OO design pattern

Design pattern: motivazioni

- La progettazione OO è complessa
- Progettare sw OO riusabile ed evitare (o, almeno, limitare) la riprogettazione è ancor più complesso
- I progettisti esperti non risolvono ogni problema partendo da zero ma riusano soluzioni che hanno già funzionato
- L'uso dei pattern è teso a realizzare architetture più piccole, semplici e comprensibili
- È difficile trovare un sistema OO che non usi almeno due dei pattern più comuni e sistemi di dimensioni più elevate li usano quasi tutti

Design pattern: definizioni

- "Un pattern descrive un problema che ricorre nell'ambiente e l'essenza della soluzione del problema, in modo tale che si possa riusare questa soluzione milioni di volte senza mai ripeterla in maniera identica due volte" (Christopher Alexander, 1977, riferendosi a edifici e città)
- Progetto importante e ricorrente nei sistemi OO, la cui realizzazione è orientata al riuso, inteso ad aiutare il progettista a progettare bene più in fretta
- È come un template che può essere applicato in molte situazioni diverse
- È una collaborazione comune fra oggetti di un sistema

N.B. Nei pattern si assumono disponibili le caratteristiche dei due linguaggi di programmazione prescelti (C++ e Smalltalk)

Design pattern: elementi essenziali

Elemento	Significato
Nome	È la descrizione di livello di astrazione più elevato del pattern (la sua maniglia), utile per discutere del pattern e pensare allo stesso
Problema	Descrizione astratta del contesto, delle precondizioni di applicazione del pattern e del problema da esso risolto (che può essere rappresentato a diversi livelli)
Soluzione	Descrizione a) degli elementi del progetto (classi e oggetti), delle loro relazioni, responsabilità e collaborazioni b) delle decisioni, delle alternative considerate e dei trade-off che hanno condotto al progetto (queste informazioni sono necessarie per il riuso)
Conseguenze	• Spesso sono relative al trade-off spazio-tempo • Possono toccare questioni relative al linguaggio (C++ e Smalltalk) e all'implementazione, fornendo suggerimenti ed esempi • Comprendono l'impatto sulla flessibilità, estensibilità e portabilità del sistema • Sono cruciali nella valutazione delle alternative di progettazione

Design pattern: elementi descrittivi

Elemento	Significato
Nome e classificazione	Nome (vedi sopra) + duplice classificazione (vedi oltre)
Intento	Problema considerato
Noto anche come	Sinonimi
Motivazione	Scenario che esemplifica l'uso del pattern
Applicabilità	• Situazioni in cui il pattern può essere applicato • Esempi di progetti scadenti in cui il pattern può essere utile • Come riconoscere situazioni e progetti di cui sopra
Struttura	Rappresentazione grafica basata su OMT (Object Modeling Technique) e diagrammi di interazione
Partecipanti	Classi e oggetti e loro responsabilità
Collaborazioni	Come i partecipanti collaborano per affrontare le loro responsabilità

Design pattern: elementi descrittivi (cont.)

Elemento	Significato
Conseguenze	• In che misura il pattern soddisfa i suoi obiettivi • Trade-off e risultati nell'uso del pattern • Quali aspetti della struttura del sistema possono essere indipendentemente modificati
Implementazione	• Tranelli, suggerimenti e tecniche di cui tenere conto nella programmazione • Questioni specifiche al linguaggio
Codice di esempio	Frammenti di codice C++ o Smalltalk
Casi noti	Almeno due esempi di applicazione del pattern in sistemi reali che afferiscono a domini diversi
Pattern correlati (altri due meccanismi di classificazione dei pattern)	• Elenco dei pattern correlati al corrente e delle sostanziali differenze reciproche • Elenco dei pattern che dovrebbero essere usati insieme al corrente

Classificazione dei pattern

- In base al proposito (purpose)
  - ✓Teso alla creazione di oggetti (creational)
  - ✓Teso alla composizione di classi/oggetti (structural)
  - ✓Teso a caratterizzare l'interazione di classi/oggetti e la distribuzione di responsabilità (behavioral)

- In base alla applicabilità (scope)
  - ✓Focalizzato principalmente sulle classi e sulle loro sottoclassi, cioè sulle loro relazioni statiche (ovvero note al momento della compilazione) di ereditarietà (class)
  - ✓Focalizzato principalmente sugli oggetti e sulle loro relazioni dinamiche (mutevoli durante l'esecuzione) (object)

Classificazione dei pattern (cont.)

Creazione di oggetti

- nei creational class pattern è in parte demandata a sottoclassi
- nei creational object pattern è in parte demandata a un altro oggetto

Composizione

- di classi mediante l'ereditarietà negli structural class pattern
- di oggetti negli structural object pattern

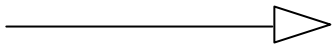
Comportamento

- che si esplica mediante l'ereditarietà (tipicamente algoritmi e flusso del controllo) nei behavioral class pattern
- che si esplica mediante la cooperazione fra oggetti (tipicamente per realizzare un compito che nessun oggetto può portare a termine da solo) nei behavioral object pattern

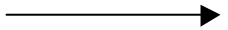
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapater (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapater (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Notazione OMT (Object Modeling Technique)

È molto simile al diagramma delle classi UML. Ad es., sono identici nei seguenti elementi:

- relazione di generalizzazione 
- nomi di classi astratte e metodi astratti (in italico)

Alcune piccole differenze:

- Nella scatola che rappresenta la classe, prima sono elencate le operazioni, poi i dati (in UML l'ordine è inverso)
- La relazione di dipendenza ha la punta triangolare piena 
  (in UML è biforcuta)

Vocabolario

In OMT i dati di una classe sono chiamati variabili di istanza (instance variables)

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

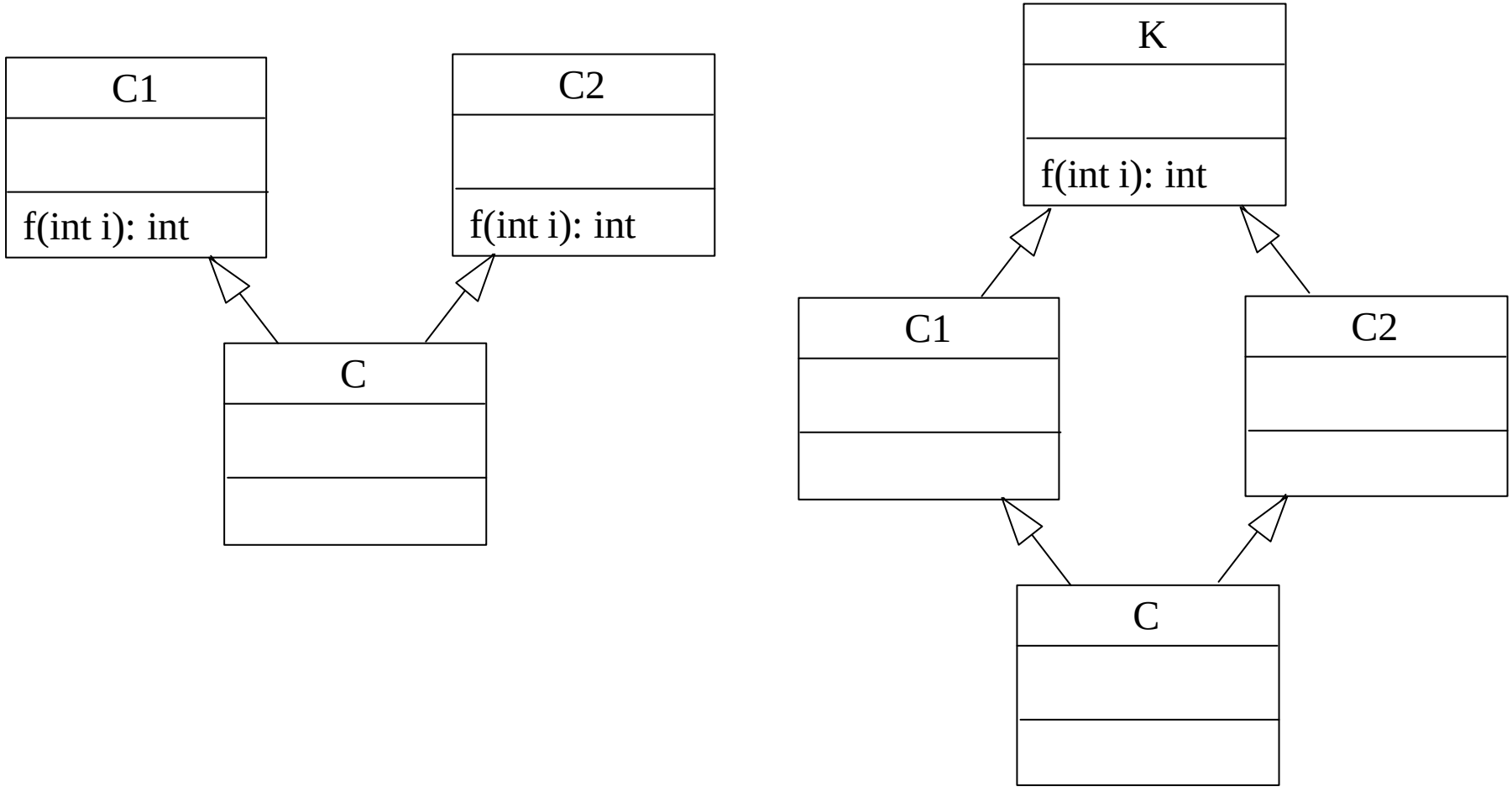
I creational pattern assicurano che il sistema sia scritto in termini di interfacce, non di implementazioni; essi cioè astraggono il processo di creazione di oggetti, offrendo più possibilità di associare un'interfaccia alla sua implementazione in modo trasparente all'implementazione

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)

La progettazione OO e i pattern

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
- Alcune astrazioni meno ovvie possono essere identificate con l'aiuto dei design pattern (ad es. Composite, Strategy e State)

La progettazione OO e i pattern (cont.)

Quali oggetti?

- La granularità degli oggetti (ovvero le loro dimensioni e il loro numero) può variare enormemente
- I design pattern aiutano a identificare la granularità più adatta nel progetto in corso (ad es. Facade, Flyweight, Factory, Builder, Visitor e Command)

Quali interfacce?

I design pattern

- aiutano a definire interfacce, identificandone gli elementi chiave e i tipi di dati da "spedire" attraverso un'interfaccia, così come a suggerire cosa non mettere in un'interfaccia (ad es. Memento)
- specificano relazioni fra interfacce (ad es. Decorator, Proxy e Visitor)

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

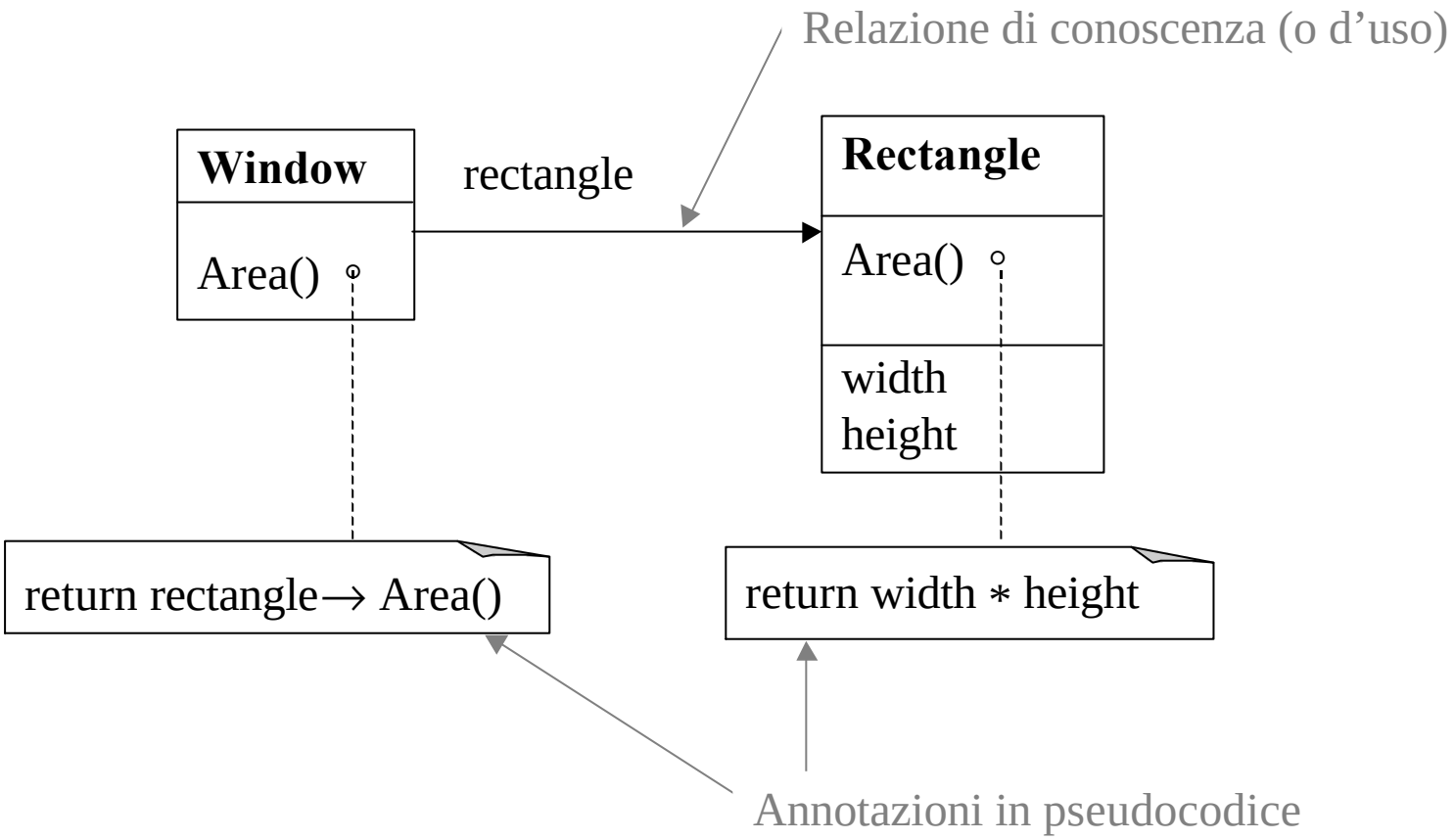
Tecniche di composizione di oggetti

- 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 (è usata pesantemente da State, Strategy e Visitor, meno pesantemente da Mediator, Chain of Resposibility e Bridge)

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 e non è supportata da alcun pattern

Strutture statiche e dinamiche

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

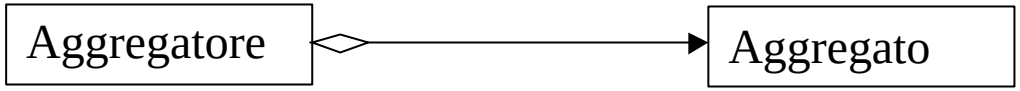
Struttura dinamica = struttura del programma al momento dell'esecuzione, reti (in costante evoluzione) di oggetti comunicanti

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 finché non si sono compresi pattern

Relazioni fra oggetti

Relazione OMT	Semantica	Implementa-zione	Rappresentazione grafica OMT
Aggregazione	Quella della composizione UML: un oggetto ne possiede o è parte di un altro → i due oggetti hanno la stessa esistenza (lifetime); natura statica	Mediante le variabili membro	Simile all'aggregazione in UML (la freccia ha la punta piena anziché biforcuta)
Conoscenza (acquaintance) o relazione d'uso	Quella della dipendenza UML: un oggetto sa dell'esistenza di un altro → relazione più debole dell'aggregazione, fra i due oggetti l'accoppiamento è minore: essi possono richiedersi l'un l'altro delle operazioni ma non sono responsabili l'uno dell'altro; natura dinamica	Mediante riferimenti e puntatori	Freccia con linea continua e punta piena

Aggregazione e conoscenza



Le due relazione sono semanticamente distinte ma implementate nello stesso modo se il linguaggio di programmazione non consente altrimenti (ad es. in Smalltak tutte le variabili sono riferimenti ad altri oggetti)

↓

La struttura dinamica del sistema deve essere imposta dal progettista, non dal linguaggio

La distinzione fra struttura statica e dinamica è catturata esplicitamente da Composite, e Decorator

Progetto per il cambiamento

La chiave per massimizzare il riuso è la capacità di anticipare nuovi requisiti e futuri cambiamenti dei requisiti esistenti, così da progettare il sistema in modo che possa evolvere di conseguenza, evitando grosse riprogettazioni a causa di modifiche inattese

I pattern assicurano che alcuni aspetti della struttura del sistema possano cambiare senza coinvolgerne altri → rendono il sistema robusto nei confronti di certi tipi di cambiamenti

Pattern e principi di progettazione

Principio	Beneficio	Pattern
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	Abstract Factory, Factory Method, Prototype
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	Chain of Resposibility, Command
Limitare le dipendenze (interfaccia verso sistema operativo e API) dalla piattaforma hw e sw	Aumento della portabilità del sw	Abstract Factory, Bridge

Pattern e principi di progettazione (cont.)

Principio	Beneficio	Pattern
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	Abstract Factory, Bridge, Memento, Proxy
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	Builder, Iterator, Strategy, Template Method, Visitor

Pattern e principi di progettazione (cont.)

Principio	Beneficio	Pattern
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	Abstract Factory, Bridge, Chain of Resposibility, Command, Facade, Mediator, Observer Tecniche usate: accoppiamento astratto e layering

Pattern e principi di progettazione (cont.)

Principio	Beneficio	Pattern
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)	Bridge, Chain of Resposibility, Composite, Decorator, Observer, Strategy
	Modificare semplicemente classi il cui codice sorgente non è disponibile o in cui un cambiamento richiederebbe la modifica di innumerevoli sottoclassi	Adapter, Decorator, Visitor

Ruolo dei pattern nello sviluppo

Consideriamo tre grandi categorie sw:

- Programmi applicativi
- Toolkit = librerie di classi predefinite general-purpose (es. classi per gestire liste, tabelle associative, pile, ecc., oppure per l'I/O)
- Framework = insieme di classi cooperanti che realizzano un progetto riutilizzabile e personalizzabile per un settore di sistemi sw (es. un framework per la costruzione di editor grafici può essere usato in domini diversi, come disegno artistico, composizione musicale e CAD meccanico; un altro framework può essere orientato alla costruzione di compilatori per vari linguaggi di programmazione e macchine destinatarie)

Sviluppo di programmi applicativi

Priorità	Contributo dei pattern
Riuso interno (in modo da non progettare né implementare più del dovuto)	• Riduzione delle dipendenze da classi, operazioni e algoritmi • Accoppiamento lasco degli oggetti cosicché sia probabile la loro collaborazione • Isolamento e incapsulamento delle operazioni
Manutenibilità	Riduzione delle dipendenze da piattaforme
Estensibilità	• Dimostrazione di come estendere le gerarchie di classi e come sfruttare la composizione di oggetti • Riduzione dell'accoppiamento

Toolkit

- I toolkit non impongono un progetto particolare alle applicazioni che li sfruttano
- Essi enfatizzano il riuso di codice (il programmatore scrive il corpo principale dell'applicazione e gli fa invocare il codice del toolkit)
- La loro progettazione è più difficile di quella dei programmi applicativi perché i toolkit, per essere utili, devono funzionare in molte applicazioni; i pattern aiutano a evitare assunzioni e dipendenze che possono limitare la flessibilità del toolkit e, conseguentemente, la sua applicabilità ed efficacia

Framework

- Un framework impone l'architettura dell'applicazione (partizionamento della struttura in classi e oggetti, responsabilità, collaborazioni, flusso di controllo), consentendo al progettista/implementatore di concentrarsi sugli aspetti specifici dell'applicazione
- La personalizzazione del framework avviene creando sottoclassi specifiche dell'applicazione le cui superclassi sono classi astratte del framework
- I framework, pur includendo anche classi concrete, enfatizzano il riuso del progetto, anziché quello del codice (il programmatore riusa il corpo principale dell'applicazione e scrive il codice che esso chiama)

Uso di framework

Vantaggi

- Costruzione più rapida di applicazioni
- Costruzione di applicazioni che hanno strutture simili → sono più facili da manutenere e più reciprocamente consistenti

Svantaggi

- Riduzione della libertà creativa del progettista perché molte decisioni di progetto sono già state prese
- Necessità di comprendere il framework prima di usarlo

Sviluppo di framework

- È il tipo di sw più difficile da progettare: l'architettura, per funzionare per tutte le applicazioni, deve essere il più flessibile ed estensibile possibile
- Le applicazioni devono continuare a funzionare, possibilmente senza avere bisogno di essere cambiate, anche se il framework evolve
- I framework maturi di solito incorporano più pattern e presentano un beneficio aggiuntivo se sono documentati da tali pattern

Pattern e framework: differenze

- I pattern sono più astratti: i framework sono scritti in linguaggio di programmazione e possono essere incorporati direttamente nel codice, i pattern no (solo esempi di pattern fanno parte del codice)
- I pattern contengono descrizioni dei lori intenti, trade-off e conseguenze, i framework no
- I pattern sono elementi architetturali più piccoli dei framework: un framework tipico contiene più pattern ma non viceversa
- I pattern sono meno specializzati dei framework: ogni framework è rivolto a un particolare tipo di applicazioni (cioè ne fissa l'architettura), un pattern invece può essere usato in quasi ogni tipo di applicazione (perché non ne fissa l'architettura)

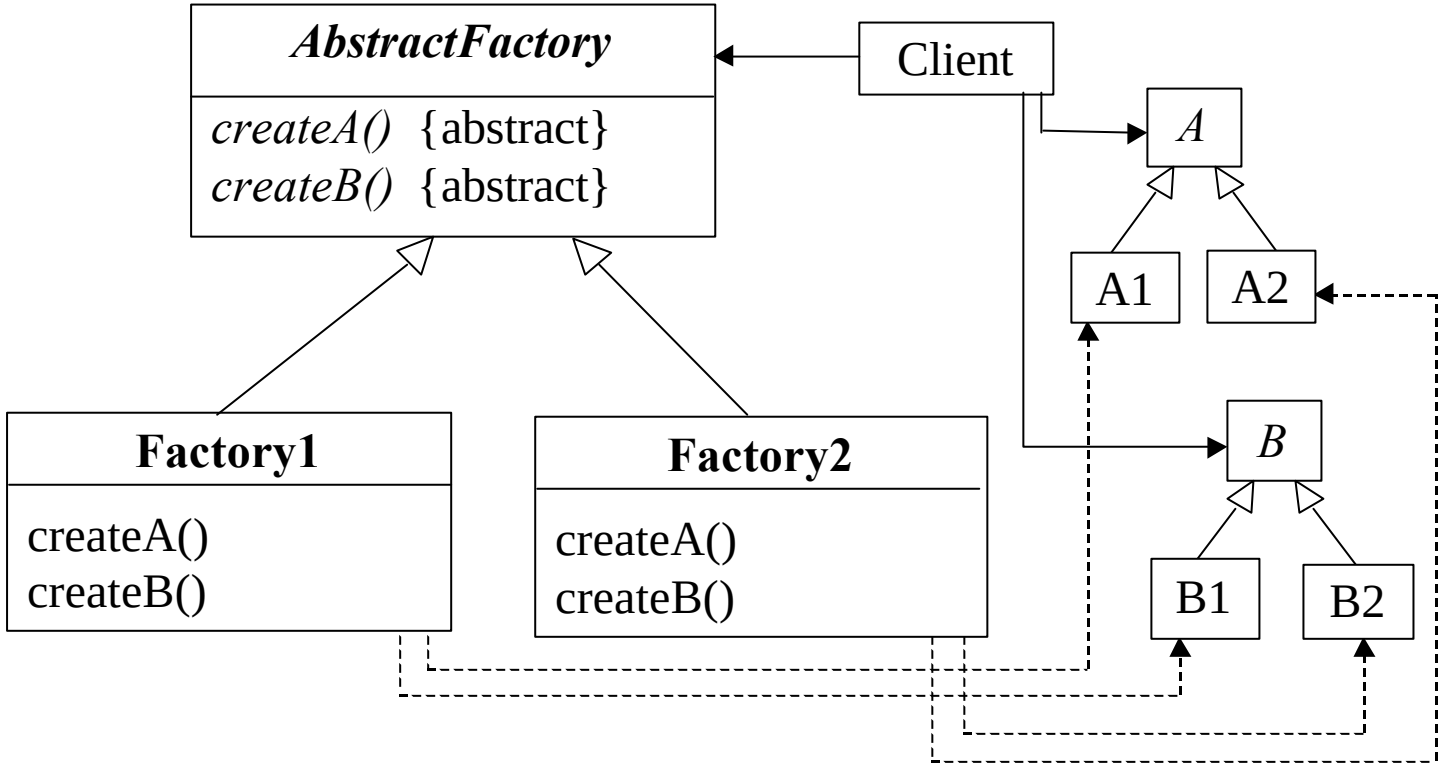
I framework sono il modo per massimizzare il riuso nei sistemi OO $\rightarrow$ le applicazioni OO di grandi dimensioni finiranno per essere costituite da livelli di framework cooperanti

Suggerimenti per l’uso dei pattern

Trasformare i nomi (astratti) degli elementi del pattern in nomi significativi per l’applicazione che si sta sviluppando che però incorporino (anche parzialmente) i nomi originali, in modo da esplicitare l’uso del pattern

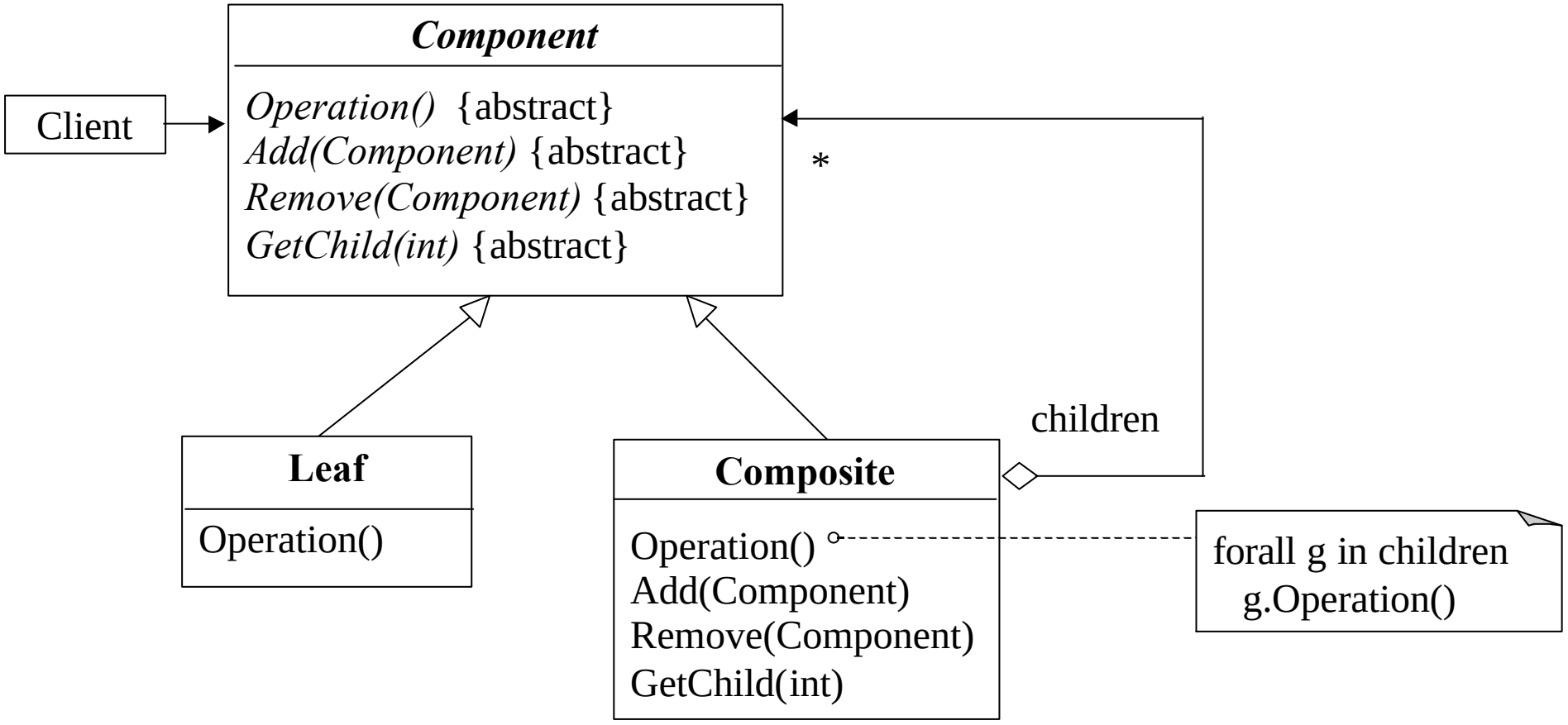
ABSTRACT FACTORY (Object Creational)

Intento: fornire un'interfaccia (la classe astratta Abstract Factory) per la creazione di famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete



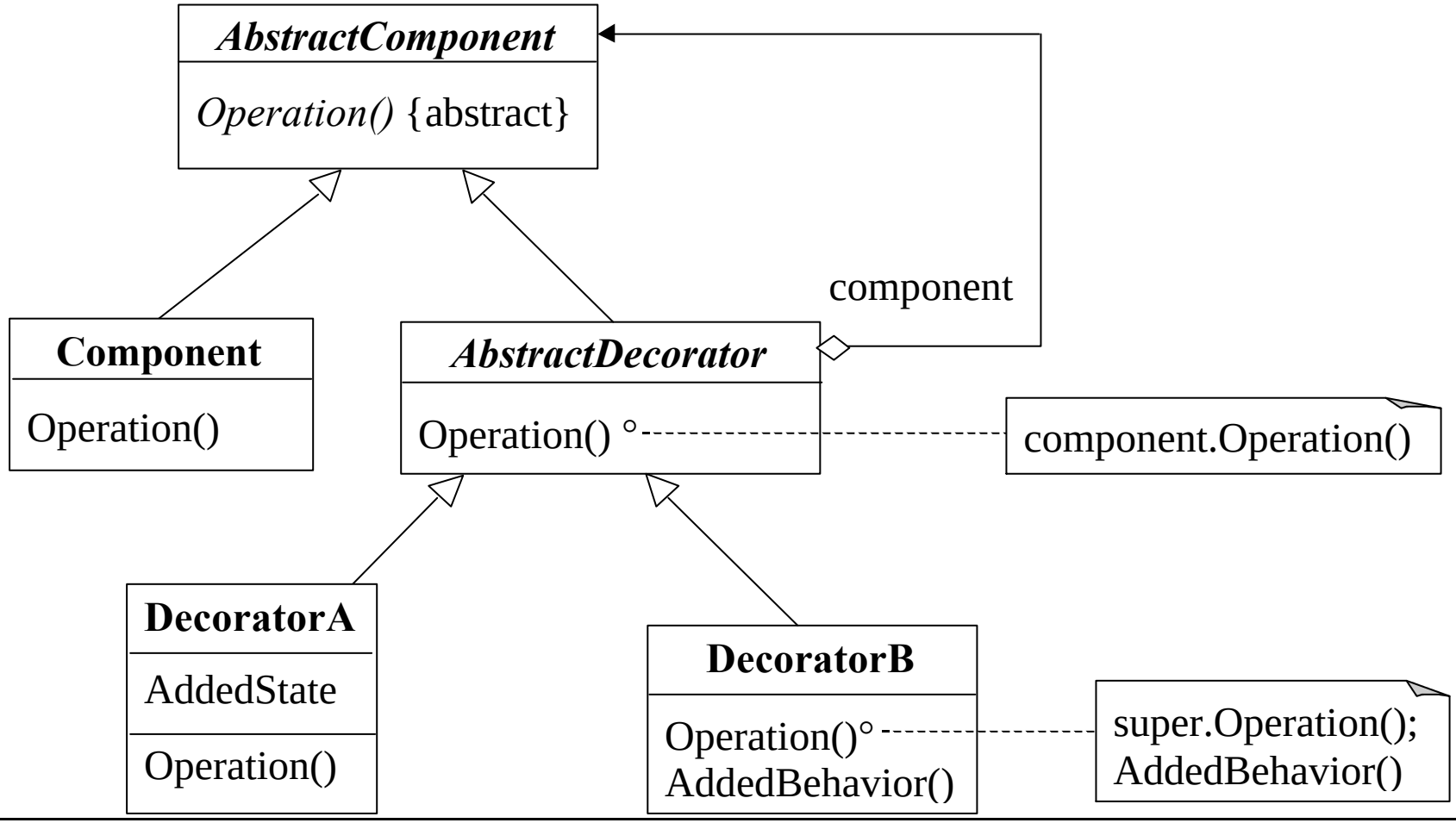
COMPOSITE (Object Structural)

Intento: comporre oggetti in strutture ad albero che rappresentino la gerarchia parte-tutto. Consente ai clienti di trattare oggetti individuali e compositi in maniera uniforme



DECORATOR (Object Structural)

Intento: attribuire dinamicamente responsabilità aggiuntive a un oggetto della classe Component (non all'intera classe Component) senza creare sottoclassi



OBSERVER (Object Behavioral)

Intento: definire una dipendenza uno-a-molti tra oggetti in modo che, quando un oggetto (di classe Subject) cambia stato, tutti quelli da esso dipendenti (di classe Observer) ne ricevano notifica e vengano aggiornati automaticamente (Noto anche come “pubblicazione e iscrizione”: gli observer possono iscriversi per ricevere le notifiche pubblicate dal subject)