Guida programmazione orientata agli oggetti

A. Introduzione

1. Premessa

Lo scopo principale di questa guida è quello di fornire una panoramica sulla tecnica di programmazione più diffusa e consolidata nel mondo dello sviluppo applicativo: La Programmazione ad Oggetti (Object Oriented Programming, da cui l'acronimo OOP, che utilizzeremo frequentemente nel corso della guida).
Non si richiedono particolari conoscenze pregresse nel mondo della programmazione per poter leggere le pagine che seguono, in quanto gli argomenti trattati vengono esaminati partendo da nozioni decisamente basilari.
Pertanto la guida è rivolta sia a coloro che non hanno mai utilizzato alcun linguaggio di programmazione "ad Oggetti" sia a coloro che hanno già conoscenza o esperienza in tale settore. Questi ultimi potranno approfittare della guida per chiarire meglio alcuni concetti che spesso vengono fraintesi o che sono talvolta fonte di confusione.
Una cosa importante da non trascurare per i tanti programmatori con conoscenza di linguaggi procedurali come il Visual Basic (fino alla versione 6.0, per lo meno) è di evitare fortemente di utilizzare il modus operandi e le regole implementative apprese con tale linguaggio, in quanto potrebbero rendere più arduo l'apprendimento dei concetti che governano il mondo della programmazione ad oggetti.

2. Introduzione e cenni storici

La Programmazione ad Oggetti rappresenta, senza dubbio, il **modello di programmazione più diffuso** ed utilizzato degli ultimi dieci anni.
Le **vecchie metodologie** come la programmazione strutturata e procedurale, in auge negli anni settanta, sono state lentamente ma inesorabilmente superate a causa degli innumerevoli vantaggi che sono derivati dall'utilizzo del nuovo paradigma di sviluppo.
Un esempio, ben noto a tanti programmatori, è rappresentato dalla profonda trasformazione che ha subito il **Visual Basic** nell'ultima versione rilasciata dalla Microsoft (Visual Basic .NET) che lo vede finalmente catalogato come un linguaggio ad Oggetti a tutti gli effetti.
Eppure, contrariamente a quanto si possa pensare, le origini della Programmazione ad Oggetti sono abbastanza remote:**i primi linguaggi "ad oggetti"** furono il **SIMULA I** e il **SIMULA 67**, sviluppati da Ole-Johan Dahl e Kristen Nygaard nei primi anni '60 presso il Norwegian Computing Center.
Entrambi questi linguaggi adottavano già parecchie delle peculiarità che sono oggi tra i capisaldi della programmazione ad oggetti, come ad esempio le classi, le sottoclassi e le funzioni virtuali ma, a dispetto dell'importanza storica, tali linguaggi **non riscossero particolare successo**.
Negli anni '70 fu introdotto **il linguaggio SmallTalk**, coinsiderato da molti il primo vero linguaggio ad oggetti "puro".
Lo SmallTalk fu sviluppato in due periodi successivi: inizialmente da Alan Kay, ricercatore dell'università dello Utah e successivamente fu ripreso da Adele Goldberg e Daniel Ingalls, entrambi ricercatori allo Xerox Park di Palo Alto, in California.

Ma, ancora una volta, il grande successo tardò ad arrivare. Probabilmente, la ragione di ciò era da ricercare nel fatto che questo linguaggio (come il Simula) era considerato, per lo più, uno **strumento destinato alla ricerca** e allo studio più che allo sviluppo.

A questo si aggiunga che negli anni '70 il linguaggio **C riscuoteva enorme successo** grazie alle sue potenzialità e, soprattutto, al fatto che il famoso sistema operativo Unix (ancora oggi fortemente usato) fosse stato scritto utilizzando proprio tale linguaggio.

Fu, insomma, necessario attendere gli anni '80 con l'avvento del **linguaggio ADA** per assistere alla definitiva consacrazione della programmazione ad oggetti come modello da utilizzare. Probabilmente la vera svolta fu rappresentata dall'avvento del C++, creato da Bjarne Stroustrup.

Tra **i più noti** linguaggi di programmazione ad oggetti, citiamo il C++, Java, Delphi, C# e, come detto, il Visual Basic .NET

3. Tecniche di Programmazione

Si è precedetemente accennato a "vecchie" metodologie di programmazione e a come queste siano state lentamente ma inesorabilmente messe da parte con la diffusione dell'OOP. In questa lezione verranno descritte brevemente tali metodologie al fine di comprendere meglio le differenze con la programmazione ad oggetti.

Programmazione Non Strutturata

Con la programmazione non strutturata il programma è costituito da **un unico blocco di codice detto "main"** dentro il quale vengono manipolati i dati in maniera totalmente sequenziale.

È importante notare che tutti i dati sono rappresentati soltanto da **variabili di tipo globale**, ovvero visibili da ogni parte del programma ed allocate in memoria per tutto il tempo che il programma stesso rimane in esecuzione.

È facile capire che un simile contesto è **fortemente limitato e pieno di svantaggi**. Ad esempio, sarà facile incappare in spezzoni di codice ridondanti o ripetuti che non faranno altro che rendere presto ingestibile ed "illeggibile" il codice, causando oltretutto un enorme spreco di risorse di sistema.

Nella figura seguente, viene rappresentata graficamente l'architettura di questo tipo di paradigma di sviluppo.

Figura 1. Schema della programmazione non strutturata

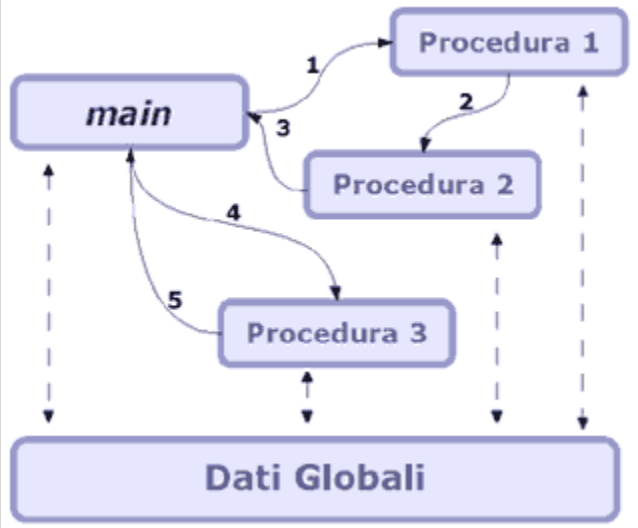
Programmazione Procedurale

Un notevole passo in avanti, rispetto alla Programmazione Non Strutturata, venne fatto con l'avvento della Programmazione Procedurale. I programmatori in Visual Basic 5.0 o 6.0 sapranno certamente di cosa si sta parlando (anche se il Visual Basic utilizza anche il paradigma di Programmazione Modulare, descritta nel successivo paragrafo).

Il concetto base qui è quello di raggruppare i pezzi di programma ripetuti in porzioni di codice utilizzabili e richiamabili ogni volta che se ne presenti l'esigenza: **nascevano le Procedure**.

Ogni procedura può essere vista come un sottoprogramma che svolge una ben determinata **funzione** (ad esempio, il calcolo della radice quadrata di un numero) e che è visibile e richiamabile dal resto del codice.
Inoltre ogni procedura ha la capacità di poter utilizzare uno o più **parametri** che ne consentono una maggiore duttilità. Anche il flusso del programma è decisamente diverso rispetto a quello visto nella programmazione non strutturata: infatti, il main continua ad esistere ma al suo interno appaiono soltanto le invocazioni alle procedure definite nel programma.
Quando una procedura ha terminato il suo compito il controllo ritorna nuovamente al main (o alla procedura che ne ha effettuato l'**invocazione**) che esegue una nuova chiamata ad un'altra procedura. fino alla terminazione del programma. Un semplice schema puo' facilitare la comprensione di tale concetto:

Figura 2. Flusso di un programma con programmazione procedurale

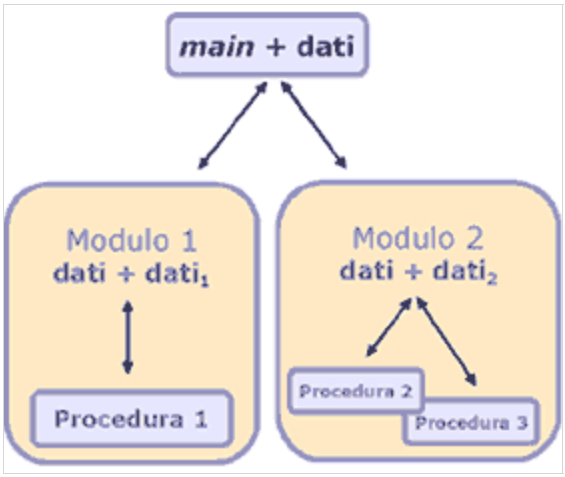


Il grande vantaggio della programmazione procedurale, rispetto alla precedente non strutturata consiste in un notevole**abbattimento del numero di errori**, che deriva dal fatto che se una procedura è corretta allora vi è la certezza che essa restituirà ad ogni invocazione dei risultati corretti in output.

Programmazione Modulare

La programmazione modulare rappresenta un' ulteriore conquista. Sorgeva l'esigenza di poter **riutilizzare le procedure** messe a disposizione da un programma in modo che anche altri programmi ne potessero trarre vantaggio.
Così, l'idea fu quella di raggruppare le **procedure aventi un dominio comune** (ad esempio, procedure che eseguissero operazioni matematiche) in moduli separati.
Quando sentiamo parlare di **librerie** di programmi, in sostanza si fa riferimento proprio a **moduli di codice** indipendenti che ben si prestano ad essere inglobati in svariati programmi.

Figura 3. Struttura di un programma «modulare»



Il risultato, dunque, adesso è che un singolo programma non è più costituito da un solo file (in cui è presente il main e tutte le procedure) ma da diversi moduli (uno per il main e tanti altri quanti sono i moduli a cui il programma fa riferimento).

È importante, inoltre, dire che i singoli moduli possono contenere anche dei dati propri che, in congiunzione ai dati del main, vengono utilizzati all'interno delle procedure in essi contenute.

Programmazione ad Oggetti

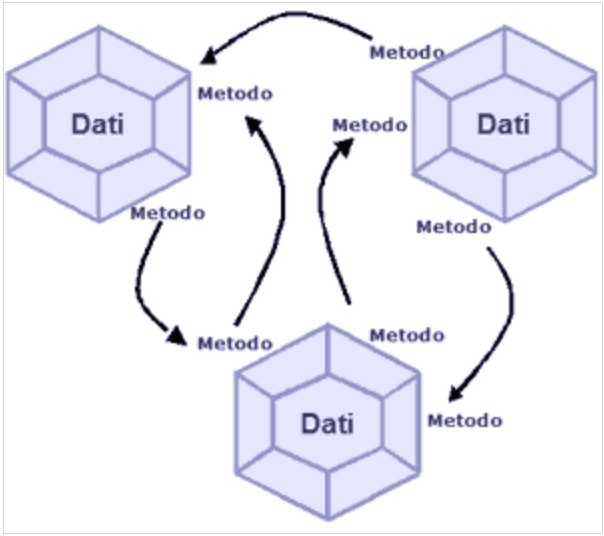
La programmazione procedurale/modulare ha rappresentato il punto di riferimento nello sviluppo applicativo per tanti anni. Gradualmente ma inevitabilmente però, man mano che gli orizzonti della programmazione diventavano sempre più ampi, si andarono evidenziando i **limiti** di tale metodologia.

In particolare, un programma procedurale mal si prestava a realizzare il concetto di "Componente Software", ovvero di un prodotto in grado di garantire le caratteristiche di riusabilità, modificabilità e manutenibilità.

Una delle cause di tale limite è da ricercare sicuramente nel fatto che esiste un evidente scollamento tra i dati e le strutture di controllo che agiscono su di essi; in altre parole i moduli risultano avere un approccio orientato alla procedura piuttosto che ai dati.

Con l'avvento della programmazione ad oggetti (i cui concetti saranno dettagliati a partire dal prossimo capitolo) questi limiti vennero superati. Contrariamente alle tecniche fin qui descritte, il paradigma OOP è basato sul fatto che esiste una serie di oggetti che interagiscono vicendevolmente, scambiandosi messaggi ma mantenendo ognuno il proprio stato ed i propri dati. Graficamente:

Figura 4. Colloquio tra oggetti



La programmazione ad oggetti, naturalmente, pur cambiando radicalmente l'approccio mentale all'analisi progettuale non ha fatto a meno dei vantaggi derivanti dall'uso dei moduli. Al contrario, tale tecnica è stata ulteriormente raffinata avvalendosi delle potenzialità offerte dalla programmazione ad oggetti.

B. Concetti Basilari di OOP

4. Rappresentazione della realtà

L'**idea principale** che sta dietro la Programmazione ad Oggetti risiede, in buona parte, nel mondo reale. Come spesso accade, gli esempi pratici forniscono una comprensione più chiara ed immediata della teoria e, pertanto, si userà tale approccio per fissare bene in mente i concetti basilari del mondo OOP.

Si prenda in considerazione un comune masterizzatore, come quelli ormai diffusi nella stragrande maggioranza dei moderni PC. Di tale oggetto si conoscono le sue caratteristiche ma quasi nessuno (se non coloro che lo hanno progettato e assemblato) è a conoscenza dei componenti elettronici e dei sofisticati meccanismi che ne regolano il corretto funzionamento, ne' a qualcuno salterebbe in mente di smontare un masterizzatore prima di acquistarlo per vedere come è fatto all'interno (supposto che il negoziante sia disposto ad accettare una tale richiesta, decisamente fuori dal comune!).

Il concetto che sta dietro alla programmazione ad oggetti nasce dallo stesso principio: ciò che importa non è l'implementazione interna del codice (che corrisponde ai componenti elettronici, nel caso del masterizzatore) ma, piuttosto, **le caratteristiche e le azioni** che un componente software è in grado di svolgere e che mette a disposizione (espone) all'esterno.

Dunque, con riferimento a quanto gia detto in precedenza, un programma che segue **il paradigma OOP** è costituito da un numero variabile di tali componenti, che ora denominiamo oggetti, i quali interagiscono tra di essi attraverso lo scambio di messaggi.

Proseguendo nell'**esempio** del masterizzatore e supponendo di voler implementare un oggetto software che ne gestisca le funzionalità, potremmo iniziare a definirne le **caratteristiche**. Ad esempio:

•Marca

•Velocità di scrittura su supporti CD-R

•Velocità di scrittura su supporti CD-RW

•Velocità di lettura

•Interfaccia

•Dimensione del buffer dati

Mentre, per quanto riguarda le **azioni** potremmo considerare le seguenti:

•Scrivi su CD

•Scrivi su DVD

•Leggi CD

•Espelli CD

In parole molto semplici, potremmo dire che le azioni altro non sono che "le cose che un oggetto è in grado di fare" mentre le sue caratteristiche rappresentano i dati che le azioni stesse possono utilizzare per eseguire le operazioni che da esse ci si aspetta.

In OOP, le caratteristiche di un oggetto vengono denominate **proprietà** e le azioni sono dette **metodi**. Questi sono due concetti fondamentali ed è molto importante che siano ben chiari prima di introdurre nuove definizioni.

Altrettanto importante, però, è ribadire il contesto in cui abbiamo iniziato a ragionare: non dobbiamo pensare più ad un programma che racchiuda tutto in un unico grande calderone ma dobbiamo iniziare a **pensare "ad oggetti"**. Dobbiamo, cioè, essere in grado di identificare gli oggetti che entrano in gioco nel programma che vogliamo sviluppare e saperne gestire l'interazione degli uni con gli altri.

In un **programma di tipo procedurale**, si è soliti iniziare a ragionare in maniera top-down, partendo cioè dal main e creando mano a mano tutte le procedure necessarie. Tutto questo non ha più senso nella programmazione ad oggetti, dove serve invece definire prima le classi e poi associare ad esse le proprietà ed i metodi opportuni.

Nella guida UML, si fa riferimento (dove si parla del Rapid Application Development) al metodo utilizzato durante l'analisi progettuale per identificare in modo ottimale le classi da utilizzare.

5. Metodi e proprietà

I metodi: le azioni che un oggetto è in grado di compiere

Come detto, un metodo rappresenta una azione che può essere compiuta da un oggetto. Una delle domande principali da porsi quando si vuole creare un oggetto è: "Cosa si vuole che sia in grado di fare?".

In effetti, quando ci si pone questa domanda, si sta involontariamente dando per buono il fatto che qualsiasi oggetto sia in qualche modo capace di eseguire delle azioni, trattandolo come se avesse natura umana. Eppure, anche se inizialmente può apparire quantomeno curioso e insolito, questo approccio rappresenta un importante riferimento nella fase di disegno e creazione degli oggetti.

In generale, potremmo delineare almeno **tre buone regole** per identificare i metodi da associare ad un oggetto:

1.**Un oggetto che abbia uno o due soli metodi deve fare riflettere**. Potrebbe essere perfettamente lecito definire un oggetto del genere (ed è sicuramente possibile farlo praticamente) ma, spesso, un oggetto creato con questi requisiti indica la necessità di "mescolarlo" con un altro oggetto con simile definizione.

2.Ancora più **da evitare sono gli oggetti con nessun metodo**. È bene che un oggetto incapsuli (vedremo meglio in seguito cosa si intenda con tale terminologia) dentro sé sia informazioni (le proprietà), sia azioni (i metodi, appunto). In linea di massima, un oggetto senza metodi può facilmente essere convertito in uno o più attributi da assegnare ad un altro oggetto.

3.Sicuramente **da evitare sono anche gli oggetti con troppi metodi**. Un oggetto, in generale, dovrebbe avere un insieme facilmente gestibile di proprie responsabilità. Assegnare ad un oggetto troppe azioni, potrebbe rendere ardua la manutenzione futura dello stesso oggetto. È consigliabile, in questo caso, cercare di spezzare l'oggetto in due oggetti più piccoli e semplificati.

Le proprietà: informazioni su cui opera un oggetto

Le proprietà rappresentano **i dati dell'oggetto**, ovvero le informazioni su cui i metodi possono eseguire le loro elaborazioni.

Uno degli errori più comuni che si commette quando si definiscono le proprietà di un oggetto è quello di associare ad esso quante più proprietà possibili, ritenendo che questo possa, in qualche modo, facilitare la stesura del programma. Un oggetto, invece, per essere ben definito deve contenere le proprietà che, effettivamente, gli competono e non tutte quelle che gli si potrebbero comunque attribuire.

Questa regola di buona programmazione nasce, oltretutto, dall'esigenza di rendere più facile la fase di disegno e quella di debug che altrimenti risulterebbero certamente complesse.

Per evitare, dunque, l'inconveniente di ritrovarsi dei super-oggetti sarà bene **porsi la seguente domanda** nella fase di definizione: «quali proprietà sono necessarie affinché l'oggetto sia in grado di eseguire le proprie azioni?»

In generale, esistono **tre tipologie di proprietà**: gli attributi, i componenti e i peer objects.

Gli attributi rappresentano quelle proprietà che descrivono le caratteristiche peculiari di un oggetto (ad esempio, riferendoci ad una persona: altezza, peso).

I componenti, invece, sono identificabili in quelle proprietà che sono atte a svolgere delle azioni (testa, corpo, mani, gambe).

Infine, **i peer objects** definiscono delle proprietà che a loro volta sono identificate e definite in altri oggetti (ad esempio: l'automobile posseduta da una persona)

6. Le Classi

Probabilmente, il termine più importante e rappresentativo nella Programmazione ad Oggetti è quello di Classe.

Riprendiamo ancora l'esempio del masterizzatore. Sappiamo benissimo che non esiste un solo tipo di masterizzatore ma, a secondo delle caratteristiche, è possibile citarne tanti modelli: ci sono, ad esempio, quelli in grado di scrivere su CD e DVD o quelli che scrivono solo su CD.

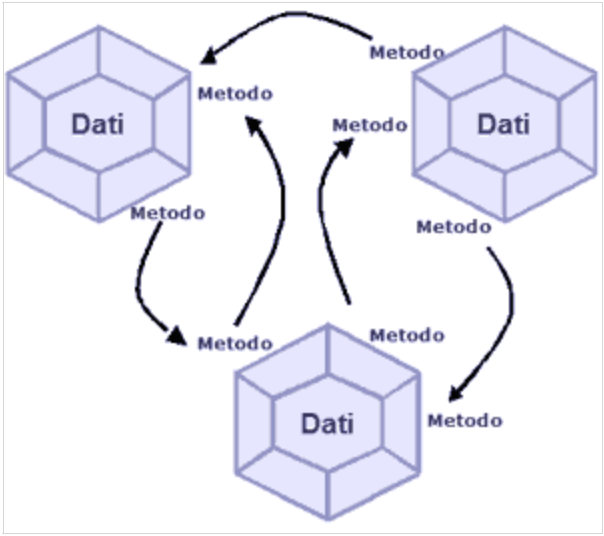
Sicuramente, però, tutti sono in grado di eseguire la scrittura di dati o file multimediali su CD e tutti hanno, altresì, le proprietà esposte nelle lezioni precedenti.

Diremo allora che più oggetti software che **hanno le stesse proprietà e gli stessi metodi** possono essere raggruppati in una classe ben definita di oggetti: nel nostro caso particolare, nella classe masterizzatore.
Dunque, una classe rappresenta, sostanzialmente, una **categoria particolare di oggetti** e, dal punto di vista della programmazione, è anche possibile affermare che una classe funge da tipo per un determinato oggetto ad essa appartenente (dove con tipo si intende il tipo di dato, come lo sono gli interi o le stringhe).
Diremo, inoltre, che un particolare oggetto che appartiene ad una classe costituisce un'**istanza della classe** stessa. In altre parole, un'istanza della classe masterizzatore sarà costituita da un oggetto di tale classe che è in grado di scrivere solo su CD mentre un'altra istanza potrà essere rappresentata da un oggetto che è in grado di masterizzare anche i DVD.
È anche possibile creare due o più istanze separate di **oggetti uguali** (sarebbe meglio dire di oggetti che hanno le proprietà valorizzate allo stesso modo) ma ciò non vorrà dire che in un determinato istante nel corso del programma tutte queste istanze eseguano la medesima operazione o che, in generale, siano necessariamente correlati.
L'univocità di ogni istanza viene definita con il termine di **identità** (identity): ogni oggetto ha una propria identità ben distinta da quella di tutte le altre possibili istanze della stessa classe a cui appartiene l'oggetto stesso.
Naturalmente, le proprietà di un oggetto possono avere valori che variano nel tempo. Ad esempio, la velocità di scrittura potrebbe essere selezionata dall'utente e, pertanto, potrebbe variare nel corso del programma. Si definisce **stato di un oggetto**, l'insieme dei valori delle sue proprietà in un determinato istante di tempo. Se cambia anche una sola proprietà di un oggetto, il suo stato varierà di conseguenza.
L'insieme dei metodi che un oggetto è in grado di eseguire viene definito, invece, **comportamento** (behavior).
Insomma, appare sempre più chiaro come un oggetto rappresenti un'entità a sé stante, ben definita che, nel corso dell'elaborazione, sia soggetta ad una creazione, ad un suo utilizzo e, infine, alla sua distruzione.
Inoltre, un oggetto non dovrebbe mai manipolare direttamente i dati interni (le proprietà) di un altro oggetto ma ogni tipo di comunicazione tra oggetti dovrebbe essere sempre gestita tramite l'**uso di messaggi**, ovvero tramite le chiamate ai metodi che un oggetto espone all'esterno

7. I Messaggi

Si è detto che i metodi rappresentano le azioni che un oggetto è in grado di eseguire. Ma, in base a quale criterio vengono scatenate ed eseguite tali azioni? Semplicemente rispondendo alle "**sollecitazioni" provenienti da altri oggetti**.
Quando, ad esempio, l'oggetto masterizzatore esegue il metodo Scrivi_CD, un altro oggetto (per esempio, il controller) gli avrà precedentemente inviato una simile richiesta sollecitandone l'azione di scrittura.
Tali sollecitazioni costituiscono quelli che, in un programma che utilizza il paradigma OOP, vengono definiti **messaggi**. Questi ultimi rappresentano il cuore del modello ad oggetti (come rappresentato in figura),

Figura 1. Colloquio tra oggetti



,ovvero un insieme di oggetti che eseguono determinate azioni in concomitanza di messaggi e che inviano messaggi a loro volta ad altri oggetti.

Probabilmente tali concetti possono suscitare, inizialmente, un po' di confusione ma sono più semplici di quanto si possa ritenere. Per fare maggiore chiarezza aggiungiamo che un oggetto può inviare dei messaggi soltanto alle sue proprietà (in particolare a quelle che sono definite a loro volta come oggetti.

Si faccia riferimento, a tal proposito, alla definizione di peer object). Quindi, con riferimento al nostro esempio, potremo dire che l'oggetto controller avrà al suo interno una proprietà masterizzatore, alla quale sarà in grado di inviare dei messaggi ogni qual volta si desideri che il masterizzatore compia una delle sue azioni.

Si è soliti suddividere i messaggi nelle seguenti categorie:

•**Costruttori**

I Costruttori costituiscono il momento in cui viene creato un oggetto. Essi devono essere richiamati ognivolta che si vuole creare una nuova istanza di un oggetto appartenente ad una classe e, solitamente, svolgono al loro interno funzioni di inizializzazione.

•**Distruttori**

I Distruttori, come si intuisce facilmente dal nome, svolgono la funzione inversa dei costruttori: distruggono un oggetto (ovvero ne eliminano la allocazione dalla memoria). All'interno di questi metodi viene, solitamente, effettuata anche una sorta di pulizia del codice, rimuovendo eventuali variabili allocate sulla memoria dinamica (heap).

Molti linguaggi orientati agli oggetti (come ad esempio Java e C#) non forniscono un uso diretto dei distruttori ma prevedono una rimozione automatica dell'oggetto quando questo esce dal contesto dell'applicazione (per approfondimenti in merito, si guardi il concetto di Garbage Collector in Java).

•**Accessori (Accessors)**

I messaggi di tipo “Accessors” vengono utilizzati per esaminare il contenuto di una proprietà di una classe. Solitamente si utilizzano questi metodi per accedere alle variabili dichiarate con visibilità Private (vedremo il significato di questo termine nel prossimo paragrafo). Una particolarità: i metodi di tipo selectors (Selettori) eseguono un confronto tra due o più elementi prima di restituire il valore di ritorno.

•**Modificatori (Mutators)**

I Modificatori rappresentano tutti i messaggi che provocano una modifica nello stato di un oggetto.

8. Relazioni tra classi

Una classe che non si interfacci con altre classi è sicuramente poco significativa in OOP. Abbiamo visto che gli oggetti, in un programma Object Oriented, interagiscono tra loro utilizzando lo scambio di messaggi per richiedere l’esecuzione di un particolare metodo.

Tale comunicazione consente di identificare all’interno del programma una serie di relazioni tra le classi in gioco la cui documentazione risulta essere assai utile in fase di disegno e di analisi.

Ad esempio, se ci si accorge dell’esistenza di due o più classi che abbiano un comportamento comune, sarà il caso di inglobare tale comportamento in una classe di livello superiore in modo da risparmiarsi un po’ di fatica (riprenderemo tale concetto quando si parlerà di ereditarietà) a tutto vantaggio dello sviluppo.

Viceversa, se alcune classi risultano essere totalmente slegate tra loro sarà possibile, in fase di implementazione del codice, procedere in modo parallelo in modo che non sia necessario che uno dei programmatori debba attendere che un altro finisca per procedere nella stesura del codice assegnatogli.

Le più comuni relazioni tra classi, in un programma ad Oggetti sono identificabili in tre tipologie:

•**Associazioni** (Use Relationship)

•**Aggregazioni** (Containment Relationship)

•**Specializzazioni** (Inheritance Relationship)

Associazione

L’Associazione è il tipo di Relazione più intuitiva ed anche più diffuso. In generale, diciamo che una classe A utilizza una classe B se un oggetto della classe A è in grado di inviare dei messaggi ad un oggetto di classe B oppure se un oggetto di classe A può creare, ricevere o restituire oggetti di classe B.

Più in dettaglio, potremmo dire che una classe è associata ad un’altra se è possibile “navigare” da oggetti della prima classe ad oggetti della seconda classe seguendo semplicemente un riferimento ad un oggetto.

Ad **esempio**, dato un oggetto di tipo Persona, è possibile giungere ad oggetti di tipo Azienda accedendo semplicemente alla variabile istanza azienda definita all’interno della classe Persona.. Nella figura seguente viene rappresentata graficamente la relazione di tipo Associazione tra queste due classi:

Figura 1. Diagramma di una relazione di associazione

Si può notare, osservando la figura, come sia anche possibile assegnare un nome alla associazione tra le due classi (Il nome in questione è: "Lavora per") e dare una direzione all'associazione stessa, intendendo con ciò il verso in cui avviene la navigazione tra le classi.

Se la navigazione è, invece, possibile in entrambe le direzioni si parlerà di **associazione bidirezionale** e non si inserirà alcuna freccia..

Aggregazione

La relazione di tipo Aggregazione si basa, invece, sul seguente concetto: Un oggetto di classe A contiene un oggetto di classe B se B è una proprietà (attributo) di A.

In sostanza, l'aggregazione è una forma di associazione più forte: una classe ne aggrega un'altra se esiste tra le due classi una relazione di tipo "intero-parte".

Ad esempio la classe Azienda aggrega la classe Persona perché una ditta (che costituisce l'"intero") è composta da persone (che costituiscono la "parte").

Una classe ContoBancario, invece, non è legata da una relazione di tipo aggregazione con la classe Persona anche se può essere plausibile che sia possibile navigare da un oggetto che rappresenta un conto bancario fino ad un oggetto che rappresenta una persona, il proprietario del conto. Dal punto di vista concettuale, però, una persona non si può far appartenere ad un conto bancario.

Composizione

La Composizione è una forma di aggregazione ancora più forte che indica che una "parte" può appartenere ad un solo "intero" in un certo istante di tempo.

Ad esempio, uno pneumatico può far parte di una sola automobile in un certo istante, mentre, al contrario, una persona potrebbe lavorare contemporaneamente per due ditte.

Ad essere sinceri, le differenze tra associazione, aggregazione e composizione possono trarre in confusione anche gli analisti più esperti. Per tale motivo si raccomanda di utilizzare tali tipi di relazioni soltanto se il loro utilizzo può essere di reale giovamento.

Specializzazione

La relazione di tipo Specializzazione si basa sul concetto di ereditarietà che verrà affrontato in dettaglio nei prossimi paragrafi: Un oggetto di classe A deriva da un oggetto di classe B se A è in grado di compiere tutte le azioni che l'oggetto B è in grado di compiere.

Inoltre l'oggetto di classe A è in grado di eseguire anche azioni che l'oggetto B non può compiere. Ad esempio, un masterizzatore rappresenta anch'esso un tipo di lettore CD (poiché riesce anche a leggere CD), e volendo si

potrebbe pensarlo come una estensione di quest'ultimo. In più, rispetto a quest'ultimo, ne estende le funzionalità, visto che è in grado anche di scrivere sui supporti ottici.

Riferimenti

Per ulteriori approfondimenti circa le relazioni tra classi si consiglia la lettura della guida UML,

C. I capisaldi della Programmazione ad Oggetti

9. "Pensare" Object Oriented

Abbiamo visto che la programmazione ad oggetti rappresenta un modo differente di pensare le applicazioni. Ogni applicazione è composta da un certo numero di oggetti, ognuno dei quali è indipendente dagli altri ma comunica con gli altri attraverso lo scambio di messaggi.

Uno dei vantaggi principali derivanti dall'uso della programmazione ad oggetti è la capacità di costruire dei componenti una volta e, quindi, riutilizzarli successivamente ogniqualvolta se ne presenti la necessità.

Così come, ad esempio, è possibile riutilizzare una gru sia che si stia costruendo un castello, un palazzo o una chiesa allo stesso modo è, allora, possibile riutilizzare una porzione di codice object oriented in svariati sistemi come un sistema di processamento di ordini o in un sistema per il calcolo della contabilità.

Certo è che per ottenere **una classe riusabile bisogna progettarla bene**. Un oggetto può essere riusato se presenta caratteristiche utili ad interfacciarsi a diversi contesti.

Da quanto detto, si evince che la programmazione ad oggetti consente una grande flessibilità e, allo stesso tempo, una enorme potenza di utilizzo. Ciò richiede, però, la conoscenza di alcuni principi fondamentali che costituiscono l'ossatura di tutto il mondo Object Oriented. Vediamo quali sono.

10. L'incapsulamento

Nei programmi Object Oriented, come abbiamo già osservato, si è soliti mettere in stretta relazione tra loro un pezzo di informazione con il comportamento specifico che agisce su tale informazione. Questo è ciò che abbiamo definito oggetto.

L'incapsulamento è proprio legato al concetto di "**impacchettare" in un oggetto i dati e le azioni** che sono riconducibili ad un singolo componente.

Un altro modo di guardare all'incapsulamento, che abbiamo già accennato, è quello di pensare a suddividere un'applicazione in piccole parti (gli oggetti, appunto) che raggruppano al loro interno alcune funzionalità legate tra loro.

Ad esempio, pensiamo ad un conto bancario. Le informazioni utili (le proprietà) potranno essere rappresentate da: il numero di conto, il saldo, il nome del cliente, l'indirizzo, il tipo di conto, il tasso di interesse e la data di apertura.

Le azioni che operano su tali informazioni (i metodi) saranno, invece: apertura, chiusura, versamento, prelevamento, cambio tipologia conto, cambio cliente e cambio indirizzo. L'oggetto Conto incapsulerà queste informazioni e azioni al suo interno.

Come risultato, ogni modifica al sistema informatico della banca che implichi una modifica ai conti correnti, potrà essere implementata semplicemente nell'oggetto Conto.

Un altro vantaggio derivante dall'incapsulamento è quello di limitare gli effetti derivanti dalle modifiche ad un sistema software.

Chiariamo meglio il concetto avvalendoci di un classico esempio. Si pensi, ad un sistema software come ad una distesa di acqua e ad una richiesta di modifica del software come ad una enorme masso. Gettando il masso nell'acqua si creeranno tante onde e spruzzi in tutte le direzioni. Tali onde si propagheranno per tutta la distesa di acqua, rimbalzeranno sulla costa e, alla fine, si scontreranno con altre onde. In definitiva, il masso lanciato in acqua avrà causato un notevole effetto di disturbo su tutta la superficie su cui si estende la distesa d'acqua.

Proviamo, adesso, ad utilizzare il concetto di incapsulamento, definito in precedenza, applicandolo a tale esempio. La distesa d'acqua viene suddivisa in tante piccole porzioni di acqua, ognuna delle quali è ben delimitata con delle barriere per evitare che l'acqua fuoriesca da essa. Se gettiamo un masso su una di queste porzioni di acqua avremo ancora una volta il verificarsi di onde in tutte le direzioni ma, questa volta, esse termineranno la loro azione scontrandosi con le barriere delimitatrici. In sostanza, tutte le altre porzioni di acqua non verranno minimamente intaccate.

Cerchiamo ora di esaminare una situazione che preveda una modifica da apportare rimanendo nel contesto del sistema bancario di cui abbiamo definito proprietà e metodi. Supponiamo che la banca in questione abbia deciso che se un determinato cliente ha un conto di credito nella stessa banca, tale conto possa essere utilizzato come copertura per eventuali scoperti sul conto corrente.

Per gestire tale situazione, in un sistema non incapsulato, si procederebbe con un approccio decisamente poco efficace. Ovvero, non sapendo dove siano localizzati nel codice i punti in cui viene gestita l'operazione di prelievo, l'unica alternativa possibile sarebbe quella di ricercare dappertutto, aggiungendo la nuova funzionalità al programma ogni volta che si identifichi un punto da modificare.

Con buona probabilità, se saremo stati bravi, saremo riusciti ad individuare l'80 per cento dei punti da modificare. Con l'utilizzo dell'incapsulamento sarà, invece, sufficiente identificare l'oggetto che gestisce l'azione di prelevamento (ovvero l'oggetto Conto) ed eseguire la modifica all'interno di esso, completando la variazione del sistema richiesta senza intaccare tutto il resto del sistema stesso.

Un concetto simile all'incapsulamento è l'**occultamento dell'informazione**, meglio noto con il termine di information hiding. Tale concetto esprime l'abilità di nascondere al mondo esterno tutti i dettagli implementativi più o meno complessi che si svolgono all'interno di un oggetto. Il mondo esterno, per un oggetto, è rappresentato da qualunque cosa si trovi all'esterno dell'oggetto stesso.

L'information hiding, come l'incapsulamento fornisce lo stesso vantaggio: la flessibilità.

Implemetare l'incapsulamento

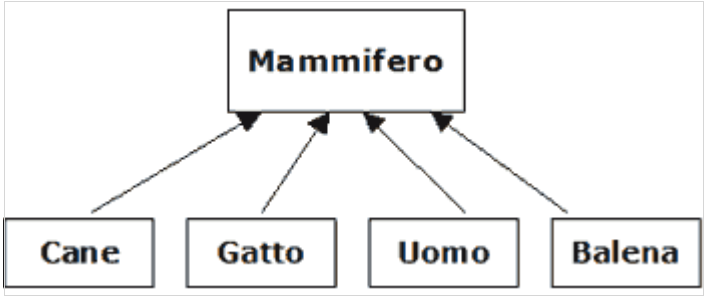
Ecco come si implementa l'incapsulamento nei vari linguaggi orientati agli oggetti per il web.

- incapsulamento in Java
- incapsulamento in C#
- incapsulamento in C++
- incapsulamento in VB.NET
- incapsulamento in Actionscript
- incapsulamento in Javascript
- incapsulamento in Python

11. L'ereditarietà

L'ereditarietà costituisce il secondo principio fondamentale della programmazione ad oggetti. In generale, essa rappresenta un meccanismo che consente di creare nuovi oggetti che siano basati su altri già definiti.
Si definisce **oggetto figlio** (child object) quello che eredita tutte o parte delle proprietà e dei metodi definiti nell'**oggetto padre** (parent object).
È semplice poter osservare esempi di ereditarietà nel mondo reale. Ad esempio, esistono al mondo centinaia tipologie diverse di mammiferi: cani, gatti, uomini, balene e così via. Ognuna di tali tipologie di mammiferi possiede alcune caratteristiche che sono strettamente proprie (ad esempio, soltanto l'uomo è in grado di parlare) mentre esistono, d'altra parte, determinate caratteristiche che sono comuni a tutti i mammiferi (ad esempio, tutti i mammiferi hanno il sangue caldo e nutrono i loro piccoli).
Nel mondo Object Oriented, potremmo riportare tale esempio definendo un oggetto Mammifero che inglobi tutte le caratteristiche comuni ad ogni mammifero. Da esso, poi, deriverebbero gli altri child object: Cane, Gatto, Uomo, Balena, etc.
L'oggetto cane, per citarne uno, erediterà, quindi, tutte le caratteristiche dell'oggetto mammifero e a sua volta conterrà delle caratteristiche aggiuntive, distintive di tutti i cani come ad esempio: ringhiare o abbaiare. Il paradigma OOP, ha quindi carpito l'idea dell'ereditarietà dal mondo reale, come mostrato nella figura seguente:

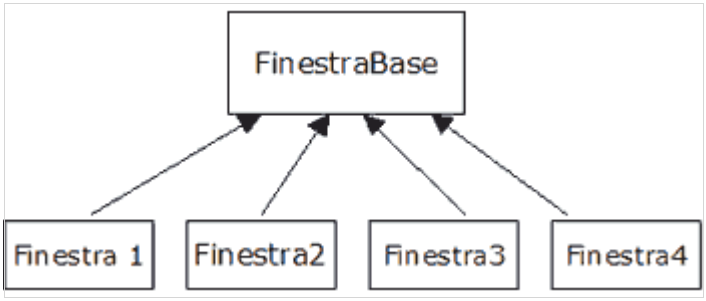
Figura 1. Esempio di classificazione



e, pertanto lo stesso concetto viene applicato ai sistemi software che utilizzano tale tecnologia.
Uno dei maggiori vantaggi derivanti dall'uso dell'ereditarietà è la maggiore facilità nella manutenzione del software. Infatti, rifacendoci all'esempio dei mammiferi, se qualcosa dovesse variare per l'intera classe dei mammiferi, sarà sufficiente modificare soltanto l'oggetto padre per consentire che tutti gli oggetti figli ereditino la nuova caratteristica.
Ad esempio, se i mammiferi diventassero improvvisamente (e anche inverosimilmente!) degli animali a sangue freddo, soltanto l'oggetto padre mammifero necessiterà di tale variazione. Il gatto il cane, l'uomo, la balena, e tutti gli altri oggetti figli erediterebbero automaticamente la caratteristica di avere il sangue freddo, senza nessuna modifica.
Un esempio che metta in luce la potenza dell'ereditarietà, in un programma Windows object oriented potrebbe avere come oggetto fulcro le finestre associate al programma stesso. Supponiamo, ad esempio, che tale software utilizzi, in totale, 100 finestre differenti, visualizzabili a secondo del contesto in cui sta navigando l'utente.

Se, un giorno, si volesse inserire un campo comune a tutte le finestre del programma, in che modo sarà bene procedere? Se non avremo utilizzato la potenza dell'ereditarietà l'unica strada percorribile sarà quella di andarsi a prendere una per una tutte le definizioni delle finestre e inserire il nuovo campo.
Se, invece, si sarà utilizzato in maniera efficiente il paradigma Object Oriented, definendo una classe FinestraBase contenente tutte le caratteristiche comuni ad ogni finestra e derivando da tale classe tutte le finestre in gioco nel programma, allora la modifica sarà banalmente (allo stesso modo dell'esempio sui mammiferi) quella di inserire il nuovo campo nella classe FinestraBase. Graficamente:

Figura 2. Classificazione di oggetti software



Ancora un esempio. In un sistema bancario, si potrebbe utilizzare l'ereditarietà per definire tutte le tipologie di conto esistenti. Supponiamo che i conti possibili siano quattro : conti correnti bancario, libretti di risparmio, carte di credito e certificati di deposito.
Tutte queste differenti tipologie di conto hanno in comune alcune caratteristiche: un numero di conto, un tasso di interesse ed un sottoscrittore. In tal modo potremo creare un oggetto padre chiamato Conto che contenga tutte le caratteristiche comuni prima enunciate.
Gli oggetti figli, in aggiunta, avranno le loro specifiche caratteristiche definite nelle rispettive classi. Ad esempio, la carta di credito avrà un limite di spesa mentre il il conto corrente bancario avrà uno o più libretti di assegni associati. Anche in questo caso, le eventuali modifiche apportate alla classe padre saranno ereditate automaticamente da tutte le classi figlie.
È importante, però chiarire un aspetto importante quando si parla di caratteristiche ereditate. Non sempre, infatti, un determinato metodo definito nella classe padre può produrre risultati corretti e congruenti con tutte le classi figlie. Ad esempio, supponiamo di aver definito una classe padre denominata Uccello, dalla quale faremo derivare le seguenti classi figlie: Passerotto, Merlo e Pinguino.
Nella classe padre, avremo definito il metodo vola(), in quanto rappresenta un comportamento comune a tutti gli uccelli. In tal modo, secondo quanto si è detto in questo paragrafo, tutte le classi figlie non avranno la necessità di implementare tale metodo ma lo erediteranno dalla classe Uccello. Purtroppo, però, nonostante il pinguino appartenga alla categoria degli uccelli, è noto che esso non è in grado di volare, seppur provvisto di ali.
In questo caso, il metodo vola() definito nella classe Uccello, sicuramente valido per la stragrande maggioranza di uccelli, non sarà utile (anzi, sarà proprio sbagliato) per la classe Pinguino. Come comportarsi in questi casi?
In OOP, ogni oggetto derivante da una classe padre ha la possibilità di ignorare uno o più metodi in essa definiti riscrivendo tali metodi al suo interno. Questa caratteristica è nota come **overriding**.

Utilizzando la tecnica dell'overriding, la classe Pinguino reimplementerà al suo interno il metodo vola(), conservando, comunque, la possibilità di richiamare in qualunque momento, anche il metodo definito nella classe padre. In quest'ultimo caso si parlerà di **overriding parziale**.

Implemetare l'ereditarietà

Ecco come si implementa l'ereditarietà in alcuni dei linguaggi orientati agli oggetti per il web.

- ereditarietà in Java
- ereditarietà in C#
- ereditarietà in C++
- ereditarietà in VB.NET
- ereditarietà in Actionscript
- ereditarietà in Javascript
- ereditarietà in Python

12. Il polimorfismo

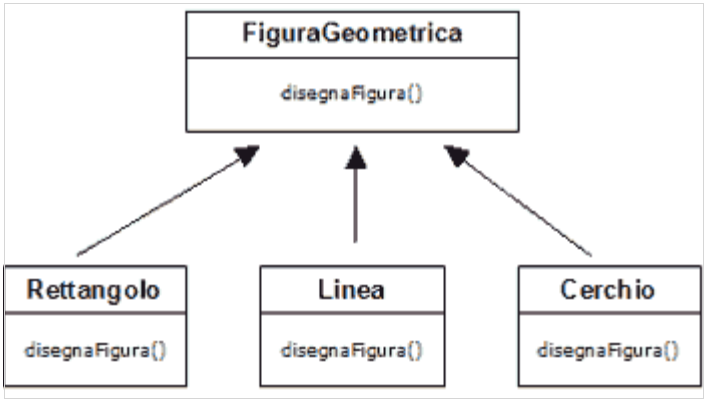
Il terzo elemento fondamentale della programmazione ad Oggetti è il polimorfismo. Letteralmente, la parola polimorfismo indica la possibilità per uno stesso oggetto di assumere più forme.

Per rendere l'idea più chiara, utilizzando ancora una volta un esempio del mondo reale, si pensi al diverso comportamento che assumono un uomo, una scimmia e un canguro quando eseguono l'azione del camminare. L'uomo camminerà in modo eretto, la scimmia in maniera decisamente più goffa e curva mentre il canguro interpreterà tale azione saltellando.

Riferendoci ad un sistema software ad oggetti, il polimorfismo indicherà l'attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità.

Per esempio, supponiamo di voler costruire un sistema software in grado di disegnare delle figure geometriche. Per tale sistema avremo definito una classe padre chiamata FiguraGeometrica dalla quale avremo fatto derivare tutte le classi che si occupano della gestione di una figura geometrica ben precisa. Per maggiore chiarezza riportiamo quanto detto nel diagramma seguente:

Figura 1. Esempio di polimorfismo



Quando l'utente desidera rappresentare una di tali figure, sia essa una linea, un cerchio o un rettangolo, egli eseguirà una determinata azione che produrrà l'invio al sistema di un messaggio che, a sua volta, scatenerà l'invocazione del metodo disegnaFigura della classe FiguraGeometrica..

Con l'utilizzo del polimorfismo, il sistema è in grado di capire autonomamente quale figura geometrica debba essere disegnata ed invocarne direttamente il metodo disegnaFigura appartenente alla classe figlia coinvolta.

In un sistema non ad oggetti (e, quindi, senza la possibilità di utilizzare il polimorfismo) un simile comportamento necessiterebbe, dal punto di vista del codice, di un costrutto tipo switch – case come il seguente, tutto implementato all'interno di un'unica classe:

```
function disegnaFigura()
{
  CASE FiguraGeometrica.Tipo
    Case 'Cerchio'
      FiguraGeometrica.DisegnaCerchio()
    Case 'Rettangolo'
      FiguraGeometrica.DisegnaRettangolo()
    Case 'Linea'
      FiguraGeometrica.DisegnaLinea()
  END CASE
}
```

Al contrario, con l'utilizzo del polimorfismo, il tutto si riconduce ad una semplice chiamata del tipo:

```
function Disegna(
{
  FiguraGeometrica.disegnaFigura()
}
```

Dietro una semplice codifica di questo tipo si nasconde tutta la potenza del polimorfismo: il sistema chiama in modo automatico il metodo disegnaFigura() dell'oggetto che è stato selezionato dall'utente, senza che ci si debba preoccupare se si tratti di cerchio, rettangolo o linea. In altre parole, potremmo dire che il polimorfismo consente ad oggetti differenti (ma collegati tra loro) la flessibilità di rispondere in modo differente allo stesso tipo di messaggio.

Uno dei maggiori benefici del polimorfismo, come in effetti di un po' tutti gli altri principi della programmazione ad oggetti, è la facilità di manutenzione del codice. Per rendere l'idea, basta domandarsi cosa accadrebbe se l'applicazione precedente volesse implementare anche la funzionalità di disegnare un triangolo.

Nel caso non polimorfico, bisognerebbe aggiungere una nuova funzione DisegnaTriangolo all'oggetto FiguraGeometrica e poi modificare la funzione DisegnaFigura globale, esaminata in precedenza, aggiungendo ad essa due nuove righe per gestire il caso della nuova tipologia di figura da inserire.

Con il polimorfismo invece, molto più semplicemente, basterà creare l'oggetto Triangolo e implementare in esso il metodo disegnaFigura(). L'invocazione di quest'ultimo avverrà, come per le altre figure geometriche già definite, in modo del tutto trasparente e automatico.

Implemetare il polimorfismo

Ecco come si implementa il polimorfismo in alcuni dei linguaggi orientati agli oggetti per il web.

- polimorfismo in Java
- polimorfismo in C#
- polimorfismo in C++
- polimorfismo in VB.NET
- polimorfismo in Actionscript
- polimorfismo in Javascript
- polimorfismo in Python

13. Astrazione e classi astratte

Come si è visto, una delle peculiarità maggiori della programmazione ad oggetti è quella di rendere agevole, snella ed efficiente la manutenzione del software.

Quando si è parlato di polimorfismo, di ereditarietà e di incapsulamento si sono messi in luce i vantaggi derivanti da tali caratteristiche nella gestione di eventuali modifiche da apportare successivamente alla fase di rilascio del software stesso.

Il concetto di Astrazione dei Dati interviene a rafforzare ulteriormente questi punti di forza, in particolare per quanto riguarda il riutilizzo del codice.

Più in particolare, si può affermare che l'**Astrazione dei Dati** viene utilizzata per gestire al meglio la complessità di un programma, ovvero viene applicata per decomporre sistemi software complessi in componenti più piccoli e semplici che possono essere gestiti con maggiore facilità ed efficienza.

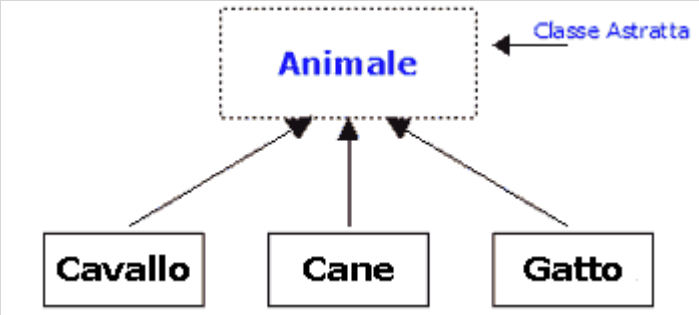
Una delle definizioni migliori sul concetto di Astrazione dei Dati è quella di Booch: «Un'astrazione deve denotare le caratteristiche essenziali di un oggetto contraddistinguendolo da tutti gli altri oggetti e fornendo, in tal modo, dei confini concettuali ben precisi relativamente alla prospettiva dell'osservatore»

Per chiarire meglio i concetti appena esposti è sicuramente utile descrivere un esempio pratico. Consideriamo una classe Animale, che incapsuli al suo interno tutte le caratteristiche (metodi e proprietà) che sono comuni a tutti gli animali.

Ad esempio, si potranno definire i metodi mangia() e respira() e le proprietà altezza e peso (solo per citarne alcuni). Ma, riflettendoci un attimo, appare chiaro che creare una istanza di un oggetto Animale ha poco senso visto che è certamente più consono definire istanze di oggetti specifici come Cavallo, Cane, Gatto, etc.

In un simile contesto la classe Animale, non potendo essere direttamente istanziata, rappresenta un dato astratto ed è denominata, in OOP, **Classe Astratta**, ovvero una classe che rappresenta, fondamentalmente, un modello per ottenere delle classi derivate più specifiche e più dettagliate.

Figura 1. Esempio di classe «astratta»



In una classe astratta, solitamente sono contenuti pochi metodi (di solito uno o due) per i quali è fornita anche l'implementazione mentre per tutti gli altri metodi è presente soltanto una mera definizione del metodo stesso ed è, pertanto, necessario (ed obbligatorio) che tutte le classi discendenti ne forniscano la opportuna implementazione.

I metodi appartenenti a questa ultima tipologia (e che sono definiti nella classe astratta) prendono il nome di **Metodi Astratti**. Nel caso limite in cui una classe astratta contenga soltanto metodi astratti allora essa verrà catalogata più correttamente come interfaccia (vedasi paragrafo inerente le interfacce).

Come detto, l'utilizzo dell'astrazione dei dati (unito al concetto di ereditarietà) facilita il riutilizzo del codice e snellisce il disegno di un sistema software. Infatti, qualora si presentasse la necessità, sarà agevole poter definire delle altre classi intermedie che possano avvalersi delle definizioni già presenti nelle classi astratte. Inoltre, risulterà di enorme utilità poter riutilizzare le classi astratte già definite, anche in altri progetti.

Implemetare l'astrazione dei dati

Ecco come si implementa l'astrazione dei dati in alcuni dei linguaggi orientati agli oggetti per il web.

- classi astratte in Java
- classi astratte in C++
- classi astratte in Javascript
- classi astratte in PHP 5
- astrazione in C#
- classi astratte in Python
- classi astratte in VB.NET

•classi astratte in Actionscript

14. La visibilità

Nella programmazione ad oggetti, riveste una grande importanza la definizione dei livelli di visibilità che vengono assegnati ai metodi e alle proprietà di una classe. Infatti, è proprio attraverso la corretta definizione della visibilità applicata ad ogni singolo oggetto che si realizzano gli importanti e vantaggiosi risultati elencati nei paragrafi precedenti, in cui si sono illustrati i capisaldi di OOP.

Ad esempio, è facile comprendere che se un oggetto mettesse a completa disposizione del mondo esterno tutti i suoi metodi e tutte le sue proprietà si perderebbe nel nulla il concetto di incapsulamento. È un po' come se, nel mondo reale, un masterizzatore venisse venduto dando all'utente il completo accesso ai suoi componenti elettronici!

Per assegnare un determinato livello di visibilità ad una proprietà o ad un metodo è necessario utilizzare quelli che in OOP vengono definiti "access specifiers" (**specificatori di accesso** o anche descrittori di visibilità).

Esistono, in generale, quattro descrittori di visibilità. Alcuni linguaggi, tuttavia, ne utilizzano tre, non considerando il package:

•**public**

•**protected**

•**private**

•**package**

In generale, poi, possiamo considerare utile valutare i **livelli di visibilità** di metodi e proprietà nei seguenti casi:

•**Classe**

•**SottoClasse**

•**Mondo Esterno**

•**Package**

Il concetto di **Package** potrebbe risultare nuovo a chi non ha mai utilizzato un linguaggio di programmazione ad Oggetti. Esso rappresenta un insieme di classi e interfacce che operano nello stesso contesto e che sono raggruppate per consentire un utilizzo più organizzato ed efficiente delle stesse.

In altre parole, un package può essere tranquillamente visto come una libreria di classi che possono essere utilizzate ogniqualvolta se ne presenti la necessità. È, altresì, possibile annidare dei package all'interno di altri package in modo da ottimizzare la suddivisione delle classi (e interfacce).

Il linguaggio Java ha introdotto il concetto di package ed alcuni classici esempi sono i seguenti: java.lang; system.out; javax.swing. Altri linguaggi più recenti, come il C# o i l VB.Net utilizzano la denominazione namespace per definire un concetto del tutto analogo.

Mettendo, ora, in una tabella i descrittori di visibilità da una parte e i contesti in cui essi operano dall'altra, è possibile definire con precisione tutte le casistiche di visibilità su una classe:

Descrittore	Classe	Package	Sottoclasse	Mondo Esterno
private	Si	No	No	No
package	Si	Si	No	No

protected	Si	Si	Si	No
public	Si	Si	Si	Si

Vediamo come **interpretare** la precedente tabella. Nella colonna Classe,ad esempio, è indicata la visibilità che una classe ha dei suoi metodi e delle sue proprietà definiti utilizzando i descrittori presenti nella colonna iniziale.
Come si vede, in tale circostanza, una **classe** ha sempre accesso e visibilità a tutti i suoi metodi e proprietà a prescindere da quali siano gli access specifiers utilizzati.
Nel caso del **Package** la situazione è simile: ovvero, tutte le altre classi appartenenti allo stesso Package di una particolare classe hanno sempre accesso ai metodi e proprietà di quest'ultima tranne nel caso in cui siano dichiarati private.
Nella colonna **SottoClasse**, invece, viene evidenziato come una classe B figlia di una classe A abbia accesso soltanto ai metodi e proprietà di A che sono definiti public o protected.
Infine, l'ultima colonna (**Mondo Esterno**) evidenzia come soltanto gli attributi e le proprietà di una classe che siano definite public possano essere visibili dall'esterno. In particolare, con la nomenclatura Mondo Esterno si suole identificare ogni classe che non rientri nelle precedenti casistiche esaminate (non sia una sottoclasse della classe in questione e non appartenga allo stesso package).
La **scelta degli access specifiers**, quando si definisce una classe è tutt'altro che trascurabile. Da essa dipende fortemente la struttura ad oggetti del progetto che si vuole creare e, conseguentemente, l'efficacia dell'implementazione del codice.
Se, ad esempio, si definissero tutti i metodi e le proprietà di un oggetto come public, si perderebbe immediatamente il concetto di incapsulamento in quanto ogni elemento (proprietà o metodo) sarebbe sempre visibile dal mondo esterno. È un po', riprendendo sempre l'esempio del masterizzatore, come se tutti i dettagli interni hardware e software che costituiscono tale dispositivo fossero sempre noti e visibili.
Viceversa, una classe che abbia tutti i metodi (compresi i costruttori) e le proprietà definite come private sarebbe una classe inutilizzabile (esiste tuttavia un tipo particolare di classe, denominata **singleton**, che vedremo a breve, il cui costruttore è private) poiché nessuno potrebbe avvalersi delle sue caratteristiche.
Vediamo qualche esempio in Java che faciliti la comprensione di quanto esposto:

```
package Prova;
public class A
{
  //Proprietà della classe
  private int private_int = 1;
  //nessun acces specifier = package access
  int package_int = 2;
  protected int protected_int = 3;
  public int public_int = 4;

  //Metodi
  private void privateMethod()
```

```
  {
    System.out.println("output con accesso Private");
  }

  void packageMethod()
  {
    System.out.println("output con accesso Package");
  }

  protected void protectedMethod()
  {
    System.out.println("output con accesso Protected");
  }

  public void publicMethod()
  {
    System.out.println("output con accesso Public");
  }

  public static void main(String[] args)
  {
    A a = new A (); // Crea un oggetto di classe A
    a.privateMethod();
    a.packageMethod();
    a.protectedMethod();
    a.publicMethod();

    System.out.println("private_int: " + a.private_int);
    System.out.println("package_int: " + a.package_int);
    System.out.println("protected_int: " + a.protected_int);
    System.out.println("public_int: " + a.public_int);
  }
}
```

L'output dell'esempio precedente sarà il seguente:

output con accesso Private

output con accesso Package

output con accesso Protected

output con accesso Public

private_int: 1

package_int: 2

protected_int: 3

public_int: 4

Tale output mette in evidenza come una classe abbia sempre accesso a tutte le sue proprietà e metodi, a prescindere dall'access specifier definito per essi.

Il prossimo esempio, invece, illustra la situazione indicata dalla seconda colonna della tabella precedente, ovvero l'accesso di una classe ai metodi e alle proprietà di un'altra classe appartenente allo stesso package:

```
package Prova;


public class B
{
  public static void main(String[] args)
  {
    A a = new A();
    a.privateMethod(); //Errore!!!
    a.packageMethod();
    a.protectedMethod();
    a.publicMethod();

    //Errore
    System.out.println("private_int: " + a.private_int);

    System.out.println("package_int: " + a.package_int);
    System.out.println("protected_int: " + a.protected_int);
    System.out.println("public_int: " + a.public_int);
  }
}
```

Come si può notare, ci sono un paio di righe di codice evidenziate in rosso in cui viene messo in luce un errore. Infatti, se si provasse a compilare un programma del genere il compilatore identificherebbe un paio incongruenze causate proprio dalle due linee in rosso.

In queste due linee si sta provando ad accedere ad un metodo e ad una proprietà della classe A (definita nell'esempio iniziale) entrambi identificati da un access specifier di tipo private. Questo è l'unico caso in cui si genera un errore poichè tutti gli altri accessi (package, protected e public) non danno alcun problema.

Passiamo ad analizzare il comportamento descritto dalla terza colonna della tabella: l'accesso di una sottoclasse alle proprietà e ai metodi della sua classe padre (nel nostro esempio la sottoclasse appartiene ad un package diverso da quello della clase A per non ricadere nella casistica dell'esempio precedente):

```
package ProvaB;
import Prova.*;

public class C extends A
{
  public static void main(String[] args)
  {
    A a = new A();
    a.privateMethod(); // Errore
    a.packageMethod(); // Errore
    a.protectedMethod(); // Errore
    a.publicMethod();

    //Errore
    System.out.println("private_int: " + a.private_int);
    //Errore
    System.out.println("package_int: " + a.package_int);
    //Errore
    System.out.println("protected_int: "+ a.protected_int);

    System.out.println("public_int " + a.public_int);

    C c = new C();
    c.protectedMethod();
    System.out.println("protected_int: " + c. protected_int);
  }
}
```

Come si vede, ora gli errori di compilazione sono parecchi (sempre quelli evidenziati in rosso). Vale la pena di soffermarsi sul terzo e sul sesto errore, ovvero quello sulla invocazione del metodo protectedMethod()e sul tentativo di accesso alla variabile protected_int.

Infatti, dalle informazioni presenti sulla tabella iniziale ci si aspetterebbe che ad una classe figlia sia consentito far riferimento alle proprietà e ai metodi della classe padre quando su di essi è utilizzato l'access specifier protected. In effetti, ciò è vero nel senso che è consentito ad una istanza di una classe figlia far riferimento ai metodi e alle proprietà implementate nella classe padre ma non è permesso accedere ad essi attraverso una istanza della classe padre. Le ultime tre righe di codice evidenziano proprio tale concetto.

L'ultimo esempio è quello correlato all'accesso del "Mondo Esterno" ai membri di una classe. Viene utilizzata, in tal caso, un'altra classe non derivante da A e appartenente ad un altro package:

```
package ProvaC;
import Prova.*;
public class D
{
  public static void main(String[] args)
  {
    A a = new A();
    a.privateMethod(); //Errore
    a.packageMethod(); //Errore
    a.protectedMethod(); //Errore
    a.publicMethod();

    //Errore
    System.out.println("private_int: " + a.private_int);
    //Errore
    System.out.println("package_int: " + a.package_int);
    //Errore
    System.out.println("protected_int: "+ a.protected_int);

    System.out.println("public_int: " + a.public_int);
  }
}
```

Anche qui, le uniche righe di codice immuni da errori sono soltanto quelle in cui viene fatto accesso al metodo e alla proprietà con specificatore di accesso public. Tutte le altre righe sono errate e non verranno accettate dal compilatore.

15. Il singleton

Il singleton rappresenta un tipo particolare di classe che garantisce che soltanto un'unica istanza della classe stessa possa essere creata all'interno di un programma.

Per ottenere un siffatto comportamento è necessario avvalersi dello specificatore di accesso «private» anche per il costruttore della classe (cosa che generalmente non viene mai praticata in una classe "standard") ed utilizzare un metodo statico che consenta di accedere all'unica istanza della classe.

Vediamo un esempio in Java:

```
public class Singleton
{
  private static Singleton istanza;

  private Singleton()
```

```
  {
  }

  public static Singleton getInstance()
  {
    if (istanza == null)
    {
      istanza = new Singleton();
    }

    return istanza;
  }

  public void helloWorld()
  {
    System.out.println("Hello World");
  }
}


public class usaSingleton
{
  public static void main(String args[])
  {
    Singleton.getInstance().helloWorld();
  }
}
```

Come si può notare dal codice, il costruttore della classe Singleton è stato definito con access specifier private e, in tal modo, l'unico punto di accesso alla classe per il mondo esterno viene fornito attraverso il metodo statico getInstance() che si occupa di restituire (creandola prima se non è mai stata creata) l'unica istanza della classe.
Il lettore critico potrebbe domandarsi, giustamente, quando possa rivelarsi utile avvalersi dei singleton. In generale, la scelta del singleton viene effettuata in tutti quei casi in cui è necessario che venga utilizzata una sola istanza di una classe.
Ciò consente di:

•Avere un accesso controllato all'unica istanza della classe

•Avere uno spazio di nomi ridotto

•Evitare la dichiarazione di variabili globali

•Assicurarsi di avere un basso numero di oggetti utilizzati in condivisione grazie al fatto che viene impedita la creazione di nuove istanze ogni volta che si voglia utilizzare la stessa classe.

Implementare una classe singleton

Ecco come si implementa una classe singleton in alcuni dei linguaggi orientati agli oggetti per il web.

- singleton in C#
- singleton in C++
- singleton in VB.NET
- singleton in Actionscript
- singleton in Javascript
- singleton in Python

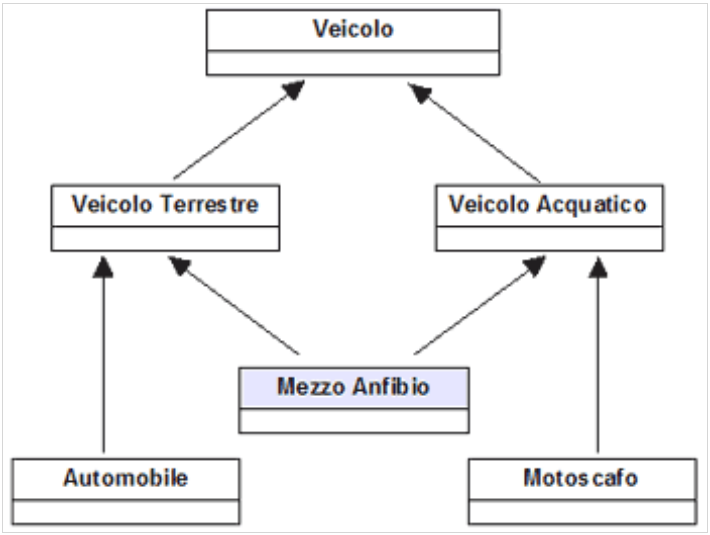
D. Nozioni avanzate

16. L'Ereditarietà multipla

Si è visto come tra i capisaldi della programmazione ad oggetti, l'ereditarietà svolga una funzione molto importante e faciliti parecchio il disegno e la codifica dei programmi. Esistono, però, delle circostanze in cui può aver senso far derivare una classe da più classi padre: è il caso dell'ereditarietà multipla.

Supponiamo di voler implementare una classe Mezzo Anfibio (un tipo di veicolo capace di camminare sia sulla terra che sull'acqua). Avvalendoci dei discorsi di astrazione dei dati ed ereditarietà fatti in precedenza potremmo tracciare il seguente diagramma di classi:

Figura 1. Class Diagram con eredità multipla



Come è facile intuire osservando la figura (in cui la gerarchia di eredità è ora rappresentata da un grafo aciclico), la classe Mezzo Anfibio, per le sue caratteristiche particolari, erediterà i metodi e le proprietà con access specifier public e protected sia dalla classe Veicolo Terrestre che dalla classe Veicolo Acquatico.

In generale, c'è da dire che l'ereditarietà multipla pur rappresentando una notevole potenzialità nel mondo Object Oriented, favorendo notevolmente la flessibilità e il riutilizzo del codice, viene solitamente considerata un approccio da evitare a causa della complessità che può derivare da una siffatta architettura. In particolare, i due problemi principali che possono sorgere quando si utilizza l'eredità multipla sono i seguenti:

•Ambiguità dei nomi

•Poca efficienza nella ricerca dei metodi definiti nelle classi

Il primo problema (**Ambiguità dei nomi**) può verificarsi se una proprietà o un metodo ereditato è definito con lo stesso nome in tutte le classi padre di una data classe. Ad esempio, rifacendoci all'esempio della figura precedente, se la Classe Veicolo Terrestre e la classe Veicolo Acquatico implementassero entrambe i metodi svolta_a_dx() e svolta_a_sx()ciò porterebbe ad un problema di ambiguità per la classe Mezzo_Anfibio nell'ereditare tali metodi da entrambe le classi.

Il secondo problema (**Poca Efficienza nella ricerca dei metodi definiti nelle classi**) è causato dalla nuova struttura che assume l'architettura in questo tipo di approccio che, come detto, è adesso rappresentata da un grafo. Infatti, con l'utilizzo di una struttura a grafo non è più possibile utilizzare la ricerca lineare (ideale sulle strutture ad albero) per l'individuazione dei metodi ma è necessario effettuare una sorta di "backtracking" sul grafo stesso.

L'ereditarietà multipla, proprio per le controverse questioni riguardanti il suo utilizzo, non è implementata in tutti i linguaggi di programmazione ad oggetti. Il C++ e Python rappresentano due esempi di linguaggi che consentono l'utilizzo di tale implementazione mentre linguaggi come Java, C# e VB.Net eliminano il problema alla radice, non consentendo ad alcuna classe di avere più di una classe padre.

Implemetare l'ereditarietà multipla

Ecco come si implementa l'ereditarietà multipla in alcuni dei linguaggi orientati agli oggetti per il web.

•ereditarietà multipla in C++

•ereditarietà multipla in Python

17. Le Interfacce

Le incertezze circa l'utilità della ereditarietà multipla nei programmi Object Oriented hanno portato alla definizione di una strada alternativa ma sicuramente più efficiente. Infatti, con l'avvento di Java è stato introdotto il concetto di interfaccia.

In generale, un interfaccia rappresenta una sorta di "promessa" che una classe si impegna a mantenere. La promessa è quella di implementare determinati metodi di cui viene resa nota soltanto la definizione (un po' come si è già visto per le classi ed i metodi astratti). Ciò che è importante non è tanto come verranno implementati tali metodi all'interno della classe ma, piuttosto, che la denominazione ed i parametri richiesti siano assolutamente rispettati.

Si supponga, ad esempio, di voler creare un'interfaccia denominata StackInterface che debba essere utilizzata da ogni classe che desideri definire la classica struttura dati a pila (LIFO). Una siffatta interfaccia sarà definita nel seguente modo (sintassi Java):

```
public interface StackInterface
{
```

```
  public void push(int i);
  public int pop();
}
```

Come si vede, sono stati definiti due metodi (push e pop), corrispondenti rispettivamente alle due operazioni di inserimento ed eliminazione di elementi dalla pila ma non ne viene in alcun modo fornita l'implementazione. Supponiamo, ora, di voler definire una classe Stack che contenga il codice necessario per la corretta gestione di una pila. Tale classe sarà definita nel seguente modo:

```
public class Stack implements StackInterface
{
  public void push(int i)
  {
   … ….
  }
  public int pop()
  {
   … ….
  }
}
```

Come si potrà osservare, all'interno della classe Stack sono implementati i metodi che sono stati definiti all'interno dell'interfaccia StackInterface. Poiché la classe Stack promette di fare uso dell'interfaccia StackInterface (lo si può notare dalla istruzione "implements StackInterface"), qualora il programmatore omettesse di definire e implementare uno o entrambi questi metodi, si otterrebbe un errore di compilazione.
Sebbene le interfacce non vengano istanziate, come avviene per le classi, esse conservano determinate caratteristiche che sono simili a quelle viste nelle classi ordinarie. Ad esempio, una volta definita un'interfaccia, è possibile dichiarare un oggetto come se fosse del tipo dichiarato dall'interfaccia stessa utilizzando la medesima notazione utilizzata per la dichiarazione di variabili.
Inoltre, allo stesso modo delle classi, è possibile utilizzare l'ereditarietà anche per le interfacce, ovvero definire una interfaccia che estenda le caratteristiche di un'altra, aggiungendo altri metodi all'interfaccia padre.
Infine, una classe può implementare più di una interfaccia. Ovvero, è possibile obbligare una classe ad implementare tutti i metodi definiti nelle interfacce con le quali essa è legata. Questa ultima caratteristica fornisce, indiscutibilmente, la massima flessibilità nella definizione del comportamento che si desidera attribuire ad una classe

18. Coesione e accoppiamento

Uno degli errori più comuni che viene commesso spesso dai programmatori poco esperti in OOP è quello di utilizzare una sorta di approccio "misto", ovvero di programmare ad oggetti ricorrendo talvolta alle vecchie abitudini proprie della programmazione procedurale.

Soprattutto i linguaggi come il C++, per compatibilità con il C, consentono di utilizzare questa modalità ibrida che rappresenta, sicuramente, una delle cose da evitare. Questa riflessione serve, tra l'altro, a mettere in luce l'importanza che riveste la qualità del codice quando si lavora in ambiente Object Oriented.
Due dei principali fattori da cui dipende una buona qualità del codice sono i seguenti:

•Accoppiamento (Coupling)

•Coesione (Cohesion)

L'**Accoppiamento** fa riferimento ai legami esistenti tra unità (classi) separate di un programma. In generale, diremo che se due classi dipendono strettamente l'una dall'altra (ovvero hanno molti dettagli che sono legati vicendevolmente) allora esse sono strettamente accoppiate (si parla anche di strong coupling).
Riflettendo un attimo su quanto detto nei paragrafi precedenti, quando si è parlato di incapsulamento, manutenzione e riutilizzo del codice, si può facilmente arguire che per una buona qualità del codice l'obiettivo sarà, dunque, quello di puntare ad un basso accoppiamento (weak coupling o loose coupling), consentendo in tal modo una migliore manutenibilità del software.
Infatti, un **basso accoppiamento** consente sicuramente di avere una buona comprensione del codice associato ad una classe senza doversi preoccupare di andare a reperire i dati delle altre classi coinvolte. Inoltre, utilizzando un basso accoppiamento, eventuali modifiche apportate ad una classe avranno poche o nessuna ripercussione sulle altre classi con cui è instaurata una relazione di dipendenza.
Per fare un esempio di basso accoppiamento nel mondo reale, si può pensare ad una Radio connessa con degli Altoparlanti attraverso l'uso di un cavo. Sostituendo o modificando il cavo, le due entità (Radio e altoparlanti) non subiranno alcuna modifica sostanziale alle loro strutture. Viceversa, un forte accoppiamento può essere rappresentato da due travi di acciaio saldate tra di loro. Infatti, per poter muovere una trave, anche l'altra subirà inevitabilmente degli spostamenti.
La **Coesione**, invece, rappresenta una informazione sulla quantità e sulla eterogeneità dei task di cui una singola unità (una classe o un metodo appartenente ad una classe) è responsabile. In altre parole, attraverso la coesione si è in grado di stabilire quali e quanti siano i compiti per i quali una classe (o un metodo) è stata disegnata. In generale, si può affermare che più una classe ha una responsabilità ristretta ad un solo compito più il valore della coesione è elevato; in tal caso si parlerà di alta coesione (strong cohesion).
Come si evince dalle definizioni appena fornite, a differenza dell'accoppiamento, il concetto di coesione può essere applicato sia alle classi che ai metodi.
È proprio l'**alta coesione** l'obiettivo da prefiggersi quando si vuole scrivere del codice di buona qualità. Infatti, il raggiungimento di un'alta coesione ha svariati vantaggi. In particolare: semplifica la comprensione relativamente ai compiti propri di una classe o di un metodo, facilita l'utilizzo di nomi appropriati e favorisce il riutilizzo delle classi e dei metodi

E. Principi di progettazione

19. Il Processo di sviluppo in OOP

Nella «guida UML», presente su HTML.IT, viene definito e discusso il **metodo RAD** (Rapid Application Development), utilizzato per lo sviluppo rapido di applicazioni. In questo paragrafo riportiamo i passi più significativi di tale metodologia, mettendo in risalto quelli più vicini allo sviluppo Object Oriented.

•**Identificazione delle classi e oggetti**: Questa è la fase di analisi in cui vengono effettuati degli incontri con il cliente per stabilire ed identificare in modo non ancora definitivo le classi che saranno coinvolte nel progetto.

•**Definizione della semantica delle classi**: In questo stadio del processo si prosegue assegnando ad ogni classe una semantica ben precisa. Al termine di questa fase dovrebbe essere ben chiara la struttura delle classi e delle interfacce eventualmente da implementare.

•**Relazioni tra le classi**: Dopo aver definito le classi del progetto è molto importante identificare ed indicare tutte le tipologie di relazione che intercorrono tra di esse.

•**Implementazione del Codice**: Lo stadio della scrittura del codice è, ovviamente, di enorme importanza. Talvolta, durante la stesura del codice possono essere evidenziate delle incongruenze di analisi o disegno che vanno notificate e riviste per poi procedere di nuovo alla codifica.

Per ulteriori approfondimenti sul RAD, si consiglia di far riferimento alla «guida UML», presente su HTML.IT.