v 1.9

Tema 5
Diseño de clases

Programación Orientada a Objetos

José Francisco Vélez Serrano

Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Índice

- Diseño Dirigido por Pruebas
- Descubrimiento de abstracciones
- Aspectos a considerar definiendo relaciones
- Relaciones entre objetos
- Relaciones de herencia entre clases
- Principios de la Programación orientada a objetos

Diseño Dirigido por Pruebas

En esta sección se presenta el concepto de prueba unitaria y el diseño de programas dirigido por pruebas.

Pruebas unitarias y de integración

Una prueba unitaria o test unitario es una forma empírica de comprobar el funcionamiento de una unidad individual de algún programa.

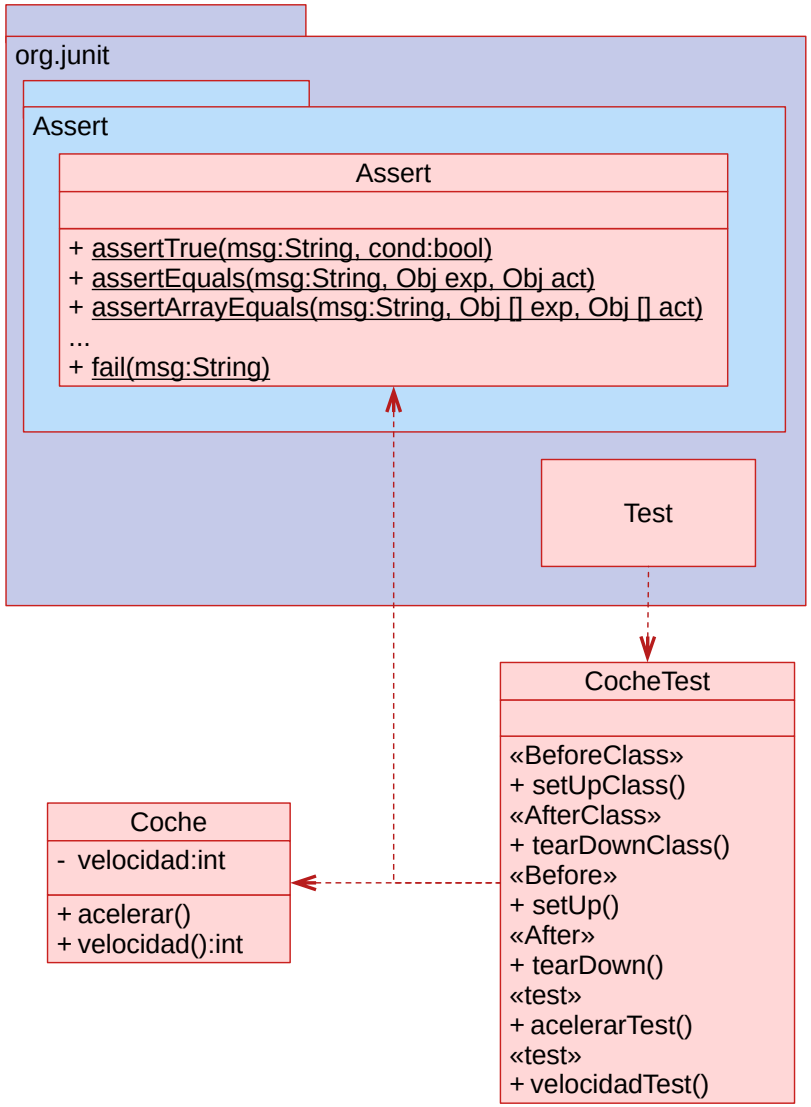
Una prueba de integración comprueba que un conjunto de componentes funcionan correctamente en grupo, comprobando que se comunican entre ellos correctamente.

Los lenguajes de programación modernos suelen incorporar bibliotecas para facilitar la creación de pruebas unitarias y de integración.

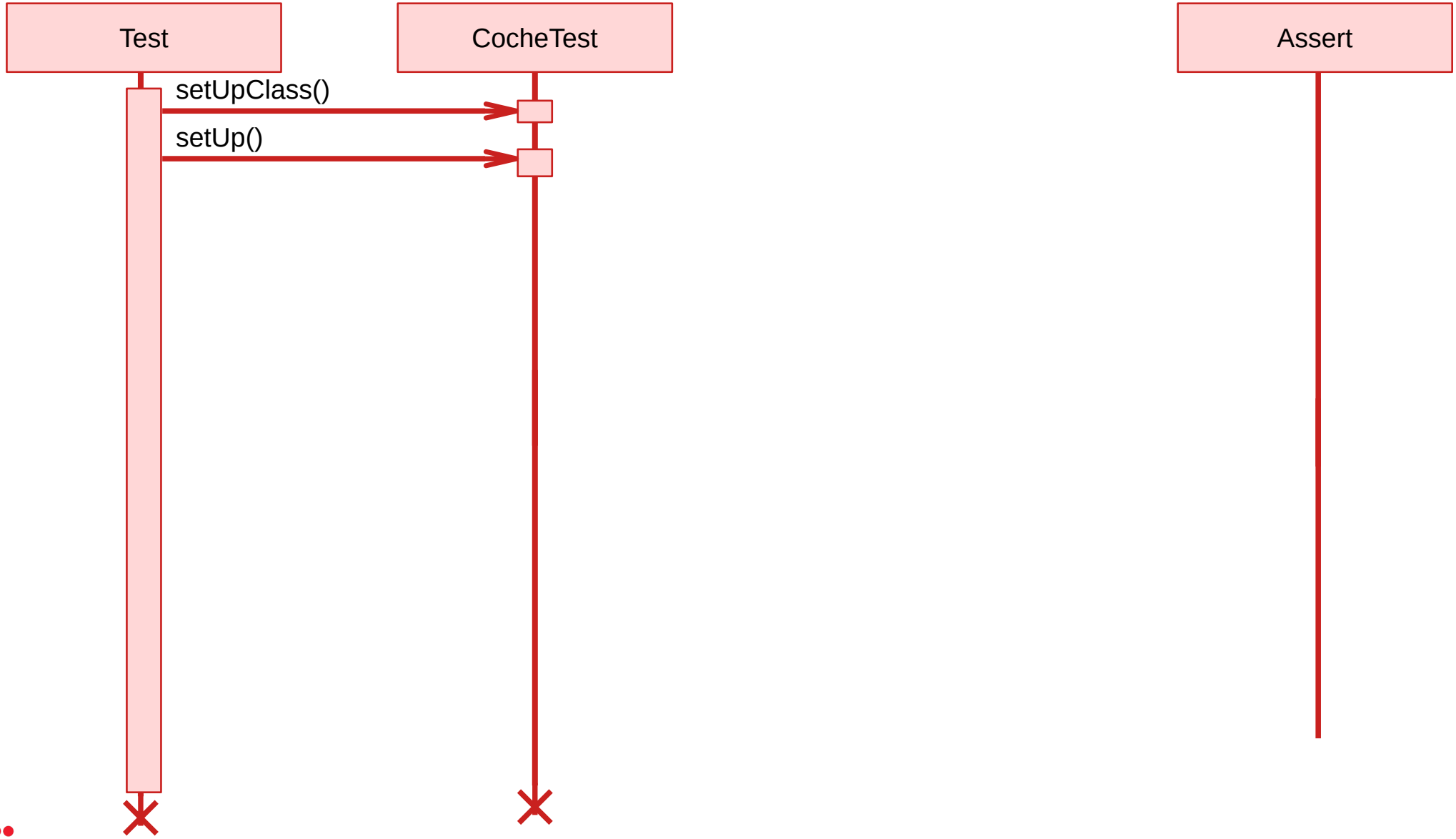
Diseño Dirigido por Pruebas

Las pruebas unitarias en Java con JUnit 4

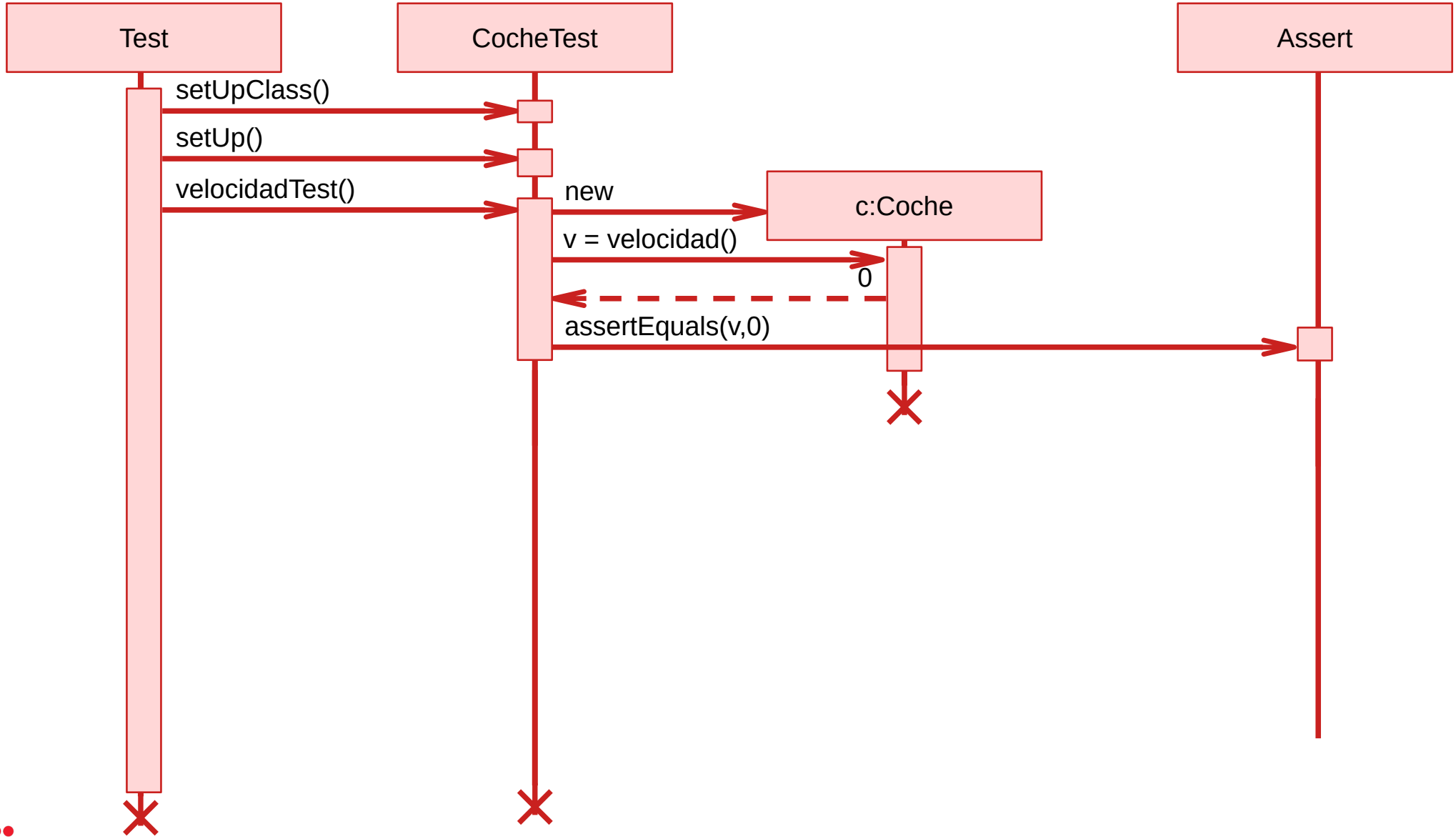
- Java dispone de diferentes bibliotecas de prueba.
- JUnit es uno de los más conocidos.
- La última versión es JUnit5 aunque Netbeans aún no la soporta.
- JUnit usa anotaciones para invertir la dependencia entre la clase de prueba y la clase Test.



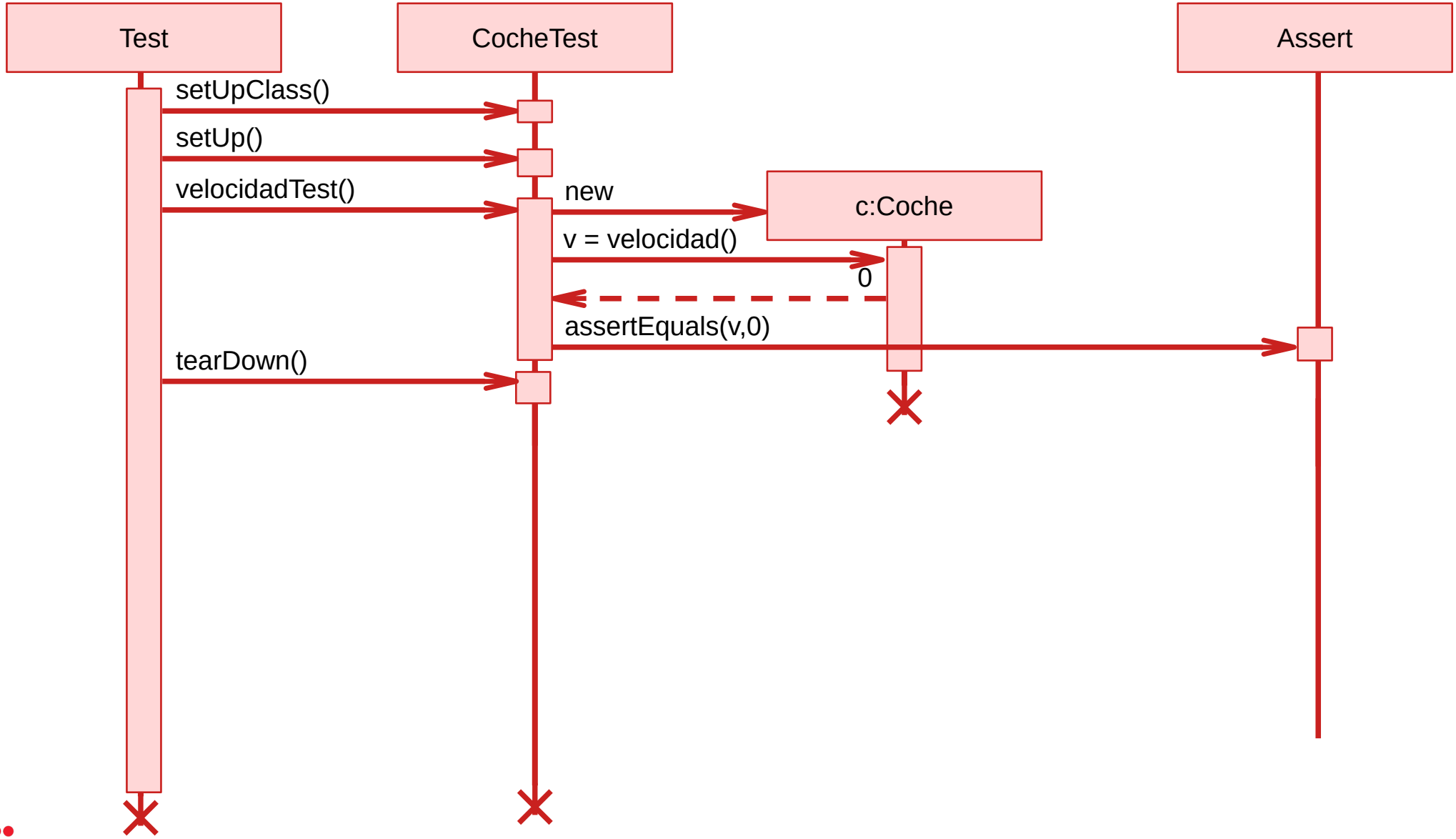
Ejemplo de una prueba con JUnit4



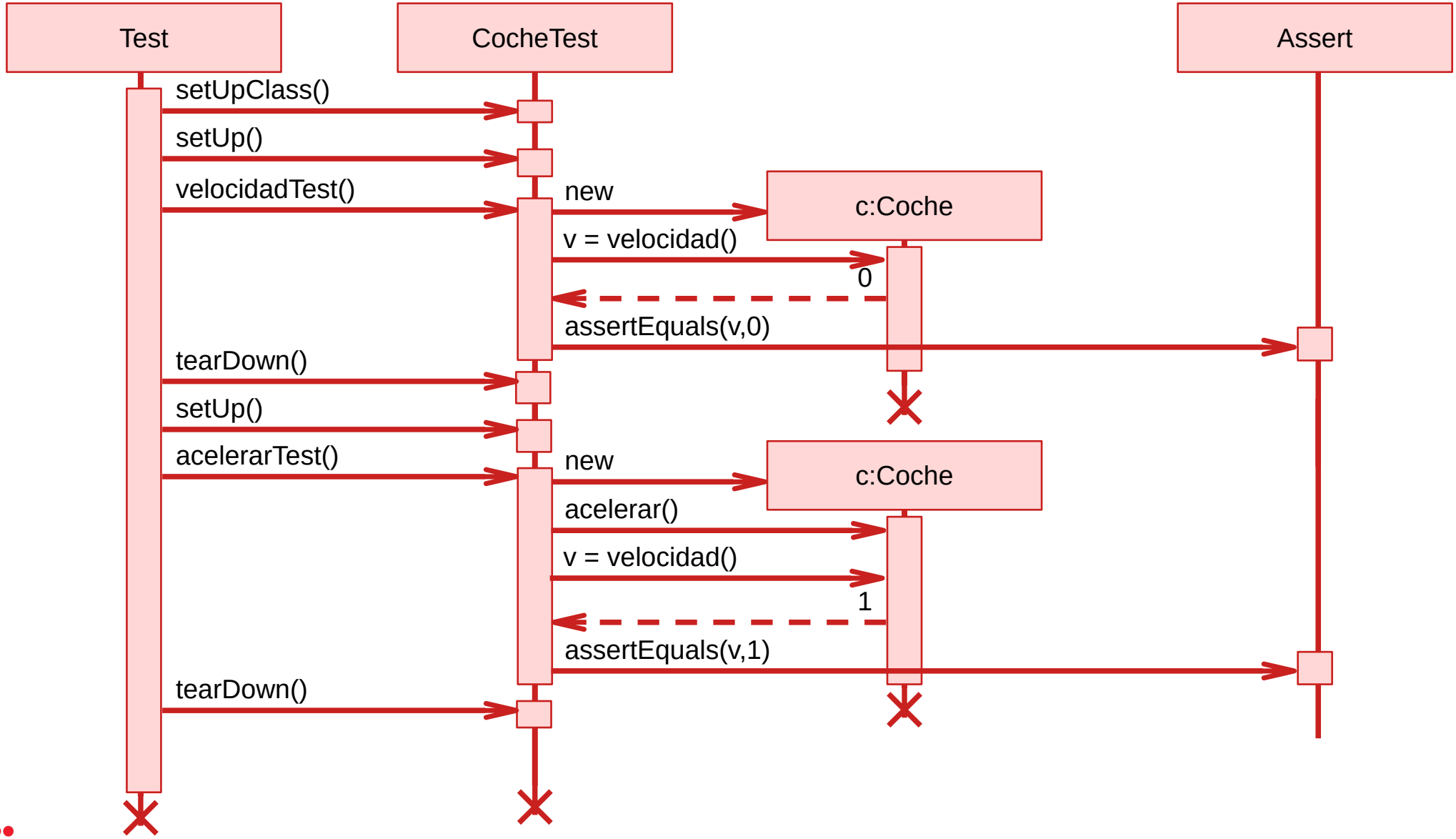
Ejemplo de una prueba con JUnit4



Ejemplo de una prueba con JUnit4

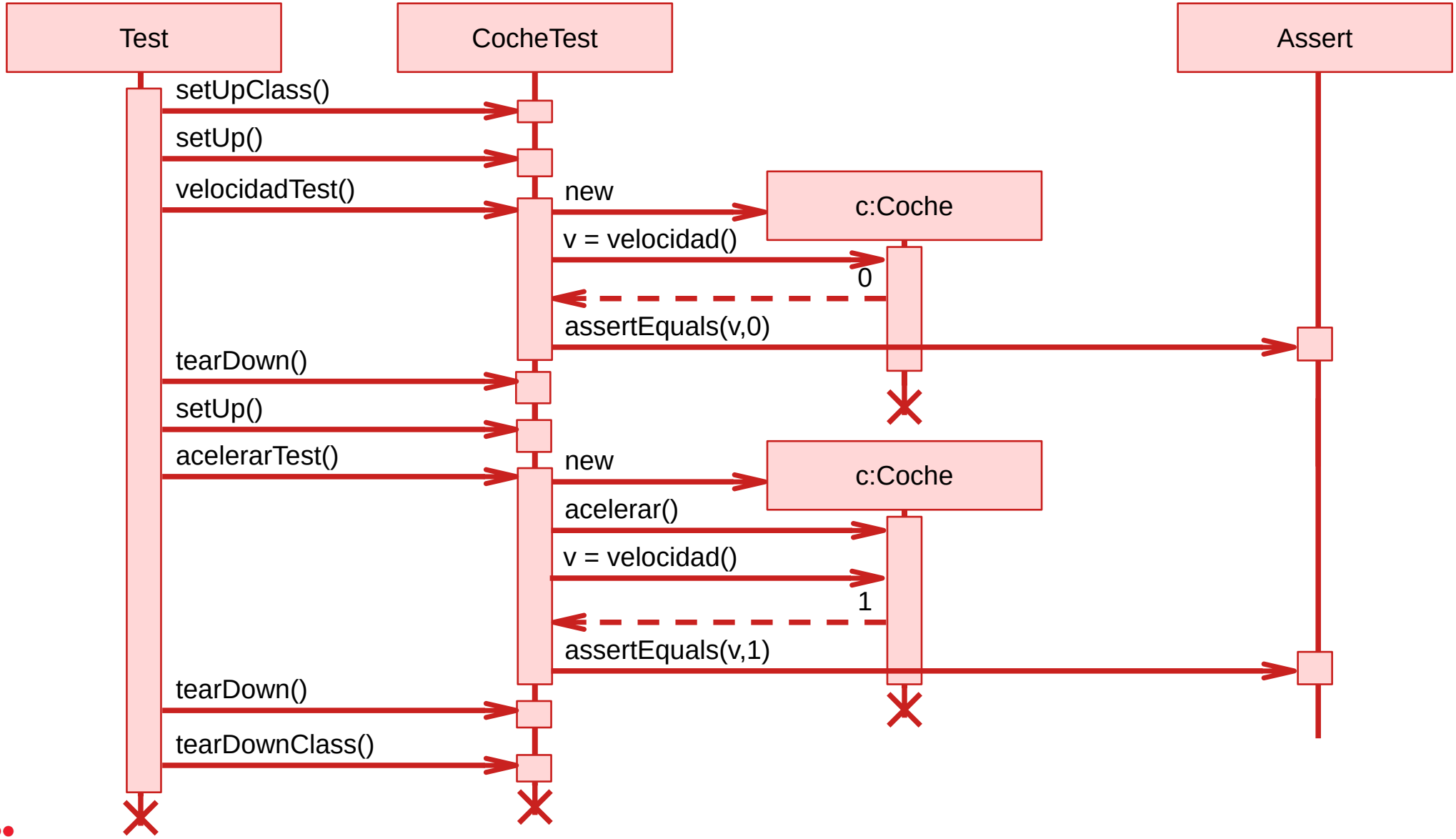


Ejemplo de una prueba con JUnit4



Diseño Dirigido por Pruebas

Ejemplo de una prueba con JUnit4



Principios del Diseño Dirigido por Pruebas

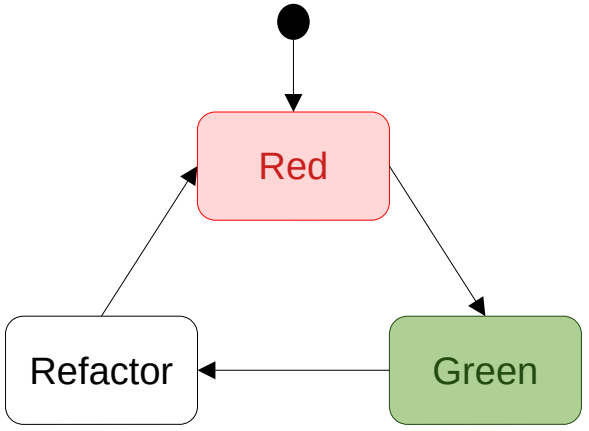
El diseño dirigido por pruebas (TDD) es una herramienta que ayuda a crear software funcional, bien definido y fácil de mantener. Se basa en tres reglas:

1. No está permitido escribir código de producción a no ser que una prueba falle.
2. No está permitido escribir más código en una prueba unitaria que el necesario para que ésta falle. Considerándose los errores de compilación como fallos.
3. No está permitido escribir más código de producción que el necesario para que una prueba que falla deje de fallar.

Fases del Diseño Dirigido por Pruebas

Las tres reglas del TDD llevan a lo que se denomina Red-Green-Refactor.

- Red: Se corresponde a la fase en la que la prueba falla.
- Green: Se corresponde a la fase en la que la prueba deja de fallar.
- Refactor: Esta etapa es la que corresponde a la parte de diseño del desarrollo. Una vez que la prueba pasa se modifica el código para que sea más comprensible y su diseño sea más evidente. En esta fase se eliminan duplicaciones, se extraen métodos, se crean nuevas clases...



Diseño Dirigido por Pruebas

Objetivos y ventajas

- Breve tiempo de compilación y el de ejecución de las pruebas para poder probar cada poco.
- El código transita por una sucesión de estados estables.
- Los errores no precisan de largos procesos de depuración para localizarlos, pues normalmente el error se encuentra en el último código añadido.
- El código desarrollado será más fácil de probar.
- El foco está en el diseño incremental.

Descubrimiento de abstracciones

La programación orientados a objetos consiste en gran medida en crear las abstracciones adecuadas. En este capítulo se analizan tácticas para encontrarlas.

Diseño Bottom-up

Bottom-up empieza por objetos de bajo nivel que resuelven problemas finales. Estos objetos se combinan para crear otros de nivel superior para problemas más generales.

Es más cómodo para los desarrolladores, pues no requiere visión de alto nivel para empezar.

El enfoque bottom-up se suele usar para empezar por partes consideradas excitantes o arriesgadas.

Por ejemplo:

- Hacer un objeto que dibuja fuego y luego crear un juego que lo usa.
- Crear objeto que cifra datos y luego pensar su uso desde un sistema de mensajería seguro.

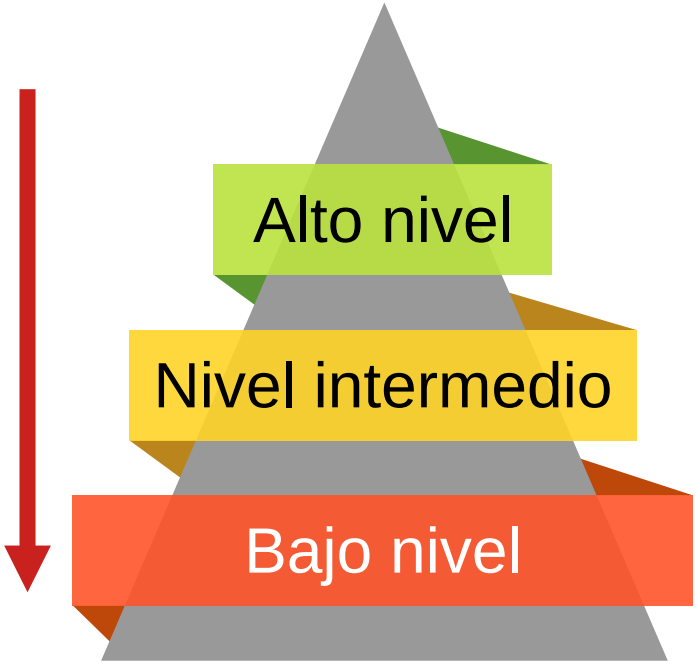
Descubrimiento de abstracciones

Diseño Top-down

En el enfoque Top-down se plantean objetos de alto nivel que resuelven los problemas más generales.

Luego, estos objetos de alto nivel se van descomponiendo sucesivamente en objetos de más bajo nivel que detallan cómo se resuelve cada problema.

Este es el enfoque cuando creamos un facsímil del programa completo y vamos dotando de funcionalidad a sus partes constituyentes.

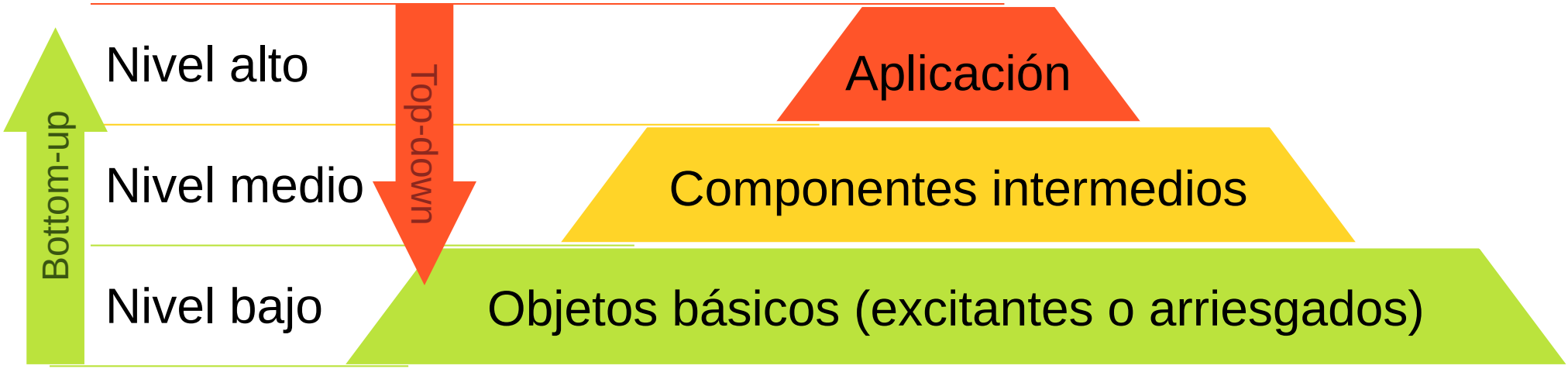


Descubrimiento de abstracciones

Diseño top-down

En los proyectos de software más orientados al cliente se suele seguir un enfoque Top-Down.

En la práctica se combinan ambos enfoques al crear programas. La POO facilita su combinación, ya que permite usar abstracciones que en cierta medida son independientes del nivel al que se usan.



Descubriendo abstracciones en el análisis

Muchas de las abstracciones necesarias en cada nivel surgen de los documentos de análisis del problema (crear dichos documentos queda fuera del alcance del curso, por lo que normalmente se proporcionan estos análisis en forma de enunciados de problemas).

Tácticas para descubrir objetos partiendo de los documento de análisis:

- Buscar sustantivos (objetos) y verbos (métodos u objetos).
- Buscar sistemas o dispositivos clientes o servidores.
- Buscar eventos que se producen contra o en el sistema.
- Buscar roles de gente que interactúa con el sistema.
- Buscar elementos organizativos del sistema o del cliente.
- Piensa en los modelos de datos, de pantallas, de control...
- Imaginar objetos que resuelvan problemas.

Asignando responsabilidad a las abstracciones

Es importante asignar responsabilidades claras a cada objeto.

Para ello también hay tácticas como:

- Crear un teatrillo poniéndose en el lugar de los objetos al resolver los problemas.
- Crear tarjetas Clase / Responsabilidad / Colaboradores (CRC).
- Crear diagramas plasmando objetos, responsabilidades y sus interacciones.

Al asignar responsabilidades suelen aparecer nuevas abstracciones.

Descubrimiento de abstracciones

Medita sobre las abstracciones necesarias

No introduzcas abstracciones que no necesitas, aunque estas abstracciones existan en la realidad. No estás modelando la realidad, estas modelando un software que resuelve un problema.

Piensa en las propiedades que debe tener una abstracción:

- Poco acoplamiento
- Cohesión
- Completitud y suficiencia
- Primitividad

Aspectos a considerar definiendo relaciones

La programación orientados a objetos consiste en definir abstracciones. En este capítulo se analizan aspectos a considerar al definirlas.

Aspectos a considerar definiendo relaciones

Versatibilidad

Un objeto se dice que es versátil si puede combinarse con otros objetos de diversas maneras para dar lugar a diferentes comportamientos.

Por ejemplo, una clase Pila que pueda almacenar cualquier tipo de objetos es una clase más versátil que una clase Pila que sólo permita almacenar números enteros.

ObjectStack **VS** IntStack

Aspectos a considerar definiendo relaciones

Versatibilidad

En general, es preferible hacer código versátil, pues facilita su reutilización en diferentes problemas. Sin embargo, un código excesivamente versátil puede ser que se ajuste poco a un problema particular que se esté resolviendo y que, con ello, se dificulte la programación en vez de simplificarla.

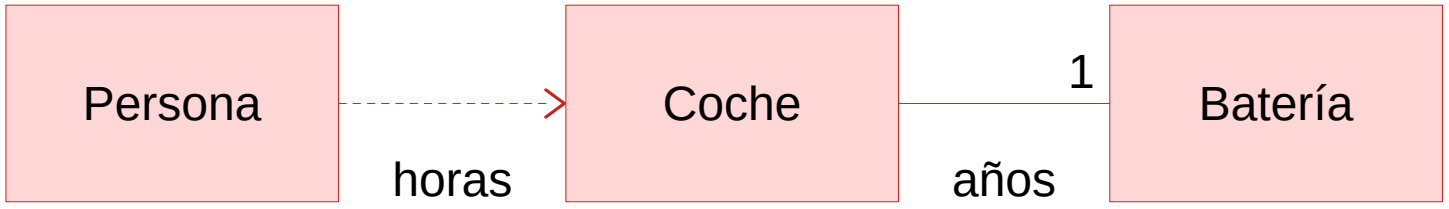
Por tanto, al desarrollar un programa, es importante estudiar cuál es el nivel de versatilidad que interesa para las clases e interfaces que se van a crear, en el contexto del problema que se está resolviendo.

Aspectos a considerar definiendo relaciones

Temporalidad

Cualquier relación entre objetos o entre clases tiene una duración temporal.

- Hay relaciones entre objetos que se dan en un ámbito muy restringido (por ejemplo dentro de un método).
- Hay relaciones entre objetos que abarcan toda la vida de los objetos (por ejemplo en las propiedades).



Aspectos a considerar definiendo relaciones

Temporalidad

Debe prestarse atención al estudio de la temporalidad de una relación porque ayuda a definir el tipo de relación.

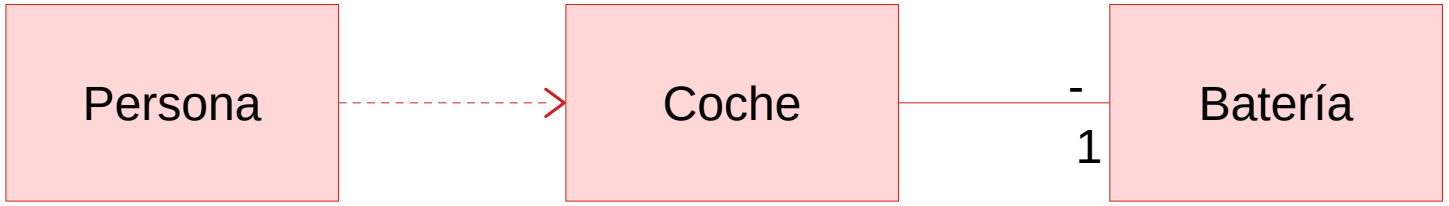
En igualdad de condiciones se prefiere una baja temporalidad, pues esto implica mayor independencia y menor coste de recursos.

Aspectos a considerar definiendo relaciones

Visibilidad

En una relación entre dos clases u objetos siempre es preciso que exista algún nivel de visibilidad para que puedan realizarse interacciones.

En general, se debe seguir el criterio de definir la mínima visibilidad posible que permita obtener la funcionalidad requerida. De esta forma se reduce la complejidad, mejorando la encapsulación y manteniendo la funcionalidad.

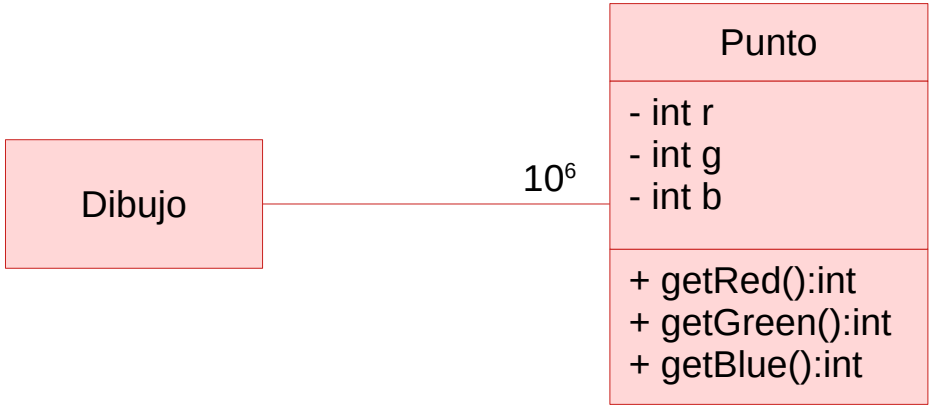


Aspectos a considerar definiendo relaciones

Otros criterios

Existen otros aspectos, como la economía y la eficiencia, que pueden condicionar el diseño de una relación. El utilizar estos criterios puede chocar frontalmente con un buen diseño orientado a objetos.

Por ejemplo, imagine un caso en el que tras detectar un problema se proponen dos soluciones. La primera consiste en rediseñar una clase. La segunda en permitir el acceso a cierta variable desde fuera de la clase.



Relaciones entre objetos

En esta sección se analizan las relaciones de asociación y dependencia entre objetos como mecanismo de creación de abstracciones.

Relaciones de asociación

Las relaciones de asociación surgen cuando se unen varios objetos para formar uno nuevo.

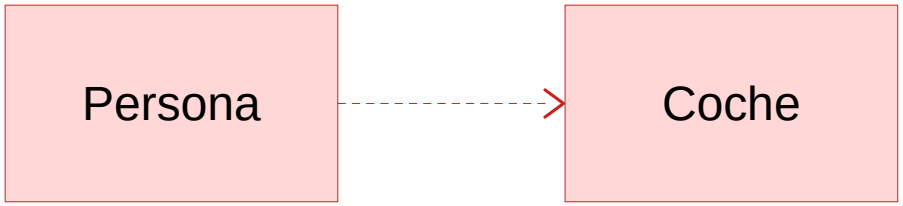
La interacción entre los objetos que se unen hace que emerja un comportamiento mayor que la simple suma de los comportamientos de sus elementos.

Relaciones de dependencia

La relación de dependencia se da cuando un objeto depende de otro pero no está asociado a él.

Hay relaciones de dependencia cuando:

- Un método de una clase recibe como parámetro un objeto de otra clase.
- Una clase crea objetos de otra dentro de un método.



Representación estática de la asociación

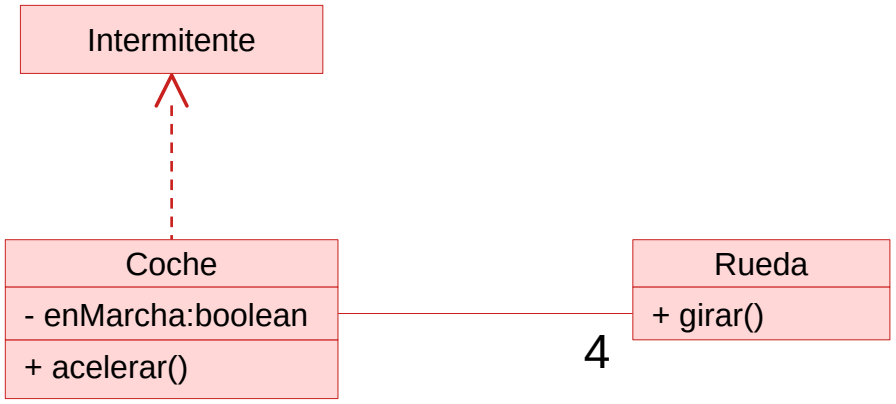
La dimensión estática debe mostrar cómo se relacionan las clases y las interfaces de los objetos que se asocian.

Los diagramas estáticos de clases de UML muestran la asociación entre dos objetos de dos formas:

- Mediante una línea continua que une las cajas que representan a cada clase de objetos.
- Mediante una propiedad.

Representación estática de la dependencia

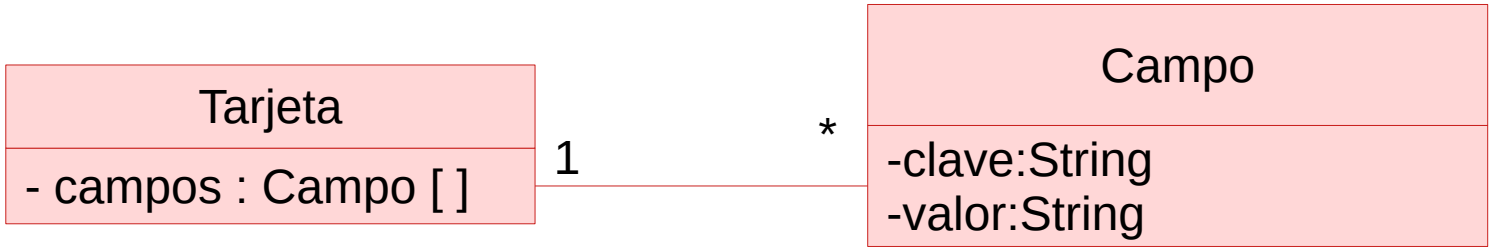
Los diagramas estáticos de clases de UML muestran la dependencia entre dos objetos mediante una línea punteada con que une las cajas que representan a cada clase de objetos y una flecha simple que apunta al objeto dependido.



Ejemplo de asociación

Este asociación entre clases permitiría crear tarjetas que contuviesen la información de un DNI, o una tarjeta de crédito, o una tarjeta de una biblioteca…

Obsérvese la flexibilidad para personalizar campos.



Dimensión dinámica

La dimensión dinámica se plasma en UML mediante los diagramas de interacción entre objetos. Estos diagramas describen, utilizando flechas y cajas, cómo interactúan los objetos de una relación de dependencia.

UML define dos tipos de diagramas de interacción:

- Diagramas de secuencia
- Diagramas de colaboración

Diagramas de secuencia

Los principales elementos de los diagramas de secuencia son:

- Los objetos
- Las líneas de vida
- Las cajas de ejecución
- Los mensajes
- Los operadores de control

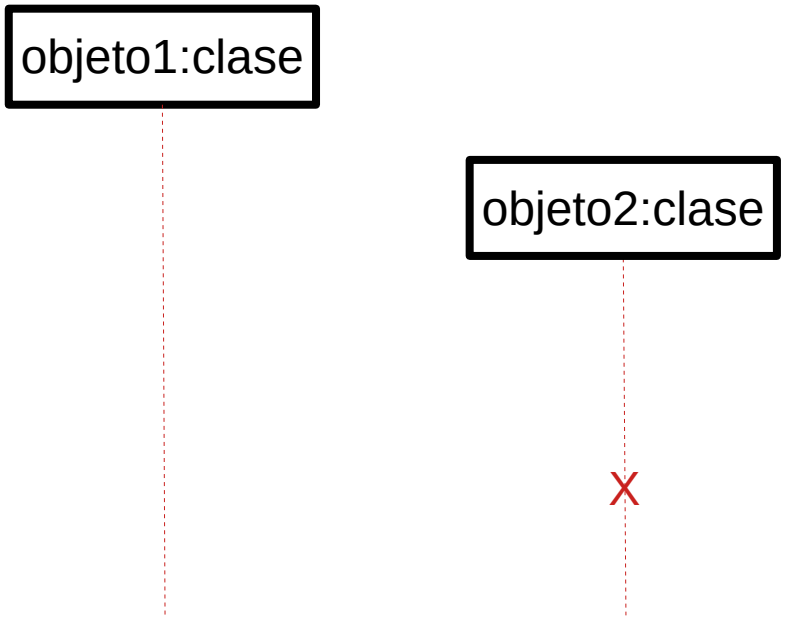
objeto1:clase

objeto2:clase

Diagramas de secuencia

Los principales elementos de los diagramas de secuencia son:

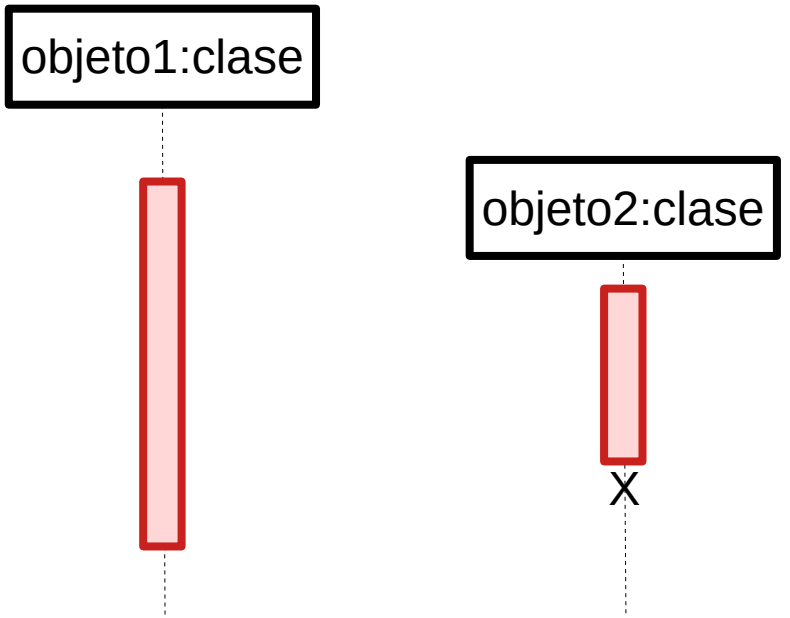
- Los objetos
- Las líneas de vida
- Las cajas de ejecución
- Los mensajes
- Los operadores de control



Diagramas de secuencia

Los principales elementos de los diagramas de secuencia son:

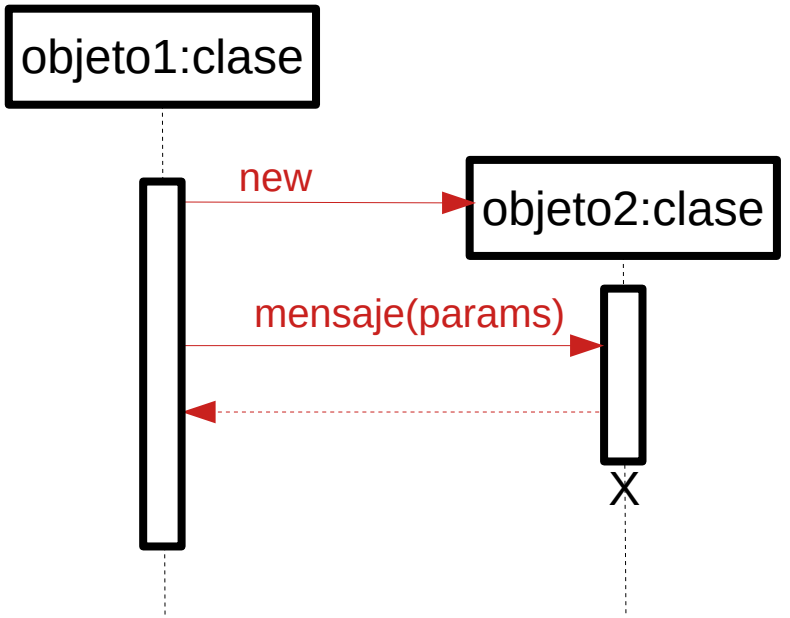
- Los objetos
- Las líneas de vida
- Las cajas de ejecución
- Los mensajes
- Los operadores de control



Diagramas de secuencia

Los principales elementos de los diagramas de secuencia son:

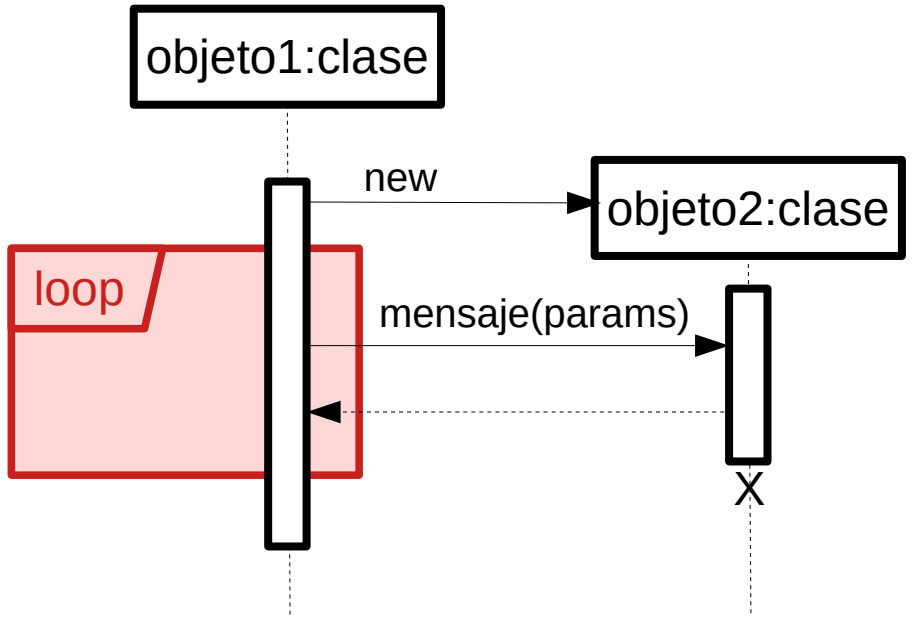
- Los objetos
- Las líneas de vida
- Las cajas de ejecución
- Los mensajes
- Los operadores de control



Diagramas de secuencia

Los principales elementos de los diagramas de secuencia son:

- Los objetos
- Las líneas de vida
- Las cajas de ejecución
- Los mensajes
- Los operadores de control

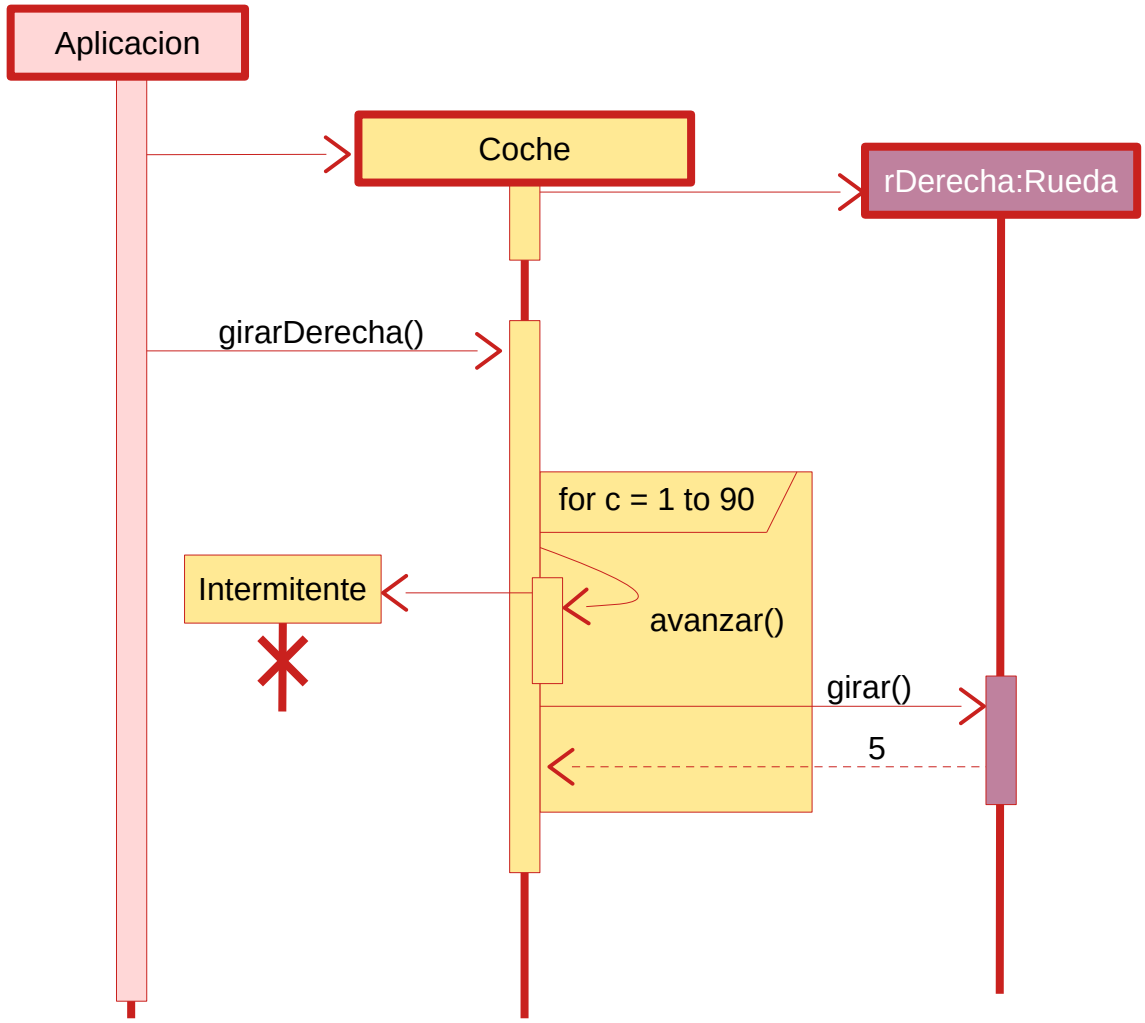


Ejemplo de diagrama de secuencia

```
class Aplicacion {
  public static void main(String [] a){
    Coche c = new Coche();
    c.girarDerecha();
  }
  ...
}
```

```
class Coche {
  private Rueda rDerecha;
  private Rueda rIzquierda;
  private int x;

  private void avanzar() {
    x++;
    Intermitente i = new Intermitente();
  }

  public void girarDerecha() {
    for (int c = 1; c <= 90; c++) {
      avanzar();
      r.girar();
    }
  }
}
```

```
class Rueda {
  private int pos;
  public int girar() {
    pos++;
    return pos;
  }
}
```

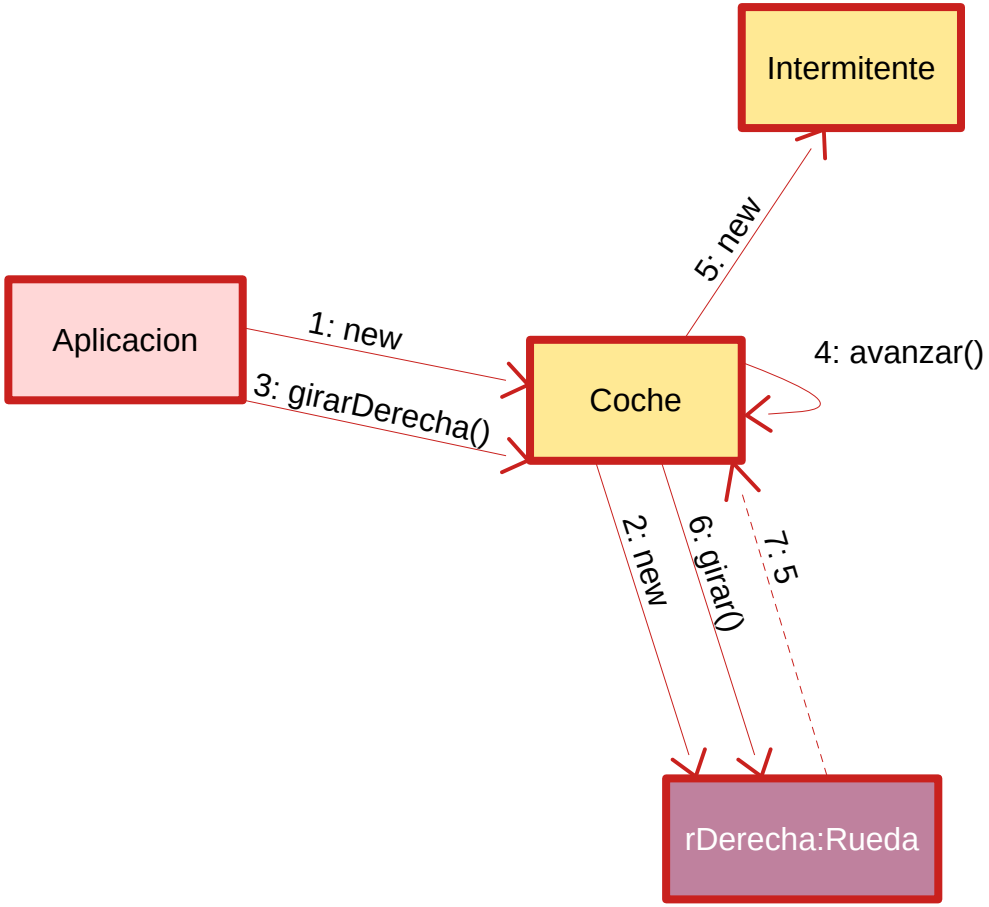


Ejemplo de diagramas de colaboración

```
class Aplicacion {
  public static void main(String [] a){
    Coche c = new Coche();
    c.girarDerecha();
  }
  ...
}
```

```
class Coche {
  private Rueda rDerecha;
  private Rueda rIzquierda;
  private int x;

  private void avanzar() {
    x++;
    Intermitente i = new Intermitente();
  }

  public void girarDerecha() {
    for (int c = 1; y <= 90; y++) {
      avanzar();
      r.girar();
    }
  }
}
```

```
class Rueda {
  private int pos;
  public int girar() {
    pos++;
    return pos;
  }
}
```



Temporalidad de las relaciones entre objetos

La dependencia debe usarse cuando la temporalidad de la relación sea baja y la asociación cuando la temporabilidad sea más larga.

- Si la relación entre dos objetos se limita al ámbito de un método debe usarse dependencia.
- Si la relación entre dos objetos es más larga debe usarse una relación de asociación.

Versatibilidad en las relaciones entre objetos

En la relaciones de dependencia y de asociación debe fomentarse la versatilidad utilizando el polimorfismo para las variables.

- Si los parámetros se declaran de clases bases de otras muchas, los métodos resultantes son más reutilizables.
- Si las propiedades se declaran de clases base de otras muchas, se puede cambiar más fácilmente la propiedad por otra.

Visibilidad en las relaciones entre objetos

Desde el punto de vista de la visibilidad, las relaciones de dependencia favorecen la encapsulación de los participantes en la relación, ya que los objetos dos no quedan expuestos.

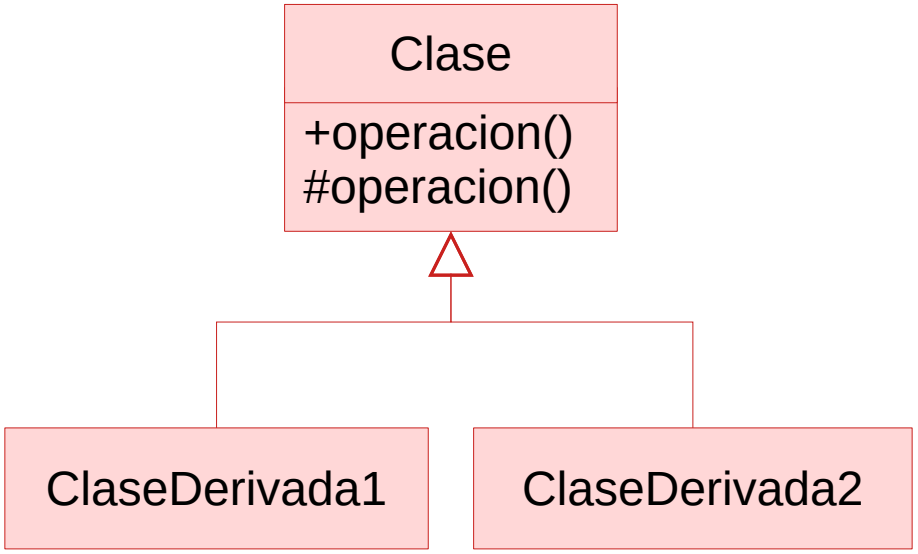
En las relaciones de asociación, el diseñador debe decidir que nivel de visibilidad otorga respecto a los valores asociados.

Relaciones de herencia entre clases

En esta sección se analizan las relaciones de herencia entre clases como mecanismo de creación de abstracciones.

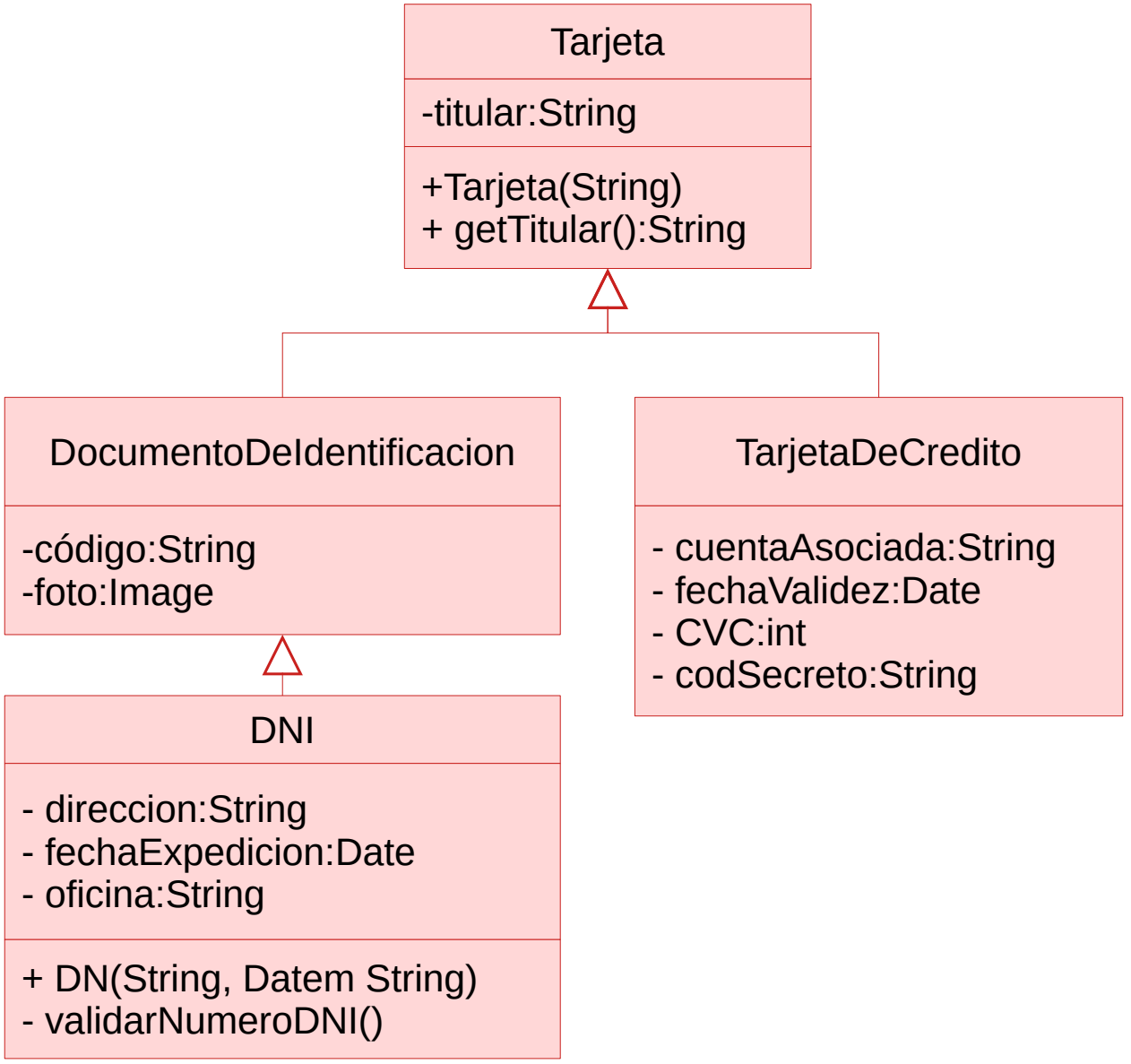
Motivación del uso de herencia de clases

La idea subyacente al uso de herencia es crear jerarquías donde las partes similares se implementan en las clases bases para que las clases derivadas las aprovechen.



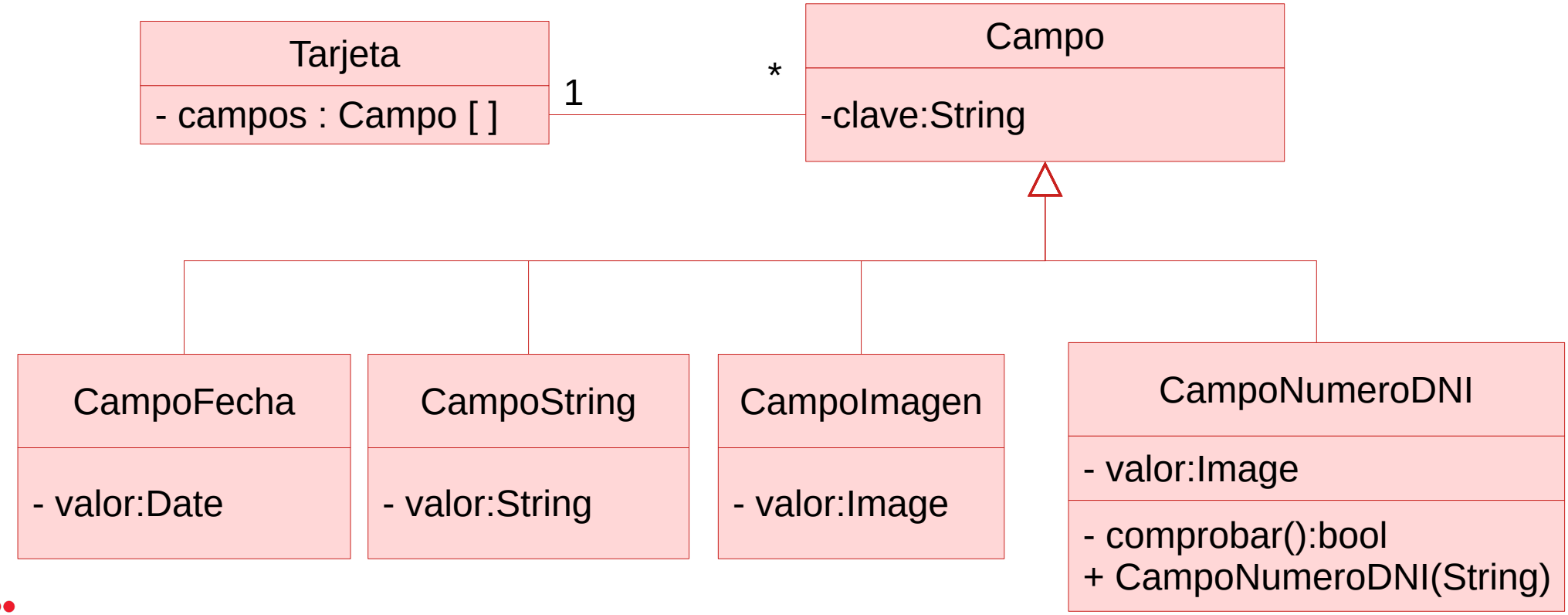
Ejemplo del uso de herencia de clases

Esta jerarquía permite incorporar elementos específicos de los diferentes tipos de tarjetas, aunque se pierde flexibilidad frente a personalización de los tipos de tarjeta sin codificar nuevas clases.



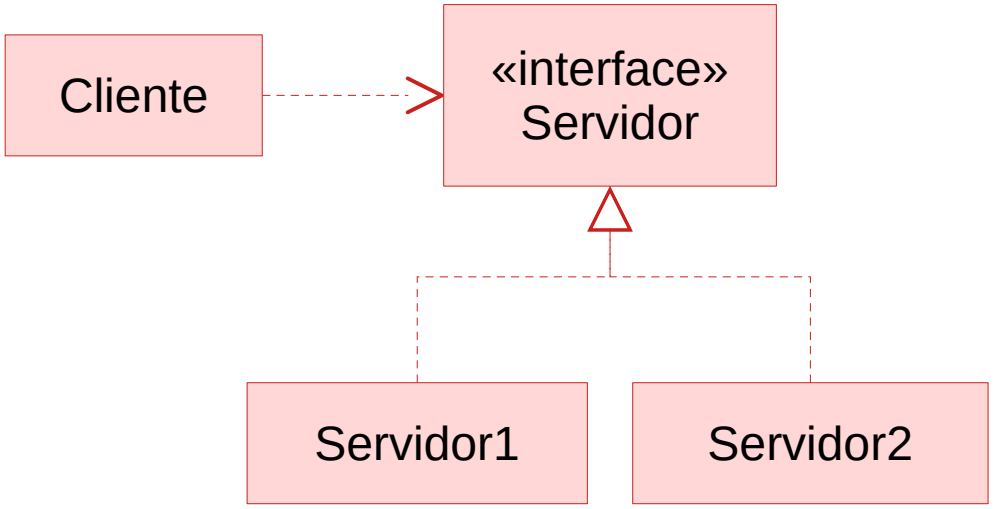
Ejemplo del uso de herencia de clases

Este otro enfoque mixto permite añadir algunas operaciones específicas y mantiene la flexibilidad en cuanto a la personalización de campos.



Motivación del uso de herencia de interfaz

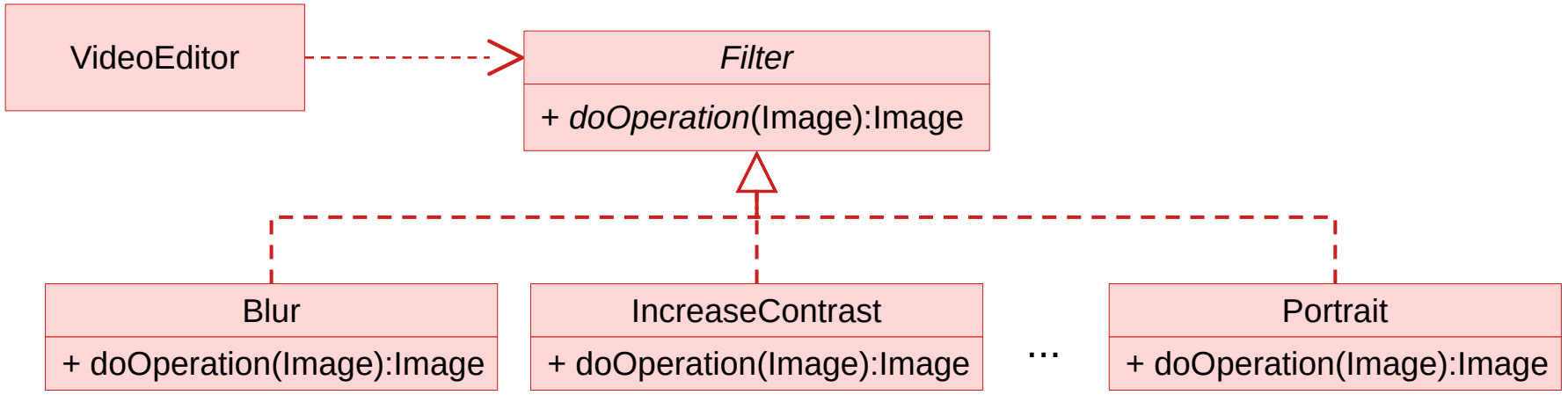
La idea subyacente al uso de la herencia de interfaz es crear jerarquías para facilitar comportamientos polimórficos.



Ejemplo de herencia para organizar algoritmos

Este ejemplo presenta una interfaz Filter de la que heredan otras que realizarán diferentes tipos de filtrados sobre imágenes.

Que los diferentes filtros hereden de Filter permite almacenar los diferentes tipos de filtros en estructuras de datos o ejecutarlos sin hacer distinción entre unos y otros.



Polimorfismo

La idea tras el uso del polimorfismo es triple:

- facilitar la comprensión de operaciones que conceptualmente hacen lo mismo dontándolas del mismo nombre y los mismos parámetros.
- facilitar el intercambio en un programa de objetos que comparten la misma interfaz.
- facilitar el uso genérico de objetos que comparten la misma interfaz.

Clases abstractas o Interfaces

Suelen utilizarse interfaces cuando sólo se desea asegurar que un conjunto de objetos cumple cierta característica que lo hace tratable por otro conjunto de objetos: clonable, comparable, modifcable...

Si además se desea añadir comportamiento a algunos métodos se usan interfaces con métodos default.

Suelen utilizarse clases abstractas cuando además de una interfaz genérica se desean añadir propiedades a todos los elementos de una jerarquía.

Cúando y cómo crear relaciones de herencia

Las relaciones de herencia tienen una alta temporalidad en comparación con las relaciones de asociación y de uso, pues se basan solo en la escritura de código.

La herencia aporta versatilidad. El polimorfismo dinámico generado por la herencia promueve la versatilidad, ya que permite referencias comunes a objetos que comparten la misma interfaz.

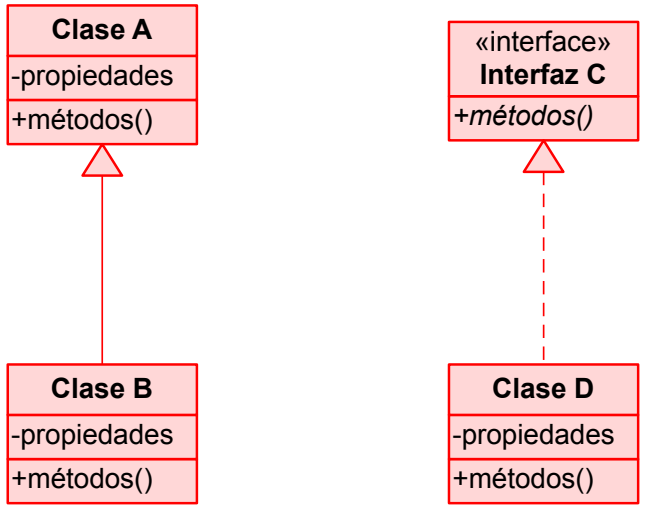
La herencia normalmente viola los principios de encapsulación. Pues suele ser preciso conocer a fondo la clase de la que se hereda. Debe limitarse la visibilidad de las propiedades: seleccionar los métodos protegidos, fomentar el modificador final...

Relaciones de herencia entre clases

Dimensión estática

En UML la herencia se representa, en el diagrama estático de clases, mediante una flecha que parte de la clase que hereda y termina en la clase de la que se hereda. Esta flecha tiene como punta un triángulo hueco para diferenciarla de la flecha que representa uso.

La representación de la herencia de interfaz es igual, salvo que se utiliza una línea discontinua.



Dimensión dinámica

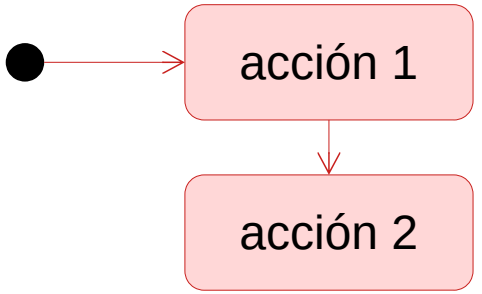
Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades

Dimensión dinámica

Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

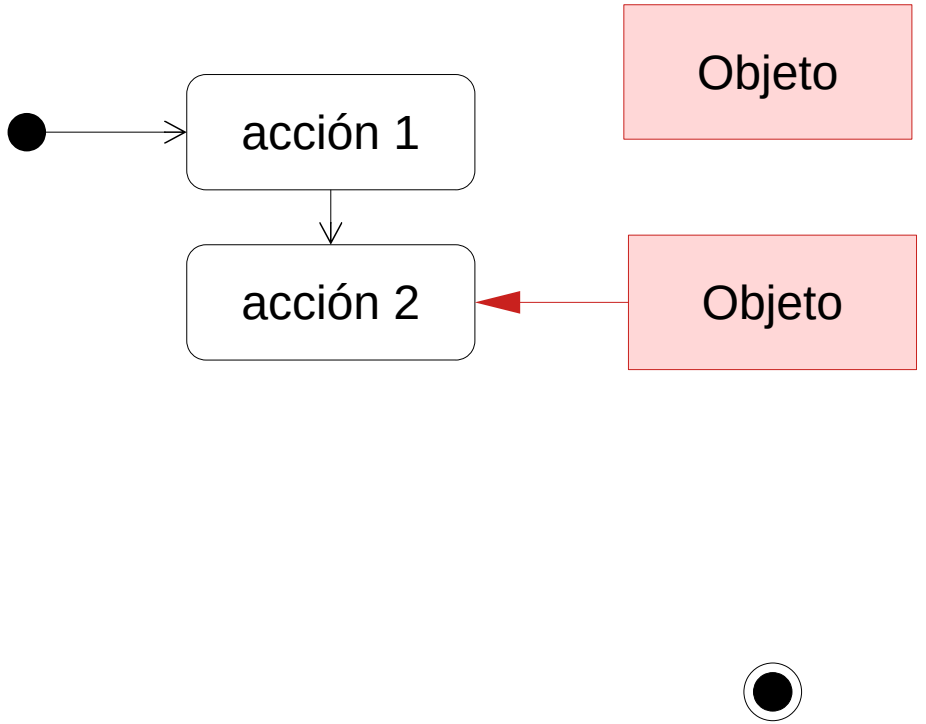
- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades



Dimensión dinámica

Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

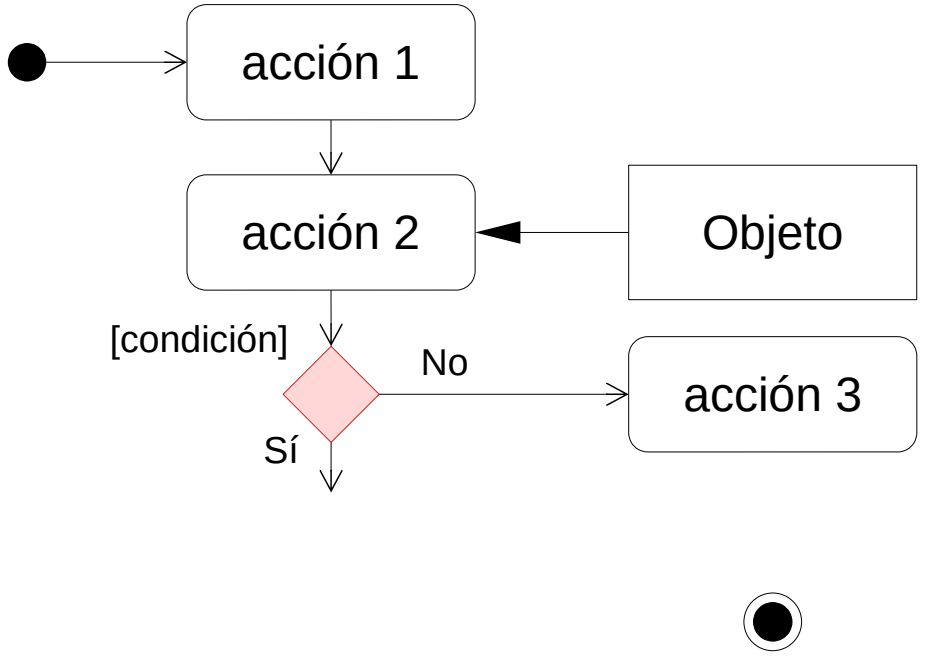
- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades



Dimensión dinámica

Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

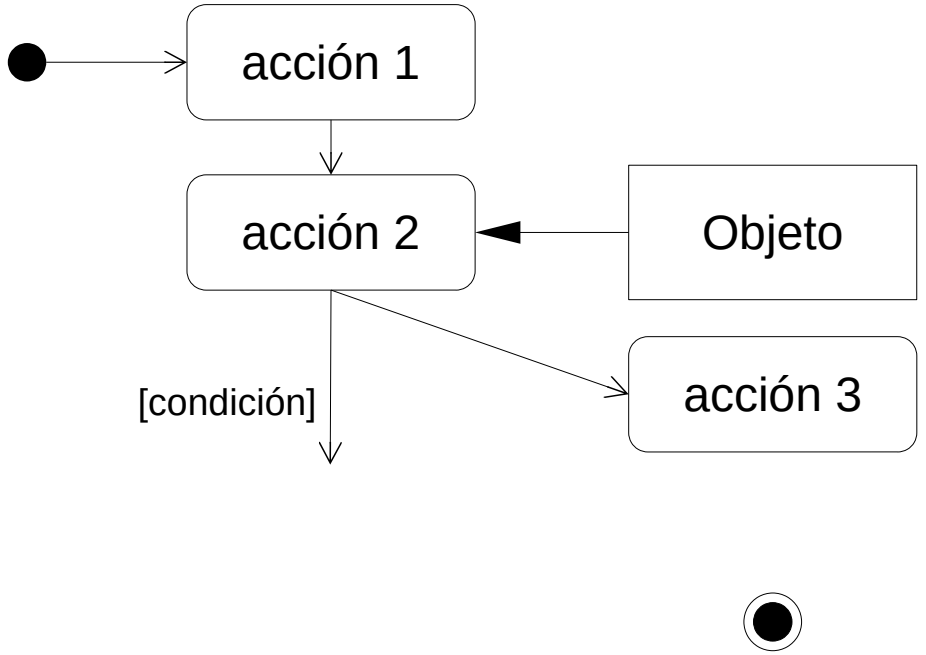
- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades



Dimensión dinámica

Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

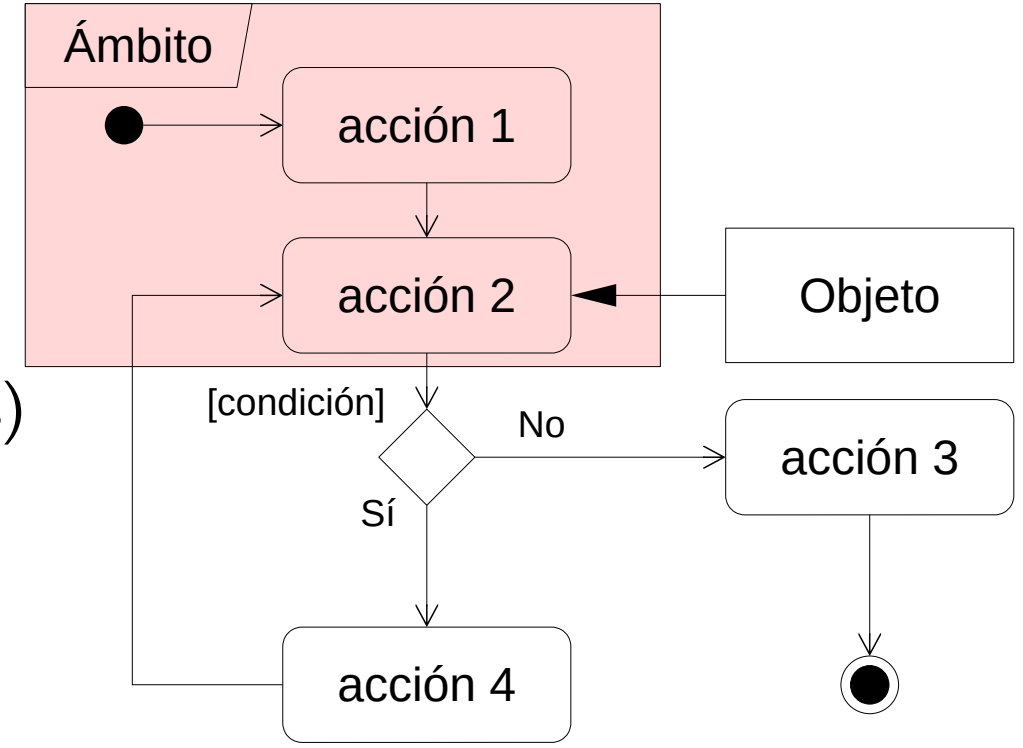
- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades



Dimensión dinámica

Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

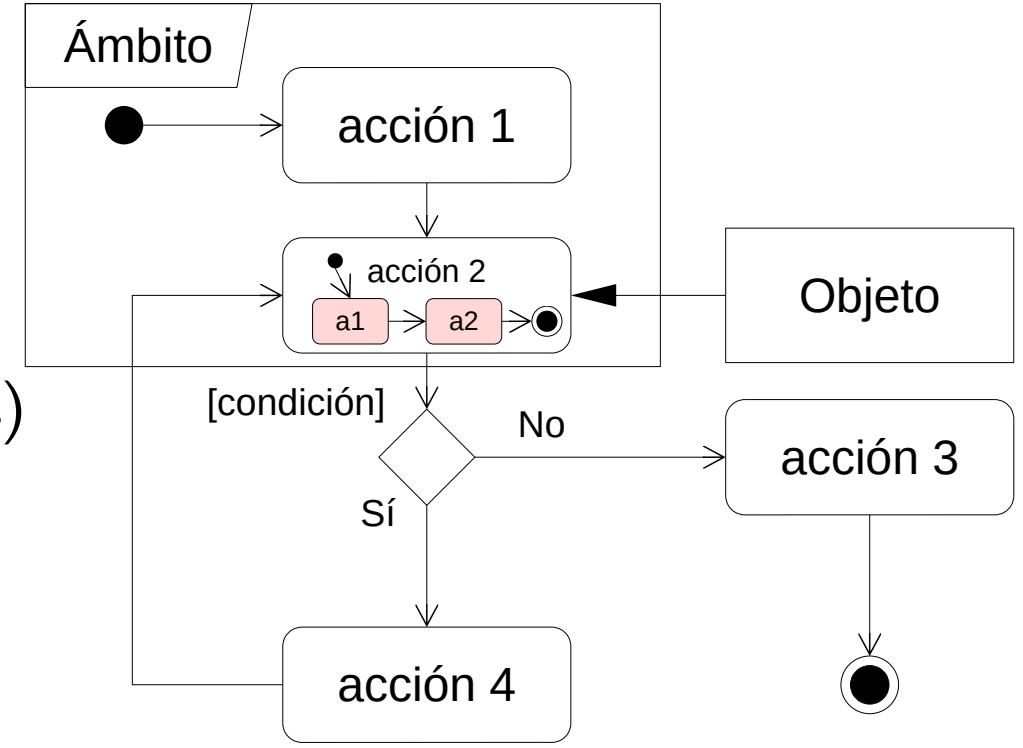
- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades



Dimensión dinámica

Para expresar en UML la dimensión dinámica de una relación entre clases es conveniente usar los diagramas de actividad. Sus elementos básicos son:

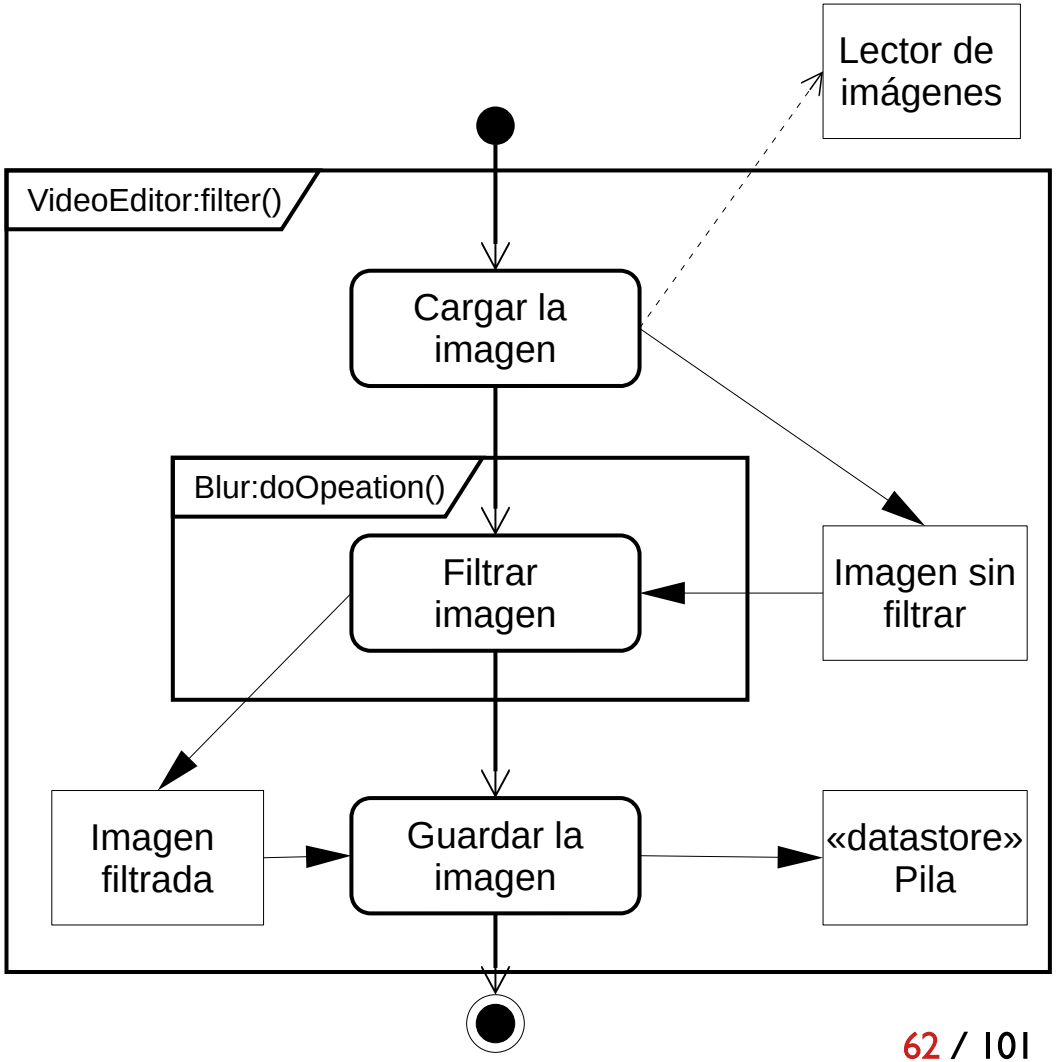
- El inicio y el fin
- Las acciones y el flujo
- Los objetos
- Las bifurcaciones (2 opciones)
- Los ámbitos
- Actividades en actividades



Los objetos en los diagramas de actividad

La aparición de objetos en los diagramas de actividad dan lugar a varios casos:

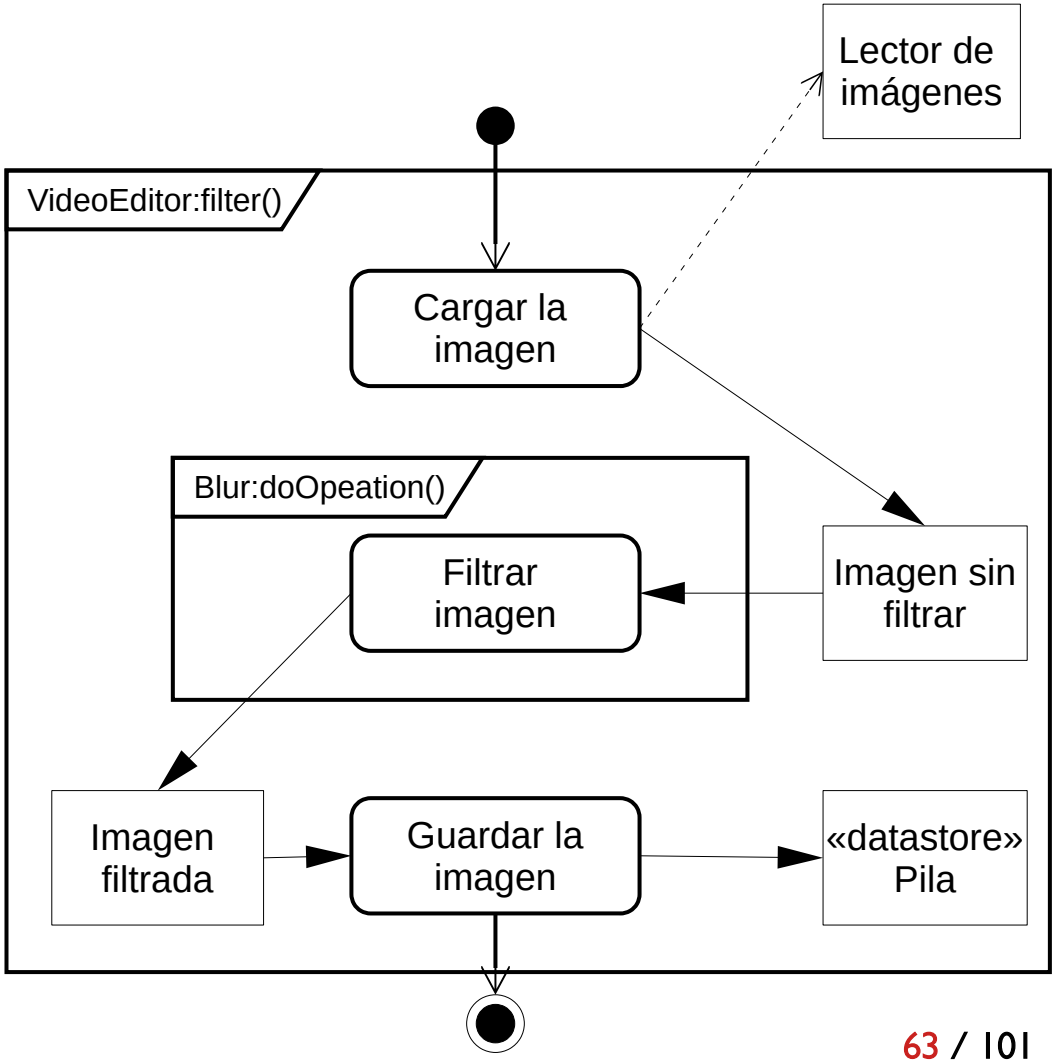
- La acción produce un objeto.
- La acción consume un objeto.
- La acción almacena un objeto.
- La acción depende de un objeto.



Los objetos en los diagramas de actividad

La aparición de objetos en los diagramas de actividad dan lugar a varios casos:

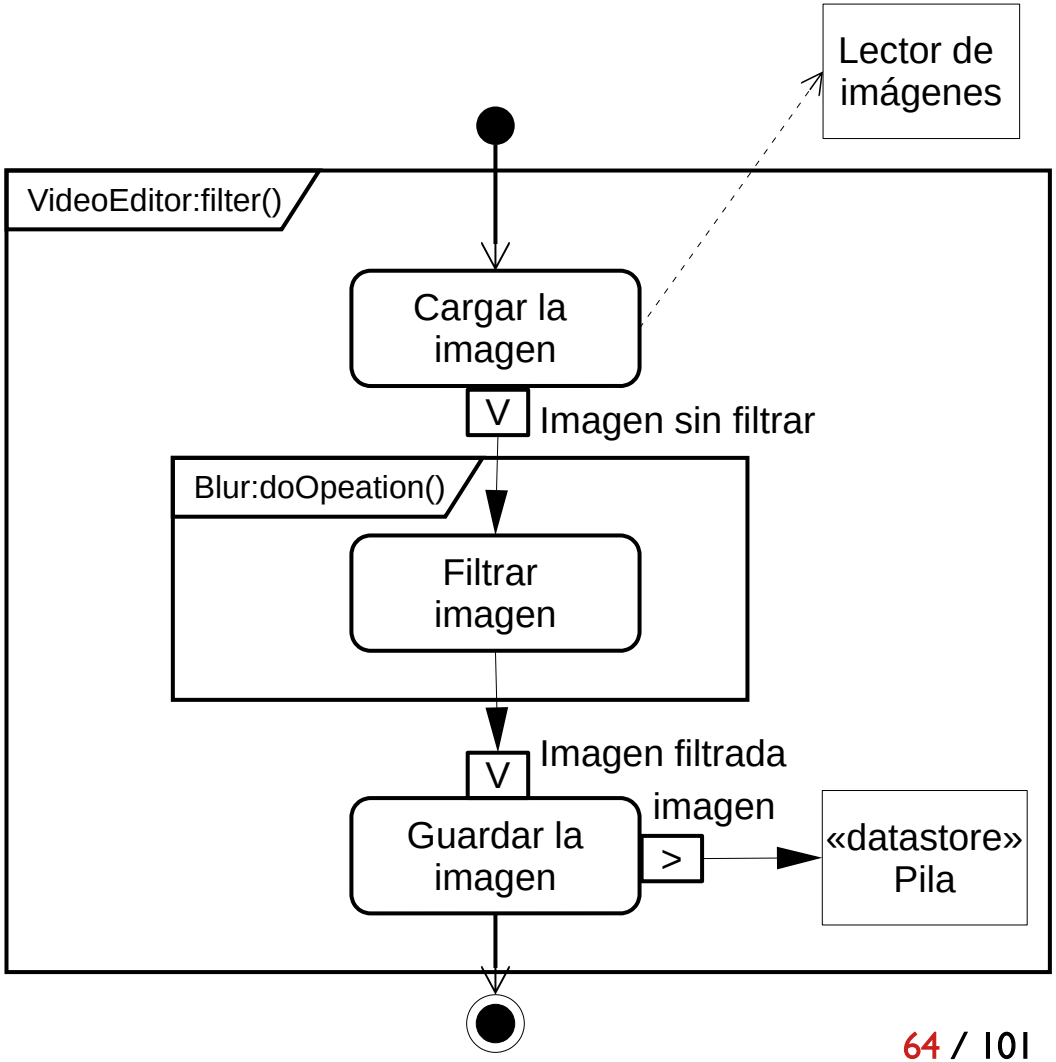
- La acción produce un objeto.
- La acción consume un objeto.
- La acción almacena un objeto.
- La acción depende de un objeto.



Los objetos en los diagramas de actividad

La aparición de objetos en los diagramas de actividad dan lugar a varios casos:

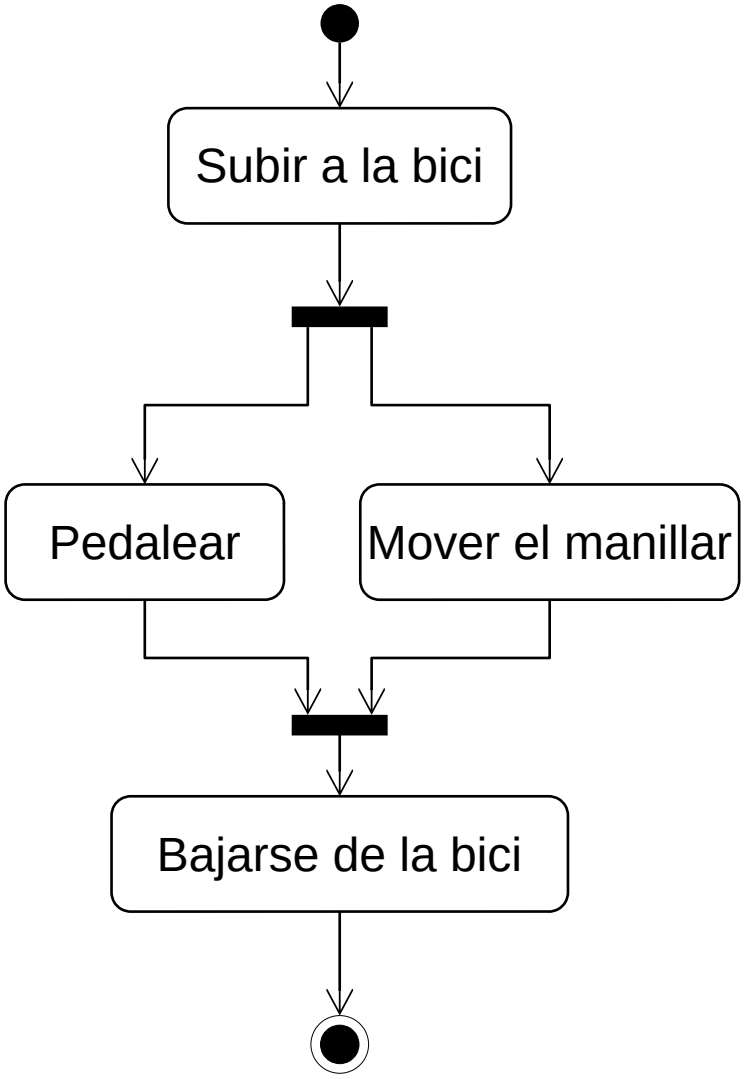
- La acción produce un objeto.
- La acción consume un objeto.
- La acción almacena un objeto.
- La acción depende de un objeto.



La concurrencia en los diagramas de actividad

El segmento de trazo grueso indica:

- Bien la apertura de nuevos hilos de ejecución, que implican la realización de varias tareas simultáneamente.
- Bien la unión de varios hilos de ejecución, mediante sistemas de sincronización (como wait-notify en Java).



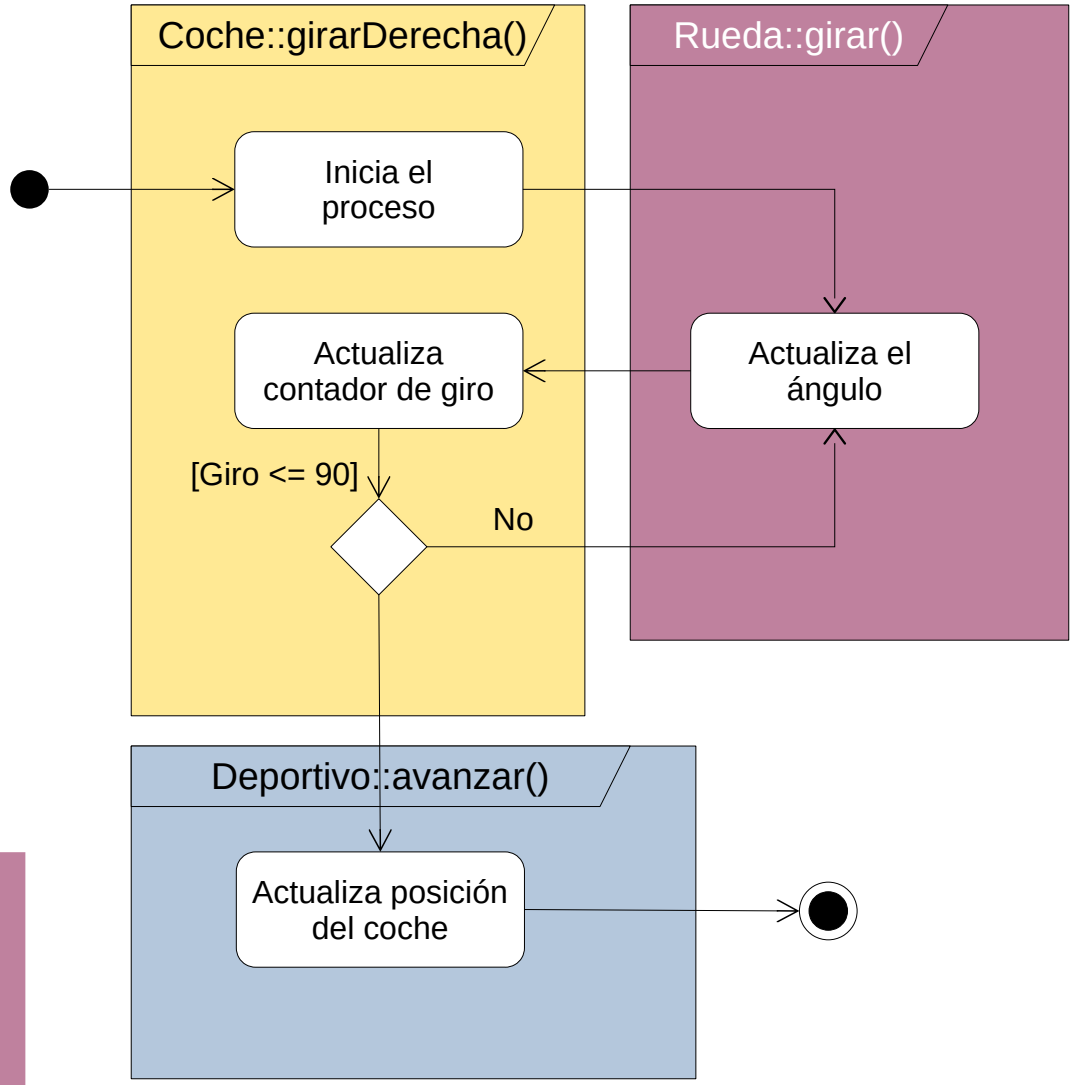
Relaciones de herencia entre clases

Ejemplo de diagrama de actividad

```
class Coche {
  private Rueda rDerecha;
  private Rueda rIzquierda;
  protected int x;

  public void avanzar() {
    x++;
    Intermitente i = new Intermitente();
    i.parpadear(5);
  }

  public void girarDerecha() {
    for (int c = 1; c <= 90; c++) {
      rDerecha.girar();
    }
    avanzar();
  }
}
```

```
class Deportivo extends Coche {

  @overrides
  public void avanzar() {
    x += 10;
  }
}
```

```
class Rueda {
  private int angulo;
  public int girar() {
    angulo = (angulo + 1) % 360;
    return angulo;
  }
}
```



Diagramas de secuencia vs de actividad

Los diagramas de secuencia deben usarse para describir o descubrir la comunicación que se produce entre diferentes clases al realizarse un proceso.

Por otro lado, los diagramas de actividad se usan cuando:

- el proceso discurre principalmente dentro de una sola clase;
- o cuando se quiere describir un proceso dando más protagonismo al algoritmo que a las clases que lo hacen posible.

Principales principios de programación

En esta sección se presentan algunos de los principales principios a tener en cuenta al programar en general.

Principales principios de Programación

Algunos principios generales de programación

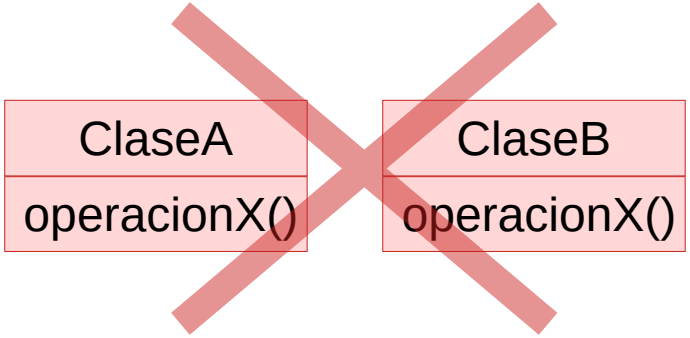
- No te **r**epitas
- Usa **E**structuras de datos adecuadas
- **C**onstantes y datos variables
- **A**lgoritmos, Procesos y Cálculos
- **Le**e documentación y errores
- No programes más de lo **n**ecesario
- Regla del **t**reinta
- **A**utomatiza tus Test
- **D**ocumenta el código
- No **O**ptimices prematuramente
- Crea código **s**imple

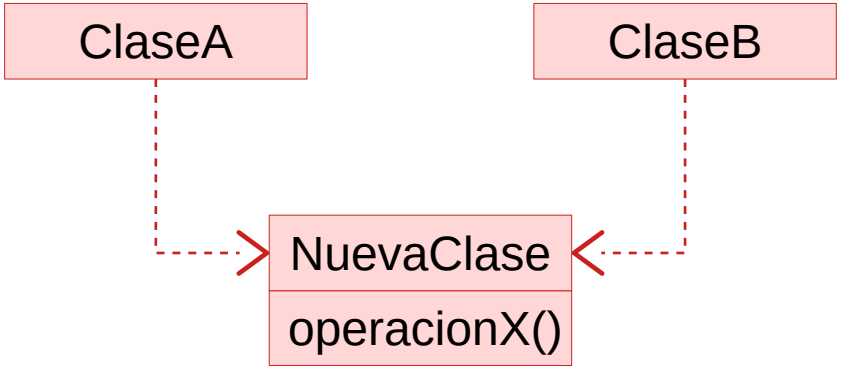
No te repitas

Don't repeat your self (DRY)

Siempre que existe una duplicación se debe a que no se ha abstraído lo suficiente. Probablemente la duplicación se pueda eliminar con la introducción de una nueva función, un nuevo método o una nueva clase.

Andy Hunt





Estructura de datos

Los programas suelen necesitar datos más o menos estáticos durante su ejecución. Dichos datos deben almacenarse en estructuras de datos eficientes en cuanto a velocidad y capacidad.

La elección de la estructura de datos no solo depende del tipo de dato, también depende del tratamiento que se vaya a hacer con ellos. Por ejemplo:

- Un árbol ordenado para poder buscar un DNI los empleados de una empresa.
- Una lista enlazada para poder insertar una tarea a mitad de lista de tareas.

Constantes y datos variables

Los datos que no cambian de una ejecución a otra deben definirse como constantes.

Los datos que pueden cambiar de una ejecución a otra no deben formar parte de la estructura del programa.

Ejemplos de constantes:

- El número Pi, la existencia de una estructura de datos, las opciones de un menú (si no es dinámico).

Ejemplos de datos variables:

- El número de salas de un multicine, el número de empleados de una empresa, la descripción de un artículo a la venta, el porcentaje de descuento de unas rebajas.

Algoritmos/Procesos/Cálculos

Identificar los algoritmos, procesos o cálculos y situarlos en métodos de clases nuevas o existentes.

Ejemplos:

- Calcular el tamaño de una ED tipo árbol.
- Copiar una parte de una lista.
- Calcular los impuestos de un empleado.
- Dibujar la carretera en el juego Gran Turismo.

Lee documentación y errores

Lee la documentación, no la adivines.

Lee los mensajes de error. No hagas que el editor los arregle sin entender el cambio.

Lee el código de otros y no uses lo que no entiendas.

No programes más de lo necesario

You Aren't Gonna Need It (YAGNI).

Este principio aporta las siguientes ventajas:

- No resta tiempo a la creación de la funcionalidad básica.
- Hasta que no se defina para qué se necesita es difícil saber qué se debe hacer.
- Dar una funcionalidad no requerida podría dificultar su correcta implementación cuando se sepa lo que se necesita.
- Evita código inflado que no proporciona más funcionalidad.
- Evita efectos de tipo bola de nieve, respecto a agregar más funcionalidades no requeridas.

Regla del 30

No hagas funciones/métodos/procedimientos de más de 30 líneas. Ni hagas clases de más de 30 métodos.

Refactoring in Large Software Projects,
por Martin Lippert y Stephen Roock

Otra regla similar dice que el máximo de líneas de código para un método o una función debería estar en el numero de líneas que quepan en la pantalla del programador.

Code Complete
Steve McConnell

Automatiza tus test

Siempre crea pruebas automáticas de tu código.

Este principio aporta las siguientes ventajas:

- Facilita la repetición de las pruebas.
- Aumenta la confianza de los programadores en el código.
- Evita regresiones al corromper funcionalidades que previamente funcionaban correctamente.

Documenta el código

Documenta métodos y clases usando el formalismo del lenguaje.

Documenta algoritmos usando diagramas dinámicos.

Documenta las clases usando diagramas estáticos.

La documentación es como el dinero. Nunca parece suficiente, pero poco es mejor que nada.

Principales principios de Programación

No optimices prematuramente

La optimización del código suele reducir su legibilidad, haciéndolo más difícil de mantener y depurar.

Las optimizaciones no añaden nueva funcionalidad, y pueden añadir nuevos errores.

Las optimizaciones necesarias deben hacerse en fases finales de desarrollo, con pruebas automáticas y manteniendo el código anterior.

"Debes olvidar el 97% de las pequeñas optimizaciones. La optimización prematura es la raíz de todos los males. Pero, no olvides la oportunidad de ese 3% crítico"

Donald Knuth

Crea código evidente y simple

Kept it simple, stupid (KISS)

Kelly Johnson
Ingeniero jefe en Lockheed Skunk Works

El principio KISS predice que los sistemas funcionan mejor si se mantienen simples en vez de complejos. Por ello, mantener la simplicidad se marca como un objetivo clave del diseño.

Principales principios de POO

En esta sección se presentan algunos de los principales principios a de la programación orientada a objetos.

Principales principios de la POO

SOLID y GRAPS

SOLID es un acrónimo de Michael Feathers basádo en 5 principios de la POO que Robert C. Martin (tío Bob) recopiló en el 2000 [1]. Ocho años más tarde, R. C. Martin publicó el clásico Clean Code que recopila muchos más principios [2].

Los principios GRASP (General Responsibility Assignment Software Principles) son un conjunto de 9 principios sobre diseño de objetos y asignación de responsabilidades publicados por primera vez por Craig Larman en 1997 (Applying UML and Patterns).

Principales principios de la POO

Algunos principios de POO

- **S**ingle responsability (SOLID)
- **O**pen/Close (SOLID)
- Principio de **L**iskov (SOLID)
- Segregación de **I**nterfaces (SOLID)
- Inversión de **D**ependencia (SOLID)

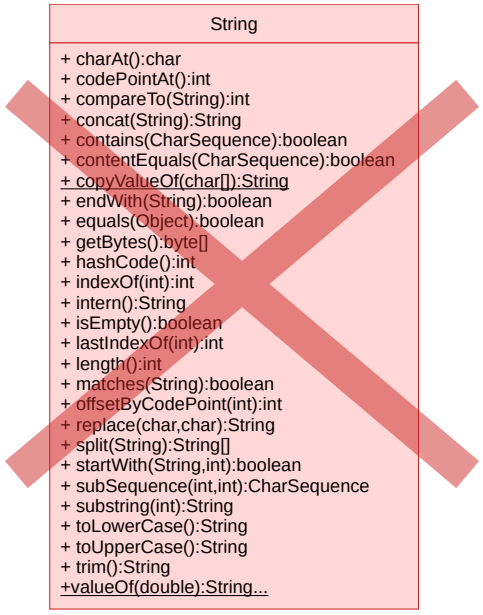
- Detecta las **a**bstracciones de tu problema
- **P**olimorfismo mejor a condición basada en tipo (GRAPS)
- Prefiere clases **i**nmutables
- Prefiere la **c**omposición a la herencia
- Principio del **e**xperto (GRAPS)
- **S**eparación de Comandos y consultas (CQS)

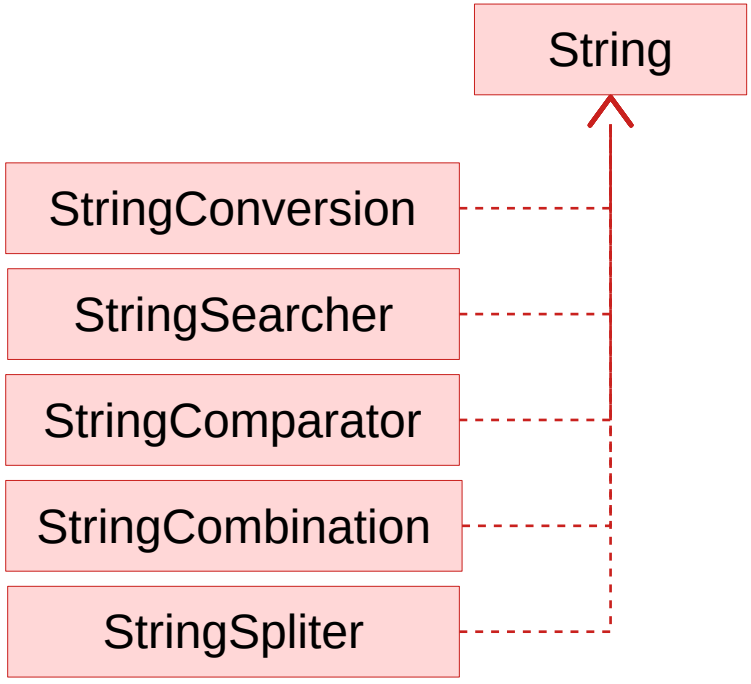
Principio de responsabilidad única (S)

Una clase debe tener un único motivo para cambiar.

Robert C. Martin

Es decir, una clase debe tener una única responsabilidad y dicha responsabilidad debe estar completamente encapsulada en dicha clase.





Principio de responsabilidad única (S)

El principio de responsabilidad única también esta relacionado con:

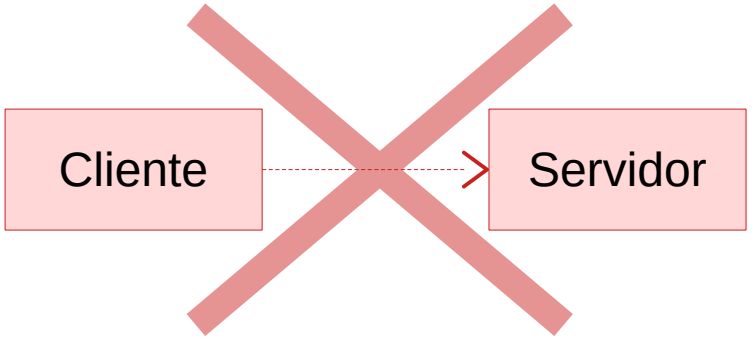
- Alta cohesión.- Si una clase solo tiene una responsabilidad, entones esa clase tiene alta cohesión.
- Bajo acoplamiento.- Siempre se produce un poco de acoplamiento en los POO en los que las tareas son realizadas por colaboraciones entre objetos.

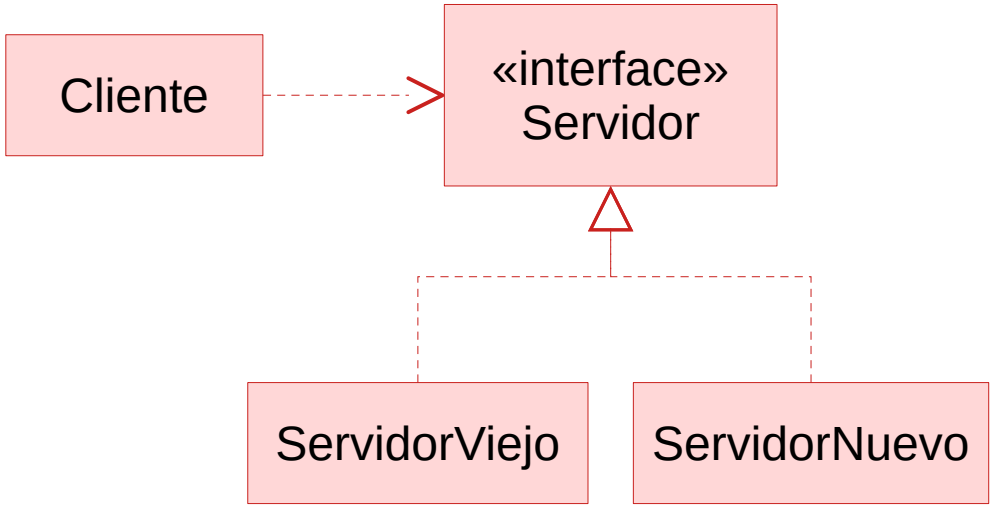
El principio abierto-cerrado (O)

Todas las entidades de software (clases, módulos, funciones...) deben estar abiertas para extensiones y cerradas para modificaciones.

Bertrand Meyer 1998

La idea general tras este principio es que se pueda añadir nueva funcionalidad sin modificar el código existente.





Principales principios de la POO

El principio abierto-cerrado (O)

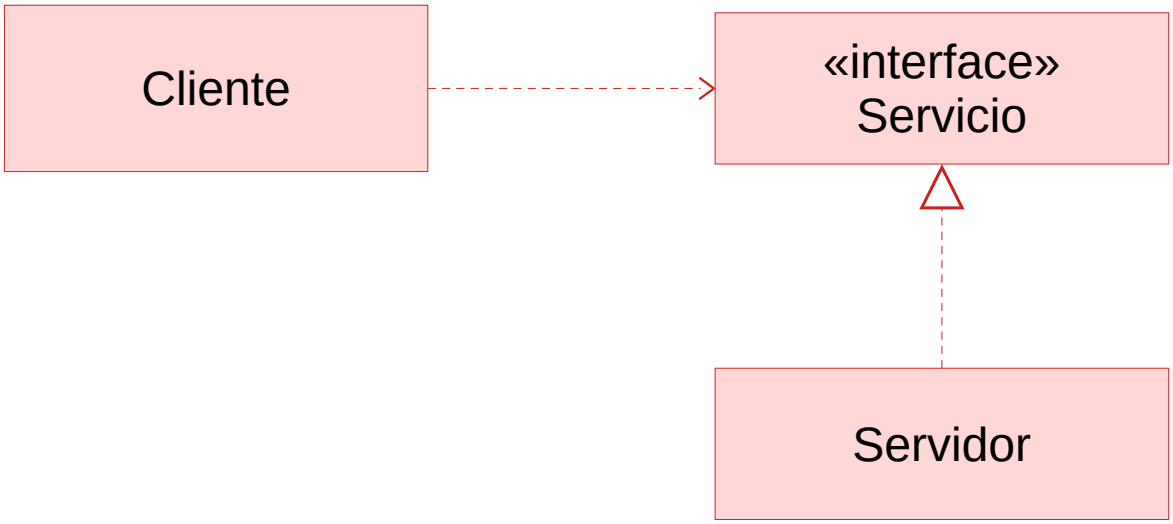
"Identifica los puntos de previsible variación y crea una interfaz estable para ellos."

Este principio es equivalente al principio GRASP de Protección de la Variación.

Principio de Liskov (L)

Toda función que utiliza referencias a un objeto de una clase debe ser capaz de usar objetos de clases derivadas de ella sin saberlo.

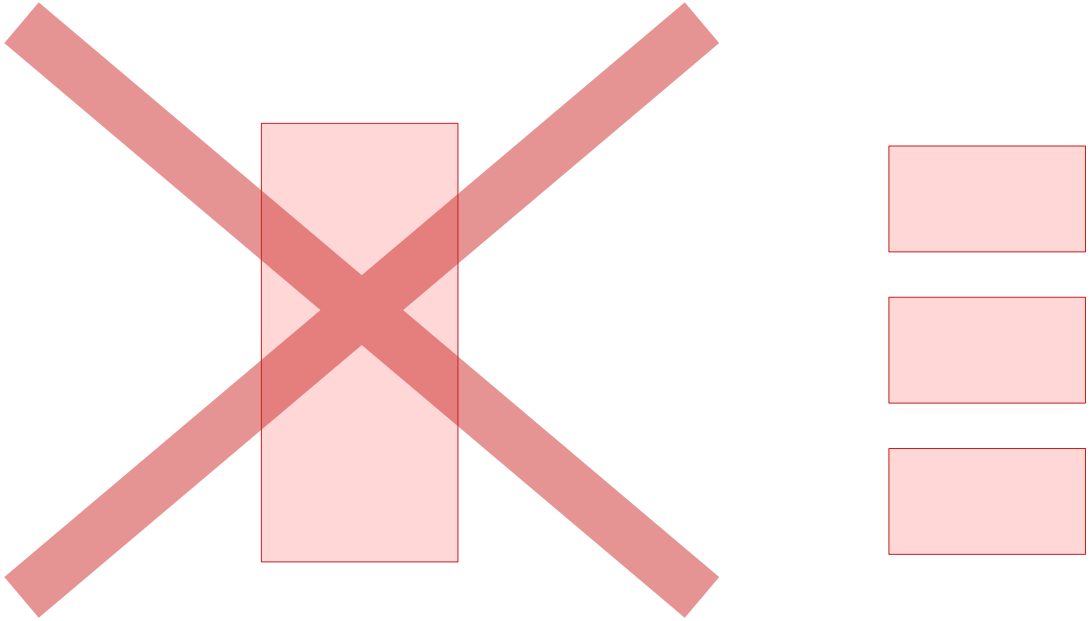
Barbara Liskov



El principio de segregación de interfaces (I)

Es preferible varias interfaces específicas que una única interfaz general.

Robert C. Martin



Seguir este principio minimiza el impacto del cambio de una interfaz, ya que no existen grandes interfaces de las que dependen multitud módulos.

Principio de inversión de dependencia (D)

Este principio se describe con dos enunciados:

a) Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones (interfaces).

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

Robert C. Martin

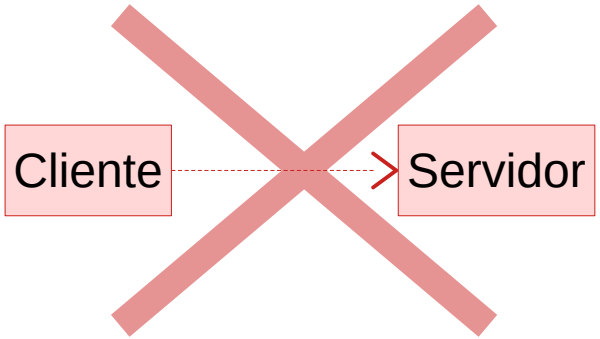
El principio evita que afecte al cliente cambios en el servidor.

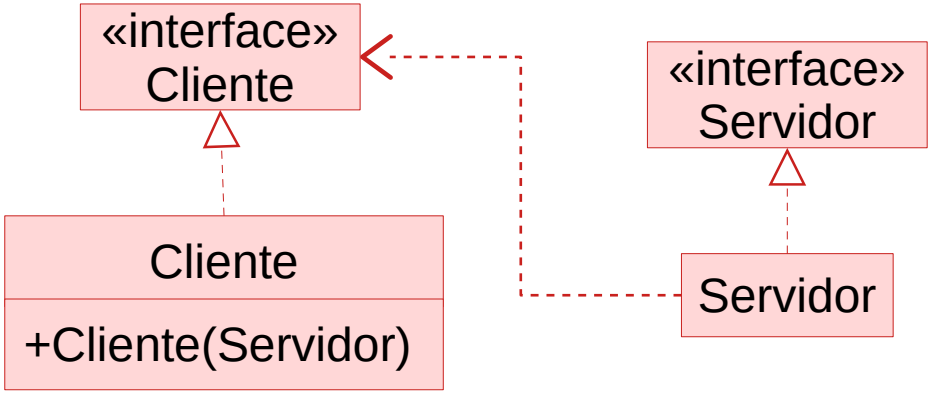
También, facilita probar el Cliente sin el Servidor.

```
class Cliente {
  public void operar(){
    Servidor s = new Servidor();
  }
  ...
}
```

Principales principios de la POO

Principio de inversión de dependencia (D)





Un cambio en el servidor afectaba al cliente. Ahora, la dependencia se invierte. Un cambio en el servidor solo afecta al servidor pues debe cumplir la misma interfaz.

La aplicación de este principio requiere mecanismos de inyección de dependencias. Por ejemplo, usar el constructor, anotaciones o *setters* para proveer a las clases de alto nivel de los módulos que necesite.

Polimorfismo vs condición basada en tipo

Si en una clase tenemos un bloque if/else, esa clase unas veces tendrá una responsabilidad (cuando entre por la rama if) y otras veces tendrá otra diferente (cuando entre por la rama else).

Muchos de los bloques if/else y de los bloques switch/case pueden reemplazarse con polimorfismo.

De hecho, la existencia de uno de estos bloques if/else puede indicar que la clase está rompiendo el principio de responsabilidad única.

Principales principios de la POO

Maximizar el uso de clases inmutables

Las clases inmutables son aquellas cuyos objetos no cambian de estado una vez creados. Por ejemplo, String es una clase inmutable.

Con objeto de forzar este comportamiento todas las propiedades de la clase deben ser finales.

La inmutabilidad simplifica mucho la implementación.

Por ejemplo, facilita el uso de un objeto desde diferentes threads sin mecanismos de exclusión.

Desgraciadamente, es imposible que ciertas clases, como las que mantienen el estado de un programa, sean inmutables.

Principales principios de la POO

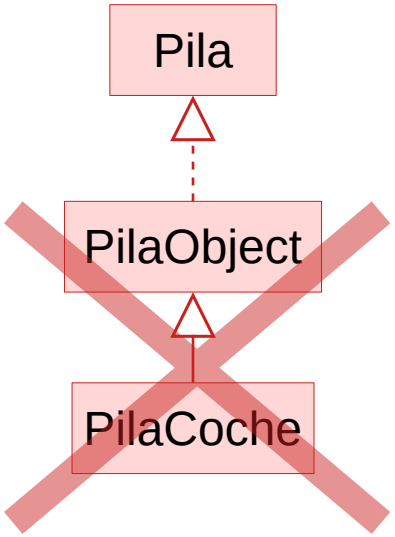
Preferir composición frente a herencia

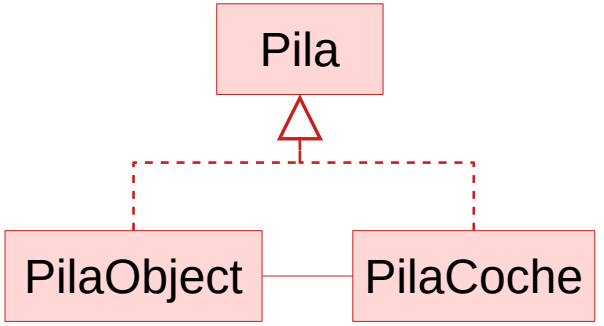
La herencia rompe la encapsulación.

Siempre es posible obtener un comportamiento similar a la herencia utilizando composición.

Para forzar esta preferencia es una buena práctica siempre definir las clases y los métodos como finales, evitando de esta forma que se pueda heredar de ellos.

Solo en los casos en los que la herencia se planifique se debe omitir el modificador final.





Command–query separation (CQS)

Cada método debe ser un comando que realiza una acción, o una consulta que devuelve datos al llamante, pero no ambos.

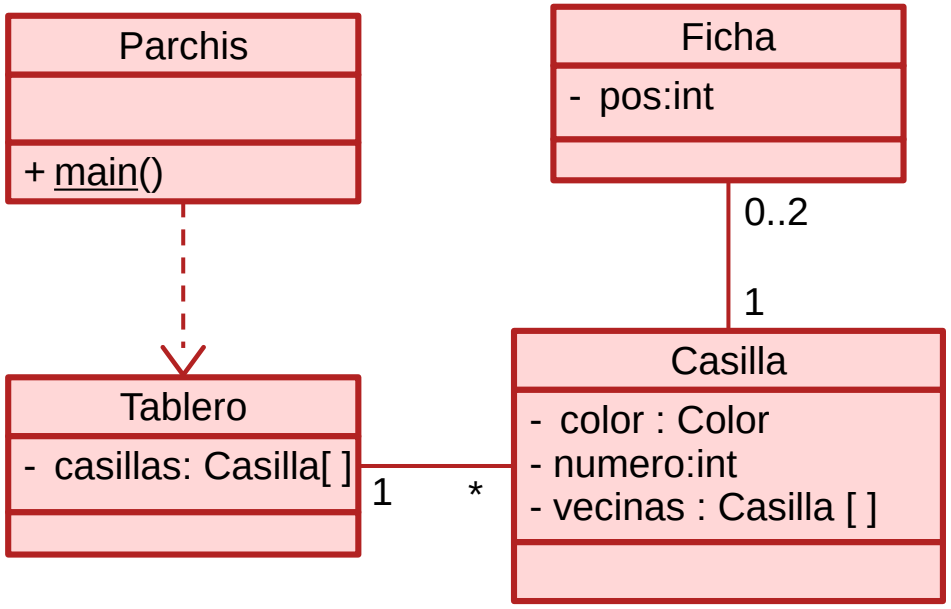
Bertrand Meyer (1988)
Object-oriented Software Construction
Prentice Hall.

Information Expert (GRAPS)

Solo se debe asignar una responsabilidad a una clase si dicha clase tiene la información necesaria para cumplirla.

Según la figura:

- ¿Qué clase debería mover las fichas y qué métodos deberían añadirse?
- ¿Qué clase debería localizar la casilla N y qué métodos deberían añadirse?
- ¿Qué clase comprobaría si una ficha come al caer en la casilla N y qué métodos habría que añadir?



Introducción a los patrones de diseño

En esta sección se presentan el concepto de patrón de diseño.

Patrones de diseño

Los diseñadores noveles no saben hacer buenos diseños, mientras que los experimentados conocen multitud de buenas soluciones a problemas que ya han tenido que resolver, y reutilizan estas soluciones adaptándolas a los nuevos problemas que abordan.

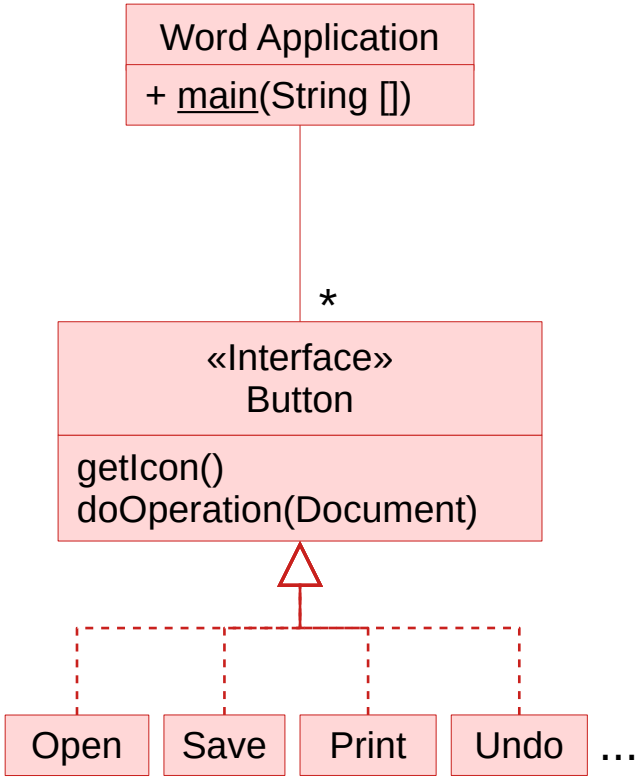
Los Patrones de Diseño capturan esa experiencia en diseños genéricos aplicables a muchos problemas.

"Cada patrón de diseño describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema de tal modo que se pueda aplicar esta solución una y otra vez, sin repetir cada vez lo mismo".
(Christopher Alexander, arquitecto)

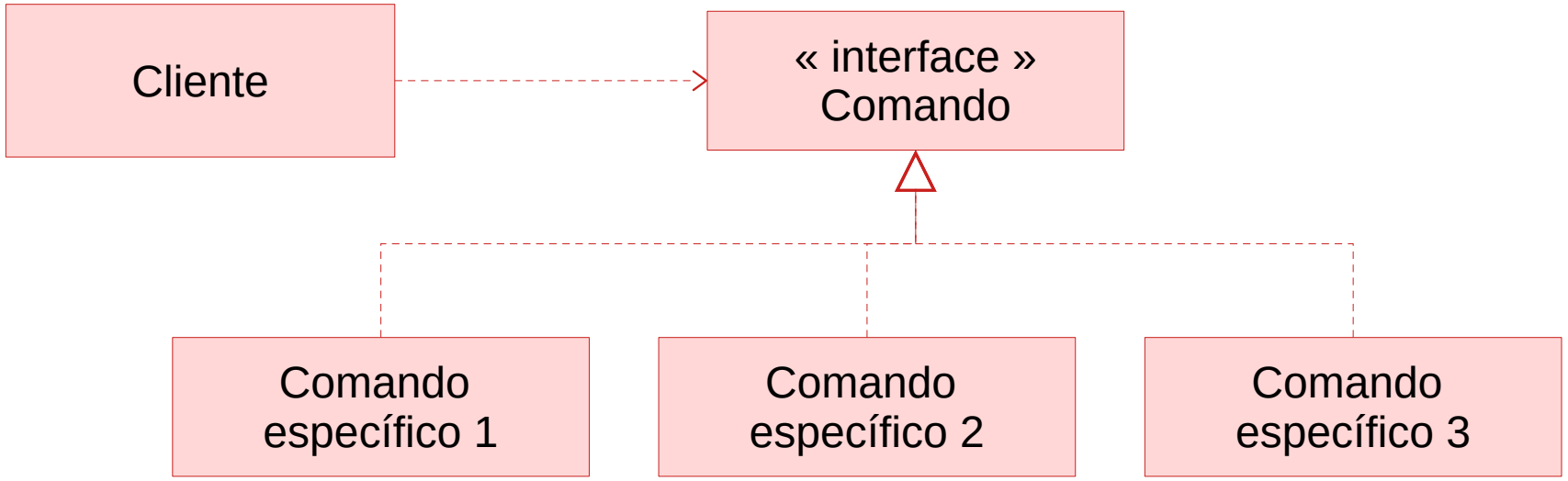
Patrones de diseño

En lo que sigue, cada vez que se explique una funcionalidad de Java que corresponda a un patrón de diseño reconocible se explicará dicho patrón de diseño.

Por ejemplo, cuando en el capítulo 3 se presentó un uso de la herencia y el polimorfismo para los botones de un procesador de texto no se dijo que se correspondía al Patrón Comando.



Patrón Comando



Referencias

- R. C. Martin, Design Principles and Design Patterns, 2000 [pdf].
- Clean Code, Robert C. Martin, Prentice Hall, 2008.
- Effective Java, Joshua Bloch, Addison Wesley, 2017.
- Head First Java, K. Sierra y B. Bates, O'Reilly, 2004.
- Head First Design Pattern, Erik Freeman, O'Reilly, 2004.
- Patrones de diseño, Gamma y Erich, Addison Wesley, 2002.
- Object-Oriented Analysis and Design, G. Booch, Benjamin Cummings, 1994.
- Unified Modeling Language – Metamodel - v2.5.1 – (Pág 374) [pdf].
- Refactoring Guru - Design Patterns [web]