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 Struttura Programma Java

Un programma Java è composta da librerie di classi del JDK, Package[4] e Moduli[5], librerie di esterne e un insieme di classi fondamentali, come la class main[6].
`public static void main(String[] args){...}`

Per importare le classi di una libreria:

- `import java...;`, importa una singola classe.
- `import java...*;`, importa l'intero Package.
- `import java.lang.*;`, importazione di default.

Nota: Il nome completo di una classe dipende dal Package in cui si trova.

4.1 Esecuzione Programma

1. Salvare la classe in un file **"NomeClasse.java"**.
2. Compilare con *javac NomeFileClasse.java*. Genererà il **bytecode NomeFileClasse.class** per la JVM.
3. Esegiure con *java NomeFileClasse*. La JVM cercherà il main da cui partire a eseguire.

Lavorando con più file: si compila tutto con *javac *.java* poi si esegue solo la classe main.

5 (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*.

Visibilità legata al blocco di definizione.

Variabili non inizializzate sono inutilizzabili.

Il **garbage collector** (componente della JVM) dealloca automaticamente memoria non più utilizzata direttamente o indirettamente dall'heap. Un'oggetto continua a esistere dopo la fine esecuzione dello scope di una variabile che gli fa riferimento.

[4] Contenitori, gerarcici tra loro, di una decina di classi di alto livello con scopo comune
[5] Insieme di Package costituente un frammento di codice autonomo.
[6] Un main è il punto d'accesso di un programma.

5.1 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.

5.2 Stack e Heap

Gli oggetti sono memorizzati nell'**heap**. Tutte le variabili sono memorizzate nello **stack**.

Le variabili di tipo primitivo contengono direttamente il valore. Le variabili tipo classe contengono il riferimento dell'oggetto oppure null.

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

5.3 Oggetti lato Utente

Dichiarazione, creazione, inizializzazione:

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

- `<Tipo> <nome>;` Si può solo dichiarare una variabile oggetto per poi crearla e inizializzarla successivamente.
- solo quando si scrive *new* (Keyword di linguaggio) si crea un oggetto dalla classe indicata.
- `var`[7] fa infierire[8] il tipo della variabile locale per allegerire il codice. Se manca l'espressione non va, esempio `var i;`.

[7] Local variable type inference

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

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

6 Classi

Sono template (tipo, struttura in memoria, comportamento) per generare oggetti (istanza).

Le classi hanno un nome (NomeClasse) che sarà anche il nome del tipo per le variabili e del file.

I membri fondamentali di una classe sono:

- **Campi**, descrivono la struttura/stato
- **Metodi**, descrivono i messaggi e il comportamento

```
1 class NomeDiUnaClasse {
2   ...
3   <Campi>
4   ...
5   <Metodi>
6   ...
7 }
```

Definisco le configurazioni.

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

6.1 Campi

Sono lo stato attuale dell'oggetto. Simili hai membri di una struttura C, con la differenza che possono essere 0,1,diversi (5-7max). Simili a variabili (tipo+nome), ma non si può usare *var*. Possono essere valori primitivi o altri oggetti (anche della classe stessa). L'ordine dei non conta.

I campi sono iniziabili alla dichiarazione dell'oggetto (coi parametri), sennò sono inizializzati in base al tipo a **0, false, null**.

Uso dei campi lato utente: Assegnamento ... `obj.campo =` ..., Lettura ... `= obj.campo` ...

6.2 Metodi

Definiscono il comportamento dell'oggetto. Simili a funzioni C. Hanno un intestazione (tipo di ritorno—void, nome, argomenti) e un corpo. I metodi di una classe possono essere 0, 1, diversi.

i metodi possono leggere/scrivere i campi.

Uso dei metodi lato utente ... `obj.metodo()`.... L'invocazione del metodo, corrisponde a inviare un messaggio al receiver (obj nell'esempio) azionando l'esecuzione del corpo del metodo.

```
tipoDiRitorno nomeMetodo([tipo1 arg1, ...]) {
  ...
  [return ...;]
}
```

6.3 La variabile *this*

Variabile contenente il riferimento all'oggetto che sta gestendo il messaggio corrente. Si usa per rendere meno ambiguo il codice accedento tramite *this* a campi o metodi. (Usare sempre). `... this.cmapo ... this.metodo()...`

6.4 Precisazioni

Inizializzazioni particolari degli oggetti Stringa:

1. `... = new String();`, stringa vuota (è diverso da null).
2. `... = "..."`, come in C, comportamento speciale degli oggetti Stringa.