APPUNTI SULLA PROGRAMMAZIONE AD OGGETTI
Prof. Francesco Taurisano

Indice

1. Paradigmi di programmazione
Per *paradigma di programmazione* si intende l'insieme di idee a cui ci si ispira per modellare e per risolvere i problemi. Qualsiasi problema, risolvibile con un certo paradigma, lo è anche con un altro; evidentemente, certi problemi sono particolarmente adatti per essere risolti con specifici paradigmi.
I principali paradigmi di programmazione sono:

- *Paradigma imperativo:* si basa sull'idea della sequenza ordinata di passi, e sull'istruzione di assegnamento che serve per cambiare il valore delle variabili. Il paradigma imperativo è molto vicino al modo di funzionamento dell'elaboratore: è molto efficiente per un uso generale e non è riservato a specifici problemi. (Es. Fortran, Cobol, Pascal, Basic, C)
- *Paradigma logico:* utilizza un approccio dichiarativo per la soluzione dei problemi. Ogni problema è definito attraverso delle asserzioni logiche di fatti e regole (dichiarazioni). Ponendo delle interrogazioni, il sistema è in grado di eseguire deduzioni sulla base dei fatti e delle regole note (base della conoscenza). Mediante il paradigma logico, il programmatore si limita a specificare solo il risultato da raggiungere, mentre il controllo computazione è demandato all'elaboratore. Il paradigma logico viene particolarmente utilizzato per applicazioni inerenti sistemi di intelligenza artificiale. (Es. PROLOG)
- *Paradigma funzionale:* ha come base l'uso delle funzioni, intese in senso matematico, per la risoluzione dei problemi. Le funzioni ricevono in input certi valori e restituiscono dei risultati; inoltre, si può sfruttare la composizione funzionale: ovvero il risultato di una funzione può essere un input per un'altra funzione. Un programma rappresenta, dunque, un insieme di definizioni di funzioni. I linguaggi funzionali furono introdotti con l'obiettivo di rendere agevole la manipolazione di informazioni simboliche, cioè di informazioni di tipo non numerico. I linguaggi funzionali sono particolarmente adatti per architetture parallele. (Es. LISP)
- *Paradigma orientato agli oggetti:* analizza il problema vedendo gli oggetti reali come composti da uno stato, che è modificabile, e da un insieme di operazioni. Più oggetti comunicano tra loro scambiandosi dei messaggi. (Es. CLOS, Simula67, SmallTalk, Eiffel, C++, Java)

2. Programmare ad oggetti
La metodologia orientata agli oggetti definisce un modo di pensare al problema in termini di sistema. Mentre la programmazione strutturata è legata ad una visione del programma in sottoprocedure, la programmazione orientata agli oggetti (*object-oriented*) si focalizza sui dati, che in questo contesto assumo il nome di oggetti. Gli oggetti rappresentano le entità del sistema. Un oggetto è in grado di memorizzare le informazioni che riguardano il suo stato; è anche possibile associare ad un oggetto un insieme di operazioni che esso può compiere. Il programma si sviluppa attraverso le interazioni tra gli oggetti: durante l'esecuzione del programma gli oggetti possono cambiare il loro stato e possono richiedere l'esecuzione di operazioni associate ad altri oggetti. Questo stile di programmazione procura diversi vantaggi: facilità di comprensione e manutenzione del software, robustezza e riusabilità del codice.

3. Gli oggetti
3.1 Caratteristiche e comportamenti
L'elemento base della programmazione ad oggetti è l'oggetto. Un oggetto è definito elencando sia le sue *caratteristiche*, sia il modo con cui interagisce con l'ambiente esterno, cioè i suoi *comportamenti*. Le caratteristiche rappresentano gli elementi che caratterizzano l'oggetto, utili per descrivere le sue proprietà e definirne il suo stato. I comportamenti rappresentano le funzionalità che l'oggetto mette a disposizione: chi intende utilizzare l'oggetto deve attivare i comportamenti dell'oggetto stesso.
(Esempio: oggetto automobile;
alcune caratteristiche: velocità, colore, numero di porte, livello del carburante, posizione della marcia;
alcuni comportamenti: avviati, accelera, fermati, gira, cambia marcia, rifornisciti)

3.2 Attributi e metodi
In termini più formali, un oggetto è completamente descritto quando vengono definiti i suoi *attributi* e i suoi *metodi*.

Le informazioni che servono per descrivere l'oggetto e che corrispondono alle caratteristiche dell'oggetto si chiamano attributi. Gli attributi rappresentano la memoria dell'oggetto e consentono di tenere traccia dello stato dell'oggetto. In ogni istante, un oggetto è caratterizzato da certi valori associati ai suoi attributi. Nella pratica di programmazione, gli attributi sono individuati dalle variabili, che vengono utilizzate dall'oggetto per memorizzare i suoi stati.
Le operazioni che un oggetto è in grado di compiere e che corrispondono ai comportamenti dell'oggetto si chiamano metodi. Attraverso i metodi un oggetto può accedere alla sua memoria (gli attributi) per compiere le operazioni e modificare il suo stato. Solitamente le operazioni vengono invocate da altri oggetti. Nella pratica di programmazione, i metodi vengono realizzati usando le funzioni. I metodi possono avere dei parametri e fornire valori di ritorno, così come avviene per le funzioni.
Un oggetto è quindi costituito dalla coppia attributi + metodi. Gli attributi descrivono le proprietà statiche, cioè le caratteristiche che definiscono lo stato dell'oggetto; i metodi descrivono, invece, le proprietà dinamiche dell'oggetto, cioè i comportamenti messi a disposizione.

3.3 Interazione tra oggetti
Un programma ad oggetti è caratterizzato dalla presenza di tanti oggetti che comunicano e interagiscono tra loro mediante scambi di messaggi.
Un oggetto può interagire con un altro oggetto per diversi motivi: per modificarne lo stato, per richiedere informazioni, per attivare un comportamento.
Un *messaggio* è costituito da tre parti:

- destinatario: corrisponde all'oggetto verso il quale il messaggio è indirizzato;
- selettore: identifica il metodo che si vuole attivare (scelto tra i metodi messi a disposizione dell'oggetto destinatario)
- elenco di argomenti: è l'insieme dei parametri che vengono passati all'oggetto destinatario quando si richiede l'attivazione del metodo.

Esempio: invio del messaggio accelera all'oggetto automobile: automobile.accelera().
Osservazione 1: l'attivazione del metodo selettore può restituire all'oggetto chiamante un valore di ritorno.
L'insieme dei messaggi, che consentono l'interazione con l'oggetto, rappresenta la sua interfaccia verso l'esterno. Così, chi utilizza l'oggetto può conoscere solo la sua interfaccia, e non necessariamente i dettagli implementativi dei singoli metodi.
Osservazione 2: per leggere o modificare i valori degli attributi di un oggetto si può inviare un messaggio all'oggetto. Oppure è possibile riferirsi direttamente agli attributi di un oggetto senza utilizzare un metodo come intermediario.
Esempio: modifica dell'attributo velocità relativo all'oggetto automobile:
automobile.velocità = 50.
La definizione degli attributi e dei metodi di un oggetto può essere effettuata, a seconda della visibilità voluta, in due sezioni: nella *sezione pubblica* per garantire la visibilità di attributi e metodi verso l'esterno; nella *sezione privata* per non rendere accessibili all'esterno attributi e metodi, che vengono usati solo dall'oggetto.

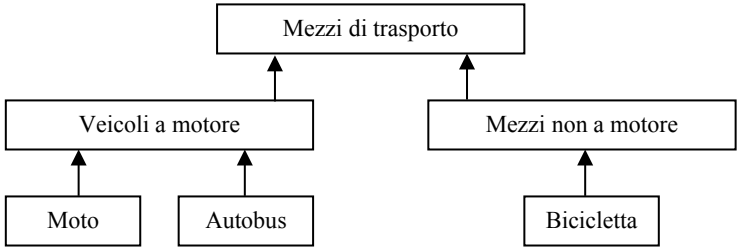
4. Le classi
4.1 Definizione di classe
La classe rappresenta un modello per descrivere un insieme di oggetti. Una classe specifica gli attributi, senza indicarne il valore, e i metodi che devono avere gli oggetti che appartengono alla classe. Un oggetto non può esistere se prima non viene creata una classe a cui l'oggetto deve appartenere.
La classe può essere immaginata come uno stampo dal quale vengono creati più oggetti, tutti con gli stessi attributi e gli stessi metodi.
Gli oggetti creati a partire da una classe vengono chiamati *istanze* della classe.
Una classe può essere rappresentata graficamente attraverso il *diagramma delle classi*, che rappresenta l'elenco generale degli attributi e dei metodi.
Esempio:

< Nome Classe >
< Attributo 1 > < Attributo 2 > ……
< Metodo 1 > < Metodo 2 > ……

Esistono particolari classi che non possono dare origine ad oggetti, ossia che non possono essere istanziate. Queste classi sono chiamate *classi astratte*. Esse hanno almeno un metodo che non è stato implementato, ma è soltanto definito. Le classi astratte sono usate per elencare tutte le operazioni che un'intera gerarchia di classi vuole fornire. Sarà compito delle classi più specializzate implementare i metodi lasciti indefiniti.

4.2 Ereditarietà
E' possibile costruire una classe a partire da un'altra già esistente. Questo concetto è noto col nome di *ereditarietà*.
Più precisamente, quando una classe è creata col meccanismo di ereditarietà, a partire da un'altra classe, essa riceve in eredità gli attributi e i metodi pubblici della classe generatrice.
La classe che è stata derivata da un'altra usando l'ereditarietà prende il nome di *sottoclasse*. La classe generatrice di una sottoclasse si chiama *sopraclasse*.

Queste relazioni tra classi individuano una *gerarchia* che nasce da un processo si specializzazione: le classi che si trovano in cima alla gerarchia sono le più generali, e man mano che si scende si trovano le classi più specializzate. La gerarchia delle classi può essere descritta graficamente attraverso il *grafo di gerarchia*.
Esempio:



La sottoclasse eredita dalla sopraclasse tutti gli attributi e tutti i metodi, evitando di ripetere la descrizione degli elementi comuni nelle sottoclassi. La nuova classe si differenzia dalla sopraclasse per due motivi:

- per *estensione*: quando la sotto classe aggiunge nuovi attributi e nuovi metodi che si sommano a quelli ereditati;
- per *ridefinizione*: quando la sottoclasse ridefinisce i metodi ereditati. Ossia, si crea un nuovo metodo che ha lo stesso nome del metodo ereditato da una sopraclasse, ma che ha una funzionalità diversa.

La ridefinizione di un metodo (*overriding* del metodo), per una sottoclasse, viene impiegata per associare un comportamento diverso o per utilizzare un algoritmo più efficiente.
Nel caso delle classi astratte, per le sottoclassi generate è possibile implementare i metodi lasciati indefiniti; in questo modo le sottoclassi possono essere usate per generare gli oggetti.
Esistono due tipi di ereditarietà:

- *eredità singola*: una sottoclasse deriva da un'unica sopraclasse;
- *eredità multipla*: una sottoclasse deriva da due o più sopraclassi. In questo caso si fondono insieme gli attributi e i metodi provenienti da classi diverse creandone una nuova.

Osservazione: l'ereditarietà è uno strumento che consente il riutilizzo del software creato.

4.3 Polimorfismo
Il termine polimorfismo indica la possibilità per i metodi di assumere forme (implementazioni) diverse all'interno della gerarchia delle classi.
Ossia, considerando una relazione di ereditarietà, le sottoclassi hanno la possibilità di ridefinire i metodi ereditati, mantenendone lo stesso nome.

Un tipo particolare di polimorfismo riguarda il caso in cui all'interno di una stessa classe siano inseriti più metodi che hanno lo stesso nome, ma che si differenziano per il diverso numero e tipo di parametri che vengono passati. In questo caso, quando il metodo viene richiamato, la scelta corretta del metodo da eseguire viene fatta verificando il numero e il tipo dei parametri. Questa situazione è nota come sovracaricamento (*overloading*) dei metodi.