

Object Orientation

Gabriele Zarcone

27 marzo 2018

- Concetti dell'Object Orientation
- Programmazione Modulare
 - Principi Programmazione Modulare
 - Relazioni fra moduli
 - Tipi di relazione
 - Astrazione e encapsulation
 - Tipi di Moduli
- Object Orientation
 - Classi
- UML
 - Metamodello
 - Diagrammi UML
 - Diagrama delle classi
- GENERALIZZAZIONE
 - IN UML
 - Overriding
 - Classi Astratte
 - Protezione
 - Protected
 - Package
 - Formattazione UML
 - Corsivo
 - Sottolineato
- POLIMORFISMO
- COLLEGAMENTO DINAMICO
- Come uso i concetti dell'OO
 - Principio di sostituzione di Liskov
 - Design by contract
- Associazioni
 - Navigabilità associazioni

- Associazioni e attributi
 - Errore
 - Prossima Lezione. >

Concetti dell'Object Orientation

l'OO non è un linguaggio ma un modo di pensare e di interfacciarsi ai metodi risolutivi

Si può programmare ad oggetti anche in C o assembly. Ovvio che se il linguaggio ha gli stessi costrutti e astrazioni dell'OO è più facile, ma non è impossibile con altri linguaggi

Per capire il problema lo penso OO e poi però posso implementarlo come voglio, anche in C

OO non risolve il problema, mi dà un modo per risolverlo, degli strumenti

Ha pochissimi concetti, anche banali

- classi e ereditarietà
- polimorfismo
- collegamento dinamico

non sempre sono facili da usare però, e spesso si fanno dei grandi errori. Soprattutto perché sono concetti semplici.

Per esempio finisce che ho 50 classi quando me ne servono solo 4

E' uno dei metodi per risolvere ma non è l'unico e non sempre è il migliore (progr. funzionale, aspect orientation)

Programmazione Modulare

è il padre della OO

es: MODULO A2

la programmazione modulare ha una serie di ragionamenti su quello che deve fare un programma che è stato ereditato pari pari dall'OO. OO aggiunge cose nuove ma moltissime le prende dalla progr modulare

Principi Programmazione Modulare

- astrazione: creare una scatola e darle un nome senza
- **encapsulation e information hiding**: posso anche renderla opaca la scatola e posso usarla

senza guardarci dentro

- **enca**: chiudo a chiave la scatola e non posso usare quello che c'è dentro
- **tipi di modulo**: ci sono diversi tipi di moduli

Relazioni fra moduli

matematicamente: dato l'insieme S di moduli allora una relazione è un sottoinsieme di $S \times S$

cioè un insieme di coppie ordinate

sono rappresentabili graficamente come un **grafo orientato**

Tipi di relazione

- **usa**: il modulo A ha bisogno di quello che ha il modulo B \rightarrow A usa B
- **is_component_of**: B è composto da a_1 a_2 a_3 , vuol dire che se li metto insieme ottengo B. Mettendo insieme delle classi ottengo una generalizzazione più ampia a cui posso dare un nome

attenzione alle relazioni cicliche. Non so più da dove partire perchè uno dipende dall'altro

Astrazione e encapsulation

Tipi di Moduli

storicamente si sono create diverse tipologie di moduli

- **PROCEDURALE** singole procedure senza dati
- **DATI** sono solo dati: tabella che salva uno stato
- **ABSTRACT OBJECT** ho solo l'oggetto astratto che mette insieme procedure e dati. E' solo l'istanza, esiste solo quell'oggetto. Mi fornisce dei dati più delle funzioni che mi servono per lavorare su quei dati e posso interagirci con lo stesso livello d'astrazione dell'oggetto
- **ABSTRACT DATA OBJECT** potrei voler più oggetti tutti uguali tra di loro. Mi serve un progetto di come è fatto un oggetto (classe) da cui posso generare tutte le istanze uguali che voglio.
- **GENERICI** sono definizioni parziali di un qualcosa perchè parametriche rispetto ad un altro tipo di cui ti devo ancora parlare (interfacce). Devo stare attento quando li uso ma sono molto utili per semplificare la compilazione e ridurre gli errori

Object Orientation

si basa sui Abstract data type che chiama CLASSI

definisce nuove relazioni:

- **GENERALIZZAZIONI** (*ereditarietà*) l'ereditarietà però è una cosa del linguaggio, è un modo per supportare la generalizzazione, ma non è l'unico, posso creare una generalizzazione anche senza usare il meccanismo della ereditarietà
- **AGGREGAZIONE** equivale a *is_part_of* non è composto ma è una parte, non lo completa tutto

nuovi concetti (non possono esserci senza avere generalizzazione):

- **POLIMORFISMO**
- **COLLEGAMENTO DINAMICO**

non sono legati uno all'altro, può esserci polimorfismo anche senza coll din e viceversa

CLassi

permettono di definire un insieme di oggetti che condividono:

- Comportamenti (*metodi*)
- Conoscenze (*attributi*)

sono il progetto di costruzione di un oggetto

possono anche essere incomplete (*classe astratta*)

permettono di separare la definizione dell'interfaccia e della definizione vera e propria. Separano ciò che interagisce con l'esterno e nascondono in funzionamento (information hiding)

UML

è un linguaggio di *progettazione*

è l'unione di tre diverse idee di 3 informatici i tre linguaggi però non erano completamente compatibili e spesso non si sono messi d'accordo su alcune cose. Quindi le hanno prese tutte e tre e fanno scegliere all'utente.

La prima versione era un gazzabuglio di molti concetti presi da concetti differenti che si uniscono tutti insieme.

in UML 2.0 si è iniziato a formalizzare i costrutti

Metamodello

posso definire 4 livelli

livello più basso: quello più vicino a noi livello 2: progettazione livello 1: come sono fatti i costrutti
sottostanti livello 0: c'è solo la classe che generalizza tutto il resto

Diagrammi UML

sono le foglie dell'albero in figura

- diagramma delle classi
- degli stati
- di sequenza

gli altri sono comunque importanti ma questi sono i **fondamentali**.

Diagrama delle classi

una classe è un rettangolo diviso in 3 zone

- nome classe
- attributi
- metodi

ci sono alcuni simboli per indicare i gradi di protezione:

- + pubblico
- - privato
- ~ package
- # protected

GENERALIZZAZIONE

è una relazione tra classi

ci permette di definire una classe come una specializzazione di un'altra classe. Non riparto a definirla da zero

E' la stessa classe con in più altre cose: è un write once. Permette quindi il riuso del codice: posso definire più classi da quella padre.

eredita tutto quello che ha la classe padre

Una classe aggiunge attributi e metodi dalla classe padre

Può anche cambiare come svolge un compito ereditato dalla classe padre


non dovrebbe mai però eliminare degli attributi o dei metodi

Può essere singola o multipla

- ho 1 padre
- ho N padri

Java supporta solo l'eredità singola. L'eredità multipla però è comoda -> permette implementare più di un'interfaccia per la stessa classe.

IN UML

 è una freccia a linea continua con punta piena bianca

classe dove c'è la punta della freccia è il padre, l'altro è il figlio da cui eredita tutto

Overriding

lo indico riscrivendo il nome della classe nella sottoclasse, se non lo riscrivo vuol dire che anche la sottoclasse ce l'ha perché lo eredita ma non lo ha ridefinito in maniera diversa. anche Non posso ridefinire degli attributi, l'unica cosa che posso fare è cambiare il tipo dell'attributo. Se lo faccio cambia proprio l'attributo

Classi Astratte

Metodo in *corsivo*: è un metodo ASTRATTO, cioè qualcosa che è definito ma non ancora implementato

se c'è almeno un metodo astratto in una classe allora non è completa. Quindi anche lei è astratta. Anche il suo nome va in corsivo

Posso avere classi astratte che non abbiano metodi astratti. Può sembrare completa ma magari io la voglio indicare come astratta e non completa.

Protezione

Protected

le classi che ereditano da un'altra classe sono privilegiate. Possono avere un accesso diverso da quello di tutto il mondo rispetto alla classe padre.

E' il livello di protezione **protected** (#)

Package

solo le classi dello stesso package possono vedere i segreti è quello di default (quando non scrivo nulla).

in UML: ~

Formattazione UML

Corsivo

- per le classi e i metodi **astratti**

Sottolineato

- per gli attributi e metodi **statici**

POLIMORFISMO

permette di usare un oggetto A (appartente alla classe CA) ovunque possa essere usato un oggetto appartenente alla classe da cui CA eredita

Se CA eredita da CB allora istanze della classe CA possono essere usate dove possono esserci istanze di CB

i figli possono presentarsi al posto dei padri ma non viceversa

non è solo riuso del codice

è come se eredita `_da ==` è un sottotipo `_di`

la classe che eredita è un sottotipo del padre (animali -> gatto)

COLLEGAMENTO DINAMICO

è possibile fare overriding quindi vorrei che l'implementazione del metodo eseguita fosse quella ridefinita nel figlio se ho un istanza del figlio

Se al posto del padre trovo un figlio e chiamo un metodo del padre ridefinito dal figlio vorrei che venga richiamata l'implementazione nuova del figlio

Devo aspettare il runtime -> è dinamico (non statico)

Chiama l'implementazione più vicina alla classe realmente presente lì

Permette di chiamare del codice che non è ancora stato scritto. E' molto flessibile è mantenibile infatti

Come uso i concetti dell'OO

Principio di sostituzione di Liskov

prima regola dell'OO è quella di non cancellare i metodi. I figli devono saper fare tutto quello che sa fare il padre, niente di meno; poi possono fare anche cose in più. Non posso togliere conoscenze.

Sia $q(x)$ una proprietà provabile da oggetti di tipo T . Allora $q(y)$ deve essere vero per oggetti di tipo y di tipo S se S è sottotipo di T .

i sottotipi devono soddisfare tutte le proprietà che rispettano i padri, altrimenti non sono sottotipi. Se non fosse così non potrei fare collegamneto dinamico: non potrei mettere il figlio al poisto del padre

Design by contract

Meier ragione su pre e post condizini di un metodo. Faccio un contratto con il metodo. Se tu mi invochi allora mi prometti che ci sono delle determinate precondizioni, s ele precondizioni sono vere il metodo mi assicura che fa tutto quello che promette. Se non sono vere le precondizione allora il metodo può fare ciò che vuole.

Se il metodo lo eredito da una classe figlia e lo ridefinisco allora la mia ridefinizione deve funzionare ancora quando funzionava l aprecedente definizione:

- precondizioni devono essere uguali o più ampie

le post condizioni devono fare la stessa cosa di prima o più precisa

- postcondizione uguali o più strette

Eifel (linguaggio di Meier) controllava con la sintassi se veniva rispettata questa cosa e quindi se rispetta il principio della Liskov.

Associazioni

LA seconda delle relazioni che vediamo; ci sono diverse relazioni (generalizzazione, associazioni...)

è una relazione tra istanze di classi, non tra classi -> != generalizzazione

un'istanza deve riconoscere e ricordare un'altra istanza. Studenti sono in relazione con bellettini, non con tutto il genere dei professori. Siamo associati con un'istanza e non con una classe

Implica una relazione usa. Devo tenere traccia di un'altra istanza perché devo usare le sue funzioni

Quando due oggetti interagiscono vuol dire che c'è una relazione

Un'associazione è la più semplice tra tutte **In UML:**

la indico come una linea semplice, con:

- **label:** nome della relazione (Es: segue_corso_di)
- **ruolo:** è utile quando sono associazioni tra due classi uguali. Lo useremo per dare dei nomi agli attributi
- **multiplicità:** indica quante istanze di una sono in relazione con quante istanze dell'altra e viceversa. La molteplicità lontana alla classe si riferisce a quella classe lontana.

Navigabilità associazioni

delle associazioni non sempre sono a doppio senso. Spesso una istanza deve conoscere l'istanza dell'altra ma può non essere necessario il viceversa.

ES: _ il professore non sa le istanze singole dei suoi studenti.

Quando progetto devo decidere quali istanze devo memorizzare

la linea senza nulla è una associazione bidirezionale

se ho una freccia non piena (->) invece le istanze sono in associazione ma quella che punta deve conoscere e ricordare quella puntata, ma quella puntata non deve ricordare chi la punta anche se è in relazione con lei.

Associazioni e attributi

spesso l'associazione si fa attraverso la definizione di un attributo, ma non è l'unico modo.

Non mi costringe ad usare un attributo. Posso anche farlo in maniera diversa.

VICEVERSA non è vero che ogni volta che ho un attributo in cui ho un'associazione

Con gli attributi posso fare anche aggregazione e composizione che sono casi particolari di associazioni.

Non è per forza necessario separare associazione da aggregazione, composizione ecc.. potrei all'inizio mettere solo associazioni e poi in secondo luogo precisare che tipo di associazione.

Errore

non va segnato in UML sia l'attributo che l'associazione perchè sto ripetendo due volte la stessa cosa. Se metto solo l'attributo o solo l'associazione ottengo lo stesso effetto. Se metto solo la freccia della associazione posso fare quello che voglio per realizzarla, se metto solo l'attributo obbligo a creare un associazione con un attributo.

Prossima Lezione. >
