Iterator (Iterador)

- **Propósito**
  - Proporciona una forma para acceder a los elementos de una estructura de datos sin exponer los detalles de la representación.
- **Motivación**
  - Un objeto contenedor tal como una lista debe permitir una forma de recorrer sus elementos sin exponer su estructura interna.
  - Debería permitir diferentes métodos de recorrido.
  - Debería permitir recorridos concurrentes.
  - No queremos añadir esa funcionalidad a la interfaz de la colección.

Iterator (Iterador)

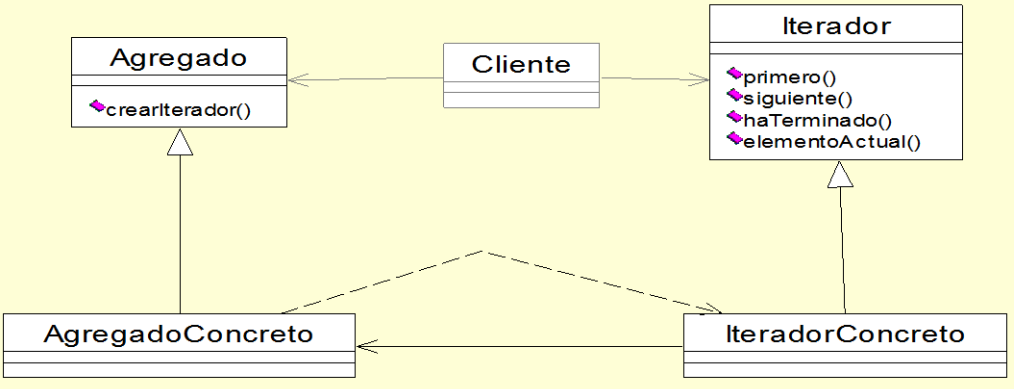
- **Motivación**
  - Una clase *Iterator* define la interfaz para acceder a una estructura de datos (p.e. una lista).
  - Iteradores ***externos*** vs. Iteradores ***internos***.
  - Iteradores externos: recorrido controlado por el cliente
  - Iteradores internos: recorrido controlado por el iterador

Iterator

- **Consecuencias**
  - Simplifica la interfaz de un contenedor al extraer los métodos de recorrido
  - Permite varios recorridos, concurrentes
  - Soporta variantes en las técnicas de recorrido

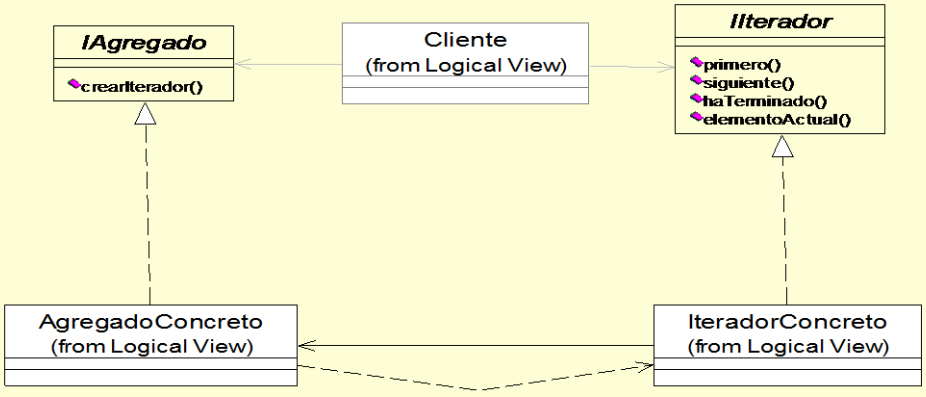
Iterador Externo Polimórfico

Herencia



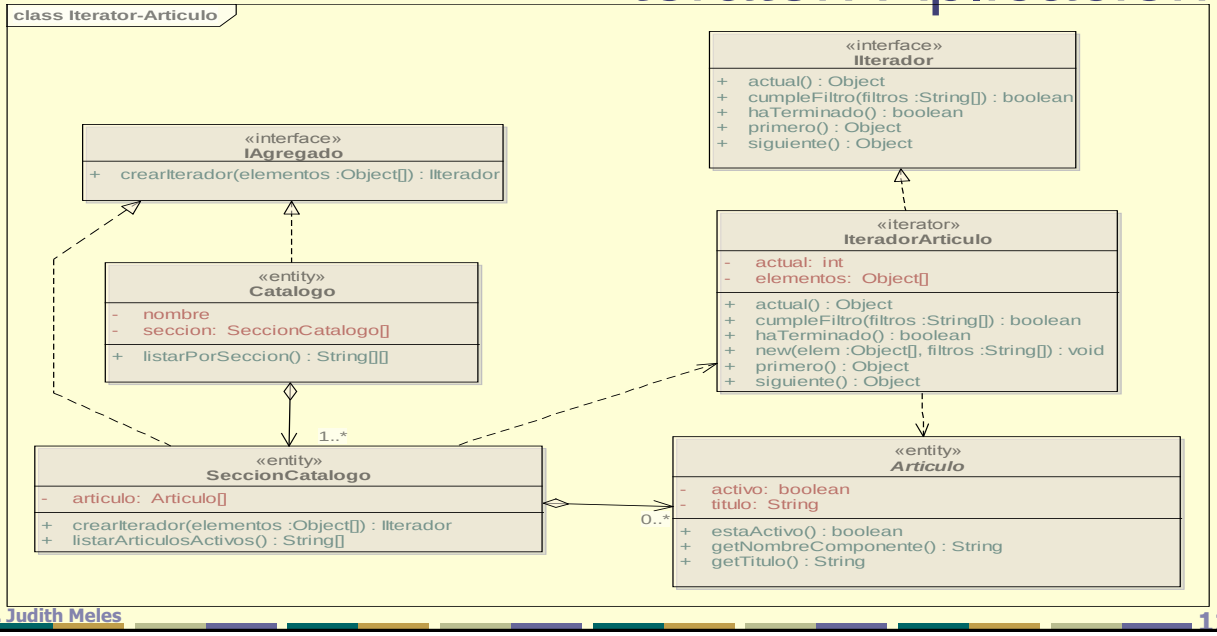
Iterador Externo Polimórfico

Realización



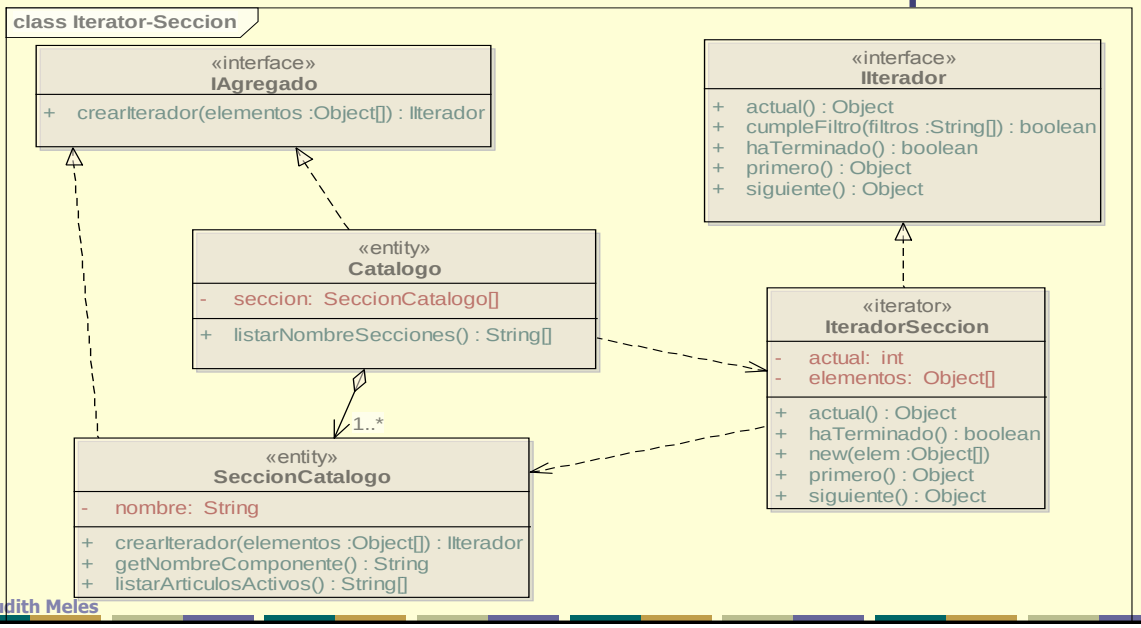
Estructura

Iterator: Aplicación



Estructura

Iterator: Aplicación

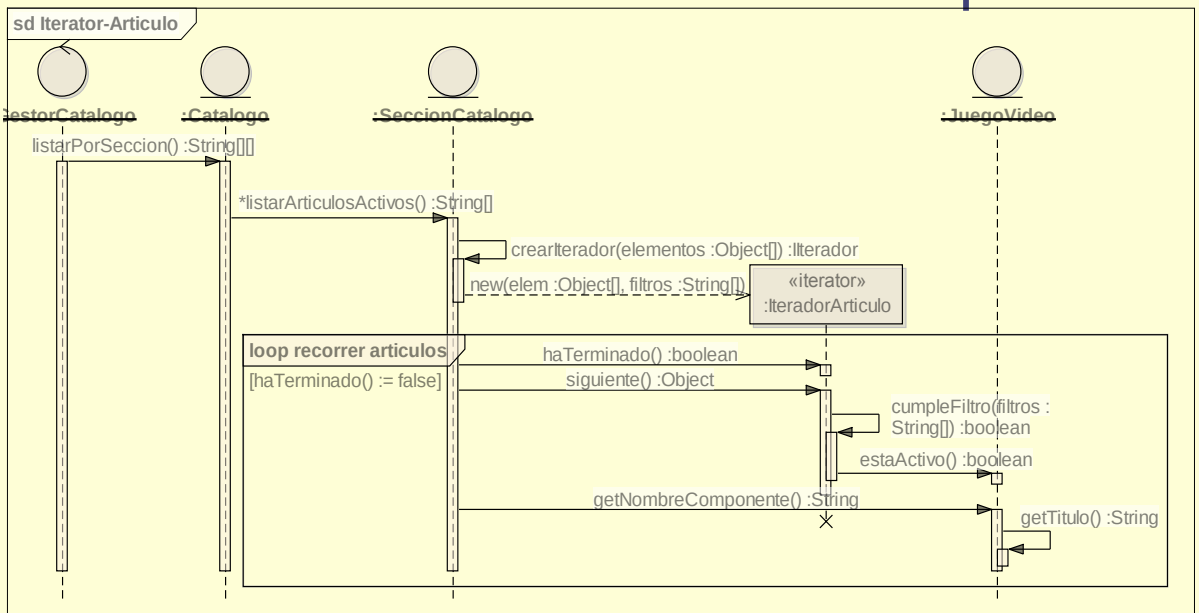


Iterator: Aplicación

- Participantes
  - Iterador [IIterador]: interfaz que permite recorrer los elementos y acceder a ellos.
  - IteradorConcreto [IteradorArticulos, IteradorSecciones]: implementa la interfaz Iterador y mantiene la posición actual en el recorrido del Agregado.
  - Agregado [IAgregado]: define una interfaz para crear un objeto Iterador
  - AgregadoConcreto [Catálogo, SeccionCatalogo]: implementa la interfaz de creación de Iterador para devolver una instancia de un Iterador concreto (IteradorArticulos o IteradorSecciones).
- Procedimiento
  - Cuando necesitamos recorrer o mostrar los artículos (libros, películas, música o juegos) o secciones del catálogo se invoca a los métodos listarArticulosActivos() y listarNombreSecciones(), respectivamente.
  - En la dinámica se mostrará el ejemplo con JuegoVideo.
  - Este método invoca a crearIterador() para establecer el Iterador.
  - Finalmente se definen los métodos necesarios para realizar el recorrido de los elementos: haTerminado(), siguiente(), cumpleFiltro(), etc.

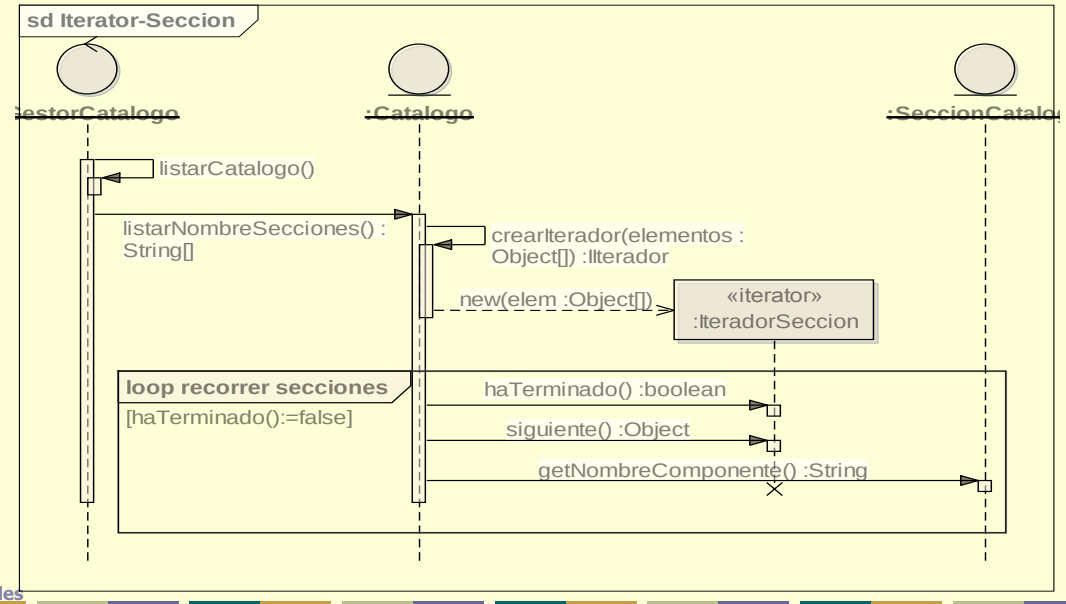
Dinámica:

Iterator: Aplicación



Dinámica:

Iterator: Aplicación



Memento

- **Propósito**
  - Captura y exterioriza el estado interno de un objeto, sin violar el encapsulamiento, de modo que el objeto puede ser restaurado a ese estado más tarde.
- **Motivación**
  - Algunas veces es necesario registrar el estado interno de un objeto: mecanismos *checkpoints* y deshacer cambios que permiten probar operaciones o recuperación de errores.
  - Mecanismo de transacciones

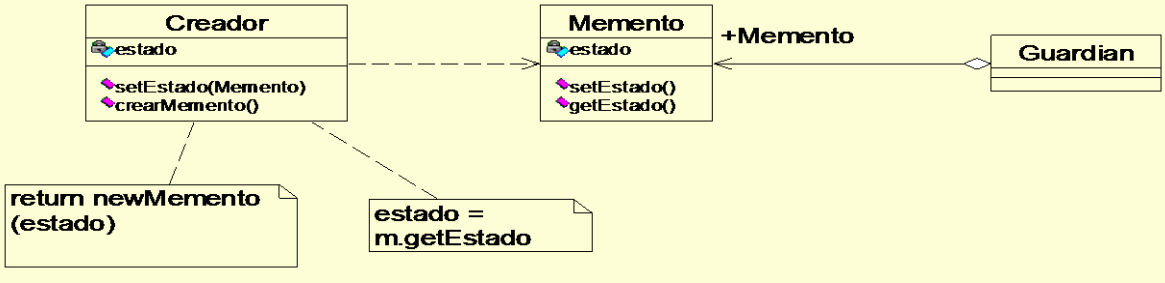
Memento

- **Aplicabilidad**
  - Una parte del estado de un objeto debe ser guardado para que pueda ser restaurado más tarde.
  - Una interfaz para obtener el estado de un objeto podría romper el encapsulamiento exponiendo detalles de implementación.

Memento

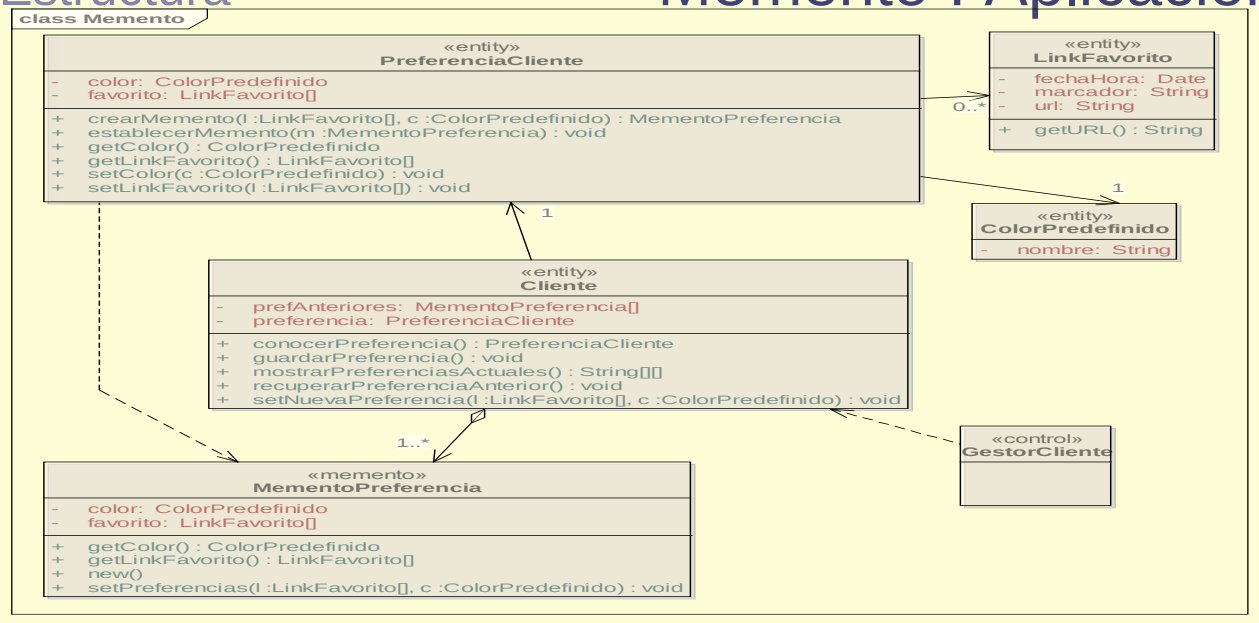
- **Consecuencias**
  - Mantiene el encapsulamiento
  - Simplifica la clase *Creador* ya que no debe preocuparse de mantener las versiones del estado interno.
  - Podría incurrir en un considerable gasto de memoria: encapsular y restaurar el estado no debe ser costoso.
  - Puede ser difícil en algunos lenguajes asegurar que sólo el *Creador* tenga acceso al estado del *Memento*.

Memento



Estructura

Memento : Aplicación



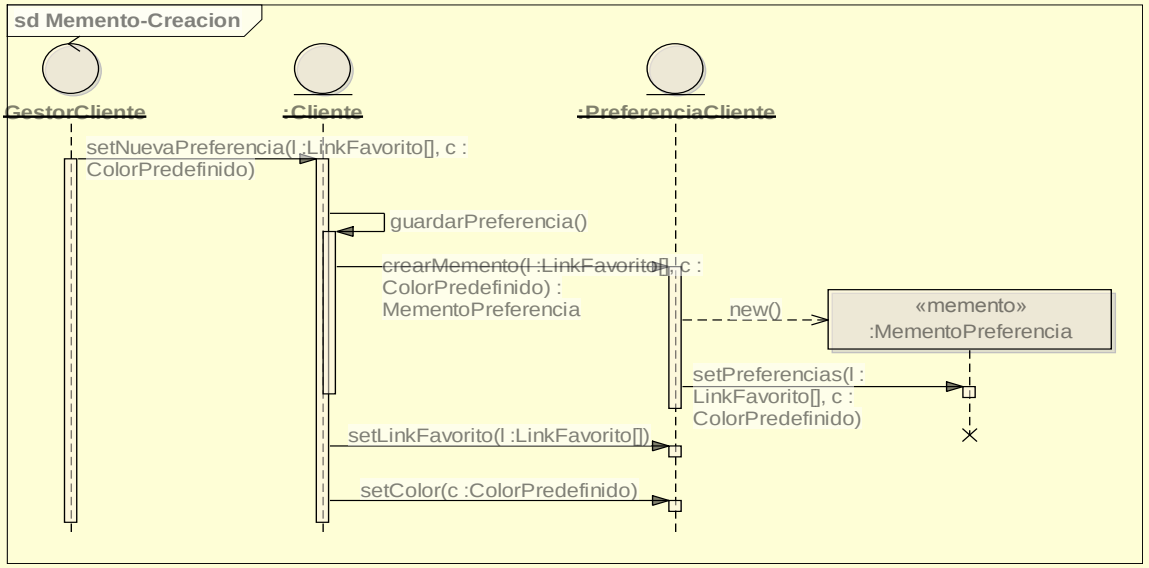
Memento: Aplicación

- Participantes
  - Creador [**PreferenciasCliente**]: es quien va a crear y/o actualizar el Memento de preferencias del cliente.
  - Memento [**MementoPreferencia**]: cada una de las instancias representa las preferencias en cuanto a visualización, links, etc. que tuvo el cliente en un momento determinado.
  - Guardián [**Cliente**]:Es quien contiene a todas las instancias de MementoDePreferencia.
  - Cliente [**GestorCliente**]: Es quién solicita las preferencias actuales o anteriores del cliente y también la creación de las mismas.

- Procedimiento
  - Definir cuáles de los atributos que conforman el estado de un objeto serán necesarios restaurar en un momento posterior. Esos atributos constituirán los atributos del **MementoDePreferencias.**
  - El creador [**PreferenciasCliente**] es el responsable de la creación de las instancias de memento [**MementoPreferencia**], y contendrá el estado actual de la preferencia de cada cliente.
  - El guardián [**Cliente**] es el responsable de contener cada una de las instancias del memento (MementoPreferencia).

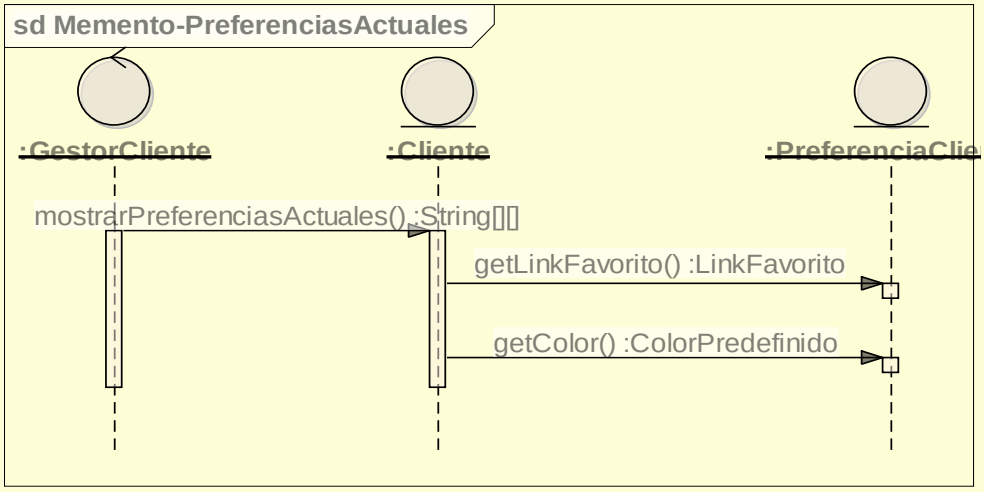
Dinámica:
Creación

Memento : Aplicación



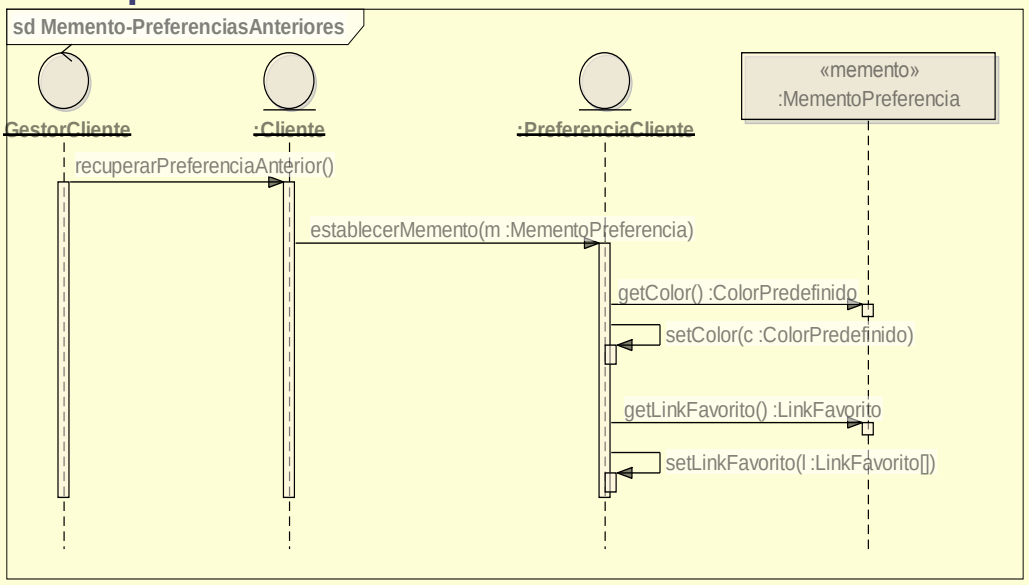
Memento : Aplicación

Dinámica:
Preferencias
Actuales



Memento : Aplicación

Dinámica:
Preferencias Anteriores

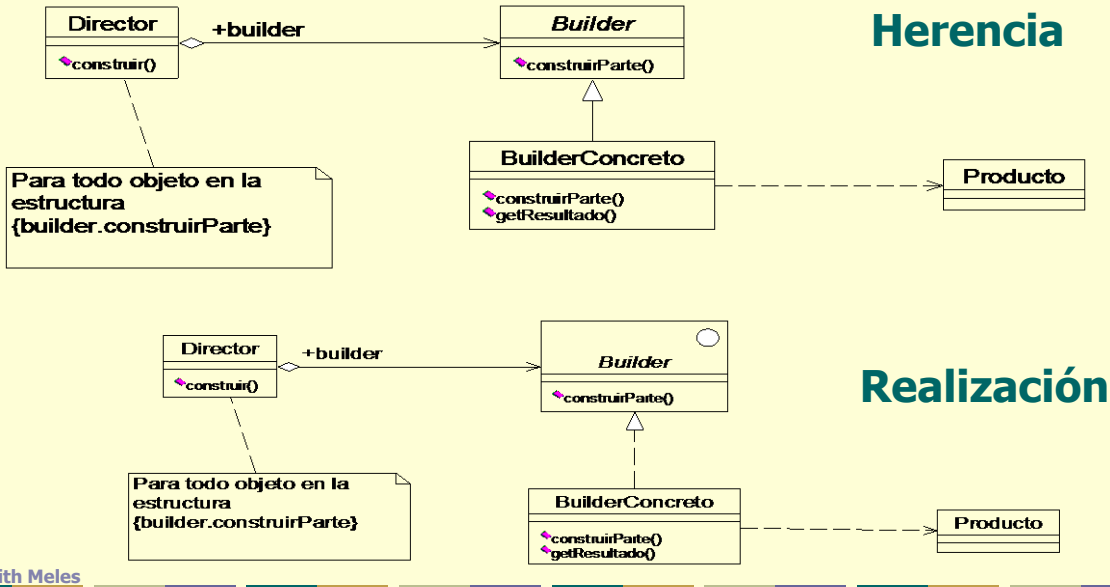


Builder (Constructor)

- **Propósito**
  - Separa la construcción de un objeto complejo de su representación, así que el mismo proceso de construcción puede crear diferentes representaciones.
- **Motivación**
  - Un traductor de documentos RTF a otros formatos. ¿Es posible añadir una nueva conversión sin modificar el traductor?

- **Patrones de Creación**
- Patrones de Estructura
- Patrones de Comportamiento

Builder



Builder

- **Aplicabilidad**
  - Cuando el algoritmo para crear un objeto complejo debe ser independiente de las piezas que conforman el objeto y de cómo se ensamblan.
  - El proceso de construcción debe permitir diferentes representaciones para el objeto que se construye.

- **Patrones de Creación**
- Patrones de Estructura
- Patrones de Comportamiento

Builder

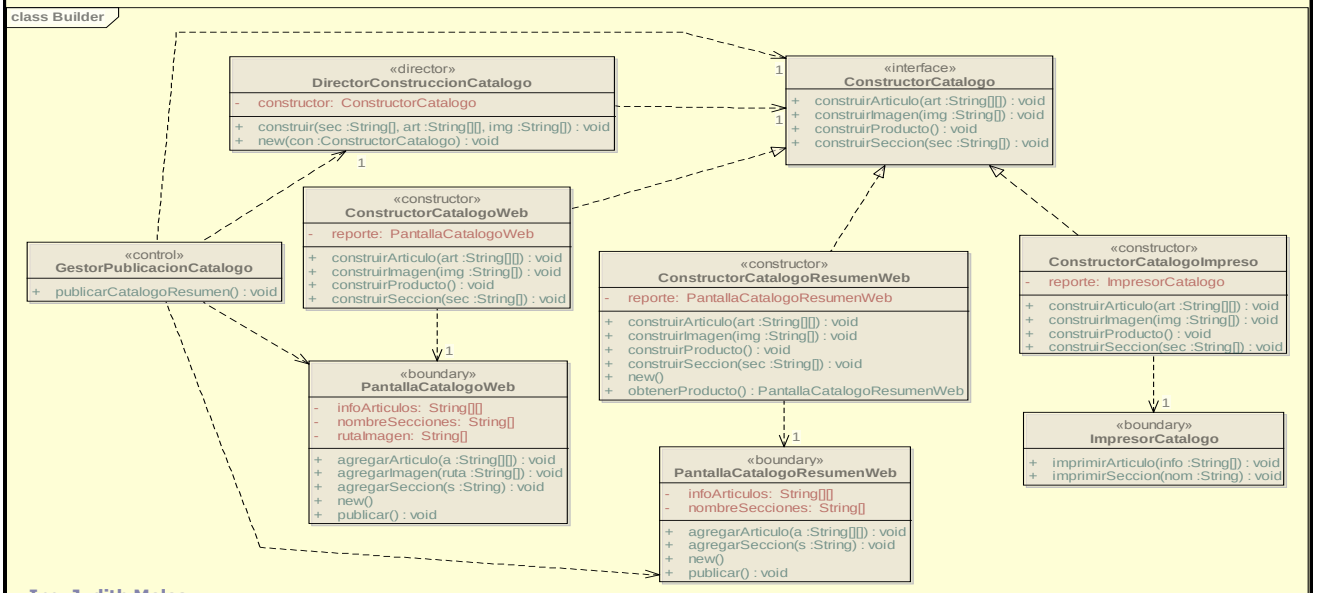
- **Consecuencias**
  - Permite cambiar la representación interna del producto.
  - Separa el código de la representación del código para la construcción.
  - Proporciona un control fino del proceso de construcción.

Builder

- **Implementación**
  - La interfaz de *Builder* debe ser lo suficientemente general para permitir la construcción de productos para cualquier *Builder* concreto.
  - La construcción puede ser más complicada de añadir el nuevo token al producto en construcción.
  - Los métodos de *Builder* no son abstractos sino vacíos.
  - Las clases de los productos no tienen una clase abstracta común.

Builder: Aplicación

Estructura



Builder: Aplicación

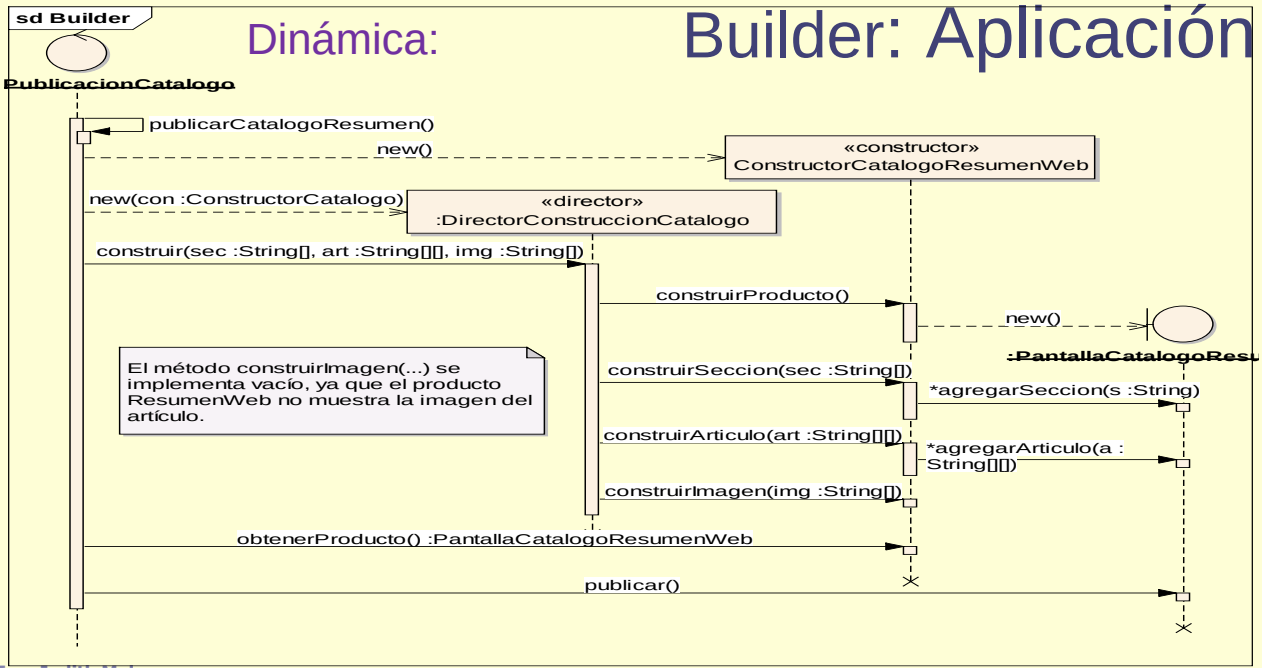
- Participantes
  - Constructor [**ConstructorCatalogo**]: especifica una interfaz abstracta para crear las partes de un objeto complejo.
  - Constructor Concreto [**ConstructorCatalogoWeb, ConstructorCatalogoResumenWeb, ConstructorCatalogoImpreso**]: implementa la interfaz Constructor para construir y ensamblar las partes del objeto producto; define la representación a crear.
  - Director [**DirectorConstruccionCatalogo**]: es el encargado de crear un objeto usando la interfaz Constructor.
  - Producto [**PantallaCatalogoWeb, PantallaCatalogoResumenWeb, ImpresorCatalogo**]: representa el objeto complejo que es construido.

Builder: Aplicación

- Procedimiento
  - Definir las diferentes partes que deberán construirse para obtener el objeto completo.
  - Especificar la interfaz Constructor que define los métodos para construir cada una de las partes detallada anteriormente.
  - Establecer la clase responsable de dirigir, coordinar y controlar la construcción, clase Director.
  - Crear la o las clases que serán los constructores concretos de los productos a ensamblar, las cuales implementan la interfaz Constructor.

Builder: Aplicación

Dinámica:

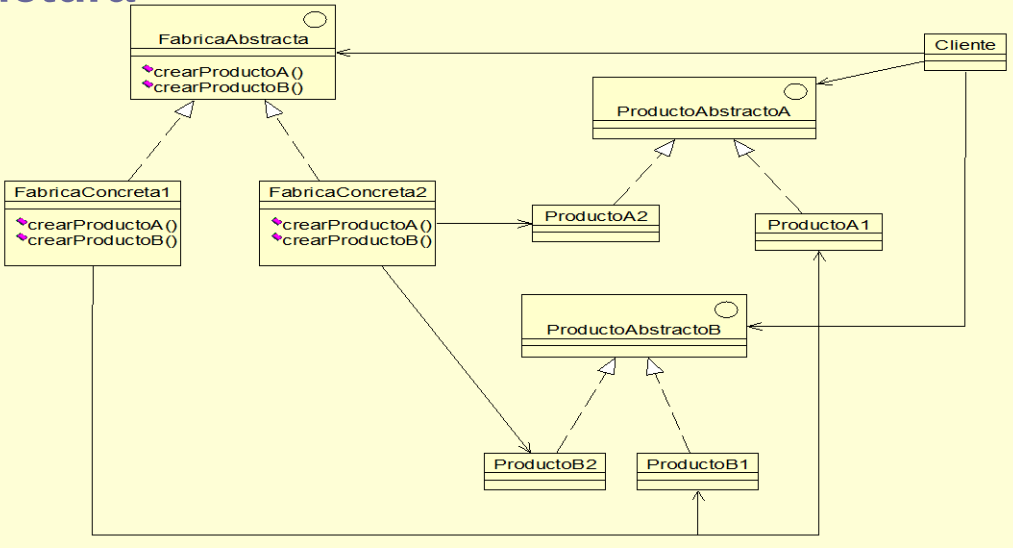


Abstract Factory (Factoría Abstracta)

- **Propósito**
  - Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar la clase concreta
- **Motivación**
  - Un *toolkit* interfaz de usuario que soporta diferentes formatos: Windows, Motif, X-Windows,..

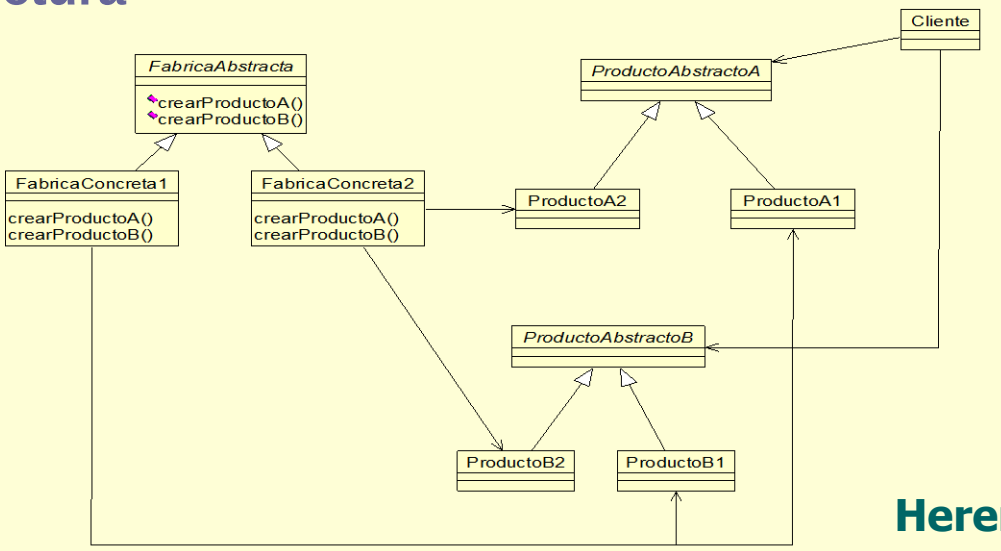
Abstract Factory

Estructura **Realización**



Abstract Factory

Estructura



Herencia

Abstract Factory

- **Aplicabilidad**
  - Un sistema debería ser independiente de cómo sus productos son creados, compuestos y representados
  - Un sistema debería ser configurado para una familia de productos.
  - Una familia de objetos productos relacionados es diseñado para ser utilizado juntos y se necesita forzar la restricción.
  - Se quiere proporcionar una librería de clases de productos y se quiere revelar sólo la interfaz, no la implementación.

- **Patrones de Creación**
- Patrones de Estructura
- Patrones de Comportamiento

Abstract Factory

- **Consecuencias**
  - Aísla a los clientes de las clases concretas de implementación.
  - Facilita el intercambio de familias de productos.
  - Favorece la consistencia entre productos
  - Es difícil soportar nuevas clases de productos.

Abstract Factory

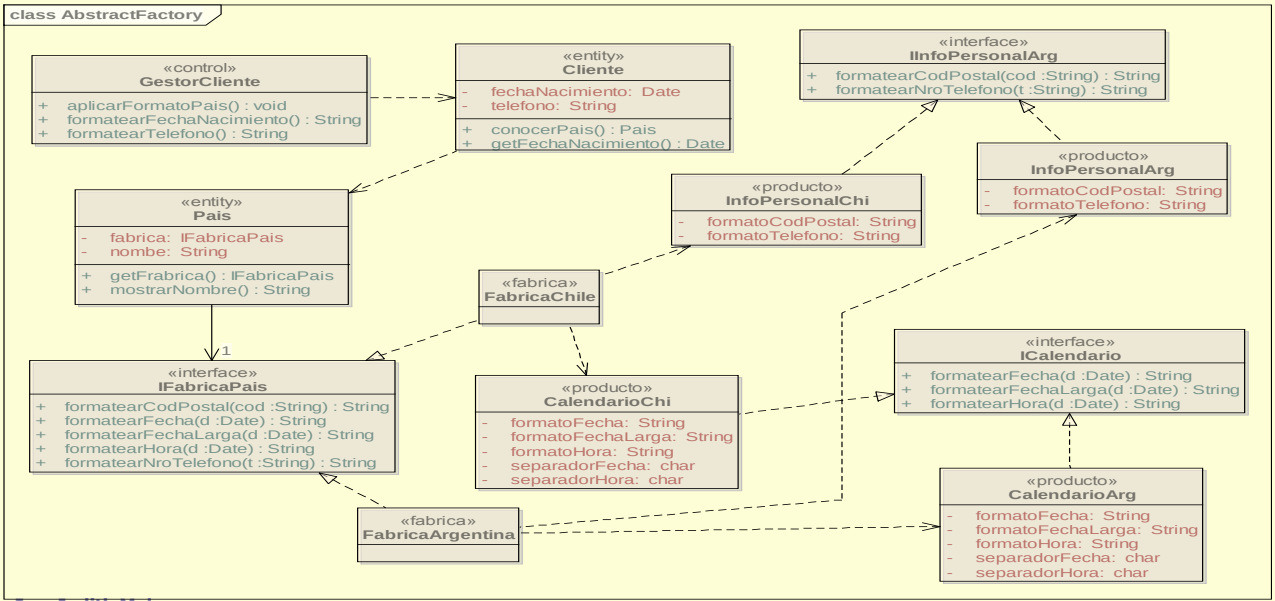
- **Implementación**
  - Factorías como *singleton.*
  - Se necesita una subclase de *AbstractFactory* por cada clase de producto que redefine un *método factoría*.
  - Posibilidad de usar el patrón *Prototype*.
  - Definir factorías extensibles: *AbstractFactory* sólo necesita un método de creación. DEPENDE DEL LENGUAJE

Abstract Factory: Aplicación

- **Procedimiento**
  - Analizar la situación determinando elementos que siendo del mismo tipo varían en su comportamiento (Productos).
  - Identificar el patrón que da origen a las variaciones en los productos (fábricas).
  - Implementar las características que varían de los productos es decir: implementar los productos concretos.
  - Implementar las fábricas concretas que crean los productos concretos.
  - Establecer en el cliente la relación de conocimiento con la fábrica abstracta y el producto abstracto.
  - Entendemos que el cliente recibirá por parámetros la identificación de la fábrica concreta. Y a partir de esta última se determinarán los productos concretos que corresponda.

Estructura

Abstract Factory: Aplicación

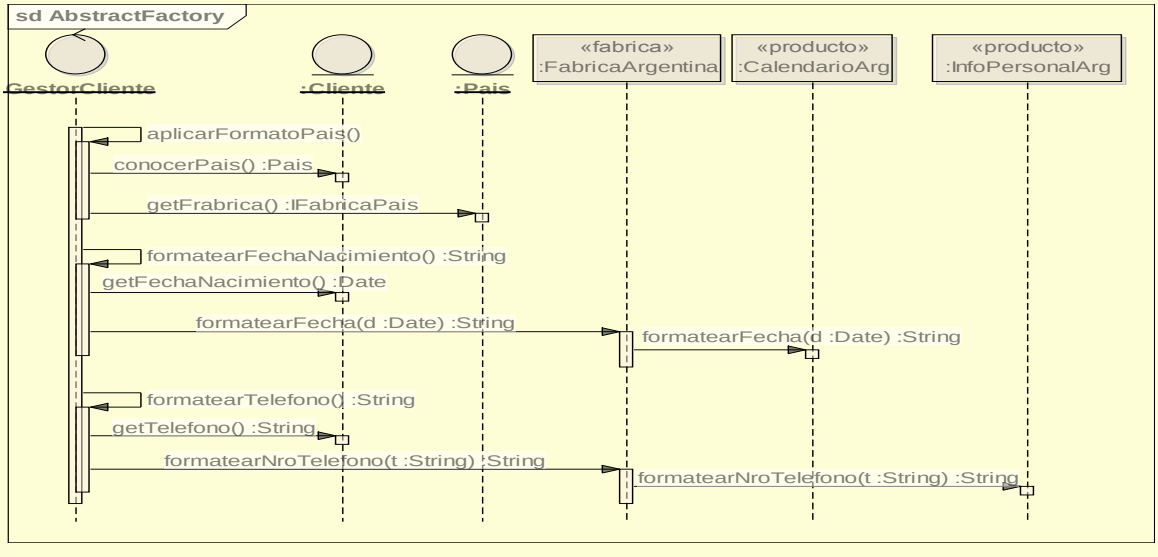


Abstract Factory : Aplicación

- **Participantes**
  - **Fábrica Abstracta** *[**IFabricaPais**]:*
    - Declara una interfaz para las operaciones que crean los objetos que determinar las preferencias de los usuarios para moneda, fecha y hora.
  - **Fábricas Concreta** *[**FabricaArgentina y FabricaChile**]:*
    - Implementa las operaciones que crean los objetos que definen las preferencias concretas de cada país.
  - **Producto Abstracto** *[**IMoneda, IInfoPersonal**]:*
    - Declara la interfaz para un crear cada producto determinado.
  - **Producto Concreto** *[**MonedaArg, MonedaChi, InfoPersonalArg, InfoPersonalChi**]*
    - Define un objeto para que sea creado por la fábrica correspondiente al país.
  - **Cliente** *[**GestorCliente**]*
    - Sólo usa interfaces declaradas por las clases fábrica abstracta y producto abstracto.

Abstract Factory: Aplicación

Dinámica

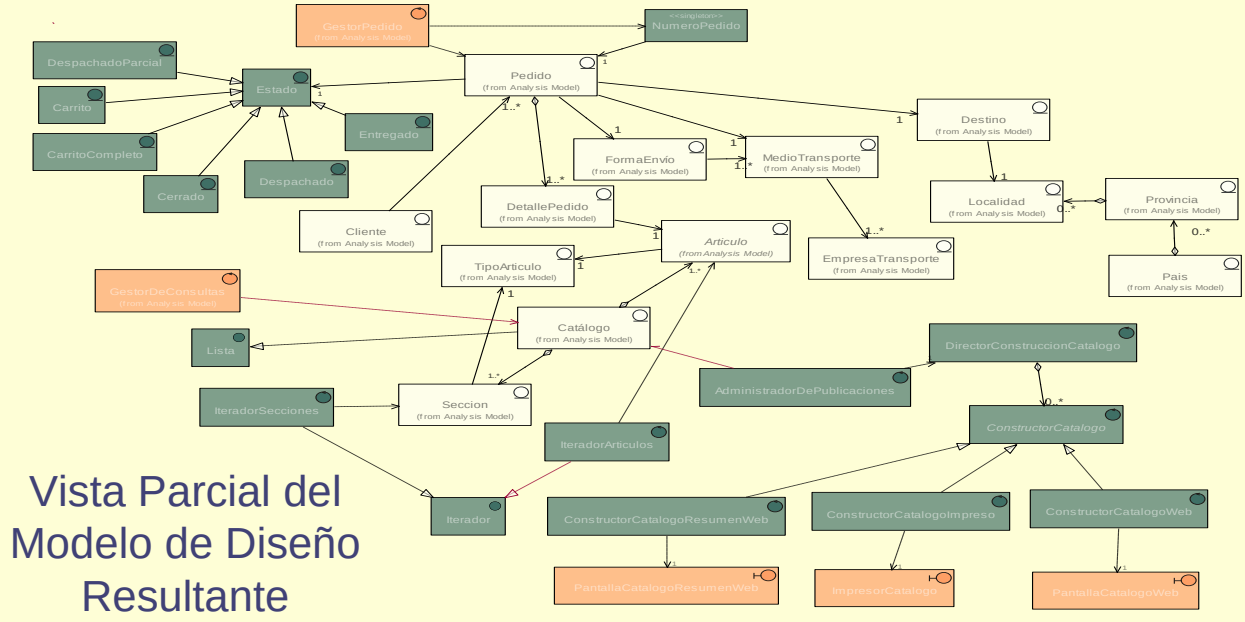


Abstract Factory: Aplicación

- **Procedimiento**

Obtener Formato Fecha

- El **GestorCliente** necesita configurar el formato de fecha en función del país, para ello le pide al **Cliente** la referencia del país al que pertenece.
- Luego le pide a la **FabricaArgentina** que cree la **InfoPersonalArg** y que obtenga la máscara de teléfono.
- Luego el **GestorCliente** le envía a la **PantallaCliente** la información para configurar campo de número de teléfono, según el país.

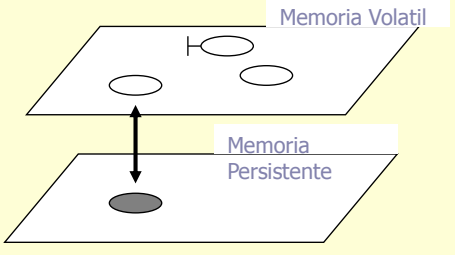


Vista Parcial del Modelo de Diseño Resultante

Mapeo de Clases a Bases de Datos Relacionales

Problema de Impedancia

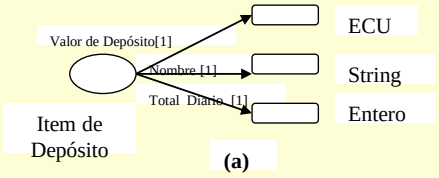
- Los tipos a ser usados en las tablas son mayormente los tipos primitivos
- La información se almacena en los objetos. Por lo tanto necesitamos transformar nuestra estructura de información de objeto a una estructura orientada a tablas
- Crea un fuerte acoplamiento entre la aplicación y el DBMS



Objetos en Tablas

- (1) Asignar una tabla para la clase.
- (2) Cada atributo (primitivo) se transformará en una columna en la tabla. Si el atributo es complejo (es decir, debe componerse de tipos de DBMS), o agregamos una tabla adicional para el atributo, o distribuimos el atributo en varias columnas de la tabla de la clase.
- (3) La columna de la clave primaria será el identificador único de la instancia, llamado identificador.
- (4) Cada instancia de la clase ahora será representada por una fila en esta tabla.
- (5) Cada asociación de conocimiento con una cardinalidad mayor que 1 (por ejemplo, [0..N]) se transformará en una nueva tabla. Esta nueva tabla conectará las tablas que representan los objetos que van a ser asociados.

Objetos en Tablas



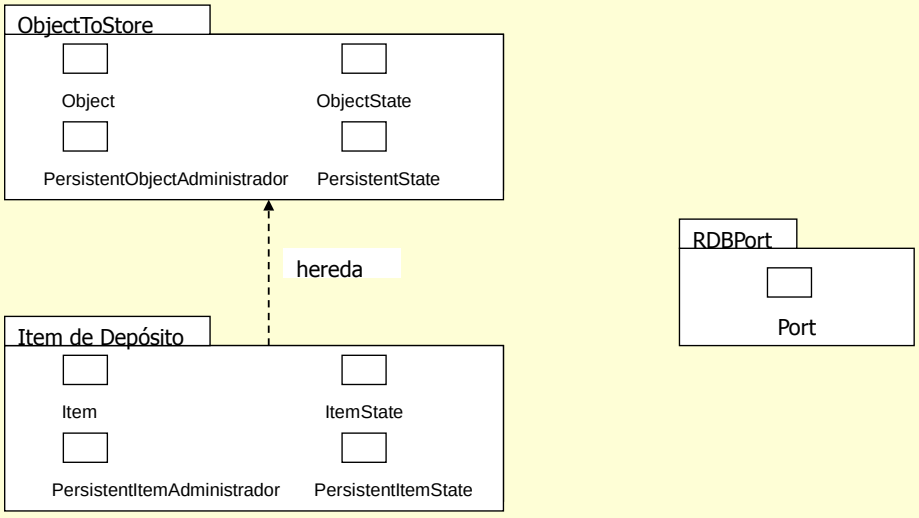
(a)

Item de Depósito

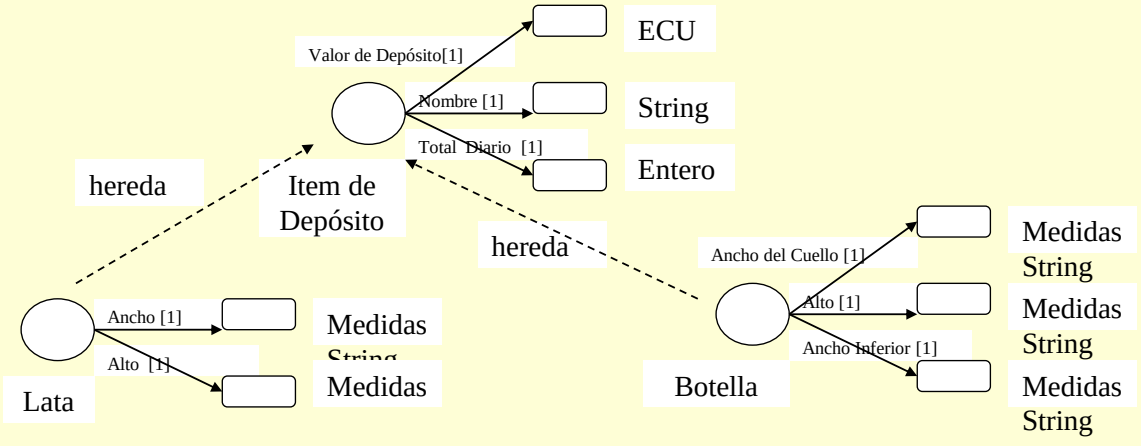
Nombre	Valor de Depósito	Total Diario
Lata33	0.25	142
Lata50	0.35	35
Botella25	0.30	173
Botella75	0.50	78

(b)

Solución: Super Objeto Persistente



Objetos en Tablas: Herencia



Eliminar la Herencia

Nombre Lata	Valor de Depósito	Total Diario	Alto	Ancho
Lata33	0.25	127	17	7.50
Lata50	0.35	283	25	7.50

Nombre Botella	Valor de Depósito	Total Diario	Alto	Ancho Inferior	Ancho del Cuello
Botella25	0.30	173	23	6	2
Botella75	0.50	78	32	9	3

Simular la Herencia

Nombre Lata	Valor de Depósito	Total Diario
Lata33	0.25	142
Lata50	0.35	35
Botella25	0.30	173
Botella75	0.50	78

Lata

Nombre Lata	Alto	Ancho
Lata33	17	7.50
Lata50	25	7.50

Botella

Nombre Botella	Alto	Ancho Inferior	Ancho del Cuello
Botella25	23	6	2
Botella75	32	9	3

Características de las BDOO

- **Objetos complejos**. Debería soportar la noción de objetos complejos.
- **Identidad de Objetos**. Cada objeto debe tener una identidad independiente de sus valores internos.
- **Encapsulamiento**. Debe soportar el encapsulamiento de datos y comportamiento en los objetos.
- **Tipos o Clases**. Debería soportar un mecanismo de estructuración, en forma de tipos o clases.
- **Jerarquía**. Debería soportar la noción de herencia.
- **Ligadura Tardía**. Debería soportar sobreescritura y ligadura tardía.
- **Completitud**. El lenguaje de manipulación debería ser capaz de expresar cada función calculable.
- **Extensibilidad**. Debería ser posible agregar nuevos tipos.

Beneficios de Usar ODBMS :

- Los objetos como tales pueden almacenarse en la base de datos (frecuentemente las operaciones no son almacenadas, solo están presentes en la librería de clases en la memoria primaria).
- No se necesita ninguna conversión del tipo de sistema de DBMS; las clases definidas por el usuario se usan como tipos en el DBMS.
- El lenguaje del DBMS puede integrarse con un lenguaje de programación orientado a objetos. El lenguaje puede ser exactamente el mismo que se usó en la aplicación, lo que no fuerza al programador a tener dos representaciones de sus objetos.

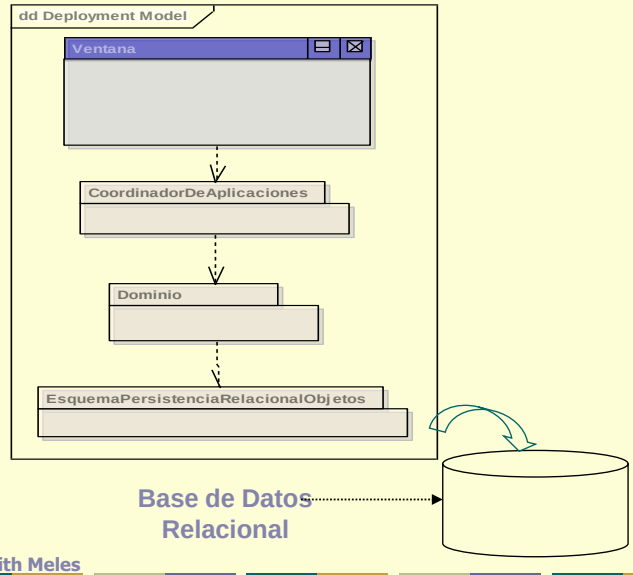
Frameworks

- Es un conjunto extensible de clases e interfaces que colaboran para proporcionar servicios de la parte central e invariable de un subsistema lógico.
- Contiene clases concretas y (especialmente) abstractas que definen las interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras variantes.
- Puede requerir que el usuario del framework defina subclases de las clases del framework para utilizar, adaptar y extender los servicios del framework.
- Tiene clases abstractas que podrían contener tanto métodos abstractos como concretos.
- Confía en el **Principio de Hollywood**: "*No nos llame, nosotros lo llamaremos*".
- Ofrecen un alto grado de reutilización, mucho más que con clases individuales.

Framework de Persistencia

- Debería proporcionar funciones para:
  - Almacenar y recuperar objetos en un mecanismo de almacenamiento persistente.
  - Confirmar y deshacer (commit y rollback) las transacciones

Framework de Persistencia



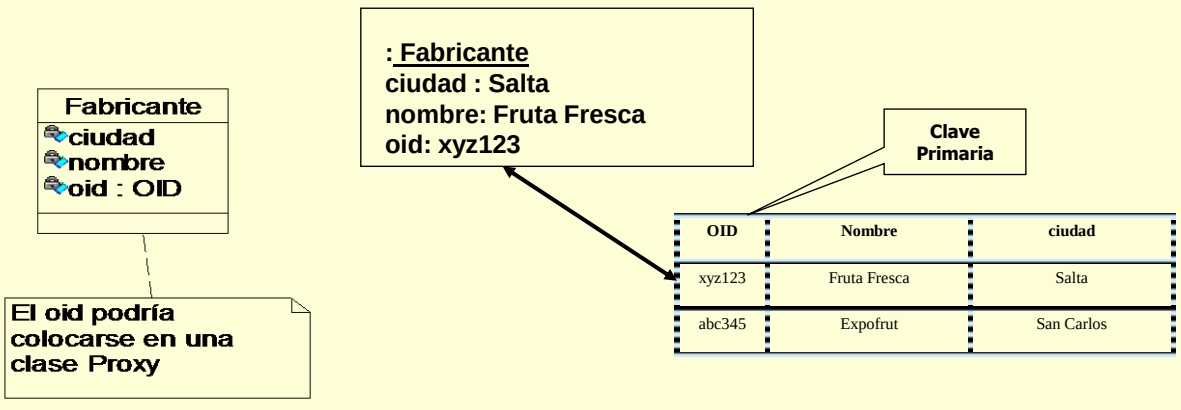
Ideas a desarrollar…

- **Correspondencia (Mapping):** entre clases y tablas; entre atributos y campos. Es decir correspondencia de “esquemas”.
- **Identidad de Objeto**: identificador único que vincula objetos con registros.
- **Materialización:** transformar un registro en objeto
- **Desmaterialización**: transformar un objeto en registro.
- **Conversor de Base de Datos**: es el responsable de la materialización y desmaterialización de los objetos.
- **Caché**: Almacén de objetos materializados en esta memoria por cuestiones de rendimiento.
- **Estado de Transacción de los objetos**: El estado de los objetos para saber si se modificaron en función de su estado almacenado.
- **Operaciones de Transacción**: confirmar y deshacer
- **Materialización Perezosa**: no todos los objetos se materializan a la vez, se hace bajo demanda, es decir, cuando se lo necesita.
- **Proxies Virtuales**: Referencia inteligente utilizada para la materialización perezosa.

Patrón: Identificador de Objetos

- Es necesario contar con una forma consistente de relacionar objetos con registros y asegurar que la materialización repetida no de cómo resultado objetos duplicados.
- El Patrón propone asignar un identificador de objeto (OID) a cada registro y objeto (o proxy de objeto)

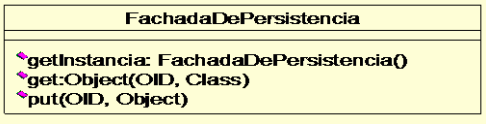
Patrón: Identificador de Objetos

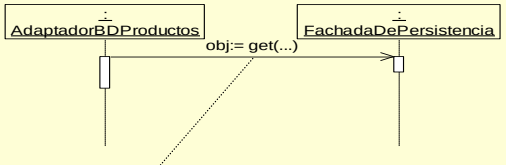


Acceso al Servicio de Persistencia

- Patrón Fachada: proporciona interfaz uniforme a un subsistema.
- Se necesita una operación para recuperar un objeto dado un OID.
- También se necesita conocer el tipo del objeto que se va a materializar, por tanto, se debe proporcionar el tipo de la clase a la que pertenece el objeto.

La Fachada de Persistencia






```
// ejemplo de uso de la fachada

OID oid = new OID ("XYZ123);
EspecificacionDelProducto ep = (EspecificacionDelProducto) FachadaDePersistencia.getInstancia(). get(oid,
EspecificacionDelProducto.class);
```

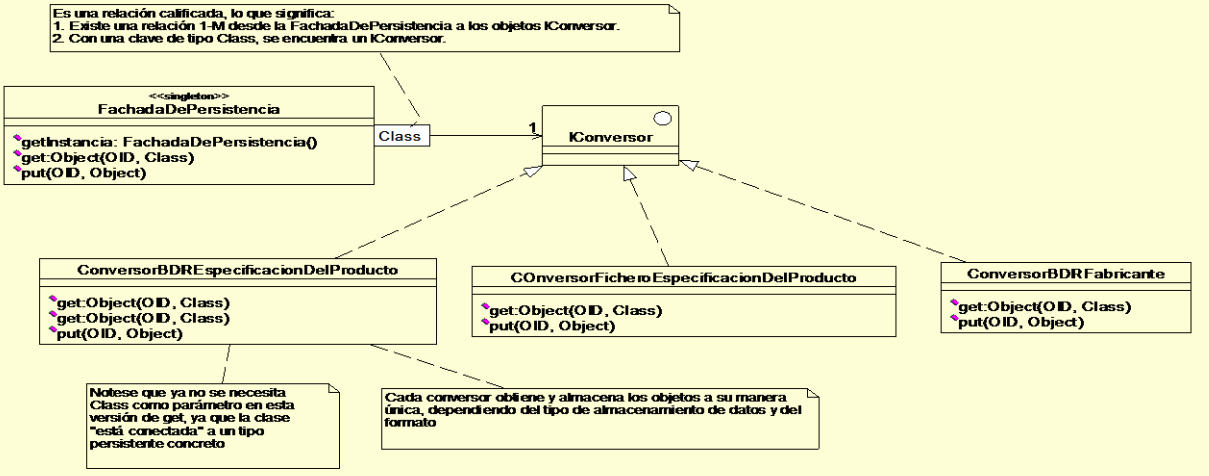
Responsabilidad de Materializar y Desmaterializar Objetos

- La fachada no hace el trabajo, solo lo delega en objetos del subsistema.
- El patrón **Experto**, sugiere que la propia clase de dominio (EspecificacionDelProducto) es candidata porque tiene los datos que se van a almacenar. Esto se denomina **correspondencia directa**.
- Se puede utilizar la correspondencia directa si el código relacionado a la BD se genera y se inyecta automáticamente mediante un compilador de post-procesamiento, y el desarrollador nunca tiene que ver o mantener ese código que añade confusión a la clase provocando:
  - Fuerte acoplamiento de la clase persistente con su almacenamiento. (Violación del Patrón Bajo Acoplamiento).
  - Responsabilidades complejas y no relacionadas (Violación del Patrón Alta Cohesión).

Patrón Conversor (Mapper) de Base de Datos o Intermediario (Broker) de Base de Datos

- Enfoque clásico de correspondencia **Indirecta**.
- Utiliza otros objetos para establecer la correspondencia con los objetos persistentes.
- Propone crear una clase responsable de materializar y desmaterializar los objetos.

Conversor de Base de Datos



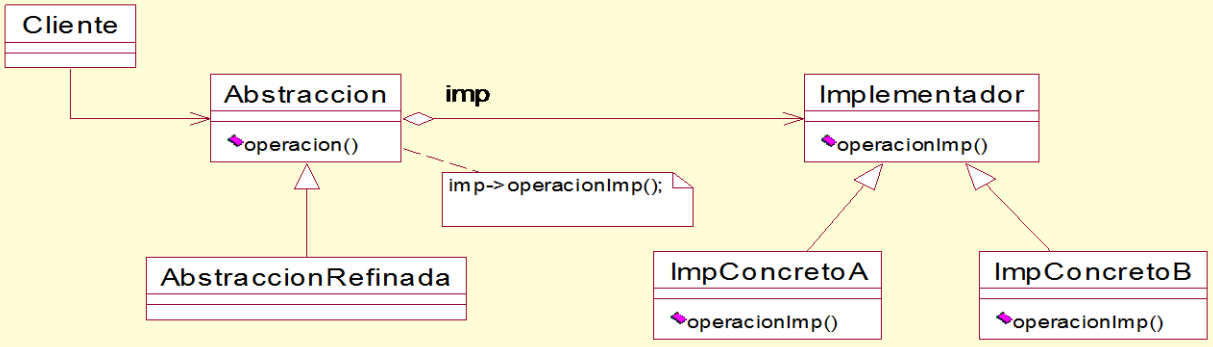
Bridge / Handle

- **Propósito**
  - Desacoplar una abstracción de su implementación, de modo que los dos puedan cambiar independientemente.
- **Motivación**
  - Clase que modela la abstracción con subclases que la implementan de distintos modos.
  - Herencia hace difícil reutilizar abstracciones e implementaciones de forma independiente
    - Si refinamos la abstracción en una nueva subclase, está tendrá tantas subclases como tenía su superclase.
    - El código cliente es dependiente de la plataforma.

Bridge / Handle

Estructura

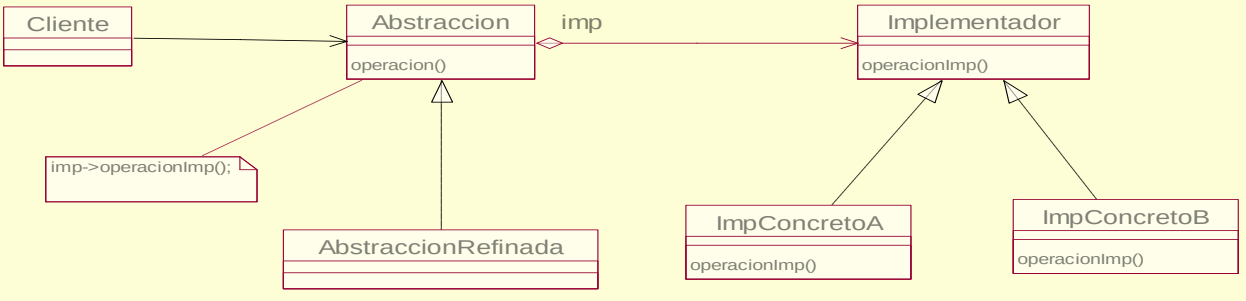
Herencia



Bridge / Handle

Estructura

Realización



Bridge

- **Aplicabilidad**
  - Se quiere evitar una ligadura permanente entre una abstracción y su implementación, p.e. porque se quiere elegir en tiempo de ejecución.
  - Abstracciones e implementaciones son extensibles.
  - Cambios en la implementación de una abstracción no deben afectar a los clientes (no recompilación).
  - Ocultar a los clientes la implementación de la interfaz.
  - Se tiene una proliferación de clases, como sucedía en la Motivación.

Bridge

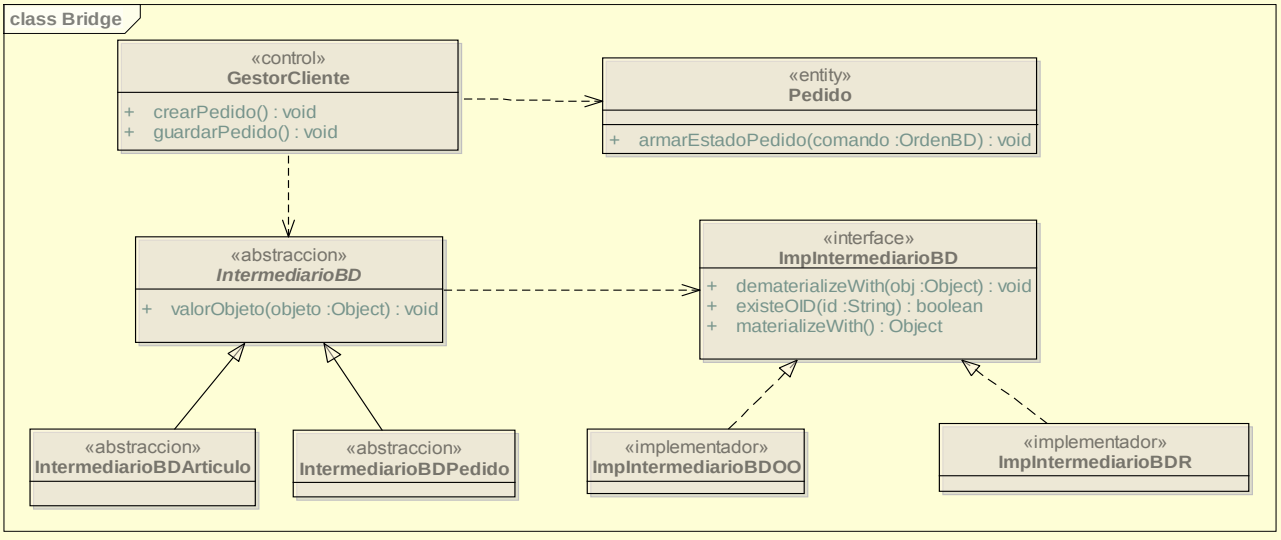
- **Consecuencias**
  - Un objeto puede cambiar su implementación en t.e.
  - Cambios en la implementación no motivarán recompilaciones de la clase Abstracción y sus clientes.
  - Se mejora la extensibilidad
  - Se ocultan detalles de implementación a los clientes

Bridge

- **Implementación**
  - Aunque exista una única implementación puede usarse el patrón para evitar que un cliente se vea afectado si cambia.
  - ¿Cómo, cuándo y dónde se decide que implementación usar?
    - Constructor de la clase *Abstraccion*
    - Elegir una implementación por defecto
    - Delegar a otro objeto, p.e. un objeto factoría
  - Se puede usar herencia múltiple para heredar de la abstracción y de una implementación concreta, pero se liga una interface a una implementación de forma permanente.

Bridge : Aplicación

Estructura



Bridge : Aplicación

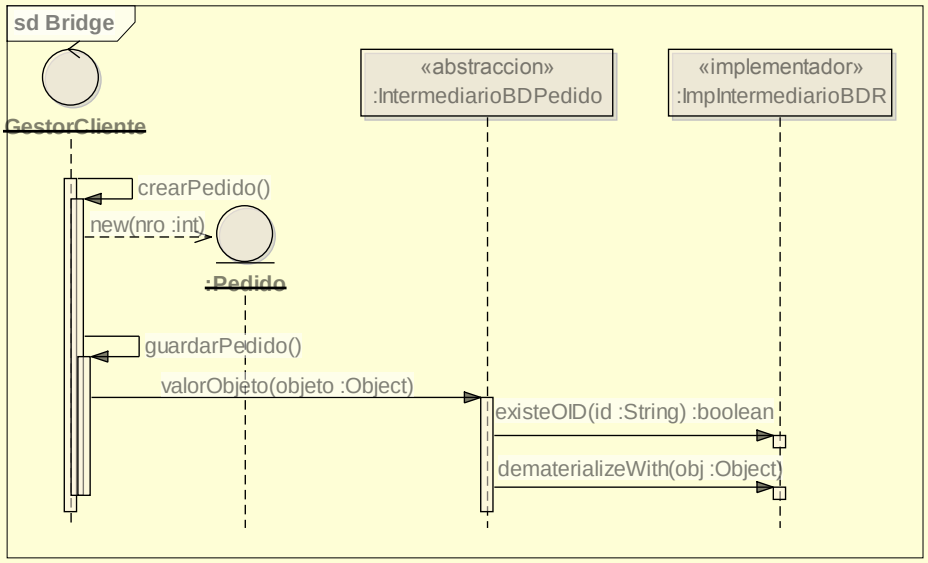
- **Participantes**
  - **Abstracción [**IntermediarioBD**]:**
    - Define la Interfaz de la abstracción, manteniendo referencia al objeto implementador
  - **Abstracción Refinada [**IntermediarioBDProd; IntermediarioBDPedido**]:**
    - extiende la interfaz definida por la Abstracción
  - **Implementador [**ImpIntermediarioBD**]:**
    - Define la interfaz de las clases de implementacion.
  - **Implementador Concreto [**ImpIntermediarioBDOO, ImpIntermediarioBDR**]:**
    - Implementa la interfaz implementador y define su implementación concreta

Bridge : Aplicación

- **Procedimiento**
  - El **GestorPedido** debe actualizar el estado del objeto. Para ello, pide colaboración al **IntermediarioBDPedido** para que se encargue de guardarlo en la base de datos.
  - **IntermediarioBDPedido**, a su vez, debe resolver el "guardado" en función a la tecnología de BD correspondiente, por lo que pide colaboración a **ImpIntermediarioBDR ,** quien se encarga de resolver las operaciones necesarias para actualizar el objeto en la base de datos

Dinámica

Bridge : Aplicación

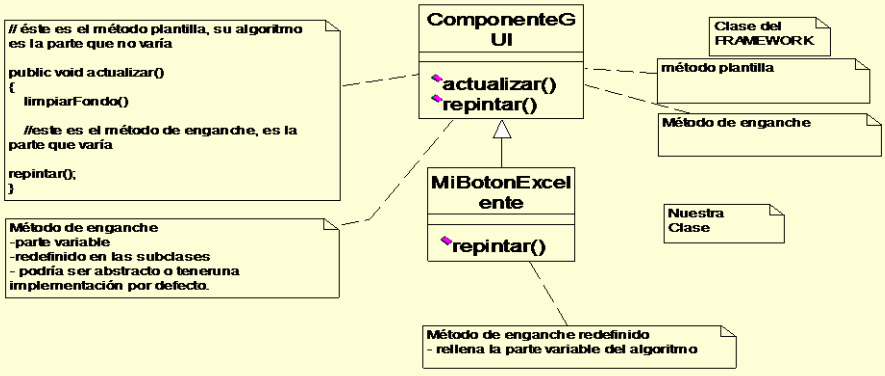


Template Method

- **Propósito**
  - Define el esqueleto (esquema, patrón) de un algoritmo en una operación, difiriendo algunos pasos a las subclases. Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.
- **Motivación**
  - Fundamental para escribir código en un framework.
  - Clase *Aplicación* que maneja objetos de la clase *Documento*: método *OpenDocument*

• Patrones de Creación
• Patrones de Estructura
• Patrones de Comportamiento

Método Plantilla para un Framework de GUI



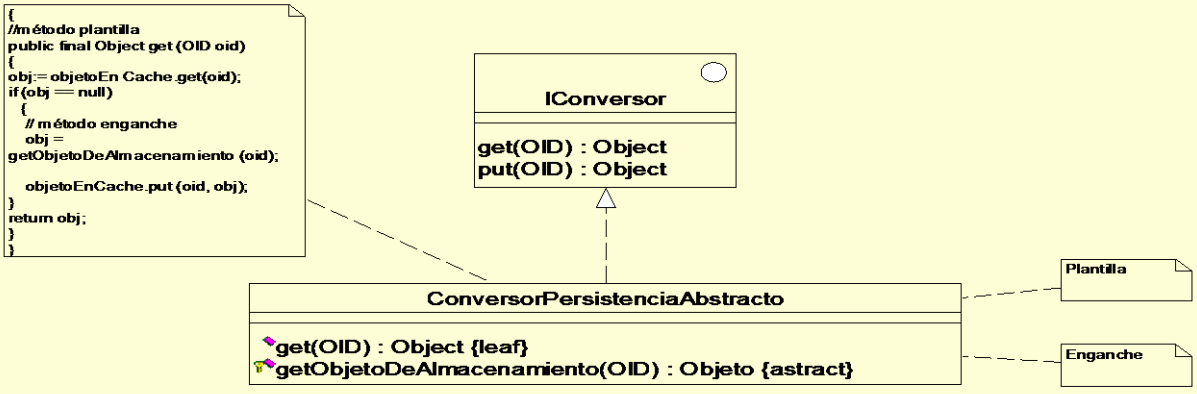
Materialización con el Método Plantilla

- La estructura básica del algoritmo para materializar un objeto es:

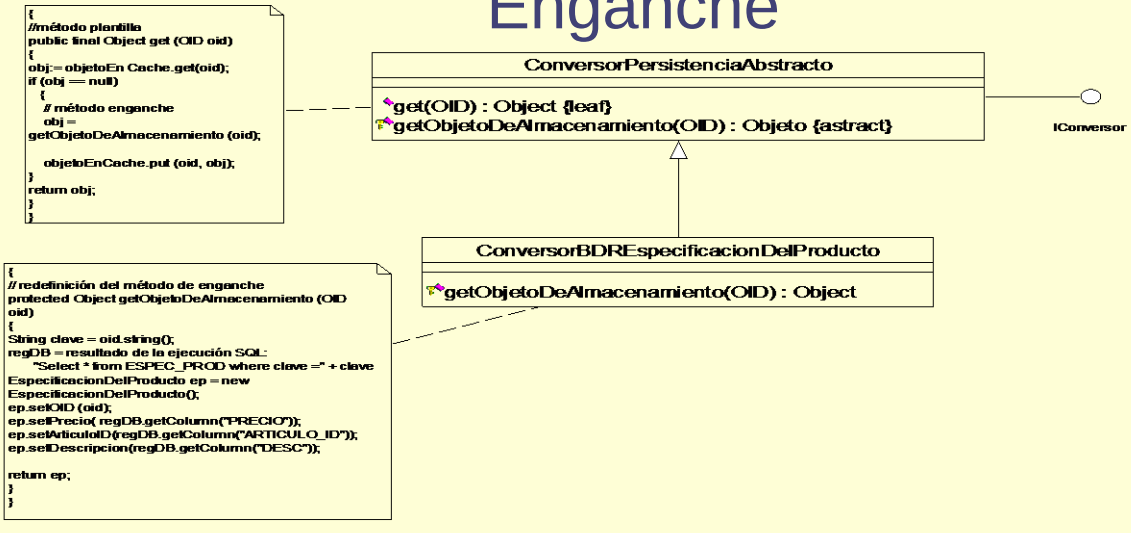
```
if (objeto está en caché)
    return objeto
else
    crear un objeto a partir de su representación en
     el almacenamiento
    guardar el objeto en la caché
    return objeto
```

- El punto de variación es la manera de crear el objeto a partir del almacenamiento.

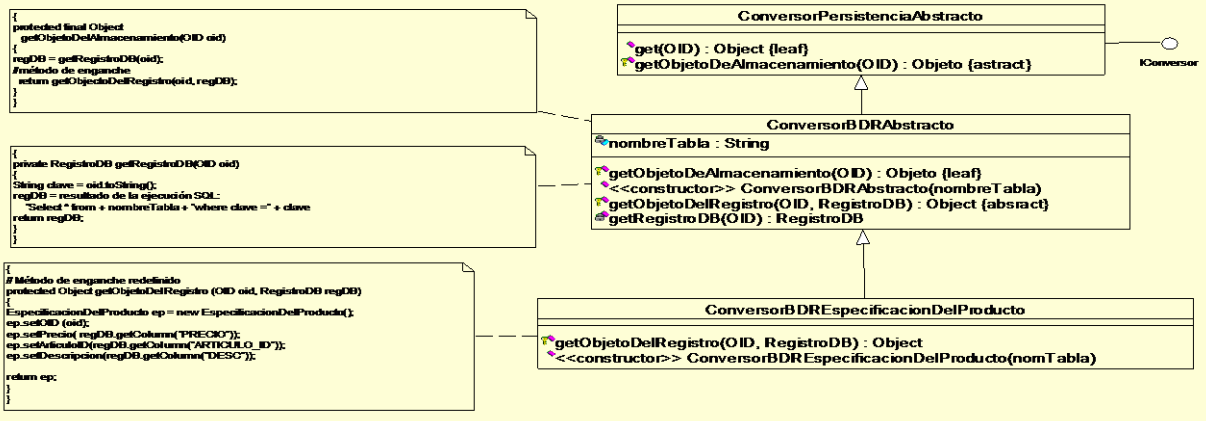
Método Plantilla para Objetos Conversores



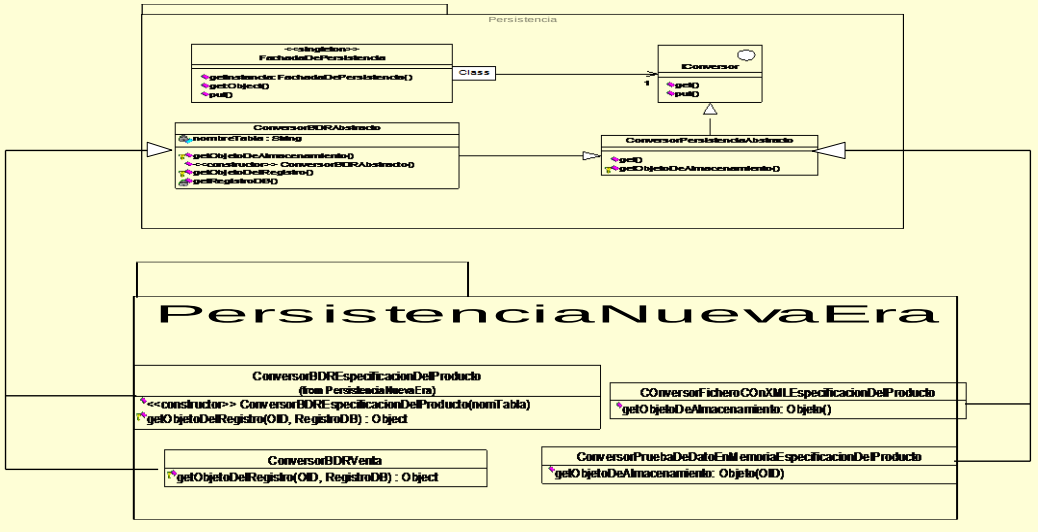
Redefinición del Método de Enganche



Ajustando el código con el Método Plantilla



Framework de Persistencia



Patrón: Gestión de Caché

- Es conveniente mantener los objetos materializados en un caché local para mejorar el rendimiento de y dar soporte a las operaciones de gestión de las transacciones.
- El patrón propone que el ConversorBDR sea el responsable de mantener esta caché, es decir cada conversor de objetos persistente sea responsable de mantener su propia caché.
- Al materializar los objetos, quedan en el caché con su OID como clave.
- El conversor primero busca en la caché para evitar materializaciones innecesarias.

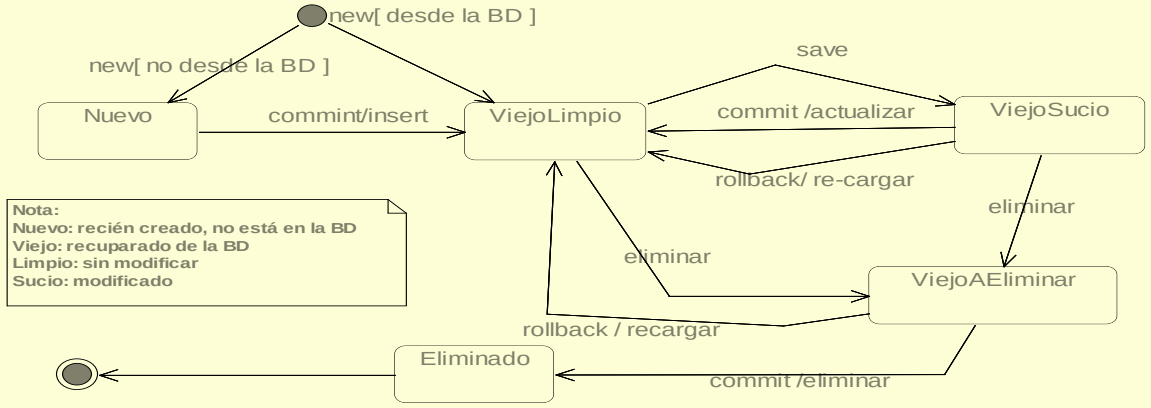
Sentencias SQL

- Sentencias SQL en distintas clases conversores no es terrible, pero se puede mejorar…
- En lugar de eso:
  - Existe una única clase *OperacionesBRD*, donde se reúnen las operaciones SQL (SELECT, INSERT…)
  - Las clases conversores colaboran con ella para obtener un registro de BD o un conjunto de registros (por ejemplo ResultSet)
- Con esta solución se obtienen los siguientes beneficios:
  - Facilita mantenimiento y rendimiento.
  - Encapsulamiento de métodos y detalles de acceso.

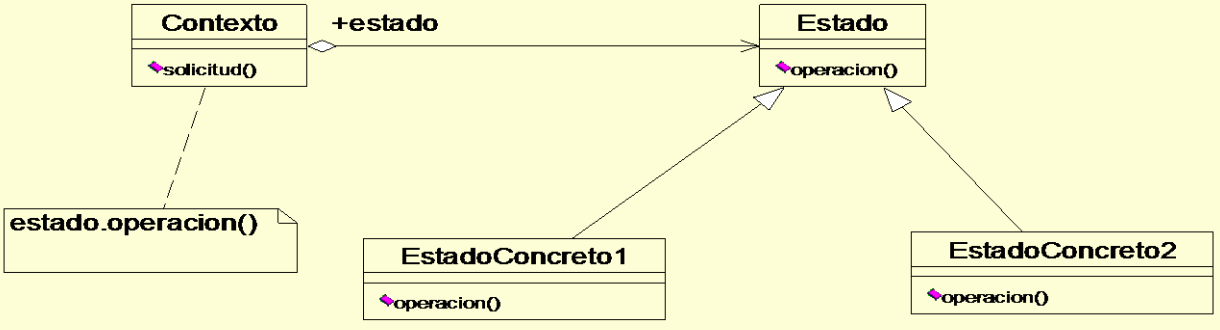
Estados Transaccionales

- Los objetos persistentes pueden insertarse, eliminarse o modificarse.
- Operar sobre un objeto persistente no provoca una actualización inmediata sobre la BD, más bien se debe ejecutar una operación commit explícita.
- Además considerar que la respuesta a una operación depende del estado de las transacción del objeto.

Máquina de Estado: Objeto Persistente

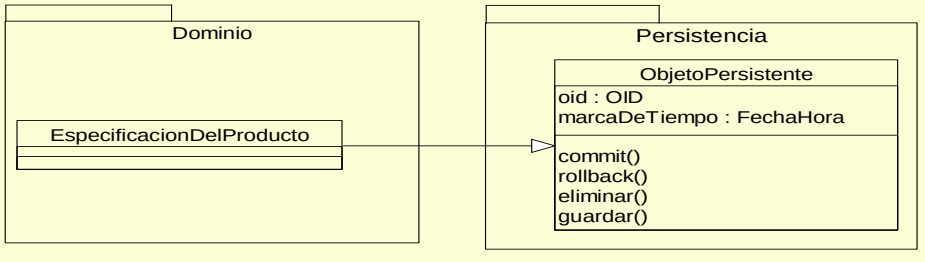


State



Objetos Persistentes

- La situación genérica de diseño responde a esta estructura:



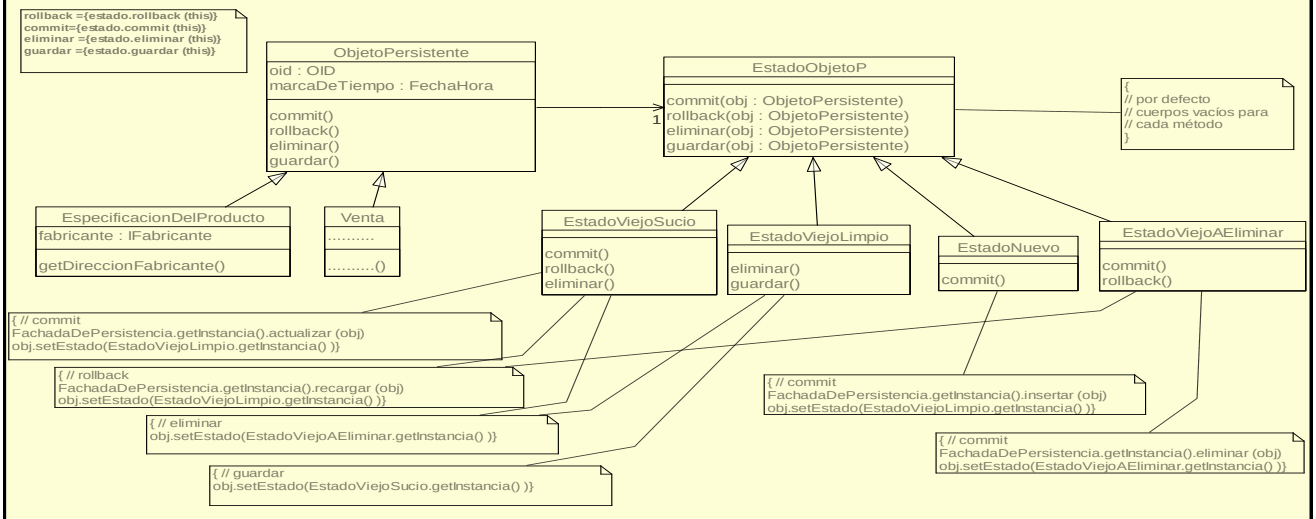
Objeto Persistente y el Patrón Estado

- Esta es la situación que se resolverá con el patrón Estado, los métodos commit y rollback requieren una estructura similar de lógica de casos, basada en el estado de la transacción, ambos métodos ejecutan diferentes acciones pero tienen estructuras lógicas similares.

```
Public void commit()
{
Switch (estado)
{
Case VIEJO_SUCIO
  //…
  break;
Case VIEJO_SUCIO
  //…
  break;
…
}
```

```
Public void rollback()
{
Switch (estado)
{
Case VIEJO_SUCIO
  //…
  break;
Case VIEJO_SUCIO
  //…
  break;
…
}
```


Aplicación del Patrón Estado



Diseño de una transacción

- Transacción: unidad de trabajo (conjunto de tareas) cuya terminación es atómica.
- En los servicios de persistencia incluyen: inserción, actualización, eliminación de objetos.
- Una transacción puede incluir por ejemplo: 2 inserciones, una actualización y tres eliminaciones.
- Considerar que el orden en que la aplicación añade las tareas podría no reflejar el mejor orden de ejecución. Las tareas necesitan que se ordenen justo antes de la ejecución.
- Para ello se utiliza el Patrón COMMAND.

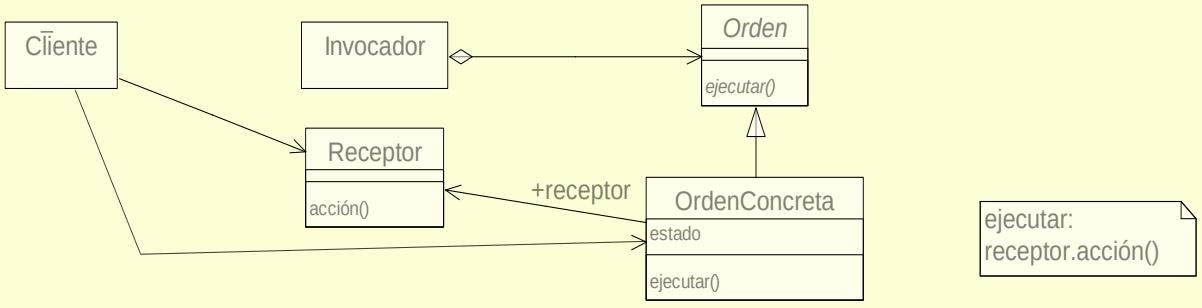
Command

- **Propósito**
  - Encapsula un mensaje como un objeto, permitiendo parametrizar los clientes con diferentes solicitudes, añadir a una cola las solicitudes y soportar funcionalidad deshacer/rehacer (undo/redo)
- **Motivación**
  - Algunas veces es necesario enviar mensaje a un objeto sin conocer el selector del mensaje o el objeto receptor.
  - Por ejemplo widgets (botones, menús,..) realizan una acción como respuesta a la interacción del usuario, pero no se puede explicitar en su implementación.

• Patrones de Creación
• Patrones de Estructura
• Patrones de Comportamiento

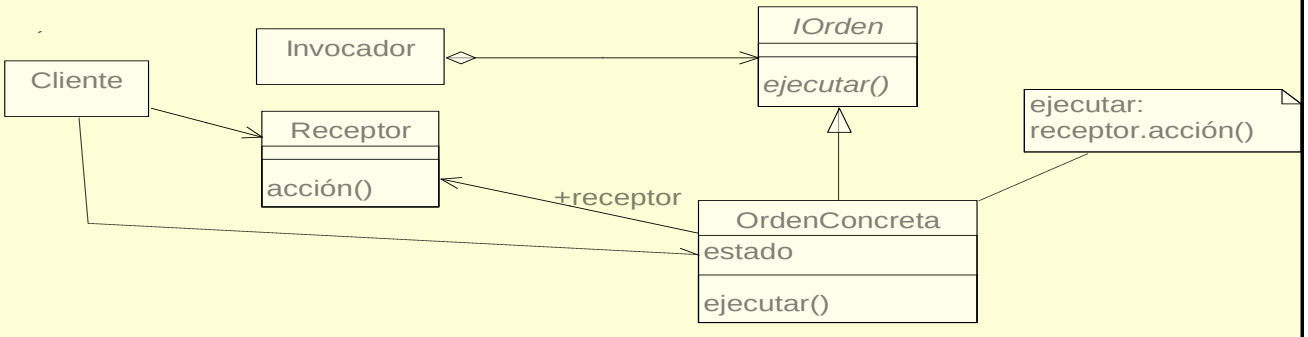
Estructura Command

Herencia



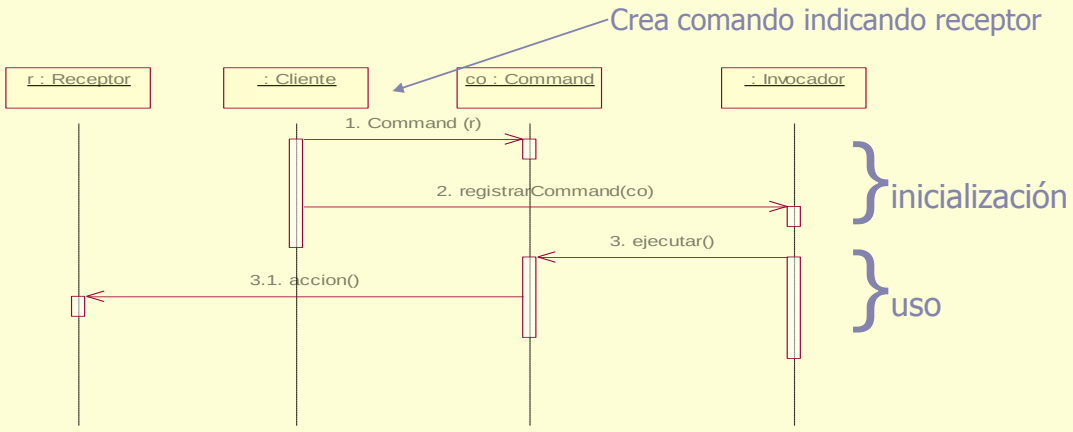
Estructura Command

Realización



Command

Colaboración



Command

- **Aplicabilidad**
  - Parametrizar objetos por la acción a realizar (en lenguajes procedurales con funciones Callback: función que es registrada en el sistema para ser llamada más tarde
  - Especificar, encolar y ejecutar mensajes en diferentes instantes: un objeto Command tiene un tiempo de vida independiente de la solicitud original.
  - Soportar facilidad undo/redo: la operación execute puede guardar información para revertir su efecto.
  - Recuperación de fallos.
  - Modelar transacciones vía conjunto de comandos

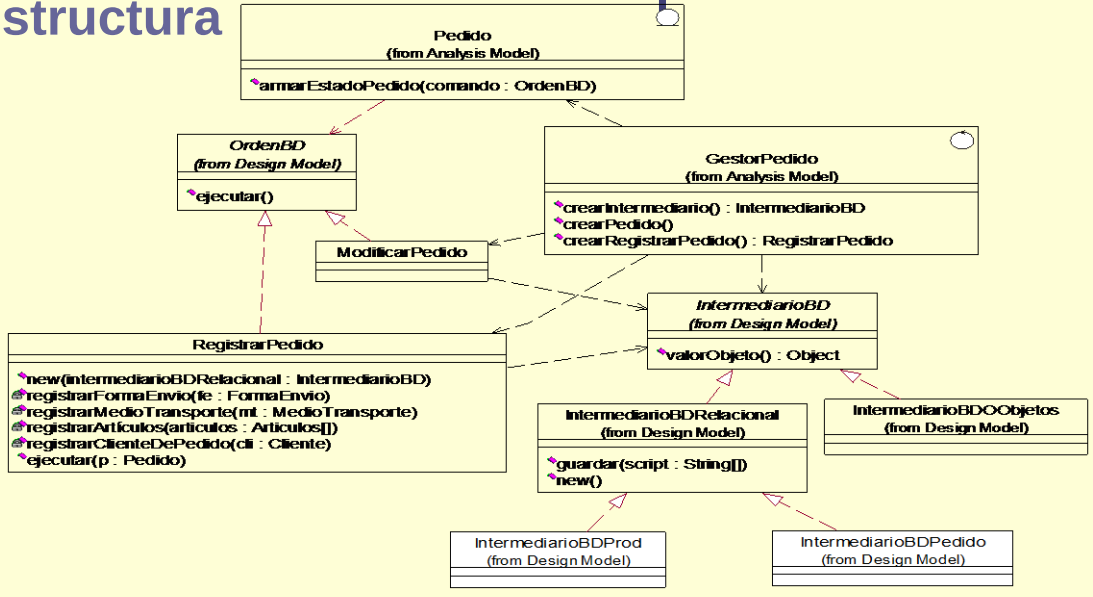
- Patrones de Creación
- Patrones de Estructura
- **Patrones de Comportamiento**

Command

- **Consecuencias**
  - Desacopla el objeto que invoca la operación del objeto que sabe cómo realizarla.
  - Cada subclase CommandConcreto especifica un par receptor/accion, almacenando el receptor como un atributo e implementando el método ejecutar.
  - Objetos command pueden ser manipulados como cualquier otro objeto.
  - Se pueden crear command compuestos (aplicando el patrón Composite).
  - Es fácil añadir nuevos commands.

Command : Aplicación

Estructura



Command : Aplicación

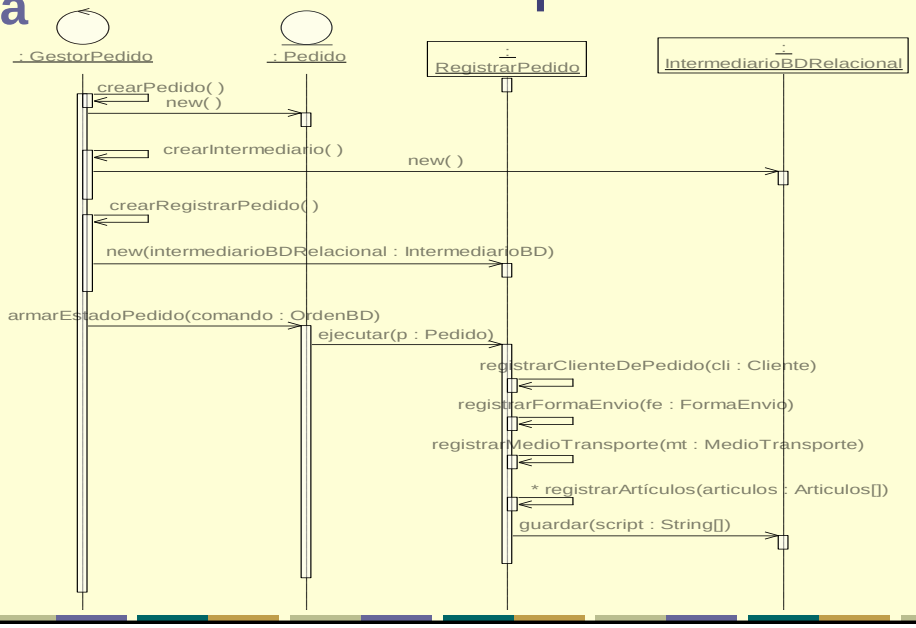
- **Participantes**
  - **Orden** [OrdenDB]: Declara una interfaz para ejecutar una acción.
  - **OrdenConcreta** [RegistrarPedido, ModificarPedido]: Define un enlace para que el objeto Receptor realice una acción.
  - **Cliente** [GestorPedido]: Crea un objeto OrdenConcreta y le indica el Receptor correspondiente.
  - **Invocador** [Pedido]: Solicita a la orden que ejecute una operación.
  - **Receptor** [IntermediarioBD, IntermediarioBDRelacional, IntermediarioBDOObjetos]: Sabe cuando realizar las operaciones asociadas a una petición**.**

Command : Aplicación

- **Procedimiento**
  - El **GestorPedido** crea la instancia del **IntermediarioBD** que utilizará para guardar el **Pedido** recién creado.
  - A continuación, crea la **OrdenBD** –comando- apropiada para registrar el pedido.
  - Luego le solicita al mismo **Pedido** que se guarde.
  - El **Pedido** ejecuta la orden para que se registre el mismo.
  - La Orden **RegistrarPedido** llama a sus propios métodos para realizar la operación solicitada, y finalmente le pide colaboración a un **IntermediarioBD** para guardarse en la base de datos.

Command : Aplicación

Dinámica

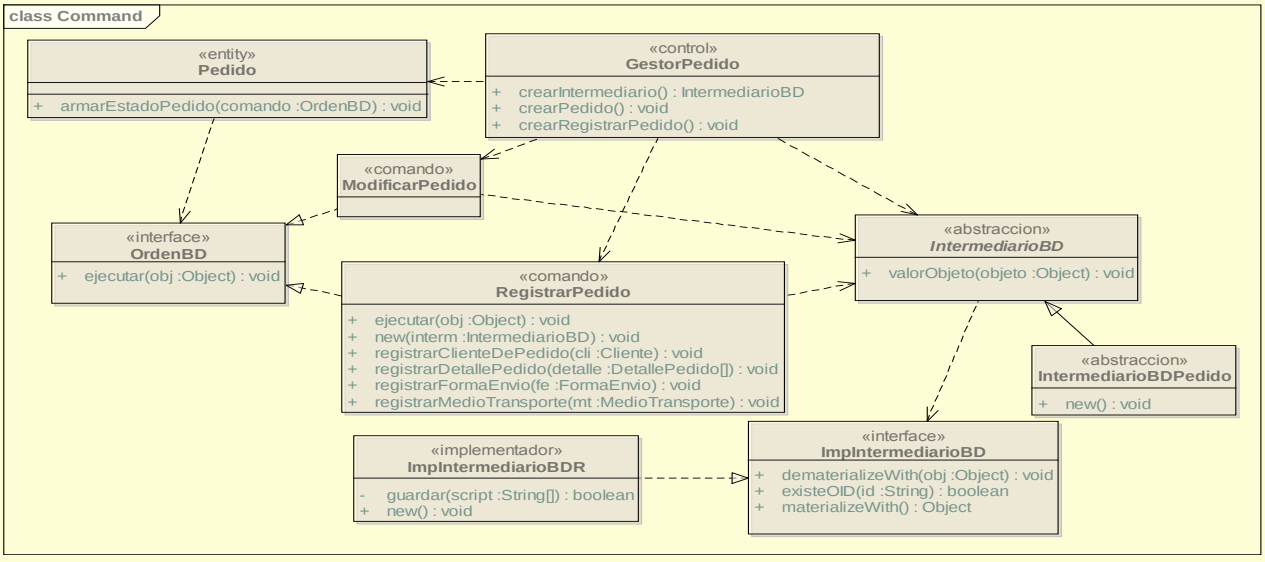


Command : Aplicación

- **Ventajas**
  - Reduce el acoplamiento: La clase **GestorPedido** no necesita conocer quien resuelve su petición para registrar un pedido en la base de datos.
  - Las órdenes son métodos complejos tratados como clases, por lo que tienen las ventajas de éstas: Reusabilidad, Cohesión, ser referenciada desde otra clase.
  - Para órdenes demasiado complejas es posible diseñar una agregación con una **OrdenMacro** que contiene objetos Orden. (Patrón Composite)
  - Existe mayor flexibilidad para agregar órdenes, ya que simplemente se agregan clases, no es necesario modificar internamente a otras clases.

Command : Aplicación

Estructura



Command : Aplicación

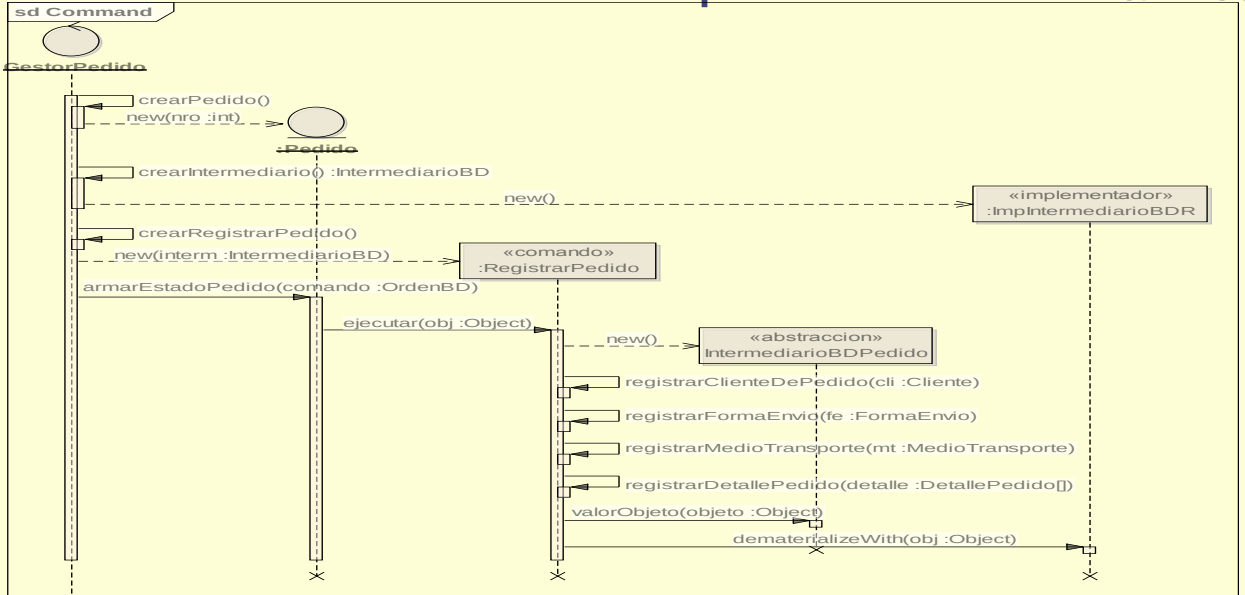
- **Participantes**
  - **Orden** [OrdenDB]: Declara una interfaz para ejecutar una acción.
  - **OrdenConcreta** [RegistrarPedido, ModificarPedido]: Define un enlace para que el objeto Receptor realice una acción.
  - **Cliente** [GestorPedido]: Crea un objeto OrdenConcreta y le indica el Receptor correspondiente.
  - **Invocador** [Pedido]: Solicita a la orden que ejecute una operación.
  - **Receptor** [IntermediarioBD, IntermediarioBDPedido]: Delega las operaciones asociadas a una petición a los implementadores**.**
  - **Ejecutor:** [ImpIntermediarioBD, ImpIntermediarioBDR]: Resuelve la petición de guardar el pedido en la base de datos correspondiente, en este caso BDR**.**

Command : Aplicación

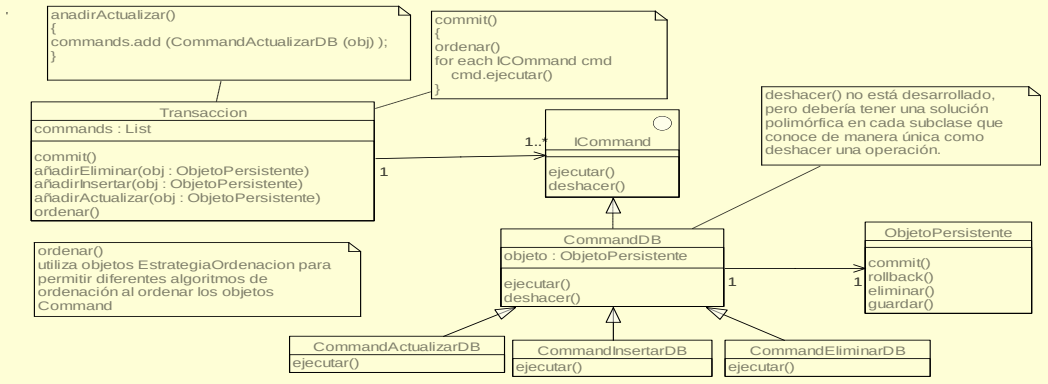
- **Procedimiento**
  - El **GestorPedido** crea la instancia del **ImpIntermediarioBDR** que utilizará para guardar el **Pedido** recién creado.
  - A continuación, crea la **OrdenBD** –comando- apropiada para registrar el pedido.
  - Luego le solicita al mismo **Pedido** que se guarde.
  - El **Pedido** ejecuta la orden para que se registre el mismo.
  - La Orden **RegistrarPedido** llama a sus propios métodos para realizar la operación solicitada, y finalmente le pide colaboración a un **IntermediarioBDPedido** para guardarse en la base de datos.

Command : Aplicación

Dinámica



Patrón Command para las operaciones de la Base de Datos



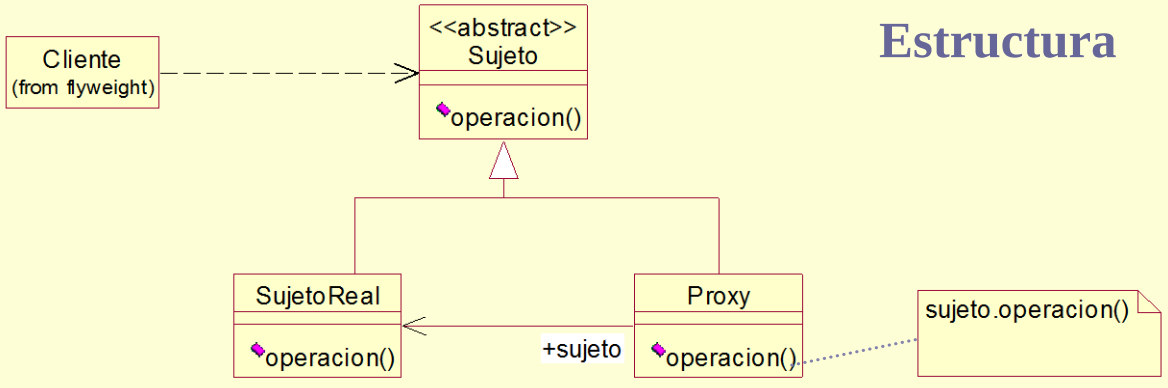
Materialización Perezosa con un Proxy Virtual

- A veces es conveniente diferir la materialización de objetos hasta que sea necesario, por cuestiones de rendimiento.
- La materialización diferida de objetos hijos se llama **materialización perezosa**.
- Esta materialización se puede implementar utilizando un Proxy Virtual.
- Un Proxy Virtual es un proxy para otro objeto (el sujeto al que se quiere acceder realmente) que materializa el sujeto real cuando se referencia por primera vez.

Proxy

- **Propósito**
  - Proporcionar un sustituto (*surrogate*) de un objeto para controlar el acceso a dicho objeto.
- **Motivación**
  - Diferir el costo de crear un objeto hasta que sea necesario usarlo: creación bajo demanda.
  - Un editor de documentos que incluyen objetos gráficos.
  - ¿Cómo ocultamos que una imagen se creará cuando se necesite?: manejar el documento requiere conocer información sobre la imagen.

Proxy



Estructura

Proxy

- **Aplicabilidad**
  - Siempre que hay necesidad de referenciar a un objeto mediante una referencia más rica que una referencia normal.
  - Situaciones comunes;

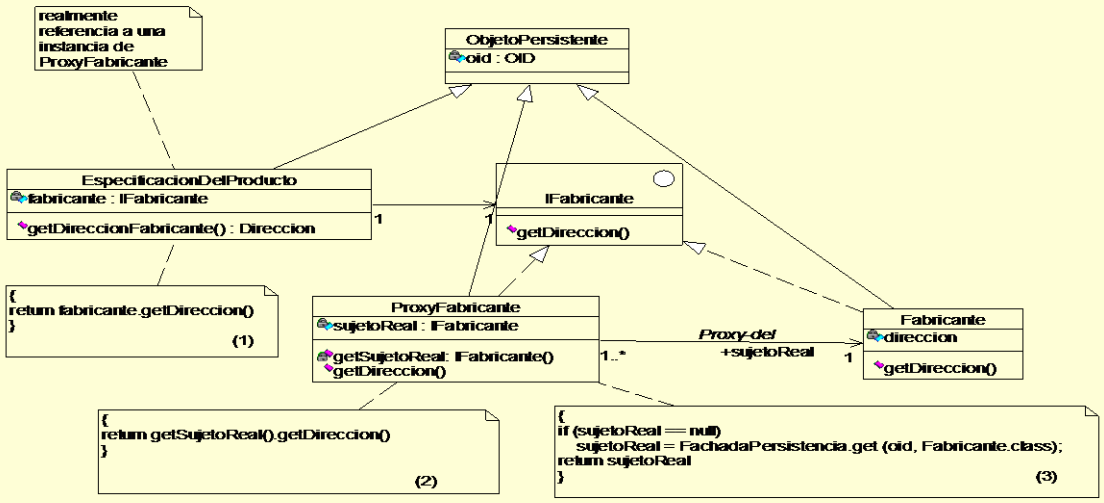
1) Proxy **acceso remoto** (acceso a un objeto en otro espacio de direcciones)
2) Proxy **virtual** (crea objetos grandes sobre demanda)
3) Proxy para **protección** (controlar acceso a un objeto)
4) **Referencia inteligente** (*smart reference*), por ejemplo un decorador, contar cantidad de referencias, etc.

Proxy

- **Consecuencias**
  - Introduce un nivel de indirección para:

1) Un *proxy* remoto oculta el hecho que objetos residen en diferentes espacios de direcciones.
2) Un *proxy* virtual tales como crear o copiar un objeto bajo demanda.
3) Un *proxy* para protección o las referencias inteligentes permiten realizar tareas de control sobre los objetos accedidos.

Materialización Perezosa con un Proxy Virtual



¿Quién crea el Proxy Virtual?

- La clase que crea la correspondencia entre la EspecificacionDelProducto y la Base de Datos,
- Esta clase es responsable de decidir cuando se materializa un objeto, cual de los objetos debe materializarse de manera **impaciente** y cual de manera **perezosa**.

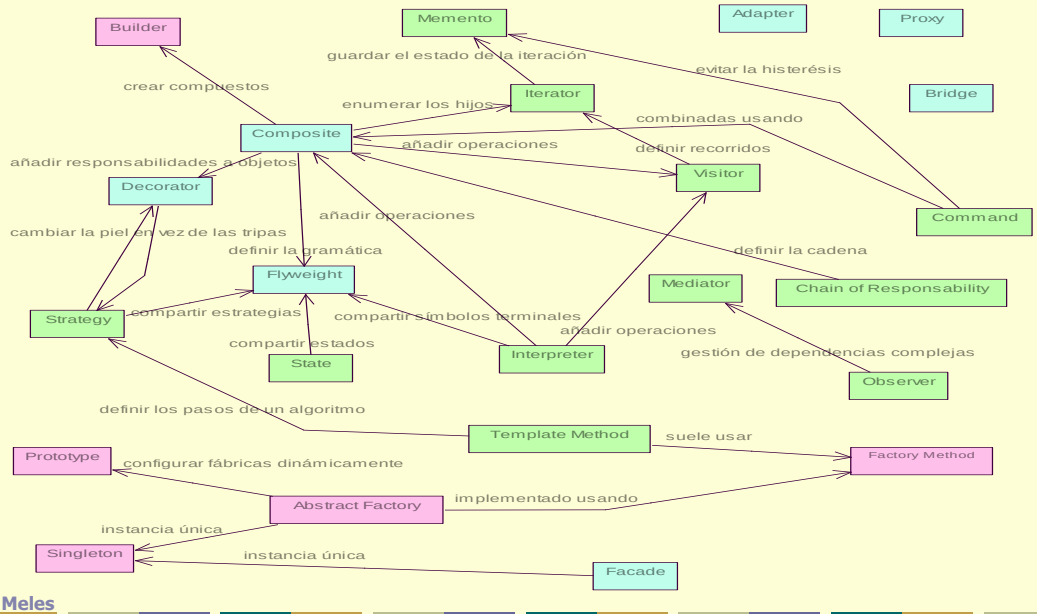
Ejemplo de Materialización Impaciente

```
// MATERIALIZACION IMPACIENTE DEL FABRICANTE
class ConversorBDREspecificacionDelProducto extends ConversorPersitenciaAbstracto
{
Protected Object getObjetoDeAlmacenamiento (OID oid)
{
ResultSet rs =
    OperacionesBDR.getInstancia().getDatosEspecificacionDelProducto (oid);
EspecificacionDelProducto ep = new EspecificacionDelProducto ();
ep.setPrecio (rs.getDouble ("Precio"));

            // aquí está la escencia
String claveAjenaFabricante = rs.getString ("FAB_OID");
ep.setFabricante ((Ifabricante)
     FachadaDePersistencia.getInstancia().get(oidFab, Fabricante.class);
…
}
```


Ejemplo de Materialización Perezosa

```
// MATERIALIZACION PEROZOSA DEL FABRICANTE
class ConversorBDREspecificacionDelProducto extends ConversorPersitenciaAbstracto
{
Protected Object getObjetoDeAlmacenamiento (OID oid)
{
ResultSet rs =
    OperacionesBDR.getInstancia().getDatosEspecificacionDelProducto (oid);
EspecificacionDelProducto ep = new EspecificacionDelProducto ();
ep.setPrecio (rs.getDouble ("Precio"));

            // aquí está la escencia
String claveAjenaFabricante = rs.getString ("FAB_OID");
OID oidFab = new  OID (claveAjenaFabricante);
ep.setFabricante (new ProxyFabricante (oidFab));
…
}
```

Relaciones entre los patrones



Causas comunes de rediseño

Crear un objeto indicando la clase.	*Abstract factory, Factory Method, Prototype*
Dependencia de operaciones específicas	*Chain of Responsability, Command*
Dependencia de plataformas Hw o Sw	*Abstract Factory, Bridge*
Dependencia sobre representación de objetos.	*Abstract factory, Bridge, Memento, Proxy*
Dependencias de algoritmos	*Builder, Iterator, Strategy, Template Method, Visitor*
Acoplamiento fuerte entre clases	*Abstract factory, Bridge, CoR, Command, Facade, Mediator, Observer*
Extender funcionalidad mediante subclases	*Bridge, CoR, Composite, Decorator, Observer, Strategy, State*
Incapacidad de cambiar clases convenientemente	*Adapter, Decorator, Visitor*

Bibliografía

- **Design Patterns: Elements of Reusable Object-oriented software.** Gamma, Helm, Johnson, Vlissides
- **Applying UML and Patterns.** Craig Larman
- **Head First Design Patterns**. Freeman & Freeman

Historia de Cambios

Fecha	Versión	Descripción	Autor
31/07/2005	1.0	Versión Inicial	Judith Meles
09/09/2010	1.8	Se cambian a UML 2.0 con EA las implementaciones de los patrones	Gerardo Boiero Judith Meles
08/09/2011	1.9	Se realizan cambios al dominio de la venta digital y por consecuencia en los ejemplos de aplicación de los patrones	Gerardo Boiero