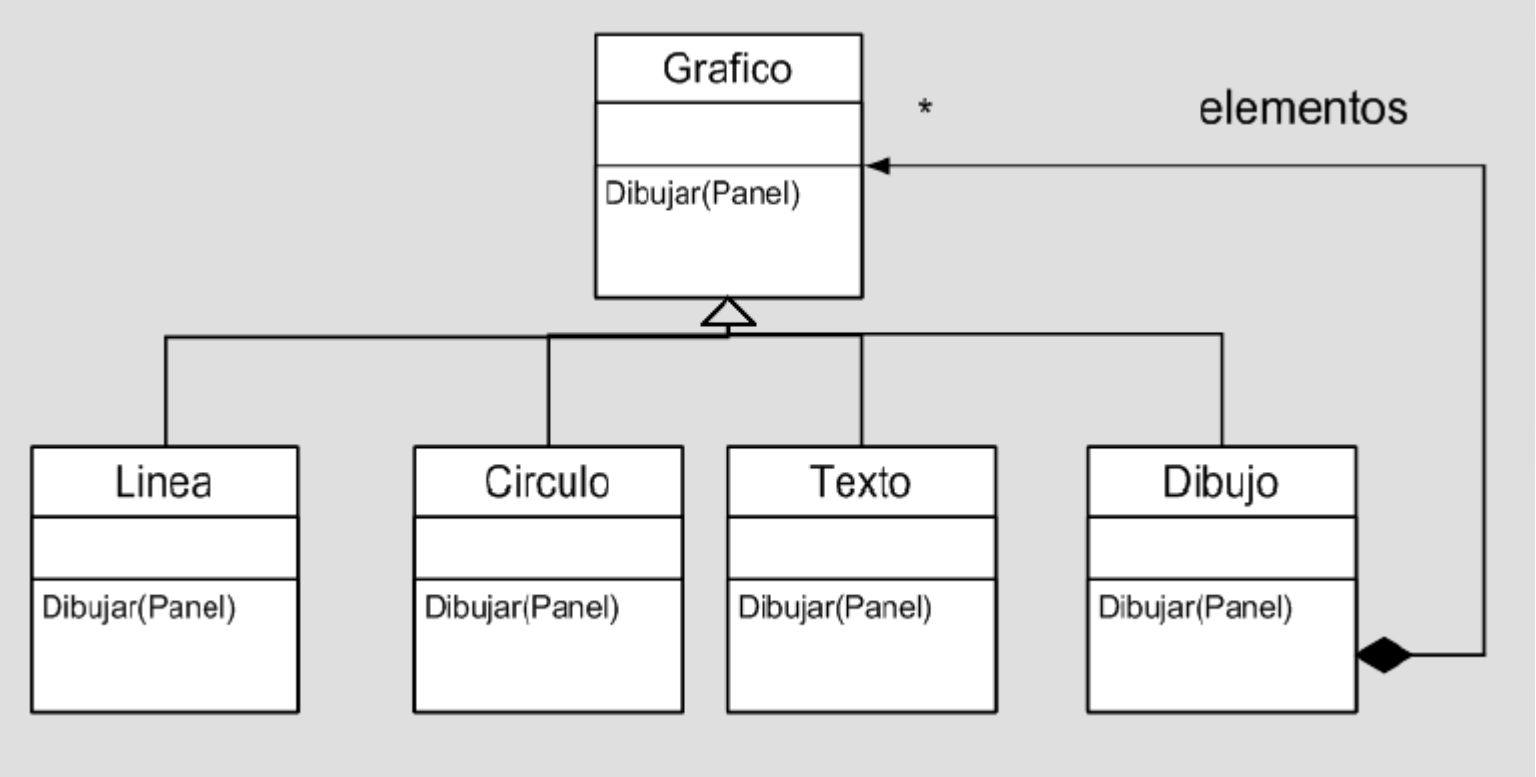
Metodologías de Programación I

Introducción a Objetos

Patrón Adapter (cont clase anterior)

Estilos de programación (OO)

Recordemos el Editor gráfico



Editor gráfico

Supongamos que queremos también mostrar fotos en el editor

- Contamos con una jerarquía Picture que maneja la representación de fotos. Picture tiene subclases para cada formato de imagen. Las clases en la jerarquía de

Picture entienden el mensaje

show(Panel panel, integer compression)

- ¿Cómo hacemos para que el editor pueda seguir enviando el mensaje dibujar() a sus componentes?

Patrón Adapter

- **Objetivo:** Convertir la protocolo de una clase en otra, que es la que el objeto cliente espera.

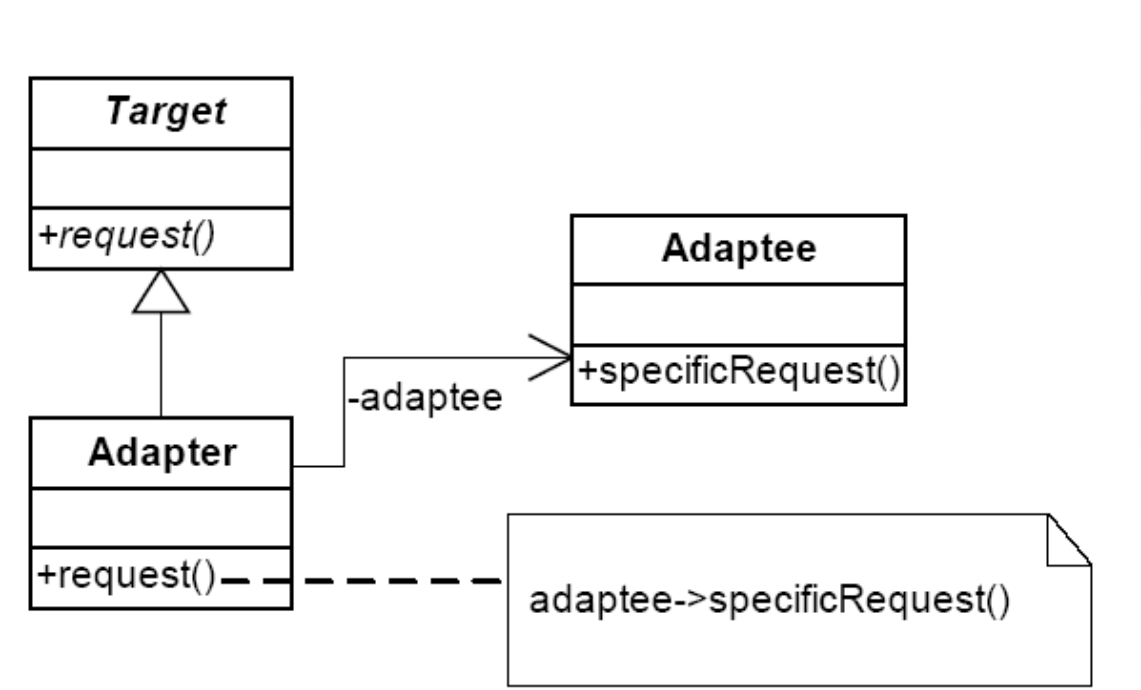
- **Problema:** Muchas veces, clases que fueron pensadas para ser reutilizadas no pueden aprovecharse porque su protocolo no es compatible con el protocolo específico del dominio de aplicación con el que se está trabajando.

- Deseamos utilizar una clase existente, cuyo protocolo no encaja con la que necesitamos.

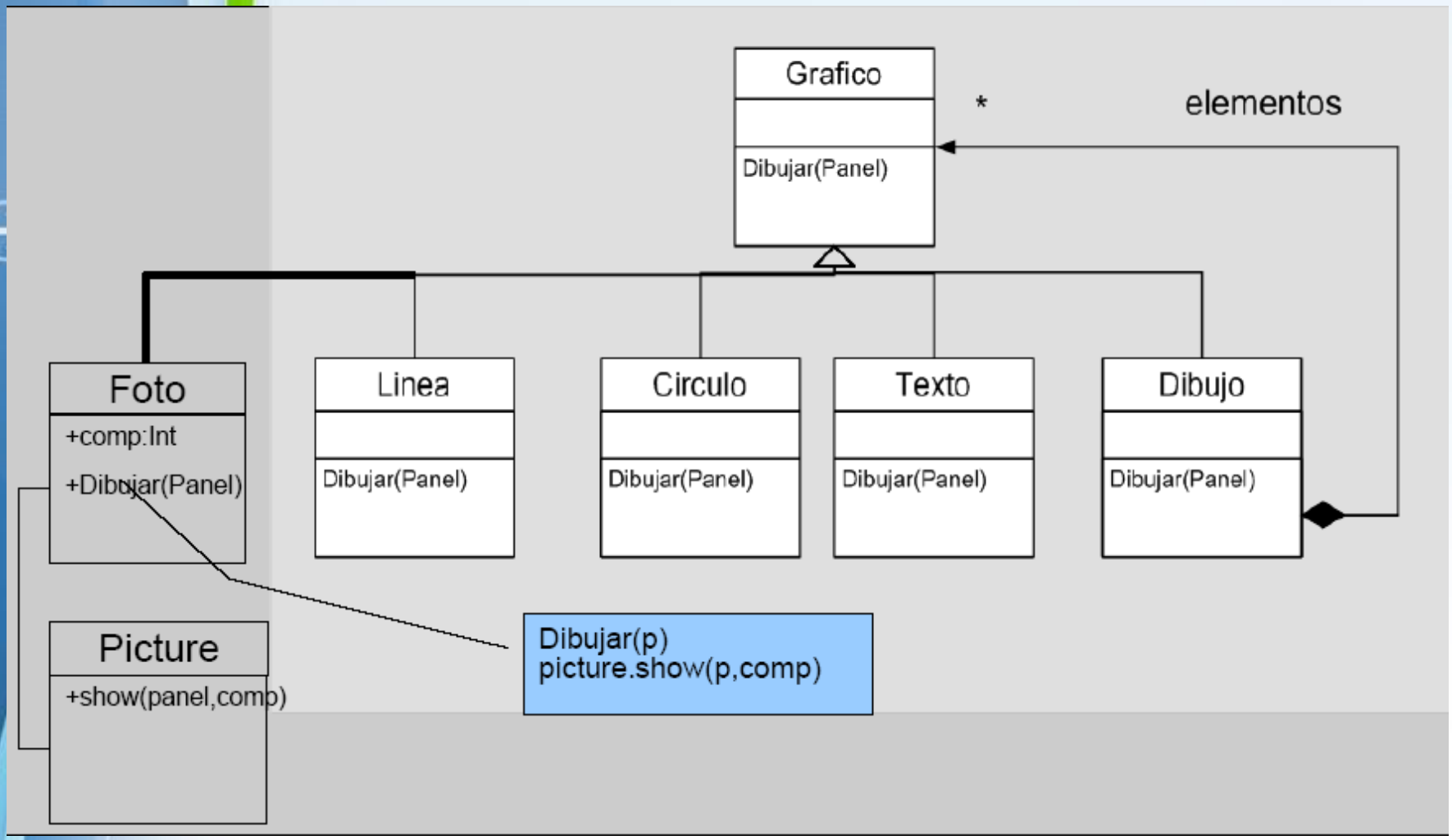
- Deseamos crear una clase que puede llegar a cooperar con otros objetos cuyo protocolo no podemos predecir de antemano.

Patrón Adapter

- **Solución:** Crear una clase que se encargue de “transformar” los nombres de los mensajes.



Patrón Adapter

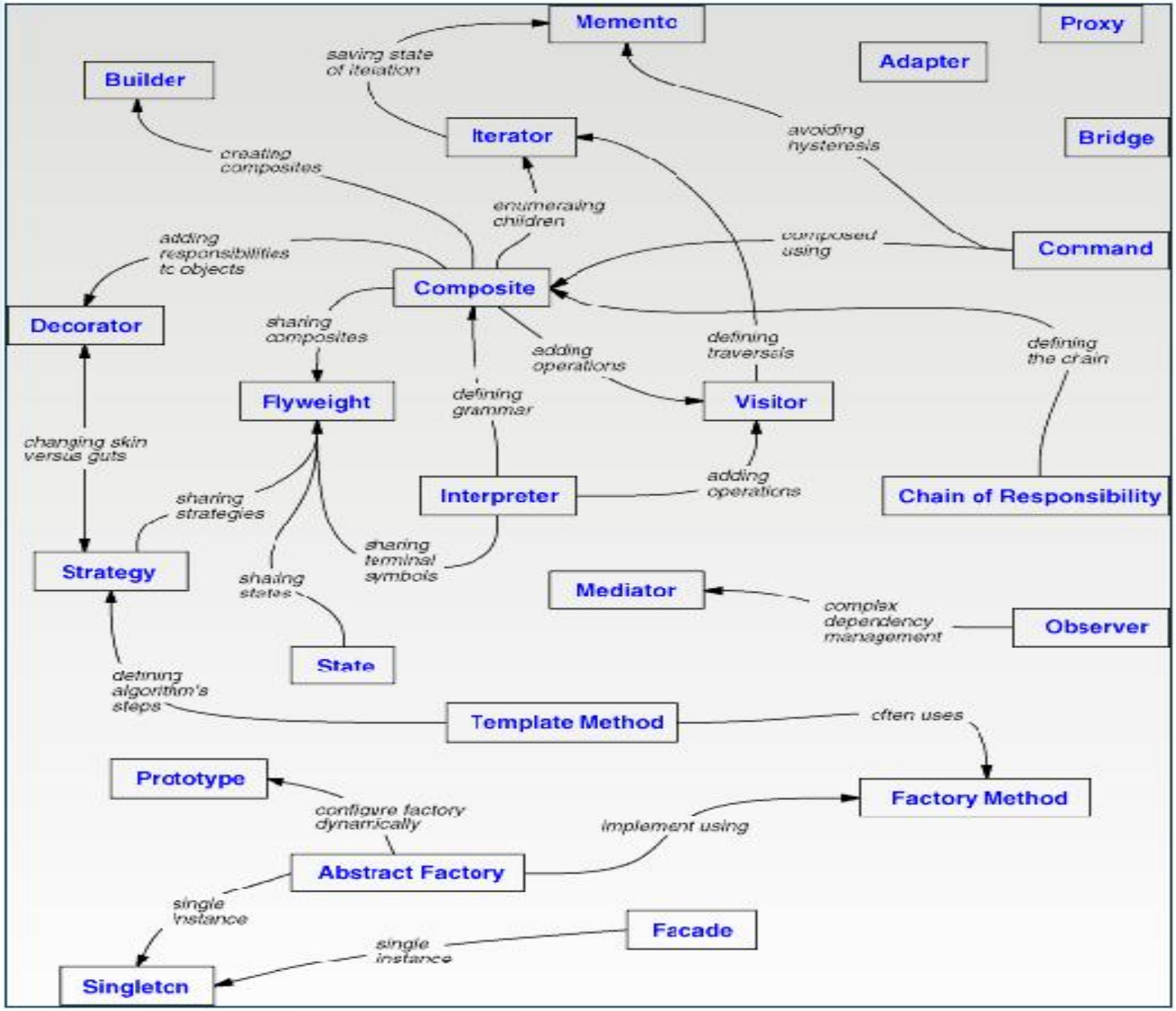


Patrón Adapter

Consecuencias:

- El Adapter puede sobre-escribir algún método del “adaptado”.
- Solo introduce un objeto como indirección.
- Adaptar toda una jerarquía requiere trabajo extra para inicializar las referencias.
- Puede ser inconveniente si el “adapter” implementademasiado comportamiento.

Interrelación entre los patrones



Tecnica Double Dispatch

"Dispatching" es el proceso que determina el método a ejecutar cuando un objeto recibe un mensaje. El método que se ejecuta cuando un objeto recibe un mensaje depende exclusivamente de la clase del objeto receptor.

Este comportamiento es suficiente en la mayoría de los casos, sin embargo en algunos casos sería deseable que el método elegido dependiera de la clase del receptor y de la clase de alguno de los parámetros del mensaje.

Técnica Double Dispatch

- **Objetivo:** Seleccionar un comportamiento que depende de dos clases de objetos.

- **Problema:** En algunos casos es necesario escribir código cuyo comportamiento dependa de la clase del parámetro de un método.

En la programación procedural este tipo de problemas se resuelve usando un *case*. Hacer esto en objetos (basados en la clase del parámetro) resulta en código que no escala.

En el paradigma de objetos tenemos la ventaja del *polimorfismo*.

Técnica Double Dispatch

Solución:

- Escribir un método secundario por cada clase. Su nombre será similar al principal, con el agregado del nombre de la clase.

- Al llamarse al método principal, éste delegará en el parámetro llamando al método secundario asociado con su clase.

- En el caso de ser necesario, se puede pasar al objeto receptor del mensaje primario como parámetro del mensaje secundario.

- Double dispatch significa que la operación que se ejecuta depende del tipo de 2 receptores

Técnica Double Dispatch

Consecuencias:

- Double Dispatch elimina las sentencias del estilo *case* basadas en la clase del parámetro.

- Esto hace que el código escale y pueda ser mantenido en forma relativamente sencilla.

- A pesar de esto no es una panacea; al agregar una nueva clase, según en qué jerarquía se agregue, puede ser que sea necesario agregar un método nuevo en todas las clases de la otra jerarquía. De todas maneras, ésta última desventaja es preferible a la sentencia *case*.

Double Dispatch no es overloading(Sobrecarga)

Double Dispatch != Overloading (sobrecarga).

- Overloading permite que la función invocada dependa del tipo de parámetros y del tipo del receptor.

- Overloading se chequea en tiempo de compilación. No considera el tipo de los objetos sino de las variables.

- DoubleDispatch asegura (por medio de envío de mensajes) que el objeto apropiado es referenciado.
- En C++ y Java es necesario utilizar funciones virtuales.

Ejemplo

Un ejemplo de esta relación se da entre los Numbers en las operaciones aritméticas.

Dado que los Numbers se representan internamente de forma distinta, no se puede definir un método que sirva para todas las combinaciones de Numbers.

Por ejemplo, cada caso de suma entre un Integer y un Float se necesita un método específico.

Ejemplo

En concreto la solución consiste en:

En el método inicial, enviar un mensaje al argumento.

El nombre del mensaje a enviar debe incluir el nombre de la clase del receptor (o alguna otra característica que permita desambiguar al receptor). El receptor del mensaje original se pasa como parámetro de este nuevo mensaje.

Al recibirse este mensaje no hay ambigüedad, ya que la clase del receptor se resuelve por el mecanismo de dispatch, y la clase del argumento forma parte del nombre del mensaje.

Ejemplo

La solución para el caso de los Numbers y la suma sería:

- Las clases Integer and Float hacen un “Double Dispatch” al argumento (que puede ser tanto un Float como un Integer):

```
Integer>>+ aNumber
^aNumber addInteger: self

Float>>+ aNumber
^aNumber addFloat: self
```

Ejemplo

Tanto Integer como Float deben implementar ambas variantes de los métodos add. Los casos Integer-Integer y Float-Float se manejan como primitives de Smalltalk:

```
Integer>>addInteger: anInteger
<primitive: 1>
Float>>addFloat: aFloat
<primitive: 2>
```

Cuando se tiene un número de cada clase, se convierte el Integer a Float :

```
Integer>>addFloat: aFloat
^self asFloat addFloat: aFloat

Float>>addInteger: anInteger
^self addFloat: anInteger asFloat
```

Estilos de Programación

Nombres

Seleccionar nombres que clarifiquen el propósito del objeto. Java permite cualquier longitud.

Nombres descriptivos requieren de pocos comentarios explicativos.

Una traducción única simplifica la comunicación humana y no permite confusión.

Ej: marcar los nombres que son descriptivos:

```
Example
timeOfDay
tod

milliseconds
millis

editMenu
eMenu
```

Estilos de Programación

Uso de Mayúsculas y Minúsculas.

Con Mayúscula y Minúscula también se distingue el alcance de las Variables.

Con Mayúscula: variables globales, variables clases y pool dictonaries. Estas variables son globales a todos los métodos definidos en el alcance de variable

Con minúscula: los métodos de clase e instancia, los parámetros de los métodos, variables temporales y variables de instancias.

Estilos de Programación

Example

Behavior	“class”
Display	“global variable”
CharacterConstants	“pool dictionary”
CurrentUser	“class variable in a class called User”

```
address
currentTime
beforeNoon
isLunchReady

readyForNextItem
readyfornextitem
```

Estilos de Programación

Nombre de Clases

El nombre de la clase es extremadamente importante, es el elemento central en cualquier programa orientado a objetos.

Utilizar nombres lo menos restrictivos posibles, pensando en el reuso de la clase.

El nombre de la clase debe empezar con mayúscula.

- Collection
- GraphicalClassBrowser
- EmployeePensionPaymentsHistory

En el 2do y 3er caso el nombre de la clase me permite describir la clase. El nombre de la clase se utiliza para indicar el propósito o intención de la clase.

Estilos de Programación

El nombre de la clase debe ser descriptivo,

nombre tales como MyClass o ProjectClass1

No son descriptivos.

	Example	
✔	ProblemReport	
✘	Application	“too generic”
✔	TreeWalker	
✘	TreeWalkerForBinaryTrees	“too specific”

Estilos de Programación

Los nombres de las clases no deben hacer referencia a la estructura con la que se implementa la solución

Example

"A database for Problem Reports that uses a Dictionary. There is no need to tell the user the implementation."

✔ `PRDatabase`

✘ `PRDictionary`

"A proper name that is stored as a String."

✔ `ProperName`

✘ `ProperNameString`

"This class is not implemented with a Set; it is a specialized Set."

✔ `SortedSet`

Estilos de Programación

Los nombres de las clases deben ser palabras del lenguaje natural.

	Example
✔	`Terminal`
✘	`UserCommunicationsInterface`
✔	`RemoteControl`
✘	`RemControl`
✔	`RandomNumberGenerator`
✘	`NumGen`
✔	`Road`
✘	`AutomotiveTransportMedium`

Estilos de Programación

Nombre de los métodos

Deben siempre comenzar con minúscula. Si está compuesto por más de una palabra, a partir de la segunda debe comenzar con mayúscula.

"Notación camello".

account deposit: 100

account printStatement.

Estilos de Programación

Los nombres de los métodos se deben seleccionar de tal forma que ilustren el propósito del método

Métodos que retornan true o false como resultado: estos métodos usan el verbo is o has concatenado al valor a testear:

isString : en este método se testea si el receptor del mensaje es un objeto de la clase String.

isActive : algún aspecto en el objeto receptor es testeado, se puede testear si un proceso esta o no activo, por lo cual el valor a testear puede formar parte del nombre del método

Estilos de Programación

Example

"A method to test if an object is a String"

✔ `isString`

"A method to test if a Person is hungry"

✔ `aPerson isHungry`

✘ `aPerson hungry`

"A method to check if a Vehicle has four wheels"

✔ `aVehicle hasFourWheels`

✘ `aVehicle fourWheels`

Estilos de Programación

Usar verbos imperativos y frases para métodos que realizan alguna acción.

	Example
✔	Dog sit; lieDown; playDead.
✔	aReadStream peekWord
✘	aReadStream word
✔	aFace lookSurprised
✘	aFace surprised
✔	anAuctionBlock add: itemUpForSale
✔	File openOn: stream
✔	record deleteFieldAt: index

Estilos de Programación

Evitar tipos o nombres de parámetros en el nombre del método si se están usando nombres de parámetros tipados

	Example
✔	`fileSystem at: aKey put: aFile`
✘	`fileSystem atKey: aKey putFile: aFile`
	"for semantic-based parameter names"
✔	`fileSystem atKey: index putFile: pathName`
	"useful when your class has several #at:put: methods"
✔	`fileSystem definitionAt: aKey put: definition`
✔	`aFace changeTo: expression`
✘	`aFace changeExpressionTo: expression`

Estilos de Programación

Usar verbos con una preposición para método que especifican objetos. Y usar la preposición **on** cuando un método opera sobre otro objeto.

	Example
✔	`at: key put: anObject`
✔	`changeField: anInteger to: anObject`
✔	`ReadWriteStream on: aCollection.`
✘	`ReadWriteStream for: aCollection.`
✔	`File openOn: stream`
✘	`File with: stream`
✔	`display: anObject on: aMedium`
✘	`display: anObject using: aMedium`

Lazy Initialization

Puede existir situaciones en las cuales no deseamos que el compilador cree un objeto por defecto, porque por cada referencia se puede incurrir en una perdida de tiempo innecesaria en muchos casos.

La técnica Lazy initialization (inicialización perezosa) crea la instancia justo antes de que se necesite utilizar el objeto.

Esta puede reducir la sobrecarga en situaciones donde el objeto no necesita ser creado cada vez.

Lazy Initialization

Realizamos modificaciones en los métodos de acceso (“getters”)

Ejemplo aplicando LI

```
>>height
      height ifNil: [height := 2.0].
      ^height
```

Ejemplo sin aplicar LI

```
>>Initialize
     height := 2.0
>>height
   ^height
```

Lazy Initialization

Usos frecuentes:

La inicialización de las var de instancia toman mucho tiempo.

Si se consumen mucho recursos.

Siempre que exista la posibilidad que la variable no sea utilizada.

Lazy Initialization

Usos frecuentes:

La inicialización de las var de instancia toman mucho tiempo.

Si se consumen mucho recursos.

Siempre que exista la posibilidad que la variable no sea utilizada.

Lazy Initialization

Ventajas:

- Asegurarse que la var esté iniciada al momento

  de usarla.

- En el caso de objetos que no se usan muy seguido, o muy grandes, así no se desperdicia memoria

Lazy Initialization

Desventajas:

-Se consume tiempo cada vez que se testea si la variable es nula.

-El código de la inicialización se propaga a otro método, por lo cual si ocurre un error de inicialización, puede ser confuso encontrar el lugar donde ocurrió.

Continuamos con UML

Asociaciones

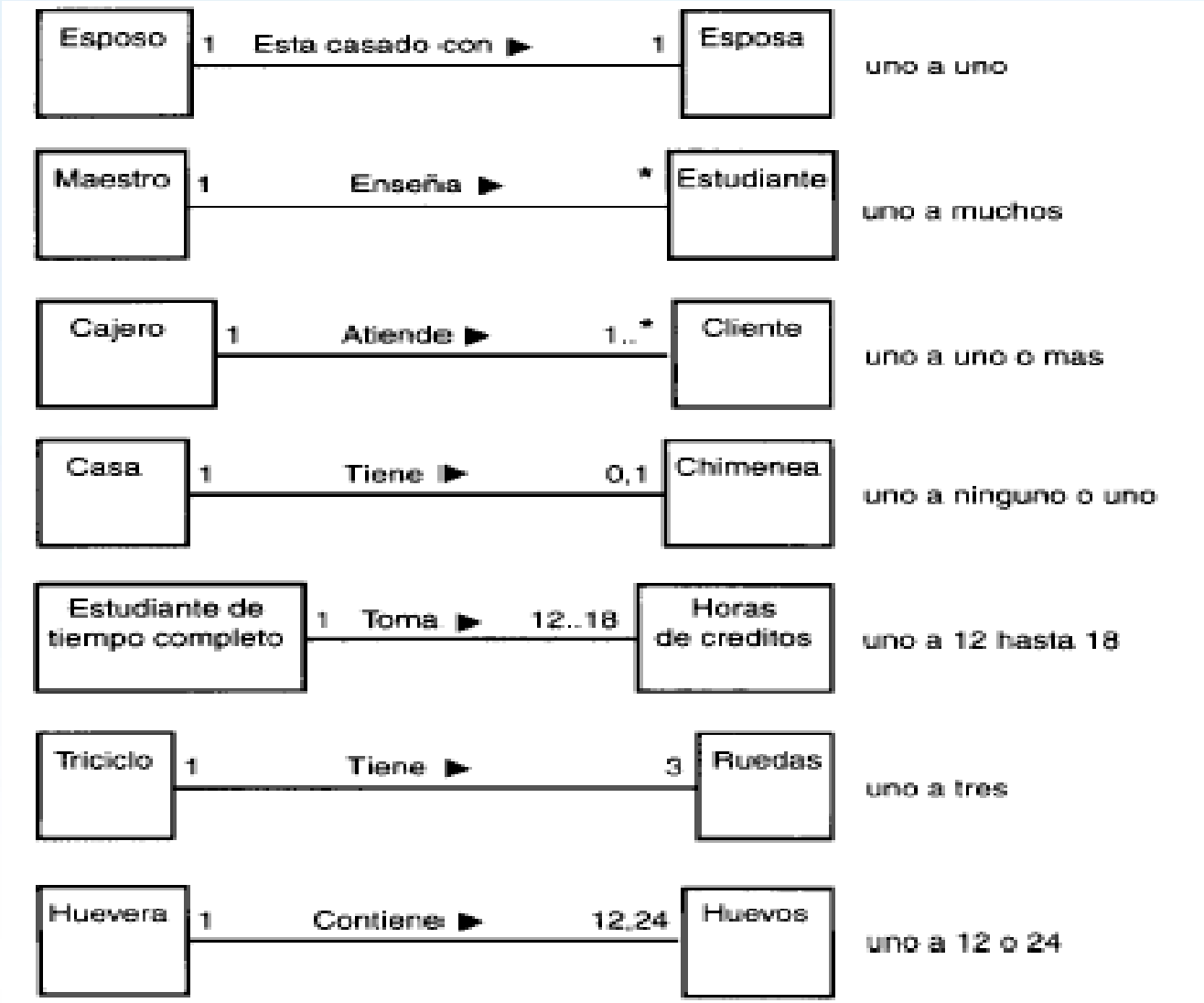
Relación estructural que especifica que los objetos de un tipo están conectados con los objetos de otro.

- Notación
- Nombre de Rol : Papel que una clase representa (a la del otro extremo)
- Multiplicidad : Para cada objeto (del extremo opuesto), cuántos puede tener asociados

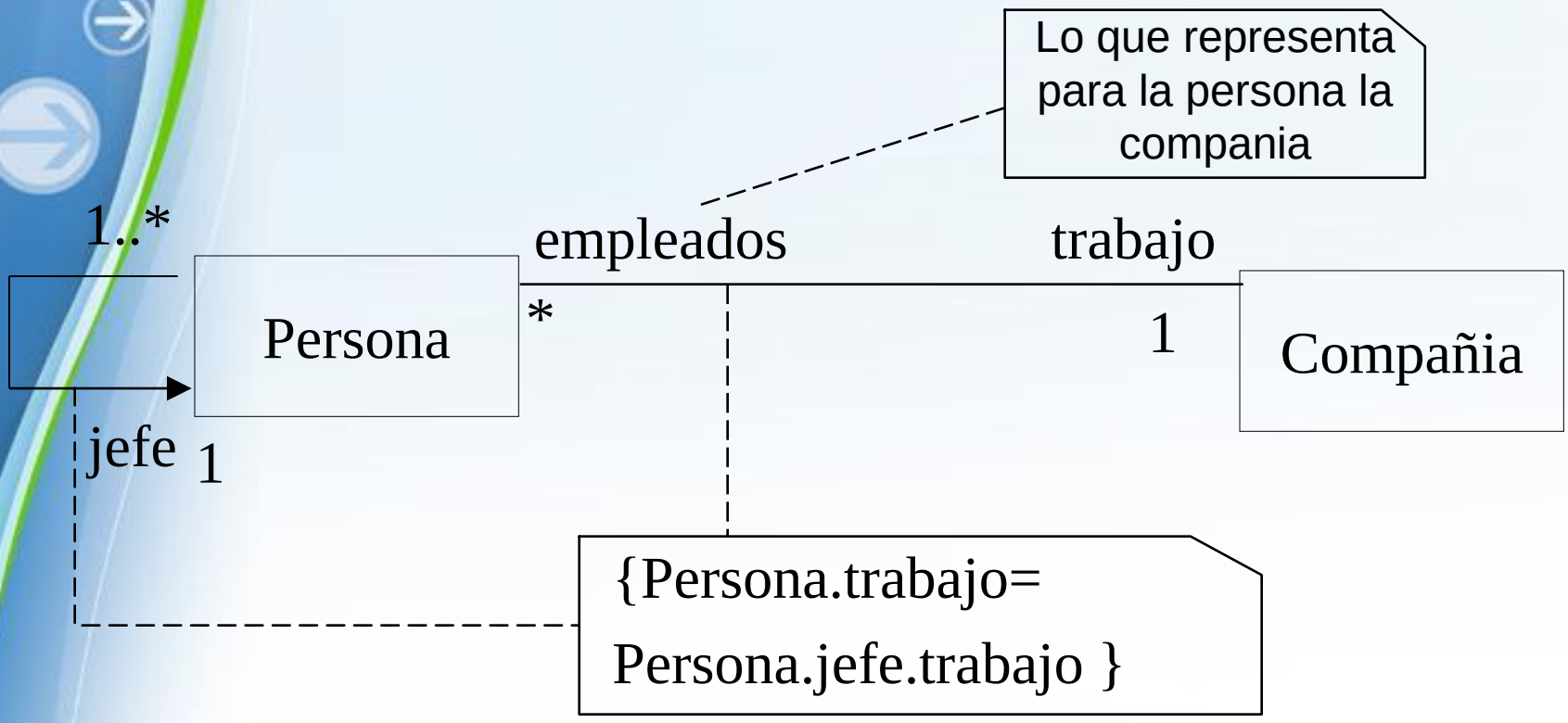
Multiplicidad para asociaciones

Representa la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada

Asociaciones y multiplicidades



Restricciones para Asociaciones



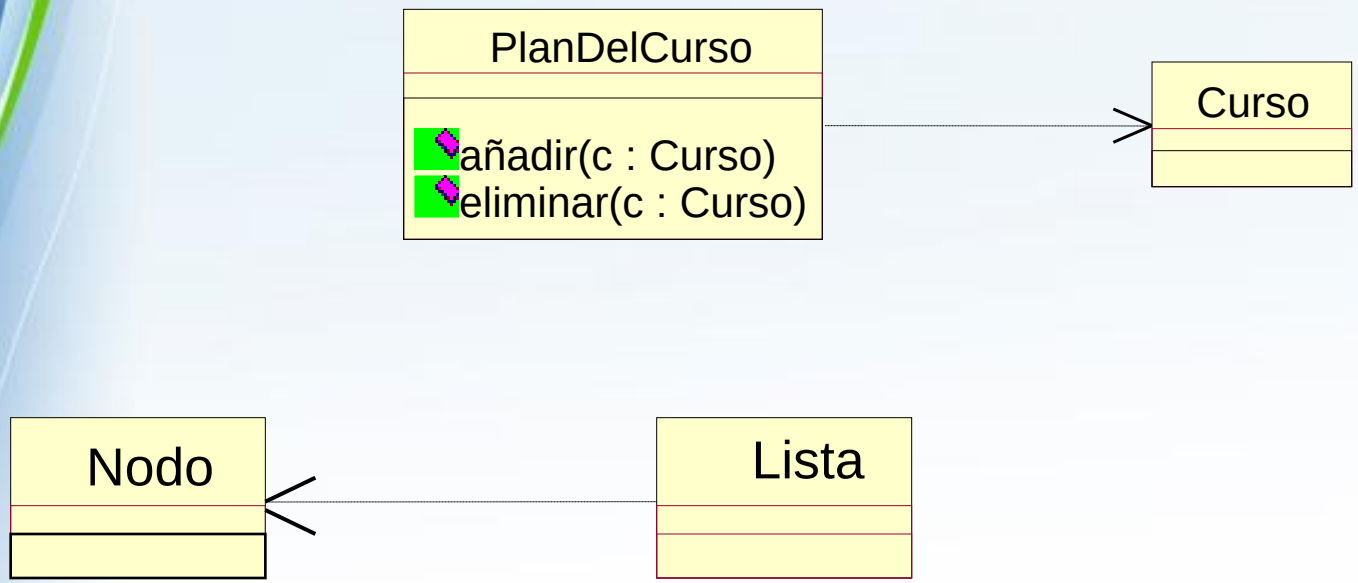
Diagramas de Clases: Relaciones

Dependencia

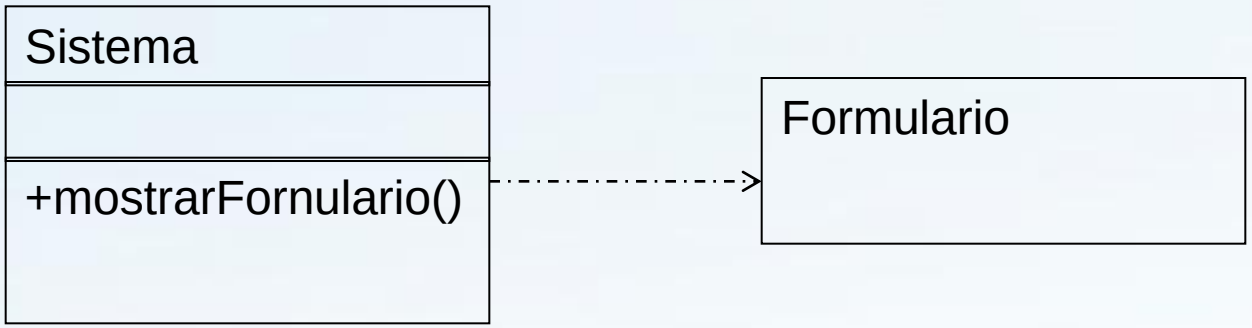
- Indica que una clase utiliza a otra
- Un cambio en la especificación de un elemento afecta a otro
- Representada mediante una línea discontinua con punta de flecha en forma de triángulo sin relleno.

  - - - - - - - - -|>

Diagramas de Clases: Ejemplo de Dependencia



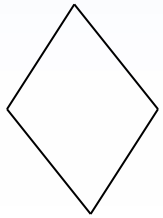
Ejemplo de Dependencia



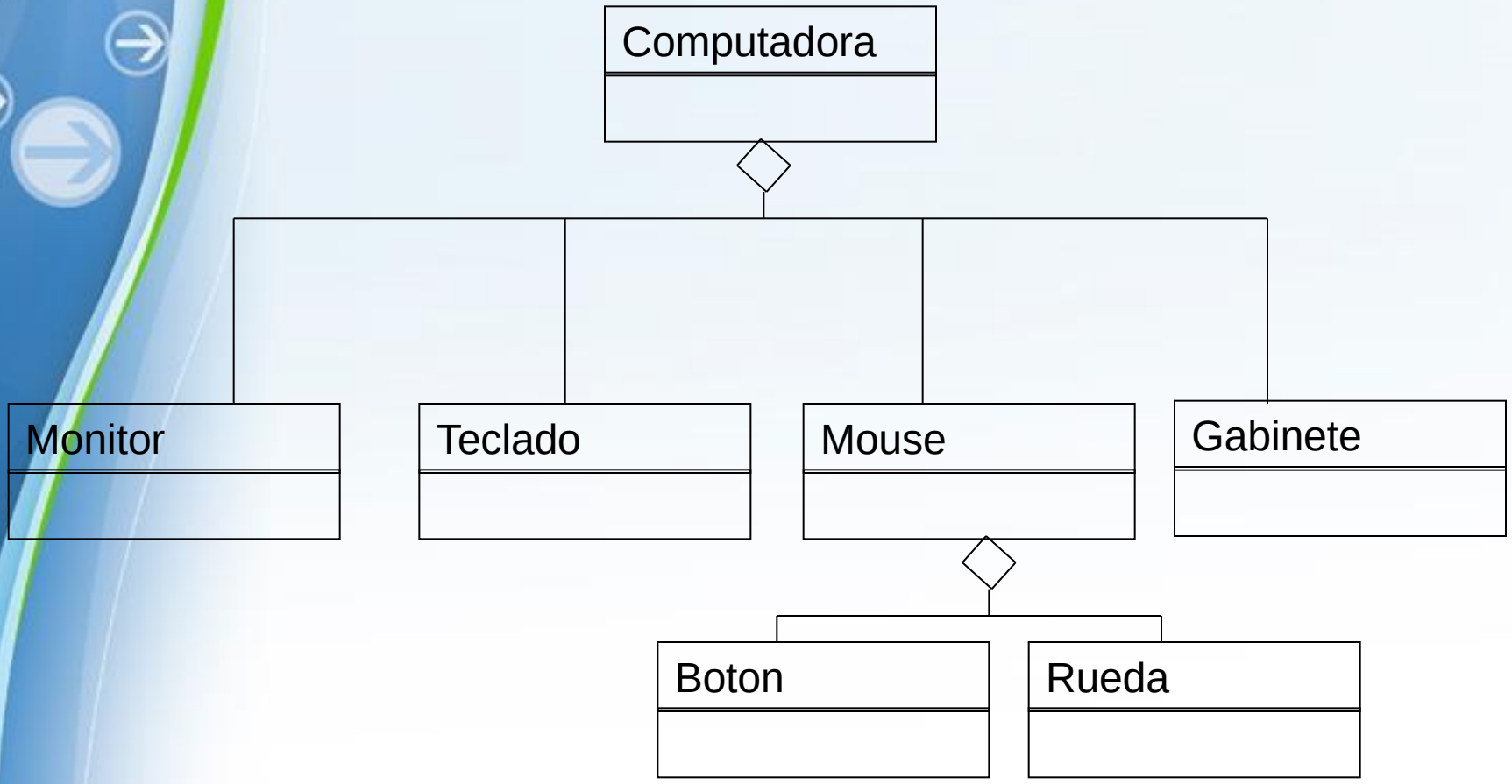
Diagramas de Clases: Agregación

- Representa la relación : ***es parte de***

- Se representa por una línea entre el componente y el todo con un rombo SIN relleno

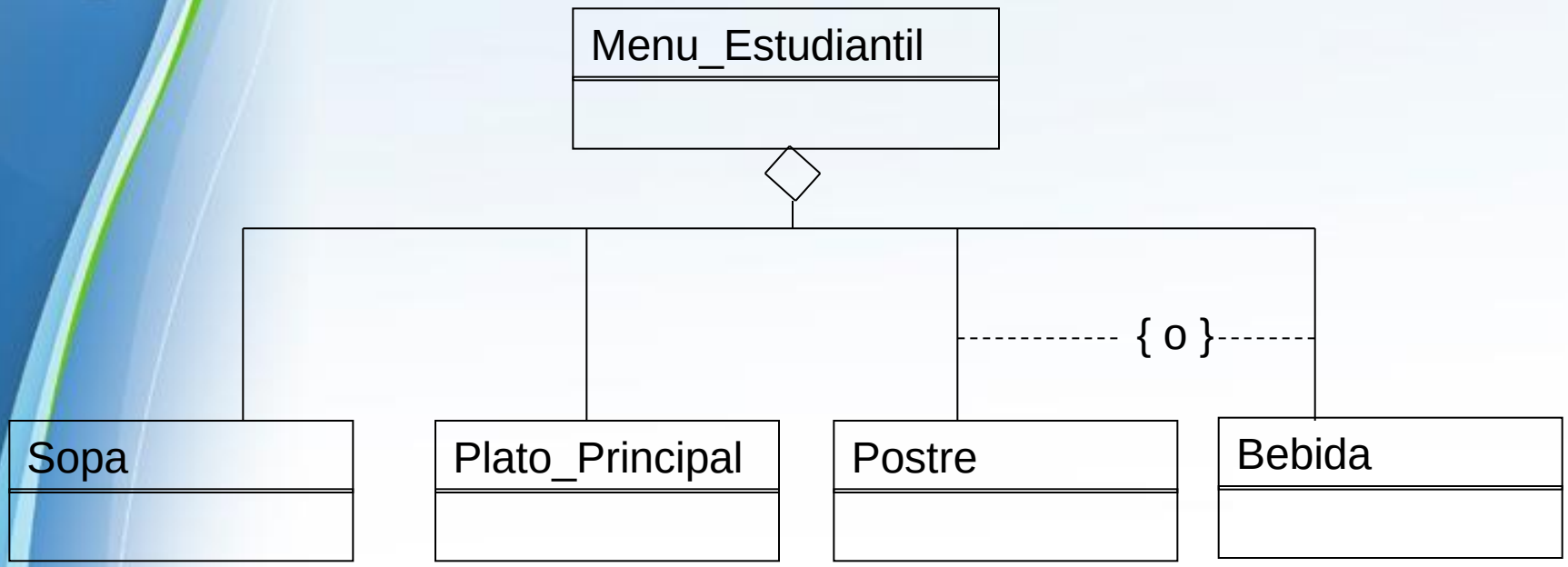


Ejemplos de Agregación



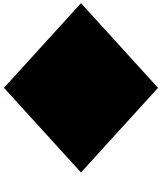
Restricciones a una agregación

- Permite mostrar que un componente u otro es parte del todo

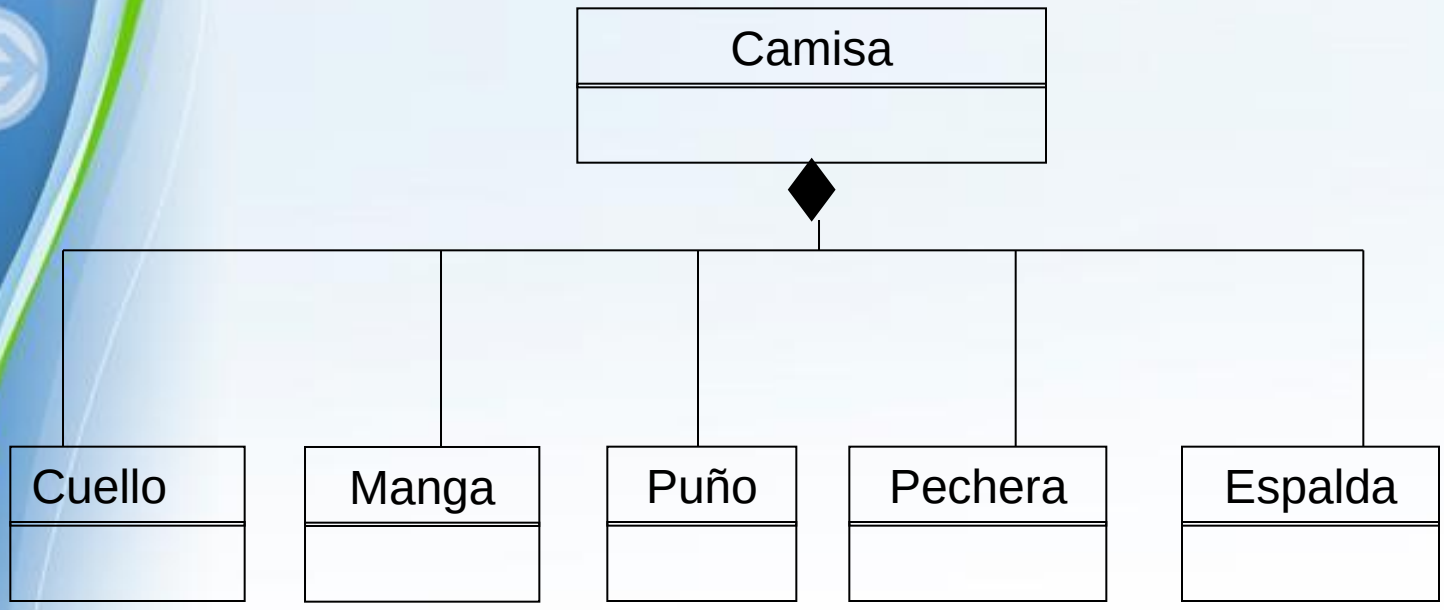


Diagramas de Clases: Composición

- Representa una forma especial de agregación
- Cada componente dentro de una agregación puede pertenecer SOLO a un TODO
- Cuando el todo desaparece, las partes también lo hacen
- Se representa por una línea entre el componente y el todo con un rombo CON relleno



Ejemplo de Composición



Diagramas de Interacción

- Describen una interacción.
- Dos tipos: Diagramas de Secuencia y Colaboración
- Diagramas de **Secuencia**:
  - Destacan la ordenación temporal de los mensajes
- Diagramas de **Colaboración**:
  - Destacan la organización estructural de los objetos participantes.
- Equivalencia semántica

Instancias

- Se representan con un rectángulo y a diferencia de las clases su nombre comienza con minúscula
- Su nombre esta subrayado
- Pueden ser:
  - **Nombradas** (miPc: Computadora)
  - **Anónimas** (:Computadora)

Diagrama de secuencia

Normalmente, un diagrama de secuencia contiene:

Objetos

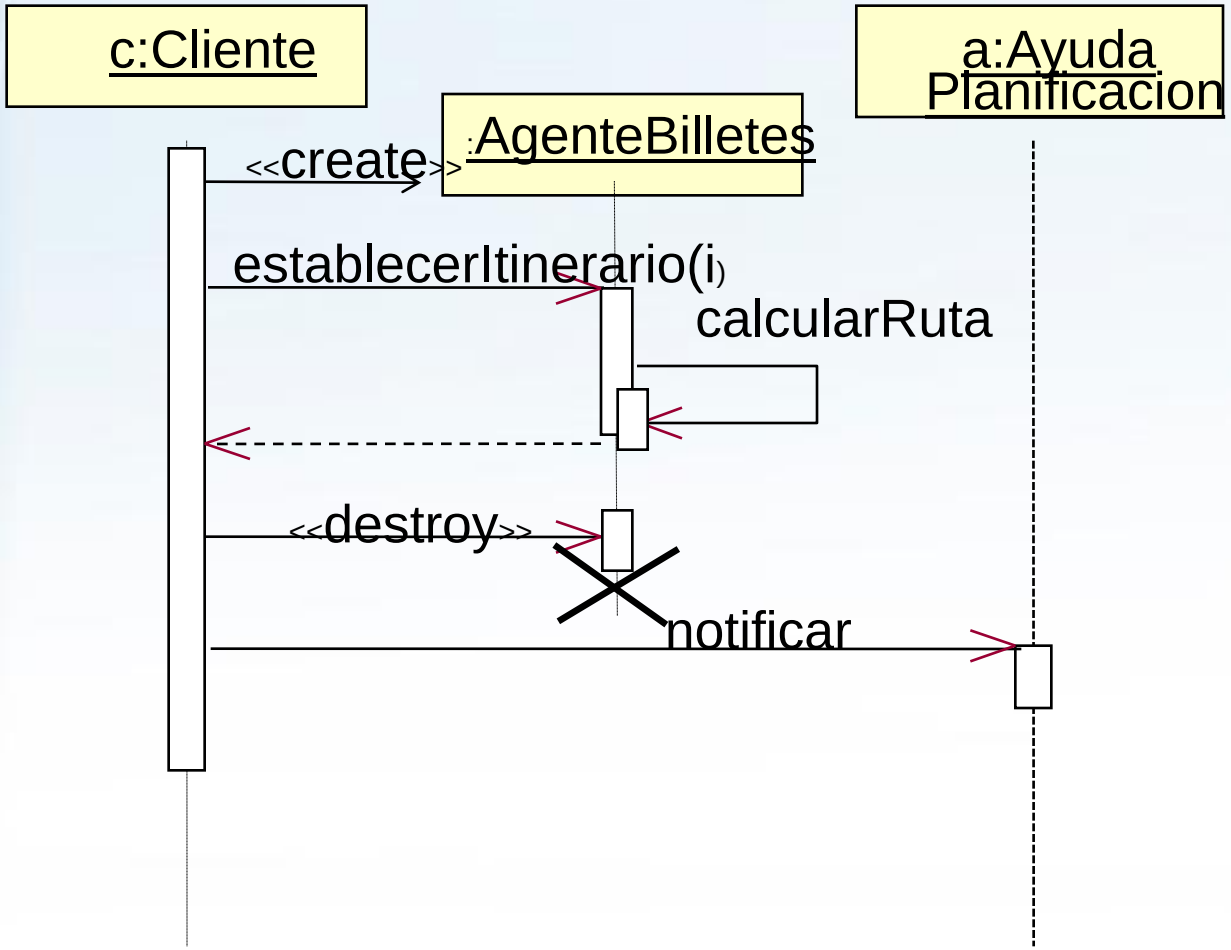
- Notación
- Linea de Vida
- Activación

Mensajes

- Notación
- create, destroy
- anidamiento de activaciones
- mensaje de retorno

Multiobjeto

Diagrama de Secuencia



Ejemplo

Acciones del actor Respuesta del sistema

1- Tomar un curso

2- Obtener pre requisitos del curso

3- validar condición del estudiante

4- validar estado del curso

5- guardar la inscripción

6- Informar al estudiante

Situaciones alternativas

3.1- El estudiante no cumple los pre requisitos -> Rechazar

4.1- El curso esta cerrado -> Rechazar

Diagrama de Secuencia - Ejemplo

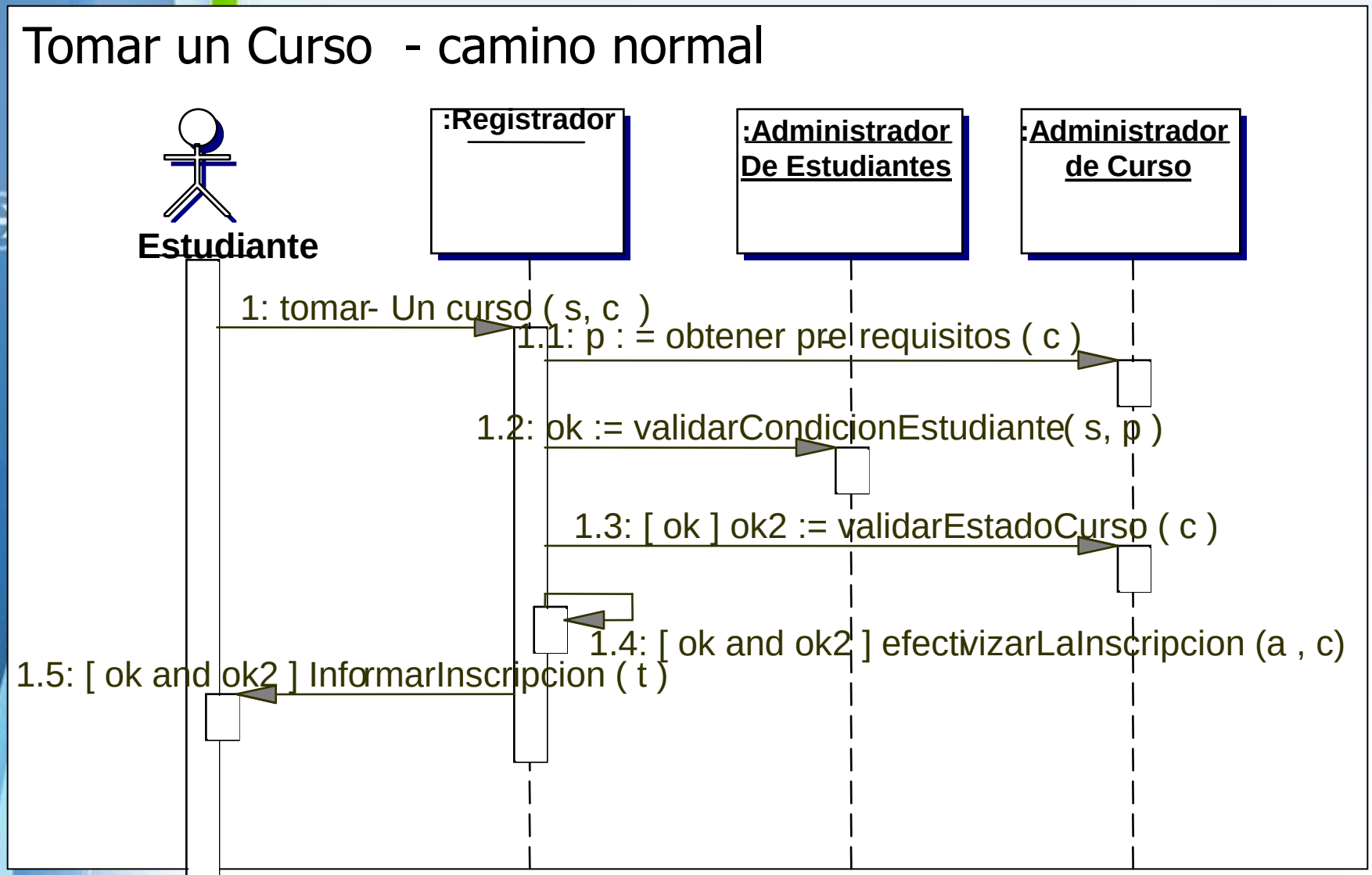


Diagrama de secuencia - Ejemplo

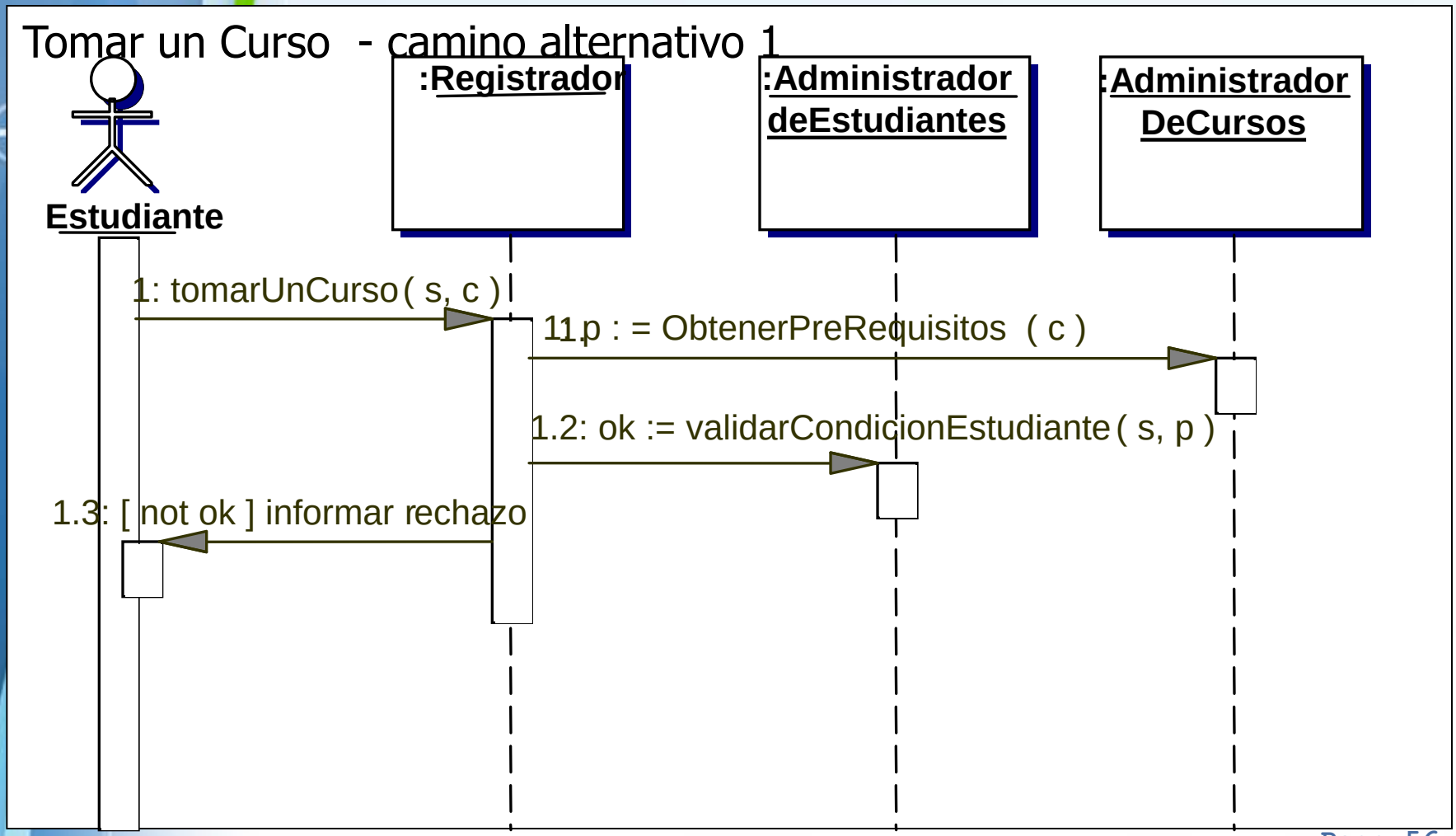
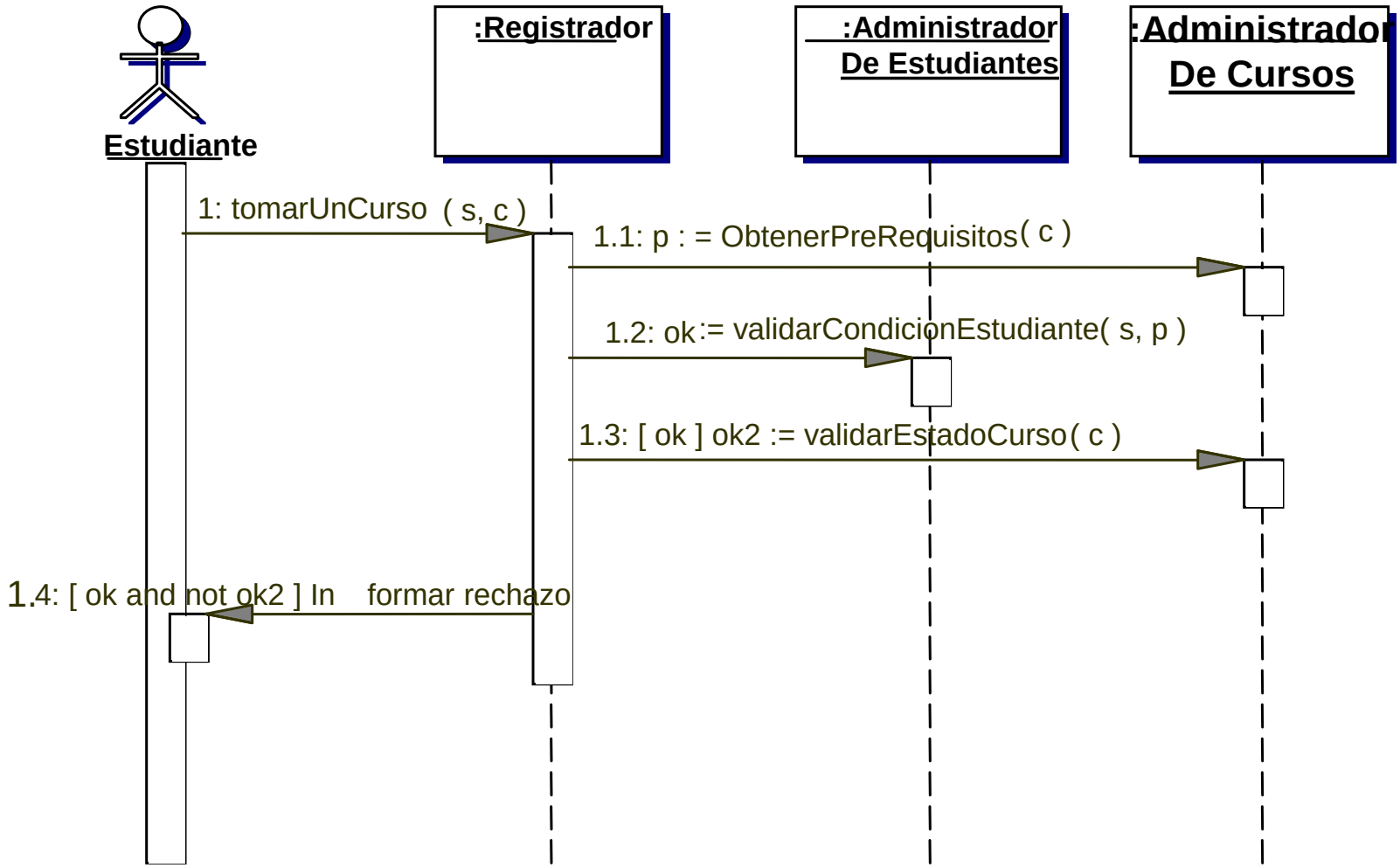


Diagrama de secuencia – Ejemplo

Tomar un Curso - camino alternativo 2



Diagramas dinámicos - Diagrama de colaboración

- Un diagrama de colaboración destaca la organización de los objetos que participan en una interacción.

- Se construye colocando en primer lugar los objetos que participan en la colaboración como nodos de un grafo. Después, se representan los enlaces que conectan esos objetos como arcos del grafo. Finalmente, se colocan los mensajes que envían y reciben los objetos.

Diagramas de colaboración

Un diagrama de colaboración generalmente contiene:

- Objetos
- Links entre objetos
- Mensajes

Múltiples Receptores

Es posible representar el envío de un mensaje a múltiples receptores

(un profesor pide una tarea a sus alumnos)

Diagramas de Secuencia vs. Diagramas de Colaboración

- Equivalencia semántica
- Simples para comportamientos simples.
- Si hay mucho comportamiento condicional, usar diferentes escenarios.
- Los diagramas de secuencia muestran mejor el **orden** en que se ejecutan los mensajes
- Los diagramas de colaboración muestran claramente los **objetos** con **que interactúa** un determinado objeto.

Diagrama de Secuencia (Caso de uso)

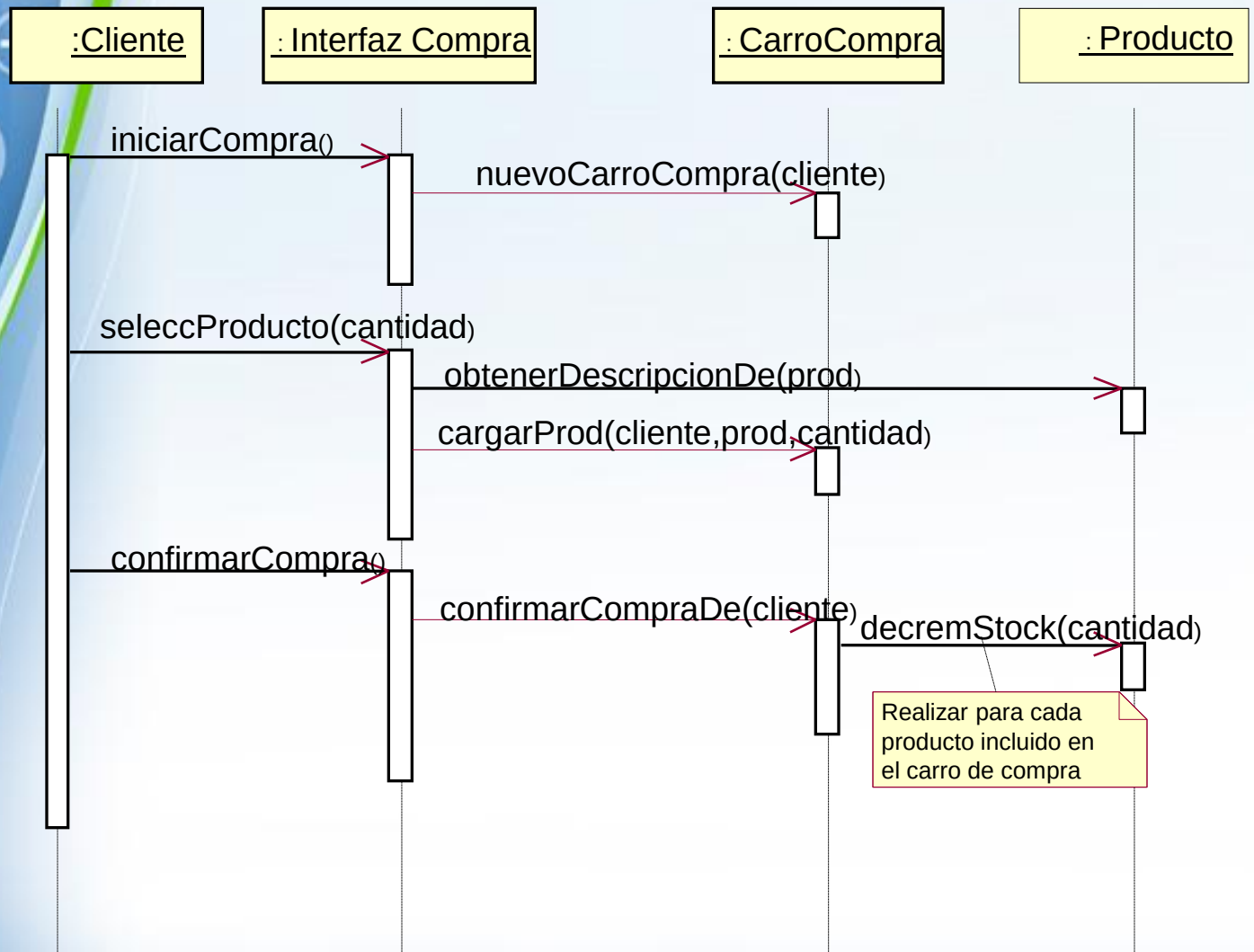
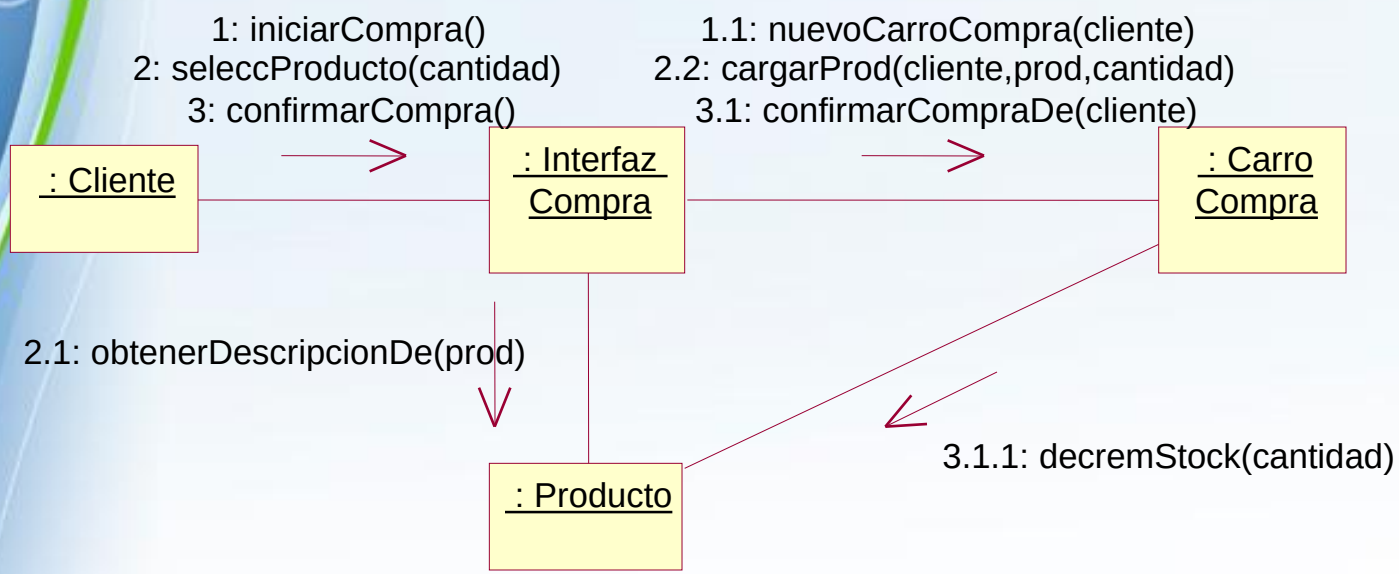


Diagrama de Colaboración (Caso de Uso)



Nos vemos en el parcial….