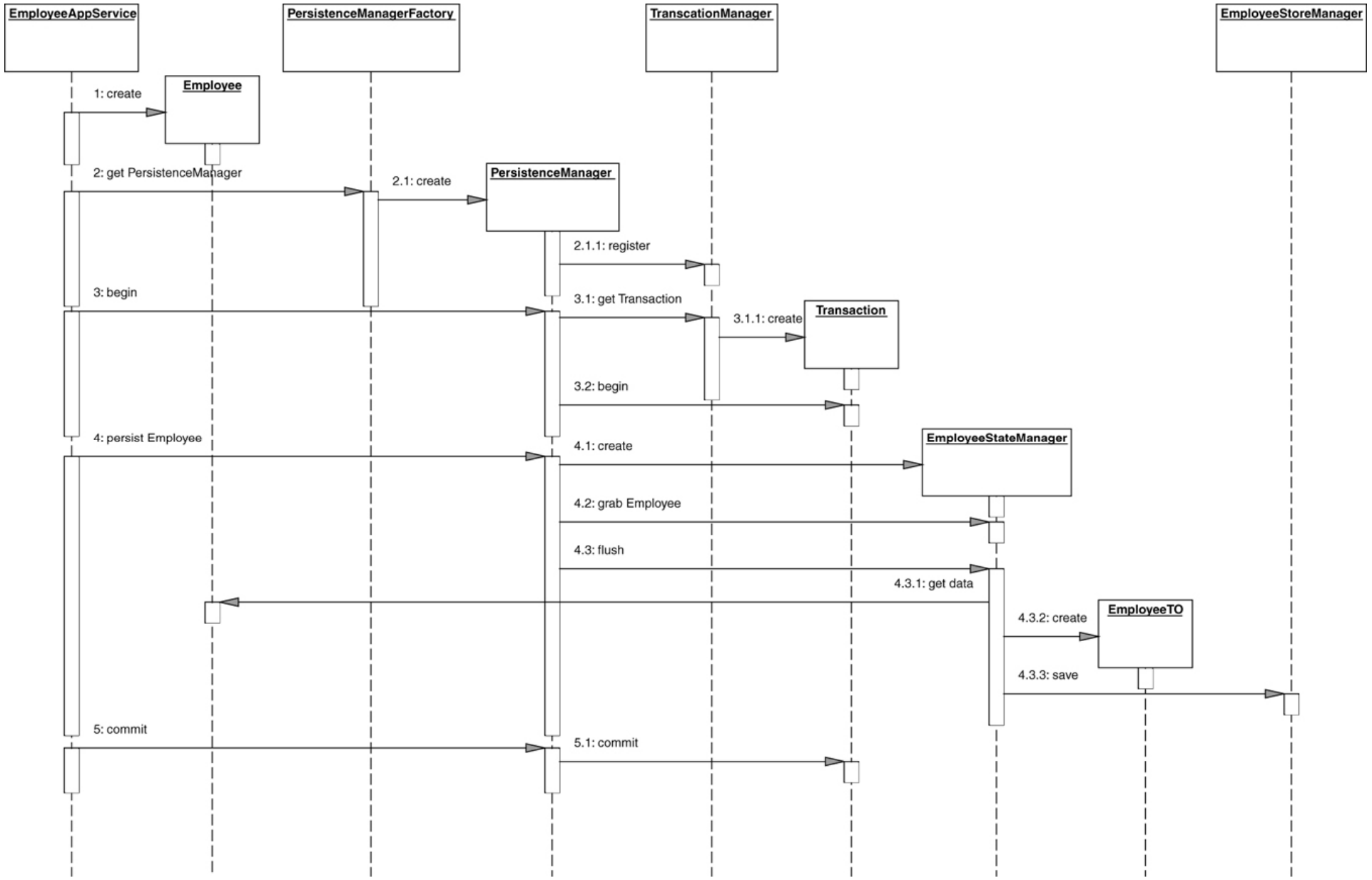
Almacén del dominio



Acceso a atributos simples de un objeto del negocio



Estructura del patrón almacén del dominio



Interacción en el patrón almacén del dominio

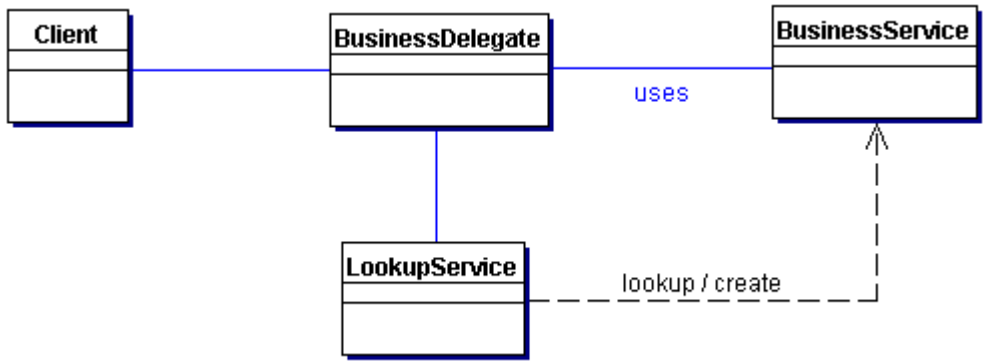
Patrón delegado del negocio

- 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

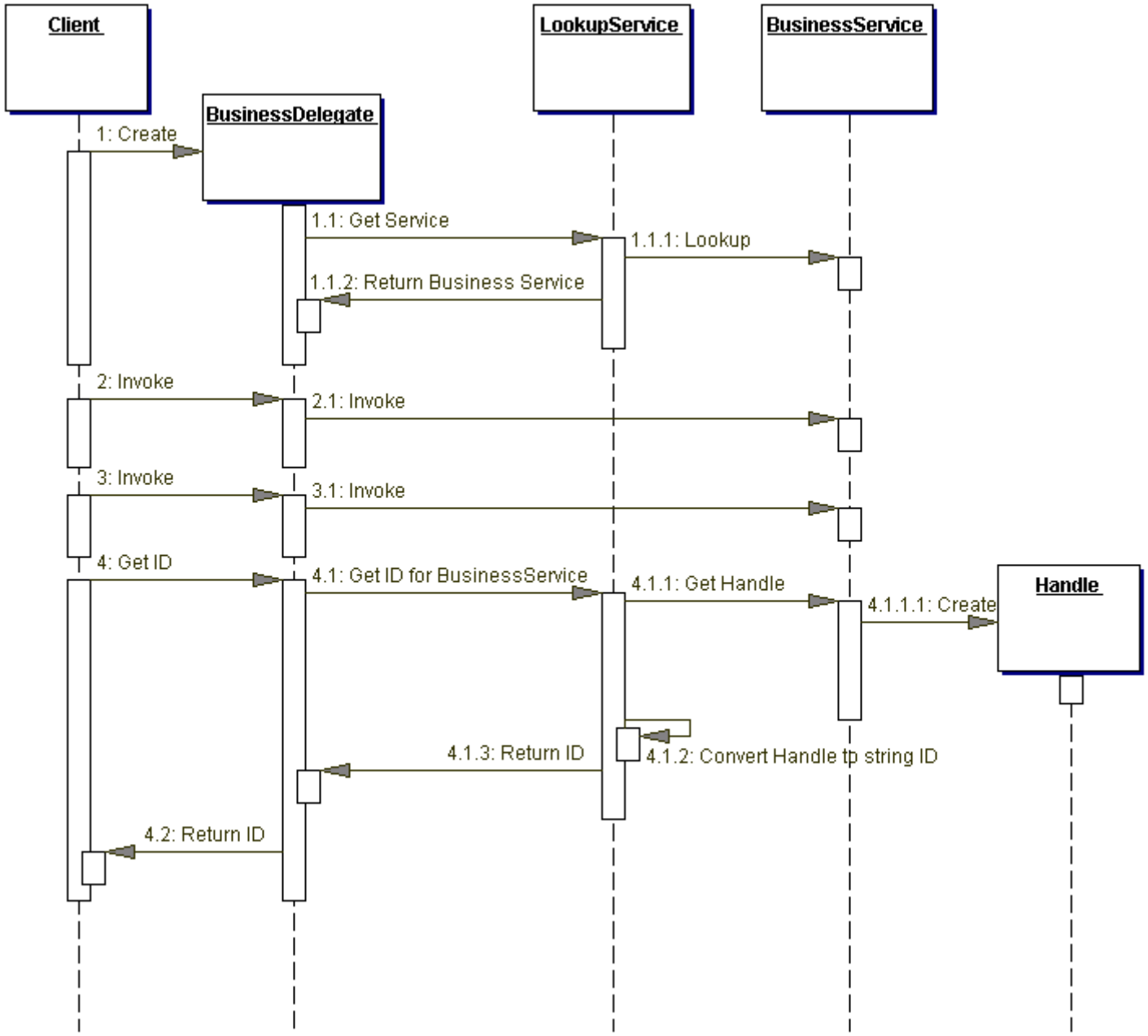
- Motivación
  - Cuando se implementa la capa de negocio con componentes distribuidos los clientes de dicha capa (p.e. la capa de presentación) tienen que tratar con detalles de conexión y acceso al servidor de aplicaciones
  - El patrón delegado se encarga de estos detalles, permitiendo que los clientes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

- 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