Il 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

Il 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

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

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

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

Il paradigma di programmazione OO (cont.)

Tipi di un oggetto = totalità dei sottoinsiemi non vuoti della sua interfaccia (dove ogni sottoinsieme 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

Binding dinamico = associazione, al momento dell'esecuzione, di una richiesta inoltrata a un oggetto a uno dei metodi dello stesso

Polimorfismo (o sostituibilità) = possibilità di sostituire l'un l'altro durante l'esecuzione oggetti che hanno la stessa interfaccia

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

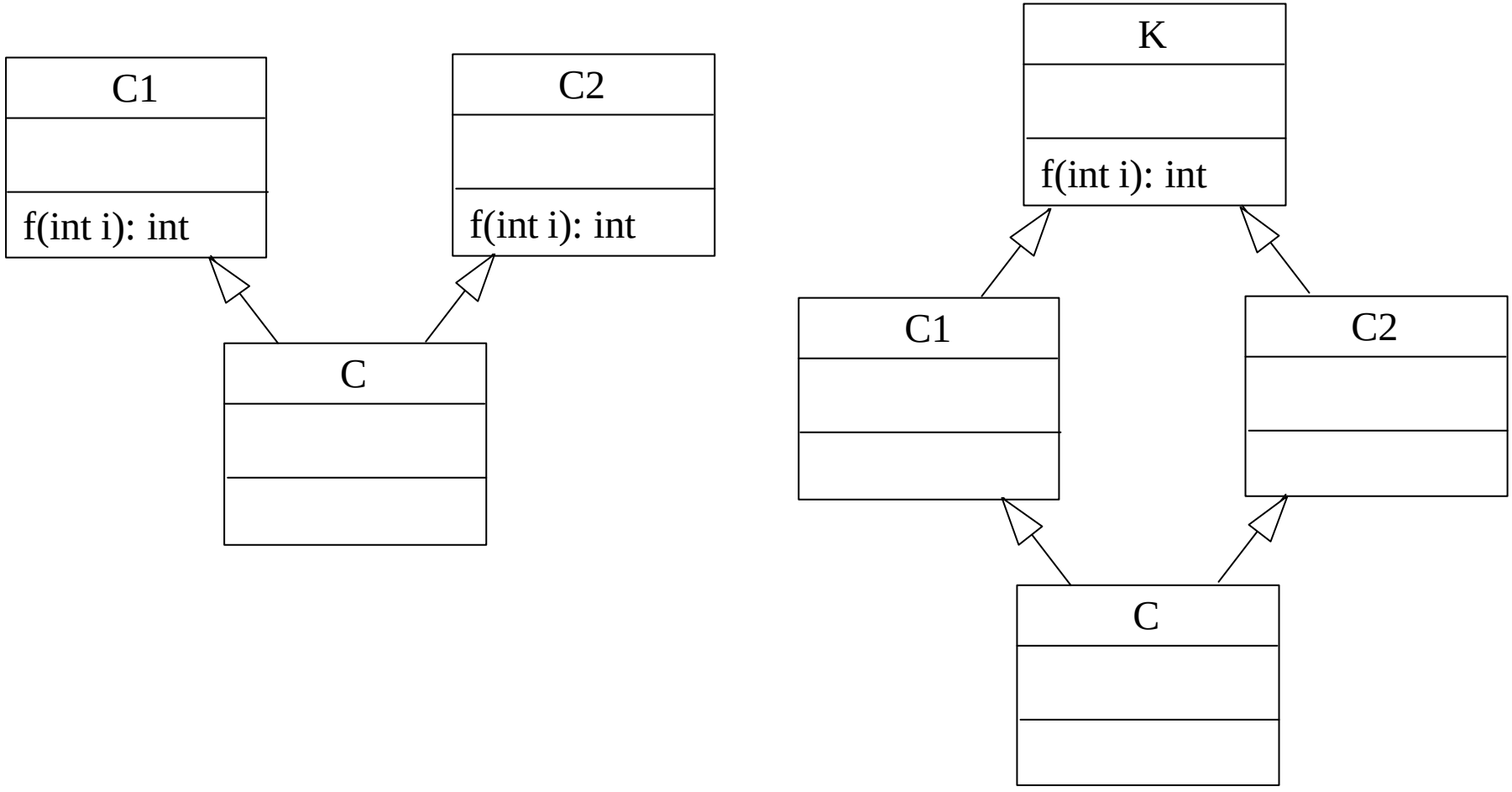
→ Primo principio di progettazione OO: programmare rivolti all’interfaccia, non all’implementazione

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

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)

Quali classi?

- Molte classi di un progetto possono provenire dal modello dell'analisi del dominio e del problema
- Molte altre classi non hanno alcun corrispettivo nel mondo reale
- Una modellazione rigida del mondo reale porta a un sistema che soddisfa le esigenze di oggi ma non necessariamente quelle di domani
- Le astrazioni che emergono durante un progetto sono cruciali per rendere flessibile e riusabile il sistema

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)

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

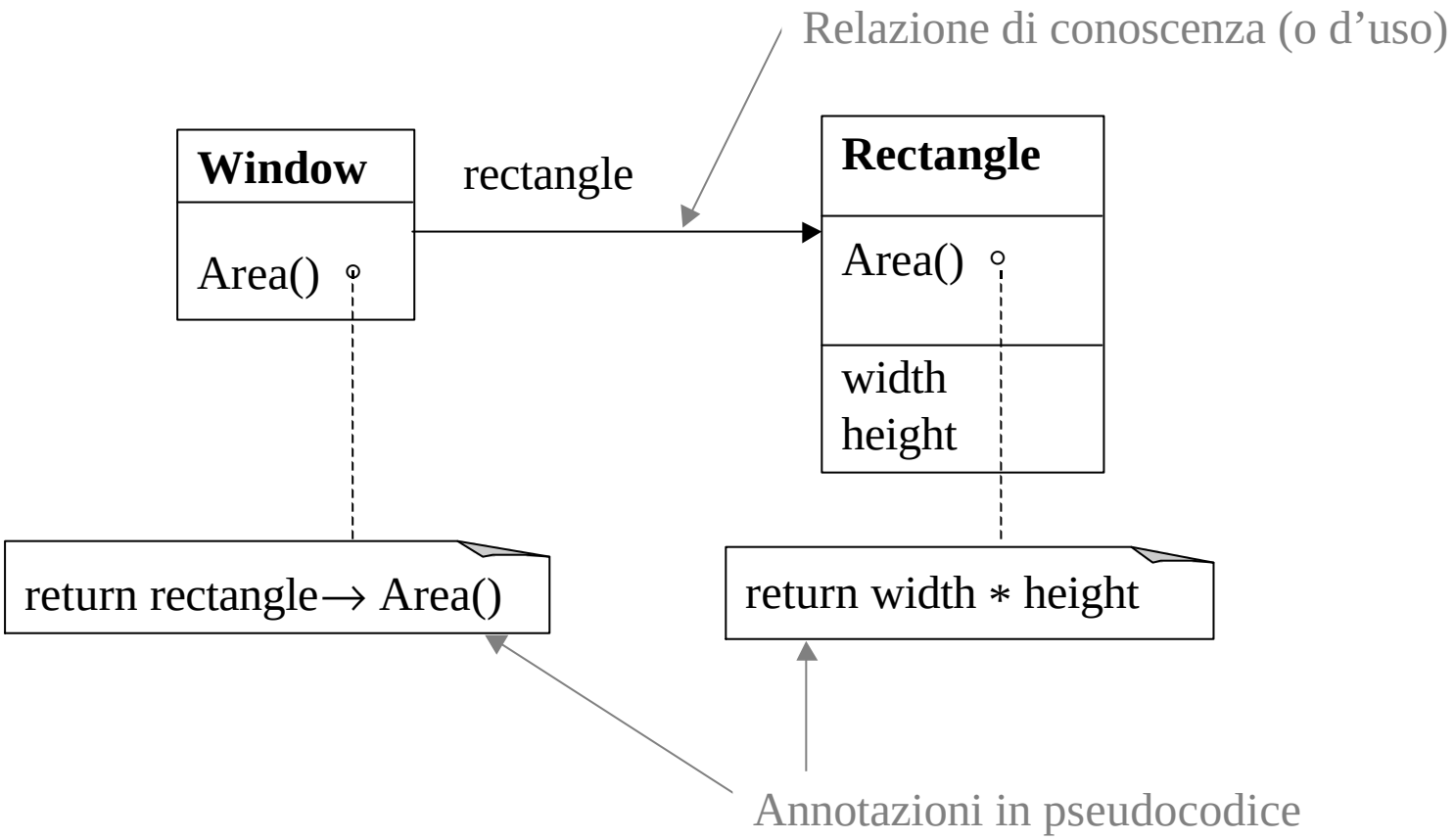
Tecniche di composizione di oggetti

- Tipi parametrici o generici (Ada, Eiffel) o template (C++)
- Delega : esempio estremo di composizione di oggetti finalizzato a rendere la stessa potente come l'ereditarietà ai fini del riuso → è sempre possibile sostituire l'ereditarietà con la composizione di oggetti

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


Relazione di conoscenza (o d'uso)
Window
rectangle
Area()
Rectangle
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)

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

Strutture statiche e dinamiche

Struttura statica = struttura del programma al momento della compilazione, classi collegate da gerarchie fisse

Struttura dinamica = struttura del programma al momento dell'esecuzione, reti (in costante evoluzione) di oggetti comunicanti

Le due strutture sono largamente indipendenti e non è possibile comprendere l'una data l'altra; in generale la struttura dinamica non è chiara guardando il codice

Principi di progettazione

Principio	Beneficio
Creare oggetti indirettamente, specificando non già il nome della classe ma quello di un'interfaccia	Si evita di legarsi a una particolare implementazione, semplificando così futuri cambiamenti
Non effettuare richieste specificando una particolare operazione	Si evita di legarsi a un modo particolare di soddisfare una richiesta, facilitando così la modifica sia statica, sia dinamica del modo in cui una richiesta è soddisfatta
Limitare le dipendenze (interfaccia verso sistema operativo e API) dalla piattaforma hw e sw	Aumento della portabilità del sw
Nascondere agli oggetti client i dettagli circa rappresentazione e implementazione degli oggetti server	Si evita che i cambiamenti dei server richiedano in cascata cambiamenti nei client

Principi di progettazione (cont.)

Principio	Beneficio
Isolare gli algoritmi che è probabile cambino (a causa di estensioni, ottimizzazioni o sostituzioni)	Si evita che gli oggetti che dipendono da un algoritmo cambino se l'algoritmo cambia
Accoppiamento lasco fra classi	Aumento della probabilità che una classe possa essere usata per conto suo e della comprensibilità, modificabilità ed estensibilità dell'intero sistema
Composizione di oggetti come alternativa alla creazione di sottoclassi	Evitare l'esplosione del numero delle classi (ad es. introduzione di una funzionalità personalizzata definendo una sola sottoclasse e componendo le sue istanze con quelle esistenti)