Programmazione ad'Oggetti

Leonardo Mengozzi

Contents

1 Fasi sviluppo Sofware

Un Programma (algoritmo) risolve una classe di problemi.

Un Sistema sofware fornisce varie funzionalità grazie alla cooperazione di componenti di diversa natura.

Fasi processo sviluppo: **1 Analisi** che fare?, **2 Design** come farlo?, **3 Implementazione/codifica** Quale algoritmo?, **4 Post-coficia**. Fasi 1-2 fatte dai senior e fase 3 junior (2-3 oggi unificate). La fase 4 più impiegare fino 70% se fase precedenti fatte male/sbrigativamente (Software crisis). Tutte fasi fattibili dalla stessa persona.

Un analisi è corretta se persone diverse giungono alla stessa soluzione.

1.1 Problem space vs Solution space

Problem space sono le entità/relazioni/processi del mondo reale che formano il problema. Solution space sono le entità/relazioni/processi nel mondo artificiale (espresse nel linguaggio di programmazione).

Per passare dal Problem space al Solution solution si esegue un "mapping" che più semplice è meglio ho fatto le **astrazioni**[1].

I linguaggi di programmazione attuano l'astrazione coi loro costrutti, più o meno performanti, che rendono il mapping più o meno facile.

I linguaggi moderni hanno un livello d'astrazione lontano dall'HardWare, i suoi problemi e la gestione della memoria.

1.2 Programmazione ad oggetti (OOP)

Vantaggi: poche astrazioni chiave, mapping ottimo e semplice, estensibilità e riutilizzo, librerie auto costruite, C-like, esecuzione efficente.

Critiche: necessaria disciplina.

2 Teoria degli oggetti

Classe: descrizione comportamento e forma oggetti. Indica come comunicare con i suoi oggetti, con messaggi che modificano stato e comportamento.

Oggetto: entità (istanza di classe) manipolabile, con memoria, che comunicano tramite le loro operazioni descritte dalla classe di appartenenza.

Oggetti della stessa classe hanno comportamento e forma indentica, sono detti simili. Un oggetto non cambia mai classe, semmai si elimina e sene crea il sostituto.

Nota: L'approccio OOP è usato anche in UML.

[1] Strumento che semplifica sistemi informatici ma anche del mondo reale evidenziando "la parte importante". Si possono fare più livelli di astrazione

2.1 Com'è fatto un buon oggetto

Un oggetto ha un interfaccia[2], deve fornire un servizio, deve nascondere le implementazioni (riutilizzabili) e l'intero oggetto deve essere riutilizzabile tramite ereditarietà. Precisazioni:

- Un oggetto fornisce un **sotto-servizio** dell'intero programma[3] (principio decomposizione). Linne giuda: 1 oggetto senza servizio si elimina, 2 oggetto con più servizi si divide.
- L'implementazione di un oggetto (logiche interne) devono essere note solo al creatore della classe (Information hidding), cosi facendo l'utilizzatore è tutelato, da modifiche interne, avendo una piccola visione del tutto. *less is more*.
- Il creatore e l'utilizzatore riutilizzano le classi con gli approcci:
  1. **has-a** (composizione), classe costituita da altre classi (oggetto ha come campi altri oggetti). Approccio dinamico, occultabile.
  2. **is-a** (ereditarietà), classe estende servizi di un'altra classe (oggetto ha campi/metodi di altri oggetti).

3 Perchè Java?

- **Write once run everywhere**, eseguibile uvunque senza ricompilazione grazie JVM (HardWare virtuale, a stack) che processa un codice specifico "byte code", creando il corrispettivo eseguibile per ogni pc/os. Meno prestante.
- **Keep it simple, stupid**, in teoria non in pratica.

[2] Insieme dei metodi definiti dall'interfaccia con cui l'oggetto riceve messaggi
[3] Set di oggetti che si comunicano cosa fare.

4 Variabili, Oggetti, Classi, Campi e Metodi

4.1 (Almost) Everything is an object

Le variabili, contenitori con nomi, ora non denotano solo valori numerici (come in C), ma anche veri e propri oggetti irriducibili.

Non ci sono meccanismi per controllo diretto memoria. Le variabili sono nomi "locali" con riferimenti ad'*oggetti* e non maschere di indirizzi in memoria a cui accedere direttamente.

Le variabili posso essere di tipo *Java Types* quindi classi predefinite e autoimplementate oppure *tipi primitivi*.

4.1.1 Tipi Primitivi

Non conviene trattare tutto come oggetto. I tipi atomici del C si sono mantenuti definendo una dimensione fissa e rimuovendo gli unsigned. Si è introdotto boolean con **true/false**.

Questi tipi sono unici e fissi da linguaggio.

Tipi primitivi	Dimensione	Minimo	Massimo
boolean	–	–	–
char	16bits	Unicode 0	Unicode $2^{16}-1$
byte	8bits	-128	+127
short	16bits	-2^{15}	$-2^{15}-1$
int	32bits	-2^{31}	$+2^{31}-1$
long	64bits	-2^{63}	$-2^{63}-1$
float	32bits	IEEE754	IEEE754
double	64bits	IEEE754	IEEE754

Le librerie *BigDecimal, BigInteger* gestiscono numeri di dimensione/precisione arbitraria.

Nota: In Java l'uso della memoria per i valori true/false di boolean, e tante altre cose, non sono date a sapere al programmatore dato che si dovrebbe concentrare su altro.

4.1.2 Stack e Heap

Tutte le variabili sono memorizzate nello **stack**. Le variabili di tipo primitivo sono affiancate dal loro valore mentre le variabili tipo classe sono affiancate dall'identità del loro oggetto. Gli oggetti sono memorizzati nell'heap.

Uno stesso oggetto può essere puntato da variabili che si riferiscono alla stessa identità.

4.2 Oggetti

1. Creazione

```
<Tipo | var> <nome> = <new Tipo([Tipo1 par1, ...])| altraVariabile |
 null>;
```

altraVariabile deve essere della stessa classe della variabile che sto definendo.

Nota: solo quando si scrive *new* (Keyword di linguaggio) si crea un oggetto dalla classe indicata.

`var`[4] fa infierire[5] il tipo della variabile locale per allegerire il codice. Se manca l'espressione non va, esempio `var i;`.

4.3 Classi

Nomi classi sempre con la maiuscola.

Definisco le configurazioni.

Le classi sono tipi di dato in un linguaggio a oggetti tutto è un oggetto fino a un certo punto.

4.3.1 Classe Object

Definizione, Creazione ed'Inizializzazione:

1. `<Tipo> <nome>;` sola definizione.
2. `<Tipo> <nome> = valore;` definizone ed'inizializzazione.

4.3.2 Precisazioni slasse String

Diversi modi per inizializzare un oggetto stringa:

1. `... = new String();` inizializza con una stringa (sequenza di caratteri) vuota.
2. `... = new String("...");` inizializza con la stringa passata come parametro.
3. `... = "..."` inizializza direttamente con una stringa (come in C), caso unico.

[4] Local variable type inference

[5] Far dedurre al compilatore il tipo della variabile locale dall'espressione assegnata.