Facultad de Informática - UNLP
Orientación a Objetos - 2021

Introducción al Análisis y diseño orientado a Objetos (Cont.)

Prof. Roxana Giandini

Heurísticas para Asignación de responsabilidades (HAR)

Responsabilidades de los objetos

- Conocer
  - Conocer sus datos privados encapsulados
  - Conocer sus objetos relacionados
  - Conocer cosas derivables o calculables
- Hacer
  - Hacer algo él mismo
  - Iniciar una acción en otros objetos
  - Controlar o coordinar actividades de otros objetos

Heurísticas para Asignación de Responsabilidades

- La habilidad para asignar responsabilidades es extremadamente importante en el diseño.

- La asignación de responsabilidades generalmente ocurre durante la creación de diagramas de interacción.

- Experto en Información
- Creador
- Controlador
- Bajo Acoplamiento
- Alta Cohesión

Experto en Información (Experto)

Descripción: Asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad). Expresa la intuición de que los objetos hacen cosas relacionadas con la información que tienen.

- Para cumplir con su responsabilidad, un objeto puede requerir de información que se encuentra dispersa en diferentes clases → expertos en información "**parcial**".

Ejemplo: ¿Quién tiene la responsabilidad de conocer el monto **total** de una **compra**? … La compra.

Entonces:

LineaDeVenta es responsable de conocer el subtotal por cada ítem

EspecificaciónDelProducto es responsable de conocer el precio del item.

Creador

Descrpción: asignar a la clase B la responsabilidad de crear una instancia de la clase A si:

- B contiene objetos A (agregación, composición).
- B registra instancias de A.
- B tiene los datos para inicializar objetos A.
- B usa a objetos A en forma exclusiva.

Ejemplo: ¿Quién debe ser responsable de crear una LineaDeVenta?

… La compra

La intención del Creador es encontrar una clase que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento.

Controlador

Descripción: asignar la responsabilidad de manejar eventos del sistema a una clase que representa:

- El sistema global, dispositivo o subsistema

Ejemplo: ¿Quién debe ser el controlador de los eventos *ingresarLibro* o *finalizarCompra*?

… ManejadorCompras, Librería

- La intención del Controlador es encontrar manejadores de los eventos del sistema, sin recargar de responsabilidad a un solo objeto y manteniendo alta cohesión.

Bajo Acoplamiento

Descripción: asignar responsabilidades de manera que el acoplamiento permanezca lo más bajo posible.

El **acoplamiento** es una medida de **dependencia** de un objeto con otros. Es bajo si mantiene pocas relaciones con otros objetos.

- El alto acoplamiento dificulta el entendimiento y complica la propagación de cambios en el diseño.

- No se puede considerar de manera aislada a otras heurísticas, sino que debe incluirse como principio de diseño que influye en la elección de la asignación de responsabilidad.

Alta Cohesión

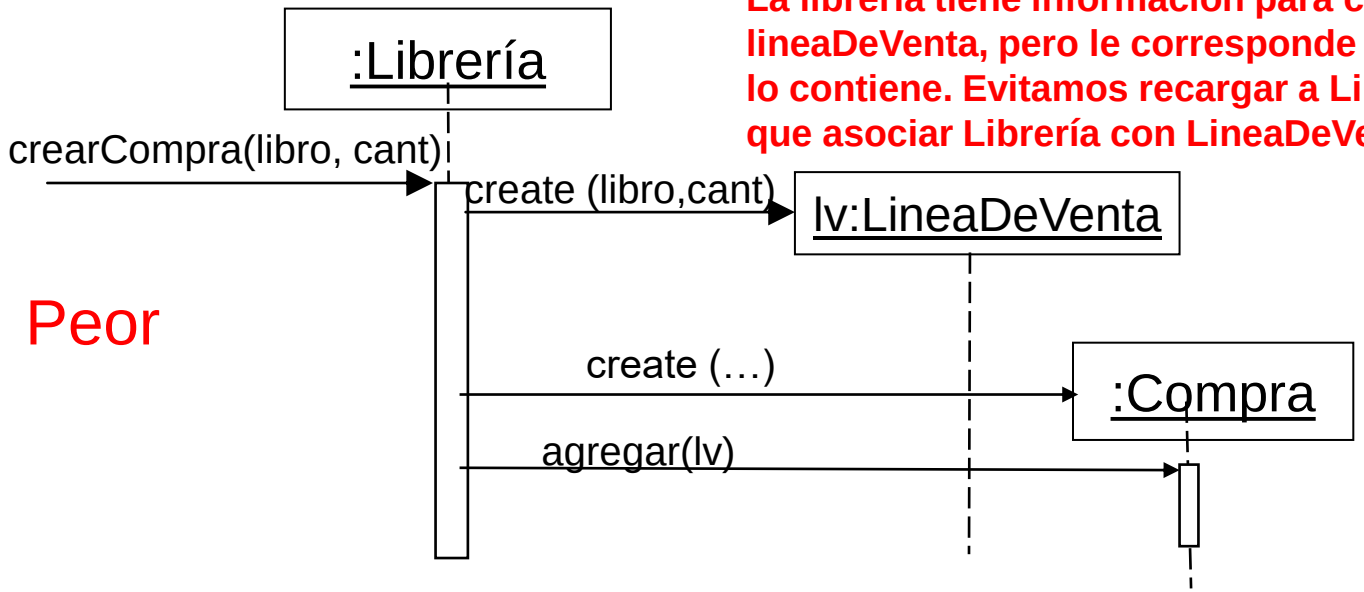
Descripción: asignar responsabilidades de manera que la cohesión permanezca lo más fuerte posible.

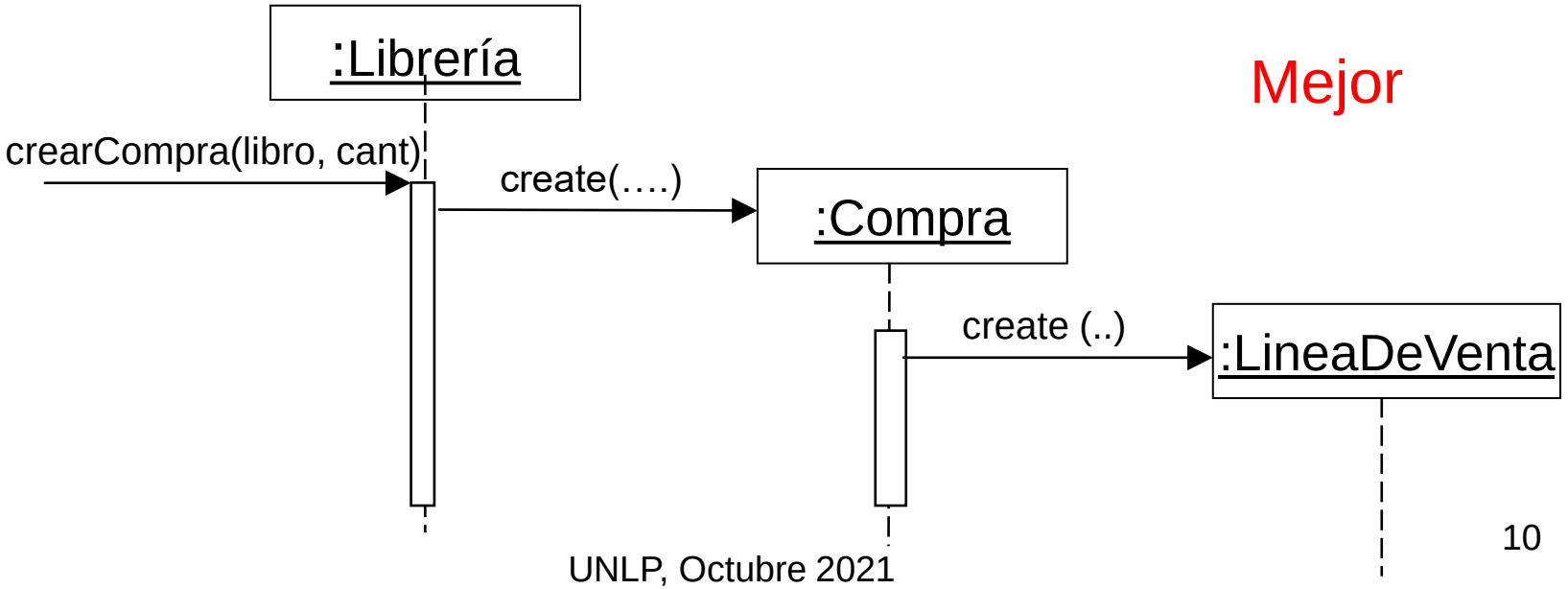
La **cohesión** es una medida de la fuerza con la que se relacionan las **responsabilidades** de un objeto, y la cantidad de ellas.

- Ventaja: clases más fáciles de mantener, entender y reutilizar.
- El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otras heurísticas, como Experto y Bajo Acoplamiento.

Bajo Acoplamiento y Alta Cohesión (Ejemplo)

La librería tiene información para crear la lineaDeVenta, pero le corresponde a Compra, que lo contiene. Evitamos recargar a Librería y tener que asociar Librería con LineaDeVenta.





Modelo de Diseño

Realizaciones de casos de uso: del Análisis al Diseño

- Los **casos de uso** sugieren los eventos del sistema que se muestran en los **diagramas de secuencia del sistema.**
- En los **contratos de las operaciones,** utilizando conceptos del **Modelo del Dominio**, se describen los efectos que dichos eventos (operaciones) producen en el sistema.
- Los **eventos** del sistema representan los **mensajes** que dan inicio a **Diagramas de Secuencia del Diseño,** mostrando las interacciones entre los objetos del sistema.
- Los objetos con sus métodos y relaciones se muestran en el **Diagrama de Clases del Diseño** (basado en el Modelo del Dominio).

Análisis
Casos de Uso
+
DSS
+
Modelo del Dominio
+
Contratos de Op.
+
HAR

Diseño
D. Secuencia
Diagrama de
Clases

Del Análisis al Diseño- Diagramas de interacción

- Cree un **diagrama de secuencia** por cada operación del sistema en desarrollo (la **operación es el mensaje de partida** en el diagrama).
- Si el diagrama queda complejo, sepárelo en diagramas menos complejos (uno por cada escenario).
- Use el **contrato de la operación** como punto de partida; piense en objetos que colaboran para cumplir la tarea (la mayoría de estos objetos están definidos en el modelo del dominio).
- Aplique las **Heurísticas para Asignación de Responsabilidades** (HAR) para obtener un mejor diseño.

Diagramas de interacción- Ejemplo

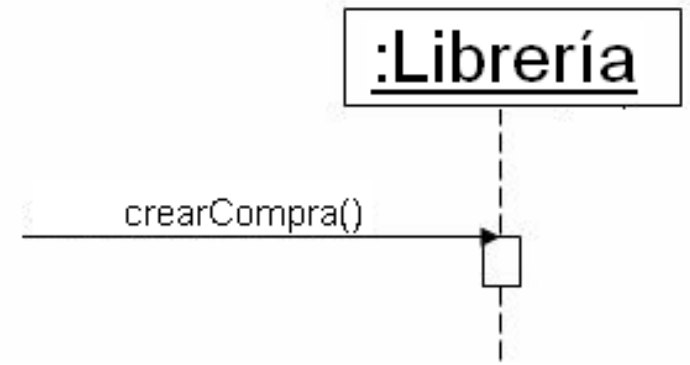
Crear una nueva compra.

… Comenzamos con el contrato de la operación.

Contrato: crear compra

Operación:	crearCompra ()
Referencias cruzadas:	Caso de Uso: Comprar libro
Precondiciones:	ninguna
Postcondiciones:	- Se creó una instancia de Compra compra (creación de inst.) - compra se asoció con la Librería (formación de asoc.) - Se inicializaron los atributos de compra

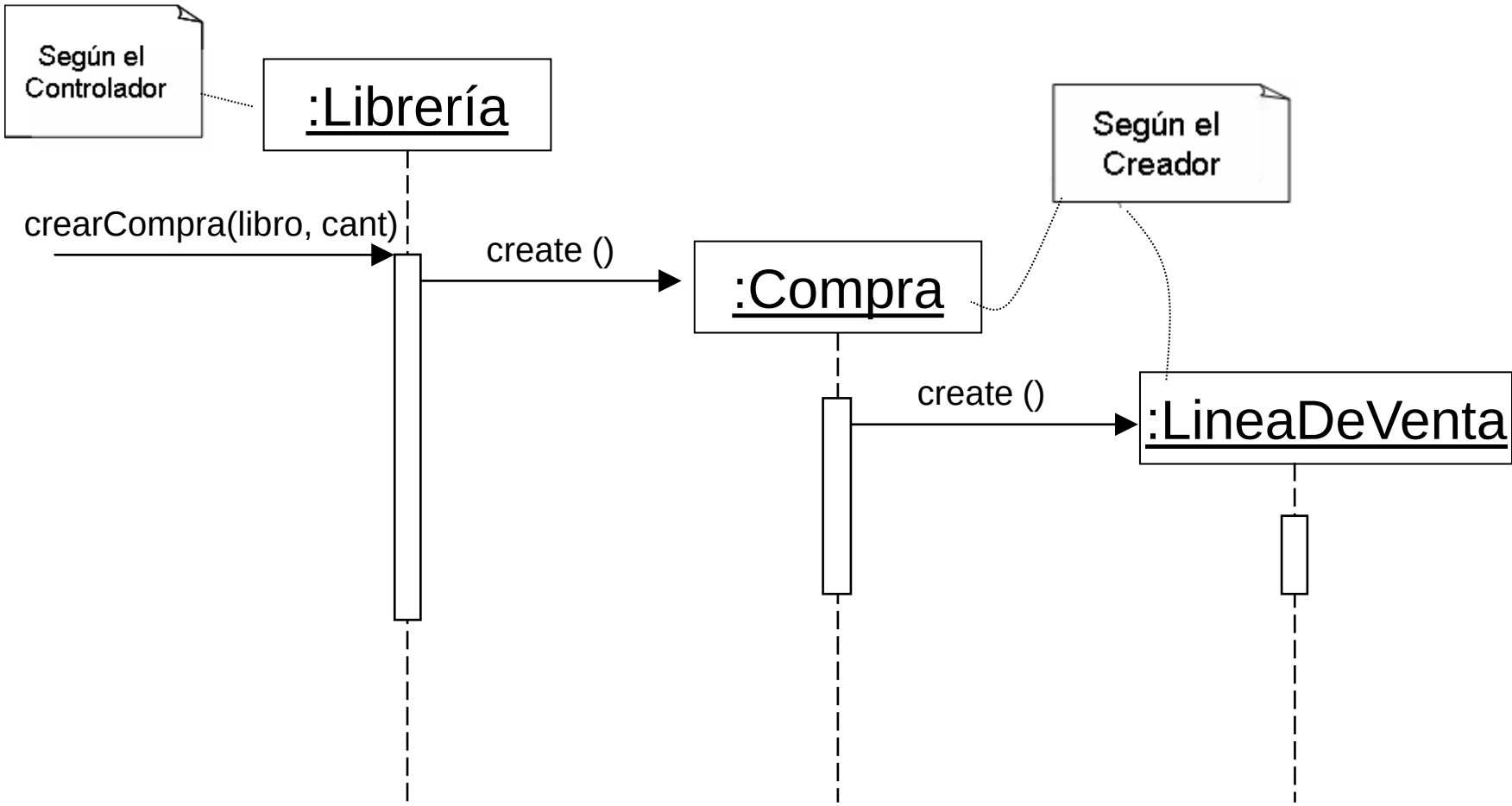
… Elección de la clase controlador



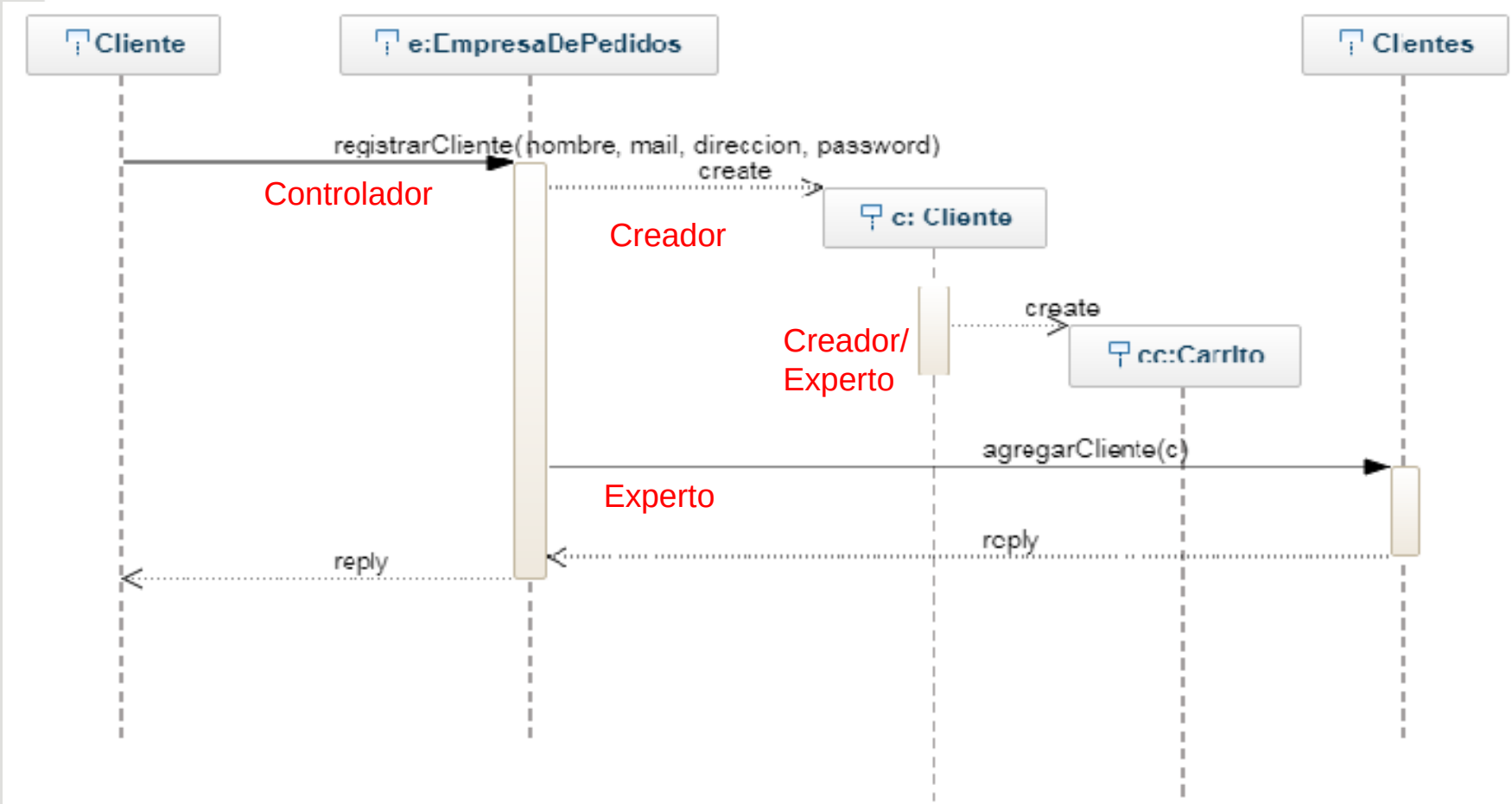
Diagramas de interacción- Ejemplo

Crear una nueva compra.

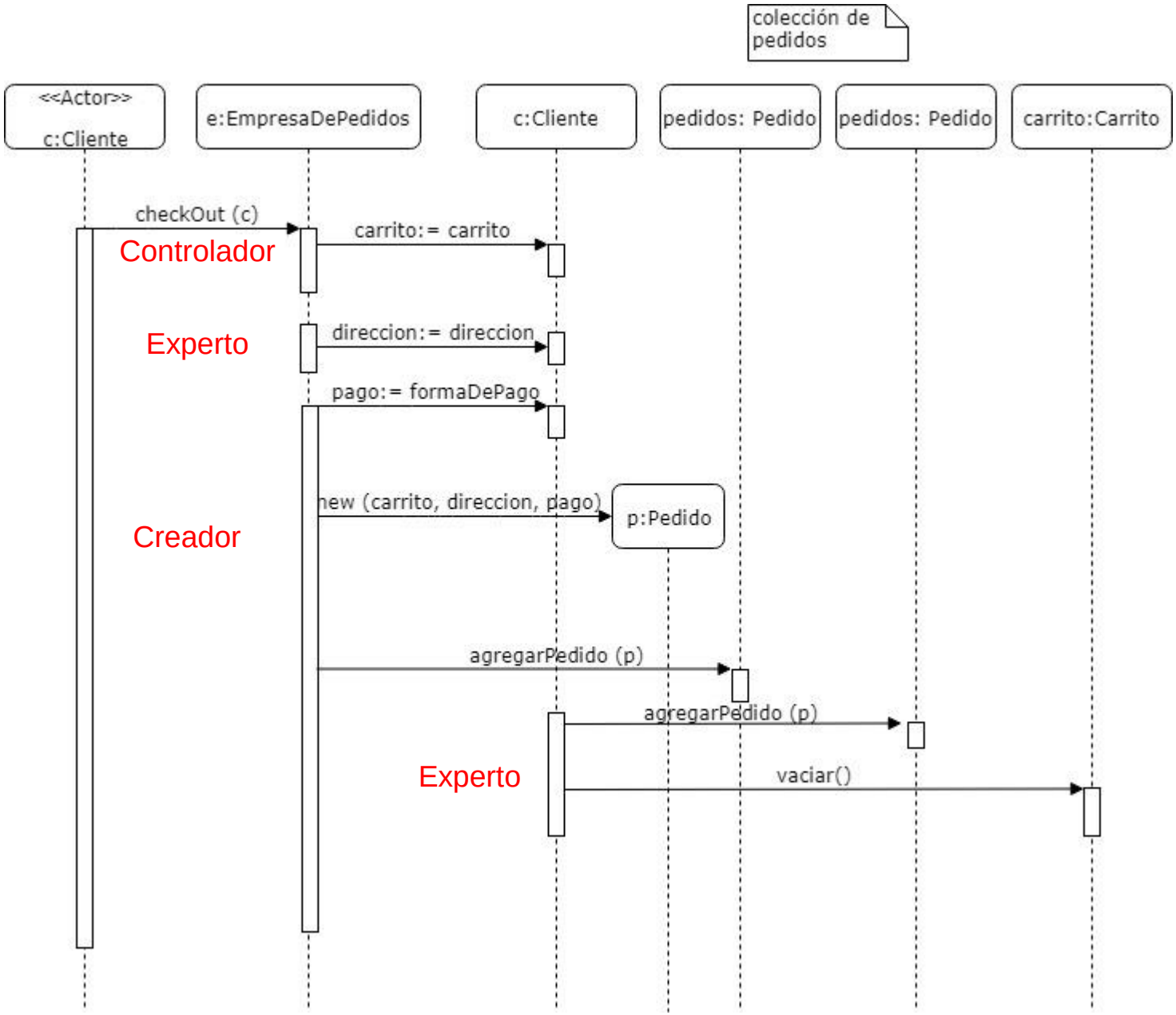
… Aplicando más HAR



Otro Ejemplo: Operación registrar cliente en Gloovo



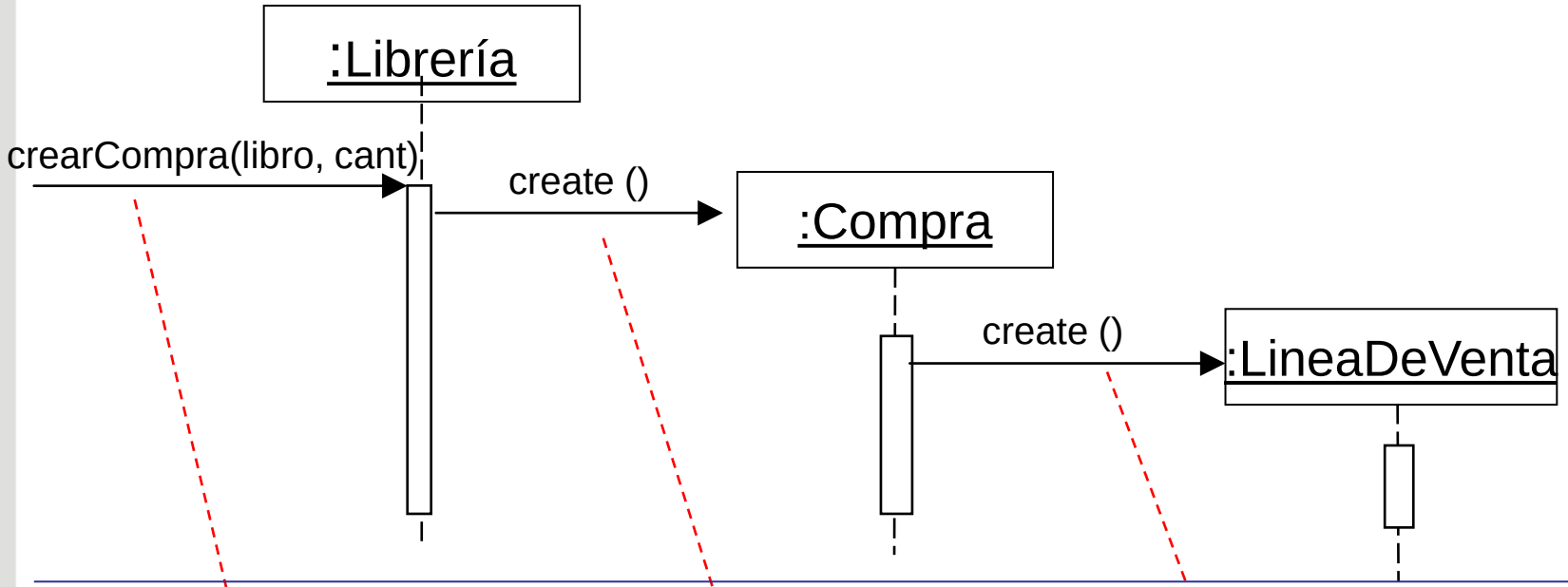
Ejemplo: checkout cliente en Gloovo

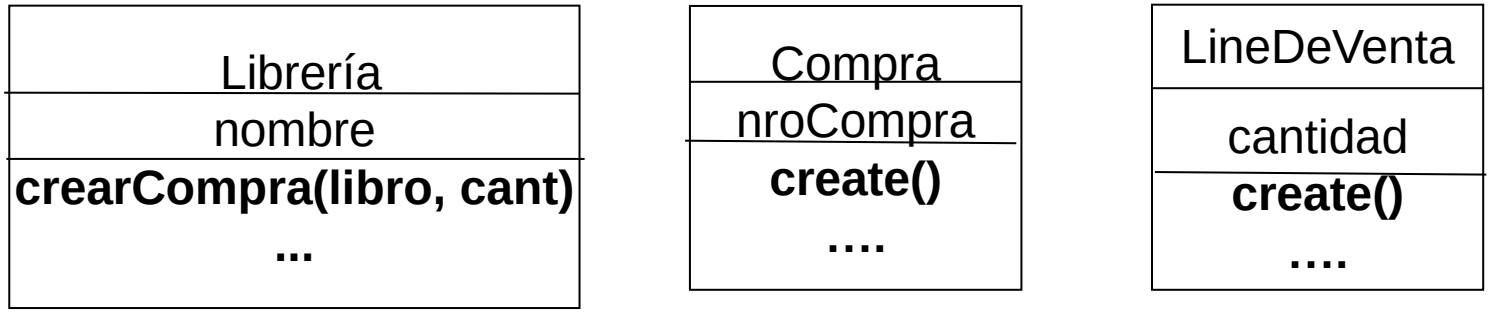


Creación de los diagramas de clases de diseño

- Identificar las clases que participan en los diagramas de interacción y en el Modelo del Dominio o Conceptual.
- Graficarlas en un diagrama de clases.
- Colocar los atributos presentes en el Modelo Conceptual.
- ***Agregar nombres de métodos analizando los diagramas de interacción.***
- Agregar tipos y visibilidad de atributos y métodos.
- Agregar las asociaciones necesarias.
- Agregar roles, navegabilidad, nombre y multiplicidad a las asociaciones.

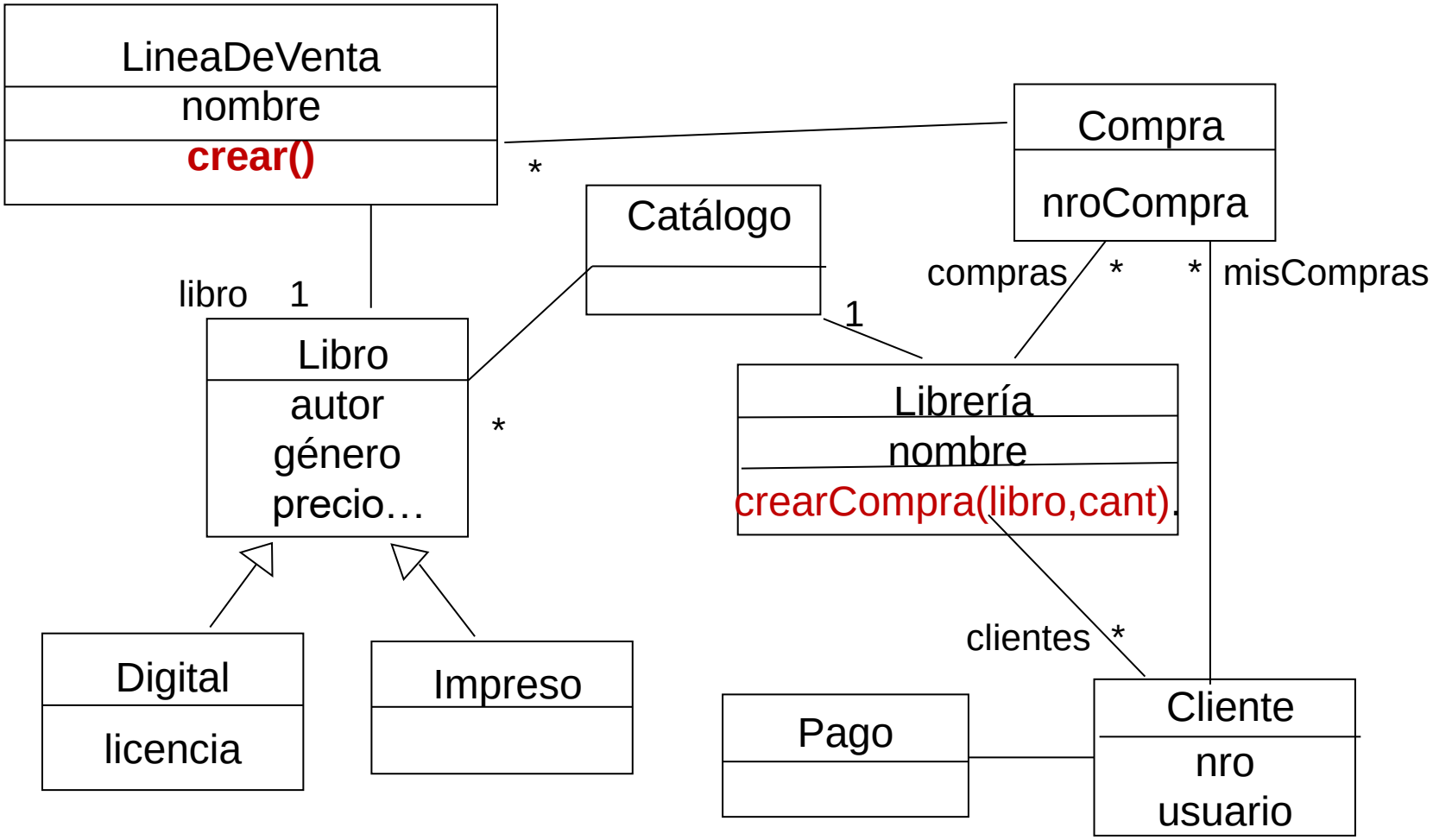
Creación del diagrama de clases de diseño





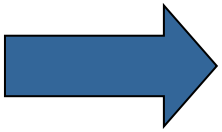
Construyendo el Diagrama de Clases (Parcial)

Agregar al Modelo Conceptual, los mensajes/métodos que surjan en los diagramas de interacción de cada operación.



Transformación de los diseños en código

- Mapear los artefactos de diseño a código orientado a objetos :

Clases		Clases
Atributos		Variables simples
Asociaciones (roles)		Variables de referencia
Métodos		Métodos
Multiobjetos		Collections

Extendiendo el modelo conceptual

Nuevos conceptos pueden ser identificados y agregados al modelo.

Supongamos, en el enunciado que:

- El sitio sólo permite hacer el pago con tarjeta de crédito (no hay lugar físico)

- El sistema suma un servicio de Autorización de Pago con tarjeta de crédito

Entonces:

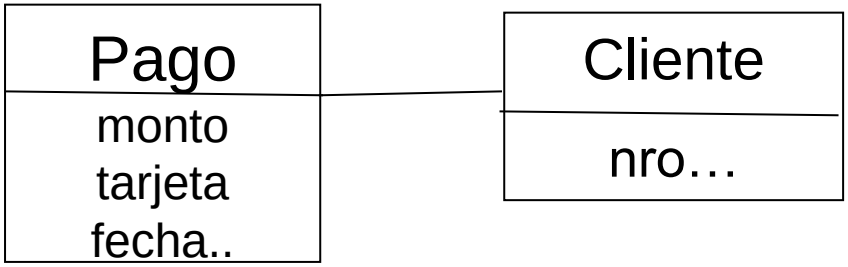
-Agregamos el atributo **tarjeta** al Pago

- Agregamos la clase ServicioAutorización...

Extendiendo el modelo conceptual

El sitio sólo permite hacer el pago con tarjeta de crédito (no hay lugar físico)

-Agregamos el atributo **tarjeta** al Pago, para registrar el tipo de tarjeta con que se realiza el pago



Ahora, supongamos que el sitio de Venta de Libros por Internet considera hacer:

- Si el pago se hace con Tarjeta Clásica, el único beneficio es pagar a precio de lista en 3 cuotas sin interés.
- una bonificación del 3% en el Pago a los clientes que pagan con tarjeta de Crédito Oro
- una bonificación del 5% si el monto supera los 1500 pesos, a los que pagan con tarjeta Platino.

Extendiendo el Modelo Conceptual

Descubriendo nuevas clases para evitar preguntar por el tipo o valor de un atributo.

Para resolver el planteo que depende del atributo **tarjeta** de la Clase **Pago**, debemos consultar por su valor (**uso de if , sentencias Case..**):

if tarjeta = `oro`…….

if tarjeta = `platino`…….

if tarjeta = `clásica`…..

Es esta una buena práctica en Orientación a Objetos?

Como lo resolveríamos usando conceptos Orientados a Objetos?

Extendiendo el Modelo Conceptual

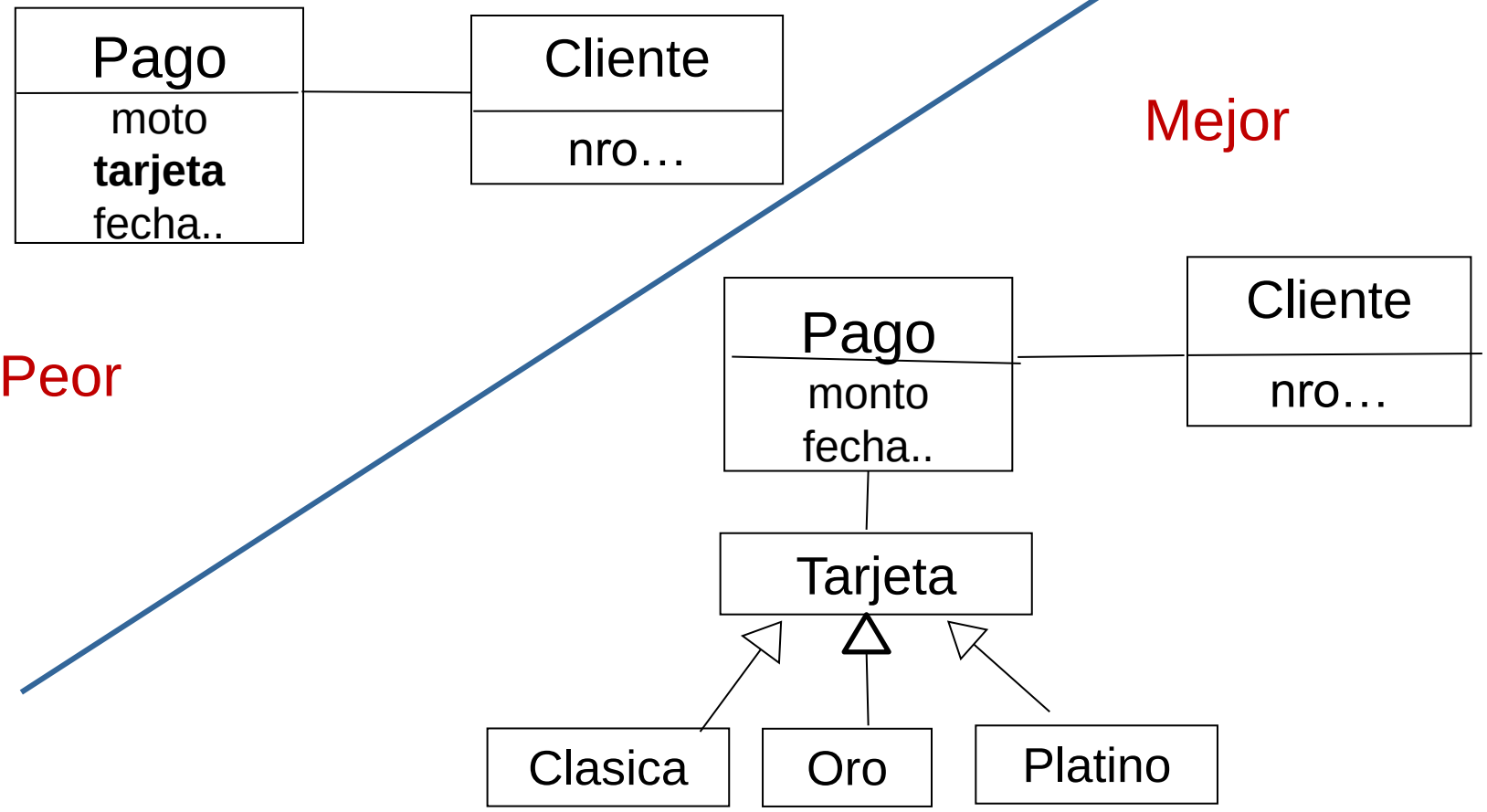
Descubriendo nuevas clases para evitar preguntar por el tipo o valor de un atributo. Polimorfismo

Debemos poder delegar en el **tipo de tarjeta** el cálculo de la **bonificación** que corresponde a cada tipo.

Es decir, debemos agregar una **nueva clase y subclases** que resuelvan, aplicando **polimorfismo**, cada cálculo de la bonificación.

Extendiendo el Modelo Conceptual

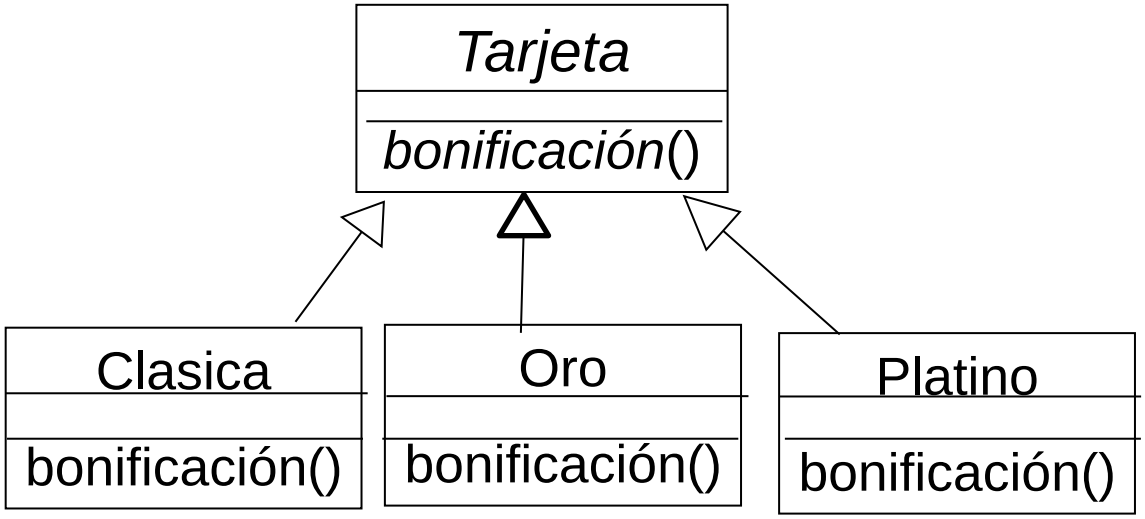
Descubriendo nuevas clases para evitar preguntar por el tipo o valor de un atributo. Polimorfismo



Extendiendo el Modelo Conceptual

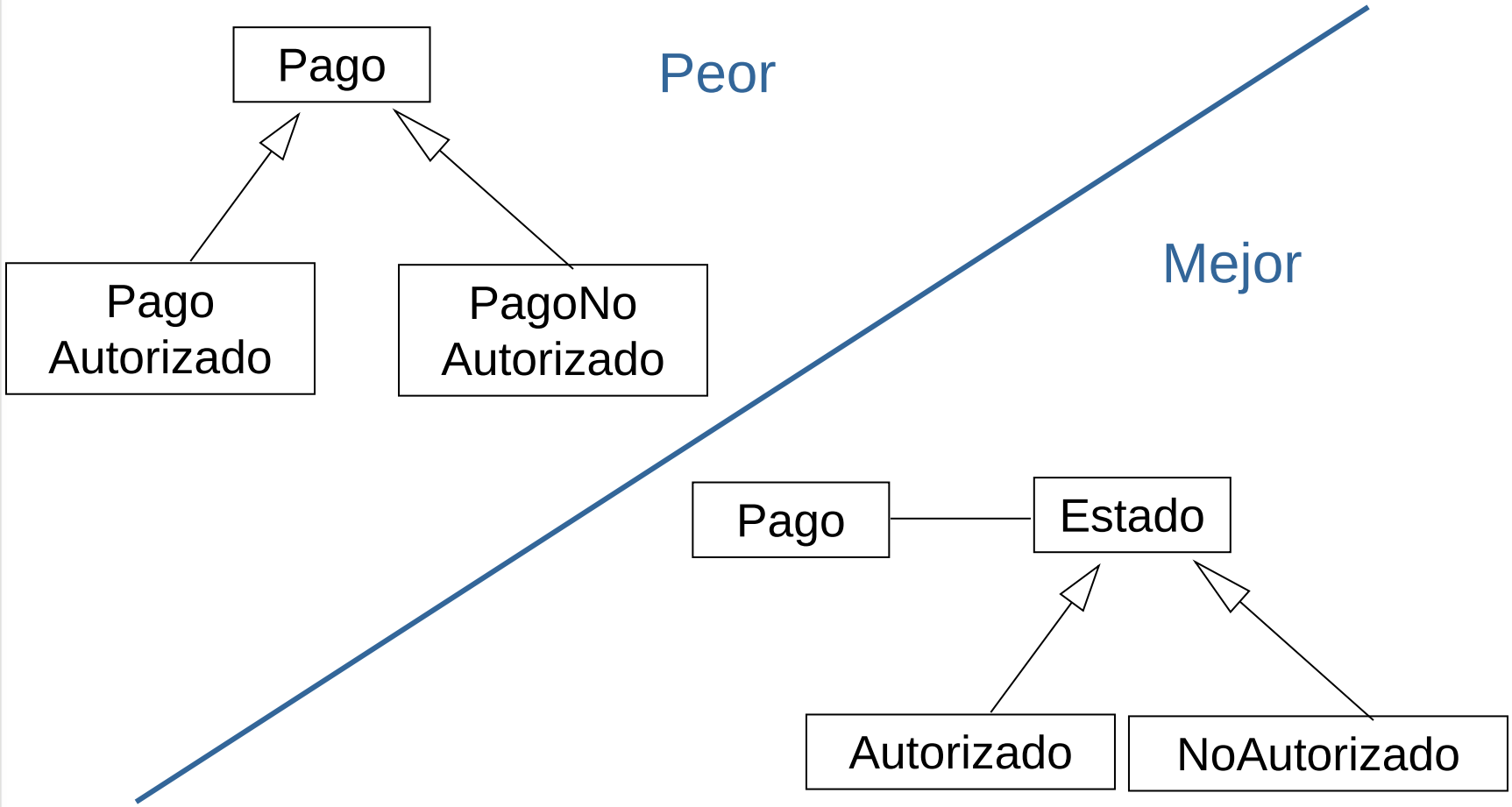
Descubriendo nuevas clases para evitar preguntar por el tipo o valor de un atributo.

Dónde aplico Polimorfismo?



Descubriendo jerarquías

¿Debería clasificar objetos por su estado, o clasificar estados?

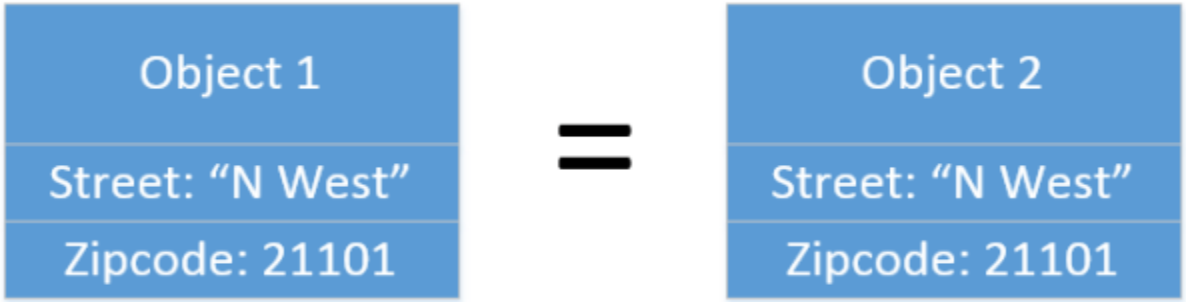


Clases Conceptuales: “Entity vs. Value Object”

- *Las Entidades* o *clases* del dominio de mi problema tienen un identificador, son modificables y comparables por *Identidad*.

Value Object:

- Son comparables por contenido (igualdad estructural), no tienen identificador.



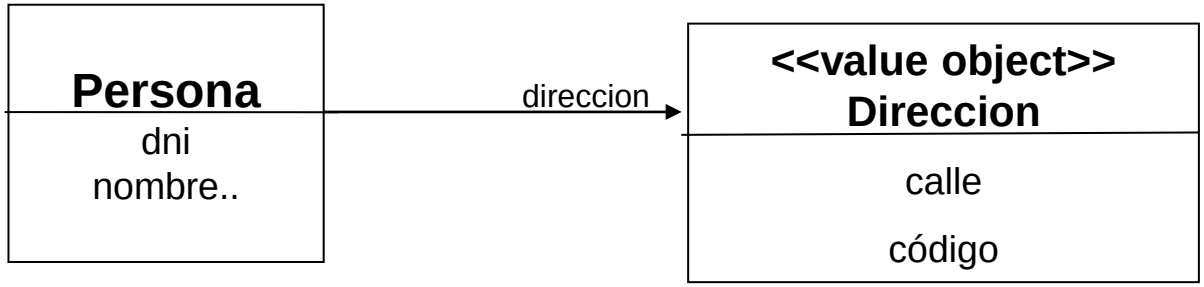
- No viven por si mismos, necesitan una entidad base, son intercambiables (un billete de 100 Pesos AR lo puedo cambiar por otro). Persisten adjunto a su base, no separadamente.
- Inmutables (No le defino *setters*).

Clases Conceptuales: “Entity vs. Value Object”

- Si el *Value Object* no es *inmutable*, entonces NO es un Value Object !
- Como identificar un *Value Object* en el modelo?

Cuando necesitamos por ej. modelar Moneda, Fecha, Dirección, que puedan tener cierto comportamiento (getters..)

Como representar “Value Objects” en el modelo?



<<Value Object>> delante del nombre

Agregando Heurísticas para Asignación de responsabilidades (HAR)

Polimorfismo

Descripción: cuando el comportamiento varía según el tipo, asigne la responsabilidad a los tipos/las clases para las que varía el comportamiento.

Ejemplo: El sistema venta de libros debe soportar distintas bonificaciones de pago con tarjeta de crédito.

(ya visto previamente)

… Como la bonificación del pago varía según el tipo de tarjeta, deberíamos asignarle la responsabilidad de la bonficación a los distintos tipos de tarjeta.

- Nos permite sustituir objetos que tienen idéntica interfaz.

“No hables con extraños”

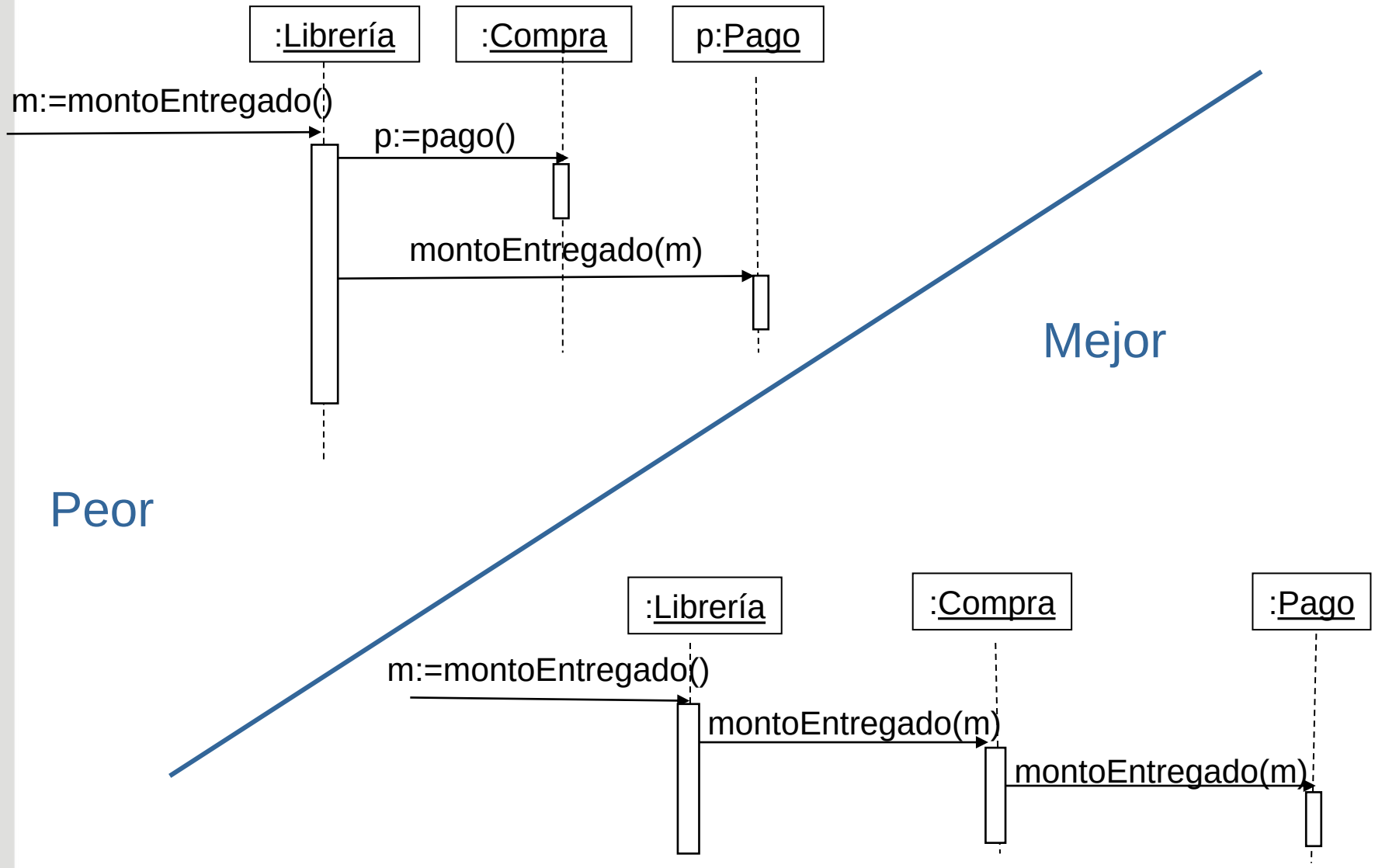
Descripción: Evite diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños).

Dentro de un método sólo pueden enviarse mensajes a objetos conocidos:

- Self/this
- un parámetro del método
- un objeto que esté asociado a self/this
- un miembro de una colección que sea atributo de self/this
- un objeto creado dentro del método

Los demás objetos son extraños (strangers)

“No hables con extraños”



Heurísticas para Diseño “ágil” Orientado a Objetos

(Principios S O L I D)

Principios para Diseño "ágil" Orientado a Objetos

Principios S O L I D

Relacionados a las **HAR**, para un buen estilo de DOO.
Promueven Alta Cohesión y Bajo Acoplamiento.

Robert C. Martin (2014) Agile Software Development, Principles, Patterns, and Practices. Pearson New International Edition (Capítulos 8 al 12)

S SRP: The Single-Responsibility Principle

Principio de Responsabilidad única. Una clase debería cambiar por una sola razón.

Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión).

O OCP: The Open-Closed Principle

Entidades de software (clases, módulos, funciones, etc.) deberían ser "abiertas" para extensión, y "cerradas" para modificación.

Abierto a extensión: ser capaz de añadir nuevas funcionalidades.

Cerrado a modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente.

Principios para Diseño “ágil” Orientado a Objetos

L LSP: The Liskov Substitution Principle

Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Es decir, que si el programa utiliza una clase (clase A), y ésta es extendida (clases B, C, D, etc…) el programa tiene que poder utilizar cualquiera de sus subclases y seguir siendo válido. Uso correcto de herencia (Is-a) y polimorfismo.

I ISP: The Interface-Segregation Principle

Las clases que tienen interfaces “voluminosas” son clases cuyas interfaces no son cohesivas.

Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deben estar forzadas a incluir métodos que no va a utilizar.

Principios para Diseño “ágil” Orientado a Objetos

D DIP: The Dependency-Inversion Principle

a. Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.

b. Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

Módulos de alto nivel: se refieren a los objetos que definen qué es y qué hace el sistema.
Módulos de bajo nivel: no están directamente relacionados con la lógica de negocio del programa (no definen el dominio). Por ejemplo, el mecanismo de persistencia o el acceso a red .

Abstracciones: se refieren a protocolos (o interfaces) o clases abstractas.

Detalles: son las implementaciones concretas, (cuál mecanismo de persistencia, etc).

Ser capaz de «*invertir*» una dependencia es lo mismo que ser capaz de «*intercambiar*» una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.

Reuso de Código

Herencia vs. Composición

Herencia de Clases

- Herencia total: debo conocer todo el código que se hereda -> Reutilización de Caja Blanca
- Usualmente debemos redefinir o anular métodos heredados
- Los cambios en la superclase se propagan automáticamente a las subclases
- Herencia de Estructura vs. Herencia de comportamiento
- Es útil para extender la funcionalidad del dominio de aplicación

Composición de Objetos

- Los objetos se componen en forma Dinámica -> Reutilización de Caja Negra
- Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

Herencia vs. Composición de Objetos

- Las clases y los objetos creados mediante herencia están *estrechamente acoplados* ya que cambiar algo en la superclase afecta directamente a la/las subclases.

- Las clases y los objetos creados a través de la composición están *débilmente acoplados*, lo que significa que se pueden cambiar más fácilmente los componentes sin afectar el objeto contenedor.

Cómo hacer un mal uso de la herencia

```
import java.util.ArrayList;

 public class Stack<T> extends ArrayList<T> {

     public void push(T object) {
         this.add(object);
     }
     public T pop() {
         return this.remove(this.size() - 1);
     }
     public boolean empty() {
         return this.size() == 0;
     }
     public T peek() {
         return this.get(this.size() - 1);
     }
}
```

Cómo hacer un mal uso de la herencia

- Esta clase ***Stack*** funcionará como una pila, pero su interfaz es *voluminosa*; está formada por mensajes que hay que anular o redefinir!!

- La interfaz pública de esta clase no es solo push y pop, (esperable para una Pila), también incluye
- add en cualquier posición por índice,
- remove de una posición a otra,
- clear y muchos otros mensajes heredados de ArrayList que son inapropiados para una Pila.

Cómo hacer un mal uso de la herencia

El ejemplo tiene errores de diseño

- uno semántico:

"una pila es una ArrayList" no es cierto; Stack no es un subtipo adecuado de ArrayList (No cumple con Is-a). Se supone que una pila aplica el último en entrar, primero en salir, una restricción que se satisface con la interfaz push/pop, pero que no se cumple con la interfaz de ArrayList que es mucho más extensa.

- Otro mecánico:

heredar de ArrayList viola el encapsulaminto; usar ArrayList para contener la colección de objetos de la pila es una opción de implementación que puede y debe ocultarse.

Composición, no Herencia

```
import java.util.ArrayList;

 public class Stack<T> {
     private ArrayList elementos <T>

     public void push(T object) {
        elementos.add(object);
    }
     public T pop() {
       return elementos.remove(elementos.size() - 1);
    }
     public boolean empty() {
        return elementos.size() == 0;
    }
     public T peek() {
        return elementos.get(elementos.size() - 1);
    }
}
```

Cómo solucionar un mal uso de la herencia

Composición, no herencia…

- Mecánicamente, heredar de ArrayList no cumple con el encapsulaminto; en cambio,

- **componer con ArrayList para contener la colección de objetos de la pila es una opción de implementación que permite ocultarla públicamente.**

- En este caso, en vez de heredar, el uso o composición permite reuso y mantiene el encapsulamiento!!