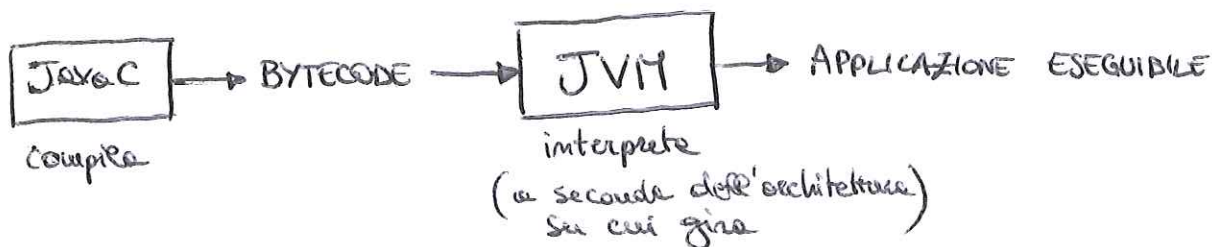


JAVA

E' un linguaggio:

- **SEMPLICE E FAMILIARE**: sintassi vicina al C e al C++ (in voga a quel tempo: '83-'95)
- **ORIENTATO AGLI OGGETTI**: ogni cosa è un OGGETTO, eccetto i tipi di dato primitivi.
Supporta i 4 fondamentali dei linguaggi O.O.:
 - INCAPSULAMENTO
 - EREDITARIETA'
 - BINDING DINAMICO
 - POLIMORFISMO
- **ROBUSTO E SICURO**: il software si comporta "bene" (errori non erroti, non va in crash o in deadlock) in condizioni di lavoro non previste (carico di lavoro eccessivo, input difformi alle attese, ...) → ROBUSTO
E' sicuro anche perché è STRONG TYPED, limiti sul casting dei tipi di dato e NON permette operazioni con i PUNTATORI (NO ALGEBRA DI PUNTATORI)
La gestione della memoria non è delegata al compilatore, ma rinviata a RUN-TIME.
Le classi devono avere tutte nomi diversi, inoltre c'è information hiding → SICURO
- **ARCHITETTURA NEUTRALE**: Le applicazioni sono agnostiche della piattaforma HW o SW. Java adotta un codice binario INDIPENDENTE dalla piattaforma HW e dal Sistema Operativo.
Il compilatore Java produce BYTECODE indipendente dalla piattaforma, il quale viene INTERPRETATO su ogni specifica architettura.
→ COMPILATO + INTERPRETATO



- **PORTABILE**: garantisce gli stessi risultati indipendentemente dalla piattaforma hardware (a differenza di C e C++).
- **AD ALTE PRESTAZIONI**: bytecode traducibile in linguaggio macchina a RUN-TIME. Inoltre buon livello di ottimizzazione del linguaggio macchina.
- **INTERPRETATO**: Il bytecode è un input per l'INTERPRETE Java, ciò contribuisce alla neutralità dell'architettura e permette di avere file compilati più leggeri di un eseguibile.
- **MULTITHREADED**: C'è il concetto di Thread in Java, con dei metodi di supporto per gestire in maniera semplice e affidabile le concorrenza.
- **DINAMICO**: C++ soffre del "constant recompilation problem";
 - Linkin fatto per riferimento ad indirizzi (numerico) nei files compilati
 - ricompilare anche le classi che riferiscono un'altra classe modificata

* JavaC use RIFERIMENTI SIMBOLICI, risolti dall'interprete al momento del linking; tuttavia Java non riesce a risolvere in maniera ottimale il "fragile base-class problem".
- **JVM**:
 - parametri passati sempre per VALORE
 - le variabili che riferiscono un oggetto sono di fatto dei puntatori

MODELLAZIONE E UML

Si vogliono utilizzare MODELLI per:

- **SINTETIZZARE**: passare da una rappresentazione di un sistema software al sistema software reale, più prima attraverso varie raffinzioni
 - **ANALIZZARE**: raccogliere dati e informazioni su un sistema software per ottenere una rappresentazione che mi permette di analizzare le caratteristiche
 - **RAPPRESENTARE**: ottenere un' ASTRAZIONE di un sistema software
 - **AUTOMATIZZARE**: automatizzare la rappresentazione, la sintesi e l'analisi con strumenti di supporto.
- **ASTRAZIONE**: Rende più agevole la comunicazione con i colleghi, i clienti e non tiene vincolati solo al codice.

UML

Unified Modeling Language → LINGUAGGIO DI MODELLAZIONE VISUALE

UML consente di scomporre un processo di sviluppo di un sistema software da 5 diversi punti di vista, detti appunto VISTE o PROSPETTI:

- USE CASE VIEW: modello come percepito dagli ATTORI ESTERNI; enfatizza COME è fatta la struttura esterna del sistema (black-box) (CHE COSA FA)
- LOGICAL VIEW: progettazione della struttura interna del sistema (white-box) descrive COME sono realizzate le funzionalità del sistema, descritte nelle Use Case View.
- IMPLEMENTATION VIEW: strutturare i modelli implementativi organizzando il codice del sistema in moduli, packages e così via.
- PROCESS VIEW: modelli che descrivono la DINAMICA del sistema (behaviour), utile per un utilizzo efficace delle risorse e per la gestione di esecuzione parallela ed eventi asincroni (esterni).
- DEPLOYMENT VIEW: modelli che descrivono il deploy del sistema software, enfatizzando la topologia e l'organizzazione dei dispositivi fisici necessari e specificando COME il software è mappato sull'architettura fisica.

OBJECT ORIENTATION

I concetti fondamentali del paradigma Object Oriented sono:

- Classe
 - Istanza
 - Metodo
 - Operazione
 - Messaggio
 - Incapsulamento
 - Ereditarietà
 - Polimorfismo
- FONDAMENTALI (quasi ASSIEME)
- } Necessari per avere O.O.

CLASSE: tipo di dato definire per rappresentare un'entità del dominio dell'applicazione (famiglie di entità) o per definire un tipo di dato utile per l'ingegnerizzazione del sistema. È COESA semanticamente. Ha le seguenti 2 caratteristiche:

- PROPRIETÀ: attributi (come la struct C)
- COMPORTAMENTO: operazioni offerte (è ciò che permette di parlare di O.O., anziché di struct C).

ISTANZA / OGGETTO: Ha un TIPO, definito dalla Classe che istanzia; inoltre ha uno:

- STATO: valore degli attributi, definiti nella classe
- COMPORTAMENTO: le operazioni definite nella classe

OPERAZIONE: funzionalità offerta da una classe e invocabile su oggetti che istanziano tale classe. Ha una SEGNAURA formata dal NOME dell'operazione e dal TIPO dei parametri in input. Può specificare il tipo di ritorno.

NO IMPLEMENTAZIONE.

METODO: Specifica IMPLEMENTAZIONE di un'operazione offerta da una classe (o da un'interfaccia), costituita da un blocco di istruzioni. È caratterizzato dalla segnaura dell'operazione.

MESSAGGIO: È la richiesta a RUN-TIME dell'invocazione di un METODO offerto da un'istanza B da parte di un'istanza A. Può fallire.

* Il concetto di METODO e di OPERAZIONE è STATICO, definito a DESIGN-TIME nel momento in cui si sta programmando; invece il MESSAGGIO è DINAMICO in quanto viene inviato a RUN-TIME.

COSTRUTTORE: E' un'operazione speciale ed è DI CLASSE (STATIC);
deve avere lo stesso nome della classe e non c'è bisogno di indicare il return type, poiché implicito.
Ce ne può essere più di 1, ma NON sono alterabili dalle classi figlie.

DISTRUTTORE: Operazione speciale che dealloca lo spazio occupato da un'istanza.
• In alcuni linguaggi è possibile chiamare esplicitamente (C++),
ci sono però problematiche da gestire riguardo le relazioni con altre istanze, delle istanze che si sta rimuovendo.
• In altri (Java), c'è il GARBAGE COLLECTOR che ripulisce dall'heap le istanze ISOLATE, che non hanno più relazioni con alcun'altra istanza.
Il garbage collector è un thread a basse priorità della JVM.
* Le istanze non possono distruggere se stesse! Il distruttore è invocato sempre da altre istanze o dal garbage collector implicitamente.

THIS: E' un* attributo speciale che indica il riferimento all'istanza corrente.
Serve per disambiguare nei metodi / costruttori:
- In un costruttore, per invocare uno non di default (this(sao));
- All'interno di metodi, per chiarire lo scope di variabili, per non confondersi tra attributi e parametri formali.

AMBITO DI CLASSE (STATIC):

- ATTRIBUTO DI CLASSE: esiste anche se non sono allocate istanze della classe, è CONDIVISO tra tutte le istanze, le quali hanno un puntatore alla stessa area nella heap in cui si trova l'attributo; (IL VALORE è CONDIVISO!)
- OPERAZIONE DI CLASSE: può essere invocata anche senza che sia presente un'istanza della classe, mediante il nome della classe stessa: NomeClasse.NomeOperazione(...);

In Java → STATIC (modificatore)

In UML → sottolineatura dell'attributo o dell'operazione.

INCAPSULAMENTO

È una TECNICA usata nell' Object-Orientation che si basa sul seguente

PRINCIPIO DI INFORMATION-HIDING:

"Bisogna separare tra la specifica di una funzionalità e la sua implementazione."

Tale tecnica coinvolge l'idea che debba essere noto il COSA si può fare, ma NON il COME tale attività viene svolta, cioè l'effettiva implementazione.

Ciò sostiene la LOCALITÀ DELLE MODIFICHE e aumenta notevolmente il RIUSO anche di parti di un sistema software; inoltre, permette di eseguire dei controlli (in operazioni di tipo setAttribute (value)) su tentativi di modifica dei dati da parte di utilizzatori esterni, aumentando SICUREZZA e COERENZA.

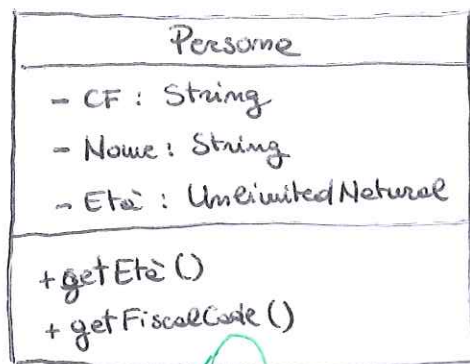
• COME SI FA? Tramite la VISIBILITÀ:

Java	UML
public	+
private	-
protected	#

• OBBIETTIVO: NASCONDERE le scelte che possano essere soggette a cambiamenti, per avere un impatto limitato.

→ Attributi PRIVATI, Operazioni tendenzialmente PUBBLICHE.

• GENERALIZZAZIONE:



→ SUPERCLASSE



→ Sottoclasse

E' una relazione:

- UNIFORME: vale per TUTTE le istanze
- TRANSITIVA: se B specializza A e C specializza B \Rightarrow C specializza A
Determina un ordinamento tra classi

* La classe figlia IS-A-KIND-OF la classe padre!

EREDITARIETA'

La sottoclasse:

- EREDITA dalla superclasse TUTTE le operazioni e TUTTI gli attributi, anche quelli privati; Eredita TUTTO 1 e 1 SOLA VOLTA!
- Può avere più attributi e/o operazioni; tipicamente è una specializzazione e contiene più informazioni della superclasse;
- Può ridefinire i METODI ereditati dalla superclasse (@Override);

• EREDITARIETA' MULTIPLA:

Se una classe può avere più di 1 Superclasse diretta! E' lecito in UML e in C++, ma non in Java.

VANTAGGI: Costruzione di oggetti complessi molto semplicemente, con sintassi semplice e può fornire migliore rappresentazione della realtà.

SVANTAGGI: NAME CLASH sulle operazioni ereditate e PROBLEMA DEL DIAMANTE, complicazione del linguaggio che la implementa: non bastano più solo this() e super()!

• PRINCIPIO DI SOSTITUIBILITA' DI LISKOV:

- Se $q(x)$ è una proprietà valida per oggetti x di tipo T , allora $q(y)$ deve essere valida per oggetti y di tipo S , dove S è un SOTTO TIPO di T .
- Sia T una classe ed S una sua sottoclasse; in tutti i contesti in cui si usa un'istanza di T deve essere possibile utilizzare una qualsiasi istanza di S (o di qualunque altra sottoclasse di T a qualsiasi livello).
- Una superclasse deve sempre essere SOSTITUIBILE da una sua qualsiasi sottoclasse, anche non diretta.

• CLASSE ASTRATTA:

Sono dette classi parzialmente definite, con uno stato (attributi) e un comportamento, dato dalle operazioni che esporta; tuttavia, alcune di queste operazioni (ABSTRACT) può non essere implementato il metodo!

QUANDO: Quando si vuole modellare contesti in cui un insieme di classi include operazioni con la STESSA SEMANTICA, ma implementate con metodi DIFFERENTI.

* Una classe estratta NON PUO' MAI ESSERE ISTANZIATA, altrimenti se arriva una messaggio su un'operazione estratta, non c'è il metodo da eseguire.

POLIMORFISMO

E' la capacità di una classe di "comportarsi" in modi differenti a seconda delle specifiche situazioni.

In Java si realizza tramite EREDITARIETA': c'è una Superclasse astratta A con classi figlie B_1, \dots, B_m , che implementano metodi differenti per una certa operazione estratta esportata da A ; si utilizza una variabile S di tipo A , che in base ad un MECCANISMO DI SCELTA, verrà fatta puntare ad un'istanza di tipo B_1, \dots, B_m .

* Il comportamento è deciso solo a RUN-TIME!

Condizioni necessarie per avere POLIMORFISMO:

- (1) Avere una GERARCHIA di classi (ereditarietà e generalizzazione);
- (2) Variabile del tipo la superclasse astratta;
- (3) Meccanismo di scelta per l'istanza da puntare;
- (4) Usare un linguaggio di programmazione che supporti BINDING DINAMICO, come Java.

* Sistemi polimorfici sono FLESSIBILI e FACILI DA ESTENDERE!

OVERRIDING E OVERLOADING:

OVERRIDING: Sovrascrivo un metodo associato ad una certa operazione ereditata dalla classe padre.

Precondizioni per avere override (Java: @Override):

1. Gerarchie di classi (A generalizza B);
2. Le operazioni devono avere la stessa signature (A.m e B.m);
3. A.m è public o protected, oppure B.m fa override di un'operazione che a sua volta ha fatto override di A.m.

OVERLOADING: Aggiungiamo una funzionalità ad una classe, fornendole una nuova operazione.

Sostanzialmente se in una classe sono definiti 2 o più metodi, con lo STESSO NOME, una signature differente (cambiamo i parametri formali o il loro ordine). I metodi sono anche ereditati!

BINDING IN JAVA

Il BINDING è l'associazione tra l'invocazione di una OPERAZIONE con l'effettivo METODO da eseguire.

A seconda del momento in cui tale associazione viene effettuata, si distinguono 2 tipi di binding:

- **EARLY BINDING:** effettuato a tempo di COMPILAZIONE/LINKING delle librerie, PRIMA dell'esecuzione del programma.
(STATICO)

Im C/C++: "Considera il metodo con OFFSET = XXX dell'inizio della classe Foo" → RIFERIMENTO NUMERICO

Im Java: "Considera il metodo che si chiama 'saludos()' dentro la classe Foo" → RIFERIMENTO SIMBOLICO

- **LATE BINDING:** effettuato a RUNTIME, basandosi sull'effettivo tipo dell'istanza coinvolta nell'esecuzione.
(DINAMICO)

Im Java: "Considera il metodo che si chiama 'saludos()' e va a cercare nella dichiarazione della classe che è il tipo dell'istanza riferita dalla variabile f a RUNTIME"

* In C/C++, a causa del riferimento numerico, soffre del CONSTANT RECOMPILATION PROBLEM e del FRAGILE BASE-CLASS PROBLEM!

* Java effettua sia l'early binding a tempo di compilazione/linking, con riferimenti simbolici, sia il late binding (DINAMICO).

• QUALE BINDING IN JAVA?

In Java viene sempre effettuato il LATE BINDING, quindi BINDING DINAMICO, trovare se l'operazione è dichiarata con almeno 1 dei seguenti modificatori:

- STATIC: l'operazione ha ambito di classe, quindi non ha bisogno di un'istanza e del suo tipo. Per di più, il metodo è CONDIVISO tra tutte le istanze, quindi sarà sempre quello!
- FINAL: si dice esplicitamente di vietare l'Overriding dell'operazione, quindi è certo che il metodo sarà sempre quello dichiarato dalla superclasse e il binding dinamico viene disabilitato.
- PRIVATE: un'operazione private può essere invocata solo dalla stessa classe che la definisce (nemmeno dai figli), quindi anche in questo caso è ovvio quale sia il metodo da eseguire e priori e il BINDING DINAMICO sarebbe solo uno SPRECO DI RISORSE.
In Java un metodo private è implicitamente final!

JVM SOTTO IL COFANO

La JVM alloca e gestisce sia lo STACK di programma che l'HEAP.
Nell'HEAP ci sono delle aree di memoria speciali:

- JVM method area: è unica e CONDIVISA tra tutti i thread e non è soggetta a garbage collection.
Contiene i vari run-time constant pool e per ogni classe altre info, quali codici di metodi e costruttori.
- RUN-TIME CONSTANT POOL: è relativa ad una SPECIFICA CLASSE A, di cui è stato istanziato almeno un oggetto, e contiene:
 - le costanti di A;
 - i riferimenti di A risolti a tempo di compilazione (EARLY BINDING);
 - i riferimenti di A da risolvere a run-time (BINDING DINAMICO).Si trova nella JVM method area!

• GESTIONE DEL BINDING IN JAVA:

La JVM implementa l'invocazione di metodi attraverso le 3 seguenti istruzioni bytecode:

- **invokevirtual:** è quella di default e che implementa binding dinamico; viene acceduto il run-time constant pool della ~~istanza~~ classe di cui è istanza l'oggetto puntato dalla variabile e usando i parametri attuali (il tipo dell'istanza si sa a RUNTIME).

Object x;
...
x.equals("hello");

→ aload_1 // Allocazione dell'oggetto x sullo stack
ldc "hello" // Allocazione parametro attuale su stack
invokevirtual java/lang/Object/equals (Ljava/lang/Object;)Z
// metodo definito dal TIPO dell'ISTANZA EFFETTIVA

- **invokestatic:** quando c'è il modificatore static; viene acceduto il run-time constant pool della classe tipo della VARIABILE (non dell'istanza!) e usando i parametri attuali.
L'istanza potrebbe non esistere

System.exit(1);

→ // Non alloco l'oggetto! Potrebbe non esistere (static)!
iconst_1 // Alloca parametro attuale su stack
invokestatic java/lang/System/exit (I)V
// metodo chiamato direttamente sulla classe!

- **invokespecial:** usato per il main e per metodi di tipo private/finale; è un ibrido tra le 2; run-time constant pool della classe del tipo della variabile (invokestatic), ma ci si porta dietro anche il riferimento (this) all'effettiva istanza.

• Perché Java è DINAMICO?

Perché NON ha riferimenti numerici, usa RIFERIMENTI SIMBOLICI e supporta il BINDING DINAMICO, a runtime!

- I parametri delle istruzioni "invoke" sono riferimenti che identificano UNICAMENTE un metodo;
- I riferimenti simbolici sono risolti la 1^a VOLTA che la JVM incontra un'istruzione invoke!

INTERFACCE

• FRAGILE BASE-CLASS PROBLEM (FBCP):

Ogni volta che aggiungi un nuovo metodo o una nuova variabile di istanza ad una classe A, qualunque altra classe che riferenzi A richiederà una RICOMPILAZIONE oppure si romperà.

~ Java White Paper

• FRAGILE BASE-CLASS STRUCTURE (FBCS):

Quando abbiamo 2 classi con una relazione di GENERALIZZAZIONE tra di loro, non per forza dirette, e con specifici metodi dichiarati e definizioni. Una FBCS è il luogo dove PUO' occorrere un FBCP se, per esempio, la sotto-classe fa @Override di un metodo della superclasse che, magari anche a cause di modifiche di metodi della classe padre (EREDITATI), si va incontro ad una RICORSIONE MUTUA INFINITA.

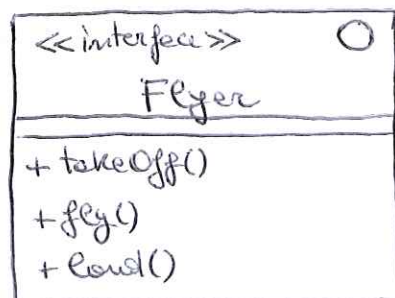
* Vedere esempio di `Sensor()` e `DualModeSensor()`, in cui il metodo `setSamplingFrequency(int f)`, di cui viene fatto OVERRIDE, e `setRateOn()`, si richiamano a vicenda all'infinito e causano una condizione di STACK OVERFLOW!

* Il FBCP NON si risolve! Ma può essere MITIGATO, enfatizzando l'aspetto funzionale e riducendo l'ereditarietà di aspetti che possono causare un FBCP.

• INTERFACCIA: E' una collezione di OPERAZIONI, che sono utilizzate per specificare un SERVIZIO di una classe o di un componente. Definisce SOLO la SEGNAURA delle operazioni e NON i metodi implementati (E' UN "CONTRATTO"). NON ha ATTRIBUTI!

* Essendo una specifica parziale, NON SI PUO' ISTANZIARE!

```
public interface Flyer {  
    public void takeOff();  
    public void fly();  
    public void land();  
}
```



INTERFACCIA VS CLASSE ASTRATTA

• CARATTERISTICHE COMUNI:

- Modellano operazioni NON associate ad un METODO;
- Impungono alle sottoclassi concrete l'OVERRIDING delle loro operazioni (astratte), costringendole ad implementarne il metodo;
- Sono COLLEZIONI DI OPERAZIONI utilizzate per specificare un servizio offerto da una classe o da un componente;
- NON sono ISTANZIABILI

• INTERFACCIA

- (1) Tutte le operazioni non hanno associato un metodo
- (2) NON definisce alcun attributo (NO STATO)!
- (3) Modella un MODO D'USO del sistema o di un sottosistema
- (4) Generalmente rappresenta una particolare VISTA del sistema, non una sua parte.

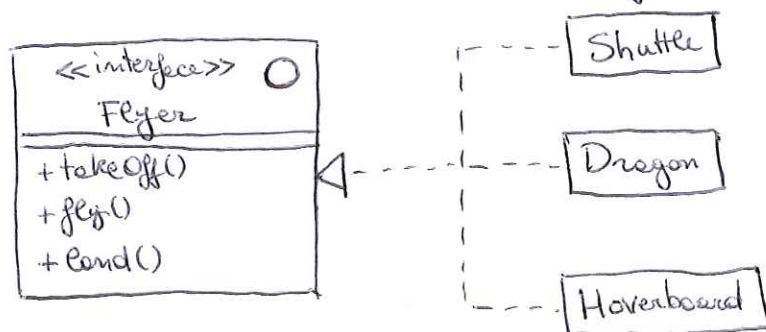
- E' molto più relativa alle USE CASE VIEW: modella quasi un caso d'uso, specificando il COSA si può fare, ma NON COME!

• CLASSE ASTRATTA

- (1) Basta che abbia ALMENO 1 operazione astratta (senza metodo);
- (2) Può avere ATTRIBUTI ed ha quindi uno STATO.
- (3) Generalmente modella una parte della struttura statica del sistema; rappresenta un' ENTITA' di dominio oppure un elemento di ingegnerizzazione.

- Ha uno STATO, che definisce tutte le possibili CONFIGURAZIONI diverse che un'istanza può assumere → RAPPRESENTA una parte della struttura statica!

- RELAZIONE DI REALIZZAZIONE: è la relazione che intercorre tra la specificazione di un'interfaccia e ciò che la realizza.



- In Java: Keyword "implements"

• PERCHÉ USARE LE INTERFACCE?

- INTER-COLLEGARE sistemi diversi : il sistema è strutturato in base all'insieme delle interfacce definite dai vari sottosistemi
- Definire ARCHITETTURE ASTRATTE, basate sulle interazioni, ma non su come tali interazioni sono implementate.
- AUMENTARE LA MODULARITÀ DEL SISTEMA : gran parte della progettazione si concentra sull'individuazione e modellazione delle principali forme di INTERAZIONE per mezzo di INTERFACCE.

• EREDITARIETÀ VS REALIZZAZIONE :

• EREDITARIETÀ :

- 1) Trasmissione di caratteristiche comuni : attributi, relazioni, metodi.
- 2) E' una relazione di tipo "IS-A-KIND-OF" : è la più forte forma di interdipendenza tra classi e può causare FRAGILE BASE-CLASS PROBLEM.
- 3) L' INCAPSULAMENTO nella gerarchia è più DEBOLE : conoscenza della classe padre.
- 4) E' davvero necessaria solo se ho bisogno di EREDITARE DETTAGLI IMPLEMENTATIVI. E' la forma primaria e basilare di RIUSO, ma ha perso un po' nel tempo questo scopo, prendendo notazioni più semantiche e legate al Principio di Sostituibilità di Liskov.

• REALIZZAZIONE :

- 1) Utile quando si vuole definire un CONTRATTO e garantire che sia rispettato almeno simbolicamente.
- 2) Implica l'accettazione delle specifiche di INTERAZIONE previste dall'interfaccia.
- 3) NON OFFRE RIUSO, in alcun modo !
- 4) E' più FLESSIBILE e ROBUSTA dell'ereditarietà, e tende a MITIGARE il verificarsi del FBCP !

In Java :

- Una classe può avere 1 solo padre (STRONG TYPED) !
- Una classe può offrire VISTE DIFFERENTI, o definire MODI D'USO diversi, quindi una classe può implementare più interfacce !

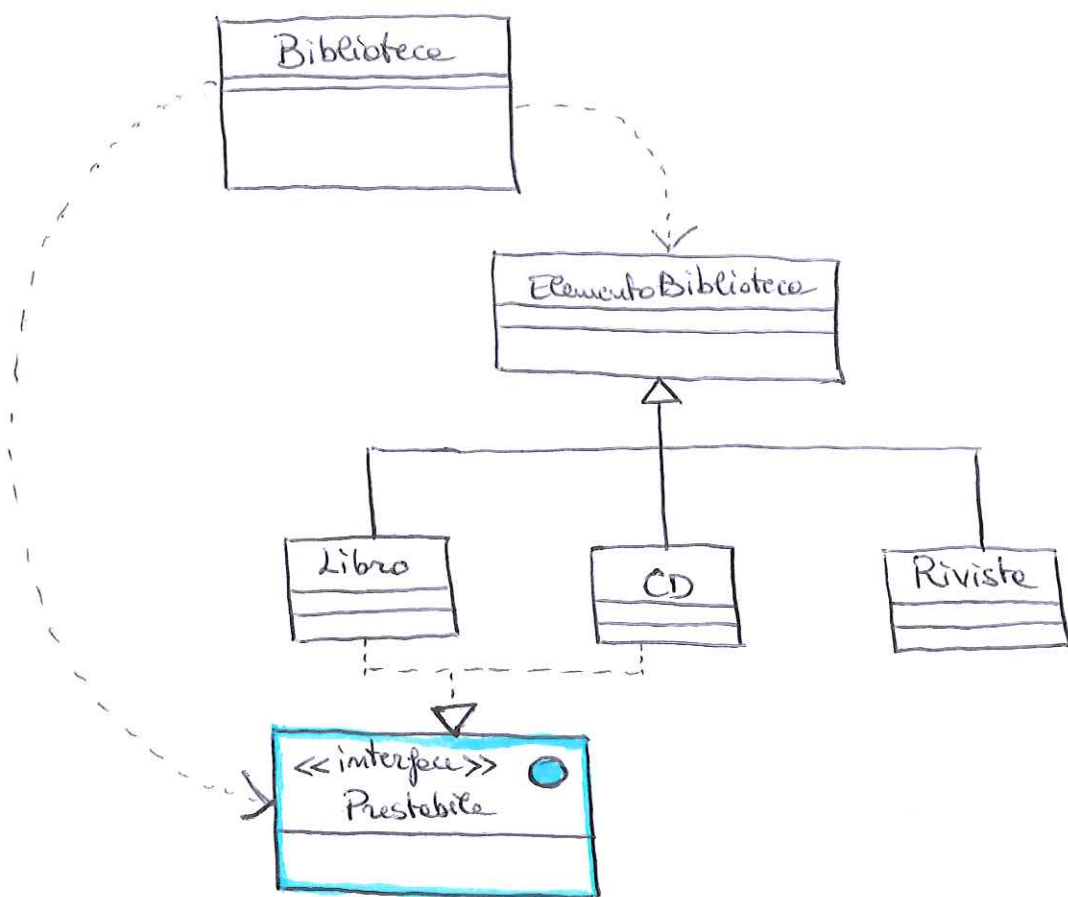
• INTERFACE COLLISION:

- E' possibile avere relazioni di GENERALIZZAZIONE tra INTERFACCIE (is a kind of);
- E' possibile combinare insieme EREDITARIETA' e REALIZZAZIONE.

Bisogna però stare attenti al cosiddetto problema di "INTERFACE COLLISION", che si verifica quando vengono ereditate operazioni e/o si è costretti a realizzare (implementare) operazioni con STESSA SEGNAURA, ma TIPO DI RITORNO DIVERSO.

* L'interfaccia definisce un CONTRATTO da rispettare, quindi bisogna ritornare il tipo di ritorno previsto!

• Esercizio: Modellare un sistema di gestione per una biblioteca, la quale è in grado di gestire il prestito di ALMENO libri e CD. Supportare anche la gestione di elementi non prestabili come le riviste.



* L'essere "PRESTABILE" è un MODO D'USO di un elemento, non una classificazione naturale tra i vari elementi, una tassonomia.

→ INTERFACCIA meglio di una GENERALIZZAZIONE!

RELAZIONI IN UN CLASS DIAGRAM


• ASSOCIAZIONE

Una RELAZIONE rappresenta una INTERAZIONE tra gli elementi modellati. Una relazione tra A e B indica che A è a conoscenza di B e (in qualche modo, che dipende dal TIPO della relazione) può interagirvi.

• TIPO: STRUTTURALE

• Possibili proprietà:

- SIMMETRICA: se è navigabile in entrambe le direzioni, altrimenti è asimmetrica;
- RIFLESSIVA: se è tra oggetti della stessa classe
- BINARIA oppure N-ARIA

• SIMBOLO: 


• SIGNIFICATO: [Indice la possibilità di una classe di inviare MESSAGGI alle classi associate (ovviamente gli oggetti delle classi!).]

• CARATTERISTICHE:

- (1) può avere un NOME
- (2) il RUOLO degli operandi
- (3) uno STEREOTIPO
- (4) una CARDINALITÀ, per esempio [0...*]

• AGGREGAZIONE

• TIPO: GERARCHICA, denominata anche relazione "WHOLE-PART"

• SIMBOLO: ; il rombo bianco è presso la classe aggregante, il whole.

• SIGNIFICATO: [Interconne tra una ~~parte~~ ^{classe} che rappresenta il tutto e delle classi che sono delle ENTITÀ AUTONOME che la compongono.
NON impone vincoli sul ciclo di vite degli elementi aggregati.]

• CARATTERISTICHE:


- (1) Può dare il NOME ai componenti, con visibilità
- (2) Può avere una CARDINALITÀ

(3) Aggregazioni CIRCOLARI sono ERRATE:

A aggrega B
B aggrega C
C aggrega A



• CORPOSIZIONE

- TIPOLOGIA: GERARCHICA (whole-part); è un'AGGREGAZIONE FORTE
- SIMBOLO: ; il rombo nero è presso la classe aggregante
- SIGNIFICATO: [E' sempre un'aggregazione whole-part, ma più forte, in quanto, nonostante le istanze composte non devono necessariamente essere create con le istanze componenti, si ha che una volta che vengono create seguono il CICLO DI VITA dell'istanza componente]

• CARATTERISTICHE:

- (1) NOTE dei componenti + visibili
- (2) CARDINALITÀ
- (3) NO CICLI

• CONSEGUENZE:

Essendo che quando si rimuove l'istanza componente (whole) si è OBBLIGATI a rimuovere anche le istanze precedentemente composte (part), considerando che in Java la rimozione avviene tramite GARBAGE COLLECTOR (istanze eliminate solo se non hanno relazioni) si giunge al seguente:

PRINCIPIO DI ESCLUSIVITÀ:

Le classi componenti (part), o meglio le loro istanze, NON POSSONO tassativamente essere relazionate con classi diverse della classe aggregante (whole)!

* Ne segue che un costruttore vuoto in cui si creano da zero le istanze componenti è da preferire ad uno parametrico (in cui si passano le istanze), poiché bisognerebbe comunque CREARE (new()) nuove istanze e poi copiarvi lo stato delle istanze passate come parametro → CLONARE!

• DIPENDENZA

• TIPOLOGIA: DIPENDENZA

• SIMBOLO: ; frecce verso la classe target

• SIGNIFICATO: [Segnalare che l'implementazione della classe source dipende fortemente da quella della classe target. Serve per dire che il cambiamento della target avrà (molto probabilmente) conseguenze anche sulla source.]

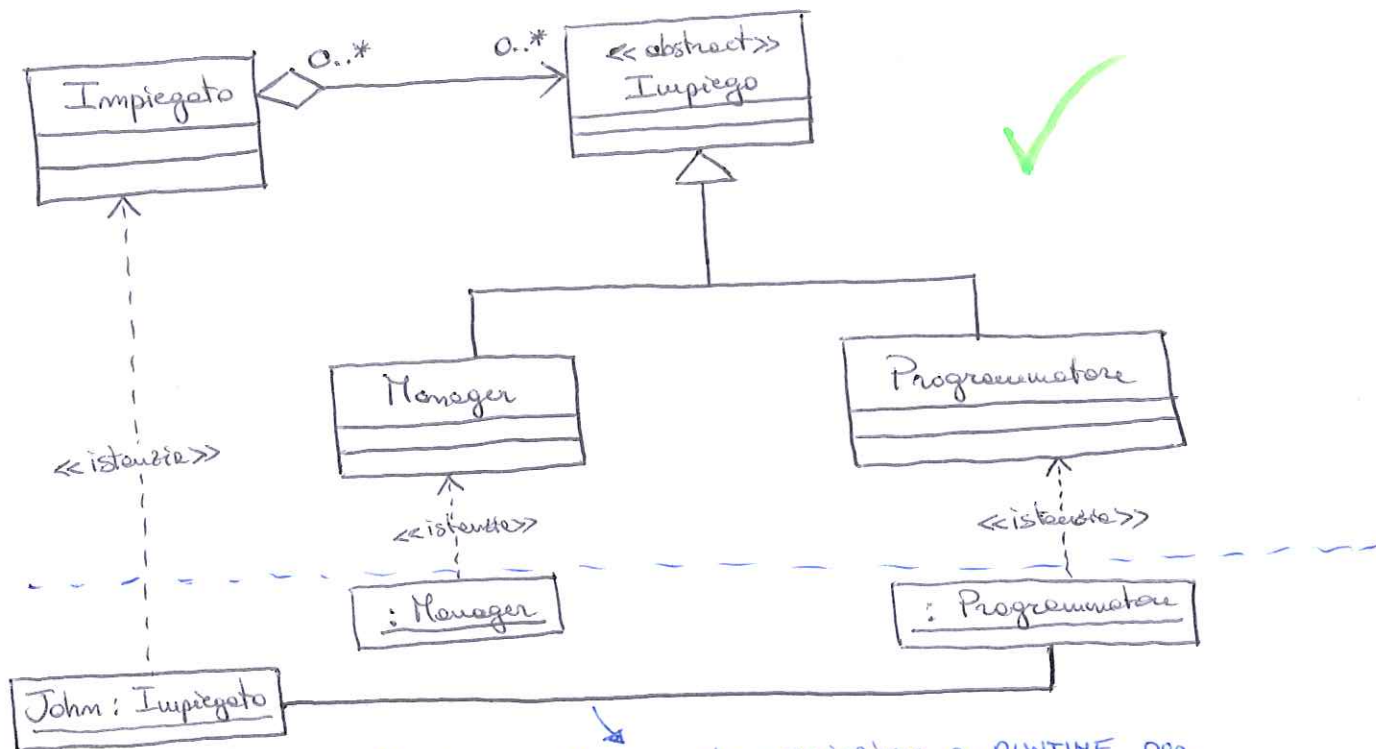
• CARATTERISTICHE:

- (1) Si usa solo per specificare FORTE dipendenze
- (2) Essendo vaga, spesso le vengono aggiunti degli STEREOTIPI:
«use», «abstraction», «derive», ...

PATTERN DELLA METAMORFOSI

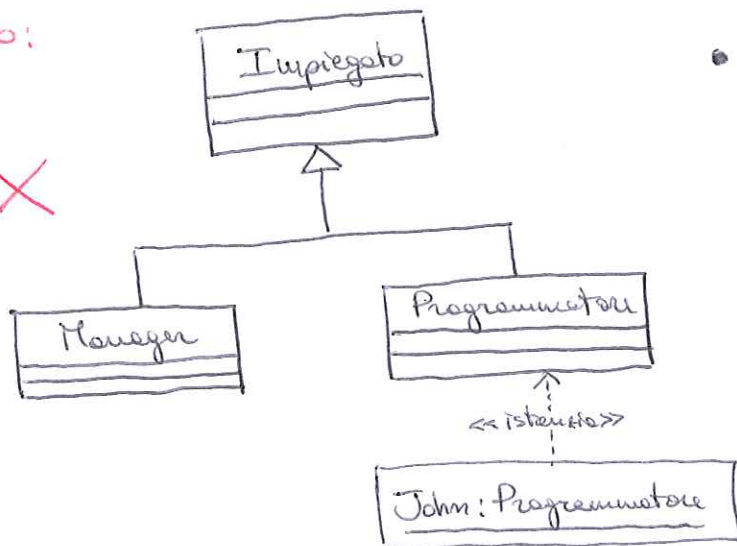
QUANDO? Quando ci sono delle istanze che EVOLVONO NEL TEMPO, cambiano (METAMORFOSI); di solito cambia proprio il tipo dell'istanza.

PERCHÉ? Non confondere le entità del dominio, con il loro RUOLO, che può CAMBIARE nel tempo!



Basta cambiare questa associazione a RUNTIME per promuovere John da Programmatore a ~~Impiegato~~ Manager.

* ERRORE:



- In questo caso, dovrei creare una nuova istanza di Manager e copiarvi i dati di John: Programmatore. Ma POTREI PERDERE DEI DATI!

I dati ereditati da Impiegato e non accessibili:

per es. un attributo private String iban;
per cui non c'è un'operazione getIban(); !!!

* Il pattern della Metamorfosi si ha quando ci sono EVOLUZIONI nel MONDO DELLE ISTANZE (!), che divergono dalle viste statiche del sistema.
E' legato alla DINAMICA!

