### 2.3.2 *Introduzione alla programmazione orientata agli oggetti*

Il termine "orientato agli oggetti" (*object-oriented*) ha recentemente avuto una grandissima diffusione in molti campi dell'informatica. Questo termine viene infatti usato in diversi contesti (ad esempio, basi di dati, ed interfacce utente) per descrivere software di varia natura costruito secondo la filosofia descritta nel precedente paragrafo

È impossibile dare una definizione univoca e precisa di cosa si intenda per orientazione agli oggetti. Informalmente possiamo dire che un programma orientato agli

oggetti è costituito da un insieme di oggetti che interagiscono tra loro, ogni oggetto essendo composto da uno stato interno costituito dai dati associati all'oggetto, ed un insieme di operazioni che modificano tale stato. Il punto centrale della programmazione è quindi l'oggetto, in opposizione alla programmazione tradizionale in cui l'attenzione maggiore è rivolta alle procedure (e funzioni) che manipolano i dati. Più precisamente, si parla di progettazione e programmazione con:

- *orientazione alle funzioni* quando il concetto primario nell'attività di progettazione o programmazione corrisponde ad un'operazione necessaria al processo risolutivo del problema; gli oggetti manipolati dall'operazione sono considerati, in questo caso, parti componenti del concetto;

- *orientazione agli oggetti* quando il concetto primario nell'attività di progettazione o programmazione corrisponde ad una astrazione sugli oggetti della realtà in gioco; questa astrazione è ottenuta classificando gli oggetti stessi sulla base delle operazioni che li manipolano; in questo secondo caso, allora, le operazioni che si applicano agli oggetti di una classe sono parti componenti dell'astrazione.

Per meglio apprezzare le caratteristiche della programmazione agli oggetti, è opportuno descrivere sinteticamente alcuni concetti essenziali che connotano tale filosofia. Nel seguito di questo paragrafo descriveremo sinteticamente tali concetti, che possiamo riassumere in:

- oggetti, classi ed incapsulamento,

- *information hiding* (dall'inglese, nascondere l'informazione),

- ereditarietà.

Ritorneremo su questi concetti quando parleremo delle tecniche di progettazione, ed in particolare di modularizzazione. Prima di iniziare la descrizione, vogliamo però sottolineare che quando si parla di *linguaggi* orientati agli oggetti (ad esempio, EIFFEL, SMALLTALK e C++), ci si riferisce al fatto che tali linguaggi forniscono dei costrutti linguistici per supportare la programmazione orientata agli oggetti. Rimane però valida l'osservazione che molte delle caratteristiche di un programma dipendono dal metodo con cui è stato progettato, piuttosto che dal linguaggio in cui viene espresso: è ad esempio possibile scrivere programmi con orientazione agli oggetti in linguaggi quali il PASCAL (che non supporta specifici costrutti per gli oggetti) e scrivere programmi assolutamente non orientati agli oggetti in C++.

***Oggetti, classi ed incapsulamento***

Il principio fondamentale dell'orientazione agli oggetti è l'*incapsulamento*. Col termine incapsulare intendiamo l'assemblamento in un'unica entità (l'oggetto) di dati e operazioni. In che forma i dati e le operazioni vengono incapsulati negli oggetti dipende dallo specifico linguaggio di programmazione utilizzato.

In C++ l'incapsulamento viene supportato tramite il costrutto di *classe*. Una classe è un tipo di dato, simile ad un tipo record, i cui valori sono oggetti descritti mediante un insieme di campi, dove ciascun campo rappresenta

- o una proprietà associate all'oggetto, specificata mediante il valore di un dato,

- oppure una funzione applicabile sull'oggetto.

Consideriamo la seguente dichiarazione di classe.

```
class NumeroComplesso
{ public:
   float re;  // parte reale
   float im;  // parte immaginaria
   float Modulo() { return sqrt(re * re + im * im); }
};
```

La classe `NumeroComplesso` è un tipo. Ogni oggetto di questo tipo ha due proprietà descritte rispettivamente dai dati `re` e `im` ed una operazione descritta dalla funzione `Modulo()`.

Dato un oggetto della classe, possiamo fare riferimento ai suoi dati e alle sue operazioni come mostrano le seguenti istruzioni.

```
NumeroComplesso c;  // dichiarazione dell'oggetto c
c.re = 4;           // riferimento al dato re associato a c
c.im = 3;           // riferimento al dato im associato a c

cout << c.Modulo(); // la funzione Modulo() viene invocata
                    // sull'oggetto corrente c: l'effetto
                    // e' la stampa del valore 5

cout << Modulo();   // errore: la funzione Modulo()
                    // e' incapsulata negli oggetti della classe
                    // NumeroComplesso ed e' quindi attivabile
                    // solo facendo riferimento ad un oggetto di
                    // di tale classe
```

Una volta definita la classe `NumeroComplesso`, abbiamo anche determinato completamente le operazioni che possono essere effettuate sugli oggetti della classe. Questo è un aspetto importante dell'incapsulamento: esso consente di aggiungere al linguaggio nuovi tipi di dato, definiti non solo attraverso una definizione della struttura del dominio, ma anche dall'insieme di operazioni che possono essere invocate sui valori del tipo stesso. Nel capitolo 4 descriveremo in dettaglio le classi in C++.

### *Information hiding*

Un altro principio fondamentale della programmazione orientata agli oggetti è l'information hiding, che è strettamente legato all'incapsulamento, e che si basa sull'idea che al fine di utilizzare un oggetto, è sufficiente conoscere la sua interfaccia verso l'esterno, mentre non è necessario conoscere gli aspetti legati alla sua realizzazione. In generale, anzi, è vantaggioso nascondere tale realizzazione, al fine di evitare un cattivo uso della stessa.

Riprendiamo ancora l'esempio dei numeri complessi; l'utente della classe non è tenuto a sapere che il numero complesso è rappresentato tramite parte reale e parte immaginaria. Quello che invece deve conoscere sono le funzioni per ottenere le grandezze di interesse associate ad un numero complesso (ad esempio, modulo, fase, parte reale e parte immaginaria).

Il C++ realizza l'information hiding attraverso l'utilizzo di due livelli di accesso ai campi di una classe. I campi di tipo *pubblico* (`public`) sono accessibili a funzioni esterne alla classe, mentre quelli *privati* (`private`) sono utilizzabili solo dalle funzioni proprie della classe. Ad esempio, possiamo raffinare la definizione della classe `NumeroComplesso` nel seguente modo:

```
class NumeroComplesso
{ public:
    float Modulo()      { return sqrt(re * re + im * im); }
    float Fase()        { if (im >= 0) return acos(re/Modulo());
                          else return -acos(re/Modulo()); }
    float Reale()       { return re; }
    float Immaginaria() { return im; }
    // ... altre funzioni per la manipolazione dei dati
  private:
    float re;  // parte reale
    float im;  // parte immaginaria
};
```

Una funzione esterna potrà ora utilizzare le funzioni pubbliche `Modulo()`, `Fase()`, `Reale()` e `Immaginaria()`, ma non i dati `re` ed `im`. Ad esempio, il seguente frammento di programma è scorretto

```
NumeroComplesso c;
cout << c.Reale(); // corretto
cout << c.Re;      // scorretto: Re e' un campo privato
```

L'information hiding si basa sul principio che l'utilizzo di oggetti della classe prescinde dalla rappresentazione dei dati. Ad esempio, se consideriamo la seguente funzione che calcola il quadrante di un numero complesso,

```
int Quadrante(NumeroComplesso c)
{ if (c.Reale()==0 || c.Immaginaria()==0)
    return 0; // sugli assi
  if (c.Reale() > 0)
    if (c.Immaginaria() > 0)
      return 1;
    else
      return 4;
  else if (c.Immaginaria() > 0)
    return 2;
  else
    return 3;
}
```

possiamo notare che il codice della funzione sarebbe stato esattamente valido se nella classe i campi privati fossero stati dichiarati in modo diverso, ad esempio come:

```
class NumeroComplesso
{ public:
    float Modulo()      { return m; }
    float Fase()        { return phi; }
    float Reale()       { return m * cos(phi); }
    float Immaginaria() { return m * sin(phi); }
    // ... altre funzioni per la manipolazione dei dati
  private:
    float m;    // modulo
    float phi;  // fase
};
```

in cui abbiamo cambiato la rappresentazione, passando alle coordinate polari. Con questa nuova dichiarazione, il corpo delle funzioni della classe viene ovviamente modificato, ma il corpo delle funzioni esterne, quali `Quadrante()`, rimane inalterato.

*Ereditarietà*

L'ereditarietà è un insieme di meccanismi che consentono di attribuire alcune proprietà ad una classe non definendole esplicitamente, ma derivandole da altre classi più generali. Questi meccanismi permettono di definire una classe esprimendo unicamente le differenze che essa ha rispetto ad una o più classi già definite, invece che specificandone completamente le funzioni e i dati. In questo caso, si dice che la classe viene *derivata* da un'altra classe (o da altre classi).

Quando una classe `D` è derivata da un'altra classe `B`, ogni oggetto di `D` è anche un oggetto di `B`, e quindi le proprietà definite per `B` vengono ereditate da ogni oggetto di `D` senza bisogno di dichiararle esplicitamente. In altre parole, la dichiarazione di una classe come derivata da un'altra classe fa sì che per gli oggetti della prima siano disponibili, oltre ai dati e le funzioni proprie, anche quelli dalla classe da cui è derivata.

In C++, la notazione per dichiarare una classe `D` come derivata da un'altra classe `B` è la seguente

```
class D : public B
{
  // .....
};
```

Se ad esempio volessimo considerare una particolare rappresentazione grafica dei numeri complessi potremmo utilizzare una classe, chiamata `NumeroComplessoGrafico`, derivata dalla classe `NumeroComplesso`, che utilizza una funzione di visualizzazione specifica.

```
class NumeroComplessoGrafico: public NumeroComplesso
{public:
  void Visualizza();
};
```

Gli oggetti della classe `NumeroComplessoGrafico` avranno gli stessi dati e le stesse funzioni degli oggetti della classe `NumeroComplesso` oltre alla funzione `Visualizza()`.

Ad esempio, la seguente funzione visualizza e restituisce il maggiore (in modulo) tra due numeri complessi.

```
void VisualizzaMassimo(NumeroComplessoGrafico c1,
                       NumeroComplessoGrafico c2)
{ if (c1.Modulo() > c2.Modulo())
    c1.Visualizza();
  else
    c2.Visualizza();
}
```

Si noti che la funzione `Modulo()` non è propria della classe `NumeroComplessoGrafico`, bensì viene ereditata dalla classe `NumeroComplesso`.

L'ereditarietà può essere utilizzata come base per una metodologia di programmazione in cui i tipi sono organizzati in una gerarchia concettuale. Se un tipo corrisponde ad un certo concetto nel dominio che si sta modellando, usiamo l'ereditarietà per descrivere tipi che rappresentino dei concetti più specifici del concetto espresso dal primo tipo. Tra il tipo che eredita (*sottotipo*) ed il tipo da cui proviene l'eredità (*tipo base*) esiste un legame di tipo *is-a* (letteralmente "è-un"), che sta a significare che un oggetto del sottotipo è anche un oggetto del tipo base.

Nel capitolo 5 riprenderemo in concetto di ereditarietà in C++ e discuteremo in dettaglio il concetto di derivazione tra classi. Nel capitolo 13 illustreremo poi vari modi di utilizzare l'ereditarietà per la realizzazione di tipi astratti in C++.