Curso 2020-2021

Ingeniería del Software

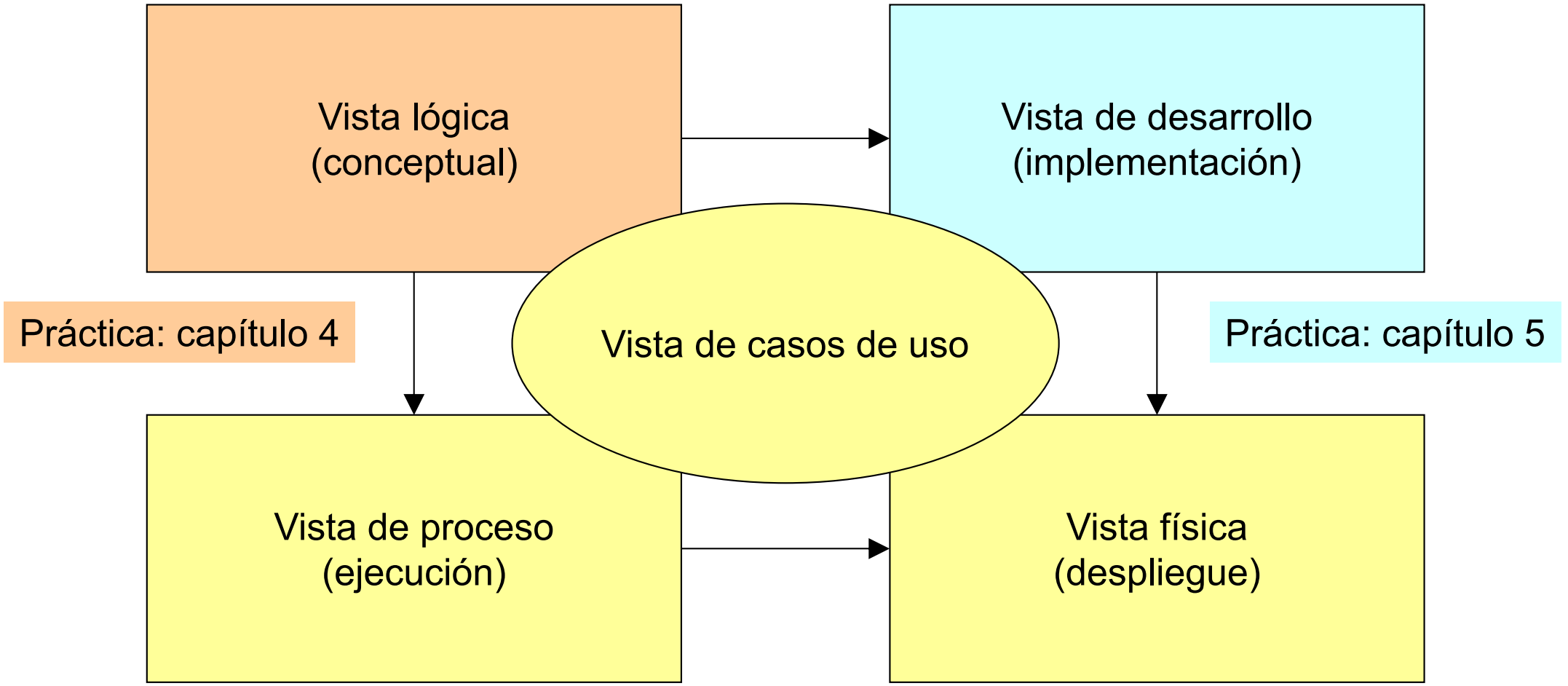
ARQUITECTURA

Arquitectura del software: definiciones

- Paul Clements 1996
  - La arquitectura del software es a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema.
- Len Bass 1998
  - La arquitectura del software de un programa o sistema de computación es la estructura o las estructuras del sistema, que contienen componentes de software, las propiedades externamente visibles de dichos componentes y las relaciones entre ellos.

- IEEE Std. 1471-2000
  - La arquitectura del software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y con el entorno, y los principios que orientan su diseño y evolución.

El modelo de 4+1 vistas arquitectónicas

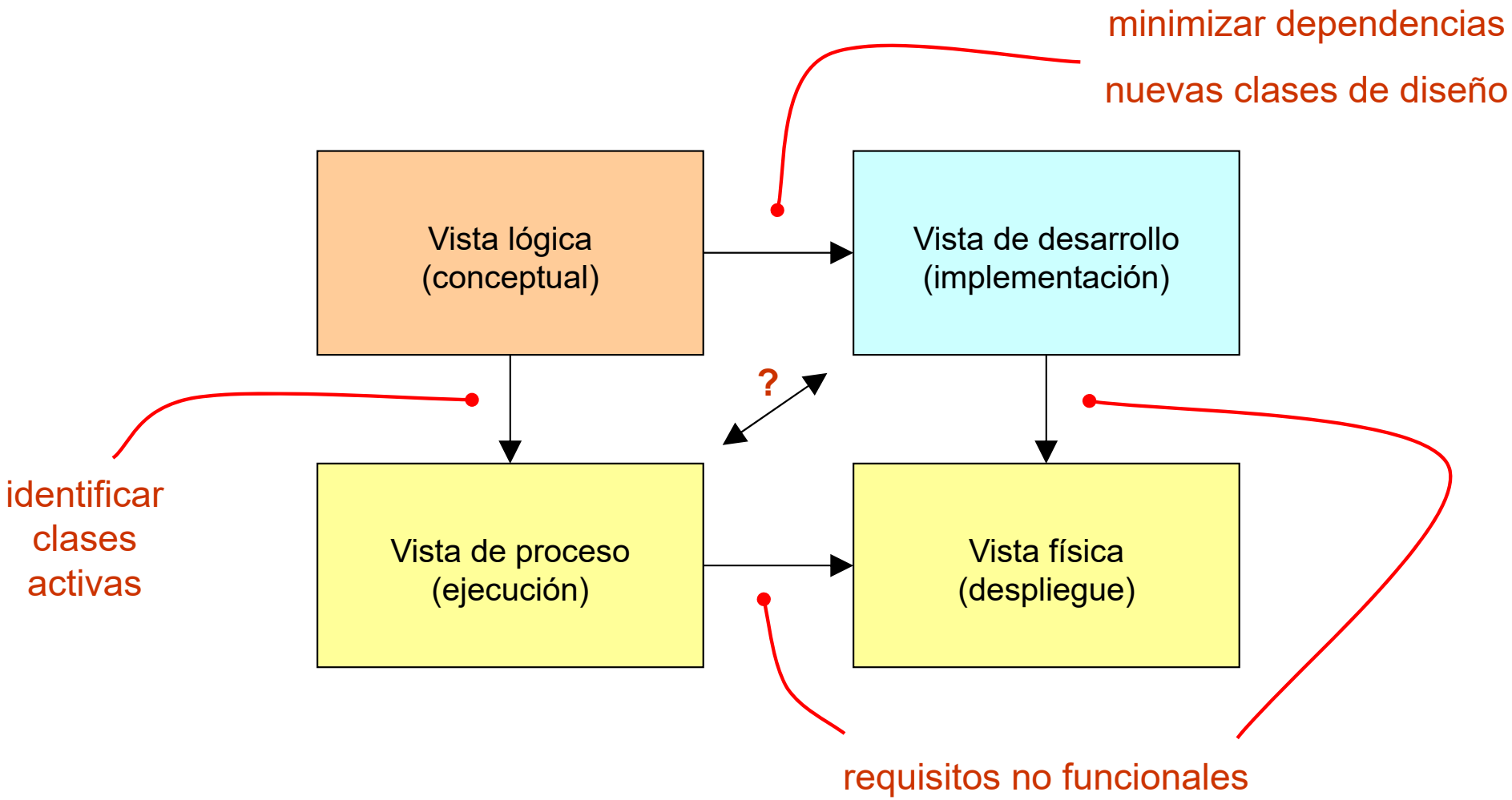


Vistas que no entran en la práctica

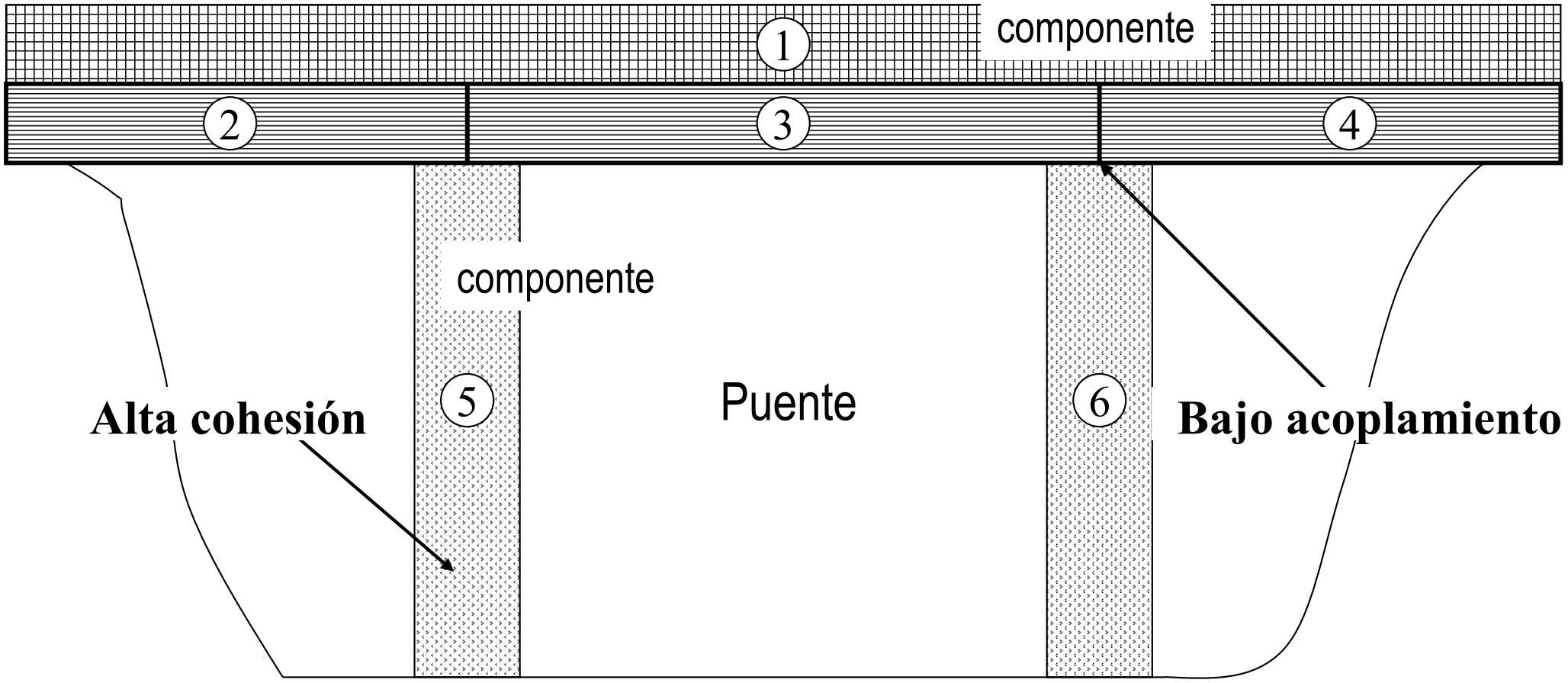
Características de cada vista en el modelo 4+1

Vista	Lógica (conceptual)	Proceso (ejecución)	Desarrollo (implementación)	Física (despliegue)
Aspecto	Modelo de información	Concurrencia y sincronización	Organización del software en el entorno de desarrollo	Correspondencia software-hardware
Stakeholders	Usuarios finales	Integradores del sistema	Programadores	Ingenieros de sistemas
Requisitos	Funcionales	Rendimiento Disponibilidad Fiabilidad Concurrencia Distribución Seguridad	Gestión del software Reuso Portabilidad Mantenibilidad Restricciones impuestas por la plataforma o el lenguaje	Rendimiento Disponibilidad Fiabilidad Escalabilidad Topología Comunicaciones
Notación	Clases y asociaciones	Procesos y comunicaciones	Componentes y relaciones de uso	Nodos y rutas de comunicación

Relaciones entre las cuatro vistas

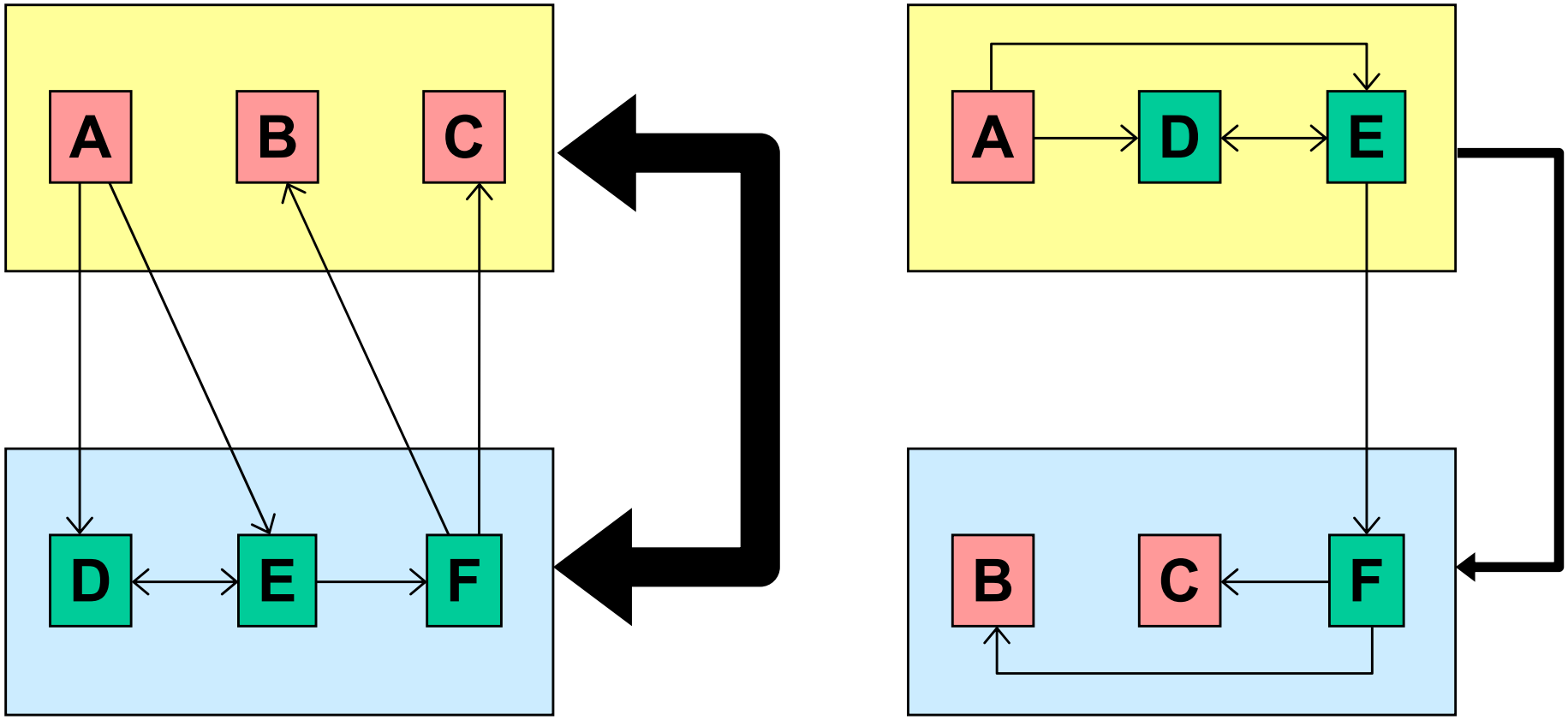


Cohesión y acoplamiento



Adaptado de E. Braude,
Software Engineering: An Object-Oriented Perspective

Cómo lograr una buena descomposición modular



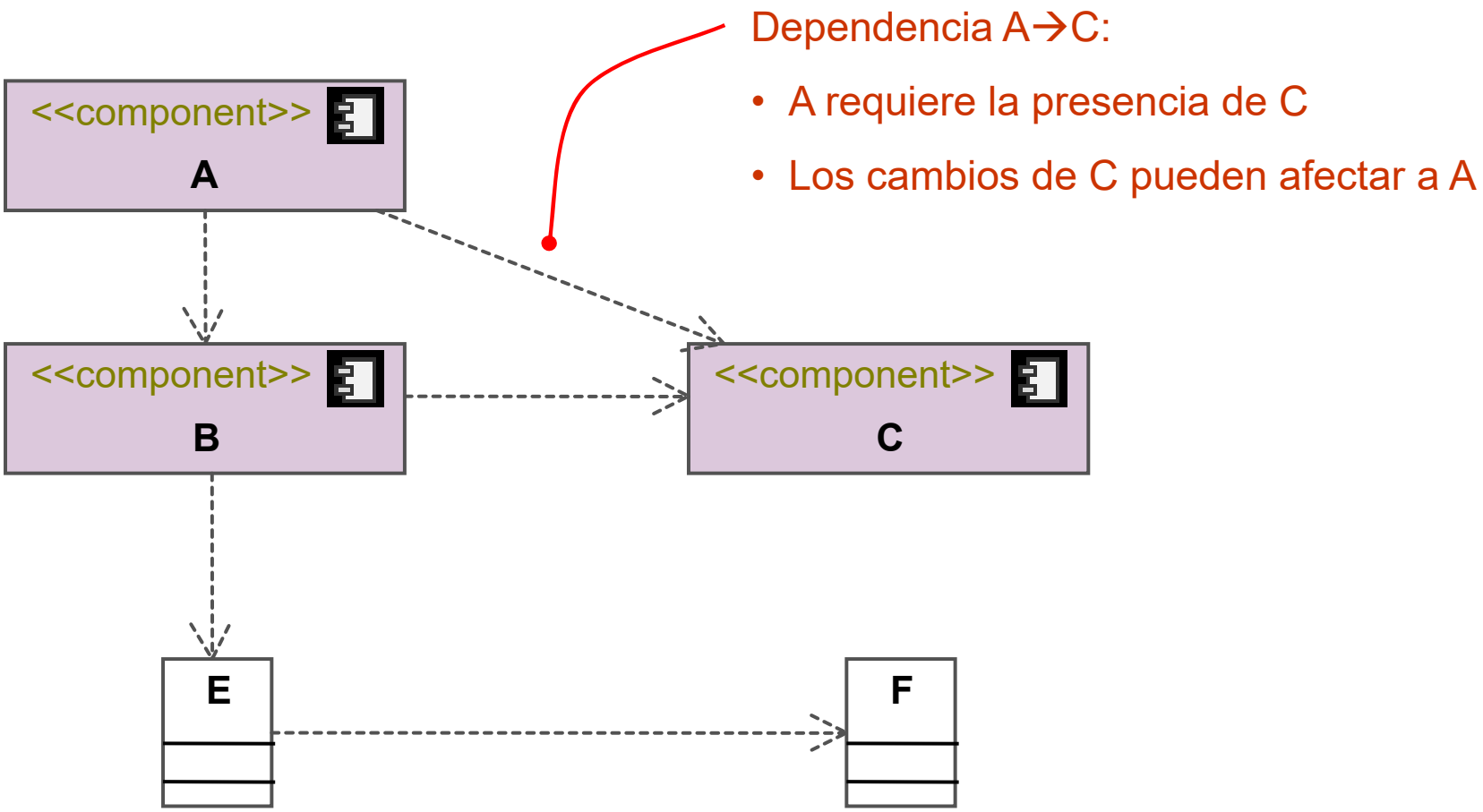
Mal **Bien**

¿Cómo acertar de antemano? → ***Estilos arquitectónicos***

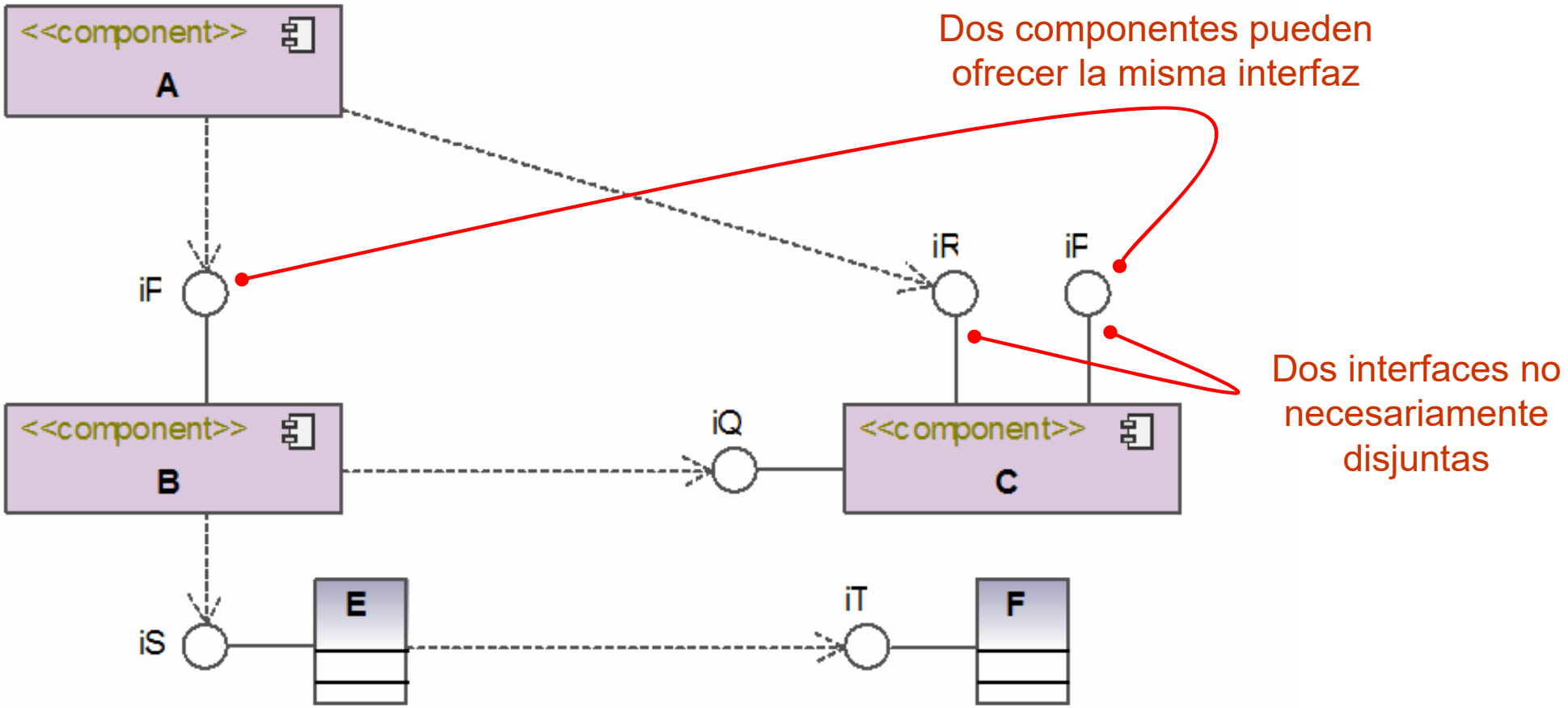
Criterios para la selección de una arquitectura

- Criterios clásicos:
  - **Extensibilidad:** facilitar la adición de nuevas caracterísiticas
    - Hace más complejo el diseño.
    - Aporta mayor grado de abstracción.
      - Ejemplo: arquitectura que soporte no este juego de tablero particular, sino cualquier tipo de juego de tablero.
    - Generalizar requiere invertir tiempo en el diseño: decidir qué tipo de extensiones pueden surgir, etc.
    - La distinción de requisitos opcionales/deseables es útil aquí, ya que señala hacia dónde apunta el desarrollo del sistema.
  - **Modificabilidad:** facilitar el cambio de requisitos.
    - Es distinto del anterior, aunque requiera técnicas similares.
      - Ejemplo: posibilidad de cambiar las reglas del juego.
  - **Simplicidad:** hacer fácil de entender, hacer fácil de implementar.
    - Difícil de coordinar con los dos anteriores.
  - **Eficiencia:** lograr alta velocidad o pequeño tamaño.
- Otros criterios: Reuso, Escalabilidad, Coste, Requisitos no funcionales…

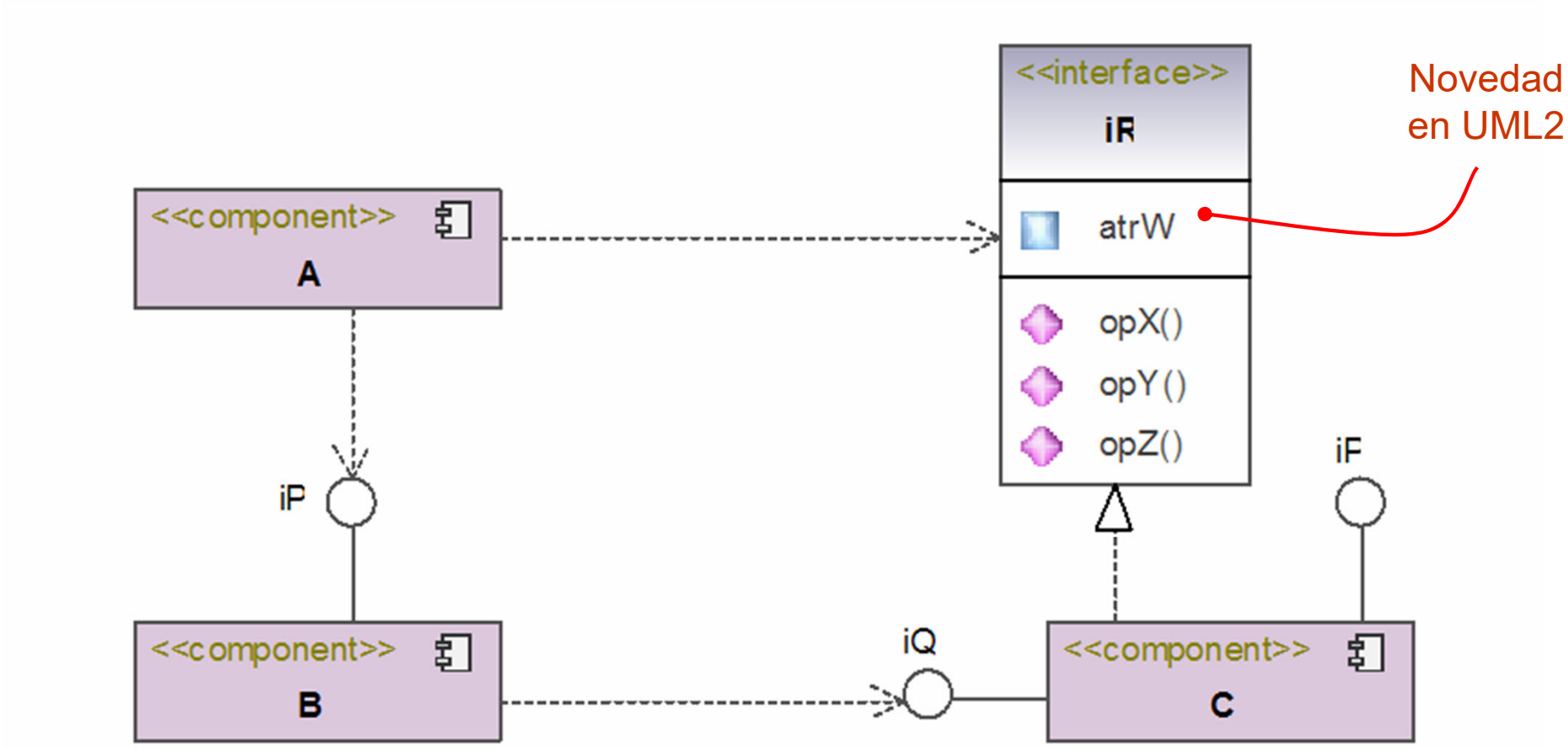
Noción de componente y dependencia



Dependencia respecto a una interfaz

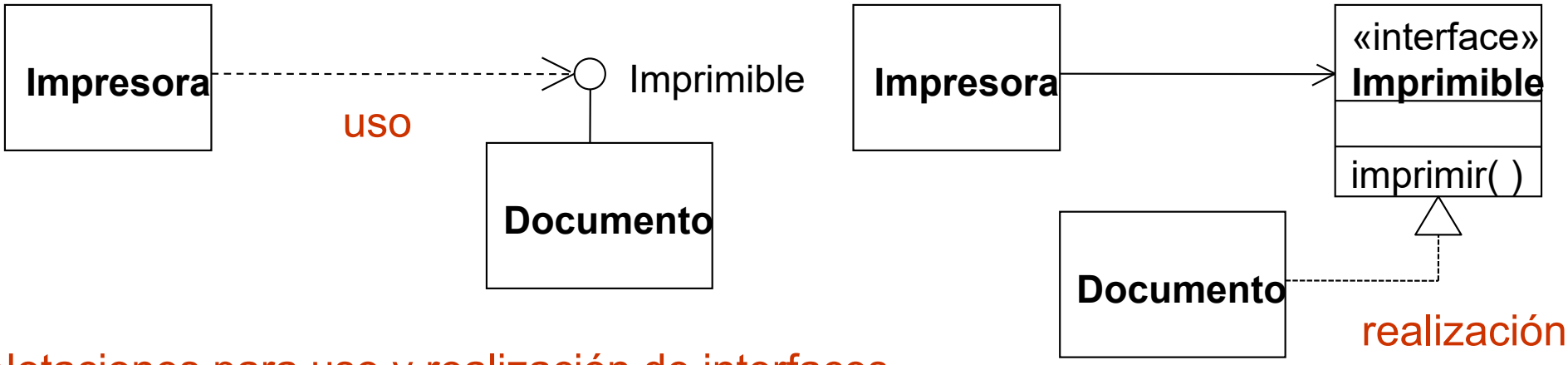


Representación de interfaces: abreviada y completa



Noción de interfaz

- Encapsulamiento: separación de interfaz e implementación:
  - una clase/componente puede realizar una o varias interfaces.
  - una interfaz puede ser realizada por una o varias clases/componentes.
- Interfaz: **conjunto de operaciones** que ofrecen un servicio coherente:
  - no contiene la implementación de las operaciones (**métodos**).
  - una interfaz no puede tener **atributos** ni **asociaciones** navegables (esta restricción ha desaparecido en UML 2.0).
  - análoga a una **clase abstracta** con todas sus operaciones abstractas: no puede tener instancias directas.

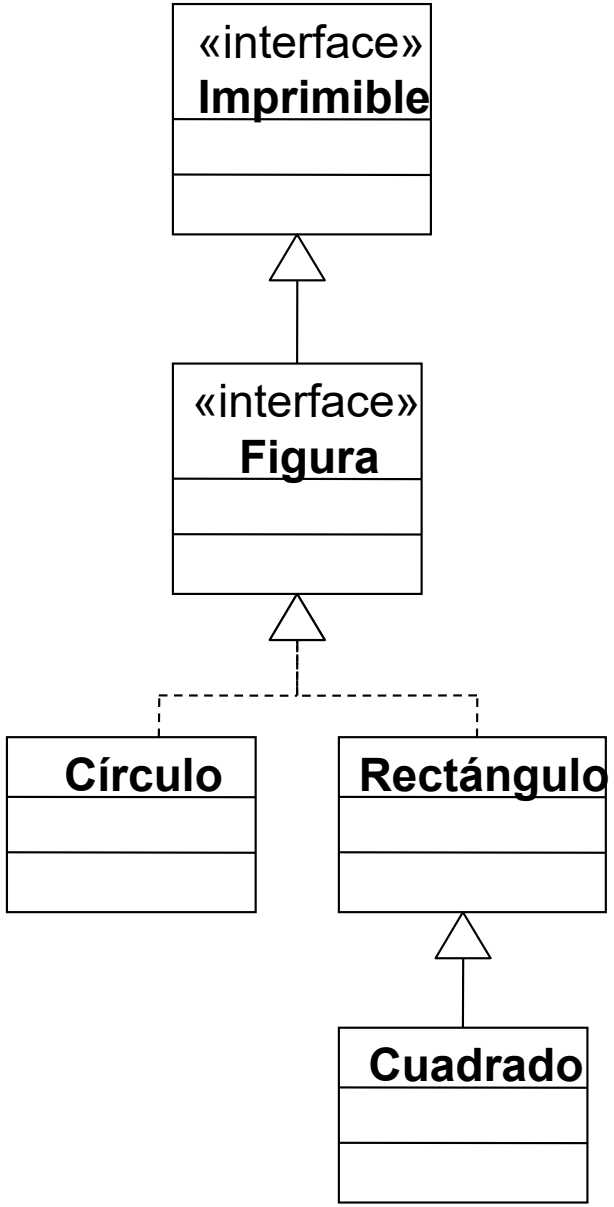


Notaciones para uso y realización de interfaces

Generalización vs. Realización

- La realización puede entenderse como una **“generalización débil”**: se hereda la interfaz, pero no la implementación:
  - reduce la dependencia.
  - disminuye la reutilización.
  - alternativa a la generalización múltiple, no soportada por muchos lenguajes.
- Las interfaces son elementos generalizables:
  - jerarquías mixtas de interfaces y clases.
- Criterio de diseño: comprometerse sólo con la interfaz.
  - declarar el tipo de las variables y parámetros de operaciones como interfaces, no como clases.
  - servirá cualquier instancia compatible con la interfaz.
- Ejemplo:

```
Figura f = new Cuadrado ( );
f.imprimir
```



Interfaces proporcionadas y requeridas

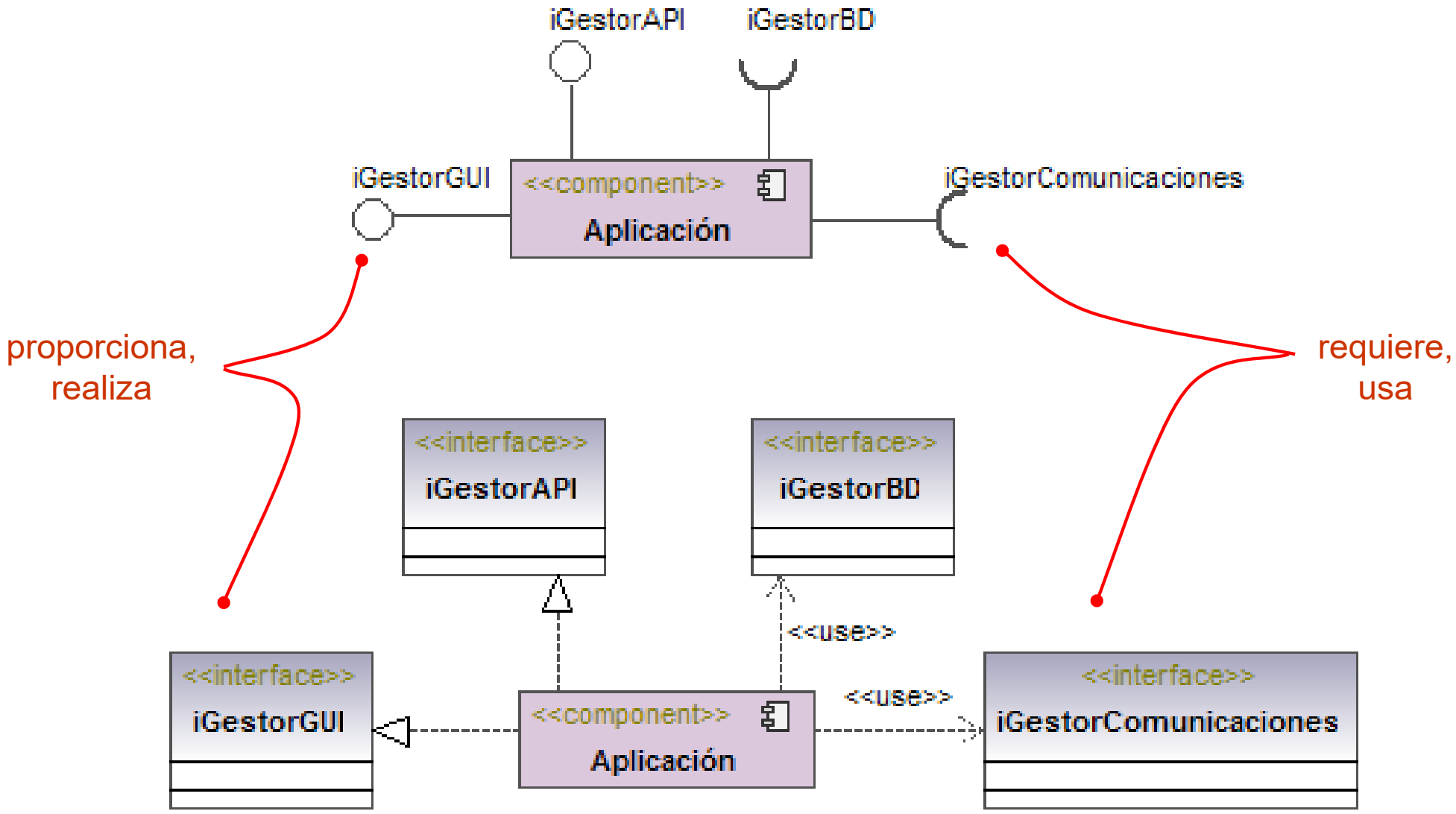
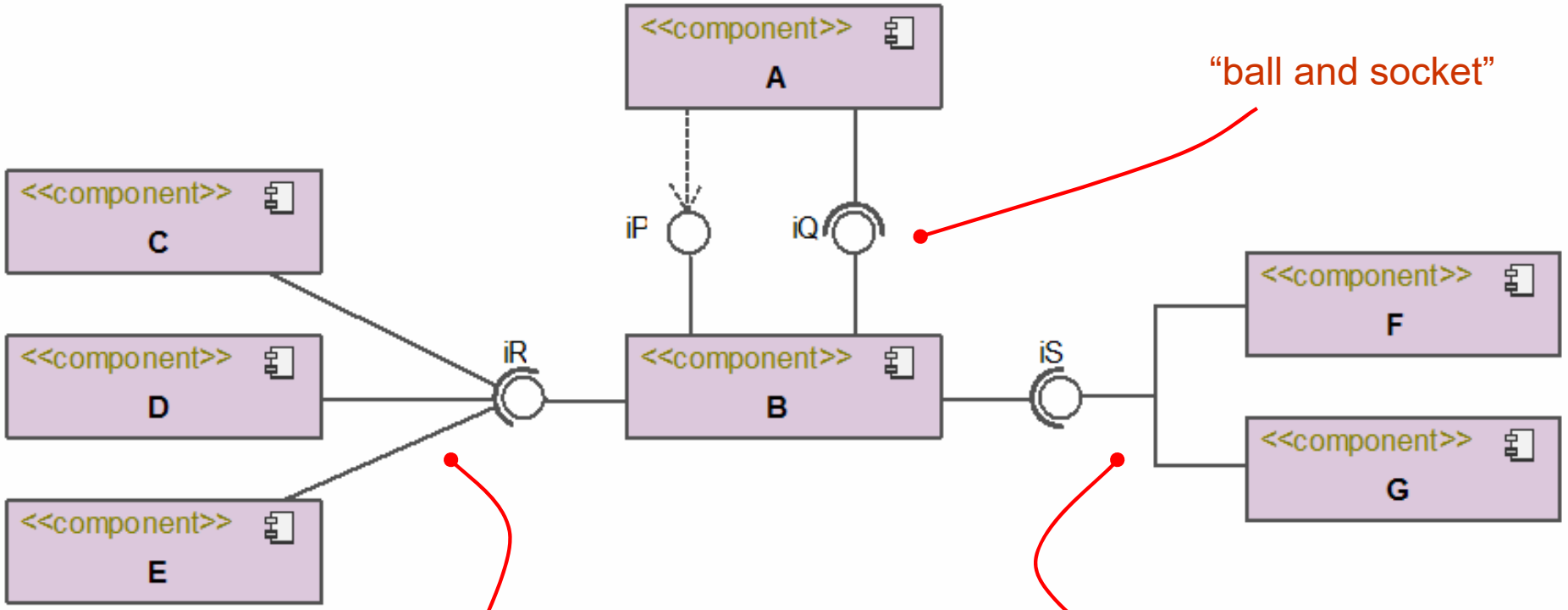


Diagrama de componentes



cableado con dependencia

Cómo instanciar las interfaces de un componente (1)

```
package GestiónEmpleados;

public interface iEmpleado {
        public modificar( ) { };
        …
 }

public class EmpleadoFijo implements iEmpleado {
        public modificar( ) { };
        …
}

public class EmpleadoTemporal implements iEmpleado {
        public modificar( ) { };
        …
}

====================

class Departamento {
        …
        iEmpleado e = new EmpleadoFijo( );
        …
        e.modificar( );
        …
}
```

El componente proporciona sólo una interfaz, el tipo abstracto de datos.

Es necesario que la clase cliente (Departamento) conozca las clases concretas EmpleadoFijo, etc.: estas clases deben ser públicas.

Nada le impide usarlas sólo para invocar los constructores (salvo el buen estilo del programador).

La dependencia es potencialmente “peligrosa”.

Cómo instanciar las interfaces de un componente (2)

```
package GestiónEmpleados;

public interface iGestorEmpleados {
      public int crearEmpleado( ); // el gestor debe mantener algún tipo de lista
      public modificarEmpleado(int e);
      …
 }

class EmpleadoFijo {…} // no visible desde fuera, idem EmpleadoTemporal

public class GestorEmpleados implements iGestorEmpleados {
      public int crearEmpleado( ) {…}; // usa new EmpleadoFijo( )
      public modificarEmpleado (int e) {…}; // usa EmpleadoFijo.modificar( )
      …
}

====================

class Departamento {
      …
      int e = ge.crearEmpleado ( ); // se supone que existe GestorEmpleados ge
      …
      ge.modificarEmpleado(e);
      …
}
```

El componente proporciona sólo una interfaz, el gestor.

GestorEmpleados no puede recibir ni devolver valores de tipo EmpleadoFijo, EmpleadoTemporal, etc.

Es necesario un mecanismo de traducción de los identificadores públicos a las instancias de EmpleadoFijo, etc. (una lista interna).

La clase cliente sólo usa el gestor y los identificadores públicos.

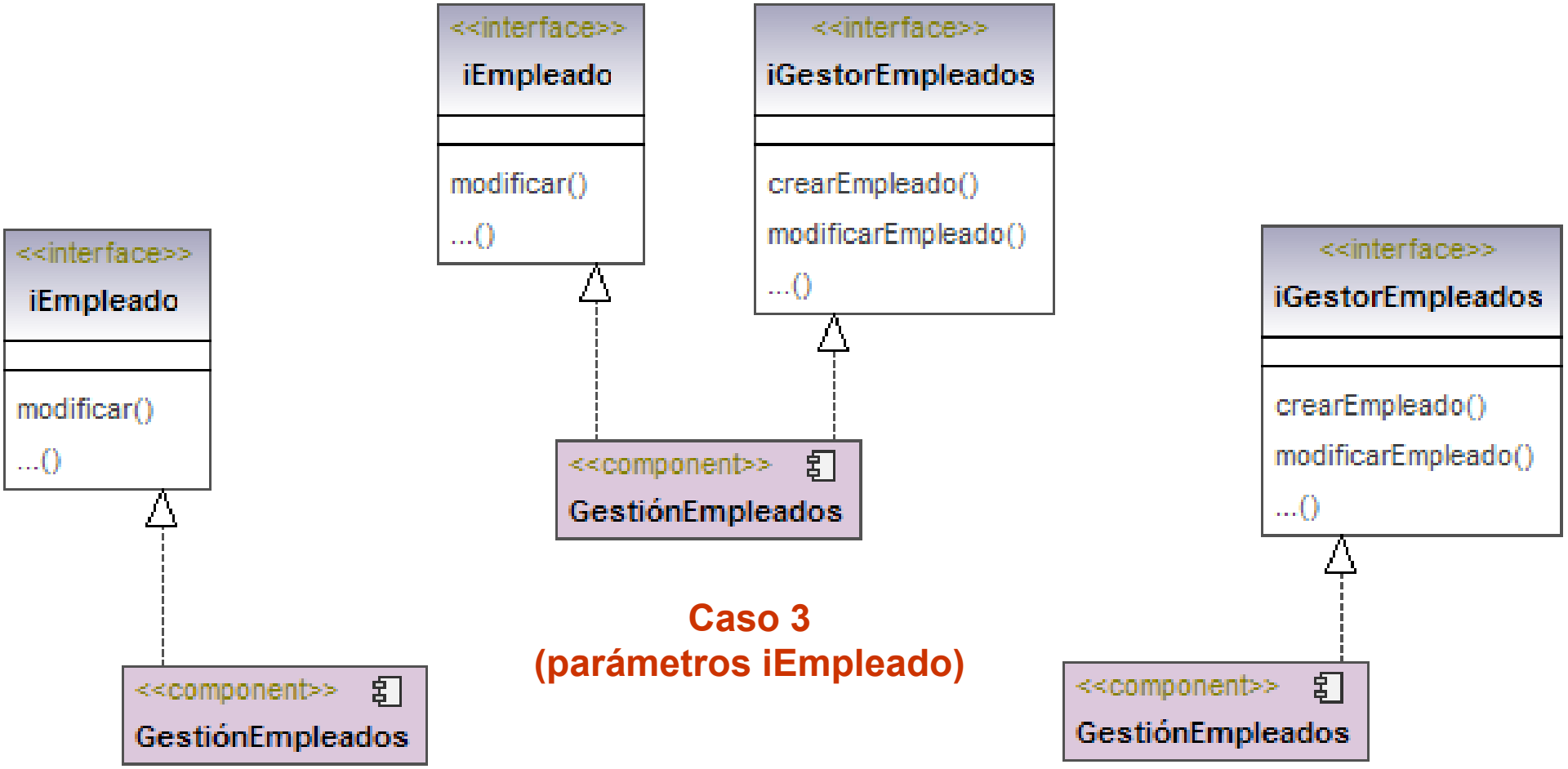
Cómo instanciar las interfaces de un componente (3)

```
package GestiónEmpleados;

public interface iEmpleado {
        public modificar( ) { };
        …
 }

public interface iGestorEmpleados {
        public iEmpleado crearEmpleado( ); // el gestor debe mantener algún tipo de lista
        public modificarEmpleado(iEmpleado e);
        …
 }

====================

class Departamento {
        …
        iEmpleado e = ge.crearEmpleado ( ); // se supone que existe GestorEmpleados ge
        …
        e.modificar( );
        …
}
```

El componente proporciona dos interfaces, el tipo abstracto de datos y el gestor.

La clase cliente no puede usar el constructor directamente, pero sí el tipo abstracto en el resto de operaciones.

Cómo instanciar las interfaces de un componente (4)



Diseño por contratos: PPCs e invariantes

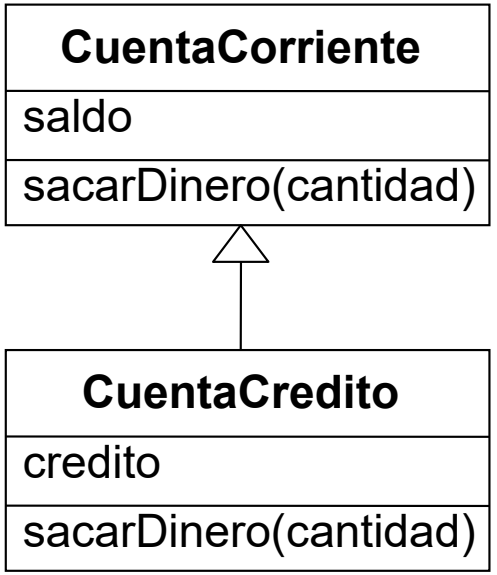
- PPCs de operación
  - CuentaCorriente.meterDinero(cantidad: Moneda)
    - Pre:
    - Post: saldo' = saldo + cantidad
  - CuentaCorriente.sacarDinero(cantidad: Moneda)
    - Pre: saldo – cantidad ≥ 0
    - Post: saldo' = saldo – cantidad

CuentaCorriente
saldo
meterDinero(cantidad) sacarDinero(cantidad)

- Invariantes de clase
  - CuentaCorriente: saldo ≥ 0

Diseño por contratos: subcontratación

- PPC's de operación
  - CuentaCorriente.sacarDinero(cantidad: Moneda)
    - Pre: saldo – cantidad ≥ 0
    - Post: saldo' = saldo – cantidad
- PPC's de operación redefinida en la subclase
  - CuentaCredito.sacarDinero(cantidad: Moneda)
    - Pre: saldo + credito – cantidad ≥ 0
    - Post: saldo' = saldo – cantidad
  - Pre: **menos restrictiva**, o igual
  - Post: más restrictiva, o **igual**



¿Es correcta esta generalización?

- Invariantes de clase (más restrictivo, o igual)
  - CuentaCorriente: saldo ≥ 0
  - CuentaCredito: (credito ≥ 0) AND (saldo + credito ≥ 0)

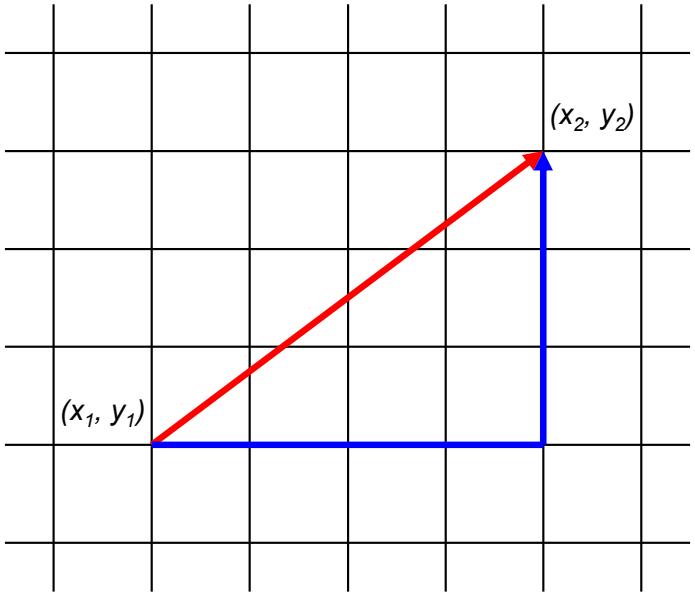
Diseño por contratos: uso correcto de jerarquías

- CaminoOblicuo.distancia.post

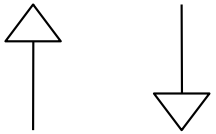
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- CaminoEsquinado.distancia.post

$$d = (x_2 - x_1) + (y_2 - y_1)$$



¿Cuál sería correcta?



CaminoOblicuo
origen: Punto destino: Punto
distancia( ): Float

CaminoEsquinado
origen: Punto destino: Punto
distancia( ): Float