Programmazione orientata agli oggetti

La programmazione orientata agli oggetti (OOP, Object Oriented Programming) è un paradigma di programmazione che prevede di raggruppare in un'unica entità (la classe) sia le strutture dati che le procedure che operano su di esse, creando per l'appunto un "oggetto" software dotato di proprietà (dati) e metodi (procedure) che operano sui dati dell'oggetto stesso.

Un programma viene realizzato progettando e realizzando il codice sotto forma di classi ossia è costituito da oggetti software (istanze di classi) che interagiscono tra di loro.

Per comprendere meglio l'evoluzione che ha portato alla programmazione OO partiamo della definizione di tipo di dato.

Un **tipo di dato** indica l'insieme di valori che una variabile può assumere e le operazioni che si possono effettuare su tali valori.

Ad esempio dichiarare che la variabile X è di tipo "numero intero" significa che X può assumere solo numeri interi e che su tali valori sono ammesse solo certe operazioni.

Ogni linguaggio di programmazione consente di usare, un certo numero di tipi di dati, dando la possibilità di convertire una variabile di un certo tipo in un altro tipo (type casting).

In quasi tutti i linguaggi imperativi è possibile definire dei nuovi tipi partendo dai tipi predefiniti, ad esempio in C attraverso il costrutto typedef.

Ma questi tipi di meccanismo non consentono di definire tipi con la stesso grado di astrazione dei tipi predefiniti nel linguaggio.

Ad esempio consideriamo la seguente definizione in pseudo codice:

```
type Int_Stack = struct {
  int P[100];
  int top;
}

Int_Stack crea_pila() {
  Int_Stack s = new Int_Stack();
  s.top = 0;
  return s;
}

Int_Stack top(Int_Stack s) {
  return s.P[s.top];
}

bool empty(Int_Stack s) {
  return (s.top == 0);
}
```

```
Int_Stack push(Int_Stack s, int k)
  {
  if (s.top == 100) errore();
  s.P[s.top] = k;
  s.top = s.top + 1;
  return s;
}

Int_Stack pop(Int_Stack s) {
  if (s.top == 0) errore();
  s.P[s.top] = k;
  s.top = s.top - 1;
  return s;
}
```

Definendo il tipo Int_Stack si vuole definire una "pila di interi", intendendo che una pila di interi è una struttura dati manipolabile con le operazioni di creazione (crea_pila), inserimento eliminazione ed accesso dell'elemento in cima (push, pop, top) e verifica pila vuota(empty).

Ma il linguaggio non garantisce l'esclusività delle operazioni definite, e nulla impedisce di poter accedere all'elemento s.top-1.

Quindi se il linguaggio fornisce delle astrazioni sui dati (i tipi) che nascondono l'implementazione, lo stesso non è possibile al programmatore.

Per questo alcuni linguaggi permettono di definire dei tipi di dato che si comportano esattamente come i tipi predefiniti, attraverso i **tipo di dato astratto** o **ADT** (Abstract Data Type)

Una possibile definizione di un ADT tipo Int_Stack è la seguente:

```
abstype Int_Stack;
   type Int_Stack = struct {
        int P[100];
        int n; int top;    }

signature
Int_Stack crea_pila();
Int_Stack top(Int_Stack s);
bool empty(Int_Stack s);
Int_Stack push(Int_Stack s, int k
Int_Stack pop(Int_Stack s);

operations
Int_Stack top(Int_Stack s) {
  return s.P[s.top];
}
```

```
Int_Stack crea_pila() {
  Int_Stack s = new Int_Stack();
  s.top = 0;
  return s;
}

Int_Stack push(Int_Stack s, int k)
  {
  if (s.top == 100) errore();
  s.P[s.top] = k;
  s.top = s.top + 1;
  return s;
}

Int_Stack pop(Int_Stack s) {
  if (s.top == 0) errore();
  s.P[s.top] = k;
  s.top = s.top - 1;
  return s;
}
```

Quindi un ADT è un tipo di dato le cui istanze possono essere manipolate con modalità che dipendono esclusivamente dalla semantica del dato e non dalla sua implementazione.

Nella programmazione per tipi di dati astratti, un tipo di dato viene definito distinguendo nettamente la sua **interfaccia** con l'**implementazione interna.**

Ossia le operazioni fornite per la manipolazione del dato con il modo in cui le informazioni sono conservate e il modo in cui le operazioni agiscono al fine di esibire, all'interfaccia, il comportamento desiderato.

La conseguente inaccessibilità dell'implementazione viene spesso identificata con le espressioni **incapsulamento**, detto anche nascondere informazioni (**information hiding**).

Un ADT è caratterizzato da:

1. Un nome per il tipo
2. Un'implementazione (rappresentazione)
3. Un insieme di nomi di operazioni
4. Per ogni operazione un'implementazione dell'operazione che usi la rappresentazione fornita al punto 2
5. Una capsula che separi nomi dei tipi (segnatura) dalla implementazione (realizzazione)

Non è possibile accedere alla struttura dati incapsulata (né in lettura né in scrittura) se non attraverso le operazioni definite su di essa.

L' Interfaccia del ADT, descrive la parte direttamente accessibile dall'utilizzatore mentre la Realizzazione implementa l'ADT.

Il cliente o utilizzatore fa uso del ADT per realizzare procedure di un'applicazione, o per costruire ADT più complessi, il produttore realizza le astrazioni e le funzionalità.

La descrizione della semantica delle operazioni di un ADT viene chiamata specifica, espressa non in termini del tipo concreto , ma per mezzo di relazioni generali astratte.

Una possibile specifica per l'ADT dell'esempio Int_Stack potrebbe essere:

- crea_pila crea una pila vuota
- push inserisce un elemento in testa alla pila
- top restituisce l'elemento in testa alla pila
- pop elimina l'elemento in testa alla pila
- empty è vero se e solo se la pila è vuota

NB Ogni cliente che usi Int_Stack sfruttando solo questa specifica, non noterà nessuna differenza tra le sue definizioni (principio d'indipendenza della rappresentazione)

Una specifica è una sorta di contratto tra l'ADT e i suoi clienti: l'ADT garantisce che l'implementazione delle operazioni soddisfa la specifica

Principio d'indipendenza dalla rappresentazione

Due implementazioni corrette di una stessa specifica di un ADT sono osservabilmente indistinguibili da parte dei clienti di quel tipo

I tipi di dati astratti sono meccanismi della 'programmazione in piccolo': sono pensati per incapsulare un tipo con le relative operazioni.
E' molto più comune tuttavia che un'astrazione sia composta da più tipi tra loro correlate.

I meccanismi linguistici che realizzano questo tipo d'incapsulamento sono in genere chiamati **moduli** o **package**

In alcuni linguaggi (ML, CLU) gli ADT sono visti come tipi predefiniti.

Nel concetto di ADT la semantica di un dato coincide con le operazioni che si possono eseguire su di esso, e da questa idea deriva il paradigma di programmazione algebrica (linguaggio OBJ) in cui i tipi di dati sono completamente definiti da una descrizione algebrica delle loro operazioni.

Il concetto di ADT è alla base della stessa programmazione orientata agli oggetti, in quanto "una classe è l'implementazione di un dato astratto" (Bertrand Meyer, padre del linguaggio Eiffel).

La classe può essere considerata l'erede del tipo di dato astratto,può essere vista come il costrutto che permette di realizzare questa astrazione con un supporto strutturato da parte del linguaggio.

Programmazione OOP

- Si parla di programmazione con oggetti con riferimento alla programmazione basata sul concetto di oggetto (dati+operazioni).
- Si parla di programmazione basata sugli oggetti (object based programming) con riferimento alla programmazione basata sui concetti di:
  - Tipo di dati astratto o Classe (tipo)
  - Oggetto (istanza di un tipo)
- Si parla di programmazione orientata agli oggetti (object oriented programming, OOP) con riferimento alla programmazione basata sui concetti di:
  - Oggetto
  - Classe
  - Ereditarietà
  - Polimorfismo

È possibile adottare tecniche di programmazione con oggetti o basate sugli oggetti anche in linguaggi tradizionali (ad es., in C o Pascal), aderendo ad un insieme di regole, il cui uso però non può essere forzato né verificato dal compilatore.

Il costrutto principale di ogni linguaggio OOP è l'**oggetto**, che può essere definito come una capsula contenente sia dati che operazioni per manipolarli e che fornisca all'esterno un'interfaccia attraverso la quale l'oggetto è accessibile.

Differenze con gli ADT

Gli ADT hanno il vantaggio di riunire in un'unico costrutto sia i dati che i modi legali per manipolarli.
Lo svantaggio è però una rigidità d'uso che si manifesta soprattutto quando si voglia estendere o riusare un'astrazione

Sarebbe preferibile avere un meccanismo automatico col quale ereditare da certe classi l'implementazioni di metodi in altre classi.

Gli oggetti e gli ADT condividono l'idea che i dati devono stare insieme alle operazioni per la loro manipolazione, ma quando si dichiara una variabile di un ADT, la variabile rappresenta solo i dati interni alla variabile ADT.

Viceversa, quando si dichiara una variabile di un oggetto, esso rappresenta sia i dati che le operazioni.

I dati di un oggetto sono detti attributi o variabili di istanza o data member, le operazioni di un oggetto sono chiamate metodi e possono accedere naturalmente ai dati contenuti nell'oggetto.

L'esecuzione di un metodo di un oggetto è invocata mandando all'oggetto un messaggio che consiste nel nome del metodo e nei suoi eventuali parametri.

oggetto.metodo(parametri)

Un aspetto importante è che l'oggetto che riceve il messaggio è anche un parametro (implicito) del metodo invocato, a tale scopo in C++, Java, e C# si usa la parola chiave this, mentre in Smalltalk, Objective-C, e Ruby si usa la parola chiave self.

Naturalmente anche i dati sono accessibili all'esterno con lo stesso meccanismo, sempre che essi non siano privati, alcuni o tutti i dati e metodi possono essere dichiarati inaccessibili dall'esterno.

Una classe è un modello di un insieme di oggetti e stabilisce:

- I suoi dati (quanti, di quale tipo, con quale visibilità)
- Il nome
- La segnatura
- La visibilità e l'implementazione dei suoi metodi

Nei linguaggi OOP, il costrutto classe consente di definire nuovi tipi di dato (astratti) e le relative operazioni sotto forma di operatori o di funzioni (i metodi).
I nuovi tipi di dato possono essere gestiti quasi allo stesso modo dei tipi predefiniti del linguaggio ossia

- si possono creare istanze
- si possono eseguire operazioni su di esse

Un oggetto è una variabile istanza di una classe e lo stato di un oggetto è rappresentato dai valori correnti delle variabili che costituiscono la struttura dati concreta sottostante il tipo astratto.

Esistono due tipi di linguaggi ad oggetti :

- Non tipizzati (Smalltalk)
  - È possibile definire oggetti senza dichiarare il loro tipo
  - In tali linguaggi, gli oggetti sono entità che incapsulano una struttura dati nelle operazioni possibili su di essa
- Tipizzati (C++, Java)
  - E' possibile definire tipi di dati astratti e istanziarli
  - Gli oggetti devono appartenere ad un tipo (astratto)
  - In tali linguaggi, una classe è una implementazione di un tipo di dati astratto. Un oggetto è una istanza di una classe.

Gli oggetti sono creati dinamicamente mediante istanziazione di una classe, ossia viene allocato uno specifico oggetto con la propria struttura.

Esempio in JAVA

```
public class UsoClasse
{ public static void main(String args[])
 { Impiegato[] Staff = new Impiegato[2];
  Staff[0] = new Impiegato("Mario Rossi", 1500);
  Staff[1] = new Impiegato("Aldo Bianchi", 1000);
  System.out.println("Situazione iniziale");  System.out.println(" ");
   for (int i = 0; i < 3; i++)
   {Staff[i].StampaDettagli(); Staff[i].AumentaSalario(5); }
     System.out.println("Situazione dopo aumento"); System.out.println(" ");
    for (int i = 0; i < 3; i++) { Staff[i].StampaDettagli(); }
 }
}
```

```
class Impiegato
{private String Nome;
  private double Salario;
  public Impiegato(String n, double s)
 {Nome = n; Salario = s; }
 public void StampaDettagli()
 {System.out.println("Nome = " + Nome + " - Salario = " + Salario); }
 public void AumentaSalario(double percentuale)
  { Salario = Salario + Salario*(percentuale/100);}
 }
```

Di ogni classe esistono almeno due forme di accesso o viste, quella privata ossia interna alla classe stessa in cui tutti i metodi e dati sono visibili e quella pubblica ossia esterna alla classe che chiamiamo interfaccia in cui solo i metodi e i dati esplicitamente esportati sono visibili.

Una classe può essere vista come l'insieme degli oggetti che sono istanza della classe stessa ossia il tipo associato alla classe.

Il tipo associato alla classe A è un **sottotipo** della classe B quando ogni messaggio compreso dagli oggetti di B è compresso anche dagli oggetti di A.

In questo caso si dice che la classe A è una classe derivata da B, o anche che è una sottoclasse di B.

Ad esempi in java l'istruzione

public class A extends B

crea la classe A come estensione della classe B, e la classe A eredita tutti i metodi della classe B.

Una caratteristica fondamentale del meccanismo delle sottoclassi è il **Polimorfismo o method overriding** ossia la possibilità che una sottoclasse modifichi la definizione di un metodo presente nella superclasse.

I metodi che vengono ridefiniti in una sottoclasse sono detti polimorfi, in quanto lo stesso metodo si comporta diversamente a seconda del tipo di oggetto su cui è invocato.

Le buone regole di programmazione a oggetti prevedono che quando una classe derivata ridefinisce un metodo, il nuovo metodo abbia la stessa semantica di quello ridefinito, dal punto di vista degli utenti della classe.

Il polimorfismo è particolarmente utile quando la versione del metodo da eseguire viene scelta sulla base del tipo di oggetto effettivamente contenuto in una variabile a runtime (invece che al momento della compilazione).

Questa funzionalità è detta binding dinamico (o late binding) richiede un grosso sforzo di supporto da parte della libreria runtime del linguaggio.

Il binding dinamico è supportato dai più diffusi linguaggi di programmazione ad oggetti come il Java o il C++, ma mentre in Java il binding dinamico è usato di default nelle classi polimorfe, il C++ per default non usa il binding dinamico e per utilizzarlo bisogna definire un metodo interessato come virtual.

Esempio in Java di un polimorfismo

```
public class B  {
      public static String buff;
      public static void leggi()
          {buff="ciao";}
      public static  void scrivi(){System.out.print(buff);}}
class A extends B{
      public static  void scrivi(){System.out.print(buff+" mondo\n");}
      public static void main(String[] args) {
            leggi();
            scrivi();
            B.scrivi();
      }
}
```

L'output generato sarà:

ciao mondo

ciao

La **creazione di un oggetto** costa di due azioni distinte:

1. allocare la memoria necessaria
2. Inizializzare i dati.

L' inizializzazione dei dati è compito del costruttore della classe ossia dal metodo in cui si istanzia la classe che normalmente viene denominato con lo stesso nome della classe.

Se la classe è una sotto classe, l'inizializzazione deve prevedere l'inizializzazione anche della superclasse.

Inoltre in molti linguaggi è consentita la possibilità di avere più di un costruttore.

Esempio in JAVA

```
public class UsoClasse
{ public static void main(String args[])
 { Impiegato[] Staff = new Impiegato[2];
  Staff[0] = new Impiegato("Mario Rossi", 1500);
  Staff[1] = new Impiegato("Aldo Bianchi");
  ….. }
}
```

```
class Impiegato
{private String Nome;
  private double Salario;
  public Impiegato(String n, double s)
 {Nome = n; Salario = s; }
 public Impiegato(String n)
 {Nome = n; Salario = 100; }
 ….}
}
```

Una particolare famiglia di classi sono le **classi astratte** o interfacce in cui vengono definite il nome ed il tipo di uno o più metodi senza che se sia data alcuna implementazione.

Queste classi servono come base di partenza per generare una o più classi specializzate aventi tutte la stessa interfaccia di base.

Queste potranno poi essere utilizzate indifferentemente da applicazioni che conoscono l'interfaccia base della classe astratta, senza sapere niente delle specializzate.

Quindi una classe astratta da sola non può essere istanziata, viene progettata soltanto per svolgere la funzione di classe base da cui le classi derivate o figlie possono essere generate.

Prima che una classe derivata da una classe astratta possa essere istanziata essa ne deve implementare tutti i metodi astratti.

Questo processo di astrazione ha lo scopo di creare una struttura base che semplifica il processo di sviluppo del software.

La maggior parte dei linguaggi OOP consentono le classi astratte, in Java una classe astratta si crea in questo modo:

```
abstract class Prova {
    public Prova(...argomenti...){
        // costruttore
    }
public void metodo1() {
        // un metodo banale
    }
}
```

I costruttori vanno necessariamente implementati nelle classi figlie.

```
abstract Prova2 extends Prova {
    public Prova2 (...argomenti...) {
        super(...argomenti...); // richiama il costruttore della super-classe
        // eventuale codice aggiuntivo
    }
    public void metodo1() { // estensione (facoltativa) del metodo metodo1()
        // eventuale codice...
        super.metodo1(); // facoltativo: richiama il metodo metodo1 della super-classe
        // eventuale codice...
    }
```

Naturalmente OOP soffre di grossi problemi di efficienza se applicata a tutto indiscriminatamente, come avviene nel linguaggio Smalltalk che utilizza gli oggetti anche per i tipi primitivi.

Un approccio più pratico e realista è quello adottato da linguaggi come Java e C++ che limitano la creazione di oggetti alle sole entità che il programmatore decide di dichiarare come tali più eventualmente una serie di oggetti predefiniti, per lo più riguardanti il sistema.

In questo modo tali linguaggi restano efficienti, ma l'uso degli oggetti creati richiede più attenzione e più disciplina ed il fatto di ridefinire i metodi ereditati dalle classi base può portare a introdurre errori nel programma se per caso questi sono usati all'interno della classe base stessa.

La OOP offre **vantaggi** in termini di:

- **modularità**: le classi sono i moduli del sistema software;
- **coesione dei moduli**: una classe è un componente software ben coeso in quanto rappresentazione di un'unica entità
- **disaccoppiamento dei moduli**: gli oggetti hanno un alto grado di disaccoppiamento in quanto i metodi operano sulla struttura dati interna ad un oggetto
- **information hiding**: sia le strutture dati che gli algoritmi possono essere nascosti alla visibilità dall'esterno di un oggetto;
- **riuso**: l'ereditarietà consente di riusare la definizione di una classe nel definire nuove (sotto)classi;
- **estensibilità**: il polimorfismo agevola l'aggiunta di nuove funzionalità, minimizzando le modifiche necessarie al sistema esistente quando si vuole estenderlo.

Il rimo linguaggio orientato agli oggetti fu il Simula (1967), seguito negli anni 70 da Smalltalk.

Negli anni 80 sono nate le estensioni orientate ad oggetti del C (C++, Objective C), del Pascal (Turbo Pascal, l'Object Pascal utilizzato nell'ambiente di sviluppo Delphi di Borland), e di altri linguaggi e negli anni ‘90 OOP è diventato il paradigma dominante, per cui gran parte dei linguaggi di programmazione erano o nativamente orientati agli oggetti o avevano una estensione in tal senso.

Oggi i linguaggi più usati, tra quelli che supportano solo il paradigma OO, sono Smalltalk ed Eiffel e tra quelli che supportano anche il paradigma OO sono C++, Java, Perl, Python, C#, Visual Basic.