Técnicas Avanzadas de Programación

Computadores II

Sobre los docentes

Lic. Pablo Andrés Barrientos
JTP de Programación Funcional en la UNLP.
Trabajo en industria desde 2006.

Lic. Carlos Lombardi
Prof. de Paradigmas de Programación en la UTN desde 2005 hasta 2009.
Trabajo en industria desde 1988 hasta 2007.
Prof. de IBD en la UNQ en 2008 y 2009.
Estudiante del Doctorado en Computación.

Equipo de trabajo en práctica

Gastón Pinat, Estudiante avanzado de IACI (UNQ), comisión 2. Trabajo en industria.

Valeria De Cristófolo, Lic. de la UNLP, instructora en TPI. Trabajo en industria.

Leandro Silvestri, estudiante de TPI (UNQ), colaborador *ad-honorem*.

Alan Rodas, estudiante de TPI (UNQ), colaborador *ad-honorem*.

Nahuel Garbezza, estudiante de TPI (UNQ), colaborador *ad-honorem*

Objetivos del curso

Estudiar y analizar los fundamentos de las técnicas de programación actuales.
Entender el paradigma de objetos, sus conceptos relacionados, características, ventajas y aplicaciones dentro del desarrollo de sistemas de software.
Desarrollar prácticas con lenguajes orientados a Objetos.
Profundizar en el diseño y programación orientados a objetos que le permitirán al estudiante realizar un enfoque adecuado a los problemas que se le presenten.
Presentar los patrones de diseño principales y sus características.
Conocer técnicas de testing y mantenimiento de aplicaciones.

Contenidos

Parte I: Conceptos básicos del paradigma orientado a objetos

Abstracción, objeto, comportamiento, estructura interna, mensaje, método, encapsulamiento, double encapsulamiento, cohesión y acoplamiento, clase, instanciación, polimorfismo, etc).

Parte II: Conceptos avanzados

Bloques. Estructuras de control.
Contenedores.
Double dispatching
Subclasificación, herencia, method look-up, redefinición de método.
Igualdad e identidad
Método abstracto, clase abstracta. Constructores, inicialización.
Reutilización
Tipos. Interfaces. Tipado estático y dinámico. Lenguajes OO.

Parte III: Aplicación de conceptos

Streams. Excepciones, manejadores.
Testing, SUnit.
Testing.
Backtracking.

Parte IV: Análisis y diseño orientado a objetos

UML y diagramas.

Parte V: Patrones de diseño

Definición de patrón, descripción, catálogo.
Presentación de algunos patrones de diseño.

Régimen de cursada (I)

Entregas semanales individuales de ejercicios de la práctica
Los ejercicios a entregar serán elegidos por los docentes.
Se tendrá una semana de plazo para hacer la entrega.
Se califican con números y se deberá tener 60% de los puntos totales para acceder a la instancia de evaluación parcial.
Recordar que las entregas son *individuales*.
Quien no cumpla esto se calificará como *AUSENTE*.

Régimen de cursada (II)

Una evaluación teórico-práctica INDIVIDUAL
Será calificada con números
Tendrá una instancia de *recuperatorio*
En caso de no presentarse a la evaluación ni al recuperatorio, se calificará AUSENTE.
Quien se presente al parcial pero no al recupertorio será calificado como AUSENTE
Quien no se presente al parcial pero sí al recupertorio y no apruebe será calificado como DESAPROBADO
En caso que se obtenga una nota menor a 6 y mayor a 4, se deberá presentar a un examen *integrador*, que se tomará antes del cierre de actas.

Régimen de cursada (III)

Un **trabajo práctico final GRUPAL**
Implicará un pequeño desarrollo o investigación, para definir la nota final.
Será entregado y expuesto antes del cierre de actas.
El alumno que llegue a esta instancia se considera APROBADO, salvo que no entregue el TP final a tiempo (con la debida justificación, que será puesta en consideración), en cuyo caso quedará AUSENTE.

Régimen de cursada (IV)

Examen integrador:

Incluirá todos los temas vistos durante la cursada

Se aprueba con 4 o más.

Tiene una instancia de recuperación en caso de no presentarse o de no aprobar el examen, que se tomará antes del comienzo del siguiente semestre, de acuerdo al calendario académico de la UNQ.

Se debe realizar también el TP final para figurar APROBADO pero en este caso se hará de forma INDIVIDUAL. En caso que no presente el TP final, se pasará AUSENTE.

En caso de no aprobar en esta instancia de evaluación, el alumno figurará DESAPROBADO.

Teoría

Repaso y consultas a partir de las 18hs

Clase “formal” a partir de las 19hs y hasta las 22hs

Referente de curso

Ayudante

Realiza seguimiento y asesoramiento de los grupos *de 3 personas* asignados a él

Corrige entregas semanales, parciales (junto al profesor) y guía en la elaboración del trabajo final.

Referente de la materia para el grupo.

No es exclusivo de los grupos.

Horas de estudio

6-8 horas de clase (teoría y práctica).
2 horas de estudio.
6 horas de práctica frente a maquina.

Cronograma tentativo

Semana 1°: comienzo de teoría y práctica
Semana 10°: evaluación teórico/práctica
Semana 11°: comienzo de trabajo práctico final
Semana 12°: recuperatorio de evaluación
Semana 17°: defensa/exposición de tp finales
Semana 18°: integrador - entrega de notas

Herramientas y materiales de estudio

Smalltalk VisualWorks (7.5)
ambiente de desarrollo + lenguaje de programación

Prácticas de la materia.
Apuntes generados por la materia. Ver apunte de MATLAB (IACI) y el de back-tracking(op).
Alguna bibliografía recomendada (opcional), de la cual algunos libros pueden encontrarse en la biblioteca.

BLOG: http://pbarrientos.blog.unq.edu.ar

Bibliografía recomendada

Budd, Timothy A. An introduction to Object-Oriented programming. 2004.

Design Patterns. Elements of Reusable Objects Oriented Software. Garnma, Helm, Johnson, Vlissides, Addison-Wesley, Professional Computing Series.

Visual Works Application Developer's guide. Documentación de Visual Works Non commercial 3.0. Publisher: ObjectShare

Keogh, J., Giannini, M., and Rinaldi, W. 2004 *Oop Demystified (Demystified)*. McGraw-Hill Osborne Media.

West, David. Object thinking. Microsoft Press. 2004

The UML Reference Manual. Rumbaugh, Jacobson and Booch. Addison Wesley Longman, Inc, 1998.

Otros... (ver programa de la materia)

Parte I

Conceptos básicos del paradigma OO

Algunas preguntas antes de empezar...

¿Qué es la programación?

¿Qué es un programa?

¿Cuáles son las tareas de un programador?

¿Qué es un paradigma de programación?

Veamos...

Conceptos básicos: abstracción

Proceso de eliminación de los detalles acerca de un fenómeno, proceso o entidad, para concentrarse en los aspectos significativos (sustantivo)
Concepto formado a partir de la generalización de características comunes (sustantivo)
Proceso por el cual las ideas son separadas de los objetos concretos (verbo)

El proceso es subjetivo y depende del contexto de análisis
Ejemplo: concepto de *viaje*

Conceptos básicos: modelo

Versión simplificada de cierto fenómeno, proceso o entidad
Proceso de abstracción/simplificación
No representa fielmente dado que simplifica

Ejemplo: lógica proposicional
Permite realizar simulaciones y predicciones del mundo real

Conceptos básicos: relacionando conceptos

La programación OO se independiza del modelo de cómputo subyacente, permitiendo concentrarnos en el problema a resolver.
La *abstracción* se plantea en términos de similitudes entre fenómenos, conceptos, entidades, etc.
El problema a resolver se representa mediante un *modelo* construido en base a la noción de *objetos* y la interacción entre ellos
Un programa OO es un modelo que simula parte de la realidad

Conceptos básicos: paradigma de programación

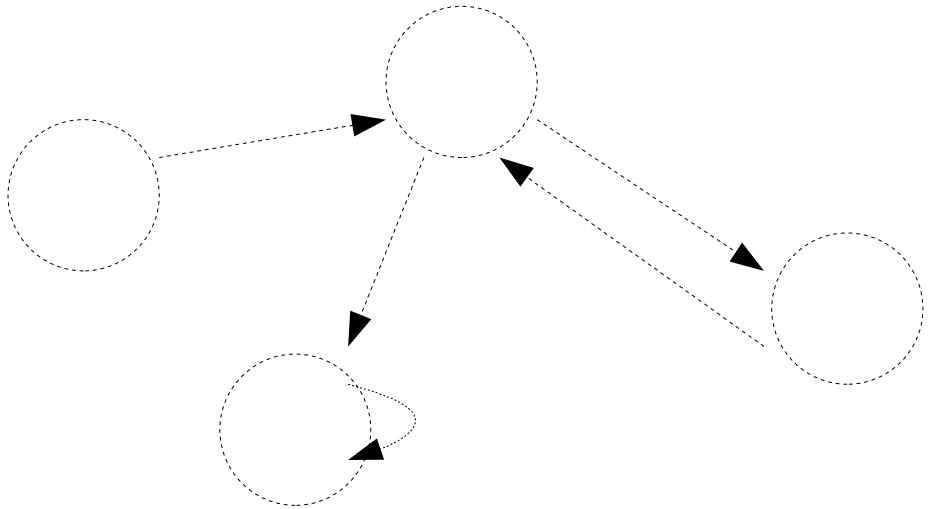
Paradigma: conjunto de elementos + reglas aceptadas para realizar una tarea
Analogía: juego de mesa

Paradigma de programación: marco para diseñar y construir programas
Ejemplos: imperativo, funcional, orientado a objetos!

¿Alcanza con saber los elementos y las reglas?

Conceptos básicos: programa OO

Conjunto de *objetos* que llevan a cabo ciertas acciones, colaborando entre ellos mediante el envío de *mensajes*



Conceptos básicos: primera tarea

Problema: obtener la lista de estudiantes del curso con su e-mail.

Solución (programa):

El profesor indica a un estudiante que le dé una hoja, y con esa hoja le pedirá que agregue su nombre y e-mail. Cuando termina la tarea, el estudiante le pasa la hoja al siguiente.

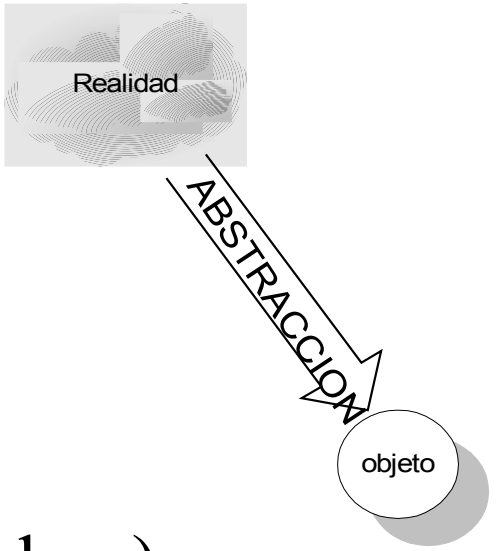
Los estudiantes tienen (?) una lapicera para escribir su nombre.

El último estudiante le alcanza la hoja al profesor.

Conceptos básicos: objeto

Abstracción de una entidad del dominio del problema a resolver

Elemento básico del paradigma que sirve para construir programas



Un objeto posee:

Comportamiento

Tiempo de vida

Identidad

Estado interno

Clase a la que pertenece (es *instancia* de esa clase)

Todo es un objeto (o debería serlo)

Conceptos básicos: comportamiento de un objeto

Define sus *responsabilidades*.
Indica *qué* sabe hacer un objeto y *cómo*.
El *qué* se especifica a través del conjunto de *mensajes* que el objeto sabe responder (protocolo).
El cómo realiza sus responsabilidades se especifica en la implementación de *métodos,* que son privados del objeto.
Por cada mensaje que el objeto puede recibir, debe existir un método.

Conceptos básicos: comportamiento de un objeto

Cuando un objeto se comunica con otro *enviando* un mensaje, el receptor responde activando el método asociado a ese mensaje.
El emisor del mensaje no tiene interés sobre cómo el receptor realizará la tarea pedida.
Para llevar a cabo alguna responsabilidad (al enviarse un mensaje), el objeto receptor podrá interactuar con los demás objetos que conoce.

Conceptos básicos: tiempo de vida

Los objetos no son eternos, en un momento se crean, en otro se destruyen.

Crear un objeto: responsabilidad especial que se resuelve con mensajes especiales, a veces llamados *constructores*.

Destruir un objeto

- ¿cuándo corresponde que un objeto desaparezca?
- ¿quién decide destruir objetos?

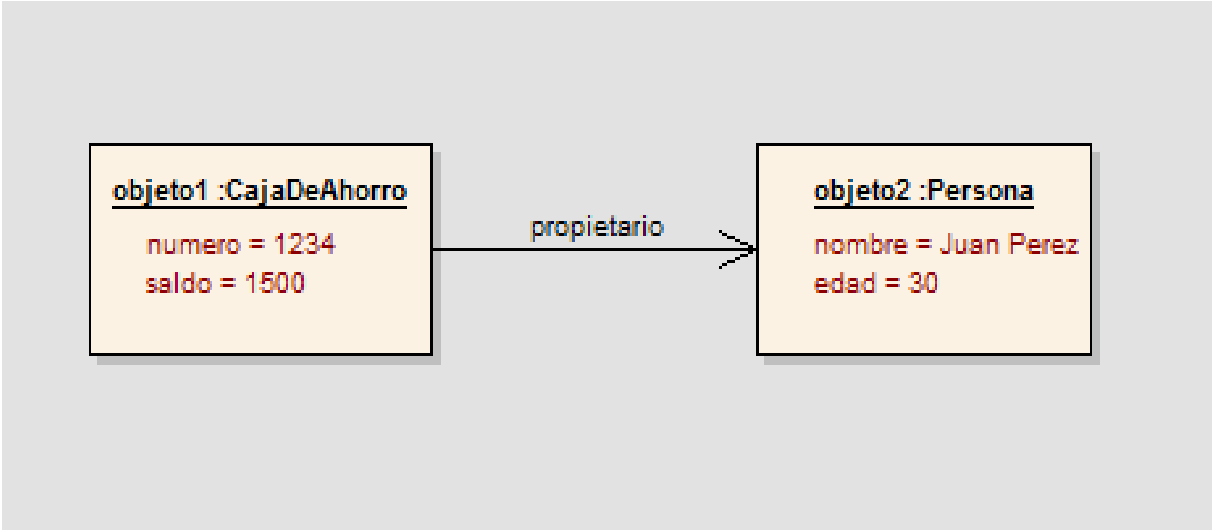
… ya lo veremos.

Conceptos básicos: estado de un objeto

Estructura interna del objeto
Conjunto de variables de instancia que posee.
Son *referencias* a otros objetos.
El estado es *privado* del objeto. Ningún otro objeto puede accederlo.

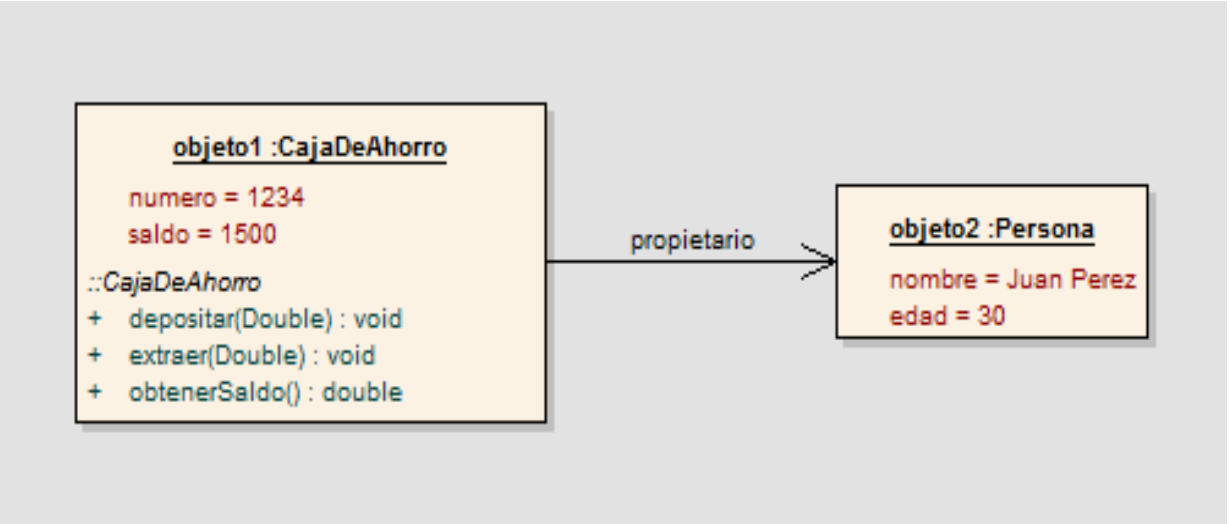
Las variables de instancia pueden referenciar a:
Propiedades o características intrínsecas del objeto
Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades
Nada (!?)

Conceptos básicos: estado de un objeto



objeto1 tiene una variable de instancia *saldo* con un valor de 1500, otra variable de instancia número con valor 1234, y finalmente otra variable de instancia *propietario* cuyo valor es **objeto2**.

Conceptos básicos: estado y comportamiento de un objeto



objeto1, tiene ahora 3 responsabilidades:

- depositar
- extraer
- obtenerSaldo

Conceptos básicos: identidad de un objeto

Un objeto es solamente idéntico a sí mismo.
Sin embargo... dos objetos podrían tener las mismas características y el mismo comportamiento!

Pero no son el mismo objeto!

En ese caso decimos que ambos objetos son *iguales* (pero NO identicos)

Conceptos básicos: envío de mensaje

Para que un objeto le envíe un mensaje a otro, lo debe conocer. ¿Cómo?
Como resultado del envío del mensaje, puede retornarse un resultado.
El resultado no puede ser otra cosa que un objeto.

¿Cualquier objeto puede entender cualquier mensaje?
¿Qué pasa si le envío a un objeto un mensaje que no entiende?

Conceptos básicos: envío de mensaje

Mensajes que entienden los números

abs, even, sin

+ unNumero, * unNumero

> unNumero

Mensajes que entienden los booleanos

&, |, not

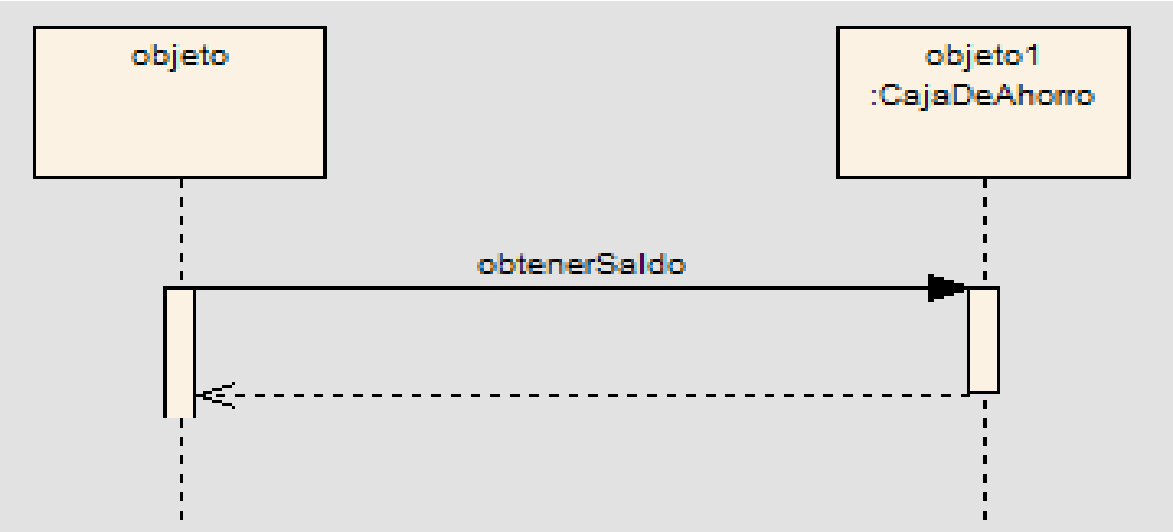
Mensajes que entienden las cuentas bancarias

saldo, extraer:, depositar:

¿Cuál es el resultado en cada caso?

¿Qué pasa si le envío a un objeto un mensaje que no entiende?

Conceptos básicos: envío de un mensaje



objeto le envía el mensaje *obtenerSaldo* a **objeto1**, y éste luego de activar el método correspondiente, retorna el valor de su variable de instancia.

Conceptos básicos: envío de un mensaje

Cada lenguaje de programación tiene una sintaxis particular para indicar el envío de mensajes.
Smalltalk tiene 3 tipos de mensajes, dados en prioridad de evaluación:

Unarios:

cuentaBancaria obtenerSaldo

Binarios o aritméticos:

8 + 5

De palabra clave: con uno o más parámetros, cada uno precedido por una palabra clave seguida de (:)

cuenta extraer: 500

Los parámetros son objetos que servirán para la ejecución del método.

Conceptos básicos: mensajes en cascada

En Smalltalk se pueden enviar mensajes a un mismo objeto en cascada.
Sintaxis:

objetoReceptor mensaje1; mensaje2; ... mensaje n.

Cada mensaje será enviado a *objetoReceptor*.

Ej: unPerro ladrar; comer; sentarse.

Conceptos básicos: ejemplos de mensajes

pers1 dni even
pers1 dni + 25
pers1 dni between: pers2 dni + 10 and: 99999999

unCajero deposita:5+6 aCuenta:123 fecha: unaFecha

(unCajero deposita:5+6 aCuenta:123) saldo
unCajero deposita: 5 + 6 aCuenta:123 saldo
unCajero deposita: 5+6 aCuenta:123; saldo

unAvion arrancarMotor; avanzar: 10; despegar.

Conceptos básicos: especificación de un método

Un método puede realizar básicamente 3 cosas:

Modificar el estado interno del objeto (receptor), es decir, asignar valor a una variable de instancia

```
>>asignarPropietario: unPropietario
    propietario := unPropietario.
```

Colaborar con otros objetos (delegar responsabilidades)

```
>>extraer: un Monto
    banco extraerme: unMonto.
```

Retornar un objeto como resultado y terminar

```
>> obtenerSaldo
    ^ saldo.
```

Un método **no debería** tener más de 15 líneas de código.

Conceptos básicos: sintaxis dentro de un método

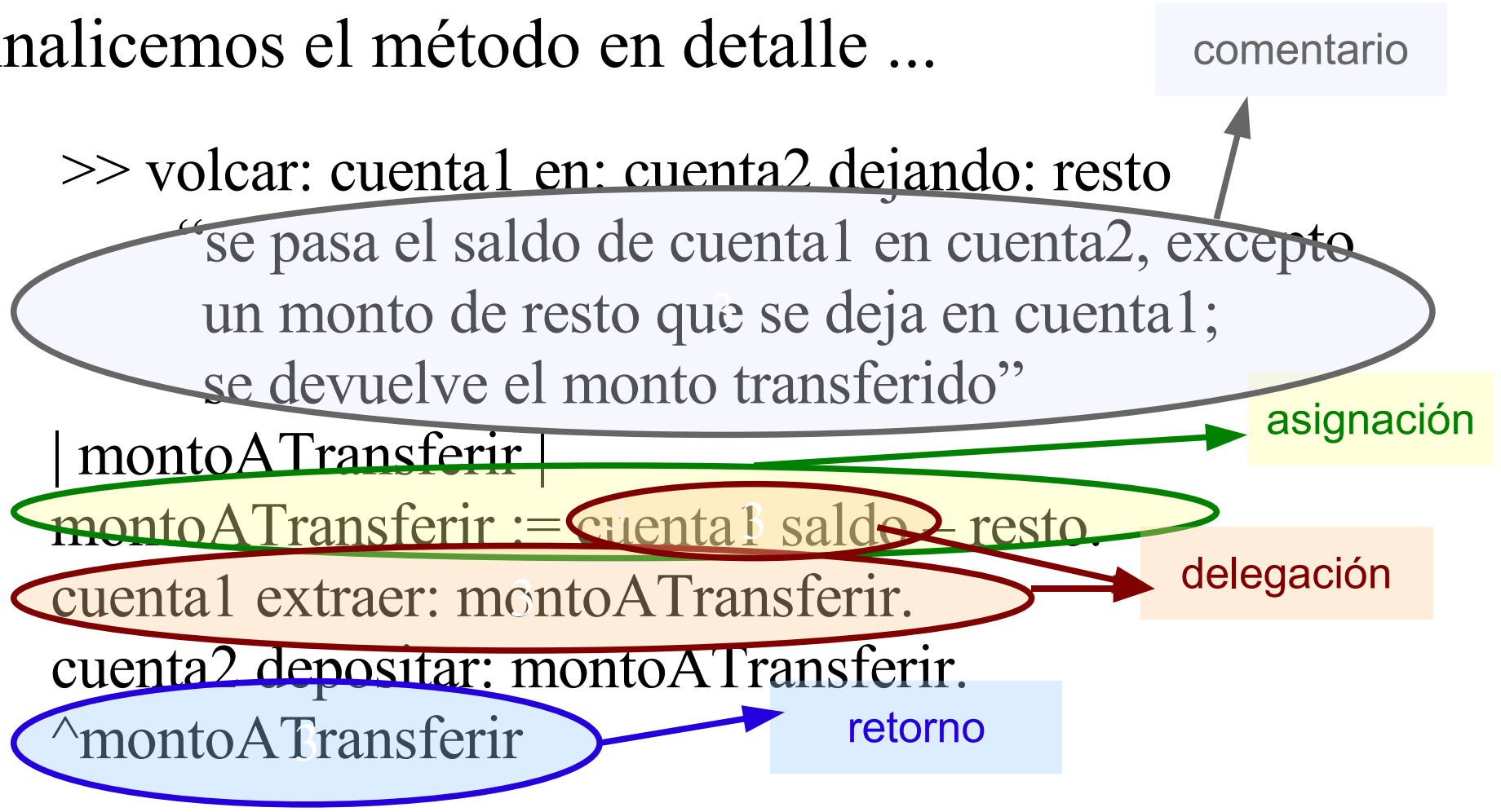
Supongamos que queremos definir el comportamiento de un banco para pasar el saldo de una cuenta a otra, dejando un resto. Definimos:

```
>> volcar: cuenta1 en: cuenta2 dejando: resto
        "se pasa el saldo de cuenta1 en cuenta2, excepto un
         monto de resto que se deja en cuenta1;
         se devuelve el monto transferido"
| montoATransferir |
montoATransferir := cuenta1 saldo – resto.
cuenta1 extraer: montoATransferir.
cuenta2 depositar: montoATransferir.
^montoATransferir
```

Conceptos básicos: combinando opciones en el mismo método

Analicemos el método en detalle ...

```
>> volcar: cuenta1 en: cuenta2 dejando: resto
    "se pasa el saldo de cuenta1 en cuenta2, excepto
    un monto de resto que se deja en cuenta1;
    se devuelve el monto transferido"
| montoATransferir |
montoATransferir := cuenta1 saldo - resto.
cuenta1 extraer: montoATransferir.
cuenta2 depositar: montoATransferir.
^montoATransferir
```



Conceptos básicos: encapsulamiento

Es la cualidad de los objetos de ocultar los detalles de implementación y estado interno del resto.

Ventajas:

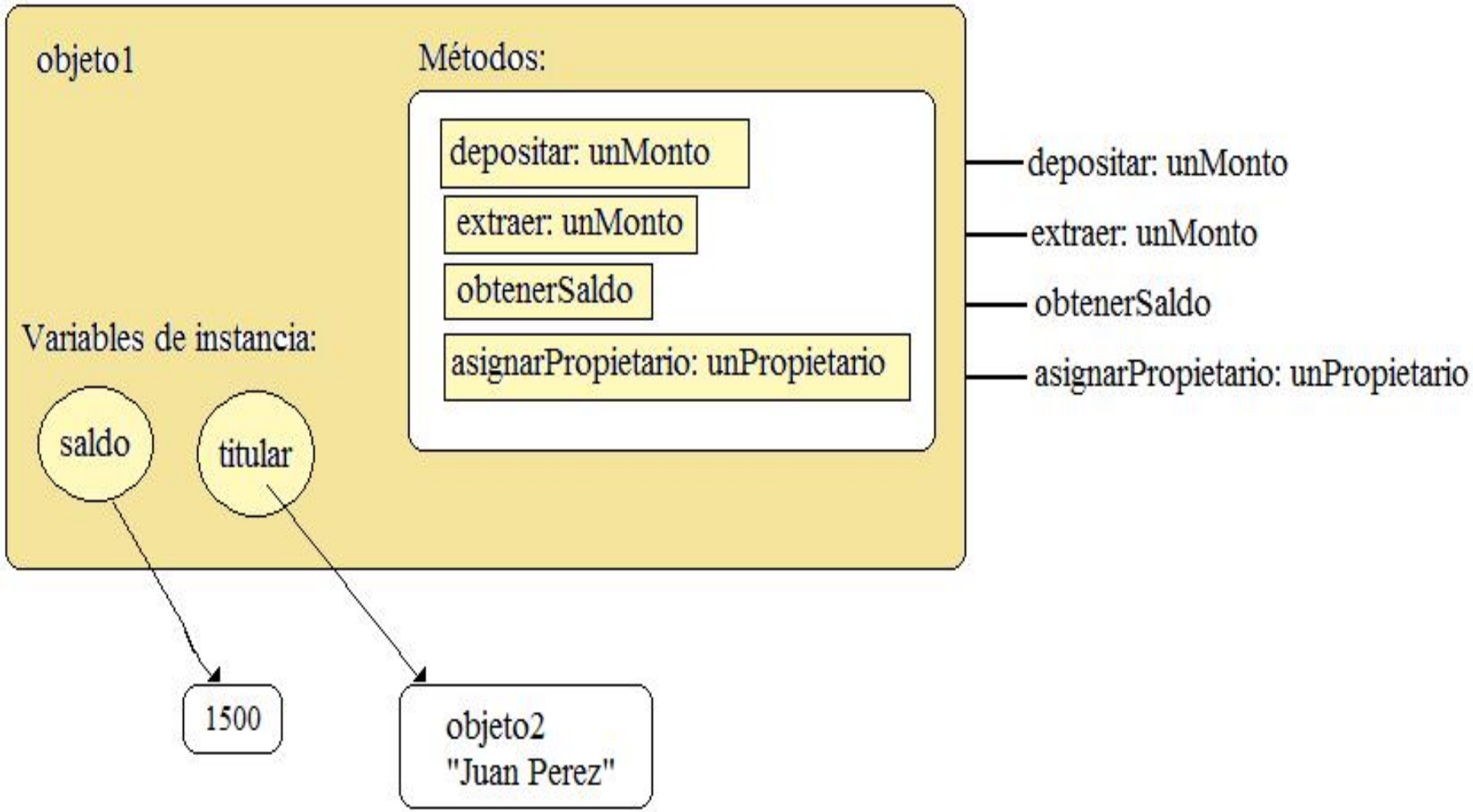
Facilita la modificación de la implementación, sin afectar a los demás objetos que lo conocen.

El cambio es transparente dado el bajo *acoplamiento*.

Escalabilidad del software frente a cambios.

Cuando desarrollamos con objetos debemos orientarnos a *protocolos* si queremos obtener sus beneficios (algunos lenguajes permiten violar el encapsulamiento de diversas formas).

Conceptos básicos: encapsulamiento



Conceptos básicos: doble encapsulamiento

Patrón de codificación que implica el ocultamiento total de la estructura interna del objeto. Si un objeto requiere "información interna" sobre otro **y sobre él mismo**, entonces la solicita con un mensaje.
Se usan métodos *getters* y *setters* (accessors) como la única forma de acceder a la representación interna.

Conceptos básicos: doble encapsulamiento

Pero!

Los métodos getters y setters no necesariamente reflejan la estructura interna de un objeto.

Puede haber getters y setters que no se relacionan con la estructura interna del objeto

Ejemplo:

unCD capacidadMaxima

Puede haber variables de instancia que pueden no ser accesibles mediante accessors.

Ejemplo:

Una persona que conoce sus hijos pero que sólo expone la cantidad que tiene

Conceptos básicos: doble encapsulamiento

Definición de getter

```
>> nombre
    ^nombre
```

Definición de setter

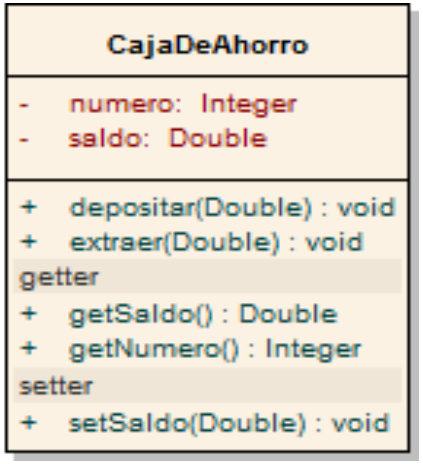
```
>> nombre: unNombre
   nombre:= unNombre.
```

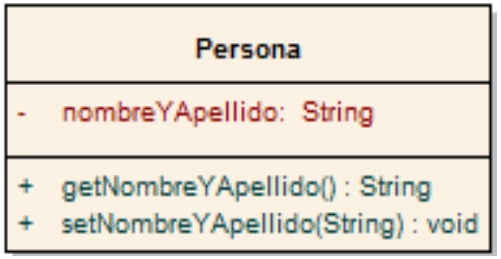
Ventajas:

Existe un sólo punto de entrada al estado interno

El impacto es mínimo a la hora de modificar la estructura interna del objeto

Conceptos básicos: doble encapsulamiento





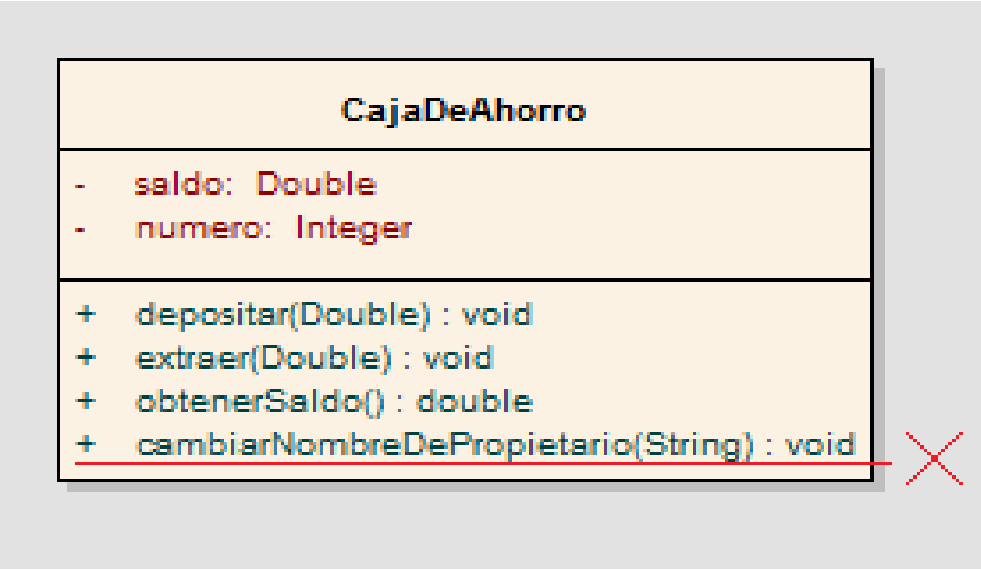
En el caso de CajaDeAhorro, no se definió un setter para el número de cuenta, ya que sólo se asignará el número al crear el objeto (luego lo veremos).

¿Qué cambios hay que hacer si nombreYApellido se divide en dos variables (nombre, apellido)? ¿Cómo repercute este cambio en los clientes de estos métodos? ¿Y en los mensajes del mismo objeto?

Conceptos básicos: cohesión

Un objeto con alto grado de cohesión tiene un conjunto *claro* de responsabilidades *relacionadas* que lo definen.

Marca cuán independiente es el objeto del resto del sistema.



Conceptos básicos: acoplamiento

Es la medida de cuán interrelacionados están dos elementos entre sí.
Cuando un objeto depende de otro se dice que está acoplado.
Cuando un objeto interactúa con otro, pero sin saber detalles de implementación, se dice que está *débilmente acoplado*.
Ejemplo:
~~(unSeminario getEstudiantes) agregar: unEstudiante~~
unSeminario agregar:unEstudiante

Conceptos básicos: cohesión y acoplamiento

Un objeto debe ser:
Fuertemente cohesivo
Débilmente acoplado a otros objetos

Un método debería ser:
Altamente cohesivo

Estos dos conceptos se aplican de la misma forma a las clases (las veremos a continuación).

Conceptos básicos: clases e instancias

Hasta ahora trabajamos sólo con objetos, que vimos que son una *abstracción* (de una caja de ahorro, un perro, etc).

Si quisieramos modelar un banco ¿deberíamos especificar el comportamiento de cada una de las caja de ahorro?

¿Qué podemos *abstraer* en común a todas las cajas de ahorro?

¿Cómo capturamos esa noción de abstracción?

Utilizaremos un segundo concepto de abstracción: **clase**

Conceptos básicos: clases e instancias

Descripción abstracta de un conjunto de objetos.

Cada objeto es *instancia* de una clase.

Dos aspectos básicos

- Agrupamiento de comportamiento común de sus instancias.
- Molde para un conjunto de objetos.

Conceptos básicos: clases e instancias

Agrupamiento de comportamiento común de sus instancias.

Las instancias de una clase se comportan de la misma manera; entienden los mismos mensajes, usando los mismos métodos.

La implementación de un método está en la clase y no en el objeto particular.

El *method look-up* comienza en realidad en la clase del objeto, no en el objeto.

Method lookup: determinación del método para un mensaje.

Conceptos básicos: clases e instancias

Molde para un conjunto de objetos.

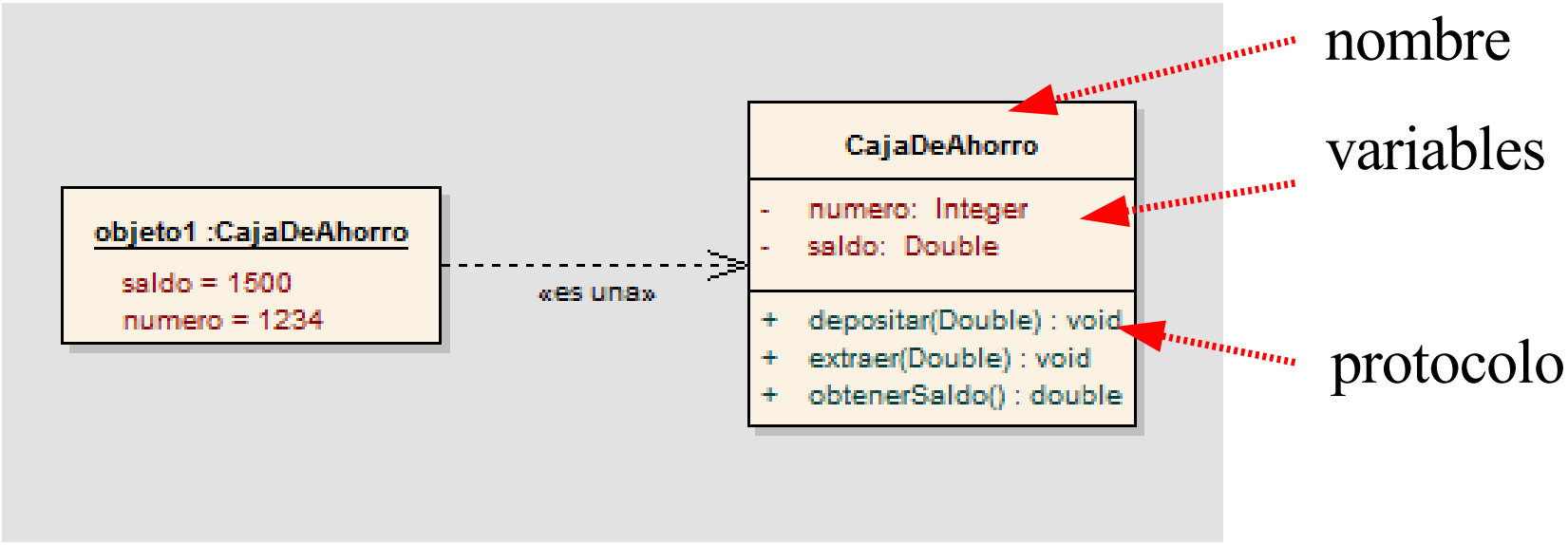
Define la forma o estructura interna de sus intancias, pero no define los valores particulares que tiene cada una, que pueden cambiar de instancia en instancia.

Creador de objetos que son instancias de ella.

Cada objeto que queramos crear … debe ser instancia de una clase.

Para crearlo le voy a enviar un mensaje a la clase correspondiente: “creame una nueva instancia”.

Conceptos básicos: clases e instancias



Por convención se usa *CamelCase* (la primera letra de cada una de las palabras es mayúscula), los nombres de clase empiezan con mayúscula, y las variables y métodos de las instancia comienzan con minúscula.

Conceptos básicos: declaración de una clase

Tomaremos como convención para describir una clase:

NombreDeClase

Variables de instancia

variable_1, variable_2,..., variable_n

Protocolo

>> metodo_1

...

>> metodo_n

Más tarde veremos nuevos elementos que conforman una clase.

La forma de definir clases en Smalltalk quedará como parte de la práctica

Conceptos básicos: cohesión y acoplamiento en clases

Una clase debe ser:
Fuertemente cohesiva
Débilmente acoplada a otras clases

Un método (definido ahora en la clase) debería ser:
Altamente cohesivo

Conceptos básicos: relaciones de conocimiento

Un objeto sólo puede enviarle mensajes a aquellos objetos que *conoce*.

Para que un objeto conozca a otro, dos formas

- Accede mediante un nombre:
  - Conocimiento interno: variables de instancia
  - Conocimiento externo: parámetros
  - Conocimiento temporal: variables temporales
  - Conocimiento global: variables globales
  - Pseudo-variables
- Accede a partir de otro que ya conoce mediante mensajes
  - Objetos anónimos

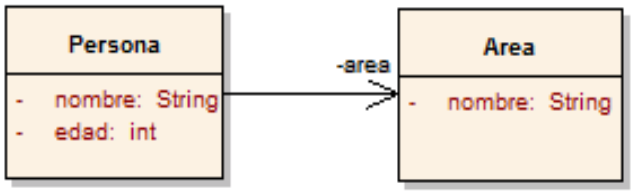
Conceptos básicos: variable de instancia

Define una relación entre un objeto y sus atributos
Se definen explícitamente como parte de la estructura interna de la clase
La relación dura tanto como viva el objeto, aunque el objeto ligado pueda cambiar.

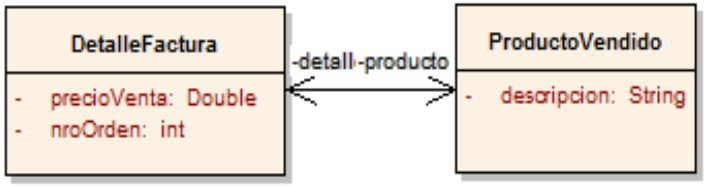
Conceptos básicos: variable de instancia

Al haber una variable de instancia, se crea una relación ***estructural*** de conocimiento entre las clases que puede ser:

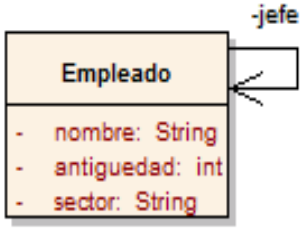
Relación unidireccional



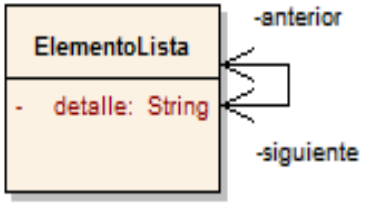
Relación bidireccional



Relación recursiva unidireccional



Relación recursiva bidireccional



Conceptos básicos: parámetros

El nombre de la relación de conocimiento se define explícitamente en el nombre del mensaje.
La relación dura el tiempo que el método se encuentre activo.
La ligadura entre el nombre y el objeto no puede alterarse durante la ejecución del método
Ejemplo (clase Cajero).

```
>> montoExtraccionPosibleDesde: cuenta
        cuenta registrarConsultaEnFecha: fechaActual.
        cuentasConsultadas agregar: cuenta.
        ^cuenta saldo min: montoMaximoExtraccion
```

Conceptos básicos: variables temporales

Definen relaciones temporales entre un nombre y un objeto
La relación con el objeto se crea durante la ejecución del método o conjunto de sentencias donde se definen.
Durante la ejecución del método, la ligadura entre el nombre y el objeto puede alterarse.
Ejemplo:

```
>> asDays
    | yearIndex |
    yearIndex := self year - 1901.
    ^yearIndex * 365  + (yearIndex // 4) + ((yearIndex + 300)
      // 400)  - (yearIndex // 100)  + self day - 1
```

Conceptos básicos: variables globales

Permiten nombrar un objeto que sea conocido en forma universal y sin restricciones por todos los demás objetos del sistema.
Su nombre se escribe con mayúscula en Smalltalk.
A medida que pasa el tiempo se puede modificar el binding del nombre con el objeto.
El uso que se le da en Smalltalk es mínimo.

Ejemplo: Hoy := Date today

Conceptos básicos: pseudo-variables

Existen dos: self y super.
self es el nombre especial para que el objeto se haga referencia a sí mismo, y sirve para que un objeto se pueda enviar un mensaje a sí mismo. En algunos lenguajes se lo denomina *this*.

```
>>extraer: unMonto
    self puedoExtraer: unMonto
    ...
```

super se verá más tarde en este curso...

Conceptos básicos: objeto anónimo

Es un objeto que no tiene nombre explícitamente, es decir, no está referenciado directamente por ninguna variable.
Es el resultado del envío de un mensaje a un objeto.

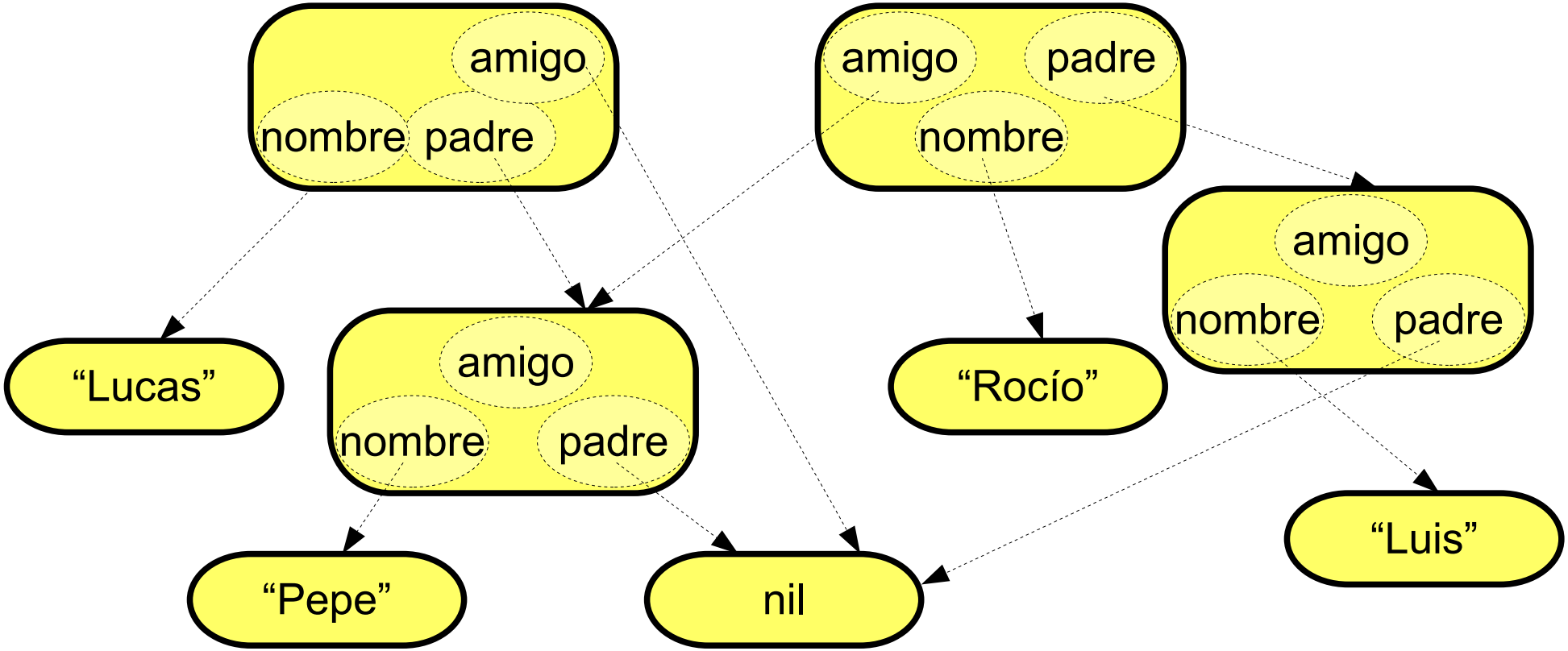
Ejemplo: unaPersona edad + 5

Conceptos básicos: objeto nulo

nil es un objeto, que es la única instancia de la clase UndefinedObject.
Representa la idea de *no objeto (*pero es un objeto).
Toda referencia no asignada tiene valor nil.
En Smalltalk existen métodos para testear que un valor sea o no nil: *isNil, notNil.*
En algunos lenguajes es una palabra clave (no es instancia de una clase) y se denomina null.

Conceptos básicos: referencias

Los nombres representan *referencias* a objetos.
Puede haber varias referencias al mismo objeto.



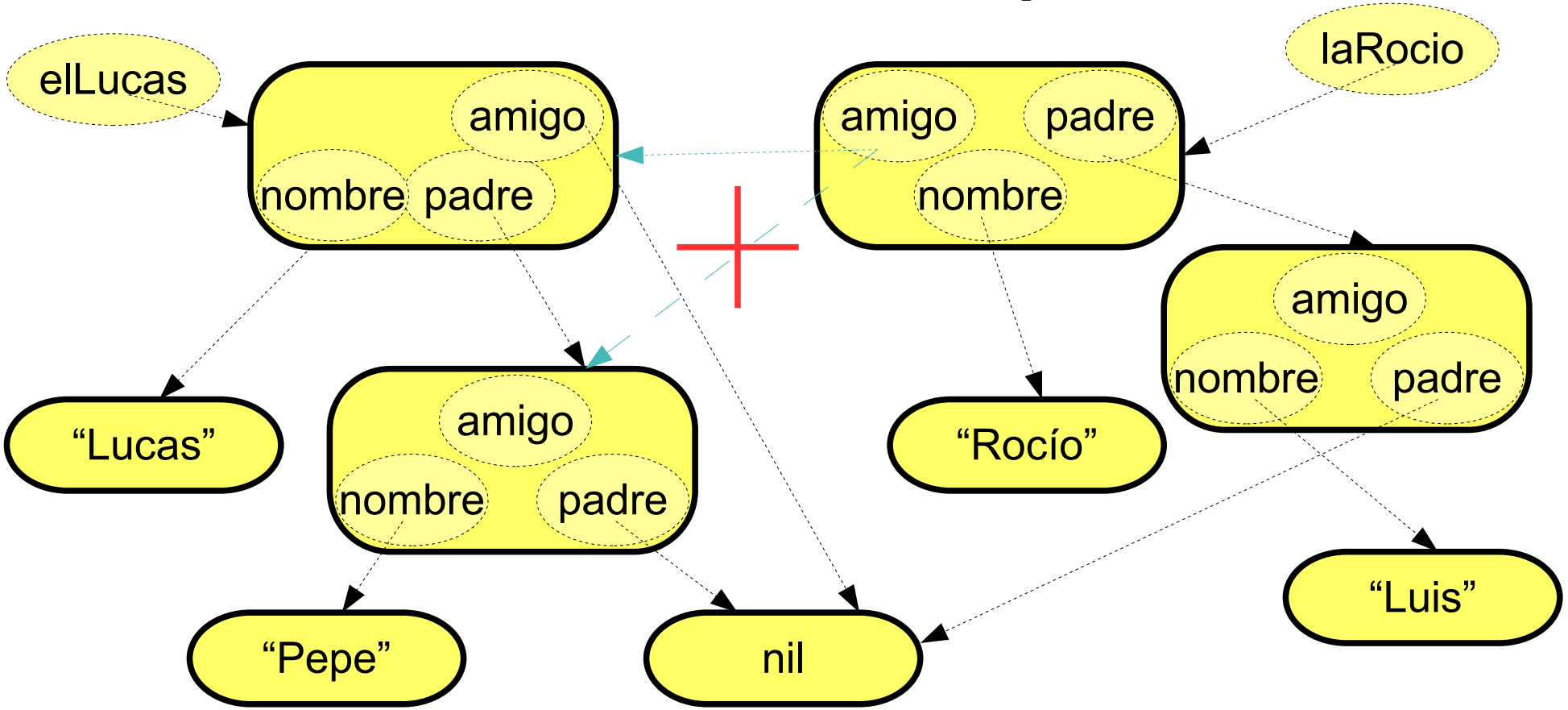
Qué pasa si se le cambia el padre a Pepe?.

Conceptos básicos: referencias

Asignación = cambio de referencia de un objeto a otro.

laRocio amigo: elLucas.

```
amigo: unaPersona
    amigo:= unaPersona
```



Conceptos básicos: mensajes de clase

En Smalltalk todo es objetos.
Las clases no pueden dejar de ser objetos!
La definición de su comportamiento es tema avanzado del curso.
Las clases pueden entonces recibir mensajes, de la misma forma que cualquier objeto.
Estos mensajes se pueden enviar sólo a la clase y no a sus instancias.

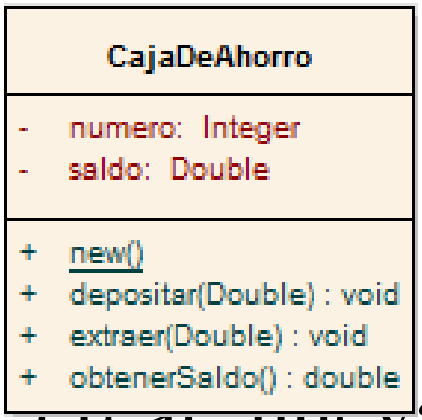
Al describir una clase, los agregaremos dentro del *protocolo,* subrayando la *signatura* del cuerpo.

Conceptos básicos: instanciación

¿Cuál es el comportamiento básico de toda clase?...

Disponemos de al menos un método en la clase para crear instancias de ella: new

```
objeto1 := CajaDeAhorro new.
```



El mecanismo por el cual se pueden crear nuevas instancias de una clase en particular se denomina *instanciación*.

Conceptos básicos: inicialización

Una instancia recién creada ¿está lista para colaborar con otros objetos? ¿una cuenta bancaria sabe su monto o su titular?

El proceso de inicialización provee a un objeto sus datos indispensables para no estar en un *estado inconsistente*.

En los casos que queremos impedir que esto suceda, debemos restringir la creación. ¿Cómo podemos lograrlo?

Conceptos básicos: constructores

Prohibir el uso del método new en esa clase.

Crear métodos de clase especiales, denominados *constructores*, que tomen todos los parámetros necesarios para crear una instancia bien formada.

Otra alternativa que a veces puede no resultar suficiente o adecuada es el uso de constructores que den valores por defecto a las variables.

Conceptos básicos: constructores

Ejemplo:

```
CajaDeAhorro >> new: unNumeroDeCuenta
                        propietario: unaPersona
    |cuenta|
    cuenta := CajaDeAhorro new.
    cuenta numero: unNumeroDeCuenta.
    cuenta propietario: unaPersona.
    cuenta saldo: 0.
    ^cuenta.
```

Conceptos básicos: garbage collection

Aquellos que hayan tomado el curso de **Introducción a la programación** previamente, habrán hablado quizás de posiciones de memoria, punteros y manejo de los mismos.

Esta tarea es compleja y propensa a errores.

Los lenguajes OO en general incluyen un mecanismo denominado *Garbage collection* que:

Detecta objetos que ya nadie conoce/referencia y libera la memoria que ocupan

Es automático y seguro

Es transparente al programador

Conceptos básicos: polimorfismo

Supongamos que tenemos modeladas las clases CajaDeAhorro y CuentaCorriente. Ambas clases definen el método *extraer: un Monto*. Instancias de estas clases podrán responder al mismo mensaje, aún respondiendo de modo diferente.

Estas dos instancias se dicen *polimórficas* respecto al mensaje *extraer: unMonto*. Son intercambiables.

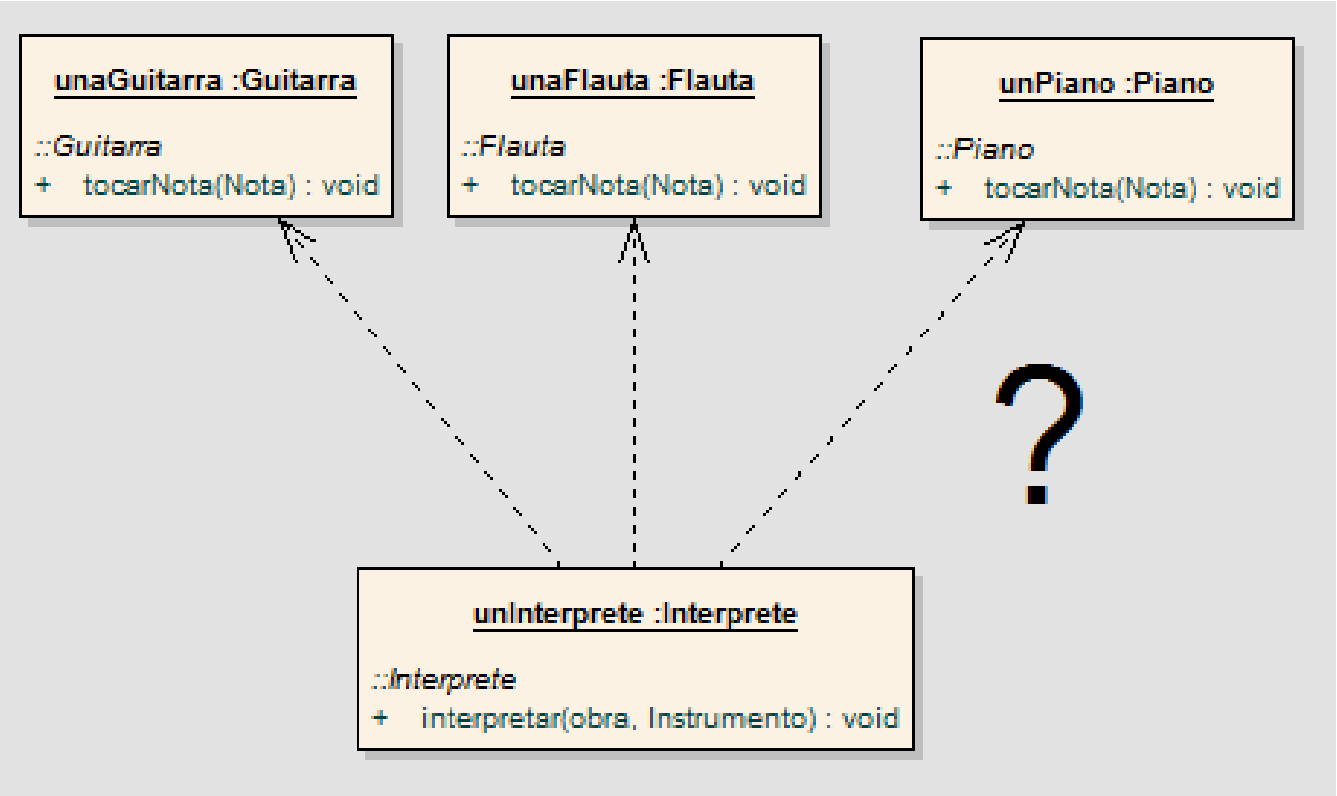
Polimorfismo significa la cualidad de tener más de una forma. En POO, el polimorfismo se refiere al hecho de que una misma operación puede tener diferente comportamiento en diferentes objetos.

Conceptos basicos: polimorfismo

Supongamos que existe una instancia de una clase Cajero, al que se le ordena quitar $100 de una cuenta del banco. Sabiendo que ambos tipos de cuentas responden al mensaje *extraer: unMonto*, al cajero no se interesará el tipo de cuenta de la que realiza la extracción y sólo mandará el mensaje. Sólo le interesa que sepa extraer!

"If it walks like a duck, and it talks like a duck, then it might as well be a duck"

Conceptos básicos: polimorfismo



Al intérprete no le importará el instrumento musical, dado que delegará en él la acción de tocar cada nota de la obra.

Conceptos básicos: polimorfismo

Analicemos el código siguiente:

```
(unaCuenta esCajaDeAhorro)
    ifTrue:[unaCuenta extraerDeCajaDeAhorro: 100]
    ifFalse:[unaCuenta extraerDeCuentaCorriente: 100]
```

¿Qué cambios habría que hacer si se crea una nueva cuenta?

¿y si se crean 20 más?

Esta opción sólo sirve cuando son comparaciones simples, y estamos seguros que no van a cambiar/evolucionar.

Conceptos básicos: polimorfismo

La opción adecuada es delegar en cada tipo de cuenta la forma en que extrae dinero. El código del cajero siempre será el mismo, e independiente de la cuenta:

```
unaCuenta extraer: unMonto
```

```
CuentaCorriente>> extraer: unMonto
    (self saldo - unMonto + self limite < 0)
        ifFalse: [ self saldo : self saldo - unMonto].
CajaDeAhorro>> extraer: unMonto
    (self saldo - unMonto < 0)
        ifFalse: [ self saldo : self saldo – unMonto].
```

Más adelante veremos una mejor solución con patrones...

Conceptos básicos: polimorfismo

¿Cuáles son entonces las ventajas del polimorfismo?
Código genérico
Objetos desacoplados
Objetos intercambiables
Objetos reutilizables
Programación por protocolo, no por implementación

Conceptos básicos: polimorfismo

¿Cuándo es adecuado usar “if” y cuando polimorfismo?

Cuando la condición a evaluar no depende del receptor del mensaje

Cuando la condición a evaluar no se ve alterada en el caso de agregar nuevos tipos de objetos

Se puede usar “if”

En caso contrario, la opción adecuada es el uso de polimorfismo.

Evitar preguntar de qué clase es un objeto (al menos hasta llegar al tercer curso con objetos)

Conceptos básicos: binding dinámico

El binding establece el momento en que se asocia un método a un envió de mensaje.
El binding *dinámico* permite tener polimorfismo, dado que el binding entre un método y el mensaje se realiza en run-time y puede cambiar.
Un envío de mensaje no se resuelve hasta el tiempo de ejecución, cuando se conoce cuál es realmente la clase del objeto receptor.
Más tarde veremos cómo se busca el método adecuado para un mensaje enviado.

Parte II

Conceptos avanzados

Conceptos avanzados: bloques en Smalltalk

Los bloques representan secuencias de instrucciones cuya ejecución es diferida.
Los bloques también son objetos (pertenecen a la clase BlockClosure). ¿Qué otra cosa podían ser?
Se notan escribiendo el código que los componen entre [].
El mensaje *value* pide a un bloque que se evalúe. El valor que devuelve un método, es el valor que devuelve la última sentencia
Un bloque puede contener una cantidad arbitraria de parámetros y declarar variables temporales.

Conceptos avanzados: bloques en Smalltalk

Ejemplos:

bloque := [1 / 0]. “no se ejecuta”

bloque value. “ahora se evalúa”

bloque := [:i :j | i+j]. “con parámetros”

bloque value:1 value:3. “se evalúa, pasándole los parámetros”

[5 * i]

En este caso, para evaluar el bloque, será necesario que en el *contexto* en que se ejecute, *i* esté definida.

Conceptos avanzados: sentencias de control en Smalltalk – if ~ else

¿Cómo se implementa la sentencia de control si se tiene sólo métodos y mensajes?

En la clase Boolean, se definen los métodos abstractos:

ifTrue: alternativeBlock

ifFalse: alternativeBlock

ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

Las subclases True y False dan una implementación concreta para cada método. Este es un ejemplo claro de polimorfismo.

También se incluyen *and: alternativeBlock* y *or: alternativeBlock*

Conceptos avanzados: sentencias de control en Smalltalk - for

Se expresa como dos métodos de Number:

to: stop do: aBlock

```
1 to: 3 do:[:i | ... ].
```

to: stop by: step do: aBlock

```
1 to: 20 by: 2 do:[:i | .... ].
```

Number construye primero una instancia de Interval y es el intervalo el que va pasando sus valores como parámetro al bloque

También en Integer el método:

timesRepeat:aBlock

Conceptos avanzados: sentencias de control en Smalltalk - while

Se expresa como métodos de un bloque.
whileTrue
whileTrue: aBlock
whileFalse
whileFalse: aBlock
Evalúa el receptor en cada iteración.
Ejemplo:

```
Transcript clear.
    i:=2.
    [i>0] whileTrue:
                [ Transcript show: i printString; cr. i := i - 1.].
```

Conceptos avanzados: colecciones

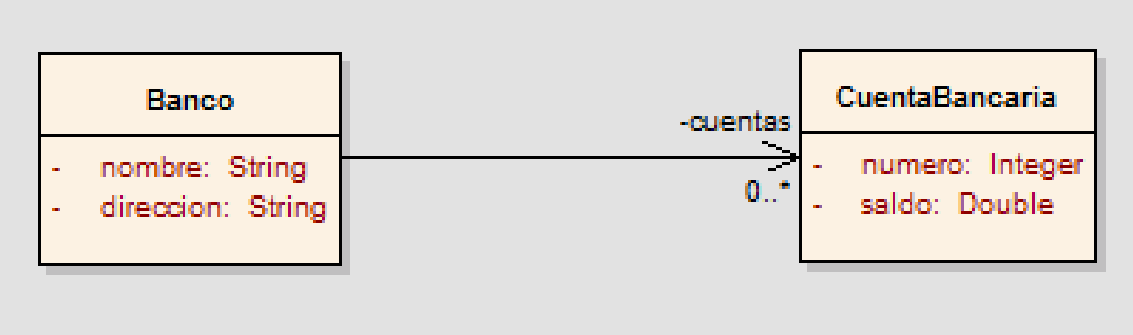
A veces se necesita conocer a muchos objetos bajo el mismo nombre. Ej: empleados de una empresa, canales de un televisor, documentos pendientes de una impresora, etc.

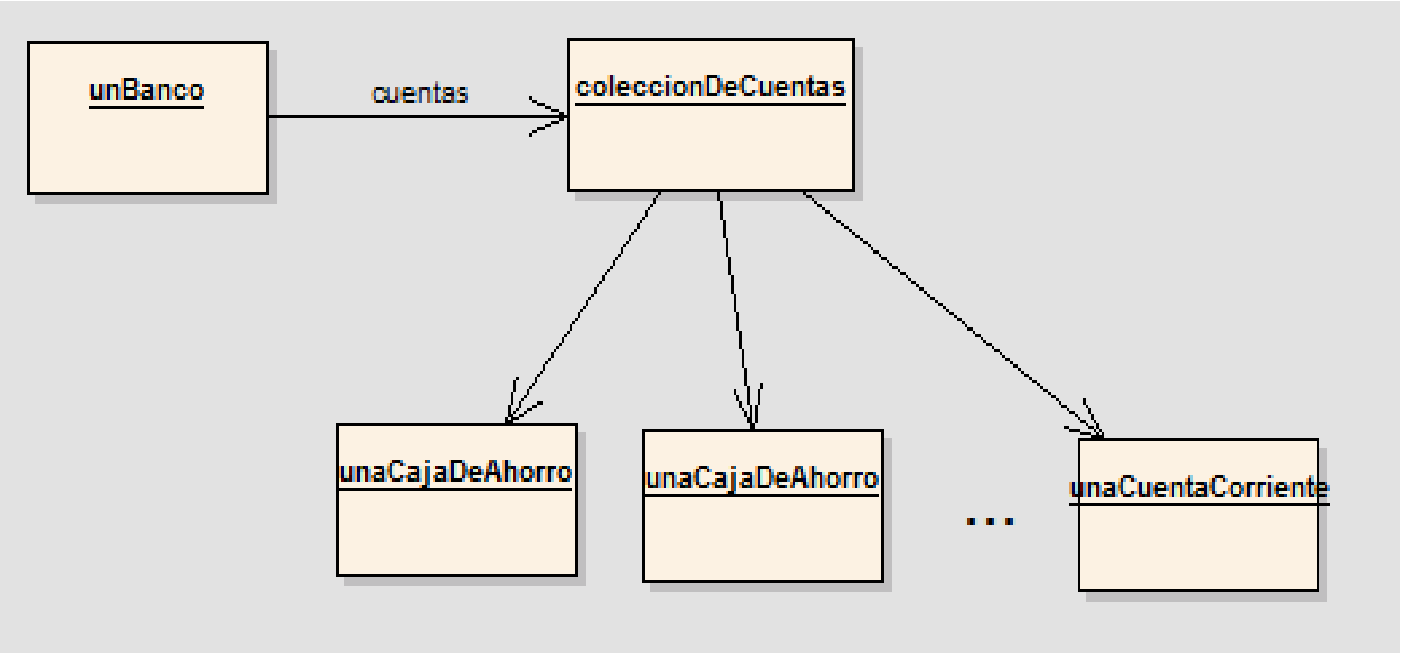
Las variables de instancia no alcanzan para representar relaciones *uno a muchos*.

Debemos modelizar colecciones de objetos... como objetos!

Las colecciones nos deben permitir agrupar un conjunto de objetos en una relación de conocimiento

Conceptos avanzados: colecciones





Conceptos avanzados: variantes de colecciones

Hay varias variantes de colecciones que nos pueden interesar

- ordenadas o no ordenadas
- con o sin repetidos
- etc ...

empecemos con una variante muy fácil …

Set = conjunto

Conceptos avanzados: usando colecciones

¿Qué puedo hacer con un Set, si es un objeto?

¡¡enviarle mensajes!!

p.ej. si tengo miConjuntito y quiero saber si está vacío … se lo pregunto usando un mensaje que entienden los conjuntos

miConjunto isEmpty

Conceptos avanzados: protocolo de conjuntos en Smalltalk

¿Qué se puede pedir a un conjunto?

>>size retorna el tamaño de la colección

>>isEmpty indica si la colección está vacía

>>notEmpty indica si la colección no está vacía

>> includes: anObject
indica si la colección incluye un cierto elemento

Conceptos avanzados: protocolo de conjuntos en Smalltalk

¿Cómo hago para agregar elementos a un conjunto?
¡Más mensajes!

>>add:	le agrega un elemento
>>remove:	le saca un elemento

Las colecciones en Smalltalk son *heterogéneas*, es decir, podemos poner cualquier *tipo* de objeto como elemento.
En algunos lenguajes OO las colecciones son *homogéneas*, por lo que se restringe el *tipo* de elementos que se puede agregar.

Conceptos avanzados: colecciones y referencias

Si quiero representar un conjunto con tres personas ¿cuántos objetos necesito?

¿Está bien decir que un conjunto *tiene adentro* a sus elementos?

¿Puede ser que un mismo objeto sea elemento de dos conjuntos? (p.ej. una persona que cursa dos materias)

Si pienso en las referencias, los conjuntos ¿conjuntos *de qué* son?

Conceptos avanzados: iteradores de colecciones

Todo lindo, pero … ¿cómo hago para *acceder* a los elementos de un conjunto?

```
for (i = 0; i < miConjunto size; i++) {
  ...hacer algo sobre miConjunto[i]...
}
```

De esto se va a encargar el conjunto, yo le tengo que decir qué va a hacer con cada elemento

```
miConjunto do:
  [:elem |...lo que quiera hacer con elem...]
```

Conceptos avanzados: iteradores de colecciones en Smalltalk

do: aBlock

recibe un bloque con un parámetro y evalúa el argumento aBlock para cada uno de los elementos del receptor

```
miConjunto do: [ :i | Transcript show: 'mostrando ', i printString].
```

select: aBlock

recibe un bloque con un parámetro, evalua el argumento aBlock para cada uno de los elementos del receptor generando una nueva coleccion del mismo tipo que el receptor con únicamente los elementos para los cuales aBlock evalua a true

```
miConjunto select: [ :i | i even.].
```

Conceptos avanzados: iteradores de colecciones

reject: aBlock

Idem select:, pero el valor del bloque es false

```
miConjunto reject: [ :i | i even.].
```

collect: aBlock

recibe un bloque con un parámetro, evalua el argumento aBlock para cada uno de los elementos del receptor generando una nueva coleccion del mismo tipo que el receptor, y para cada elemento del receptor agrega el resultado de aplicarle aBlock

```
miConjunto collect: [:i | i+1.].
```

Conceptos avanzados: iteradores de colecciones

detect: aBlock

recibe un bloque con un parámetro, evalua el argumento aBlock para cada uno de los elementos del receptor y devuelve el primer elemento para el cual aBlock evaluó en true. Si ninguno da true, error

```
miConjunto detect: [ :i | (i > 1) and: [i even].]
```

inject: thisValue into: binaryBlock

recibe un bloque con dos parámetros. El primer parámetro es el resultado de la evaluacion anterior, empezando por thisValue. El segundo parámetro es cada uno de los elementos del receptor. Devuelve el resultado de la última evaluacion.

```
miConjunto inject:0 into: [ :i :j | i + j].
```

Conceptos avanzados: subclasificación y generalización

Durante el curso identificamos dos tipos de cuentas bancarias, y vimos que tenían comportamiento en común (protocolo) e incluso variables de instancia.
Podríamos pensar en un nivel de abstracción mayor, que generalice estos elementos en común entre clases:

subclasificación y generalización

Conceptos avanzados: subclasificación y generalización

Se reúne el comportamiento y estructura interna común en una clase, que cumplirá el rol de *superclase*. Las clases que *heredan* lo que la superclase define serán *subclases* de la clase.

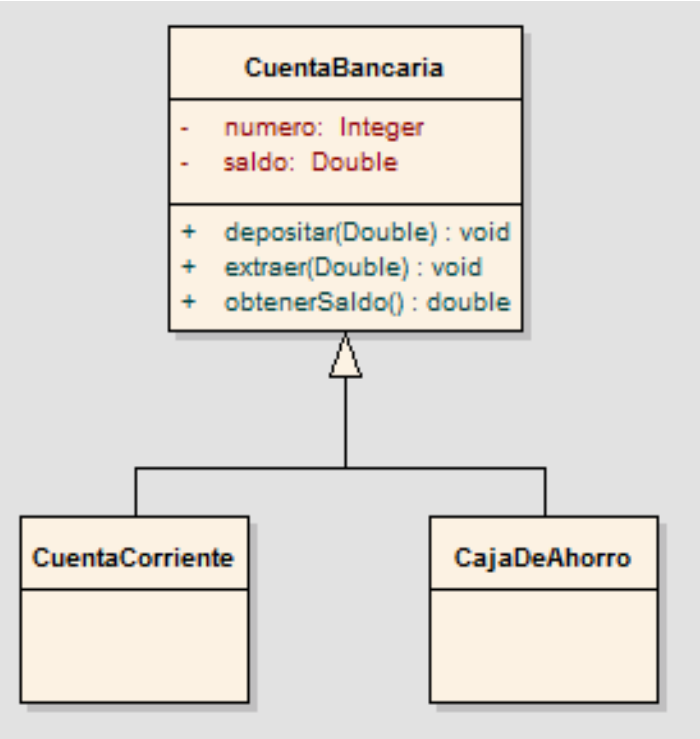
Las subclases *heredan* el comportamiento y estructura interna de la superclase, y pueden definir más comportamiento y nuevas v.i.

Se establece una relación *es-un* entre subclasesy superclase.

La superclase es una *generalización* de sus subclases.

Las subclases son una *especialización* de la superclase

Conceptos avanzados: subclasificación y generalización



CuentaCorriente y CajaDeAhorro son subclases de CuentaBancaria, heredando todo lo que ella defina.
Algunos lenguajes permiten especificar qué partes se pueden heredar.

Conceptos avanzados: definición de subclase

Para indicar que una clase extiende otra, tomaremos como convención la siguiente notación:

NombreDeClase

Superclass: nombreSuperclase

Variables de instancia

variable_1, variable_2,..., variable_n

Protocolo

>> metodo_1

Conceptos avanzados: jerarquía de clases

La relación *es-un* establecida entre clases, genera una *jerarquía de clases*.
En lenguajes como Smalltalk o Java, la raíz de esa jerarquía es la clase Object, que tiene métodos de instancia básicos de todo objeto como pueden ser para comparación por igualdad y por identidad.
Toda clase formará parte de la jerarquía de Object y extenderá de ella, directa o indirectamente.

Conceptos avanzados: herencia

Es el mecanismo por el cual las subclases reutilizan el comportamiento y estructura en su superclase.
La herencia permite
Crear una nueva clase como *refinamiento* de otra.
Diseñar e implementar sólo la diferencia que presenta la nueva clase.
Abstraer las similitudes en común.

Conceptos avanzados: herencia múltiple

Algunos lenguajes permiten que una clase “cuelgue” de más de una superclase. En ese caso, no tendremos jerarquía de clases (ref. C++).

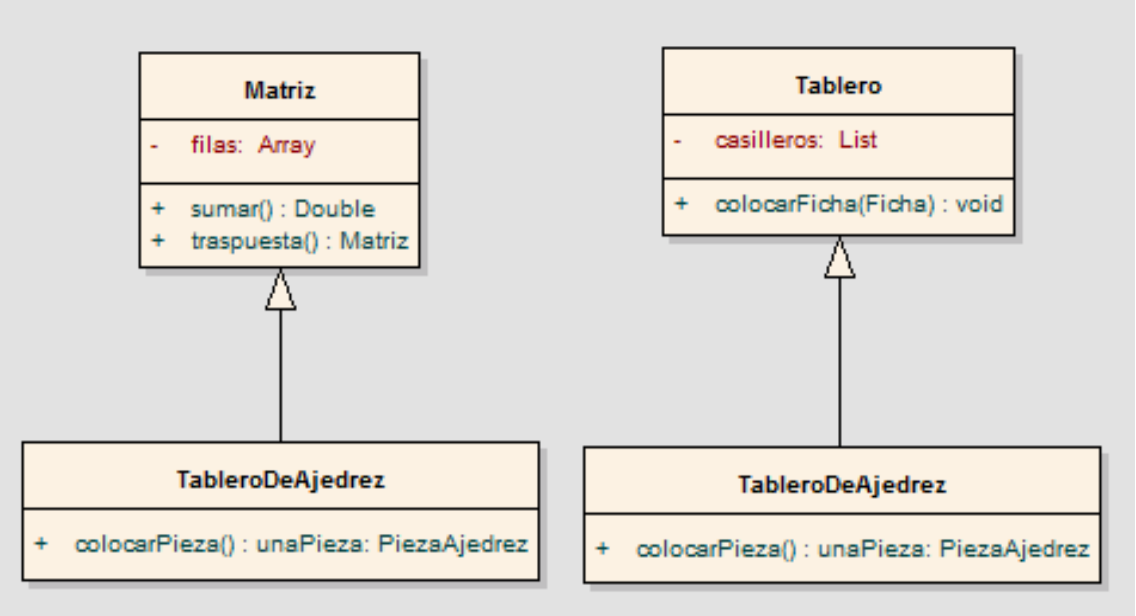
La clase hereda comportamiento y estructura de todas las superclases.

¿Qué pasa si se heredan dos métodos o variables de instancia con el mismo nombre?

Generalmente es mejor evitarla

Conceptos avanzados: herencia

¿Cuál de las dos alternativas es más adecuada? ¿Cuál respeta la relación *es-un*?

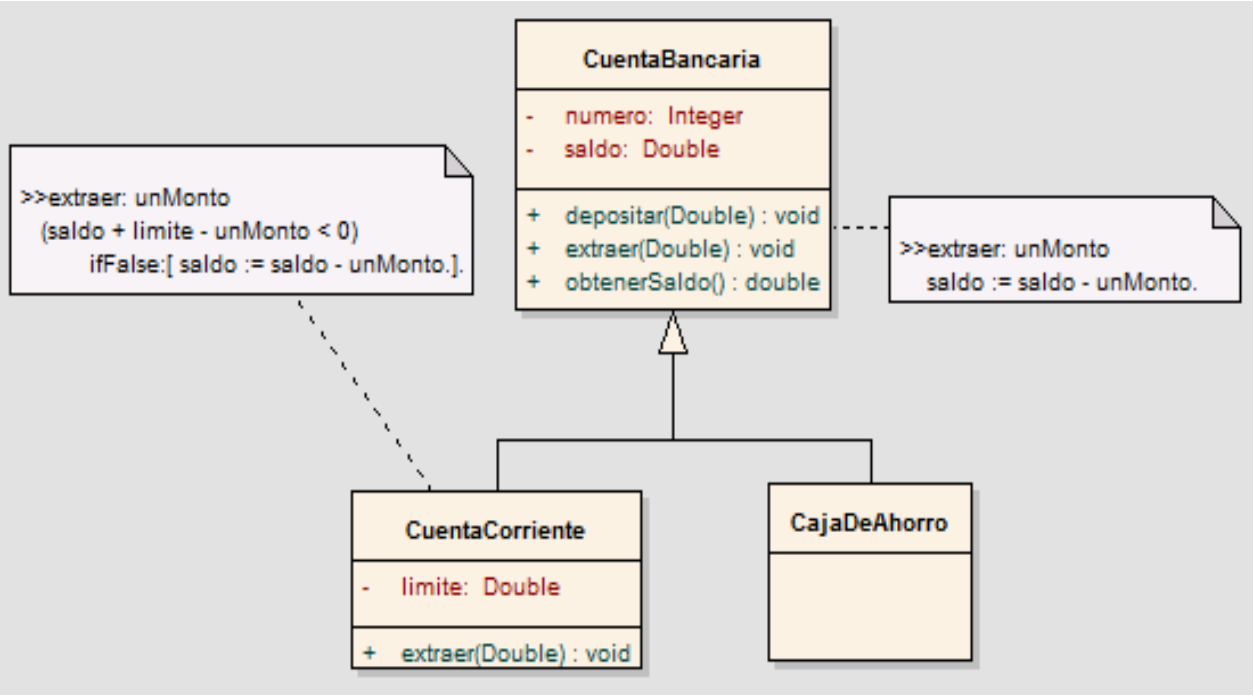


La herencia por estructura no es adecuada en general, dado que se puede heredar comportamiento no deseado

Conceptos avanzados: herencia y redefinición de métodos

Cuando se hereda estructura, no se puede redefinir (cambiar). ¿Qué pasa con el comportamiento?
¿Qué pasa si toda cuenta bancaria sabe extraer, pero las instancias de CuentaCorriente tienen límite de extracción?
Necesitamos *redefinir* el método *extraer: un monto* en la clase CuentaCorriente, reutilizando las características comunes de la extracción de dinero de las cuentas bancarias.

Conceptos avanzados: herencia y redefinición de métodos



Definimos un atributo *limite* y redefinimos el método *extraer: unMonto* en la clase CuentaCorriente, para obtener el comportamiento deseado. Notar que no alteramos la *signatura* del método heredado.

Conceptos avanzados: pseudo-variable 'super'

En el ejemplo anterior redefinimos completamente el método *extraer*: *unMonto*, desechando la parte en común que tiene la extracción de dinero de las cuentas bancarias.

¿Y si quisiéramos reutilizar el comportamiento que habíamos heredado pero redefinimos?

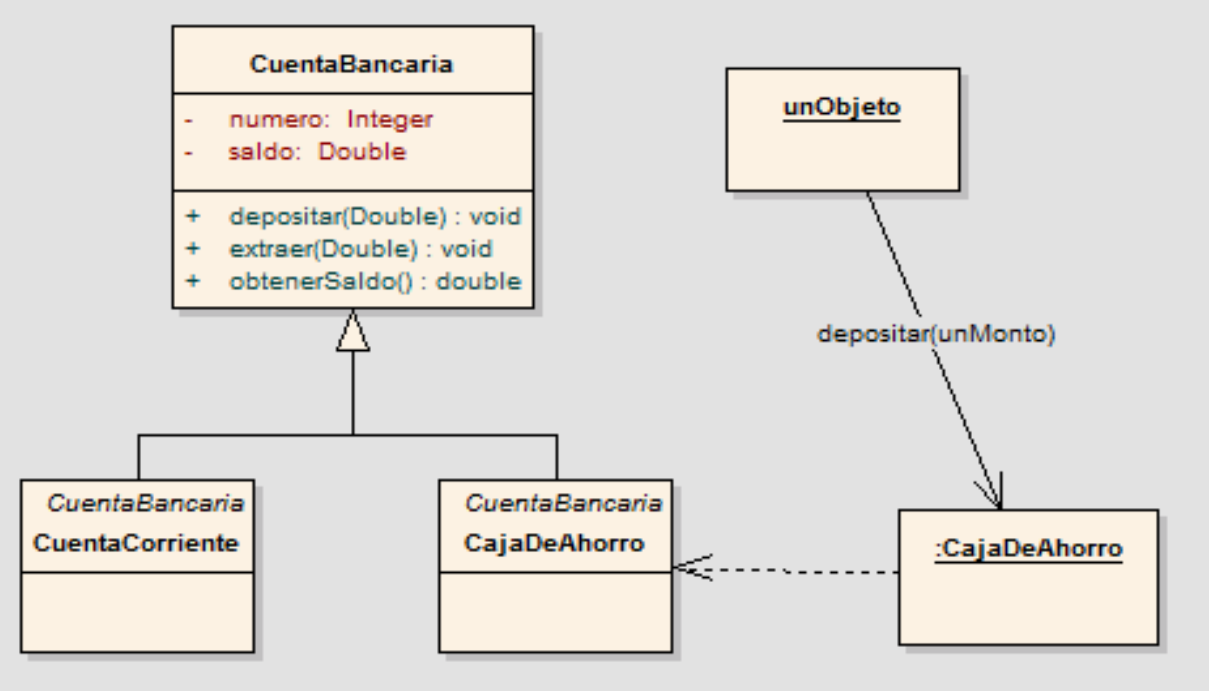
Podemos hacer referencia al método de la superclase con la pseudo-variable *super*.

```
>>extraer: unMonto
   (saldo - unMonto < 0)
     ifFalse:[ super extraer: unMonto.].
```

Conceptos avanzados: pseudo-variable 'super' y method lookup

La pseudo-variable *super*, NO hace que el receptor del mensaje cambie. Sólo cambia el lugar desde donde se empieza el *method lookup*.
El method lookup es el mecanismo que se activa cuando un objeto de una clase recibe un mensaje. Este mecanismo se lleva a cabo buscando en la clase del receptor el método asociado al mensaje. En caso de no encontrarse, se busca una definición yendo 'hacia arriba', en la superclase. Si tampoco se encuentra una definición, se seguirá recursivamente.

Conceptos avanzados: pseudo-variable 'super' y method lookup



Conceptos avanzados: pseudo-variable 'super' y method lookup

En caso de no encontrarse una definición de método para el mensaje recibido, se producirá una *excepción*.
Esta forma de buscar los métodos prioriza las definiciones en la subclase sobre las heredadas.
Enviar el mensaje a *super* permite "saltear" la clase actual donde está definido el método, en la búsqueda "hacia arriba" del método adecuado. **No se refiere a otro objeto.**
No necesariamente es la clase del objeto receptor sino que puede ser una superclase de ella.

Conceptos avanzados: method look-up y pseudo-variables

Cuando se envía un mensaje a *self*, el method look-up comienza desde la clase a la que pertenece la instancia.
Cuando se envía un mensaje a *super*, el method look-up comienza desde la clase en la que el método está definido (que no necesariamente es la clase del objeto sino una superclase).

Conceptos avanzados: redefinición del método new

Podemos redefinir el método de clase *new*, de forma tal que creamos la instancia de acuerdo al comportamiento heredado, pero además inicializamos el objeto con valores *default* para sus variables de instancia (entre otras cosas posibles).

Ejemplo:

TableroDeAjedrez >> new

"En el método de instancia *initialize*, inicializamos la matriz que usaremos como colaborador."

^super new initialize.

Conceptos avanzados: redefinición del método new

En Smalltalk cuando se crea una clase se (re)define por defecto (no es necesario) el método new. Ej:

```
Punto>>new
"Answer a newly created and initialized instance."
        ^super new initialize
Punto >>initialize
"Initialize a newly created instance. This method must answer the
   receiver."
" *** Edit the following to properly initialize instance variables ***"
        x := nil.
        y := nil.
        " *** And replace this comment with additional initialization code
          *** "
        ^self
```

Conceptos avanzados: redefinición del método new

Podemos redefinir el método de instancia para poner valores default:

```
Punto >>initialize
    self x : 0.
    self y : 0.
    ^self
```

Podemos también anular el new para evitar el uso de este contructor:

```
Punto>>new
    ^self error: 'No se pueden crear instancias con este contructor'
```

Pero...

Conceptos avanzados: redefinición del método new

... debemos definir otra forma de construir instancias para esa clase:

```
Punto>>newX: cordenadaX y: cordenadaY
    ^super new initializeX: cordenadaX y: cordenadaY
```

Por qué es necesario llamar a *super?*
Notar que ahora debemos quitar el método *initialize* pero tenemos que definir un nuevo inicializador:

```
Punto>>initializeX: cordenadaX y: cordenadaY
    ^self x: cordenadaX; y: cordenadaY; yourself.
```

Conceptos avanzados: redefinición del método new - reuso

Y si tuviésemos una subclase Punto3D, cómo creamos instancias?
El newX:y: no lo queremos, tienen que pasarle las 3 coordenadas

```
Punto3D>>newX: cordenadaX y: cordenadaY
    ^self error: 'No se pueden crear instancias con este contructor'
```

Lo correcto sería definir:

```
Punto3D>>newX: cordenadaX y: cordenadaY z: cordenadaZ
```

Cómo definirlo?

```
Punto3D>>newX: cordenadaX y: cordenadaY z: cordenadaZ
    | instancia |
    instancia:= super newX: coordenadaX y: cordenadaY.
    instancia z: coordenadaZ
    ^instancia
```

Conceptos avanzados: redefinición del método new - reuso

Si no hubiésemos anulado el new, también podemos definirlo:

```
Punto3D>>newX: cordenadaX y: cordenadaY z: cordenadaZ
    ^super new initializeX: cordenadaX y: cordenadaY z:
      cordenadaZ
```

Además deberíamos definir:

```
Punto3D>>initializeX: cordenadaX y: cordenadaY z:
  cordenada>
    ^self x: cordenadaX; y: cordenadaY; z: cordenadaZ;
  yourself.
```

Y redefinir el initialize heredado:

```
Punto3D>>initialize
  ^self x: 0; y: 0; z: 0;yourself.
```

Conceptos avanzados: igualdad e identidad

En Smalltalk, el mensaje == permite verificar que un objeto receptor es *idéntico* a otro dado.
El mensaje = permite verificar que el objeto receptor es *igual* a otro.
Para el segundo mensaje (heredado de Object también) puede darse una definición particular para la clase, de acuerdo a las necesidades.
Ej:

```
Persona>> = otraPersona
    ^persona nombre = otraPersona nombre
```

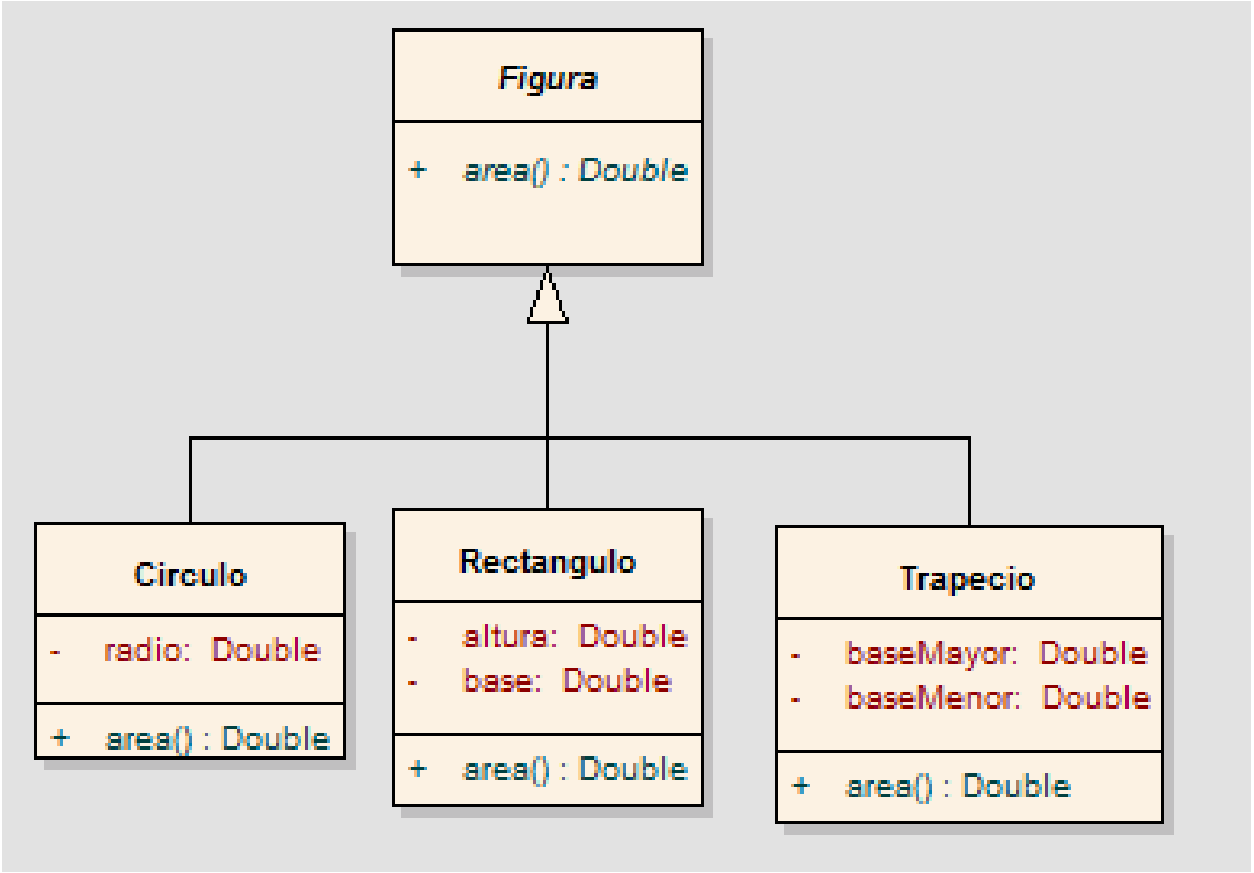
Conceptos avanzados: método abstracto

Son métodos que no especifican comportamiento y son definidos en clases que son raíz de una subjerarquía de clases.
Permiten establecer un protocolo común en la jerarquía.
Obligan a toda subclase a dar una implementación concreta del método.
En Smalltalk, un método abstracto debe tener sólo el código *self subclassResponsibility*.
Se identifican con letra *cursiva* en los diagramas.

Conceptos avanzados: clase abstracta

Son clases a partir de las cuales no pueden crearse instancias.
Sirven para definir métodos en común de las subclases y para definir estructura interna que heredarán las subclases.
Pueden tener métodos abstractos, aunque no es requisito.
El nombre se escribe en *cursiva* en los diagramas.
Nosotros indicaremos que son abstractas con un *.
NombreDeClase *
¿Qué pasa si una clase tiene un método abstracto?

Conceptos avanzados: clases abstractas y métodos abstractos




```
Figura>> area
   ^self subclassResponsibility
```

Conceptos avanzados: double dispatching

Problema 1: modelar el juego de piedra-papel-tijera

En este juego intervienen dos jugadores, los cuales tienen cada uno tres opciones para elegir por jugada. El problema consiste en tomar dos jugadores, pedirles a cada uno que elija una opción, y determinar quién fue el ganador.
Para quienes nunca jugaron este juego (?), las opciones y ganador son...

Conceptos avanzados: double dispatching

Jugador 1	Jugador 2	Ganador
PIEDRA	PIEDRA	EMPATE
PIEDRA	PAPEL	JUGADOR 2
PIEDRA	TIJERA	JUGADOR 1
PAPEL	PIEDRA	JUGADOR 1
PAPEL	PAPEL	EMPATE
PAPEL	TIJERA	JUGADOR 2
TIJERA	PIEDRA	JUGADOR 2
TIJERA	PAPEL	JUGADOR 1
TIJERA	TIJERA	EMPATE

Debemos modelar el juego, y en particular el mensaje:
Juego>> hacerJugar: unJugador contra: otroJugador
“retorna el jugador ganador o nil si hubo empate.”

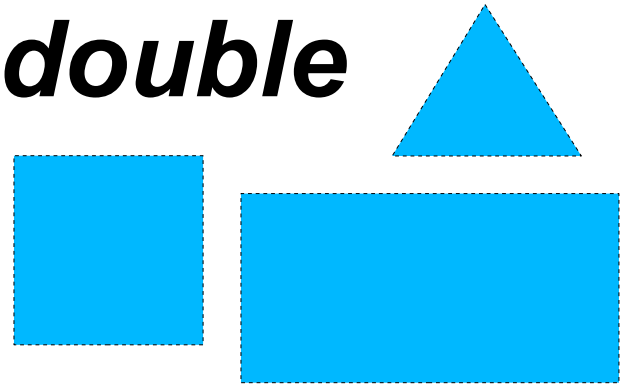
Conceptos avanzados: double dispatching

```
Juego>> hacerJugar: unJugador contra: otroJugador
        ^(unJugador eleccion = otroJugador eleccion)
            ifTrue:[^nil].
        (unJugador eleccion ganaA: otroJugador eleccion)
            ifTrue:[^unJugador]
            ifFalse:[^otroJugador].
```

¿Cómo resolvemos el problema de determinar si la elección del primer jugador le gana a la elección del segundo jugador?

¿De quién depende el resultado del mensaje “ganaA:” definido para las elecciones?

Conceptos avanzados: double dispatching



Problema 2: fusión de figuras

Se tiene un editor gráfico que puede fusionar automáticamente distintas figuras geométricas: cuadrados, rectángulos, triángulos isosceles y polígonos compuestos. Cada figura modelada como objeto entiende el mensaje *fusionarCon: otraFigura.* Algunas reglas:

Fusionar dos cuadrados retorna un rectángulo
Fusionar dos triángulos retorna un cuadrado
Fusionar un triángulo y un rectángulo retorna un polígono irregular
Fusionar dos rectángulos retorna un nuevo rectángulo.
etc.

Conceptos avanzados: double dispatching

Solución posible para problema 1:

Preguntar por la clase del argumento

```
Piedra>> ganaA: otraEleccion
    (otraEleccion isKindOf: Papel) ifTrue:[ ^false].
    (otraEleccion isKindOf: Tijera) ifTrue:[ ^true]
```

Desventajas de esta solución:

¿Qué pasa si quisieramos definir pierdeCon:?

¿Y si aparece un nuevo tipo de elección – ej: bolsa?

Conceptos avanzados: double dispatching

Solución posible para problema 2:

Preguntar por la clase del argumento

```
Triangulo>> fusionarCon: otraFigura
    (otraFigura class == Triangulo) ifTrue:[...]
    (otraFigura class == Cuadrado) ifTrue:[...]
    (otraFigura class == Rectangulo) ifTrue:[...]
    (otraFigura class == Poligono) ifTrue:[...]
```

Tenemos las mismas desventajas!

Conceptos avanzados: double dispatching

Solución: double dispatching

Usar la información dada por el primer mensaje enviado y hacer un segundo envío de mensaje para el argumento.

En el primer método (correspondiente al primer mensaje) se envía un mensaje al parámetro, “codificando” el tipo del receptor original en el nombre del mensaje y eventualmente el receptor original.

El segundo método sabe el tipo del argumento (pues lo tiene “codificado” en el método mismo) y el tipo del receptor

El segundo método está en condiciones de resolver el problema ya que sabe el tipo de ambos objetos.

Conceptos avanzados: double dispatching

Al recibir el mensaje, el receptor manda un mensaje al parámetro recibido, indicándole explícitamente su clase

```
Piedra >> ganaA: otraEleccion
            ^otraEleccion leGanaAPiedra
        >> leGanaAPiedra
            ^false
        >> leGanaATijera
            ^true
```

```
Tijera>> ganaA: otraEleccion
            ^otraEleccion leganaATijera
            >> leGanaATijera
                        ...
>> leGanaAPiedra
    ^false
```

El argumento al recibir el mensaje, sabe la clase del parámetro (objeto receptor original) y la suya, y puede realizar el comportamiento esperado.

Conceptos avanzados: double dispatching

Y con la fusión de figuras:

```
Triangulo>> fusionarCon: otraFigura
   otraFigura fusionarTriangulo: self.

Triangulo >> fusionarTriangulo: unTriangulo
     ^(self altura = unTriangulo)
              ifTrue: [ ^Cuadrado  newConAltura: self
   altura ]
              ifFalse: [^Poligono new ....]
Cuadrado >> fusionarTriangulo: unTriangulo
         ^Poligono new ....
```

Conceptos avanzados: double dispatching

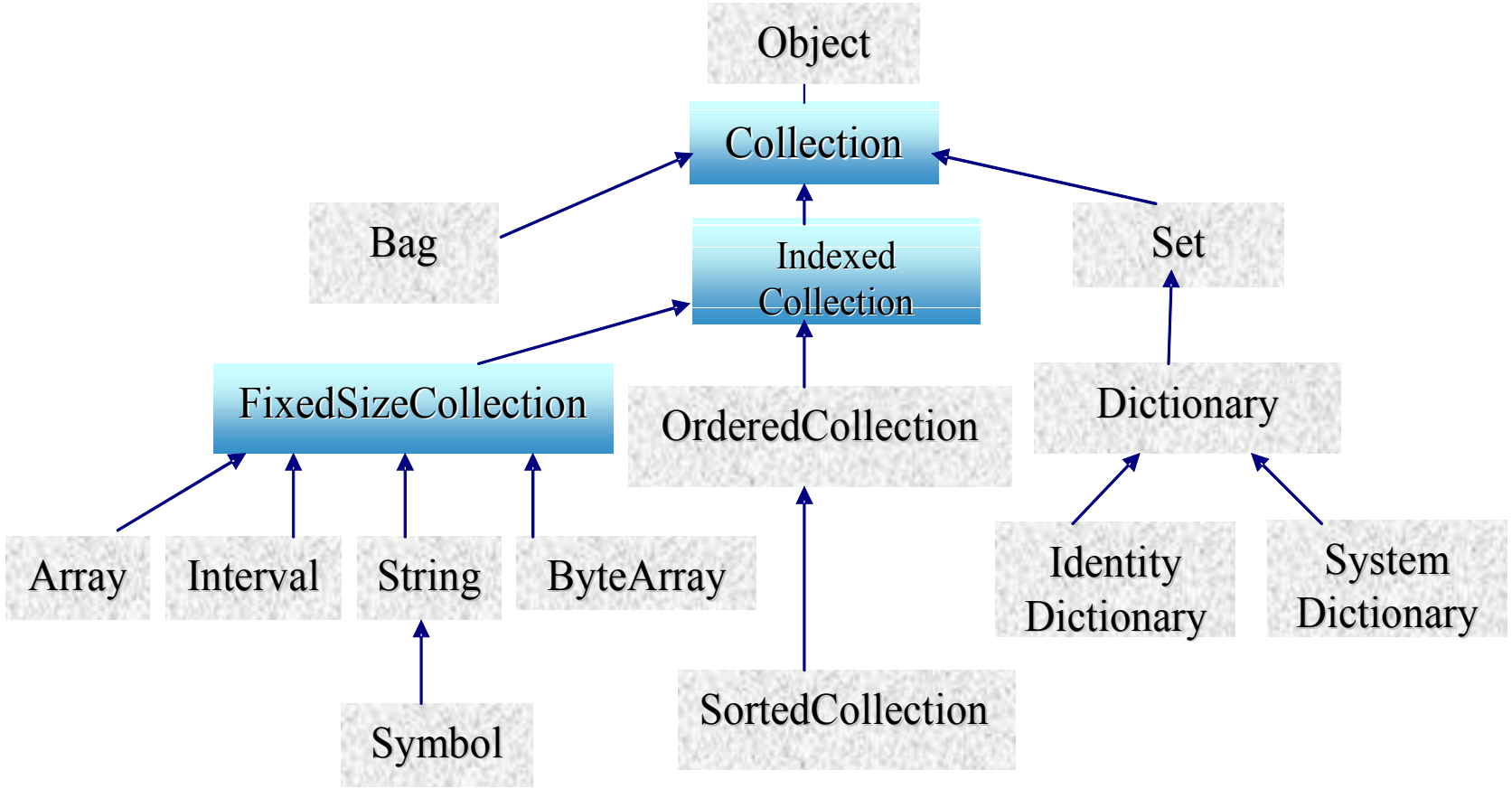
Técnica usada cuando el comportamiento de un objeto ante un mensaje no depende sólo de su clase, sino de la clase del parámetro que se recibe.
Evita preguntar por la clase del receptor
Como desventaja, si aparece una clase más, hay que escribir un método más en las demás clases que se pueden recibir como parámetro.
Primer ejemplo de *patrón de diseño* que vemos en el curso.

Conceptos avanzados: variantes en colecciones

Hay varios tipos de colecciones:
Ordenadas/sin orden
De tamaño fijo/variable
Acceso por posición/nombre

Esto se logra teniendo distintas clases, una para cada tipo de colección. Para tener una colección de las características que quiero, debo instanciar la clase correspondiente.
Las clases que modelan colecciones están organizadas usando el concepto de herencia.

Conceptos avanzados: colecciones en Smalltalk



Conceptos avanzados: variantes en colecciones

Recordamos que

Las colecciones en Smalltalk son *heterogéneas*, es decir, podemos poner cualquier *tipo* de objeto.
En algunos lenguajes OO las colecciones son *homogéneas*, por lo que se restringe el *tipo* de elementos que se puede agregar.

Esto es así para todos los tipos de colecciones.

Conceptos avanzados: colecciones en Smalltalk

Array: se usa cuando la cantidad (máxima) de elementos de la colección se conoce a priori. Es una colección de acceso por índice numérico.

Ej: asientos de un vuelo

Dictionary: es una colección de pares clave-valor (instancias de Asociation) . La clave no puede ser nil. Cada clave tiene asociado un único valor.

Ej: índice telefónico

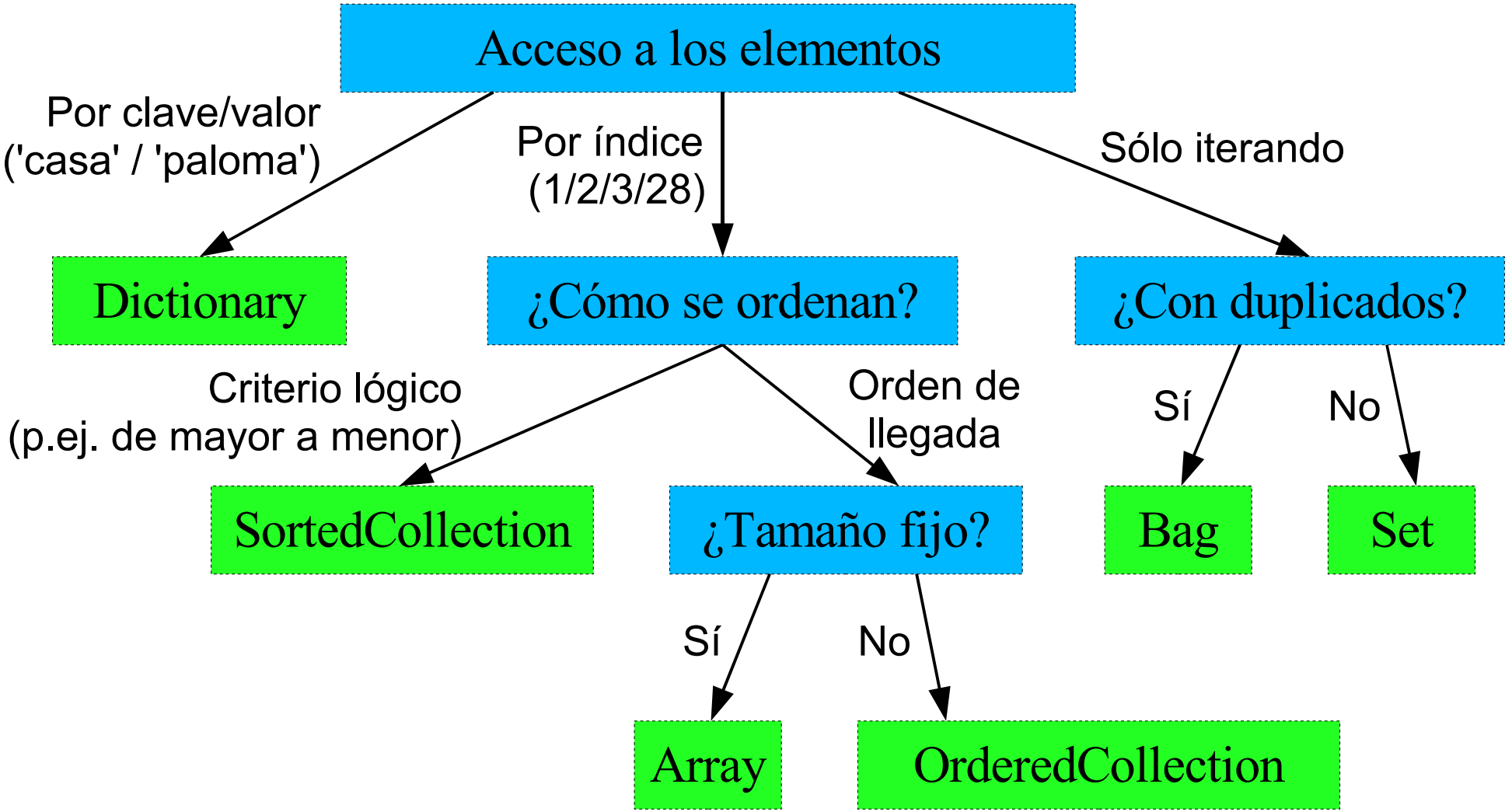
Bag: se usa cuando no importa el orden en que se agregan los elementos. Admite duplicados.

Ej: bolsa de compras

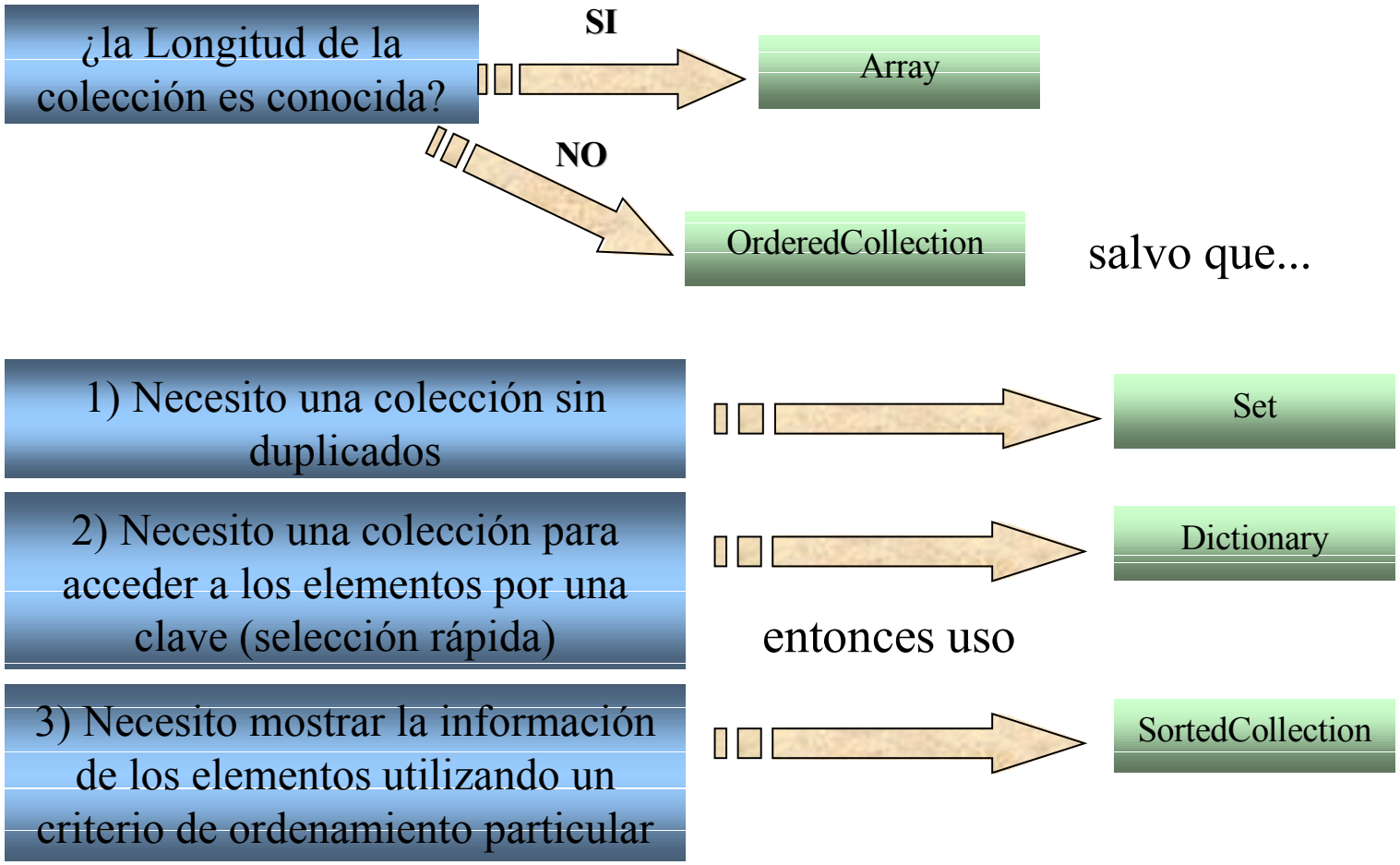
Conceptos avanzados: colecciones en Smalltalk

Set: representa los conjuntos como se entienden matemáticamente. No admiten duplicados.
OrderedCollection: es una colección que podemos ver como lista doblemente enlazada. Tiene mayor flexibilidad que un array, al cambiar su tamaño dinámicamente. Es la opción default para colecciones.
SortedCollection: una colección ordenada por algún orden dado (o el natural, <).
Interval: una colección que representa un rango de números y un intervalo para recorrerlo.

Conceptos avanzados: selección de la colección adecuada (I)



Conceptos avanzados: selección de la colección adecuada (II)



Conceptos avanzados: protocolo de colecciones en Smalltalk

Algunos mensajes son solamente para algunos tipos de colecciones

>>add:

Para Dictionary no sirve, porque hay que indicar el valor.
Para Array no sirve, porque no se puede agregar, sólo cambiar.
En ambos casos, usar at:put:

>>at:

Devuelve el elemento correspondiente a una clave o índice.
No tiene sentido ni para Bag ni para Set.

Conceptos avanzados: protocolo de colecciones en Smalltalk

Otros mensajes útiles

>>ocurrencesOf: anObject

Retorna la cantidad de ocurrencias de un objeto

>>asXYZ

Retorna una nueva colección de tipo XYZ con los mismos elementos que el receptor

P.ej. `unaOrderedCollection asSet`

P.ej. `UnaOrderedCollection asArray`

Conceptos avanzados: clases y métodos abstractos en Collection

La clase Collection y el método do: son abstractos en Smalltalk.

Ventaja: definición común de varios métodos, p.ej. select:

```
select: aBlock
    | newCollection |
    newCollection := self species new.
    self do: [:each |
        (aBlock value: each)
        ifTrue: [newCollection add: each]
    ].
    ^newCollection
```

Conceptos avanzados: reutilización

Existen diversas formas de reutilización

Herencia (ya fue vista)

Delegación

Colaboración

Composición

Conceptos avanzados: delegación

Toda la responsabilidad de la ejecución de una tarea se delega a otro objeto conocido que *ya sabe* hacerla.

```
Archivo >> imprimir
                (self getImpresora) imprimir: self.
```

La delegación es adecuada cuando la tarea a realizar no está dentro de las *responsabilidades* del objeto, y otro objeto sabe hacerla.

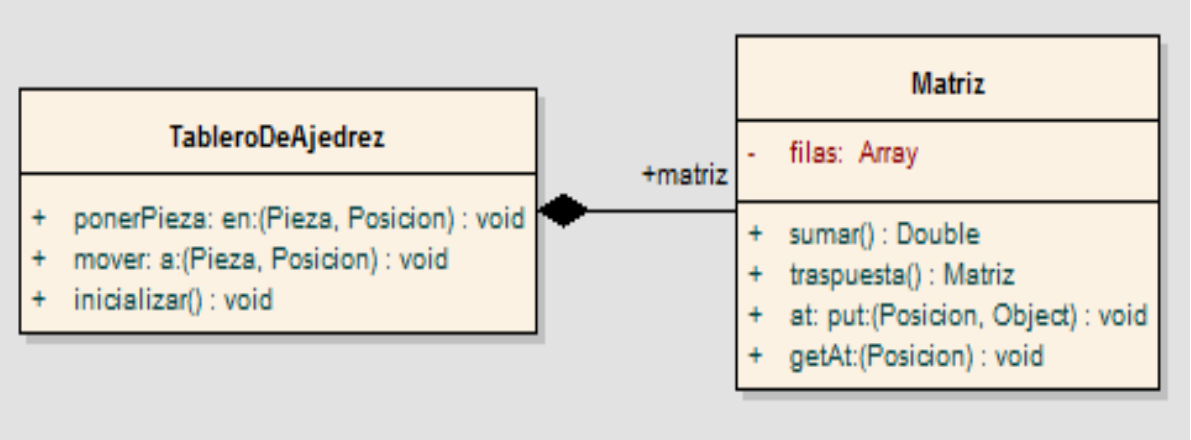
Conceptos avanzados: colaboración

El objeto delega parte de la resposabilidad en uno o más objetos
Nunca pierde el control de la ejecución

```
ReproductorCD>> reproducir: unCD
                        numeroTrack: unNumero
        |track|
        track :=  self getTracker buscarTrack: unNumero
    en: unCD.
        self reproducirTrack: track.
```

Conceptos avanzados: composición

Para implementar las responsabilidades del TableroDeAjedrez, usamos una Matriz.



La composición es reutilización *de caja negra*.
Modela la relación *tiene-un*.
Observar la notación usada.

Conceptos avanzados: composición vs. herencia

La composición se define al nivel de instancia, y de forma dinámica en run-time.

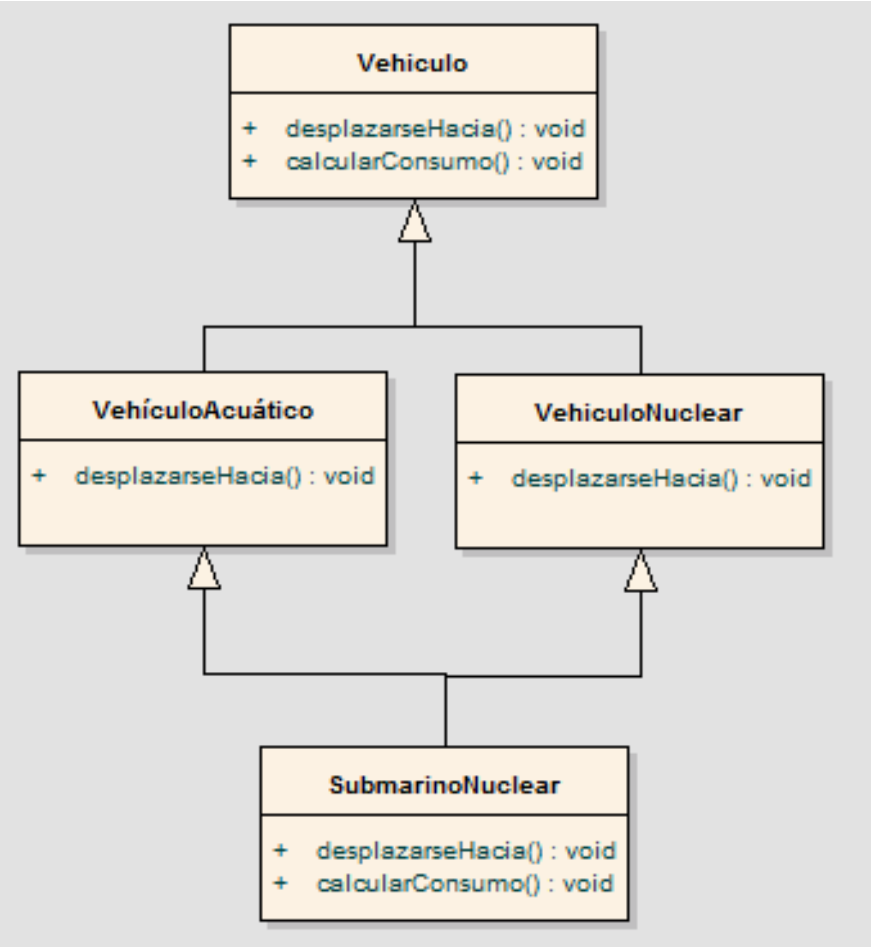
A diferencia de la herencia, no viola el encapsulamiento.

La herencia es una forma de reutilización denominada *de caja blanca*.

La herencia se define estáticamente.

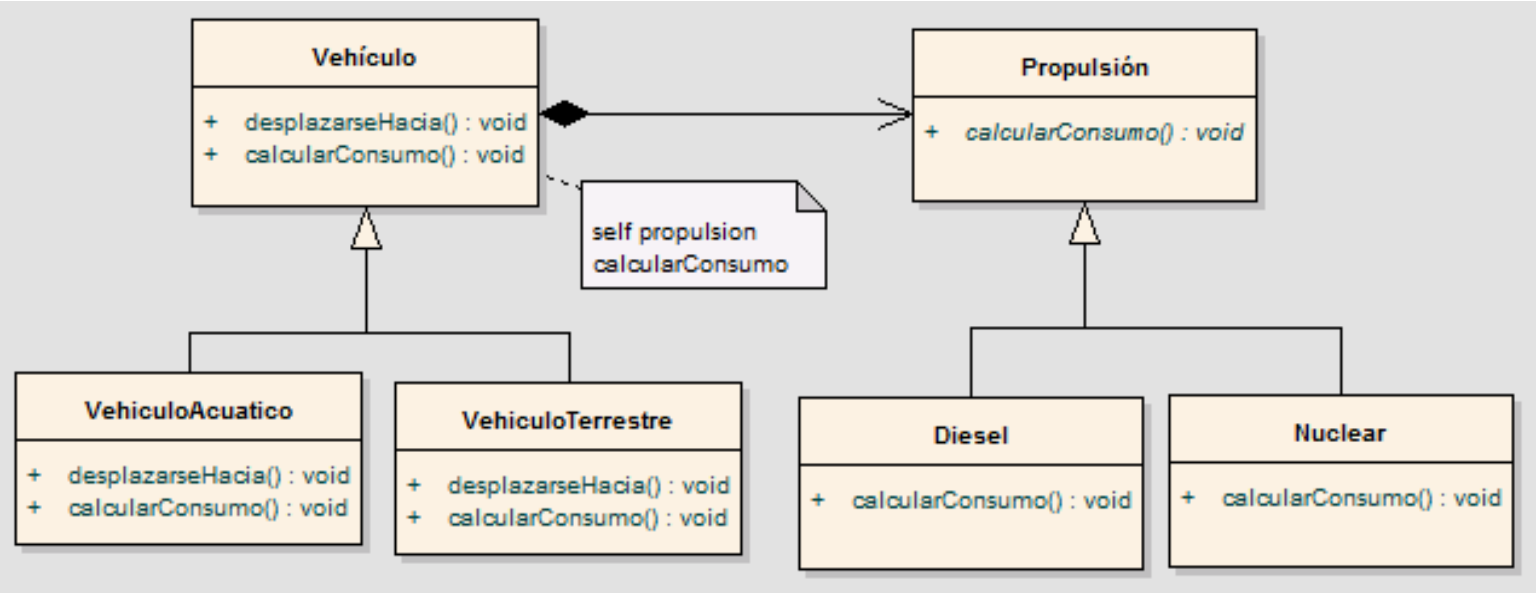
Conceptos avanzados: Composición y herencia múltiple

Herencia múltiple, y...



Conceptos avanzados: Composición vs. herencia múltiple

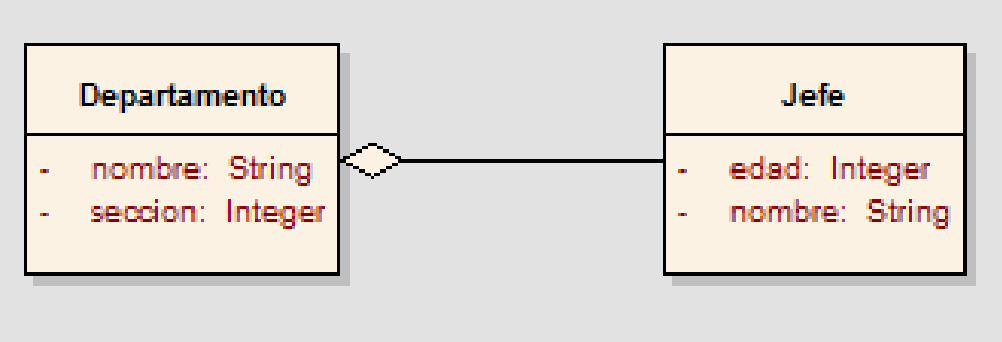
... composición



¿Cuál es mejor? ¿Por qué?
¿La composición es una solución a la h. múltiple?

Conceptos avanzados: agregación

Modela la relación “todo-parte” entre dos objetos, en la cual uno de ellos (el todo) contiene al otro (parte). Observar la notación usada.



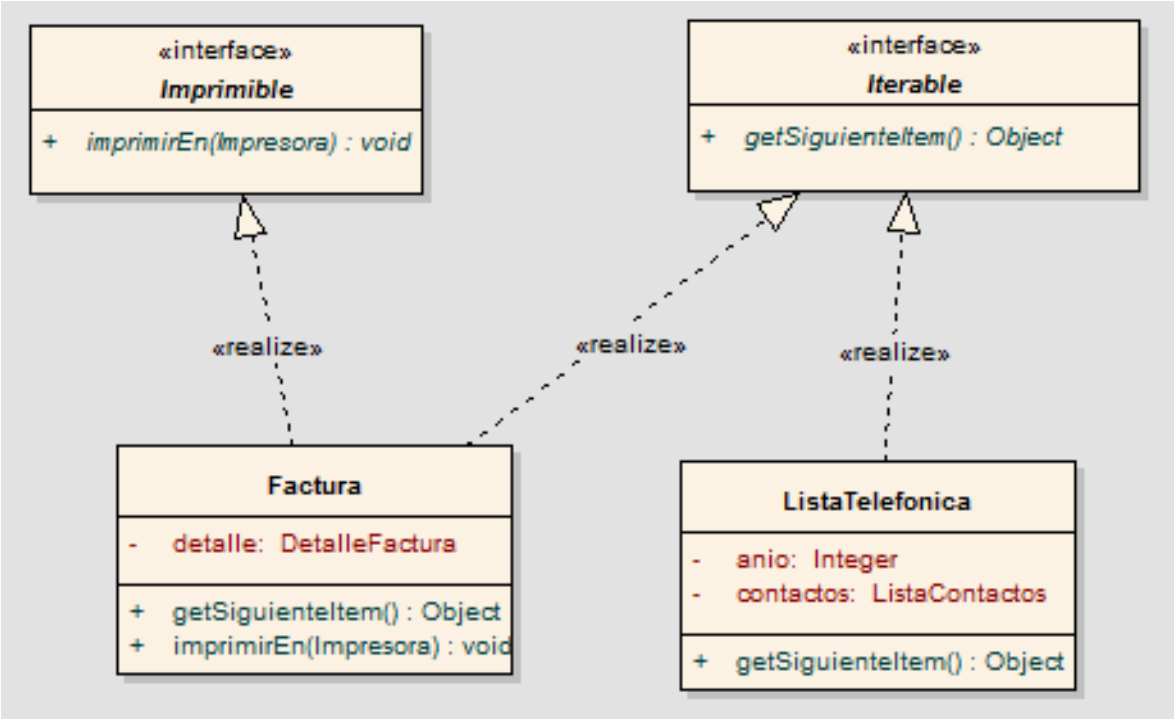
La composición es un tipo de agregación más fuerte, en el que el tiempo de vida de la parte, es menor que el del todo, y además la parte se asocia sólo a un todo.

Conceptos básicos: tipos y clases

En “Introducción a la programación” se vio la noción de *tipo de datos*.
En POO, un tipo asocia un nombre a un protocolo o conjunto de mensajes particular.
El concepto de tipo es independiente de la implementación, aunque tiene asociada semántica para cada uno de los métodos que lo componen.
Las clases y los tipos NO son lo mismo.
Un objeto puede ser de muchos tipos, pero sólo es instancia de una clase (y sus superclases).

Conceptos básicos: interfaces

En algunos lenguajes OO, el concepto de tipo se concretiza en *interfaces*, que son especificaciones de protocolos. Luego una clase puede implementar esa interfaz.



Conceptos básicos: tipado estático vs. tipado dinámico

Los lenguajes estáticamente tipados:

asocian a cada variable/referencia un tipo, y debe ser definido en tiempo de compilación

los valores que puede tomar una variable, están acotados al tipo definido.

Los lenguajes dinámicamente tipados

asocian el tipo al objeto, la variable es sólo la forma de nombrarlo.

La noción o concepto de tipo está en la cabeza del programador, dado que el concepto no está explícitamente reflejado por los lenguajes OO.

Conceptos básicos: tipado estático

Se pueden detectar errores de tipo en tiempo de compilación.
El compilador podría realizar optimizaciones.
La especificación del tipo es parte de la documentación que contiene el código fuente del programa.

Conceptos básicos: tipado dinámico

Es más flexible que el tipado estático.
Es ideal (no requisito) para sistemas en los que se puede alterar la definición de una clase en run-time (reflection), o que soportan dynamic loading.
Como contrapartida, algunos errores se detectan sólo en run-time.

Conceptos básicos: lenguajes OO

¿Qué debería tener un lenguaje para considerarse OO?

Debe permitir representar objetos (clases no es necesario)

Debe soportar encapsulamiento

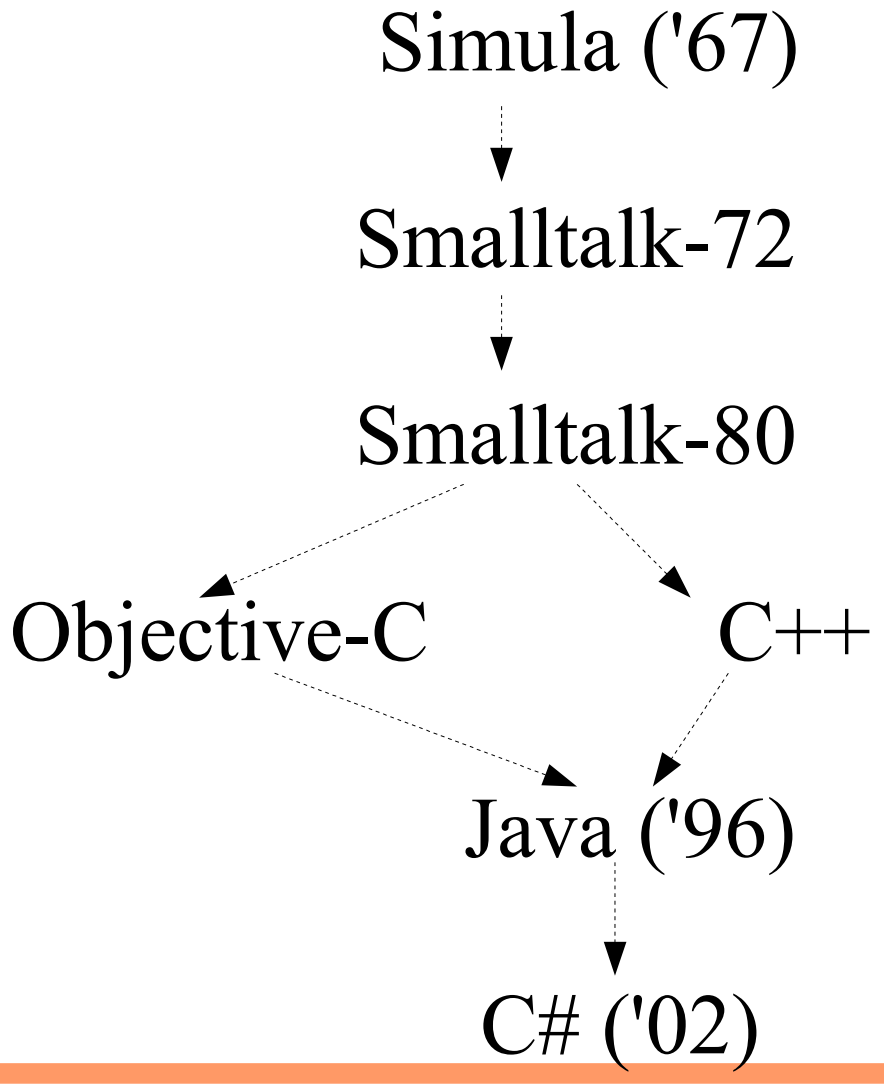
Debe proveer la comunicación entre objetos a través del envío de mensajes

Debe brindar la capacidad de modelar objetos polimórficos

Debe brindar la posibilidad de resolver el envío de mensajes por medio del binding dinámico

Conceptos básicos: lenguajes OO

Historia:



Conceptos básicos: lenguajes OO

Alternativas en el mercado actuales más conocidas:

Smalltalk

C++

Java

C#

Ruby/Ruby on rails

Conceptos básicos: lenguajes OO

Smalltalk
Lenguaje OO puro.
Sintaxis: basado en el ANSI Smaltallk de 1980
Encapsulamiento: variables de instancia privadas y protocolo público
Herencia: simple y sin restricción
Clases y métodos abstractos: no es explícito para clases. Para los métodos se usa *subclassResponsibility*.
Garbage collection: provista.

Conceptos básicos: lenguajes OO

C++

Sintaxis: sintaxis de C, con nuevos elementos para programar objetos y clases

Encapsulamiento: por default los miembros de una clases son privados.

Herencia: múltiple

Clases y métodos abstractos: definición de clases abstractas mediantes miembros virtuales que son redefinidos en subclases

Garbage collection: no posee

Conceptos básicos: lenguajes OO

Java
Sintaxis: muy similar a la de C++
Encapsulamiento: el ocultamiento de los miembros es decisión del programador en cada caso
Herencia: simple. Puede "cortarse" declarando miembros private o final.
Clases y métodos abstractos: permite ambos con la keyword *abstract*
Garbage collection: provista.

Conceptos básicos: lenguajes OO

C#
Sintaxis: similar a la de C++
Encapsulamiento: el ocultamiento de los miembros queda a cargo del programador.
Herencia: simple. Permite cortarla con el modificador private.
Clases y métodos abstractos: permitida
Garbage collection: a cargo del framework .NET

Parte III

Aplicación de conceptos

Conceptos avanzados: Streams

Los *streams* proveen un mecanismo general de acceso a cualquier dato secuenciable, más allá de la fuente del dato (archivo, socket, buffer, etc).

Se usan tanto para leer como para escribir.

Los streams *internos* operan sólo en colecciones internas de Smalltalk, mientras que los *externos* operan sobre archivos y otras fuentes de datos.

Conceptos avanzados: Streams

Creación de un stream y tipos de Streams

readStream

retorna un streeam de lectura en el receptor.

writeStream

retorna un stream de escritura (solamente) en el receptor.

readWriteStream

retorna un stream de lectura escritura en el receptor.

readAppendStream

retorna un stream de lectura/escritura que agrega datos (no borra los existentes) en el receptor.

Ej: ('../archivoDeTexto.txt' asFilename) readStream

Crea un **ExternalReadStream** sobre el archivo “archivoDeTexto.txt”.

#(1 2 3 14) readStream

Crea un **ReadStream** (que es interno) sobre el Array.

Conceptos avanzados: Streams

Jerarquía de *streams* en Smalltalk

Object
- *Stream*
  - *PeekableStream*
    - *EncodedStream*
    - *PositionableStream*
      - *ExternalStream*
        - *BufferedExternalStream*
          - **ExternalReadStream**
            - **ExternalReadAppendStream**
            - **ExternalReadWriteStream**
          - **ExternalWriteStream**
      - *InternalStream*
        - **ReadStream**
        - **WriteStream**
          - **ReadWriteStream**
          - **TextStream**

Conceptos avanzados: Streams

Protocolo usual: lectura y posicionamiento
Al crearse un stream, se ubica un *puntero* sobre la posición inicial. Luego podemos hacer

next
retorna el siguiente objeto del Stream y avanza el puntero

next: n
retorna los siguientes *n* objetos del Stream avanzando el puntero

next
retorna el siguiente objeto del Stream sin avanzar el puntero

position
retorna la posición actual del puntero

position: n
modifica el puntero a la posición *n*. No anda para todos los Streams.

skip: anInteger
mueve el puntero *anInteger* posiciones.

reset
retorna el puntero a la posición 0 del stream.

Conceptos avanzados: Streams

Ejemplo sencillo

```
numeritos := #(11 21 31 41 51 61)..
stream1 := numeritos readStream.
stream2 := numeritos readStream.
stream1 next.
stream1 next.
stream1 peek.
stream1 next.
stream1 skip: 2.
stream2 next.
stream1 next.
stream1 position: 0.
stream2 next.
stream1 next.
```

Conceptos avanzados: Acceso secuencial y acceso aleatorio

¿Qué tienen en común una OrderedCollection y un File?

Que son estructuras de elementos que están ordenados.

Conceptos avanzados: Acceso secuencial y acceso aleatorio

¿Qué tienen de diferente una OrderedCollection y un File?

Que en una OC es fácil acceder a un elemento cualquiera, p.ej.

```
miOC at: 9
```

A esto lo llamamos **acceso aleatorio**.

En un File para llegar a un elemento debo pasar por los anteriores.

A esto lo llamamos **acceso secuencial**.

Conceptos avanzados: Acceso secuencial y acceso aleatorio

Un Stream es un objeto que representa el acceso a una estructura cualquiera, interna o externa.

Si lo leo usando solamente next y skip:
estoy haciendo acceso **secuencial**.

Si lo leo usando position: y peek
estoy haciendo acceso **aleatorio**.

Conceptos avanzados: Streams

Posicionamiento

position
retorna la posición actual del puntero.

position: anInteger
modifica el puntero a la posición dada. Sólo para stream de lectura y lectura/escritura.

reset
retorna el puntero a la posición 0 del stream.

setToEnd
lleva el puntero a la última posición del stream.

skip: anInteger
mueve el puntero *anInteger* posiciones.

skipThough: anObject
posiciona el puntero en el lugar posterior a anObject, retornándose a sí mismo o nil en caso que anObject no se encuentre.

skipUpTo: anObject
idem anterior, dejando el puntero en la posición anterior a anObject.

Conceptos avanzados: Streams

Lectura

contents
retorna una copia de la colección del stream

next
retorna el siguiente objeto del stream.

next: anInteger
retorna los siguientes *anInteger* objetos del stream.

nextAvailable: anInteger
idem anterior, pero retornando todos los elementos que pueda.

upTo: anObject
retorna una colección con todos los objetos desde la posición actual hasta la ocurrencia de anObject.

upToEnd
retorna una colección con todos los objetos desde la posición actual hasta el fin del stream.

Conceptos avanzados: Streams

Escritura

NextPut: anObject
agrega anObject en la posición siguiente del stream.

nextPutAll: aCollection
agrega los objetos de aCollection a partir de la posición siguiente del stream.

cr
agrega un “enter”.

space
agrega un espacio.

tab
agrega un tab.

print: anObject
agrega al stream la representación como String de anObject (obteniéndola con el mensaje printString).

Conceptos avanzados: Streams

Cierre del stream

Para streams internos no es necesario.

Para los externos, es necesario enviar el mensaje close para liberar el recurso.

Conceptos avanzados: Streams

Ejemplos:

Streams internos (lectura y escritura)

```
|array readStrm|
array := Array with: $a with: $b with: $d with: $d.
readStrm := array readWriteStream.
readStrm position: 2.
readStrm nextPut: $c

|col1 readStrm writeStrm char|
col1 := 'This is a test' copy.
readStrm := coll readStream.
writeStrm := coll writeStream.
[ readStrm atEnd ] whileFalse: [ | char |
                                 char := readStrm next.
                                 writeStrm nextPut: char asUppercase ].
^coll
```

Conceptos avanzados: Streams

Ejemplos:

Streams externos (lectura)

```
|file fileStrm|
file := '..\notas.txt' asFilename.
fileStrm := file appendStream.
fileStrm nextPut: Character cr; nextPutAll: 'Pablo Barrientos'.
fileStrm commit.

|file|
file := '..\bin\win\visual.exe' asFilename.
fileStrm := file readStream binary.

| rStrm |
rStrm := '..\alumnos.txt' asFilename readStream.
Transcript cr; show: rStrm next printString; cr;
                    show: (rStrm next: 3) printString
```

Conceptos avanzados: Streams

Ejemplos:

Streams externos (escritura)

```
| wStrm |
wStrm := '..\newFile.tmp' asFilename writeStream.
#(eliot dave sam bruce vassili tamara bob) do: [ :name |
                                    wStrm nextPutAll: name printString;
                                          nextPut: Character cr ].
wStrm close.
```

Streams internos

El método *printOn: aStream* está definido en Object. Se puede redefinir en cada clase para que al visualizar una instancia o hacer "print-it" se vea como uno desea.

Conceptos avanzados: printOn

El método *printOn: aStream* está definido en Object. Se puede redefinir en cada clase para que al visualizar una instancia o hacer "print-it" se vea como uno desea.
Lo que se muestra cuando se hace "print-it" es lo que se agrega a aStream, no lo que devuelve el printOn: .
Lo que devuelve no se usa, no tiene sentido devolver algo.

```
Gato>>printOn: aStream
   aStream nextPutAll: 'un gato llamado '.
   aStream nextPutAll: self nombre.
   "no se devuelve nada, lo que se muestra es lo que agrega en aStream"

mish := Gato newConNombre: 'benito'.
mish  "si pinto esto y pongo print-it, muestra 'un gato llamado benito'"
```

Conceptos avanzados: Excepciones

Una *excepción* es una condición inusual que ocurre durante la ejecución de un programa, que obliga a salir del contexto donde se ejecuta.
Es un problema que merece ser *manejado* de alguna forma.

Ej: se quiere leer un archivo que no existe
ocurre una división por cero
etc.

Vamos a ver cómo manejar estos casos excepcionales.

Conceptos avanzados: Excepciones

En un ambiente con objetos y mensajes ¿con qué podremos representar una excepción?

Con un objeto, por supuesto.

Las excepciones son objetos, instancias de clases en la jerarquía de Exception.

Algunos mensajes que entienden los objetos excepción

defaultAction	qué hacer si no se la maneja.
description	un String que puede indicar qué pasó.
originator	el objeto donde ocurrió la excepción.

Conceptos avanzados: Excepciones

Algunas excepciones comunes y sus defaultAction
Notification (no hace nada)
Warning (abre un cuadro de diálogo yes/no)
Error (abre un notificador = “cartel de error”)

Algunas subclases directas o indirectas de Error son
- MessageNotUnderstood
- ArithmeticError
- ZeroDivide
- ArithmeticError

Conceptos avanzados: Excepciones

La ocurrencia de una excepción implica descartar el método en ejecución, salir del *contexto de ejecución* y lanzar una nueva excepción.

La excepción lanzada puede *atraparse* mediante un *manejador*.

Si una excepción no se atrapa, se ejecuta su `defaultAction` (para errores, el cartel de notificación).

Conceptos avanzados: manejador de excepciones

Un manejador tiene dos partes

La clase de excepción que maneja

El bloque de código (de un argumento) que ejecuta en caso que ocurra la excepción.

Se define usando el mensaje on: do: sobre el bloque de código donde se puede producir la excepción.

```
x := 7. y := 0.
[x / y] on: ZeroDivide
        do: [ :ex | Transcript show:
                        'Hubo una división de ',
                        ex dividend printString, ' por cero'; cr.]
```

Si **ZeroDivide** ocurre, se evalúa el bloque con el **objeto excepción** como parámetro.

Conceptos avanzados: manejador de excepciones

Si se quiere manejar una excepción se deberá “convertir” el conjunto de sentencias en un bloque.

```
(x / y) on: ZeroDivide do: [ :ex | … ]
```

es incorrecto porque a lo que le aplico on:do: tiene que ser un bloque.

¿Por qué pasa esto? (pensar en mensajes que entiende cada objeto).

Conceptos avanzados: manejador de excepciones

Es necesario, en general, que el manejador sea específico para el tipo de excepción que pueda ocurrir.
Sin embargo, un manejador para un tipo de excepción capturará las excepciones que ocurran de la clase y de las subclases:
Ej:

[x / y] on: DomainError do: [:e | 0]

DomainError es la superclase de ZeroDivide y capturará también el error si *y* es cero.

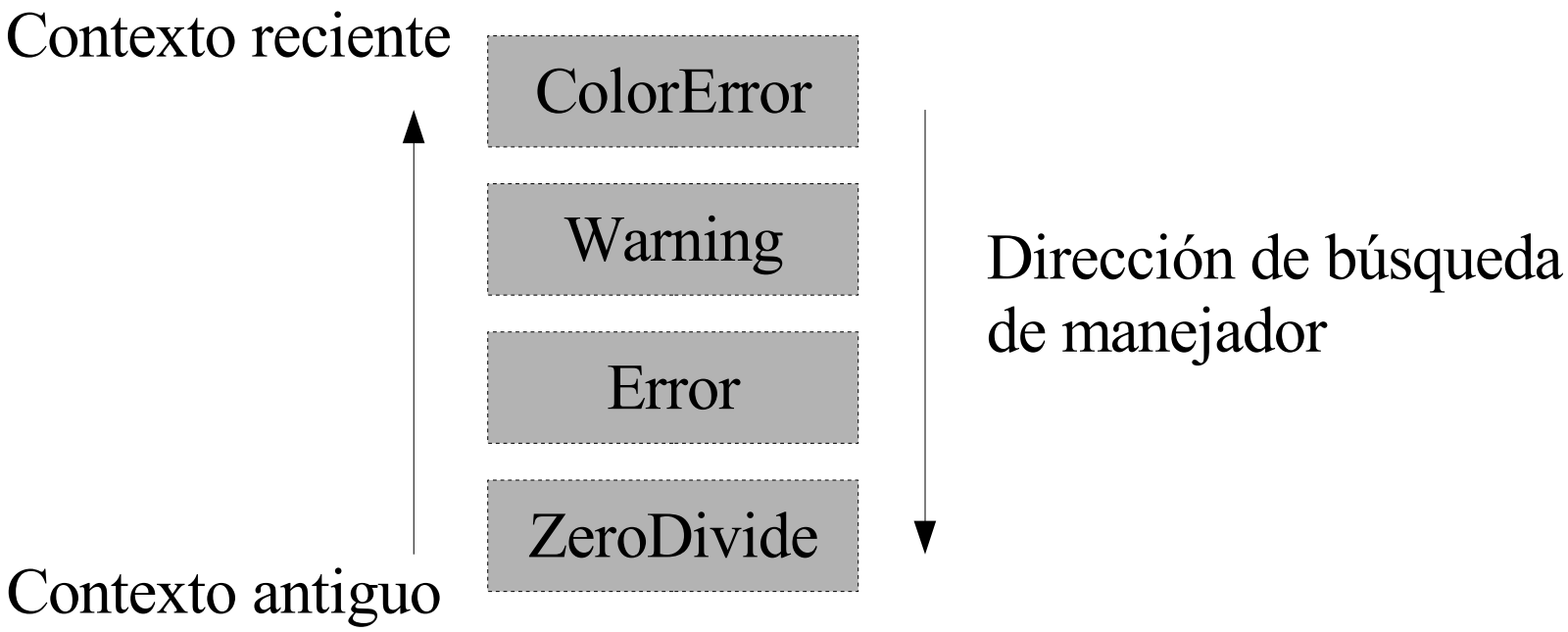
Conceptos avanzados: manejador de excepciones

Una vez lanzada la excepción, irá saliendo de los contextos que no la manejen hasta que encuentre un contexto con manejador adecuado.

Ej:

[*bloque 1*
  [*bloque 2*
    [*bloque 3*
      [*bloque 4*] on: ColorError do: [*manejador para 4*]
    ] on: Warning do: [*manejador para 3*]
  ] on: Error do: [*manejador para 2*]
] on: ZeroDivide do: [*manejador para 1*]

Conceptos avanzados: manejador de excepciones



Conceptos avanzados: manejador de excepciones

Información de la excepción:

```
[x / y]
on: Exception
do: [:theException |
Transcript show: theException description.
^'uncomputable'].
```

Capturar y manejar más de una excepción con el mismo manejador:

```
[“hacer algo...”] on: ZeroDivide, Warning
                    do: [ :theException | “manejar excepción...”]
```

Conceptos avanzados: manejador de excepciones

Excepciones con Streams

- Notification
  - EndOfStreamNotification
- Error
  - StreamError
    - IncompleteNextCountError
    - PositionOutOfBoundsError

```
rstm:= #(1 'bernal' 2 'es' 3 're' 4 'grosa') readStream.
[15 timesRepeat: [rstm skip: 1. Transcript show: rstm next; cr.]]
    on: PositionOutOfBoundsError
    do: [:e | Transcript show:
                    'te pasaste, ya estas en la posicion ',
                    e parameter printString; cr]
```

Conceptos avanzados: propagación de excepciones

```
Cliente>>gustaDe: unProducto    “el producto puede ser una Mesa”
   ^[unProducto esLindo & … ] on: ZeroDivide do: [:e | false]
Mesa>>esLindo   “la superficie es un Rectangulo”
   ^superficie escala > 2 & …. .
Rectangulo>>escala
   ^self proporcion + 0.5
Rectangulo>>proporcion
   ^base / altura
“los rectángulos tienen base y altura como variables”
```

Si un método que lanza una excepción no la atrapa, se cancelan todos los contextos de ejecución hasta que alguno la atrape.

Conceptos avanzados: lanzando excepciones

Desde los métodos que yo escribo también puedo *lanzar* una excepción cuando yo quiera.
Para eso Exception y sus subclases entienden los mensajes **de clase**

raiseSignal
raiseSignal: description.

```
CajaDeAhorro>>extraer: unMonto
    (unMonto > self saldo)
        ifTrue: [Error raiseSignal: 'saldo insuficiente'].
    saldo := saldo – unMonto.
    ...
```

Conceptos avanzados: lanzando excepciones

Se pueden crear excepciones “custom” subclasificando Exception o una de sus subclases.
P.ej. puedo crear la clase
SinSaldoError
subclase de Error y queda

```
CajaDeAhorro>>extraer: unMonto
    (unMonto > self saldo)
        ifTrue: [SinSaldoError raiseSignal: 'saldo insuficiente'].
    saldo := saldo – unMonto.
    ...
```

Conceptos avanzados: testing

El desarrollo de pruebas asegura una mejor *calidad* del software (correctitud y robustez)

Tipos de pruebas

De unidad (lo que nosotros haremos en el curso)

integración

funcional

carga, etc

Las pruebas o tests, son *especificaciones ejecutables* de la funcionalidad que cubre un artefacto.

La introducción de *bugs* o errores de código se detecta tempranamente con los *tests cases*

Conceptos avanzados: testing

Cuando se crea código nuevo, es adecuado escribir *antes* el test correspondiente.
Cuando es código existente, cada *fix* ante el hallazgo de un bug, implica correr **todos** los tests nuevamente.
Los tests deben ser:
Repetibles
Ejecutables sin intervención humana
No ser modificados, salvo que la funcionalidad haya cambiado, no la forma de implementarla.
Independientes y no superpuestos

Conceptos avanzados: test cases

Se utilizan para realizar testeos en forma organizada.
Muchas metodologías de desarrollo de software se basan en esta técnica.
Se escribe una clase de test por cada clase, y uno o varios mensajes de test por cada mensaje de la clase a testear.
Cada test es independiente del otro.
Existe, en general, para cada lenguaje OO una herramienta para especificar test cases.

Conceptos avanzados: SUnit

Herramienta incluida en diferentes versiones de Smalltalk para la administración de tests.

¿Cómo se especifican test cases?

Conjunto de tests => subclase de *TestCase*

Test unitario => Método en la clase que comienza con el prefijo *test*

Contexto de ejecución general y particular para cada test

Conceptos avanzados: SUnit - ejemplo

Deseamos testear la implementación de la clase Set.

```
XProgramming.SUnit defineClass: #TestSet
superclass: #{XProgramming.SUnit.TestCase}
indexedType: #none
private: false
instanceVariableNames: 'empty full '
classInstanceVariableNames: ''
imports: ''
category: 'SUnit'
```

Conceptos avanzados: SUnit - setUp

Para inicializar el contexto de ejecución de cada test, redefinimos el método *setUp*, heredado de TestCase.

```
>> setUp
    empty := Set new.
    full := Set with: 5 with: #abc.
```

Definimos el primer test sobre *includes:*

```
>> testIncludes
    self assert: (full includes: 5).
    self assert: (full includes: #abc).
```

Para borrar el contexto, sobreescribimos el método *tearDown.*

Conceptos avanzados: SUnit

assert: verifica una condición que debe ser true
Existen otros métodos para verificar diferentes condiciones:
assert: aBoolean description: aString
deny: aBoolean
deny: aBoolean description: aString

Se debe redefinir el método isLogging de la clase de test (debe retornar true) para que muestre las descripciones en caso de fallo.

signalFailure: aString provoca el fallo del test explícitamente.

Conceptos avanzados: SUnit

should: verifica una condición en un bloque

Existen otros métodos para verificar diferentes condiciones:

should: aBlock description: aString
shouldnt: aBlock
shouldnt: aBlock description: aString

Para testear ocurrencia de errores:

should: aBlock raise: anExceptionalEvent
should: aBlock raise: anExceptionalEvent description: aString
shouldnt: aBlock raise: anExceptionalEvent
shouldnt: aBlock description: aString raise: anExceptionalEvent

Ejemplo: self should: [1/0] raise: Error

Conceptos avanzados: SUnit

El código en los test no debería incluir control de flujo (if, loops, excepciones, etc), porque eso puede hacer al test no repetible

¿Cómo correr este test?

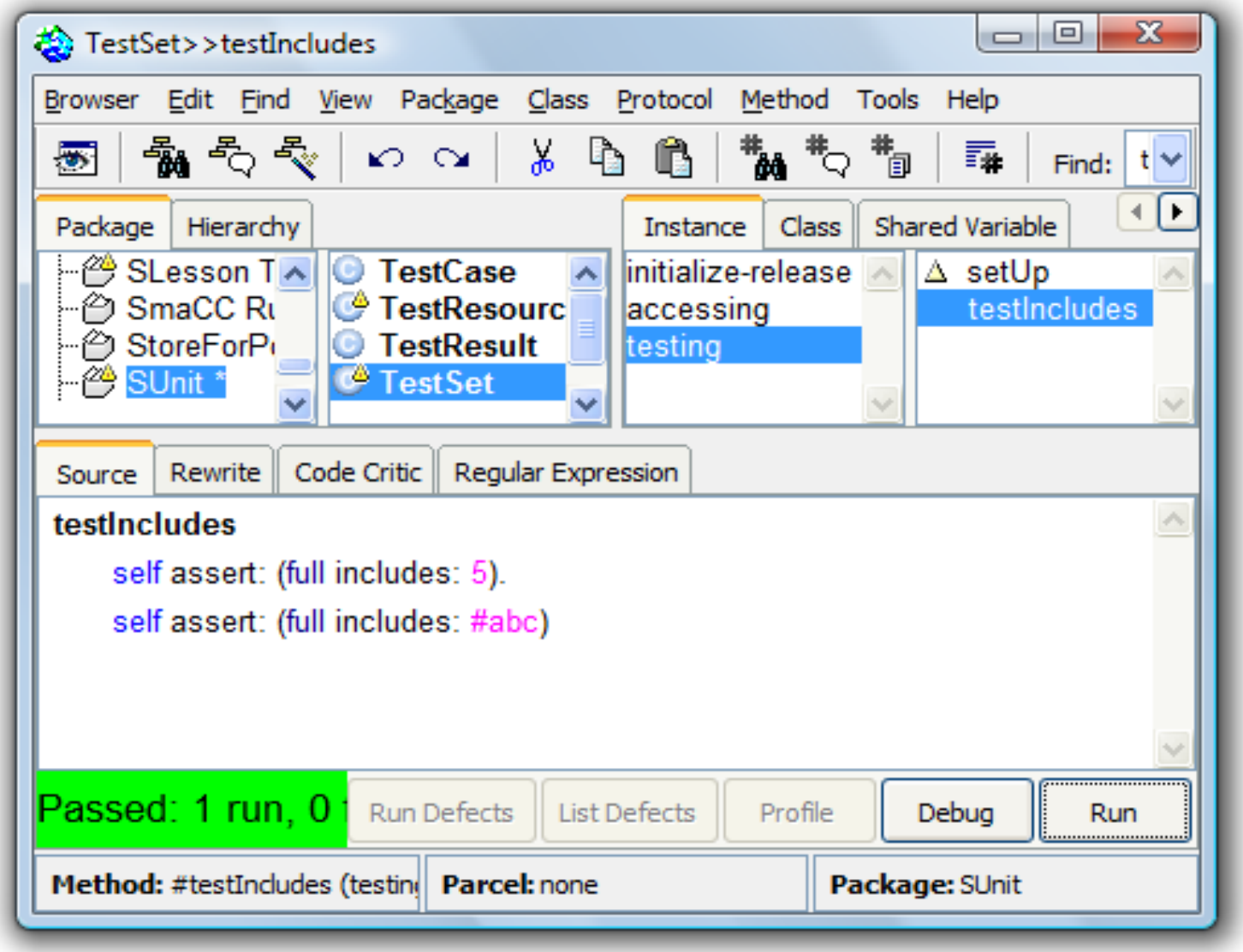
    TestSet run: #testIncludes

¿Cómo correr todos los tests de la clase?

    TestSet suite run

También se puede usar el parcel *RBSUnitExtensions* para ambos casos

Conceptos avanzados: SUnit



Parte IV

Análisis y diseño orientado a objetos

Unified modeling language

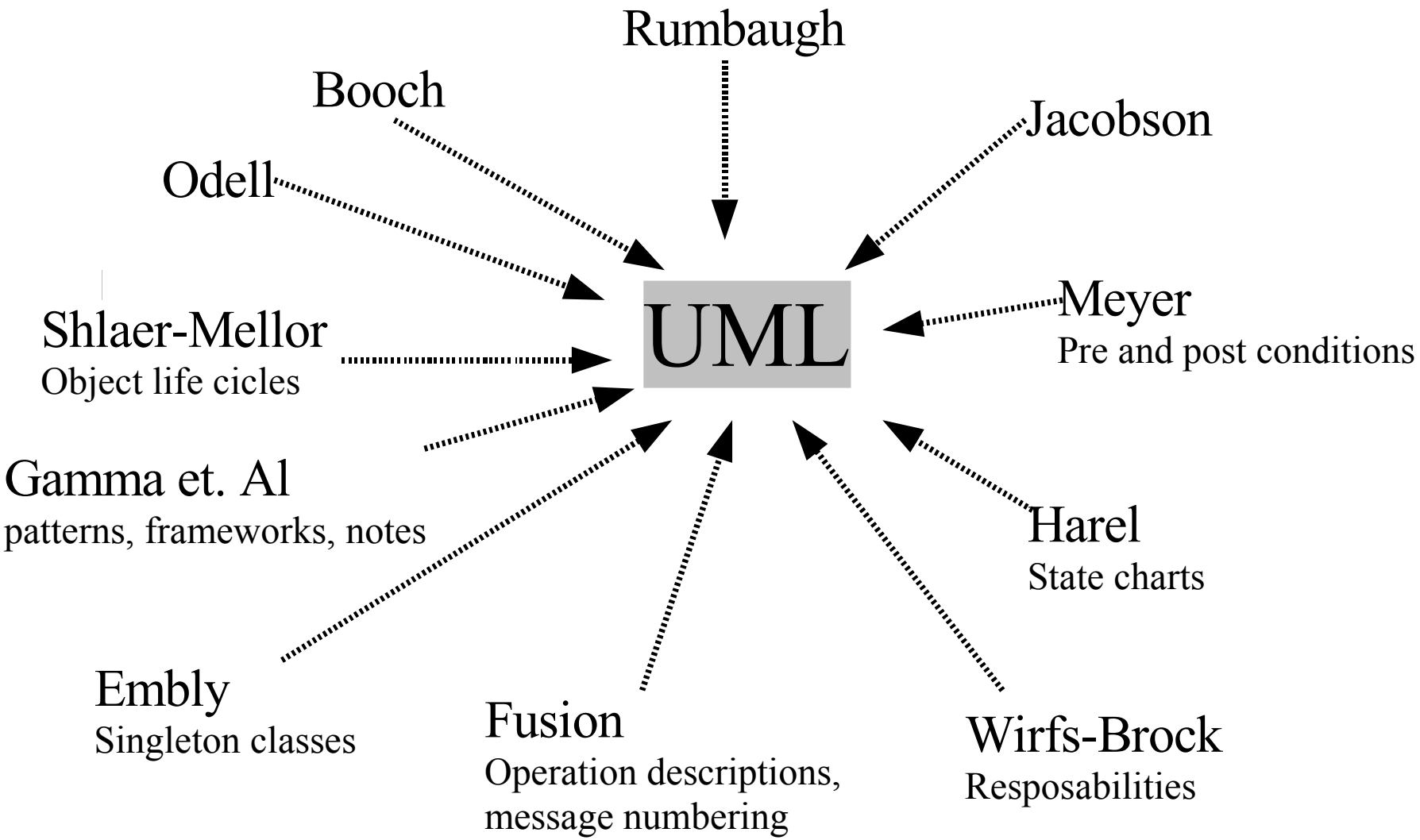
UML es un lenguaje para la especificación, visualización, construcción y documentación de documentos de sistemas de software.

Es independiente del lenguaje de implementación y proceso de desarrollo del software.

Surge como resultado de la unificación de varios lenguajes de modelado que existían a mediados de los '90.

Booch, Rumbaugh y Jacobson se unieron y promovieron UML como standard de la OMG (Object Management Group).

UML

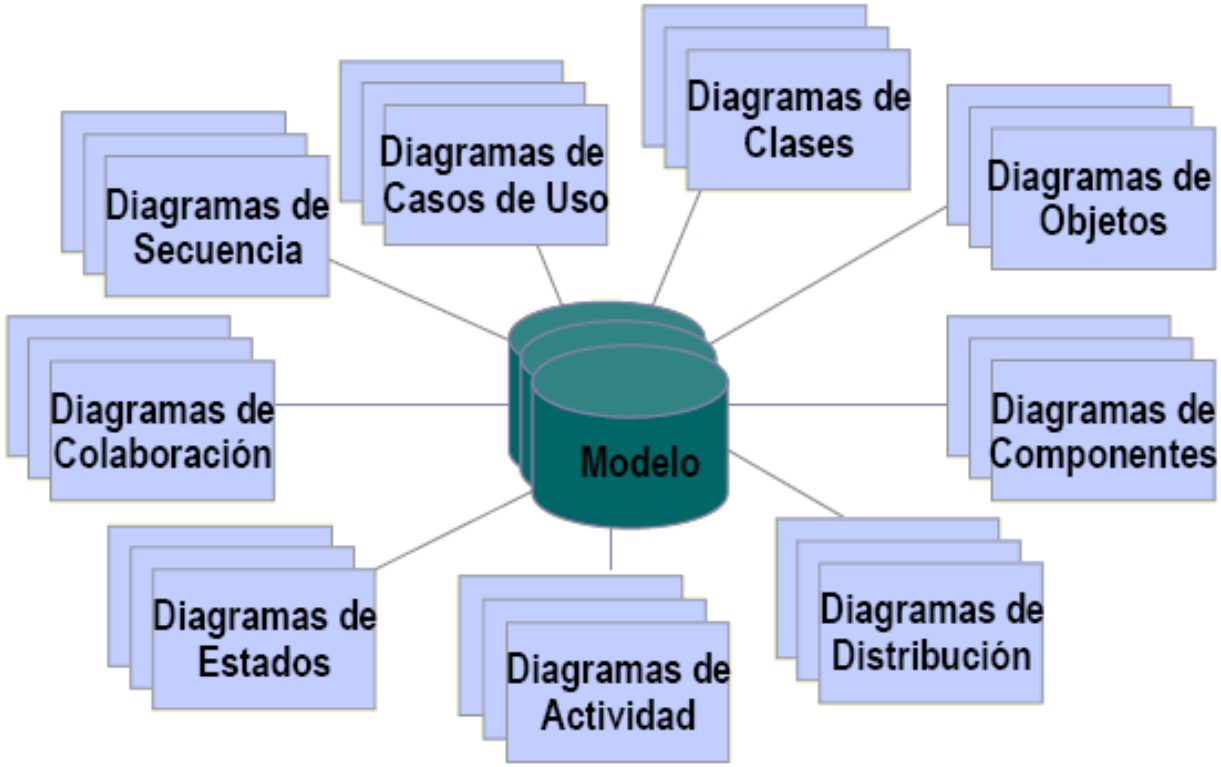

Rumbaugh
Booch
Jacobson
Odell
UML
Meyer
Pre and post conditions
Shlaer-Mellor
Object life cicles
Gamma et. Al
patterns, frameworks, notes
Harel
State charts
Embly
Singleton classes
Fusion
Operation descriptions,
message numbering
Wirfs-Brock
Resposabilities

UML – clasificación de diagramas

Diagramas de Casos de Uso (*)
Diagrama de Clases (*)
Diagrama de Objetos (*)
Diagrama de Estados (*)
Diagramas de comportamiento:
Diagramas de Estado
Diagrama de Actividad
Diagramas de Interacción:
Diagramas de Secuencia (*)
Diagramas de Colaboración
Diagramas de implementación:
Diagrama de componentes
Diagrama de distribución

UML – diagramas y modelo

Los diagramas son la forma de describir un mismo modelo que provee UML

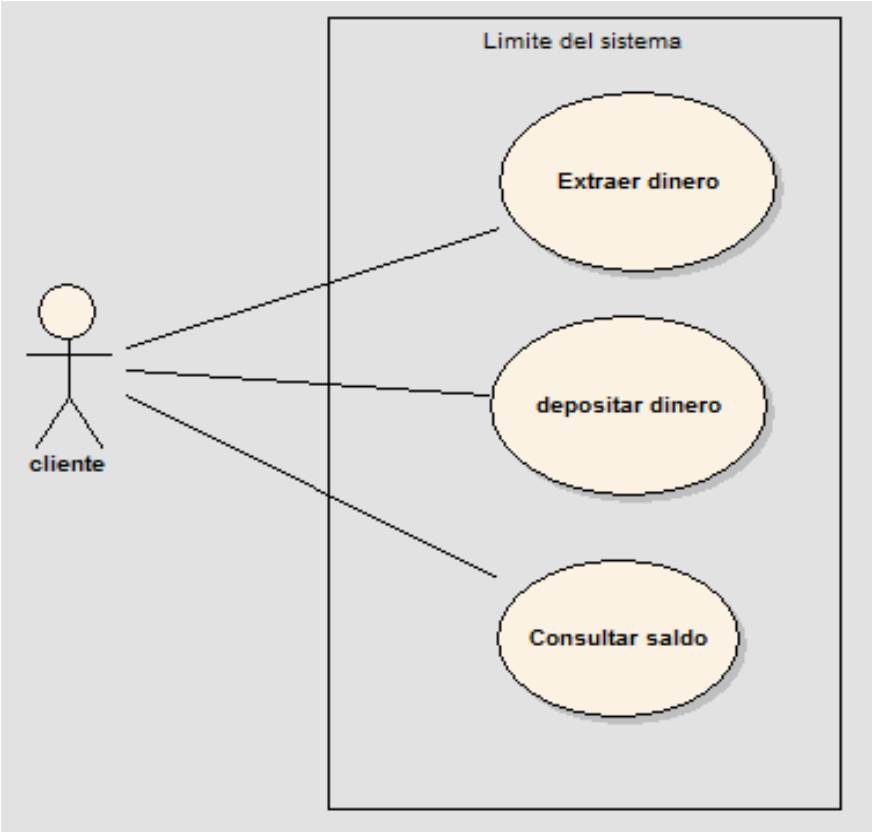


UML – diagrama de casos de uso

Permite modelar el sistema desde el punto de vista del usuario (actor).
Cada caso de uso modela la interacción entre un usuario y el sistema.
Componentes:
Sistema
Caso de uso: unidad funcional completa
Actor: entidad externa que interactúa con el sistema. Modela un tipo de rol que juega la entidad.
Relaciones: entre casos de uso y entre actores

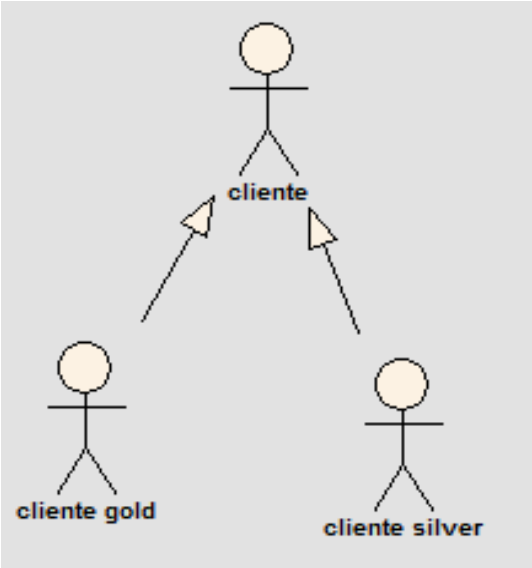
UML – diagrama de casos de uso

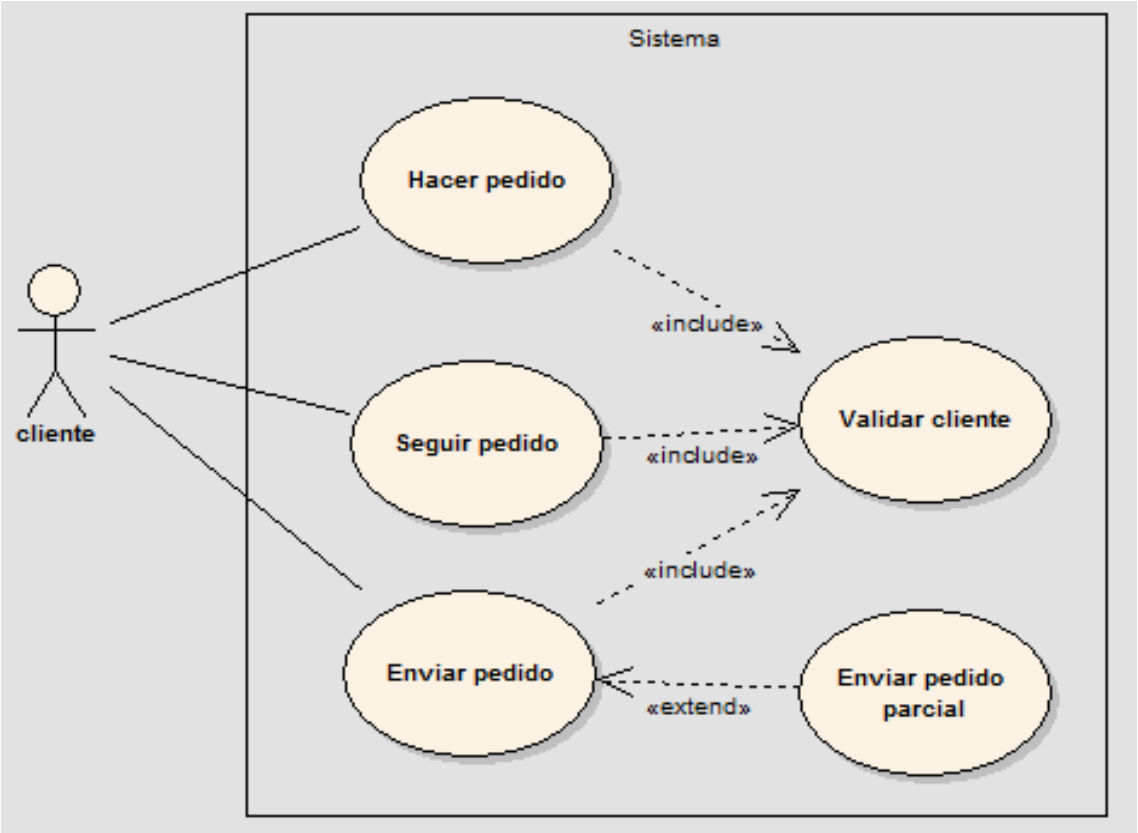
Notación: ejemplo



UML – diagrama de casos de uso

Relaciones





UML – diagrama de casos de uso

Junto a cada caso de uso se genera documentación que describe:

Restricciones (ej: cliente sin rojo para extraer)

Escenarios del caso de uso (ej: camino normal, camino si no hay dinero en el cajero para extraer)

Precondiciones y postcondiciones

No hay forma estándard de documentar

UML – diagrama de clases

Es una vista gráfica del modelo estático y estructurado del sistema.

Componentes:

Clases

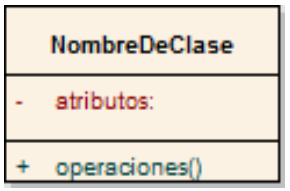
Atributos

Operaciones

Asociaciones entre clases

Interfaces (no se verá en este curso)

UML – diagrama de clases



La descripción de una clase se compone de sus atributos y operaciones.

¿qué se puede describir de los atributos?

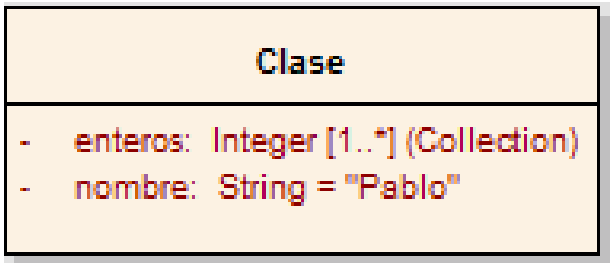
Tipo

Alcance (clase o instancia)

Valor inicial

Multiplicidad

Visibilidad:



En Smalltalk todo atributo o variable de instancia es visible desde la clase o las subclases que lo heredan (protegida).

Exiten otros tipos de visibilidad

Privada (-): sólo los objetos de la clase lo ven. No se hereda el atributo.

Protegida (#): sólo los objetos de la clase y las subclases lo ven.

Pública (+): todo objeto puede acceder al atributo.

Default (~): sólo acceden objetos en el paquete de la clase

UML – diagrama de clases

¿Qué se puede describir de las operaciones?

Tipo de retorno

Parámetros, en orden y separados por comas

por cada uno: tipo, nombre, entrada/salida

Alcance: clase o instancias

Los métodos de clase se subrayan

Abstracto o concreto

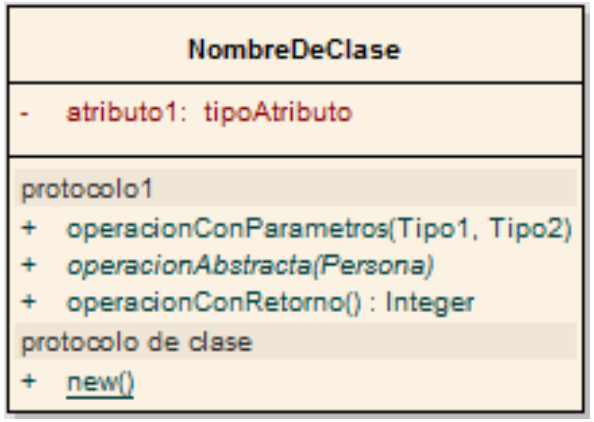
Los métodos abstractos están en cursiva

Protocolo al que pertenece

Visibilidad

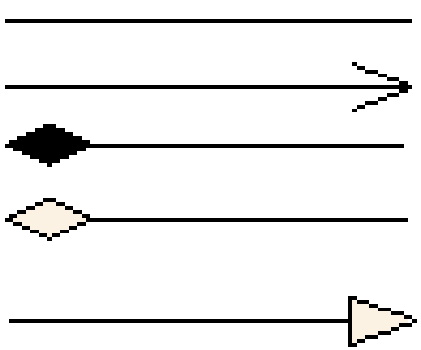
En Smalltalk todo método es público

Descripción del comportamiento en pseudocódigo (no estándar)



UML – diagrama de clases

Relaciones entre clases:



Asociación

Composición

Agregación

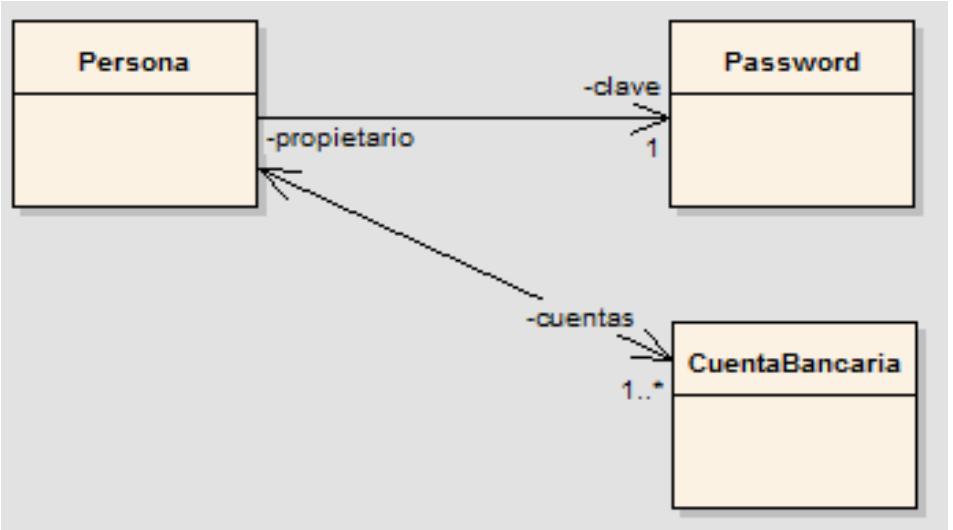
Generalización

Las relaciones de asociación tienen:

Visibilidad

Multiplicidad

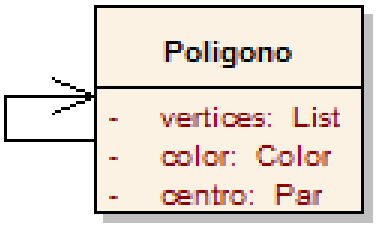
Roles



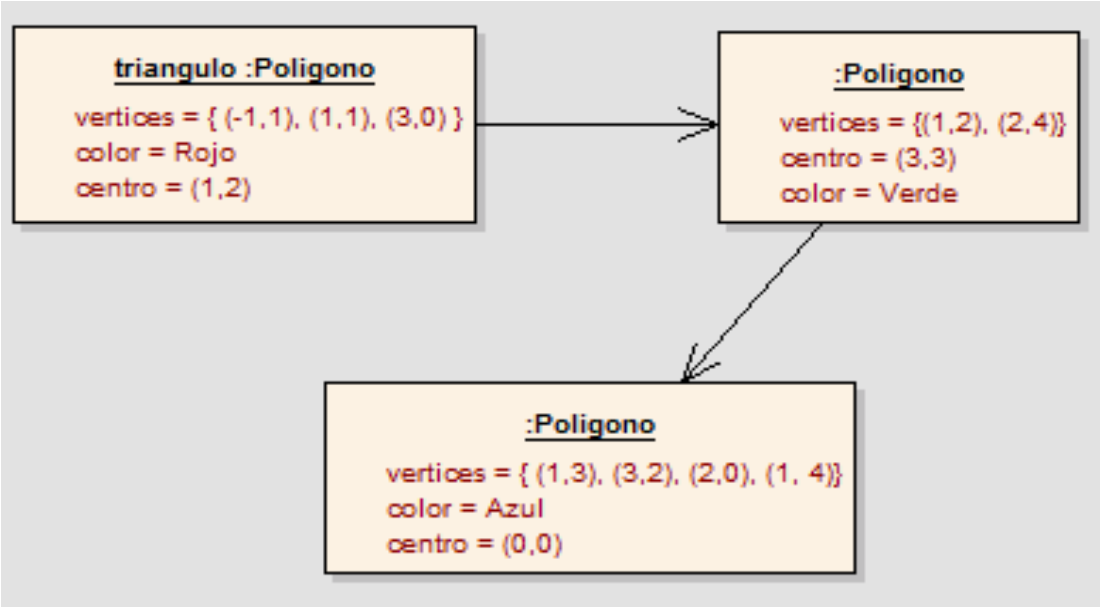
UML – diagrama de objetos

Permite modelar las instancias de los elementos contenidos en los diagramas de clases.
Muestra un conjunto de objetos y sus relaciones en un momento concreto.
Es muy similar a un diagrama de clases.
Componentes:
Objetos
Relaciones entre objetos

UML – diagrama de objetos



Descripción de un objeto
Nombre (opcional)
Clase
Estado



Relaciones, similares a las relaciones de asociación de un diagrama de clases.

UML – diagrama de estados

Permite definir los diferentes estados que podría tener una entidad durante su tiempo de vida.

Componentes:

Estados

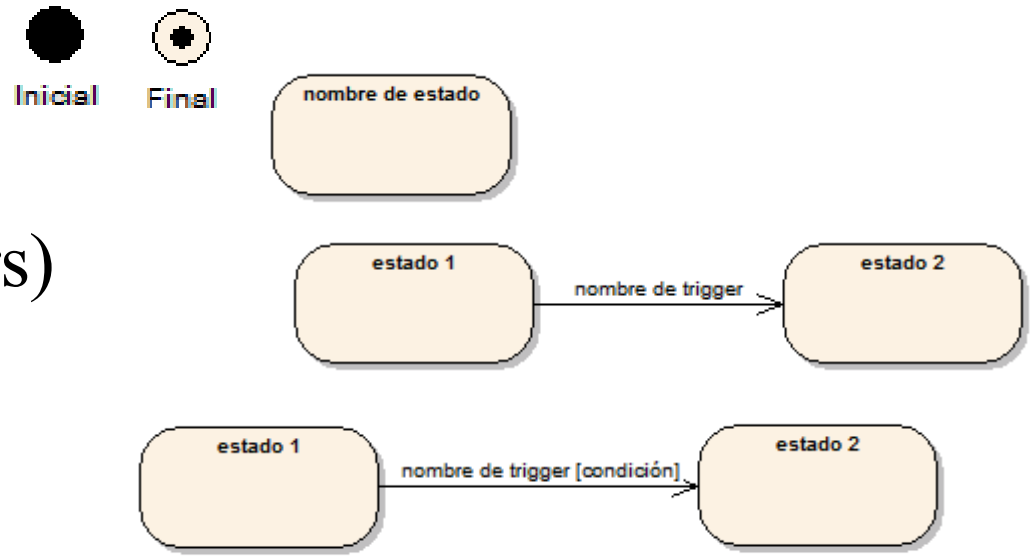
Inicial, final

Intermedio

Transiciones (triggers)

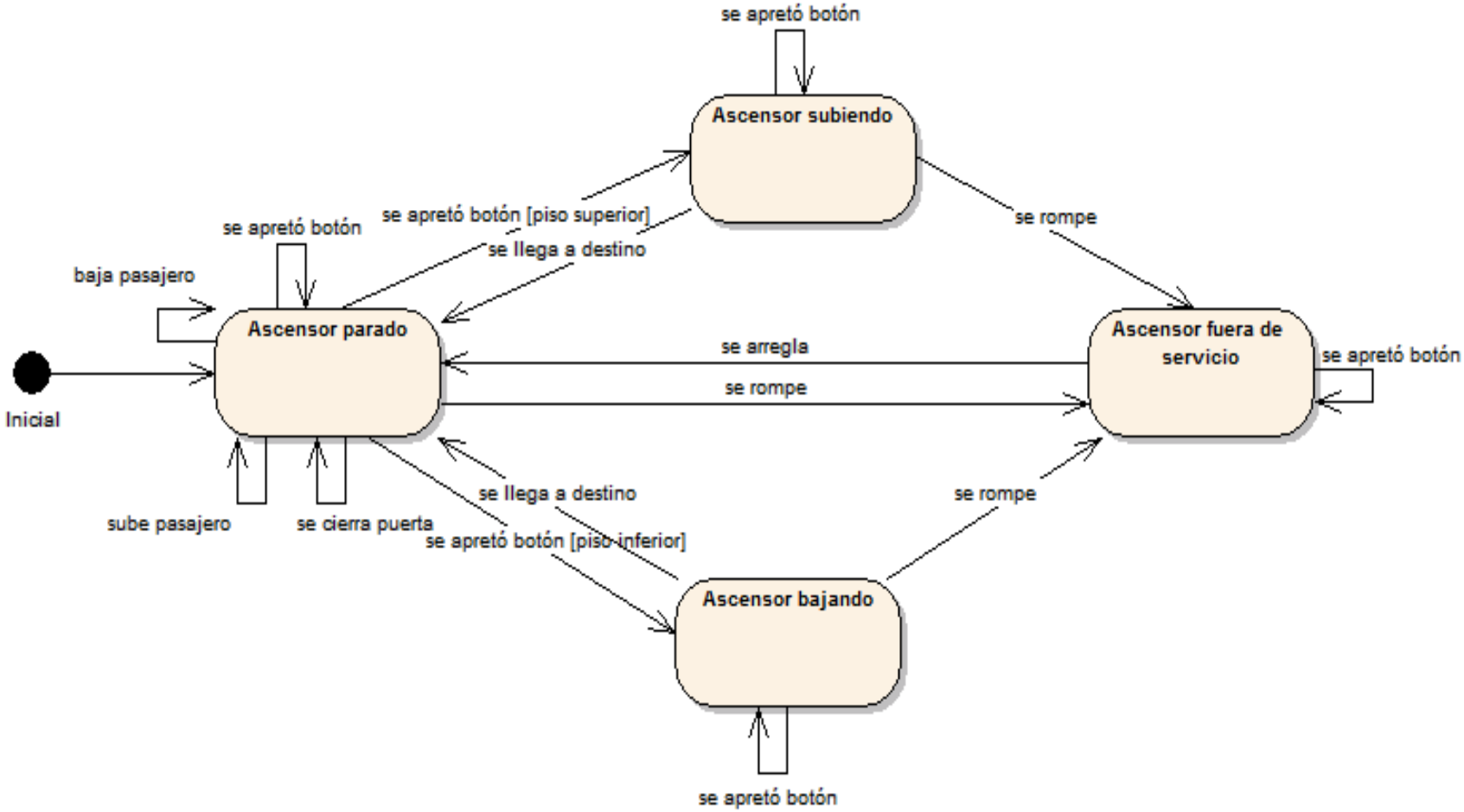
incondicional

condicional



UML – diagrama de estados

Ejemplo: estados de un ascensor



UML – diagramas de interacción

Describen una interacción que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que se pueden enviar, para realizar un comportamiento.

Existen dos tipos:

Diagramas de secuencia

Diagramas de colaboración (no lo veremos)

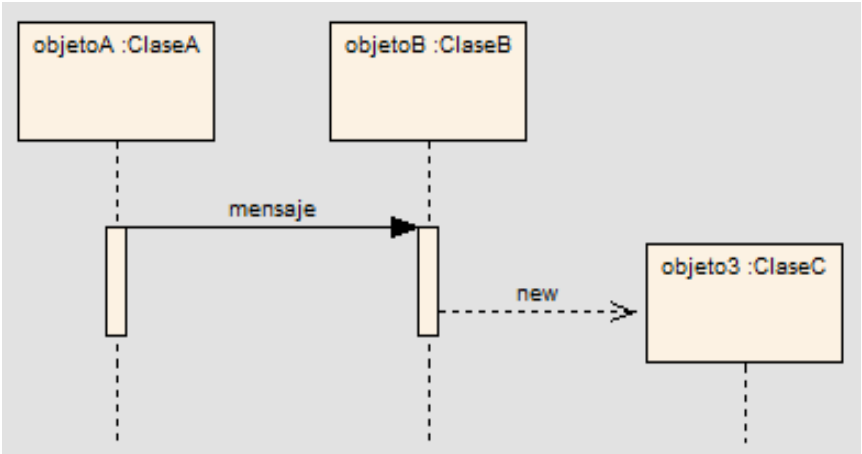
UML – diagrama de secuencia

Detaca la ordenación temporal de los mensajes que ocurren entre objetos

Cada objeto cuenta con una *línea de vida*, que muestra el tiempo de vida del mismo.

La activación de un objeto representa la ejecución de una operación que realiza el mismo.

Notación:

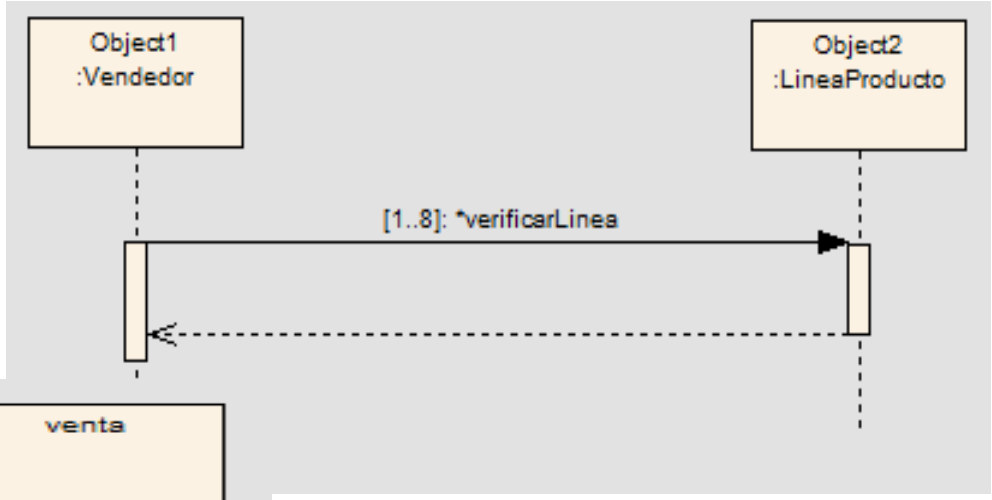


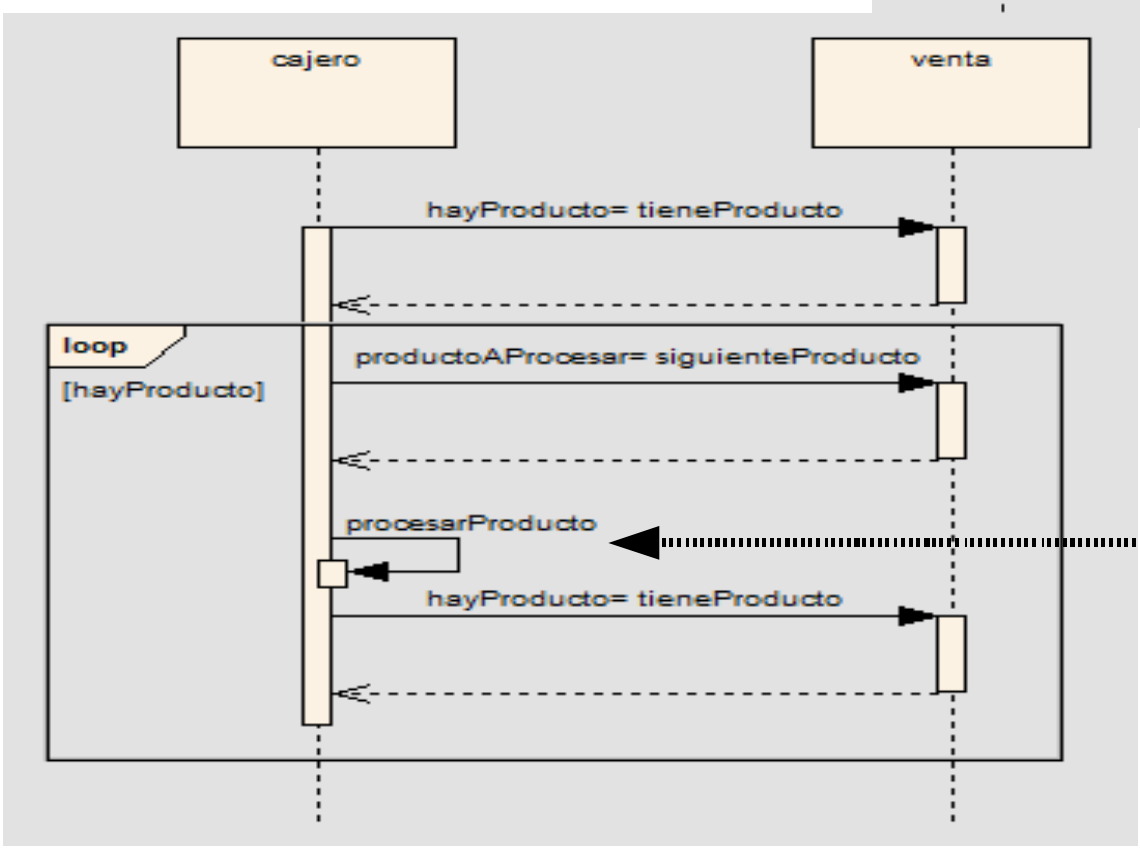
UML – diagrama de secuencia

Tipos de mensajes:
directos
a la clase o instancia
respuesta/resultado
Sincrónicos y asincrónicos
Sintaxis:
[Condición] * [expresión de iteración] [valor de retorno :=]
mensaje (parametros)

UML – diagrama de secuencia

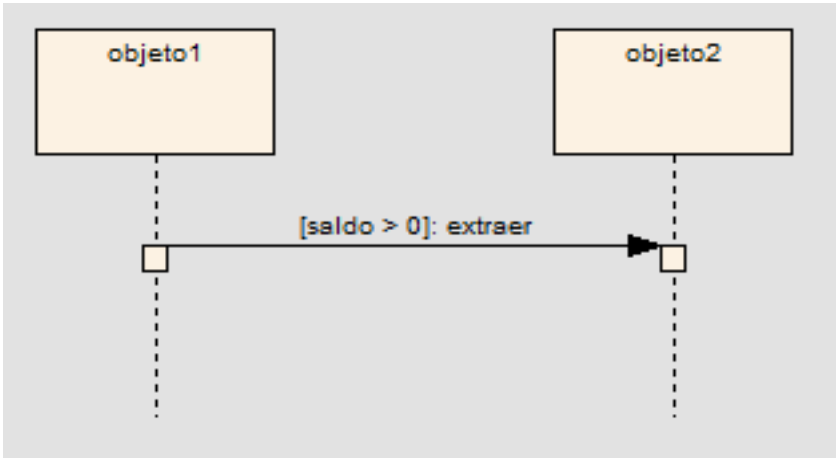
Iteración:

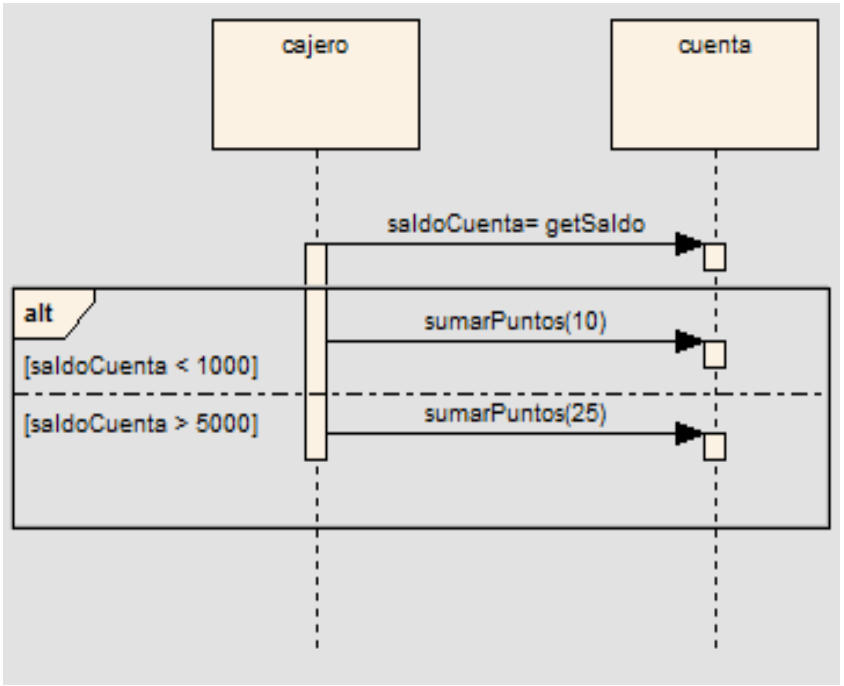




UML – diagrama de secuencia

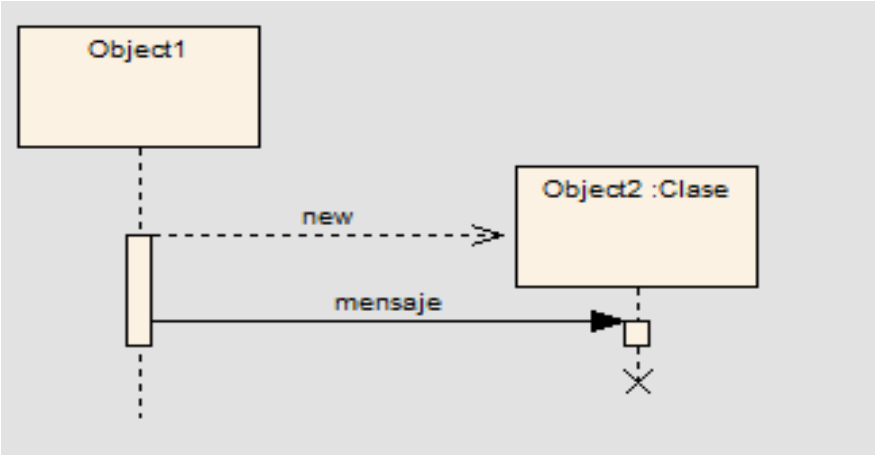
Bifurcación:



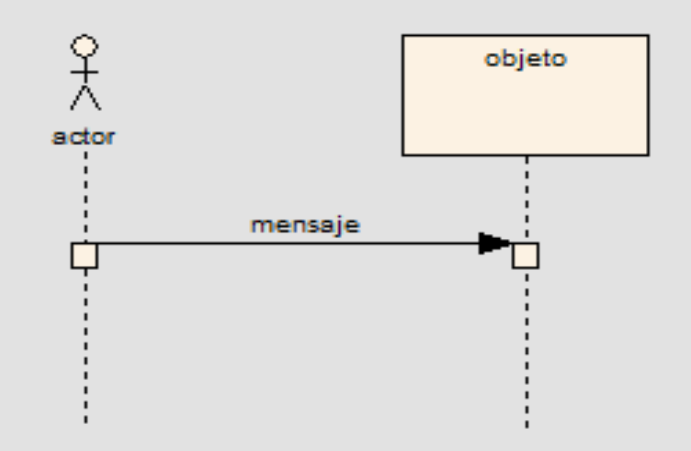


UML – diagrama de secuencia

Creación:



Cliente:



Parte V

Patrones de diseño

Patrones de diseño

Según el arquitecto Christopher Alexander:
"un patrón es una descripción de un problema recurrente en un contexto determinado, junto con una (buena) solución que puede reusarse"
La idea es reutilizar la experiencia previa de otras personas con problemas similares y que dieron una buena solución

¿Y cómo se lleva esto a la programación OO?
The gang of four –> *Design* Patterns (1995)!

Patrones de diseño

Patrón de diseño orientado a objetos:

Muestra soluciones expresadas en términos de objetos, mensajes y clases, para problemas recurrentes en el diseño OO, en un contexto determinado.

Capturan experiencia anterior de expertos en soluciones similares.

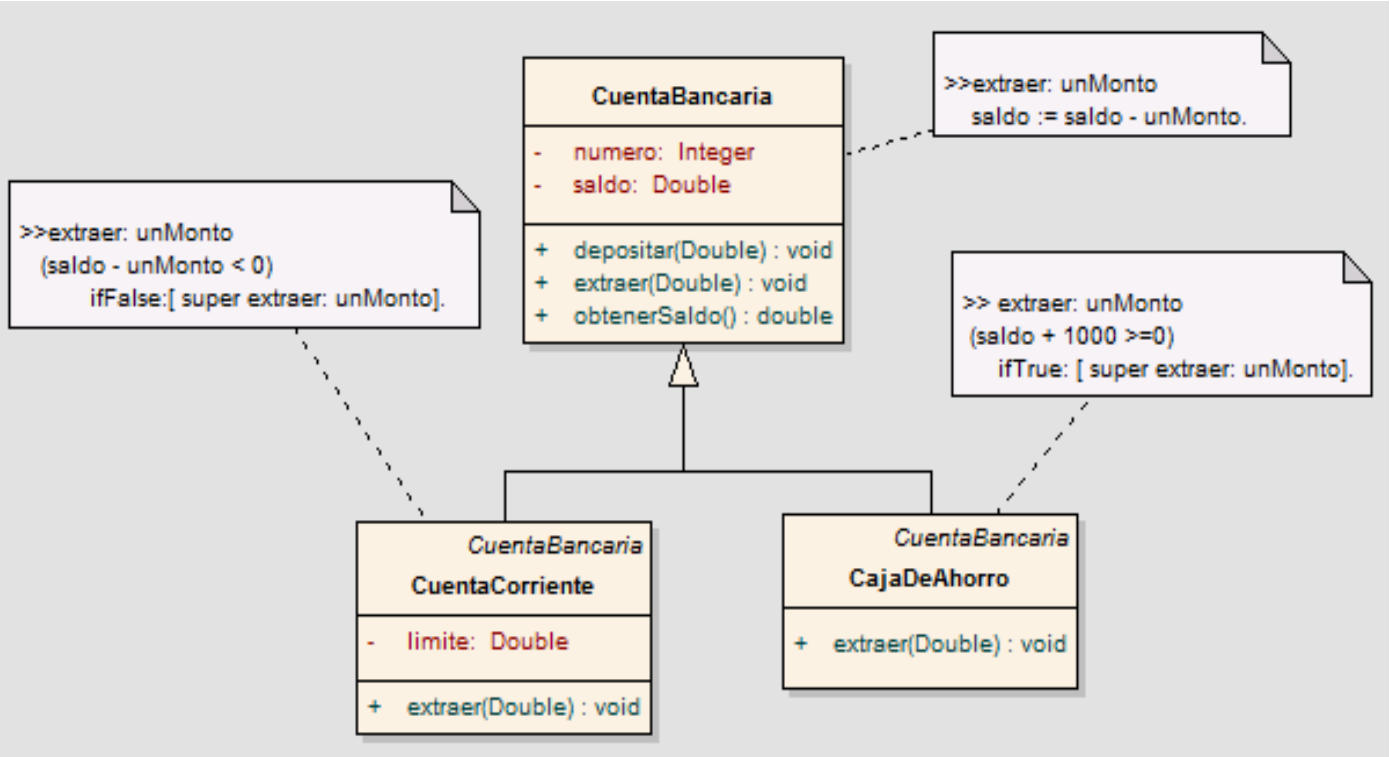
Clasificación:

Creacional, de comportamiento, estructural

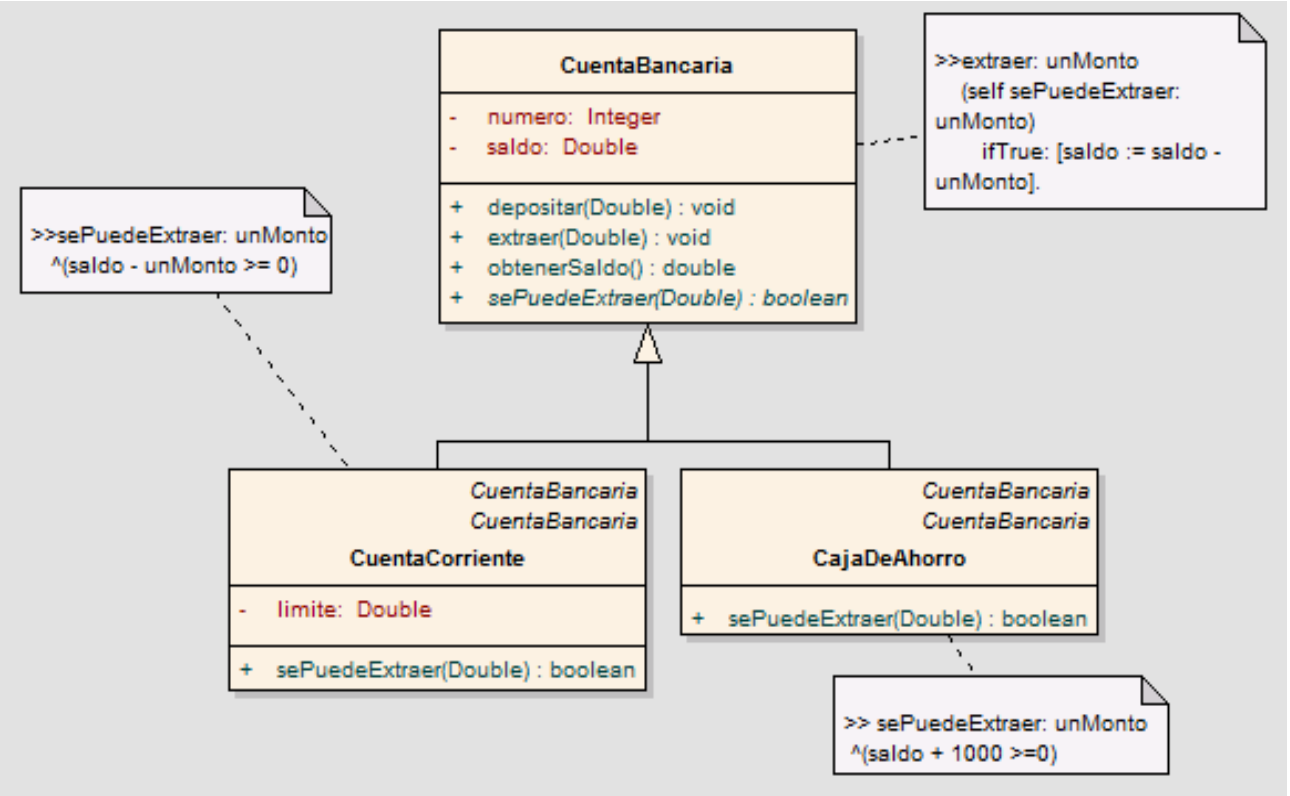
Existe una forma de documentar un patrón que incluye: nombre, intención, ejemplo de motivación, estructura, consecuencias, implementación, ejemplo y usos conocidos.

Patrones de diseño – template method

¿Qué comportamiento común hay entre todas las cuentas?
¿Qué comportamiento diferente existe?



Patrones de diseño – template method



Se *factoriza* comportamiento común en la superclase.

Se *reutiliza* el comportamiento factorizado y se *especializa* en las subclases algunos pasos.

Cambios generales en el comportamiento factorizado se aplican automáticamente en las subclases.

Patrones de diseño – template method

Intención

Definir el esqueleto de un algoritmo en una clase, dejando algunos pasos para definir en las subclases.

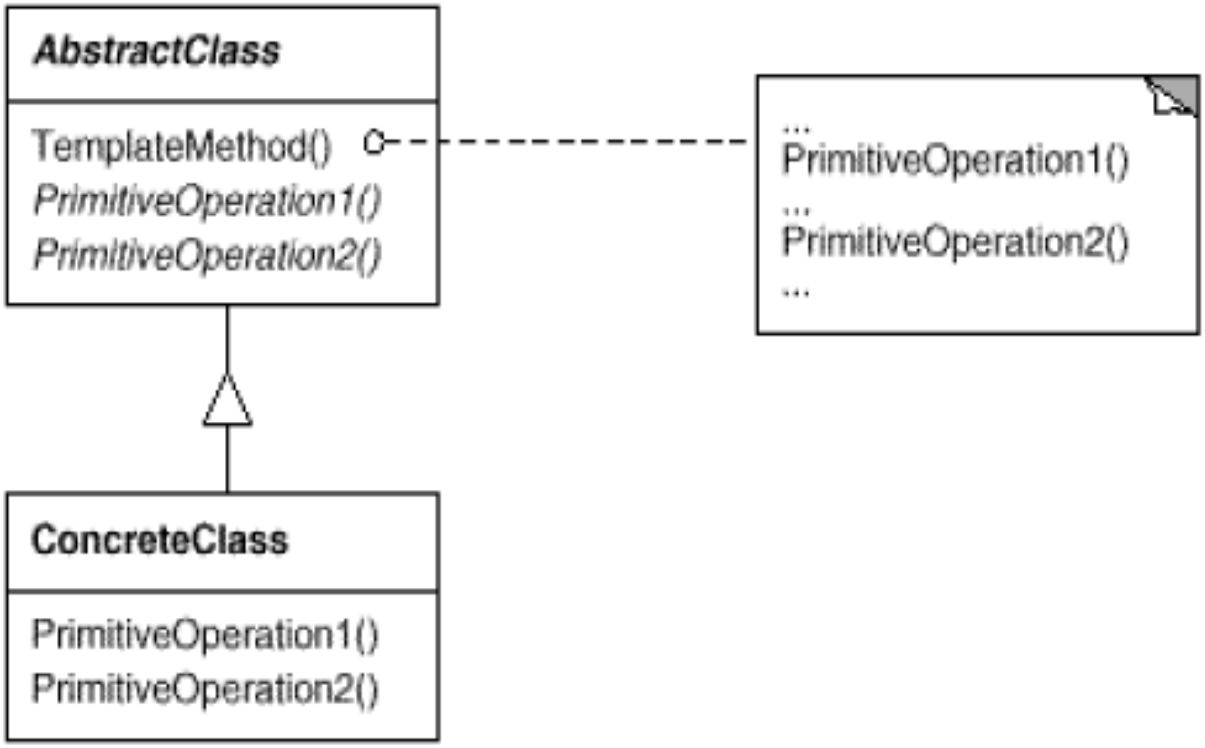
Aplicabilidad

Para implementar las partes invariantes de un algoritmo una vez y dejar a las subclases definir el comportamiento que puede variar.

Para factorizar comportamiento común de las subclases y evitar duplicación de código.

Patrones de diseño – template method

Estructura del patrón:



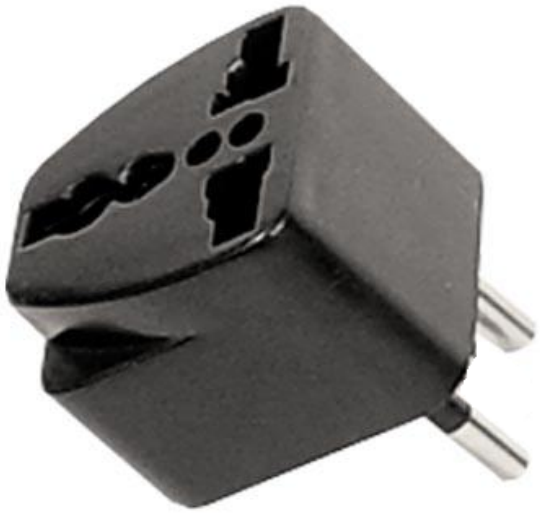
Patrones de diseño - adapter

- Laptop super moderna... con enchufe raro...

- Tomacorriente algo viejo...

Patrones de diseño - adapter

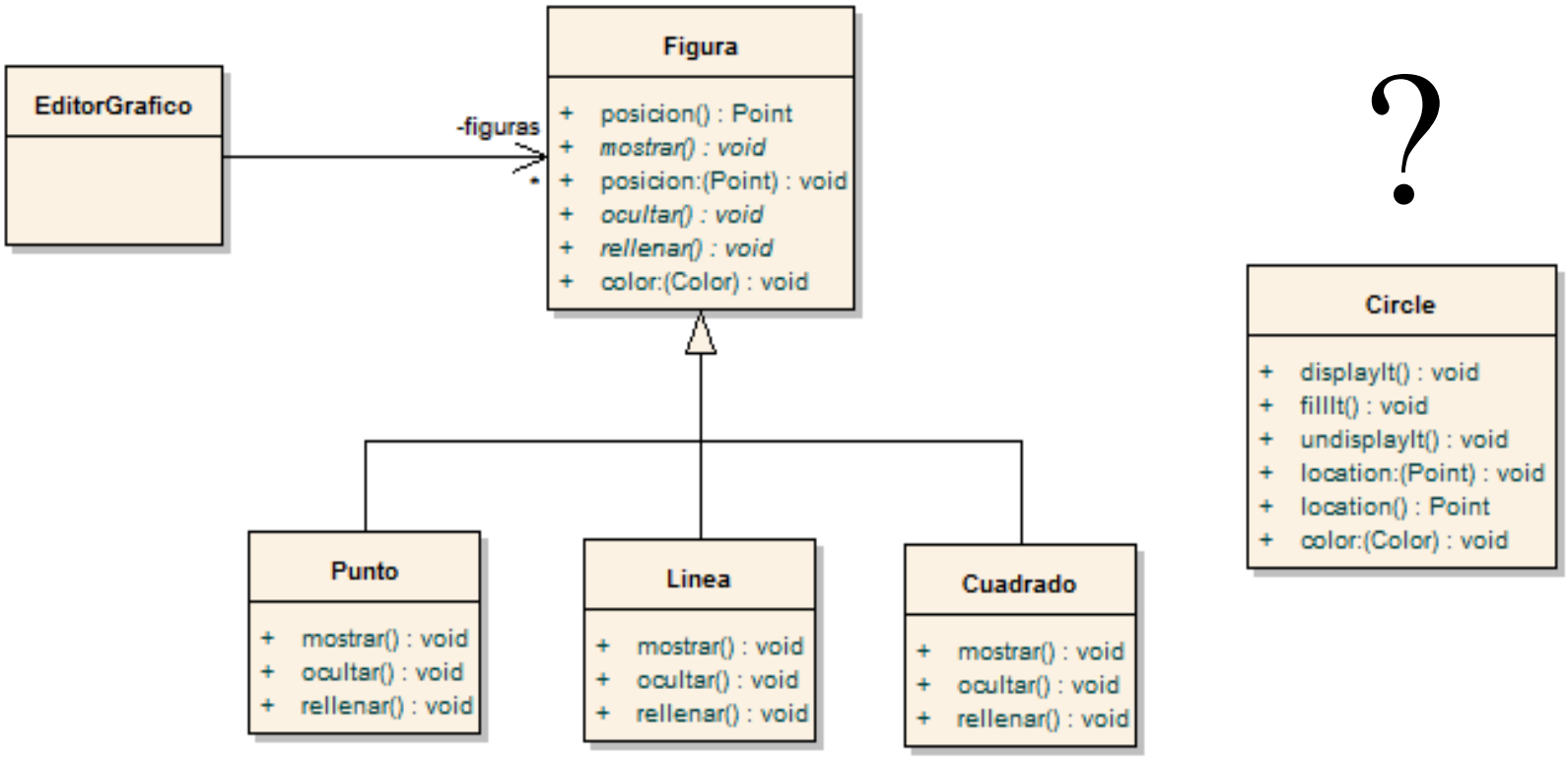
- Tiramos nuestra laptop nueva?
- Cambiamos el tomacorriente por uno igual al enchufe?
- Nada de eso. En la vida real... existe nuestro amigo adaptador!



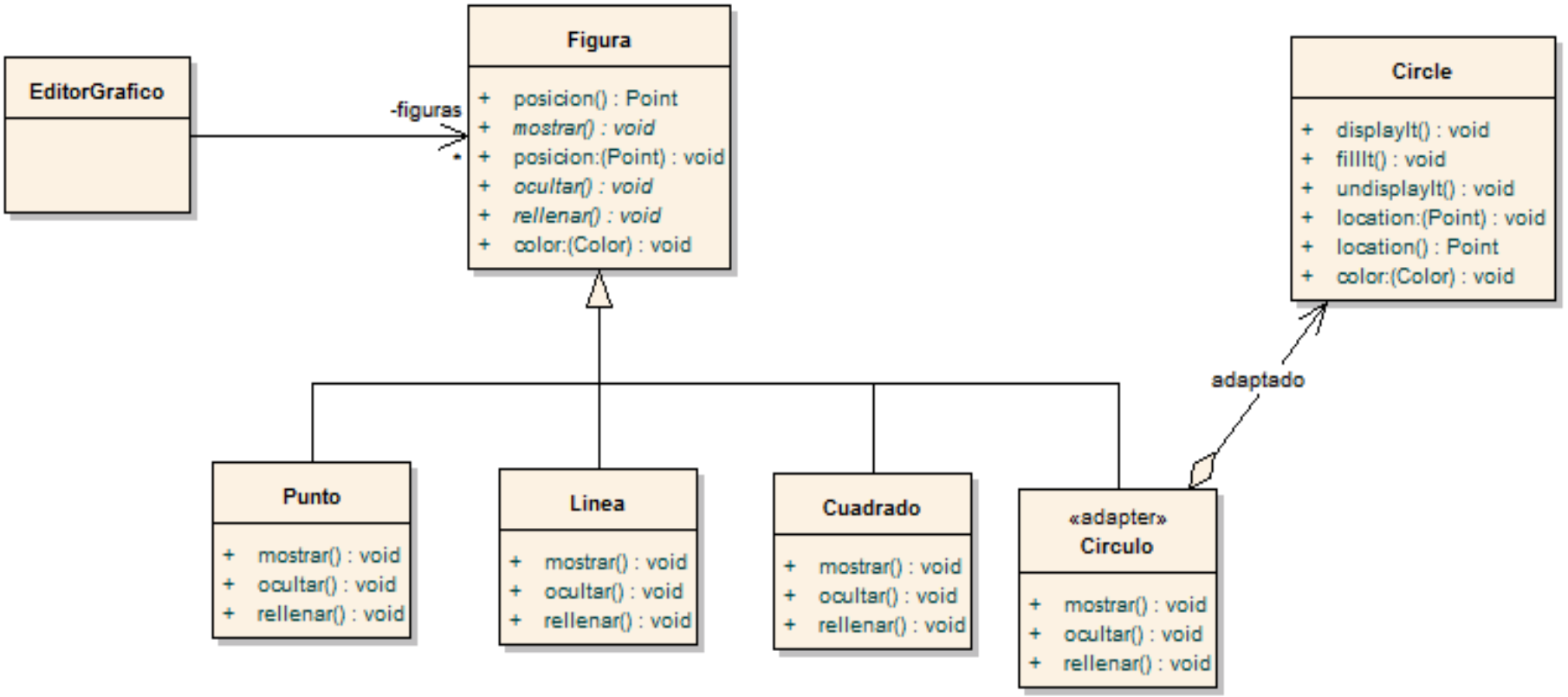
Patrones de diseño - adapter

- Esa misma idea se traslada a la POO.
- Objetos con diferentes interfaces se pueden "comunicar" usando adaptadores (adapters).
- El patrón de diseño también se conoce como "wrapper".

Patrones de diseño - adapter



Patrones de diseño - adapter



Patrones de diseño - adapter

Intención:

Convertir una interface de una clase en otra interfaz que un cliente espera.

Los adapters permiten a diferentes clases trabajar juntas que no podrían trabajar por sus interfaces incompatibles.

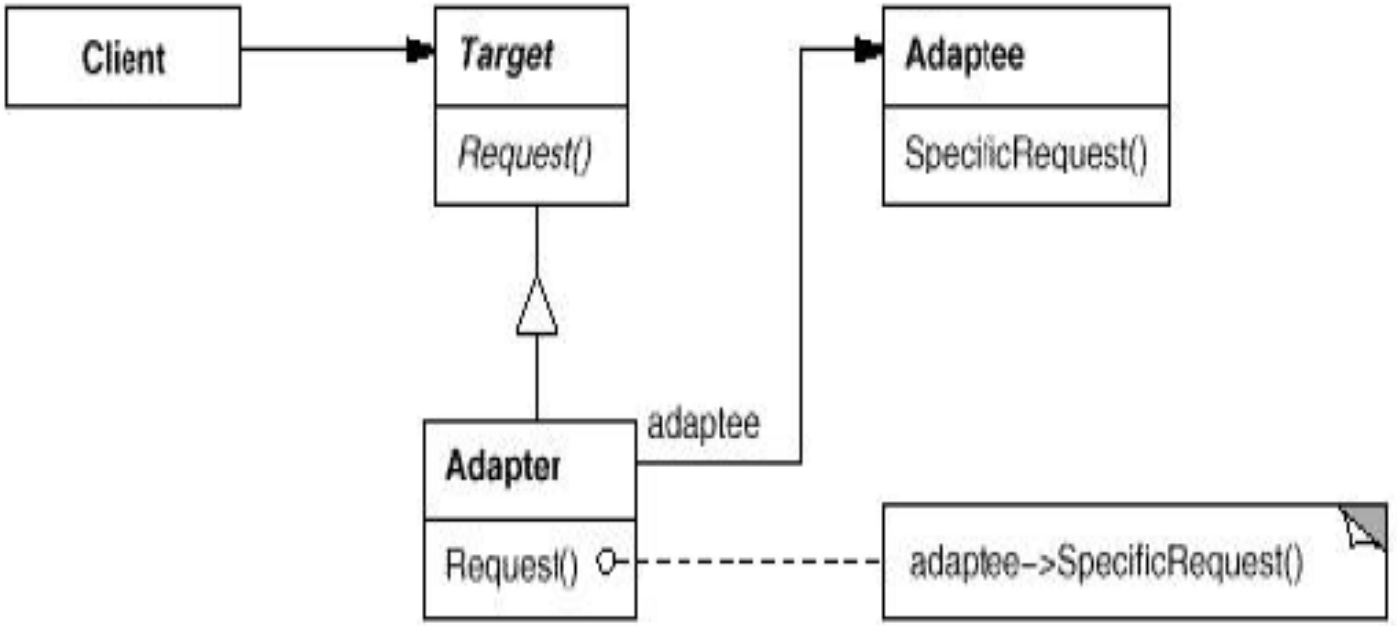
Aplicabilidad:

Cuando se quiere usar una clase y su protocolo no es la que se necesita.

Crear una clase reusable que coopera con clases no relacionadas con protocolos incompatibles.

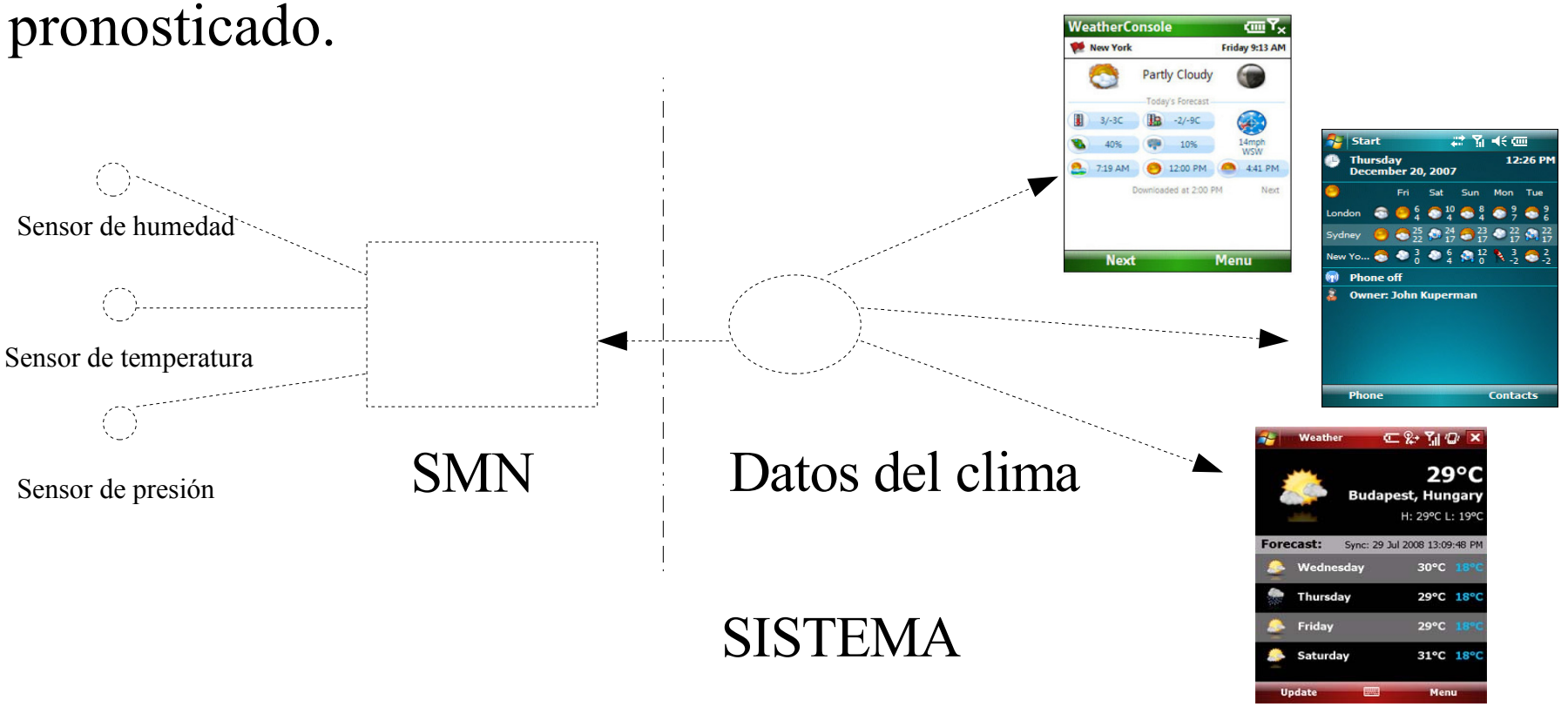
Patrones de diseño - adapter

- Estructura:



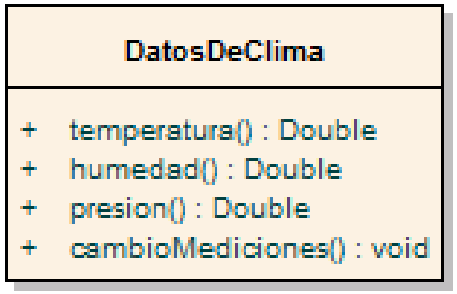
Patrones de diseño - observer

Supongamos que tenemos un dispositivo que lee condiciones del SMN. Este dispositivo luego debe mostrar los datos en otros tres dispositivos que muestran tiempo actual, estadítico y pronosticado.



Patrones de diseño - observer

Modelo



Los tres primeros métodos se usan para obtener los datos, y el tercero es llamado cada vez que se produce un cambio en las mediciones.

Debemos implementar las interfaces de los tres dispositivos

Puede haber más dispositivos en el futuro (escalabilidad).

Patrones de diseño - observer

```
>>cambioMediciones
        |hum temp pres|
        hum := self humedad. temp := self temperatura. pres := self presion
        displayCondicionesActuales actualizar(temp, temperatura,
          presion);
        displayEstadistico actualizar(temp, temperatura, presion);
        displayPronostico actualizar(temp, temperatura, presion);
```

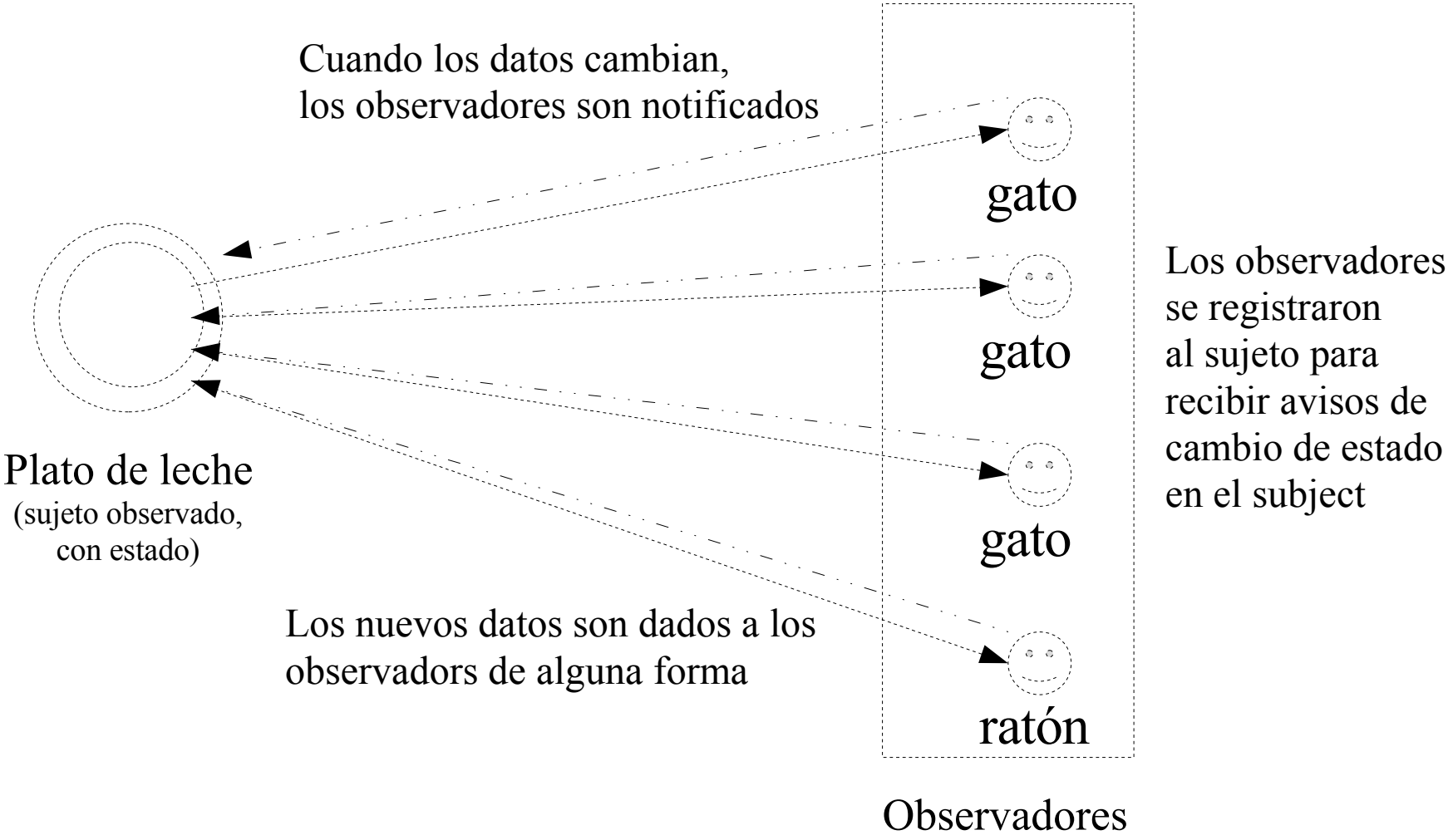
Se escribe código en función de los dispositivos que se conocen. Y si hay más?

Patrones de diseño - observer

Ideas:
Un *publisher* realiza avisos de un evento (ej. cambio de clima)
Varios *subscriptores* se subscriben al publisher, y cada vez que éste realiza un aviso, los subscriptores son notificados
Los subscriptores pueden desuscribirse, y pueden aparecer otros subscriptores.

publisher + subscribers = observer pattern

Patrones de diseño - observer



Patrones de diseño - observer

Intención:

Definir una relación uno-a-muchos entre un sujeto y varios observadores, de forma que sean notificados cuando se produce un cambio de estado en el sujeto y se actualicen.

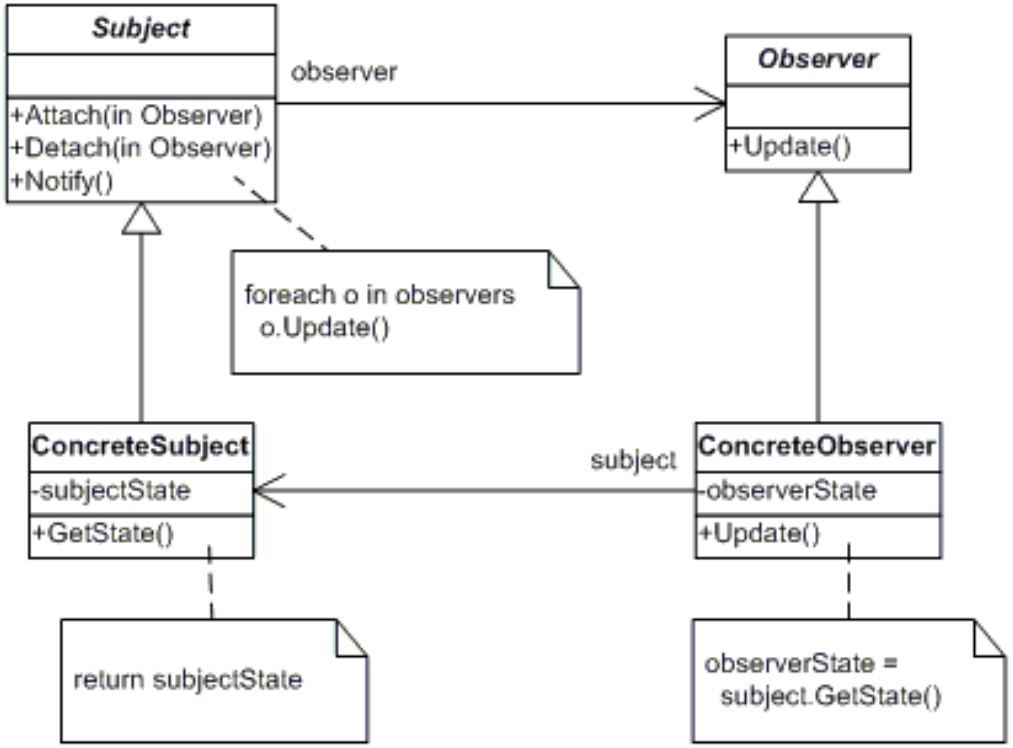
Aplicabilidad

Cuando el sujeto no sabe exactamente la cantidad de observadores.

Cuando el sujeto debe notificar a sus observadores sin interesarle realmente quiénes son.

Patrones de diseño - observer

Estructura del patrón:



Cómo sería la solución al problema inicial con observer?

Patrones de diseño - composite

Si quisiéramos representar expresiones algebraicas, vemos que existen:

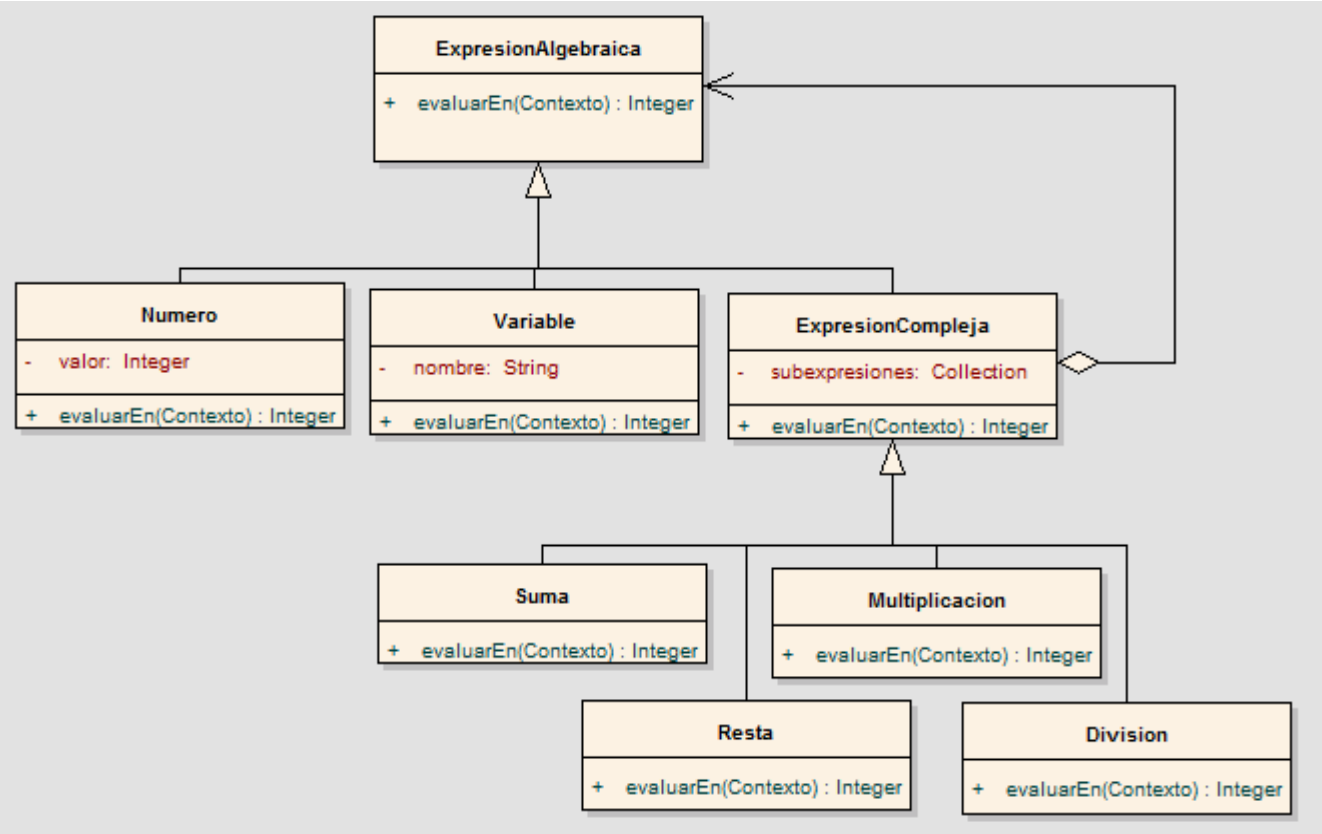
Expresiones simples (números y variables)

Expresiones Complejas (sumas, restas, etc)

¿Cómo logramos representar instancias de expresiones algebraicas, de forma que quien las use no tenga que preocuparse por el tipo de expresión algebraica que es?

Supongamos que quien las usa quiere saber su valor en un contexto determinado

Patrones de diseño - composite



Toda expresión algebraica *hereda* el protocolo definido en la clase raíz de la jerarquía.

Las clases complejas deben agregar métodos para agregar o quitar subexpresiones.

Patrones de diseño - composite

Intención:

Componer objetos en estructura de árbol para representar jerarquías de “parte-todo”. El cliente trata los objetos uniformemente.

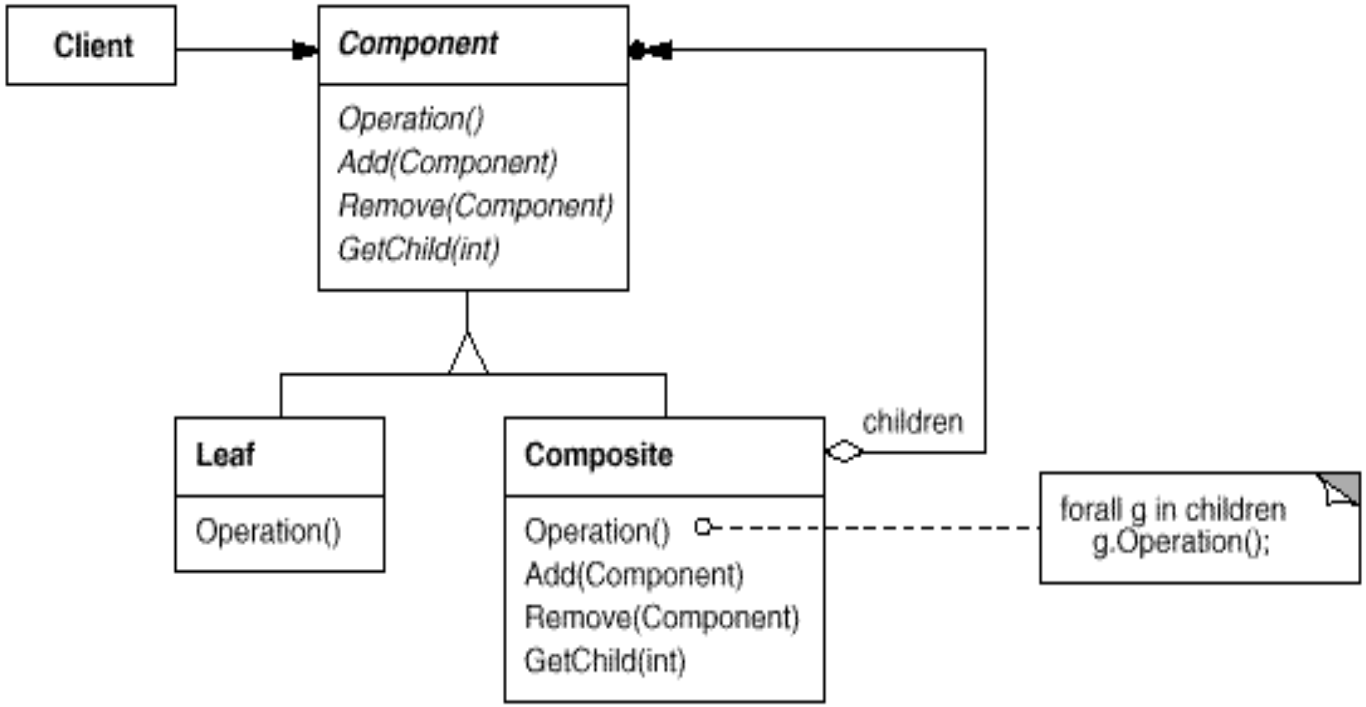
Aplicabilidad:

Representar jerarquías “parte-todo” de objetos.

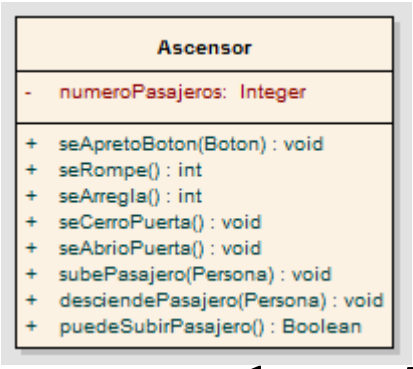
Hacer que el cliente ignore las diferencias entre objetos individuales y compuestos.

Patrones de diseño - composite

Estructura del patrón:

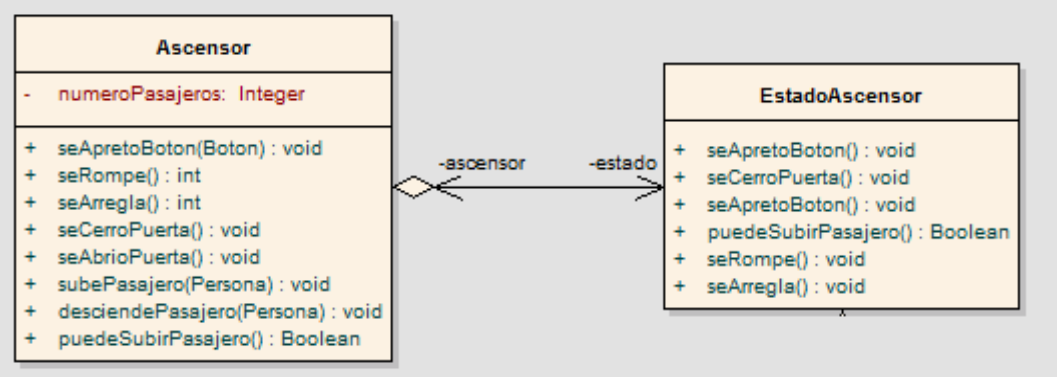


Patrones de diseño - state

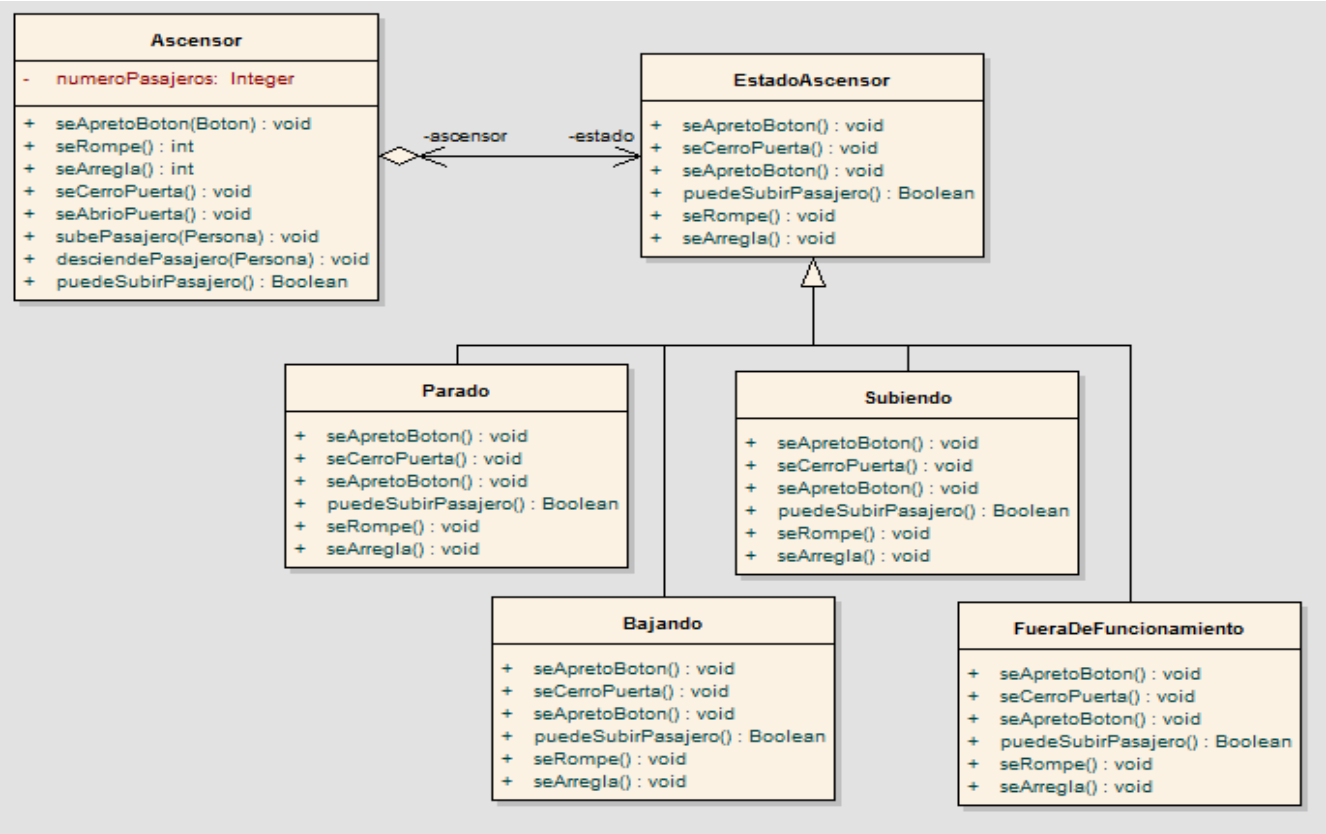


¿Cómo se implementa el método *seApretoBoton: unBoton* del ascensor? ¿El comportamiento es siempre el mismo? ¿De qué depende?

El ascensor *delegará* en su estado el comportamiento ante un suceso!



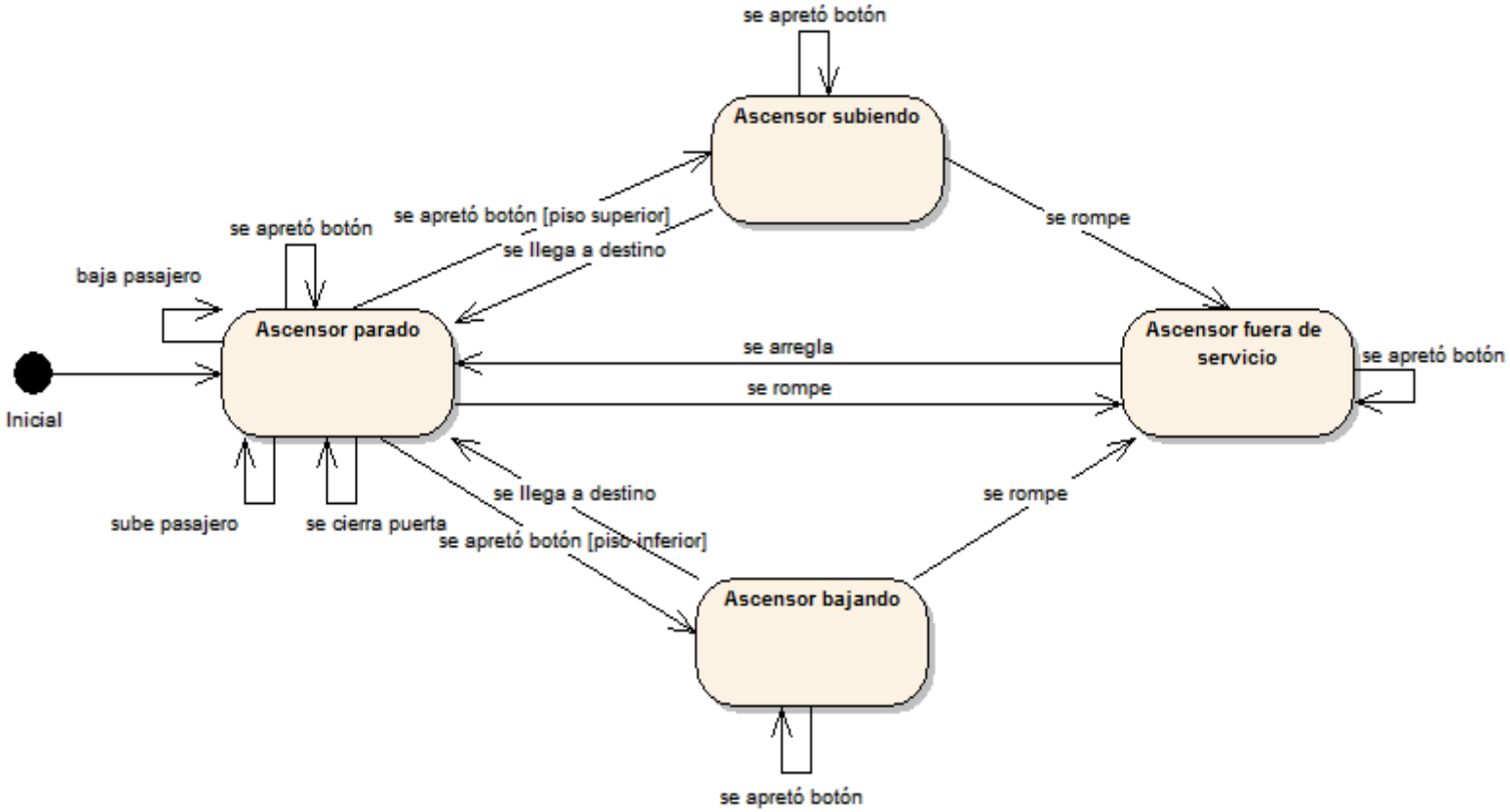
Patrones de diseño - state



Algunos estados harán que el ascensor cambie de estado, con lo cual ese estado dejará de ser referenciado.
El estado podría recibir al ascensor como parámetro, y así poder compartirse entre varios ascensores

Patrones de diseño - state

Qué relación encuentra entre el diseño de clases y el diagrama de estados visto anteriormente?



Patrones de diseño - state

Teniendo un diagrama de estados de la entidad, es sencillo diseñar una solución OO:

La entidad conoce un estado y le delega parte de sus responsabilidades (¿cuáles?)

Los estados del diagrama se traducen a clases en una jerarquía de estados de la entidad.

Las transiciones se traducen a métodos en la entidad y en las clases estado, siguiendo el patrón *State*.

Los estados deben responder (haciendo nada o con un error) por transiciones que no están explícitamente en el diagrama (ej. transición baja pasajero).

Patrones de diseño - state

Intención:

Permitir a un objeto alterar su comportamiento de acuerdo a su estado interno. El objeto parecerá cambiar de clase.

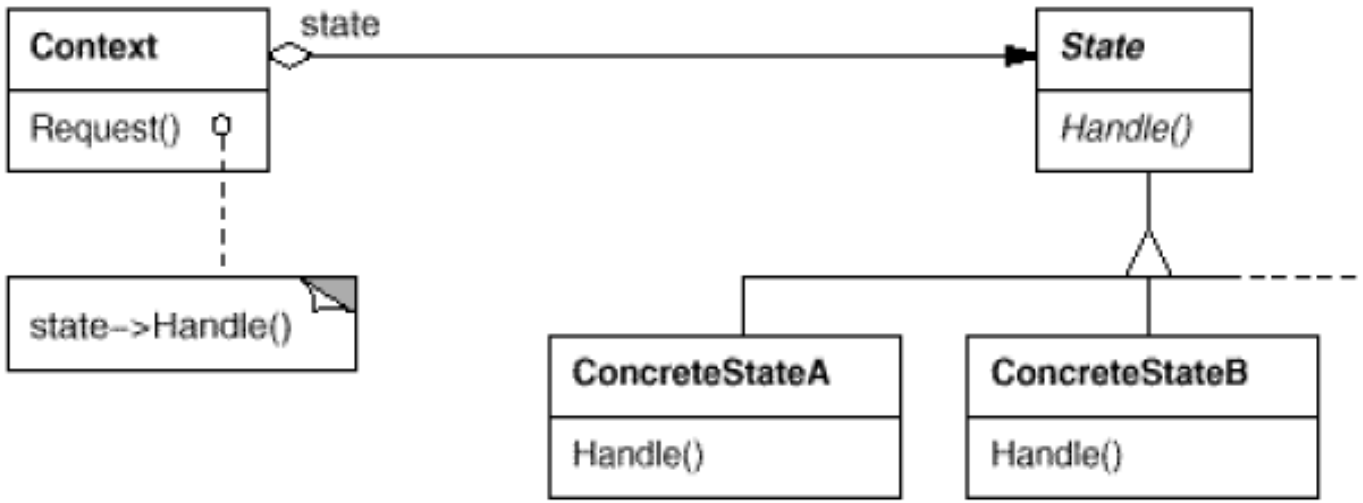
Aplicabilidad:

El comportamiento del objeto depende del estado, y éste debe cambiar dinámicamente.

Cuando un método tiene muchas sentencias condicionales que dependen del estado del objeto.

Patrones de diseño - state

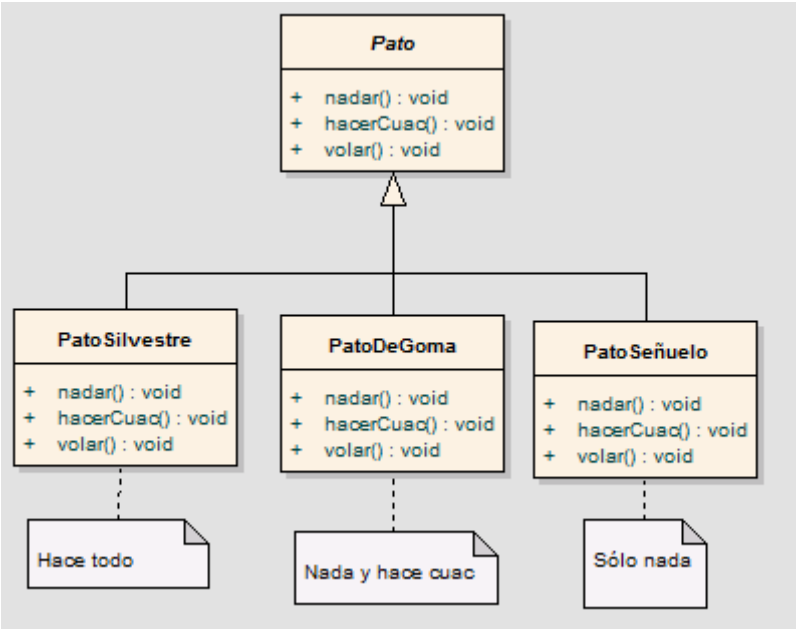
Estructura del patrón:



Patrones de diseño - strategy

Supongamos que queremos modelar un pato
Un pato puede volar, hacer cuac y nadar, entre otras cosas, pero lo puede hacer de muchas formas.
Hacemos una jerarquía para representar los distintos patos...

Patrones de diseño - strategy



Cómo hacemos para hacer que un pato cambie su forma de volar o empiece a nadar?

¿Qué problemas encontramos utilizando la jerarquía de herencia?

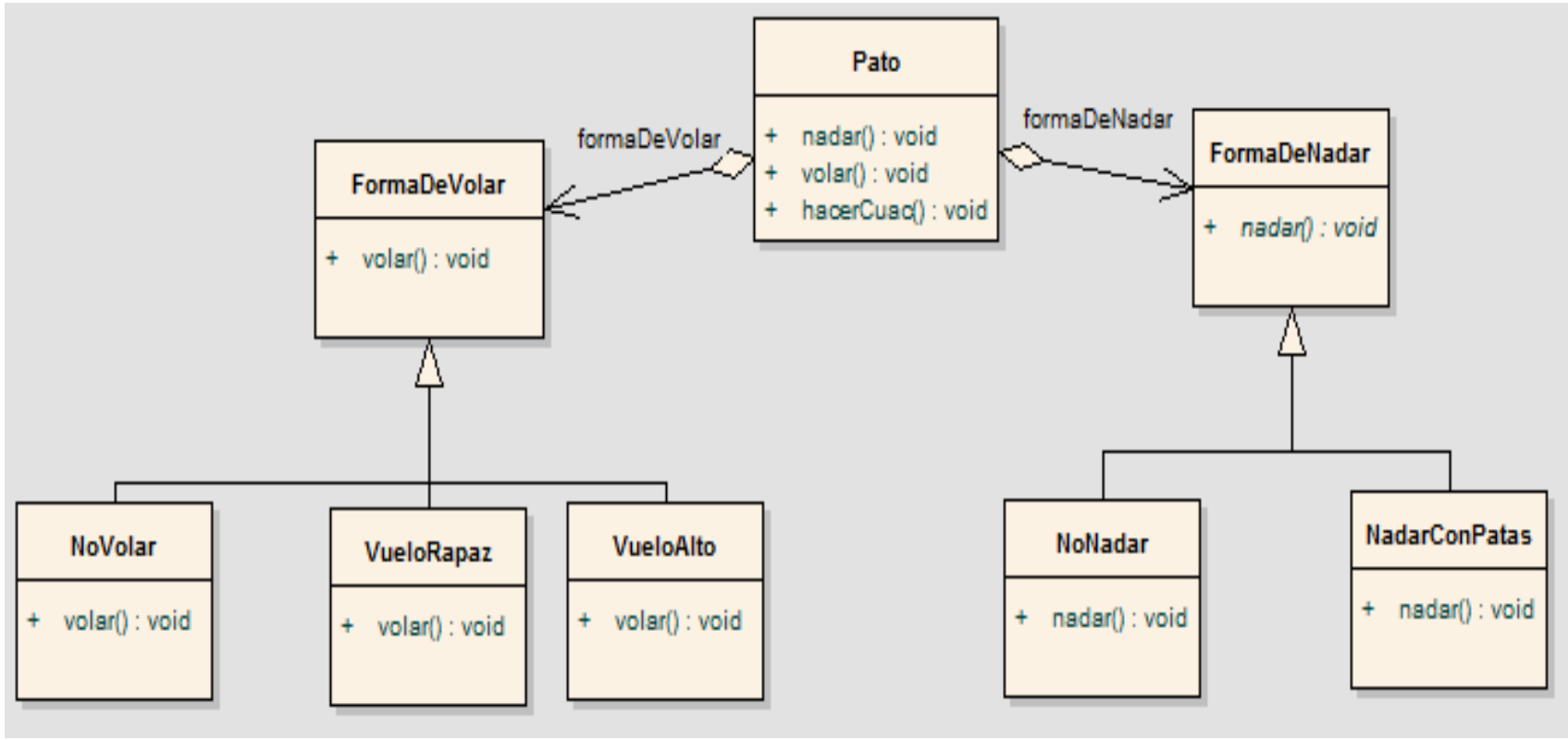
Qué problemas nos está trayendo la herencia en esta solución?

Patrones de diseño - strategy

Solución: no usemos herencia, sino composición!
Desacoplamos la forma en que un pato vuela, hace cuac y nada en jerarquías, y cada pato conocerá instancias particulares de cada jerarquía de clases.
Agregar una nueva forma de volar implica crear una nueva subclase en la jerarquía de ordenadores. Cómo impactaría esto si seguimos usando herencia?

La jerarquía de strategy podría usar *template method*.

Patrones de diseño - strategy



Patrones de diseño - strategy

Intención:

Definir una familia de algoritmos, encapsular cada uno y hacerlos *intercambiables*.

Aplicabilidad:

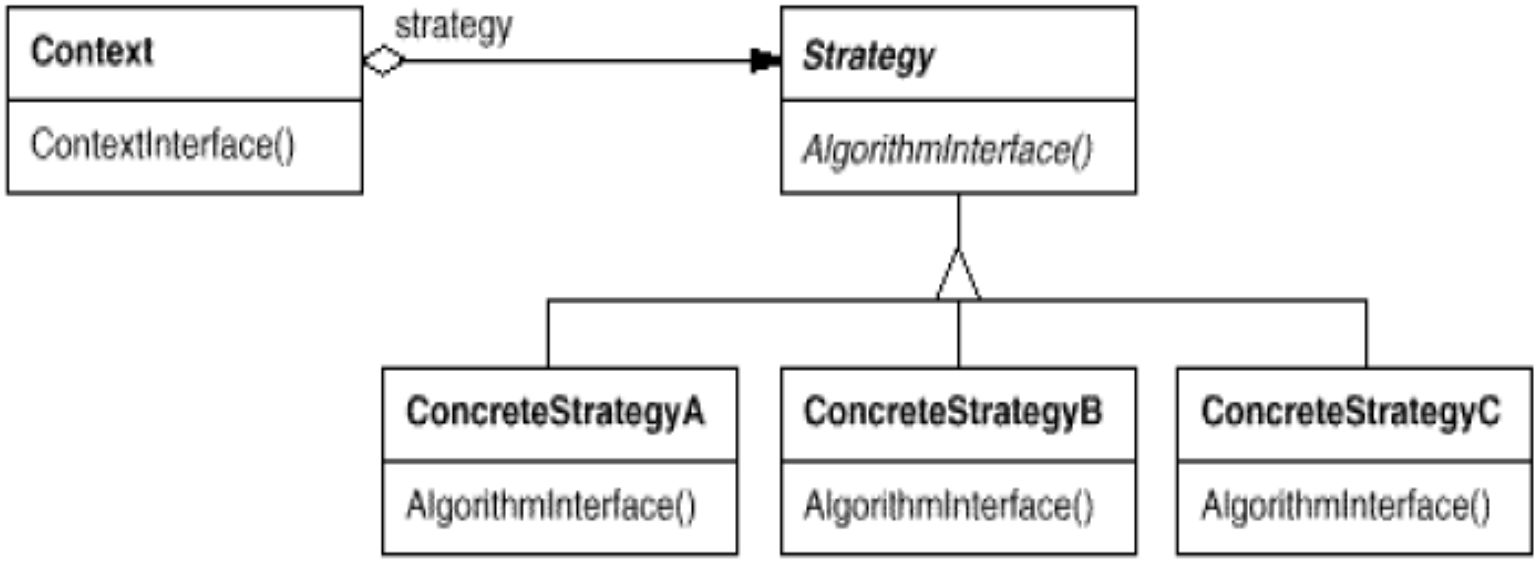
Cuando muchas clases relacionadas difieren en el comportamiento

Cuando se necesitan diferentes variantes de un mismo algoritmo

Cuando un metodo tiene sentencias condicionales para definir el comportamiento

Patrones de diseño - strategy

Estructura del patrón:

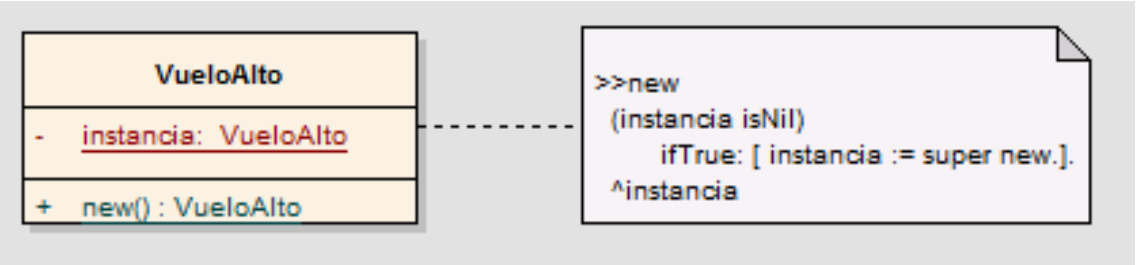


Patrones de diseño - singleton

En el ejemplo anterior de los patos, observamos que se crea una instancia que pertenece a la jerarquía de FormaDeVolar cada vez que se crea un pato.
Sin embargo la instancia de FormaDeVolar no tiene información sobre el pato, y sólo encapsula comportamiento.
Sería bueno evitar crear más de una instancia, y compartir la instancia de VueloAlto (por ejemplo) entre diferentes instancias de patos.

Patrones de diseño - singleton

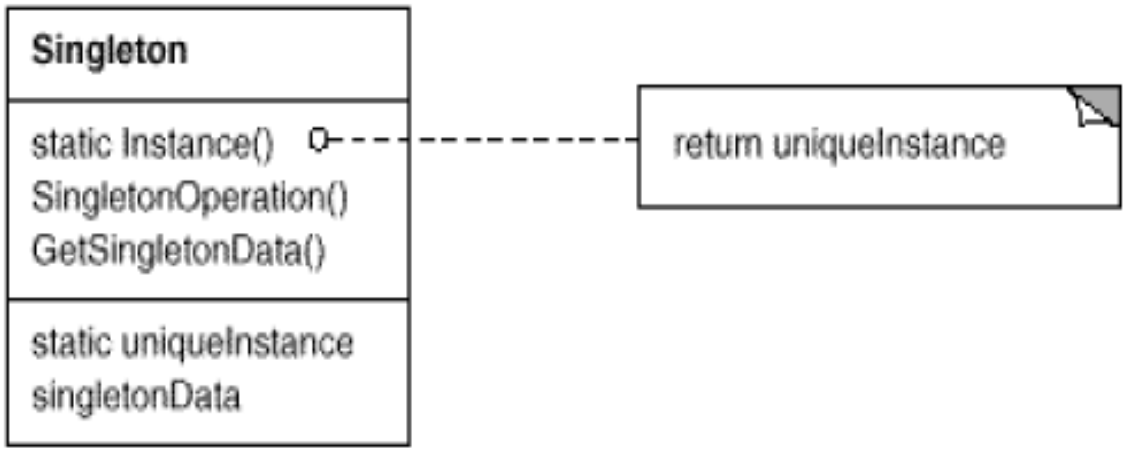
Las subclases de FormaDeVolar deben definir una sola forma de crear instancias, y retornar siempre la misma.



¿Cómo vemos (como objetos) a dos patos que tienen la misma forma de volar?

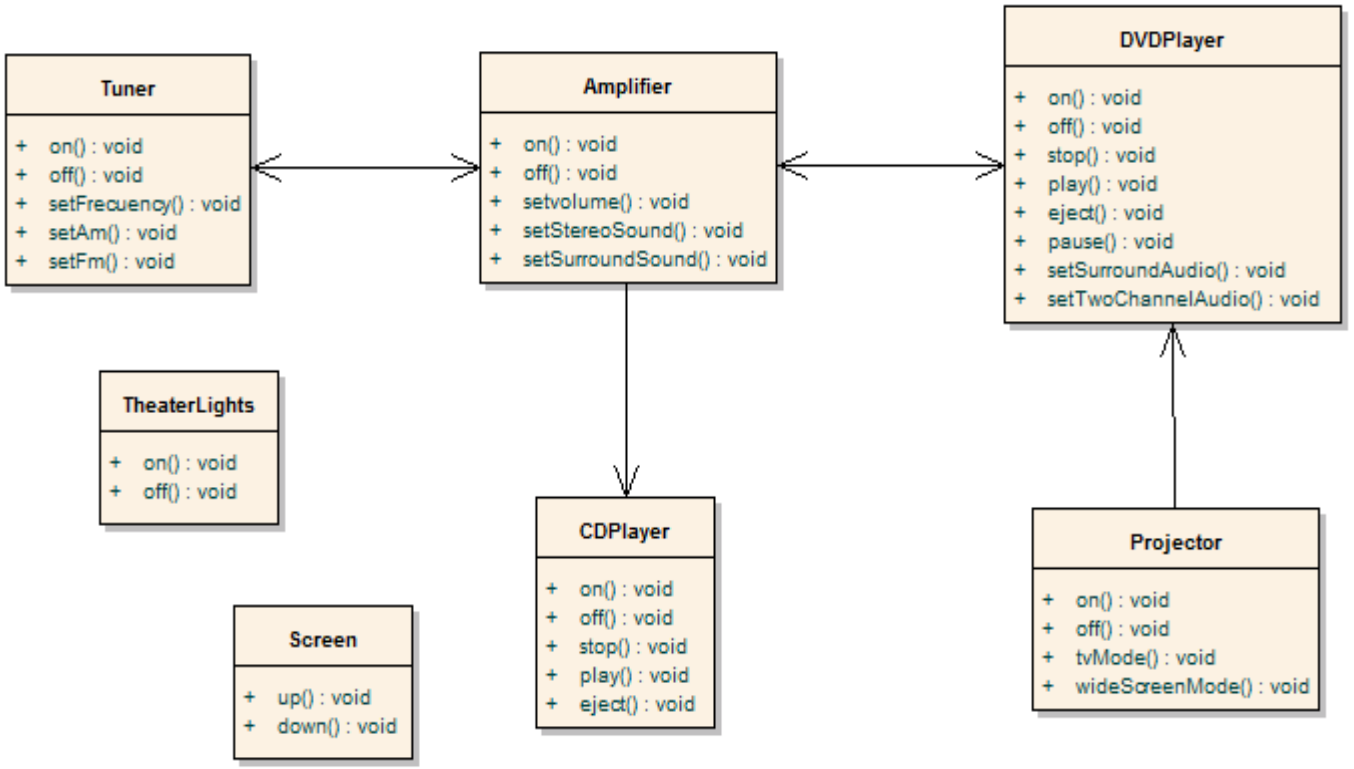
Patrones de diseño - singleton

Estructura del patrón:



Patrones de diseño - facade

Queremos instalar nuestro nuevo home theatre.



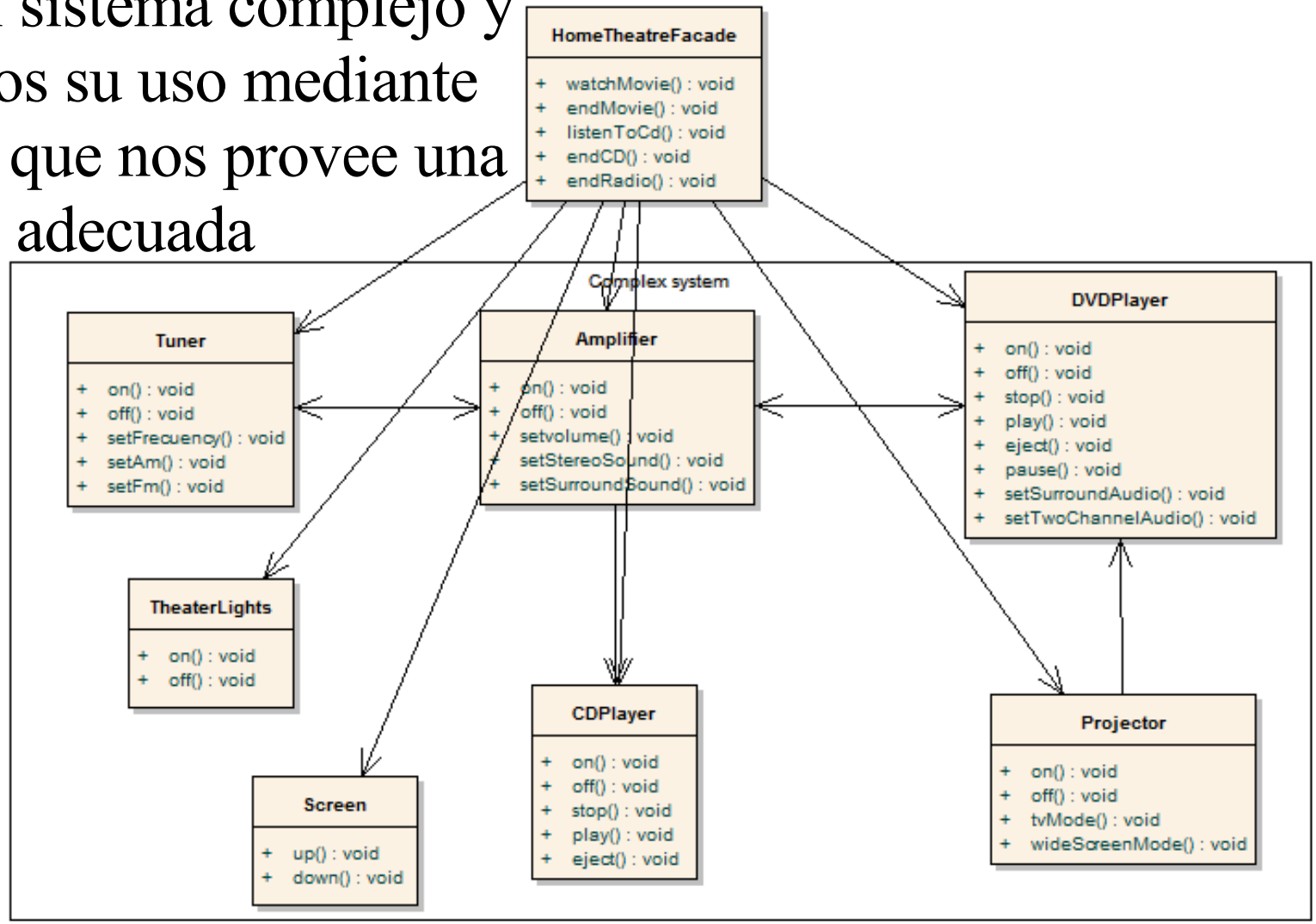
Demasiadas clases, muchas interacciones, mucho por aprender.

Patrones de diseño - facade

Para ver una película debemos:
Apagar las luces
Prender el proyector
Conectar el proyector al dvd
Poner el proyector en wide-screen mode
Prender el amplificador
Conectar el amplificador al reproductor de DVD
Poner el amplificador en modo surround
Poner el volumen del amplificador a gusto
Encender el reproductor de DVD
Insertar y reproducir la película en el DVD
Y para apagarlo... y para actualizar alguna parte!

Patrones de diseño - facade

Tomamos el sistema complejo y simplificamos su uso mediante una fachada que nos provee una interfaz más adecuada y fácil de usar.



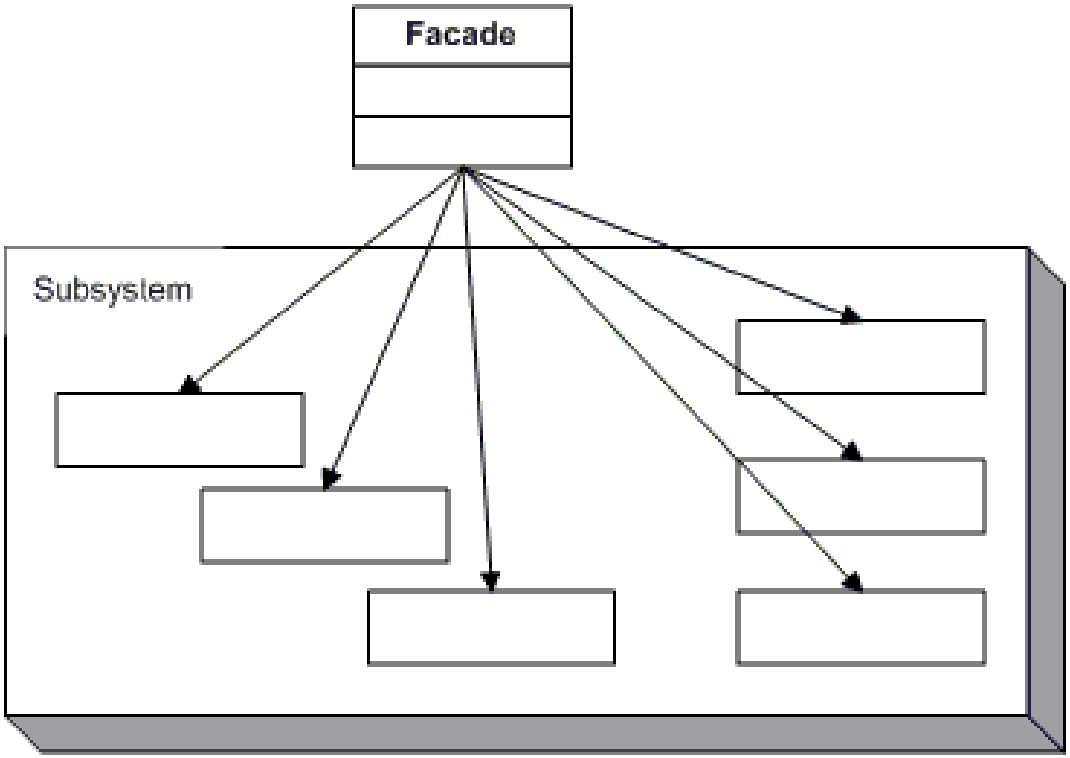
Patrones de diseño - facade

Intención:
Simplificar la interfaz con un sistema complejo para las tareas que se quieren realizar con él.

Aplicabilidad:
Proveer una interfaz simple para un sistema complejo. Esta interfaz es suficiente para los clientes, pero sin embargo ellos podrían no usar el facade
Desacoplar las clases del subsistema de sus clientes y otros subsistemas, promoviendo indenpendencia y portabilidad.

Patrones de diseño - facade

Estructura del patrón:



Patrones de diseño

Otros patrones:

Iterator
Command
Abstract factory
Adapter
Bridge
Decorator
Visitor
Role object
Rule object
Null object
Type object
Object pool
Lazy initialization

Flyweight
Chain of responsability
Factory method
Interpreter
Mediator
Memento
Prototype
Proxy
....

Patrones de diseño

Existen muchos más patrones de diseño.
En este curso se dieron algunos de los patrones y no se profundizó en el análisis de los beneficios y precauciones del uso de cada patrón.
No considerar que hacer uso de un patrón es garantía de buena solución. El *sobrediseño* usando patrones puede llevar a malas soluciones.
Se recomienda a los interesados leer el libro de Gamma et. al.

Reflexiones iniciales... finales

¿Qué es la programación?

¿Qué es un programa?

¿Cuáles son las tareas de un programador?

¿Qué es un paradigma de programación?

Fin de la teoría