Programmazione a oggetti

Appunti a cura di *Andrea Abriani (@aabriani), Davide Gagliardi (@davidegagliardi), Andrea Braghiroli*

Fattori qualità software:

Esterni

- Estendibilità: sistema amplia funzionalità che il software aveva in precedenza. Programmazione ad oggetti viene resa più semplice.
- Riusabilità: software e componenti possono essere riutilizzati per un insieme di requisiti che possono essere diversi da quelli precedenti.

Interni:

fattori percepibili al codice della sorgente

- Strutturazione: codice viene scritto in modo tale da astrarre delle singole parti. E' possibile un'individuazione di parti di software indipendenti dalle altre (es. Input/output può essere posizionato in zona differente).
- Modularità: collegata alla strutturazione. Va oltre la struttura e indica che può essere usata in modo diverso e con componenti può raggiungere scopi e requisiti differenti. (Il codice può essere utilizzato per scopi diversi).
- Comprensibilità: software strutturato in modo che corrisponda alla descrizione del dominio.

Caratteristiche della programmazione a oggetti

Key idea: connessione esplicita funzioni-dati (non vi sono più struct con metodi e costruttori)

Connessione esplicita fra funzioni e dati - Incapsulamento

In precedenza, vi era una procedura che lavorava su variabili globali. Ora invece, le funzioni sono connesse ai dati. Questo delimita ciò che le funzioni possono fare. Questo avviene tramite la procedura di incapsulamento. Questa procedura non avviene per la sicurezza dei dati bensì per la visibilità. Solo alcuni metodi e funzioni potranno accedere agli elementi di una determinata classe.

Concetto di classe

Le classi sono legate in gerachie. Una eredita l'altra per creare una struttura.

Si pone enfasi sulla rappresentazione. In caso di errori si andrà a realizzare una struttura sbagliata.
L'enfasi viene spostata da computazione a rappresentazione (più o meno astratta) sul quale il software deve agire

Enfasi sulla rappresentazione

Il software può essere visto come un insieme di scatole, nelle quali entrano ed escono dati.

Viene utilizzato l'UML (*unified modeling language*), uno standard che contiene un insieme di linguaggi grafici per realizzare un diagramma delle classi e degli oggetti, ottenendo una rappresentazione grafica della programmazione.

Oggetti in UML
All'interno dell'UML, la relazione oggetto-classe è simile a elemento-insieme

Classe	Insieme
Oggetto	Elemento

Gli oggetti risultano essere elementi del dominio con vita propria indipendente con identificatore e istanza.

La parte grafica in UML non è arbitraria ma unificata, ovvero vengono utilizzate convenzioni per rappresentare determinati elementi del programma.

Ereditarietà

L'ereditarietà consente di creare classificazioni gerarchiche. E' possibile creare una classe generale che definisce le caratteristiche comuni a una serie di oggetti correlati. La classe può, in seguito, essere ereditata da una o più classi, ognuna delle quali aggiunge alla classe solo elementi specifici.
Si definisce *classe base* la classe ereditata, mentre la classe che "riceve" l'eredità è detta *classe derivata*. Avviene mediante la seguente sintassi:

```
class NomeClasseDerivata : tipoEreditarietà NomeClasseBase{
  // contenuto della classe
  // il tipoEreditarietà può essere public, private, protected
};
```

Quando, ad esempio, il tipo di accesso alla classe base è `public`, tutti i membri `public` della classe base diverranno membri `public` della classe derivata, Lo stesso discorso varrà per i membri `protected`.
Per gli elementi di tipo `private` della classe base, questi non saranno accessibili da parte dei membri della classe derivata.

Una classe derivata può ricevere in eredità elementi di due o più classi base.

Di seguito alcuni esempi di ereditarietà:

```
class A : tipoEreditarietà B{
  public:
    void m4();
}

class B {
  private:
    void m1();
  public:
    void m2();
    void m3();
};

A seconda delle ereditarietà

  A a;
  a.m1();
  a.m2();
  a.m3();

  // Chiamo i metodi m1, m2, m3
```

Ereditarietà di tipo public `class A: public B{};`

- Quando è private in B diventa inaccessibile in A ed inaccessibile dall'esterno

```
main() {

  A a;
  a.m1(); // NO m1 è private in B (inaccessibile da classi derivate à private)
```

```
}

void A::m4(){
  m1(); // NO m1 è private in B e quindi inaccessibile da A
}
```

- Quando è public in B diventa public in A

```
main() {

  A a;
  a.m2(); //OK
}
void A::m4(){
   m2(); //OK accessibile anche per classi derivate
}
```

- Quando è protected in B diventa protected in A

```
main() {
  A a;
  a.m3(); // NO, non posso accedere dall'esterno
}

main() {
  B b;
  b.m3();  // NO
}
void B::m2(){
  m3(); // OK
}
void A::m4(){
  m3(); //OK, se in B qualcosa è protected -> resta inaccessibile dall'esterno, ma
}

IS-A -> ogni istanza di A è anche istanza di B -> EREDITARIETA'
```

Ereditarietà di tipo private `class A: private B{};`

- Quando è private in B diventa inaccessibile da A ed inaccessibile dall'esterno

```
main() {

  A a;
```

```
  a.m1(); //NO
}

void A::m4(){
  m1(); //NO *come nel caso precedente con ereditarietà public*
}
```

- Quando è public in B diventa private in A

```
main() {

A a;
a.m2(); //NO -> privatizzato per l'esterno uguali
}

void A::m4(){
  m2(); //OK -> accessibile anche per classi derivate
}
```

- Quando è protected in B diventa private in A

```
main() {
  A a;
  a.m3();  //NO
}

void A::m4(){
  m3(); //OK
}
```

Ereditarietà di tipo protected `class A: protected B{};`

- Quando è private in B diventa inaccessibile da A ed inaccessibile dall'esterno

```
main() {

  A a;
  a.m1(); //NO
}

void A::m4(){
  m1();  //NO
}
```

- Quando è public in B diventa public in A

```
main() {

  A a;
  a.m2(); //NO à esterno no
}

void A::m4(){
  m2(); //OK -> interno ok
}
```

- Quando è protected in B diventa protected in A

```
main() {
  A a;
  a.m3(); //NO
}

void A::m4(){
  m3(); //OK
}
```

		Base			
		Private		Public	
Derivata	Public	Inaccessibile dall'esterno	Inaccessibile dalla derivata	Inaccessibile dall'esterno	Access dalla derivat
	Private	Inaccessibile dall'esterno	Inaccessibile dalla derivata	Inaccessibile dall'esterno	Access dalla derivat
	Protected	Inaccessibile dall'esterno	Inaccessibile dalla derivata	Inaccessibile dall'esterno	Access dalla derivat

Le 2 righe `private` e `protected` sono uguali per accessibilità: ci sono conseguenze su classi derivate

	Private	Public	Protected
Public	Inaccessibile	Public	Protected
Private	Inaccessibile	Private	Private
Protected	Inaccessibile	Protected	Protected

Quando ha senso fare ereditarietà private/protected?
Ereditare in modo `private` implica la natura della classe base è nascosta dall'implementazione (ereditarietà per implementazione). In sostanza si effettua questo tipo di ereditarietà quando attributi e metodi sono utilizzati per l'implementazione.

La classe derivata quindi non utilizzerà i metodi come la classe sovrastante.

Differenze valore / puntatore / riferimento

- Passaggio per valore: il valore di una variabile viene copiato in una struttura stack. Successivamente la funzione recupera il dato.
- Passaggio per indirizzo: viene copiato l'indirizzo della variabile in modo esplicito. Il passaggio per indirizzo viene utilizzato se la variabile è "grande".
- Passaggio per riferimento (esclusivo del c++): alla funzione viene passato l'indirizzo e non il valore dell'argomento. Questo approccio richiede meno memoria rispetto alla chiamata per valore, e soprattutto consente di modificare il valore delle variabili che sono ad un livello di visibilità (scope) esterno alla funzione o al metodo.

//Conviene passare per indirizzo se la variabile è grande, altrimenti va bene /per valore perché copio solo la variabile.
P. riferimento: la variabile grande passa così per non spostare molti dati nella memoria. Si tende a non fare la copia.

Operatore "++"

Può essere pre-fisso (`++i`) o post-fisso (`i++`).

Nel caso di incremento pre-fisso, la variabile verrà incrementata nello stesso ciclo:

```
int i = 5;
cout << "Il valore di i e' " <<  ++i;

//L'output del programma sarà "Il valore di i è 6"
```

Nel caso di operatore post fisso, la variabile verrà incrementata nel ciclo successivo:

```
int i = 5;
cout << "Il valore di i e' " <<  i++;

//L'output del programma sarà "Il valore di i è 5"
```

Se l'operatore viene applicato ad un puntatore, il valore della cella di memoria rimarrà il medesimo. Verrà però incrementato l'indirizzo di memoria contenuto dal puntatore.

Operatore "new"

L'operatore `new` alloca un puntatore di memoria, in questo modo l'allocazione avverrà in modo dinamico. Bisogna però preoccuparsi di utilizzare il comando `delete`, per non aumentare l'occupazione di memoria da parte del programma.

Costruttori

Quando viene costruita una classe vuota, viene fatta una definizione astratta. Proprio per questo non è scontato che il compilatore allochi memoria (come per il `typedef`).
L'allocazione di memoria avverrà una volta istanziata una classe. In questo modo verrano inseriti in memoria anche un puntatore `this` (che punta a dove la variabile è allocata) e un secondo puntatore verso i metodi di quella classe.

//Punta i costruttori che ci sono sempre (default, 0 parametri, distruttore, costruttore di copia, operatore di assegnazione). Questo è a livello di memoria (lo fa il compilatore).

Header

Si tratta della dichiarazione all'inizio del file con cui si dichiara il nome che il compilatore usa per chiamare il metodo (che è l'unico nel programma).

Costruttore

Funzione membro che si occupa di:

- allocazione della memoria per tutti i suoi membri (dati e funzioni) nello heap o nello stack;
- inizializzazione dei dati membri

Caratteristiche del costruttore sono:

- nome uguale a quello della classe di appartenenza;
- non ha un valore di ritorno

Il costruttore può accettare uno o più argomenti. Nel caso di costruttore a zero parametri, prende il nome di costruttore di *default*. Nel caso fossero assenti i costruttori, il compilatore è in grado di generarne uno automaticamente (con i limiti del caso).

L'uso di un costruttore appositamente definito, invece, consente di inizializzare tutti i membri della classe assieme, di modo che l'istanziazione di un oggetto sia sempre consistente con il nostro modello di dati e indipendente dalle scelte del compilatore.

```
NomeClasse::NomeClasse(){
  cout << "Costruttore di default/zero parametri";
}
NomeClasse::NomeClasse(Nome _nome, Cognome _cognome){
  cout << "Costruttore specifico a due parametri";
  nome = _nome;
  cognome = _cognome;
}
```

Distruttore

Il distruttore di classe assolve il compito di rilasciare tutte le risorse associate ad un oggetto, quando esso non è più necessario.

```
NomeClasse::~NomeClasse(){
  cout << "Il contenuto e' stato distrutto";
}
```

Il distruttore è caratterizzato dal fatto che non ha valori di ritorno né parametri. Per questo motivo non può essere sovraccaricato.

Il suo compito primario dovrebbe essere sempre e solo quella di rimuovere un oggetto e tutte le sue dipendenze dallo stato del programma in maniera sicura e completa.
Viene invocato automaticamente per tutte le variabili, ogni volta che viene raggiunta la fine del loro ambito di visibilità, oppure nel caso di deallocazione tramite il comando `delete`.