Paradigma di programmazione OO

Metafora per descrivere l'invocazione di procedure/funzioni

a) Gli oggetti si scambiano messaggi
b) Per rispondere a un messaggio, un oggetto invoca un'operazione appropriata

Incapsulamento: un oggetto incapsula

- Dati, la cui configurazione di valori è lo stato dell'oggetto, invisibile all'esterno dello stesso
- Operazioni, dette metodi o *member function*, che sono le sole che possono modificare lo stato dell'oggetto

Classe (concreta) = implementazione di un oggetto, cioè ogni oggetto è istanza di una classe

Paradigma di programmazione OO (cont.)

Classe astratta = classe che differisce l'implementazione di uno o più metodi (anche tutti), detti metodi astratti, alle sue sottoclassi → non può essere istanziata

Signature di un metodo = nome + parametri (+ tipo del valore di ritorno)

Sovraccarico di un metodo (o *method overloading*) = presenza all'interno di una classe di più metodi che hanno lo stesso nome ma signature diverse

Ereditarietà fra classi = una classe, detta sottoclasse, può essere definita in termini di una o più classi esistenti, dette superclassi o classi genitrici; la sottoclasse, oltre alle sue proprie definizioni e operazioni, comprende le definizioni dei dati e delle operazioni della sua superclasse e può sovrascrivere queste ultime, ma senza modificarne la signature (sovrascrittura di un metodo o *method overriding*)

Paradigma di programmazione OO (cont.)

Interfaccia di un oggetto = insieme di tutte le *signature* dei suoi metodi; un oggetto è accessibile solo attraverso la sua interfaccia → oggetti con implementazioni dei metodi molto diverse possono condividere l'interfaccia

Tipo di un oggetto = sottoinsieme non vuoto della sua interfaccia (dove ogni sottoinsieme è un tipo e può essere visto, a sua volta, come un'interfaccia) → oggetti molto diversi possono condividere dei tipi

Sottotipo e supertipo = un tipo è un sottotipo di un altro, detto supertipo, se la sua interfaccia contiene quella del supertipo

Il paradigma di programmazione OO (cont.)

Binding dinamico (o *late binding*) = associazione, al momento dell'esecuzione, di una richiesta inoltrata a un oggetto a uno dei metodi dello stesso

Sostituibilità = possibilità di sostituire l'un l'altro durante l'esecuzione oggetti che condividono un tipo

Polimorfismo = fatto di ottenere, a fronte di una medesima richiesta di servizio da parte di un oggetto client, comportamenti diversi al momento dell'esecuzione a seconda dell'oggetto che soddisfa dinamicamente la stessa; è causato dalla sovrascrittura dei metodi

Vantaggi dell'uso delle classi astratte

Esse definiscono l'interfaccia comune, che consente la manipolazione uniforme di oggetti che sono istanze di classi diverse e che, a loro volta, usano tipi di oggetti diversi
→ riduzione delle dipendenze di implementazione fra sottosistemi

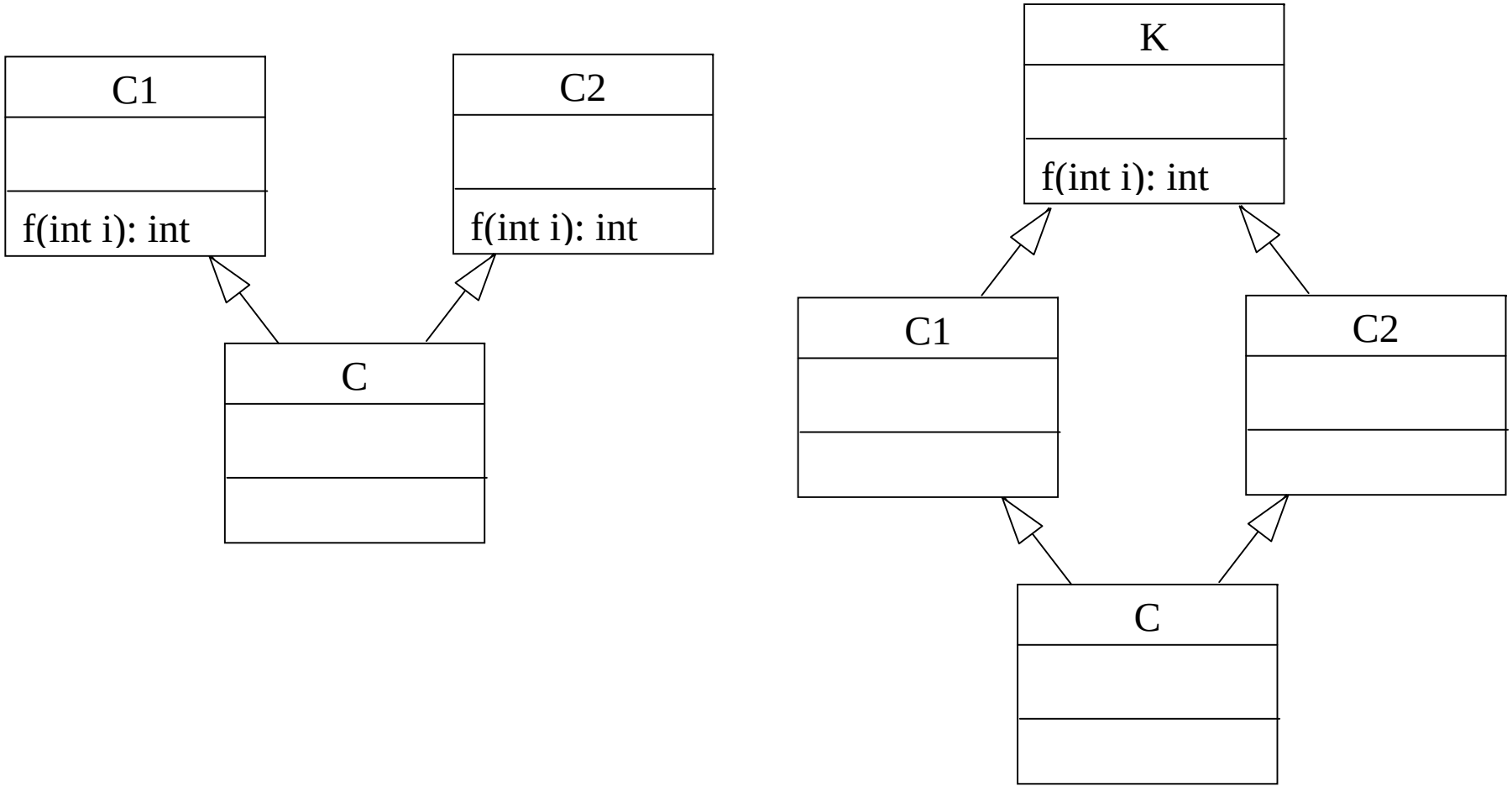
→ Primo principio di progettazione OO: programmare rivolti all'interfaccia, non all'implementazione

Limiti dell’ereditarietà semplice

Non permette di descrivere numerose situazioni reali (ad es. date due classi Giocattolo e Automobile, non è possibile definire la classe AutomobileGiocattolo)

Problemi dell'ereditarietà multipla

È possibile ereditare due o più metodi con la stessa signature da più superclassi
→ conflitto fra implementazioni diverse



Ereditarietà semplice e multipla: la soluzione Java

Usare l'ereditarietà

- semplice (fra classi) per descrivere una gerarchia di implementazione, finalizzata al riutilizzo del codice
- multipla (da *interface*) per descrivere una gerarchia di tipi

Interface Java = classe, priva di attributi non costanti, i cui metodi sono tutti pubblici e astratti

a) Una *interface* può ereditare da una *interface*
b) Una classe (astratta o concreta) può implementare una o più *interface*

↓

nessun conflitto fra implementazioni diverse di metodi omonimi
perché i metodi delle *interface* sono astratti

Interface, polimorfismo e binding dinamico in Java

Una *interface* può essere usata come tipo di una variabile → questa variabile potrà riferirsi a qualsiasi oggetto che implementi *l'interface* (polimorfismo)

Tipo statico di una variabile = tipo usato nella dichiarazione della variabile (o nel passaggio della stessa come parametro di un metodo)

Tipo dinamico di una variabile = tipo del costruttore usato per creare la variabile

Vincolo: data una variabile, il suo tipo dinamico è un sottotipo (proprio o meno) del tipo statico

A fronte dell'invocazione di un metodo *m* su una variabile *x* (es. *x.m( )*), l'implementazione scelta per *m* dipende dal tipo dinamico di *x* (binding dinamico)

Riuso di funzionalità

- Riuso white-box: è quello che avviene attraverso l'ereditarietà; il termine white-box si riferisce alla visibilità del contenuto delle classi genitrici da parte delle sottoclassi (l'ereditarietà spezza l'incapsulamento)

- Riuso black-box: è quello che avviene attraverso la composizione di oggetti, ovvero una nuova funzionalità è ottenuta dinamicamente al momento dell'esecuzione assemblando oggetti (degli oggetti acquisiscono i riferimenti di altri); gli oggetti appaiono come black-box perché i loro dettagli interni non sono visibili

Riuso white-box

Vantaggi

- È supportato direttamente dal linguaggio di programmazione

Svantaggi

- L'implementazione ereditata dalle classi genitrici non può essere modificata durante l'esecuzione
- Ogni cambiamento nella classe genitrice forza un cambiamento nella sottoclasse
- Le dipendenze implementative limitano flessibilità e riusabilità
- Ogni nuova classe ha un overhead implementativo fisso (inizializzazione, finalizzazione, ecc.)
- La definizione di una sottoclasse richiede una comprensione profonda della superclasse (ad es. la sovrascrittura di un'operazione potrebbe richiedere quella di un'altra; un'operazione sovrascritta può dover chiamare un'operazione ereditata; una semplice estensione può comportare la creazione di molte nuove sottoclassi)

Ragioni per cui usare l’ereditarietà

- Classificare gli oggetti in tassonomie comprensibili (relazioni IS-A)
- Distinguere il comportamento nel caso generale (superclasse) da quello dei casi particolari (sottoclassi)
- Ridurre la ridondanza e aumentare l’estensibilità (fattorizzando il comportamento ridondante in una singola superclasse si riduce il rischio di introdurre inconsistenze durante i cambiamenti)

Principio di sostituzione di (Barbara) Liskov (1988)

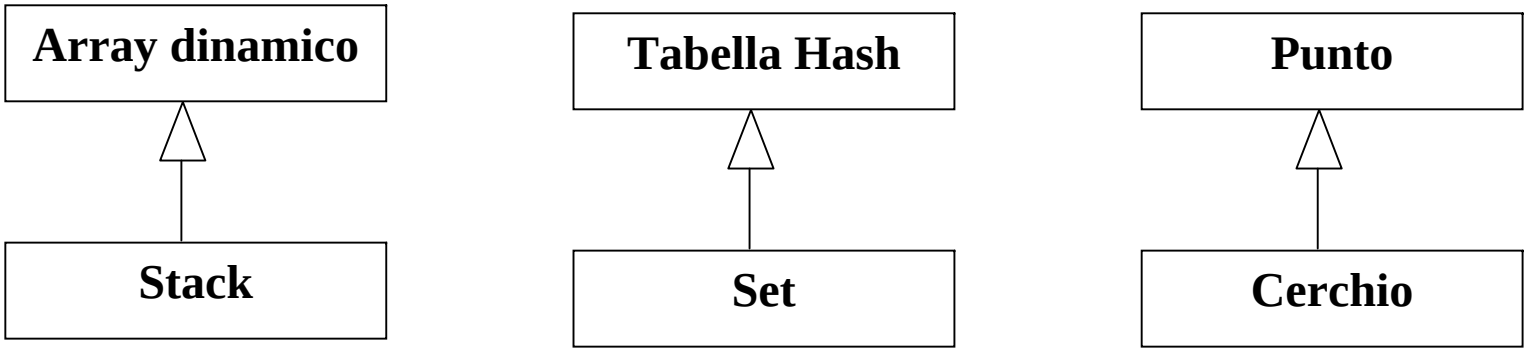
Se un oggetto di tipo S può essere sostituito ovunque ci si aspetti un oggetto di tipo T, allora S è una sottoclasse corretta di T

ovvero

È corretto (e conveniente) usare l'ereditarietà aggiungendo una sottoclasse S a una (super)classe T esistente sole se il codice preesistente (di T e/o client di T) non deve essere cambiato di conseguenza

↓

Non è corretto usare l'ereditarietà fra concetti logicamente scorrelati perché in tal caso possono esistere dei metodi di T che non hanno significato per S (ad es. sono scorrette tutte le derivazioni illustrate)



When Not to Use Inheritance (da Horstmann)

- From a tutorial for a C++ compiler:

```
public class Point
{
   public Point(int anX, int aY) { ... }
   public void translate(int dx, int dy) { ... }
   private int x;
   private int y;
}

public class Circle extends Point // DON'T
{
   public Circle(Point center, int radius) { ... }
   public void draw(Graphics g) { ... }
   private int radius;
}
```

When Not to Use Inheritance (da Horstmann)

- Huh? A circle isn't a point.
- By accident, inherited `translate` works for circles
- Same tutorial makes `Rectangle` a subclass of `Point`:

```
public class Rectangle extends Point // DON'T
{
  public Rectangle(Point corner1, Point corner2) { ... }
  public void draw(Graphics g) { ... }
  public void translate(int dx, int dy) { ... }
  private Point other;
}
```

When Not to Use Inheritance (da Horstmann)

- That's even weirder:

```
public void translate(int dx, int dy)
{
   super.translate(dx, dy);
   other.translate(dx, dy);
}
```

- Why did they do that?
- Wanted to avoid abstract class `Shape`
- Remedy: Use aggregation.
- `Circle`, `Rectangle` classes *have* points

When Not to Use Inheritance (da Horstmann)

- Java standard library:

```
public class Stack extends Vector // DON'T
{
   Object pop() { ... }
   void push(Object item) { ... }
   ...
}
```

- Bad idea: Inherit all `Vector` methods
- Can insert/remove in the middle of the stack
- Remedy: Use aggregation

```
public class Stack
{
   ...
   private Vector elements;
}
```

Dipendenze

Un elemento (di qualsiasi tipo) dipende da un altro se la modifica del secondo può causare la necessità di cambiare anche il primo

Una classe C1 dipende da un classe C2 se

- deriva da C2, e/o
- invoca operazioni di C2 (anche solo un costruttore), e/o
- un suo attributo è di tipo C2, e/o
- un parametro di una sua operazione è di tipo C2, e/o
- accede ai campi di C2

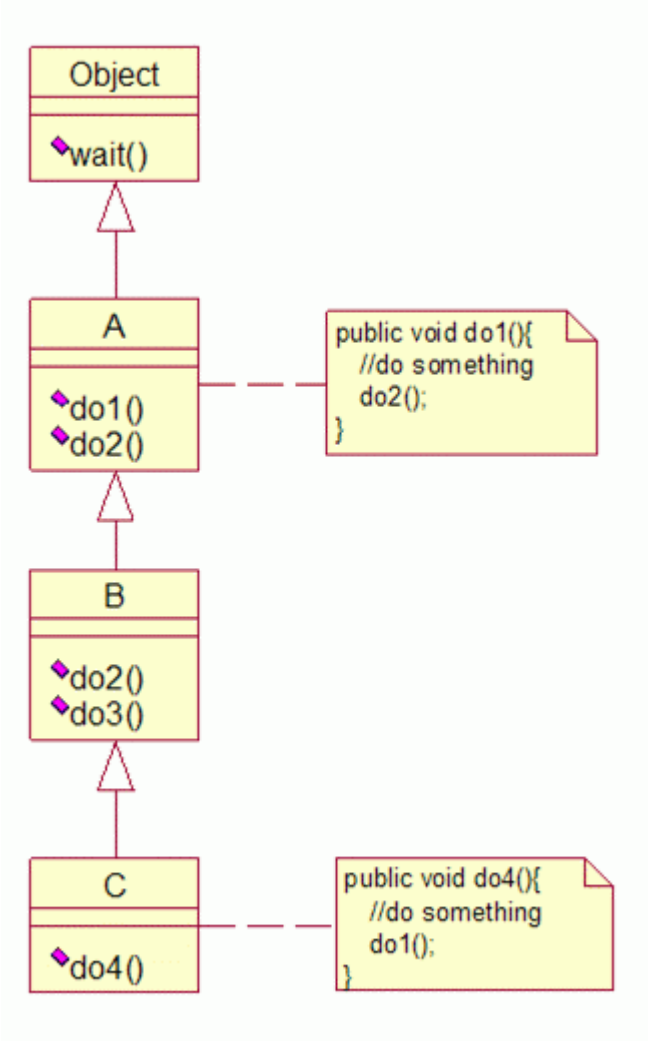
Una classe dipende da un interface se

- realizza tale interface (si dice che *fornisce* quell'interface)
- *richiede* tale interface (relazione «uses»)

Dipendenze di ereditarietà al momento della compilazione

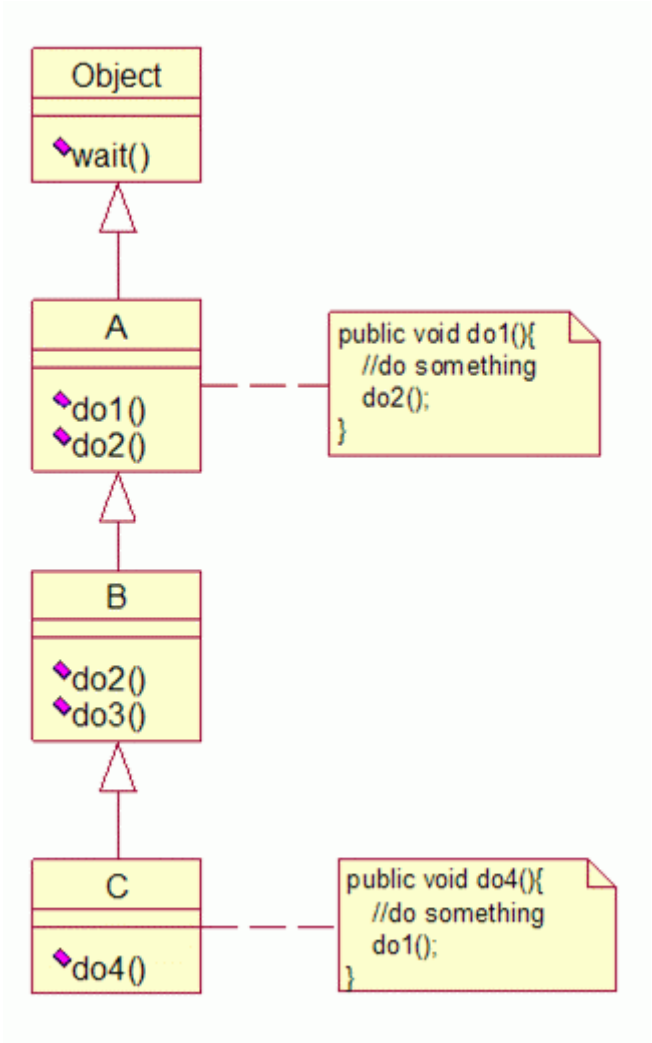
Queste dipendenze (statiche) hanno il verso delle relazioni di generalizzazione

Ogni cambiamento di `wait()` viene staticamente ereditato da tutte le sottoclassi

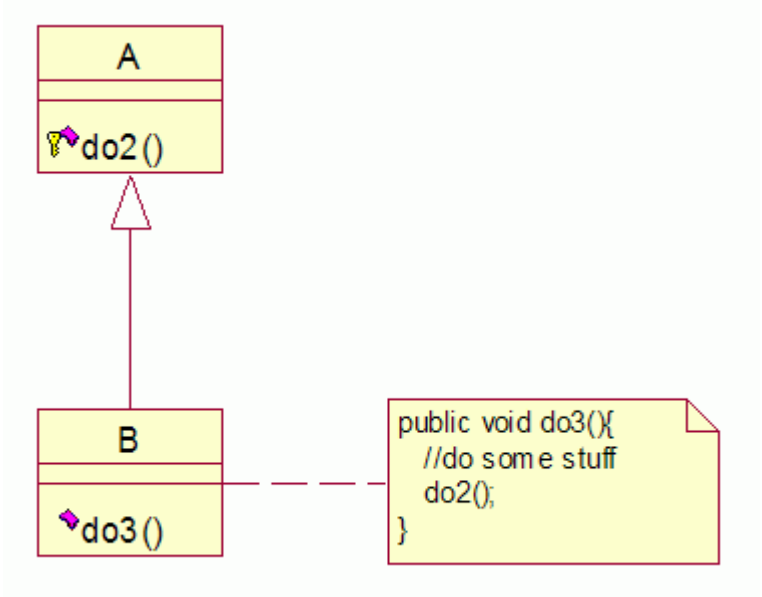


Dipendenze di ereditarietà al momento dell'esecuzione

Inoltre A dipende dinamicamente da B perché l'esecuzione del metodo `do1()` di A su un'istanza di B richiede l'esecuzione del metodo `do2()` di B → si crea una dipendenza circolare fra A e B difficile da controllare

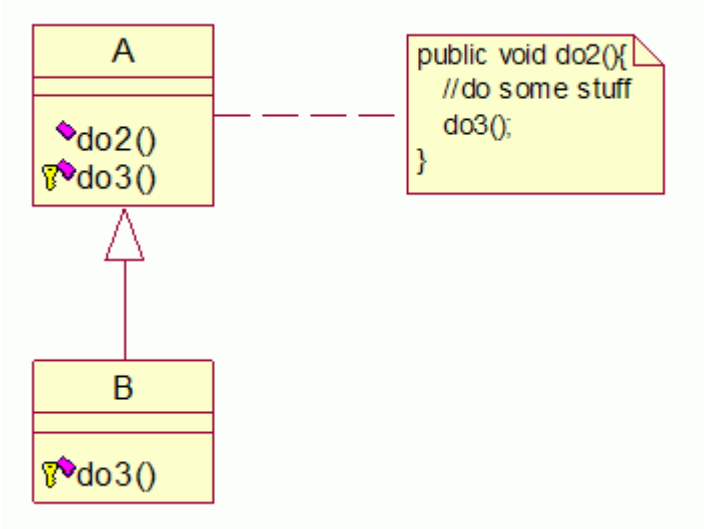


Ereditarietà senza polimorfismo



- Avviene quando non sussiste alcuna sovrascrittura di metodo
- È corretta secondo il principio di sostituzione di Liskov
- È la più facile da comprendere e gestire ma …
- … non è molto utile, mentre la sovrascrittura di metodi consente di ottenere comportamenti complessi con poco lavoro

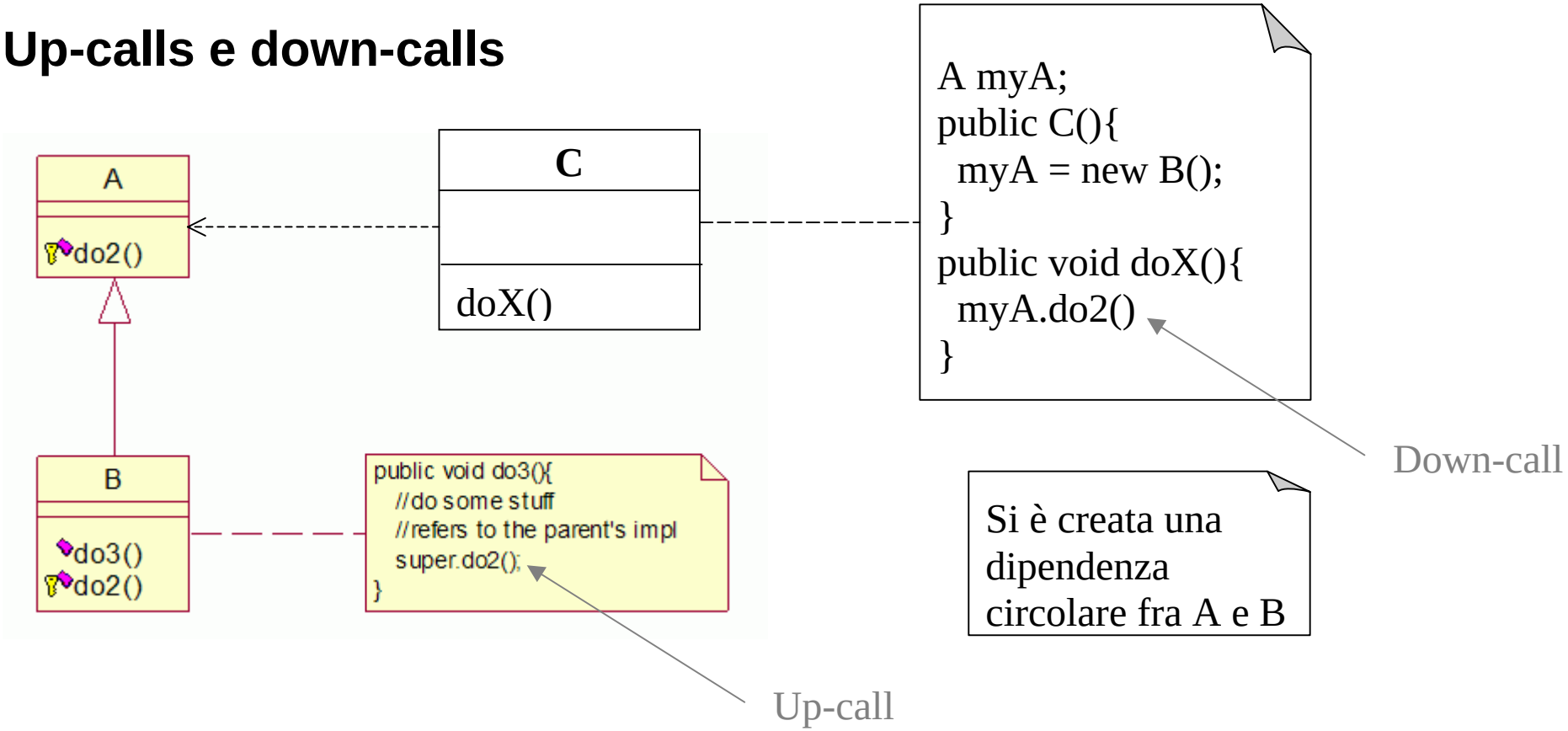
Ereditarietà con estensione (*extension inheritance*)



- Ogni sottoclasse è un caso particolare della superclasse
- La sovrascrittura di un metodo ereditato avviene solo per far funzionare lo stesso nello specifico contesto della sottoclasse

È da evitare l'ereditarietà con restrizione (*restriction inheritance*) secondo cui la sovrascrittura di un metodo ereditato avviene sopprimendo (in parte o in toto) la sua funzionalità

Up-calls e down-calls



Una combinazione dinamica di up- e down-call

- può dar luogo a problemi di manutenzione (rende complessa l'analisi di impatto di un cambiamento)
- un errore riscontrato nell'esecuzione di un servizio è difficile da localizzare perché non si conosce a priori quale metodo è stato effettivamente eseguito

Asserzioni ed ereditarietà

Le sottoclassi possono

- Aggiungere attributi e metodi, rispettando però il vincolo semantico
  $inv_{sottoclasse} \rightarrow inv_{classe}$
- Ridefinire i metodi, soddisfando però i seguenti vincoli semantici:
  $post_{sottoclasse} \rightarrow post_{classe}$
  $pre_{classe} \rightarrow pre_{sottoclasse}$

In questo modo le asserzioni impediscono che le operazioni ridefinite o aggiunte nella sottoclasse siano inconsistenti con quelle della classe padre

Tipi parametrici

Un tipo viene definito senza specificare tutti gli altri tipi che esso usa (questi ultimi sono i tipi parametrici), es. lista di elementi di tipo parametrico
È un tecnica di riuso statica come l'ereditarietà

Riuso black-box

Vantaggi

- Non spezza l'incapsulamento →
  - ✓ ogni classe resta focalizzata su un solo compito
  - ✓ ogni classe resta piccola
  - ✓ ogni gerarchia resta piccola
- Le dipendenze implementative sono meno numerose

↓

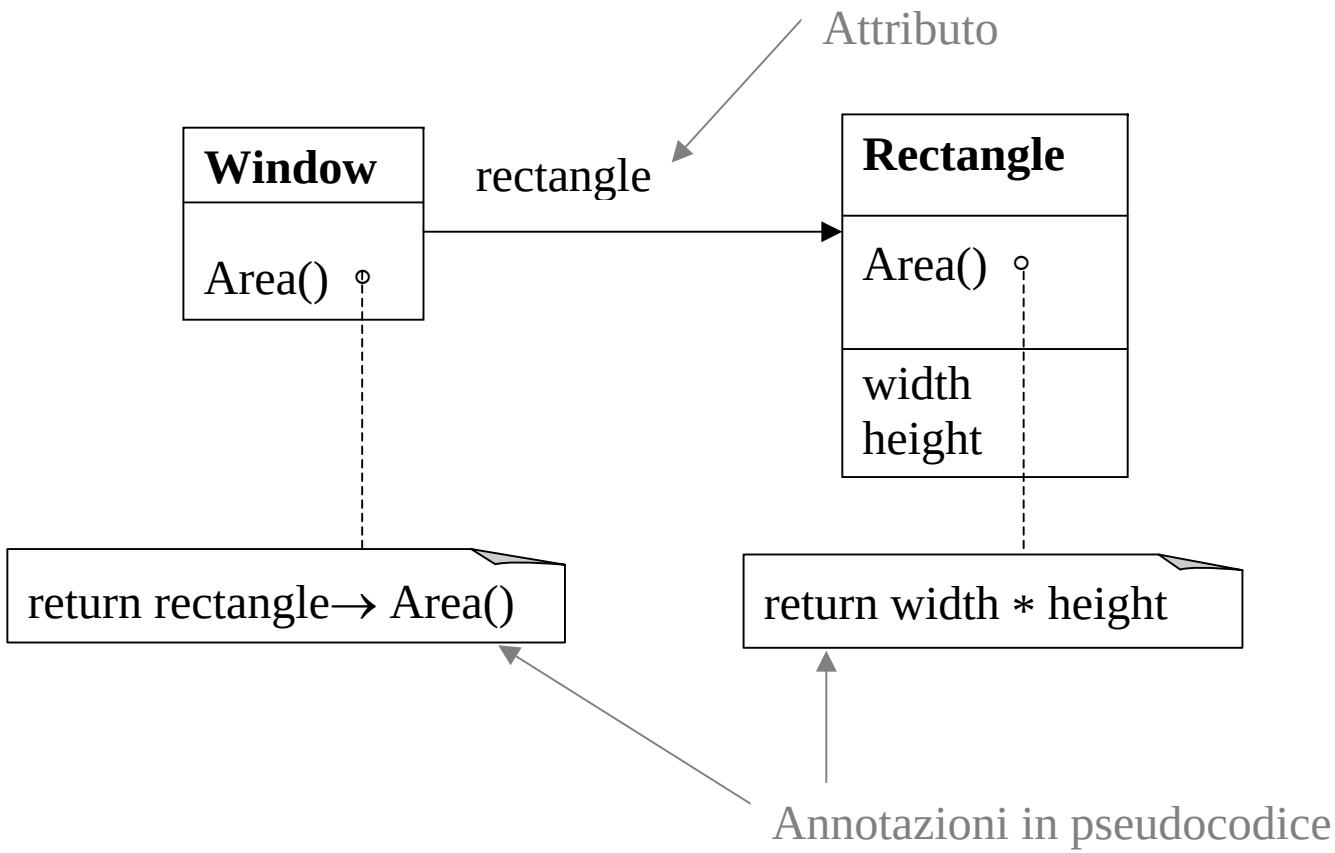
Secondo principio di progettazione OO: favorire la composizione di oggetti rispetto all'ereditarietà delle classi

Un progetto basato sulla composizione di oggetti avrà più oggetti (e meno classi) e il comportamento del sistema dipenderà dalle loro interazioni invece di essere definito in una sola classe

Ereditarietà e delega a confronto

Ereditarietà	Delega
Una richiesta inviata a un oggetto che è istanza di una sottoclasse deferisce la richiesta alla sua classe genitrice	Una richiesta viene gestita mediante due oggetti: l'oggetto ricevente la richiesta delega l'operazione a un suo oggetto delegato
Es. Window è una sottoclasse di Rectangle, cioè una Window è un Rectangle → Window riusa il comportamento delle operazioni ereditate da Rectangle	Es. Window ha una variabile che è istanza di Rectangle, cioè una Window ha un Rectangle → Window inoltra le richieste ricevute alla sua istanza di Rectangle, che risponde alle stesse mediante le operazioni di Rectangle
Un'operazione ereditata può riferirsi all'oggetto ricevente (variabile `this`)	Il ricevente passa se stesso al delegato per consentire alle operazioni delegate di riferirsi al ricevente

Delega


Attributo
Window
rectangle
Rectangle
Area()
Area()
width
height
return rectangle→ Area()
return width ∗ height
Annotazioni in pseudocodice

Delega (cont.)

- Vantaggi
  - ✓ Rende facile la composizione dei comportamenti durante l'esecuzione
  - ✓ Rende facile la modifica della composizione dei comportamenti durante l'esecuzione: il comportamento dell'oggetto ricevente cambia se cambia l'oggetto a cui esso delega la richiesta (ad es. la Window può divenire circolare durante l'esecuzione semplicemente sostituendo l'istanza di Rectangle con un'istanza di Circle, ammesso che Rectangle e Circle abbiano lo stesso tipo)

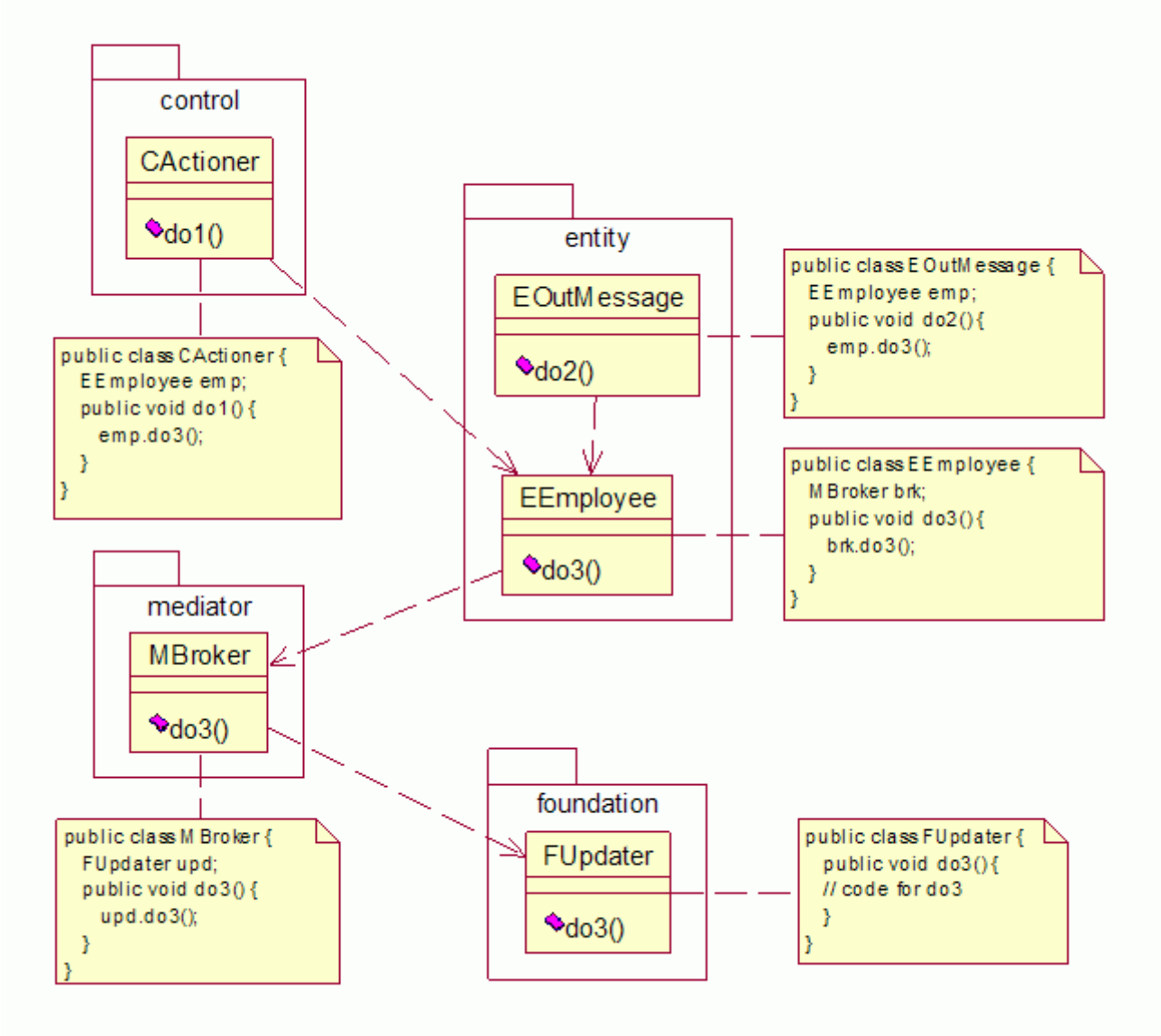
- Svantaggi
  - ✓ Il sw dinamico e altamente parametrizzato è più difficile da comprendere di quello statico
  - ✓ Inefficienze al momento dell'esecuzione (a causa dell'indirettezza)

Altri vantaggi della delega

- Rende esplicite le dipendenze che invece sono implicite se si sfrutta l'ereditarietà (e quindi sono difficili da riconoscere in base alla sola analisi del codice nonché da manutenere)
- Non interferisce con componenti preesistenti
- Conduce a codice più robusto
- È normalmente sfruttata per consentire a un oggetto client di ottenere un servizio da un oggetto di un layer distante, così da conservare una architettura verticale a layer (basata sulla comunicazione fra soli layer adiacenti), dove un layer è un package (che può raggruppare più package)

Attenzione: l'oggetto delegante non è sollevato dalle sue responsabilità contrattuali nei confronti degli oggetti client

Delega: un esempio (da Maciaszek-Liong)



Dipendenze fra classi e package

Le dipendenze fra metodi determinate da up- e down-call sono una sfida perché difficili e onerose da individuare analizzando il codice (in fase di manutenzione)

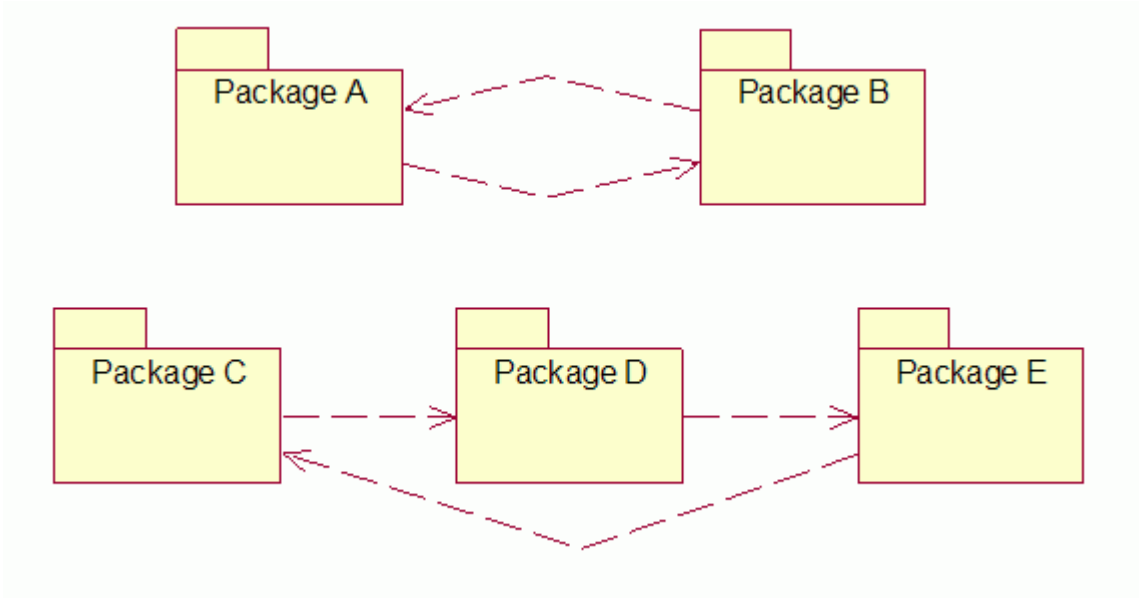
↓

È fortemente consigliato rendere tali dipendenze esplicite nel codice

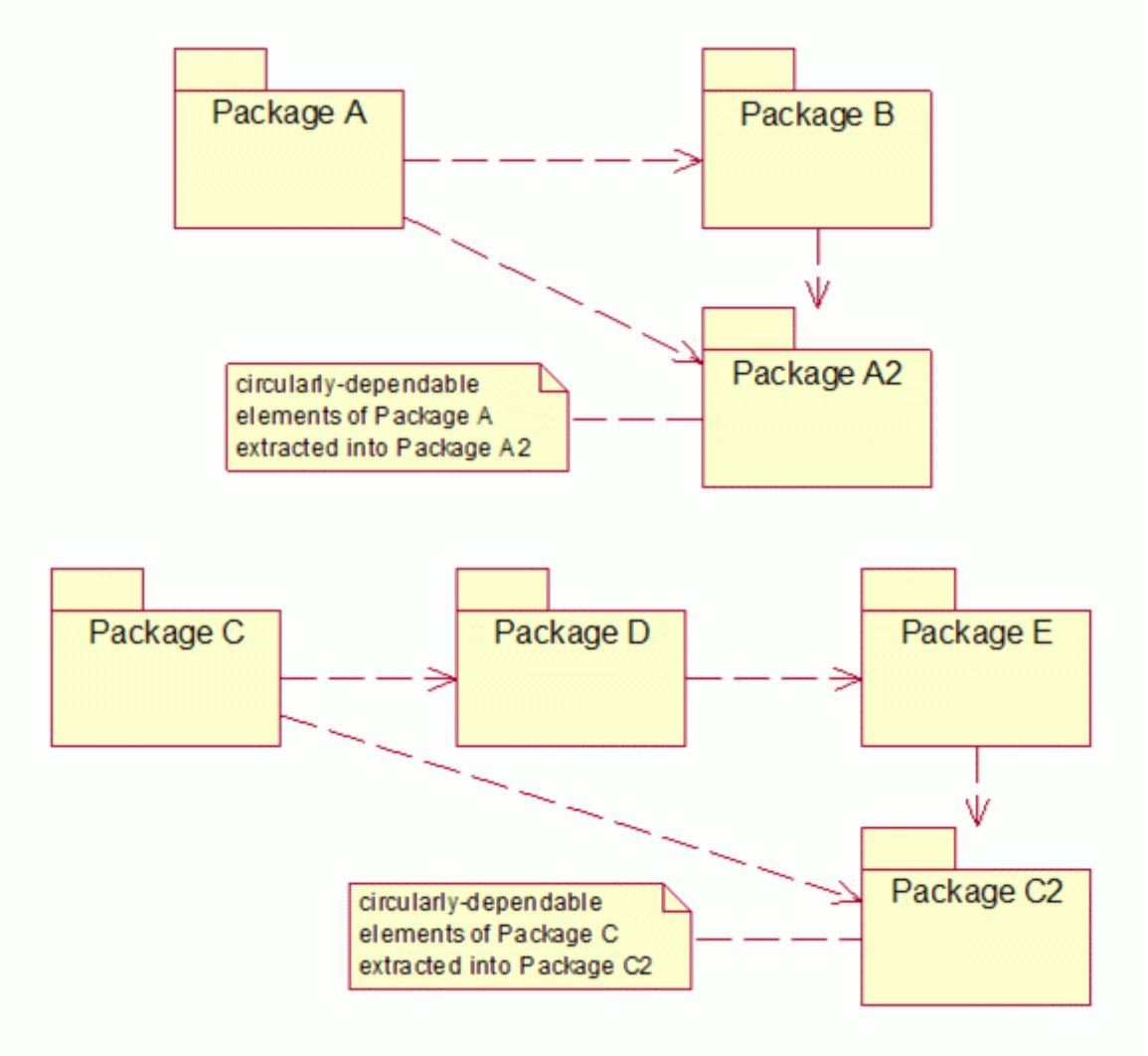
Tali dipendenze diventano dipendenze fra classi e fra package

L'uso di interface consente agli oggetti client delle stesse di rimanere inconsapevoli delle specifiche classi che le implementano (cioè di non dipendere da tali classi)

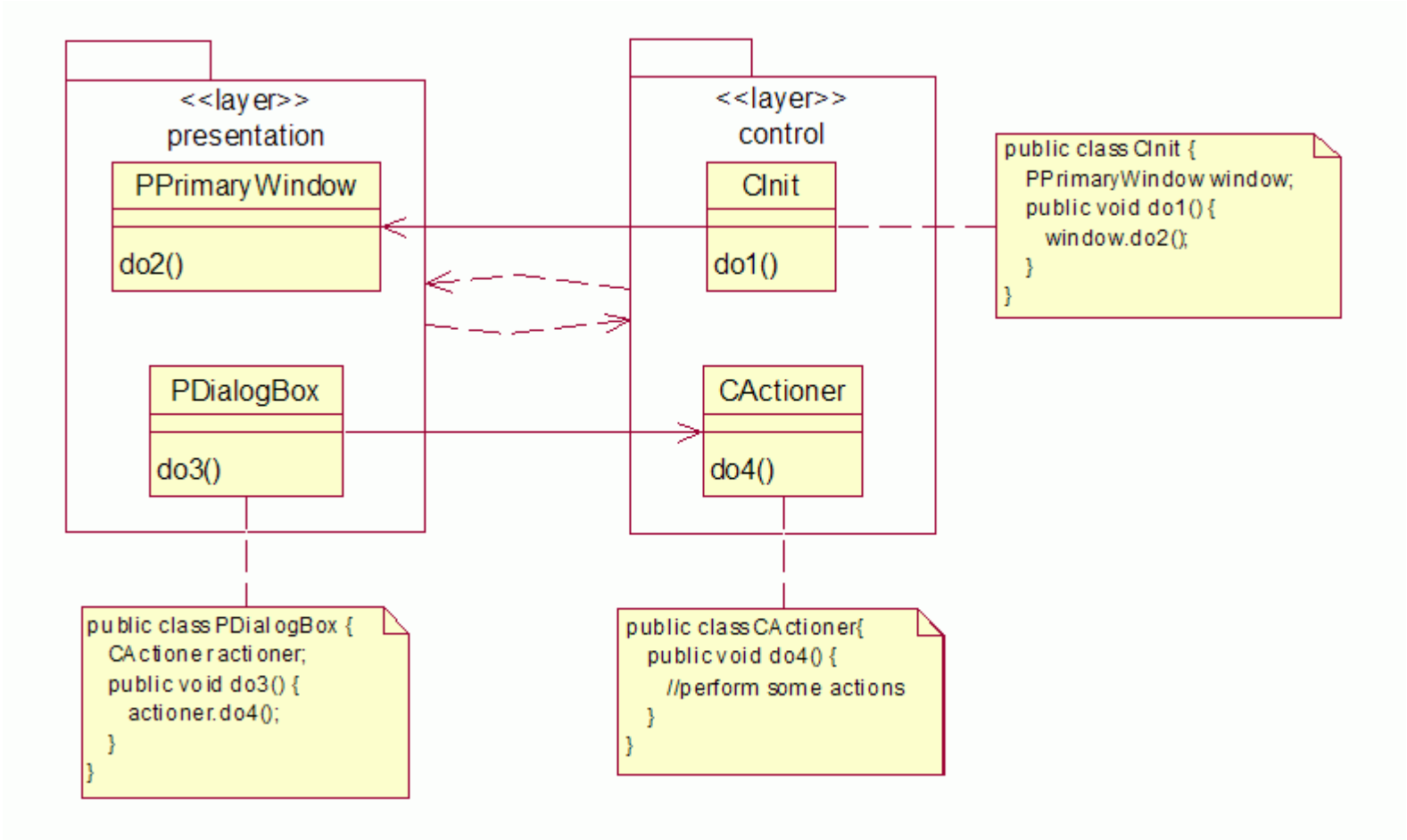
Dipendenze cicliche fra package



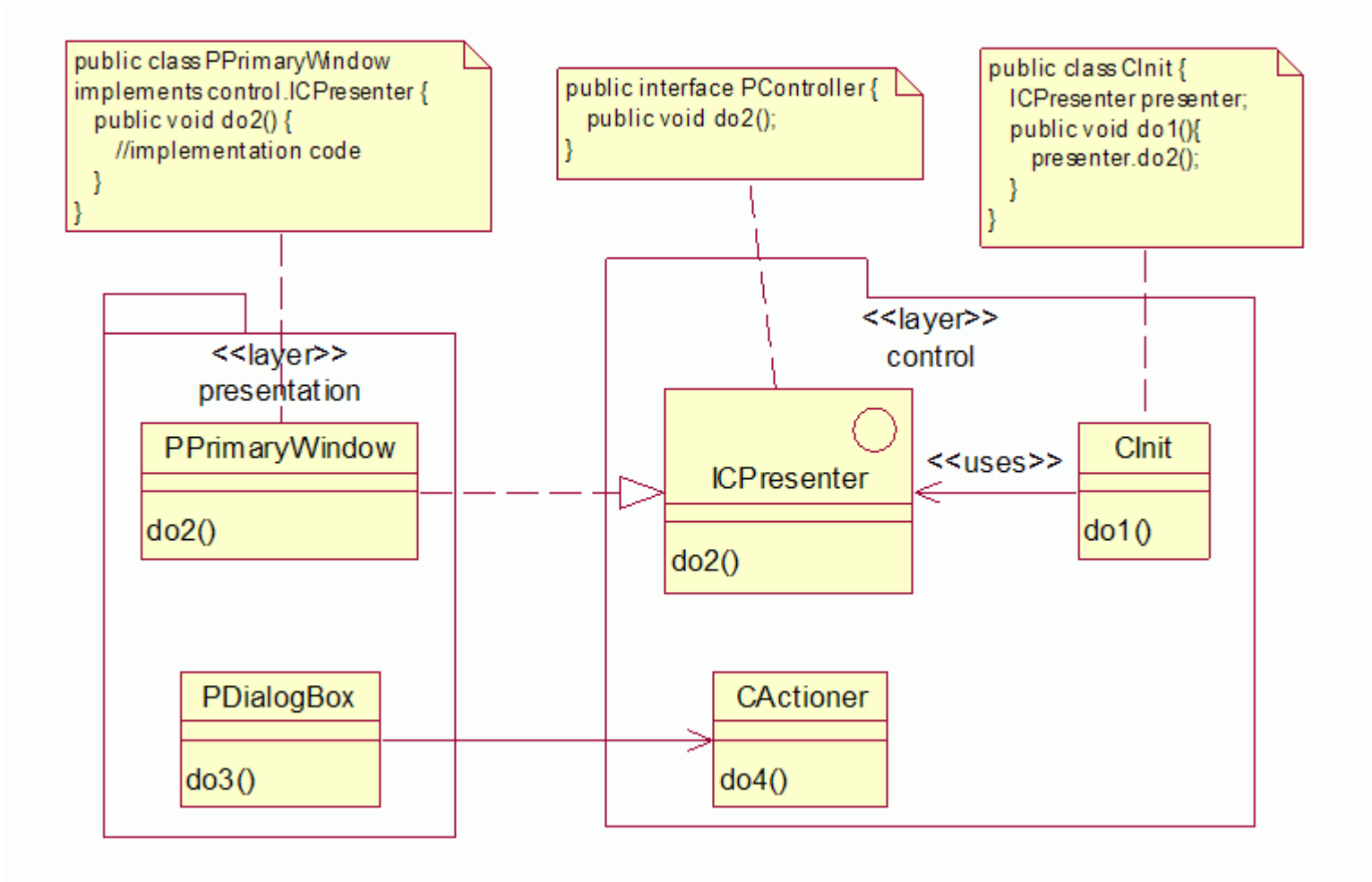
Eliminazione delle dipendenze cicliche fra package



Dipendenze cicliche fra package



Uso di interface per eliminare le dipendenze cicliche fra package



È spesso più desiderabile mettere un interface in un package che lo usa piuttosto che in uno che lo implementa

Questo stratagemma è il pattern Separated Interface di Fowler