Introduzione all'Object Oriented Programming

Leggere sez. 1.3 di Programmazione di base e avanzata con Java

Una domanda fondamentale

Che cosa significa realizzare del buon software?

La risposta del programmatore C

“Realizzare del buon software significa ottimizzare ogni istruzione, in modo da ottenere il codice più compatto ed efficiente possibile”

La risposta del programmatore Visual Basic

“Realizzare del buon software significa fornire le funzionalità richieste dall’utente nel minor tempo possibile e col minor costo possibile, indipendentemente da come si arriva al risultato”

La risposta dell'ingegnere del software

- Scrivere del buon software significa trovare il **miglior equilibrio** fra diversi fattori:
  - La **soddisfazione dell'utente**
  - La **facilità di estensione** del programma
  - La **comprensibilità** delle soluzioni adottate
- Significa adottare tecniche adeguate a **gestire la crescente complessità** delle applicazioni
- Significa fare in modo che l'investimento, spesso ingente, necessario per produrre un'applicazione venga utilizzato al meglio, garantendo in particolare:
  - Il maggior **tempo di vita** possibile
  - La possibilità di **riutilizzare** in altri progetti parte del codice prodotto

Limiti del modello procedurale

- Il modello procedurale è basato su un dualismo di fondo
- Un programma è composto da due tipi di entità: **strutture dati** e **procedure**
- **Strutture dati**: entità passive che contengono informazioni
- **Procedure**: entità attive che modificano le informazioni
- Il problema è che queste due entità sono scollegate fra di loro: se dichiariamo una variabile globale, tutte le procedure di un'applicazione possono modificarla senza controllo: **manca il concetto di protezione dei dati**
- In un'applicazione complessa il fatto che in ogni punto si possa modificare qualunque dato può facilmente generare una situazione incontrollabile

Modularizzazione e decomposizione

- Lavorare in questo modo è come costruire un computer mettendo tutti i componenti su una sola scheda e dare la possibilità di collegare un componente con tutti gli altri senza alcuna limitazione
- L'industria dell'hardware è riuscita a costruire computer sempre più complessi perché ha adottato un concetto di **modularizzazione** a più livelli:
  - I singoli componenti sono racchiusi in circuiti integrati
  - I circuiti integrati vengono montati su schede
  - Le schede vengono montate su rack per comporre il computer
- Un problema complesso come la costruzione di un calcolatore diventa gestibile perché il problema viene **decomposto** in problemi più semplici: costruzione di una scheda, di un circuito integrato…

Astrazione

- Una struttura organizzata a livelli consente di concentrarsi solo sugli aspetti importanti ignorando i dettagli che sono già stati affrontati e risolti ai livelli inferiori
- Quando progettiamo una scheda ci concentriamo solo sui meccanismi di collegamento tra i vari circuiti integrati e ignoriamo completamente i dettagli di progettazione dei circuiti integrati
- Quando progettiamo un computer ci concentriamo solo su come comporre fra di loro le schede e ignoriamo completamente i dettagli di progettazione delle schede
- Questo meccanismo prende il nome di **astrazione** ed è un potente strumento per gestire la complessità

Interfaccia e implementazione

- Un circuito integrato nasconde il suo contenuto ed espone solo un numero limitato di piedini
- Si può sostituire un circuito con un altro purché i segnali elettrici ai piedini rimangano gli stessi
- Si ha quindi una separazione fra quello che un componente è in grado di fare (**interfaccia**) e come lo fa (**implementazione**)
- L'implementazione è un dettaglio interno che non influisce sul comportamento del sistema
- Lo stesso succede tra computer e periferiche: noi possiamo collegare ad un computer una qualsiasi stampante perché è stata definita un'interfaccia standard e praticamente tutte le stampanti, indipendentemente dalla tecnologia costruttiva, si adeguano a questa interfaccia

Evoluzione del modello procedurale

- Anche lavorando in C, quindi con il modello procedurale, ci si è presto resi conto che era necessario operare in modo modulare
- Nel tempo si sono quindi diffuse pratiche di progettazione e di programmazione basate sui concetti che abbiamo appena esposto
- Un bell'esempio di questo approccio è costituito dalla gestione dei file nella libreria standard del C
- **Nella gestione dei file troviamo una serie di concetti estremamente interessanti**

I file in C: esempio

```
/* Dichiariamo variabili di tipo puntatore a FILE
   (riferimenti) */
   FILE *f1, *f2;
/* Con fopen apriamo i file: il sistema alloca risorse,
   "crea" una struttura per gestire il file e ci restituisce
   un puntatore (riferimento) */
   f1 = fopen("pippo.txt","r");
   f2 = fopen("pluto.txt","w");
/* Ottenuti i riferimenti li utilizziamo in tutte le
   operazioni successive */
   fscanf(f1,...);
   fprintf(f2,...);
/* Quando non ci servono più chiudiamo i file e il sistema
   libera le risorse allocate: i file "cessano di esistere" */
   fclose(f1);
   fclose(f2);
```

I file in C: astrazione

- Non abbiamo nessuna idea di come sia fatta una struttura dati di tipo FILE
- Abbiamo a disposizione una serie di funzioni (**primitive**) che ci permettono di operare sul file senza sapere come è fatto
- La struttura dati FILE potrebbe cambiare completamente e il nostro programma non ne risentirebbe minimamente
- C'è una netta **separazione fra interfaccia e implementazione**
- Non ci occupiamo minimamente dei dettagli implementativi ma ci concentriamo su un'astrazione

I file in C: dinamicità

- I file vengono gestiti in modo **dinamico**
- Non hanno un tempo di vita limitato e "automatico" come le variabili locali
- Non esistono per tutta la durata del programma come le variabili globali
- Vengono invece **creati** con **fopen** e **distrutti** con **fclose**
- Il loro tempo di vita viene quindi gestito da chi li utilizza
- All'atto della "creazione" viene allocata un'area di memoria che viene liberata nel momento della "distruzione"
- Come tutto il resto, anche l'allocazione/deallocazione della memoria viene gestita dalle primitive e questo garantisce la consistenza

I file in C: istanze, stato e comportamento

- Grazie al meccanismo di creazione/distruzione è possibile lavorare in contemporanea con più file
- E' sufficiente dichiarare più variabili di tipo riferimento a FILE e invocare fopen() più volte memorizzando il valore di ritorno nelle diverse variabili
- Tutte le altre primitive prevedono come primo parametro un riferimento a FILE, e quindi è possibile operare indipendentemente sui vari file aperti
- Quindi: abbiamo diverse entità con un **comportamento** comune (determinato dalle primitive) ma con uno **stato** diverso (ogni variabile lavora su un file diverso, posizionato su una riga diversa ecc.)
- Abbiamo più **istanze** e l'insieme costituito dal **tipo** FILE e dalla **funzione di creazione** fopen() costituisce una "matrice" che permette di creare queste istanze

Oltre il modello procedurale

- Tutti questi meccanismi permettono di lavorare in maniera modulare e di costruire applicazioni più robuste e meglio organizzate
- Purtroppo i linguaggi procedurali non mettono a disposizione strumenti che obblighino o anche solo incoraggino a lavorare in questo modo
- Tutto è lasciato alla disciplina e all'esperienza di chi scrive le librerie e le applicazioni
- La separazione di fondo tra strutture dati e procedure rende difficoltoso operare correttamente
- Per gestire la complessità bisogna disporre di linguaggi che supportino in modo naturale uno stile di programmazione corretto
- Bisogna passare dal **modello procedurale** a quello **orientato agli oggetti**

Programmazione orientata agli oggetti
Il modello “classico”

Il concetto di oggetto

Un oggetto è un'entità dotata di stato e di comportamento

- In pratica un oggetto aggrega in un entità unica e indivisibile una struttura dati (**stato**) e l'insieme di operazioni che possono essere svolte su di essa (**comportamento**)
- E' quindi un'insieme di variabili e di procedure: le variabili vengono comunemente chiamate **attributi** dell'oggetto
- Le procedure vengono comunemente chiamate **metodi**
- **Sparisce il dualismo di fondo del modello procedurale**

Incapsulamento e astrazione

- Lo stato di un oggetto:
  - Non è accessibile all'esterno
  - Può essere visto e manipolato solo attraverso i metodi
- Quindi: lo stato di un oggetto è **protetto**

  → Il modello ad oggetti supporta in modo naturale l'incapsulamento
- Dal momento che dall'esterno possiamo vedere solo i metodi c'è una separazione netta tra cosa l'oggetto è in grado di fare e come lo fa
- Abbiamo quindi una separazione netta fra **interfaccia** e **implementazione**

  → Il modello ad oggetti supporta in modo naturale l'astrazione

Classi e oggetti - 1

- Per poter operare con gli oggetti abbiamo bisogno di un meccanismo che ci consenta di:
  - **Definire una struttura e un comportamento**
  - **Creare un numero qualunque di oggetti con identica struttura e comportamento ma con identità diversa**
- Nella programmazione procedurale abbiamo il concetto di **tipo**: una volta definito un tipo possiamo dichiarare più variabili identiche fra di loro
- Se vogliamo avere un comportamento dinamico ci serve anche un meccanismo di **creazione**, che si comporti come fopen() per i file in C
- Nell'OOP questi due ruoli vengono svolti da un'entità chiamata **classe**

Classi e oggetti - 2

- Quindi una **classe** è un'entità che permette di
  - Definire la struttura e il comportamento di un oggetto
  - Creare un numero qualunque di oggetti con la struttura specificata
- Gli oggetti creati da una classe si chiamano **istanze** della classe

  → **Ogni oggetto è istanza di una qualche classe**
- Tutte le istanze di una classe hanno la **stessa struttura** (lo stato è fatto nello stesso modo) e lo **stesso comportamento** (l'insieme dei metodi definiti dalla classe)
- Ogni istanza possiede un **proprio stato** ("contenuto delle variabili") e una **propria identità.**

Classi e oggetti - 3

Nel modello "classico" dell'OOP esistono solo classi e oggetti

- Gli oggetti sono entità **dinamiche**:
  - Possono essere creati in qualunque momento a partire da una classe
  - Vengono distrutti quando non servono più
- Le classi sono invece entità **statiche**:
  - Sono sempre disponibili durante l'esecuzione di un'applicazione
- Quando si crea un oggetto si ottiene un **riferimento** all'istanza appena creata
- I riferimenti rappresentano l'unico modo per comunicare con gli oggetti