

4.2 UML: diagramas de diseño

Ingeniería del Software Avanzada
Técnicas de análisis y diseño

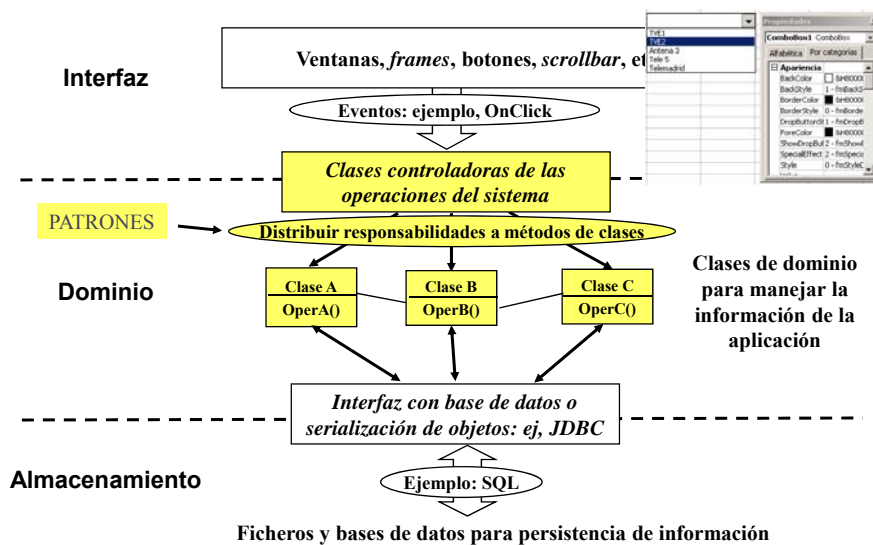
Objetivos

- Comprender la notación de los diagramas de colaboración
- Ser capaces de generar un diagrama de colaboración para cada caso de uso a partir de la información de la fase de análisis
- Conocer los patrones y cómo aplicarlos a los diagramas de colaboración
- Conocer las nociones de visibilidad, dependencias, etc. y aplicar las consecuencias de los diagramas de colaboración a la creación de un diagrama de clases de diseño

Arquitectura

- Asumimos arquitectura de 3 capas:
 - **Presentación:** ventanas, formularios, etc.
 - **Lógica de aplicación o dominio:** reglas de negocio
 - **Almacenamiento:** persistente
- Aislar la lógica en n-capas
 - **Objetos de dominio:** conceptos del problema
 - **Servicios:** funciones de interacción de BD, comunicación,...
- Distintas configuraciones Cliente/Servidor
- Ventajas:
 - Aislar la lógica, distribución en varios nodos, asignar tareas de diseño por capas
 - Apoyo en patrón separación modelo-vista

Aplicaciones de tres capas



Patrones

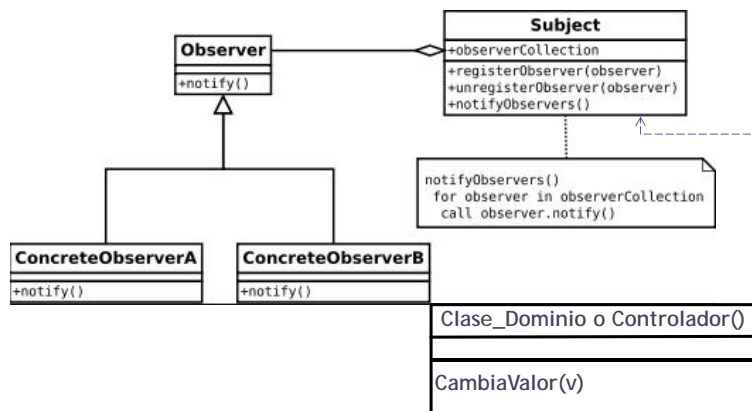
- Principios generales y guías de desarrollo
- Patrón:
 - Descripción de un problema y su solución
 - Nombre propio: facilita la comunicación
 - “Pareja problema/solución con un nombre y aplicable a otros contextos y que sugiere cómo usarlo en situaciones nuevas”
- Los patrones no suelen contener ideas nuevas
 - Solo formalizar la experiencia de desarrollo

Patrón: separación modelo-vista

- Problema:
 - Desacoplar objetos del dominio y de vistas (interfaz GUI), para mayor reutilización y aislar los cambios en interfaz
- Solución:
 - Definir clase de dominio que no tengan acoplamiento ni visibilidad directa con la vista: conservar datos de dominio en clases de aplicación y no en ventanas
 - Clases de dominio: no conocerán existencia de ventanas
 - Clases de presentación (interfaz): tienen acceso a dominio
 - No se enviarán datos ni mensajes a ventanas: sólo dar respuestas
 - En ocasiones (control, simulación), habrá necesidad de comunicación indirecta

Comunicación indirecta

- Recurrir al patrón observador (publicar-suscribir)
- Problema
 - Un cambio de estado (evento) ocurre dentro de un editor de evento y otros objetos están interesados, pero el editor no debería tener conocimiento de suscriptores
- Solución
 - Definir un sistema de notificación indirecta de eventos
 - Las ventanas se suscriben
 - Crear una clase de gestión de eventos
 - No requiere acoplamiento directo
 - Puede transmitirse a cualquier número de suscriptores



El método `CambiaValor` pide a **Subject** que ejecute el método `notifyObservers`
 En todos estos métodos se pasa el valor `v`
Subject es visible a la clase **Dominio o Controlador** (flecha de dependencia)

Fase de diseño

- Tres actividades básicas:
 - 1. Crear diagramas de colaboración o secuencia para casos de uso u operaciones
 - A partir del contrato y de diagrama de secuencia del sistema
 - Aplicar patrones GRASP y otros
 - Principalmente para la capa de dominio
 - 2. Crear el modelo de clases de diseño:
 - Actualizando el modelo conceptual con métodos, nuevas clases, especificando visibilidad, etc.
 - 3. Documentar la arquitectura del sistema
 - Diagramas de despliegue y componentes

Fases 1 y 2: Concentrarse en capa de dominio

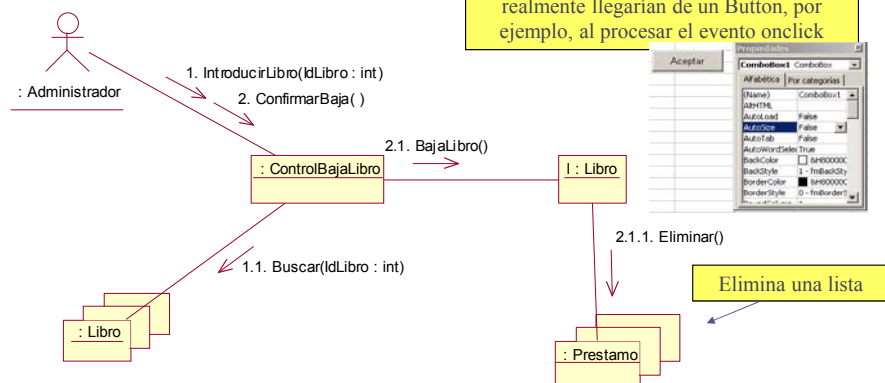
- Capa de interfaz:
 - Viene dada por la interfaz aprobada de cada caso de uso
 - Basada en prototipos enseñados al cliente
 - Usa elementos gráficos (AWT, Swing, etc.): button, fileDialog, textField, etc.
- Capa de almacenamiento:
 - Viene dada por la necesidad de almacenar datos (persistencia) usando base de datos o ficheros (por ejemplo, xml)
 - El modelo conceptual de clases o el modelo E/R deben tener reflejo en esta capa
 - Uso de interfaces: JDBC, etc.

Diagramas de colaboración

- Muestran la interacción entre objetos
 - Incluye las relaciones entre objetos
 - No indica tiempo: sólo números de secuencia
- Como en diagrama de secuencia:
 - Afecta a un subconjunto del modelo de clases:
 - Conjunto de clases y sus relaciones implicados en una acción
 - Marca el contexto de una interacción
- Colaboración:
 - Diagramas de objetos (clases) con mensajes en las asociaciones

Notación (I)





Diagrama para un caso de uso



Notación (II)

- Diagrama representa vínculos entre objetos
 - Incluye vínculos temporales
 - Paso de parámetros, variables locales
 - Los vínculos incluyen flecha de navegabilidad
- Mensajes
 - En flecha junto a línea de unión con nombre, parámetros, objeto de retorno y tipos
 - Añaden número de secuencia comenzando en el 1
 - En flujo procedimental, los números se anidan
 - Iteración: * [i:=1..n]; condición: [x > 0] (mismo número de secuencia)
- Aparecen todos los objetos implicados en ejecución
 - Incluso los afectados indirectamente o sólo accedidos

Notación (III)

- Clases y ejemplares:
 - Clase
 
 - Ejemplar
 
 - Ejemplar con nombre
 
 - Multiobjetos: conjunto (p.ej., *vector*)
 
- Sintaxis de mensajes:
 - retorno:= mensaje (param: tipo param) : tipo retorno

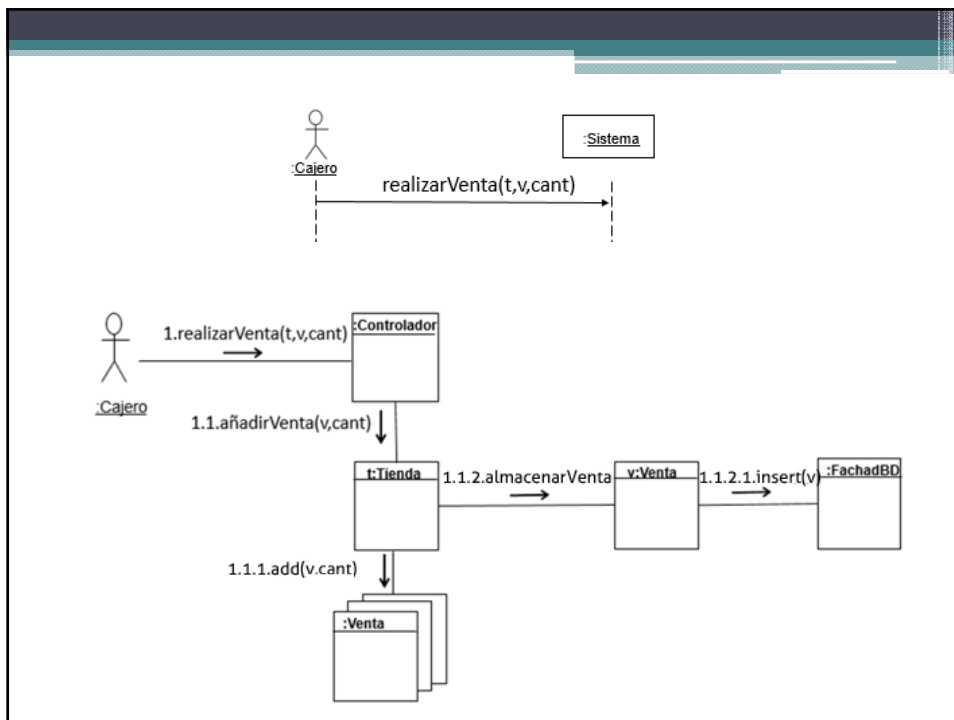
Crear diag. de colaboración

Información de partida:

• Diagrama de secuencia del sistema; Modelo de clases conceptual; Contrato de operación.

• Pasos:

- Dibujar el actor (Rational: arrastrar)
- Incorporar una clase de control específica para el caso
 - Por ejemplo, para “Alta de libro” poner “ControlAltaLibro” o “ControlBibliotecario”
- Incorporar los mensajes del diagrama de secuencia desde el actor a la clase de control:
- Asignar respuesta a mensajes a las clases:
 - Analizando en orden de ejecución las acciones a realizar
 - Cuidado con búsquedas y tratamiento de asociaciones
- Representar acciones como mensajes
- Revisar diagrama para asegurar cumplimiento de patrones



Patrones GRASP

- Muchos patrones
 - Continuamente se proponen más
- Patrones GRASP
 - *General Responsibility Assignment Software Patterns*
 - Propuestos por C.Larman
 - Aplicables a asignación de responsabilidades para clases, al dibujar diagramas de colaboración
- GRASP:
 - Experto, Creador, Gran Cohesión, Bajo Acoplamiento, Controlador
 - Gran cohesión: influye en experto y controlador
 - Bajo acoplamiento: influye en creador y controlador

Metapatrones

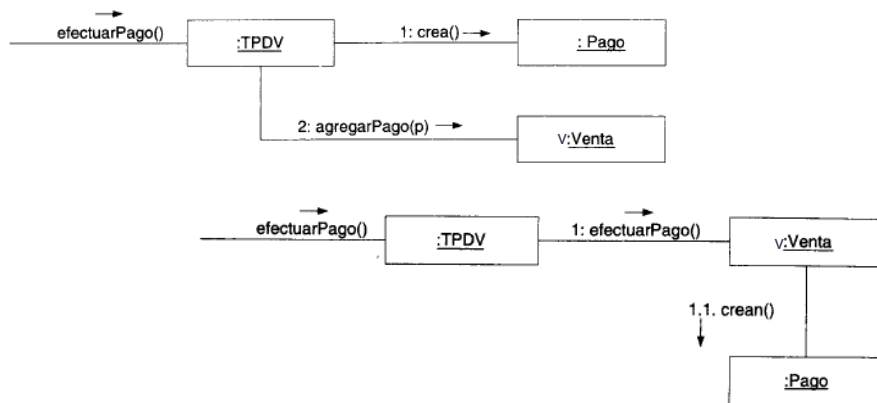
- Metapatrones: guías generales de diseño
- Bajo acoplamiento: relación entre clases
 - X tiene atributo de ejemplar de Y o clase Y; X tiene operación que referencia (parámetro) a Y o clase Y; X es subclase directa o indirecta de Y; Y es una interfaz y X lo implementa
- Problemas de acoplamiento:
 - sensible a cambios, menos entendibles, etc.
- Alta cohesión: coherencia de funciones de una clase
 - Ideal: una función por clase
 - Deseable: varias funciones similares (usan mismos datos o hacen cosas similares)

Bajo acoplamiento y Alta cohesión: clases con poca visibilidad y con pocas funciones y relacionadas entre sí

Tenemos que crear una instancia de Pago y asociarla a la Venta. Venta ya tiene visibilidad de Pago por otros casos de uso:

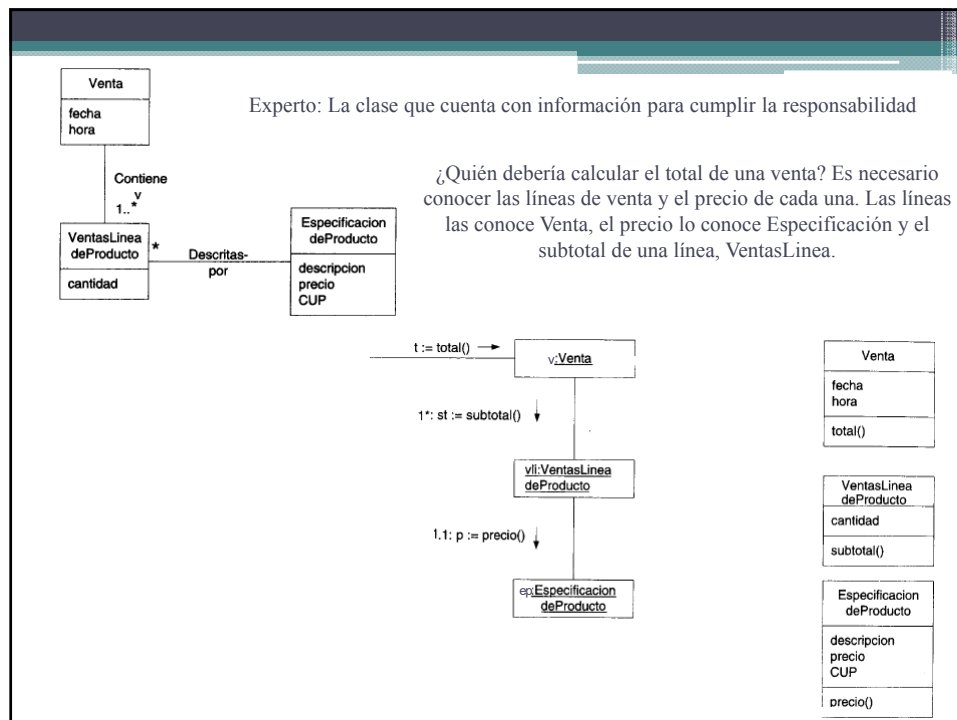
En el primer diagrama TPDV ve dos clases, en el segundo solo una (acoplamiento)

En el primero TPDV realiza dos funciones (crea Pago y lo agrega a Venta), en el segundo solo le pide a Venta que efectúe un pago y Venta lo crea agregándolo (cohesión)



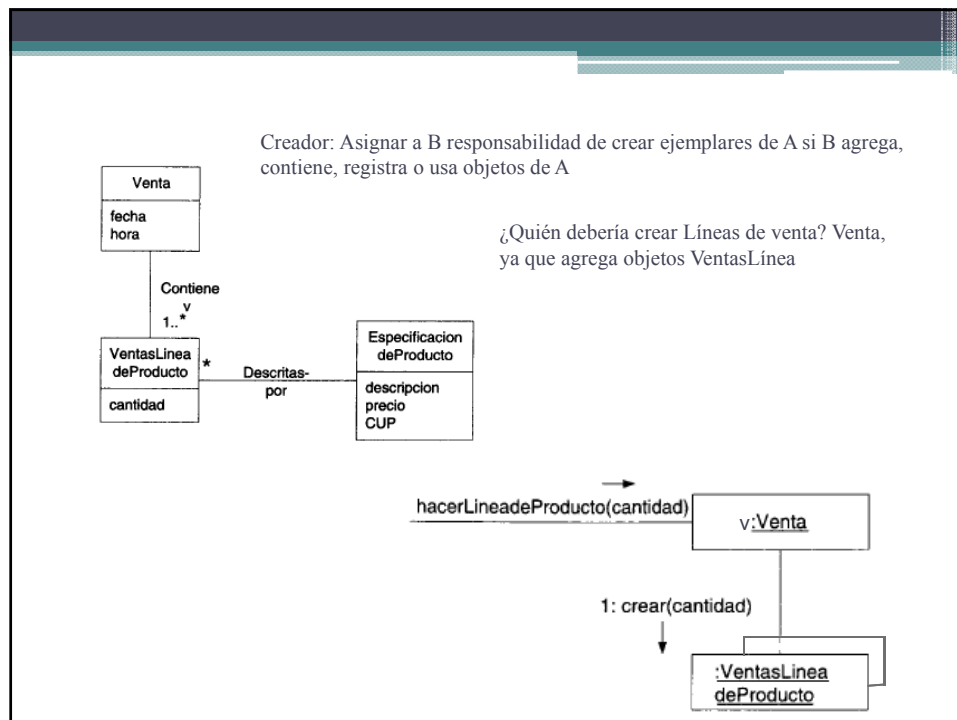
Experto

- Problema:
 - ¿Responsabilidades de manejo de información?
- Solución:
 - Asignar responsabilidad al experto en información
 - La clase que cuenta con información para cumplir responsabilidad
- Explicación:
 - Pueden existir muchos expertos parciales para colaborar
- Beneficios:
 - Bajo acoplamiento y gran cohesión



Creador

- Problema:
 - ¿Responsable de crear/borrar un ejemplar de alguna clase?
- Solución:
 - Asignar a B responsabilidad de crear ejemplares de A si:
 - B agrega objetos de A, B contiene objetos de A, B registra ejemplares de A, B usa objetos de A, B tiene datos de inicio de A
- Explicación:
 - Contenedor, Registrador, Agregador...crean contenido, registro, agregado
 - Se aplica también al borrado de objetos
- Beneficios:
 - Bajo acoplamiento: creado es visible ya al creador



Controlador

Ingeniería de software

- Problema:
 - ¿Responsable de manejar eventos del sistema (evento generado por un actor)?
- Solución:
 - Asignar la responsabilidad al “controlador”, que será una clase que decide que clases de dominio intervienen para responder al evento
- Explicación:
 - Opciones para determinar el controlador:
 - Que trate todos los eventos del sistema global o de la empresa, solo si hay pocos eventos
 - Que trate todos los eventos generados por un actor
 - Que trate eventos para cada caso (controlador de caso)
- Beneficios:
 - Bajo acoplamiento: independencia interfaz-dominio y Gran cohesión: Los controladores solo se encargan de gestionar eventos y solo lo hacen ellos
 - Problema de controladores saturados: demasiados eventos disminuyen los beneficios anteriores

Controlador: la clase interfaz no ve a las clases de dominio, solo a la clase controlador

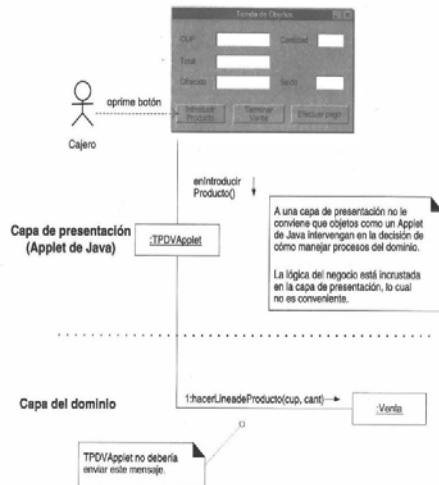


Figura 18.18 Acoplamiento inadecuado de la capa de presentación a la capa del dominio.

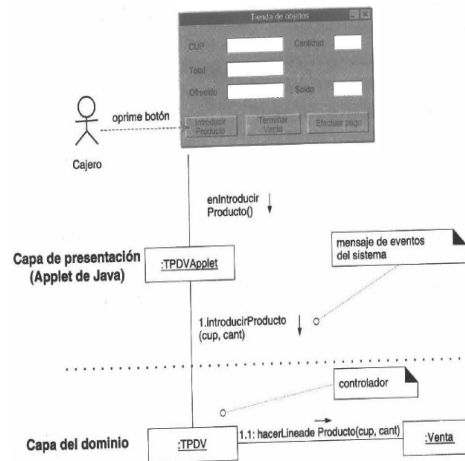


Figura 18.17 Acoplamiento adecuado de la capa de presentación a la capa del dominio.

Diagrama de clases de diseño

- **Exige crear antes:**
 - Diagramas de colaboración de diseño: identifican métodos y nuevas clases (por ejemplo: de control, de lista de objetos, etc.)
 - Modelo conceptual: a partir de él, se agregan detalles de definición de clases o incluso nuevas clases
 - Suelen crearse en paralelo a los diagramas de interacción
 - Herramientas como Rational lo hacen
- **Deben incluir los detalles para pasar al código:**
 - Clases, asociaciones y atributos
 - Interfaces, con operaciones y constantes
 - Métodos
 - Información sobre tipos de atributos
 - Navegabilidad
 - Dependencias

Cómo crear diagrama de diseño

- Identificar nuevas clases a partir de diagramas de interacción
- Dibujar en un diagrama
- Incluir atributos siguiendo el modelo conceptual
- Agregar nombres de métodos del diagrama de colaboración de diseño
- Incorporar tipos a atributos y métodos
- Agregar flechas de navegación (visibilidad de atributo)
- Agregar líneas de dependencias (visibilidad no de atributos: global o declarada localmente)

Métodos en el diagrama de clases de diseño

- **Métodos u operaciones**
 - Implementación de un servicio que puede ser requerido por otro objeto
 - Una clase puede tener 0, 1 o varias operaciones
 - Especificación:
 - Nombre: si hay varias palabras, iniciales en mayúscula
 - Signatura: nombre, tipo y valores por defecto de todos los parámetros y, en el caso de funciones, un tipo de retorno

Cliente
valor () ponerSaldo (s: saldo) verEstado: estado

Visibilidad

- Capacidad de “ver” de un objeto a otro
 - Alcance o ámbito de objeto
- Cuatro tipos de visibilidad
 - Atributos
 - Parámetros
 - Declarada localmente
 - Global
- Para enviar mensajes a un objeto, debe ser visible
 - Lo visible o accesible para cada clase se debe determinar con los diagramas de colaboración

Visibilidad de atributos

- Visibilidad de atributos de A B:
 - B es atributo de A
 - Relativamente permanente porque persiste mientras existan A y B
- ```
Public class A
{
...
Private B ObjB;
...
}
```

## Visibilidad de parámetros

- Visibilidad de parámetros de A B:
    - B se transmite como parámetro de método de A
    - Relativamente temporal porque persiste sólo dentro del ámbito del método
- ```
HacerMetodo (B , int c)
{
...
}
```

Visibilidad local

- Visibilidad declarada localmente de A B
 - Se declara que B es objeto local dentro de un método de A
 - Es relativamente temporal porque sólo persiste en el ámbito del método
- ```
HacerMetodo (int d, int c)
{
B = obtener (d, c)
}
```



## Visibilidad global

- Visibilidad global de A B
  - Cuando B es global para A
  - Es relativamente permanente porque persiste mientras existan A y B
  - Medio más obvio:
    - Asignar ejemplar a variable global

## Navegabilidad

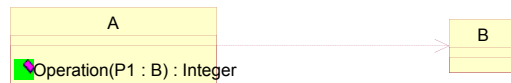
- Navegabilidad: Indica que un elemento usa otro y, por tanto, un cambio en ese elemento puede afectar a otro
- Relación de asociación
  - Relación de asociación con flecha si:
    - A envía mensaje a B
  - Visibilidad de atributos: flecha normal de asociación y navegabilidad
- Relación de dependencia
  - Visibilidad de parámetro, global o declarada local

## Notación de navegabilidad

- Atributos



- Parámetros



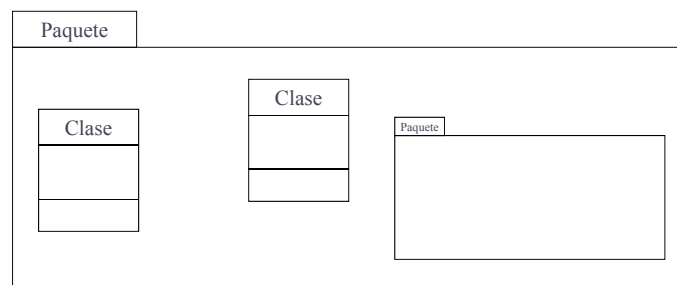
- Local:

- Metodo 1 (p1: p)
- b: B



## Organizar modelos: paquetes

- Útil para el diagrama de clases de diseño
- Permite grupos de elementos o subsistemas en UML
  - Define en un ámbito de nombres
- Notación

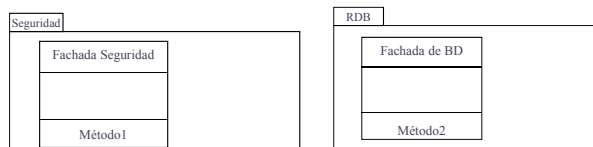


## Paquetes y capas

- **Formas de utilizar paquetes**
  - Aislar cada capa en un paquete: interfaz y dominio
  - También agrupar elementos para un servicio común, por ejemplo clases para cálculos estadísticos y matemáticos
  - Acceso a servicios del paquete: limitar acceso a una o pocas clases (fachada)
- **Patrones relacionados:**
  - Fachada y Separación modelo-vista

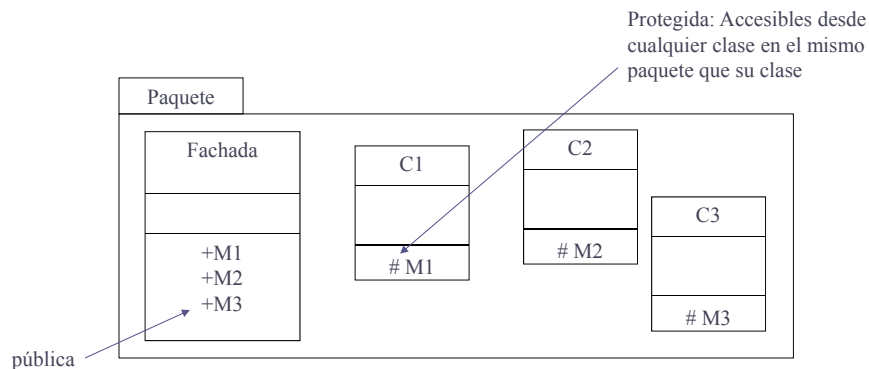
## Patrón adicional: fachada

- **Problema:**
  - Se requiere una interfaz unificada y común para un conjunto heterogéneo de interfaces
- **Solución:**
  - Definir una clase individual que unifique la interfaz y que se responsabilice de colaborar con clases de otros paquetes y con clases privadas del mismo paquete



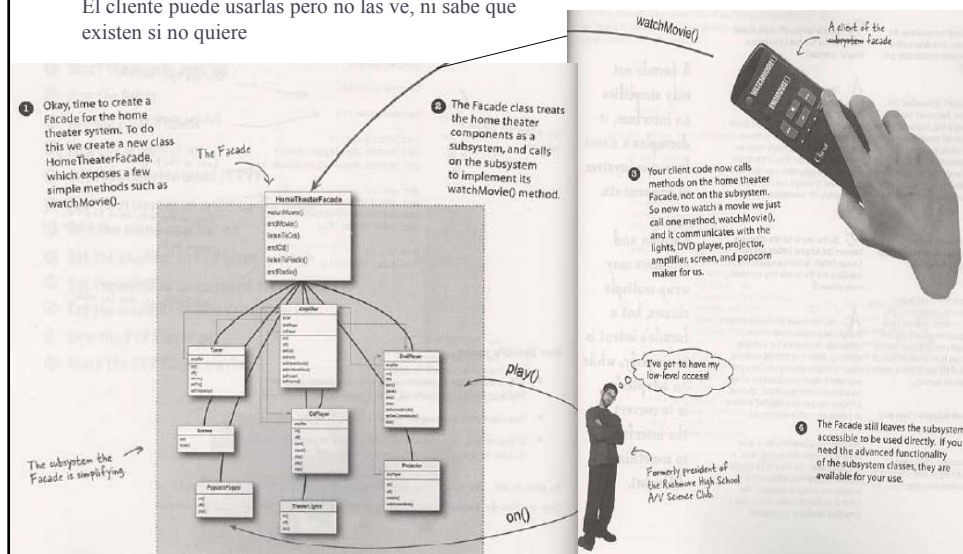
# Fachada

- Coordinado con la visibilidad de paquete

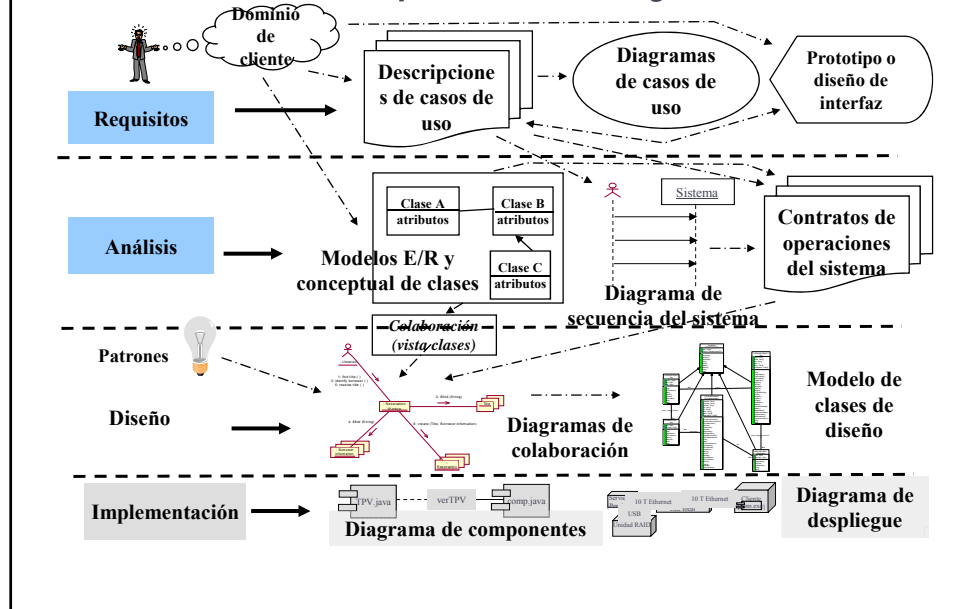


Fachada – Una clase que contiene métodos que combinan o llaman a métodos de otras clases

La clase fachada tiene visibilidad de las otras clases  
El cliente puede usarlas pero no las ve, ni sabe que existen si no quiere



## Conexión rápida fases y UML



## Referencias

- Libros:
  - E. Freeman et al., "Head First Design Patterns". O'Reilly Media, 2004