La programmazione orientata agli oggetti

A) INTRODUZIONE

A1 Premessa

Lo scopo principale di questo documento 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 del documento).

Pertanto questo documento 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 C (o 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.

A2 Brevi cenni storici

La Programmazione ad Oggetti rappresenta, senza dubbio, il **modello di programmazione più diffuso** ed utilizzato degli ultimi 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**, considerato 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.**

A3 Tecniche di Programmazione a confronto

Si è precedentemente 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

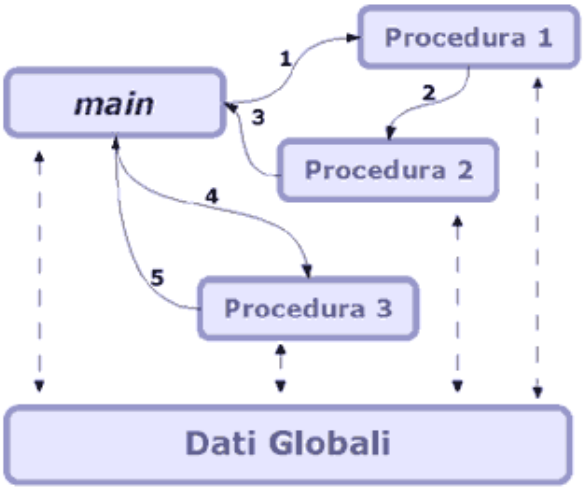
Un notevole passo in avanti, rispetto alla Programmazione Non Strutturata, venne fatto con l'avvento della Programmazione Procedurale. Il concetto base qui è quello di raggruppare i pezzi di programma ripetuti in porzioni di codice utilizzabili e richiamabili ogni volta che se ne presenti l'esigenza: **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 può facilitare la comprensione di tale concetto:

Figura 2. Flusso di un programma con programmazione procedurale



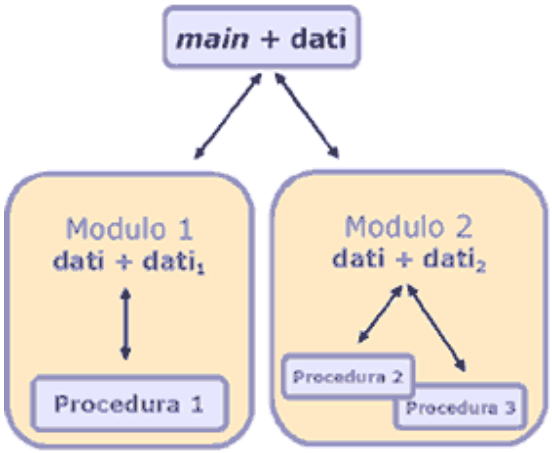
Il grande vantaggio della programmazione procedurale, rispetto alla precedente non strutturata consiste in un notevole **abbattimento del numero di errori**, che deriva dal fatto che se una procedura è corretta allora vi è la certezza che essa restituirà ad ogni invocazione dei risultati corretti in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

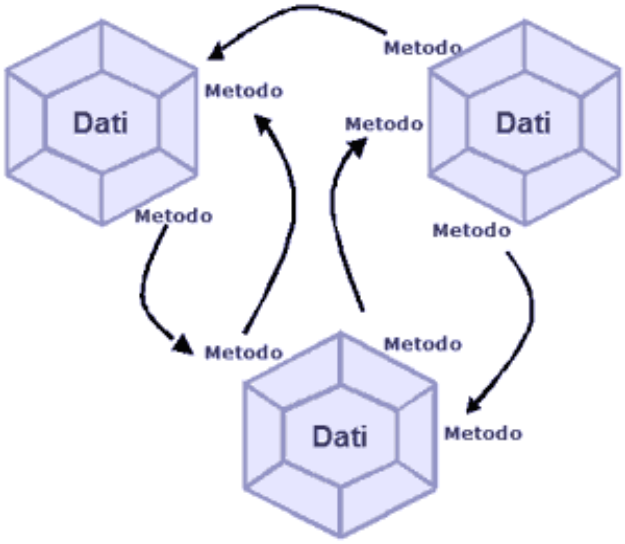
La programmazione procedurale/modulare ha rappresentato il punto di riferimento nello sviluppo applicativo per tanti anni. Gradualmente ma inevitabilmente però, man mano che gli orizzonti della programmazione diventavano sempre più ampi, si andarono evidenziando i **limiti** di tale metodologia.

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

B1 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.

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.

N.B. 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.

B2 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).

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).

B3 Le Classi

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

Esempio: 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.

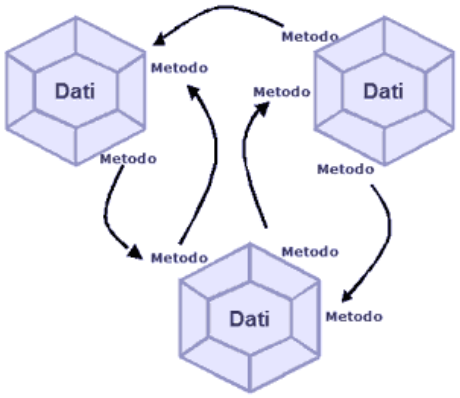
B4 I Messaggi

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 4. 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 allocandolo dinamicamente in memoria (heap). Essi devono essere richiamati ogni volta 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 allocata dinamicamente (heap).
N.B. 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.

B5 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 5. 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.

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

C) I CAPISALDI DELLA PROGRAMAMZIONE AD OGGETTI

C1 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 ogni qualvolta se ne presenti la necessità.

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.

C2 INCAPSULAMENTO (il primo principio fondamentale paradigma OOP)

L'incapsulamento è il primo principio dell'OOP ed 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.

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.

Esempio: Implementare l'incapsulamento in C++

```
//*********************************
// file cubo.h
//*********************************
class Cubo
{
        // Dichiarazione delle proprietà: si noti che sono definite tutte private.
        private:
                int lunghezza;
                int larghezza;
                int altezza;
        // Dichiarazione dei metodi Mutator e Accessor
        public:
                void setLunghezza(int lun);
                void setLarghezza(int lar);
                void setAltezza(int alt);
                int getLunghezza();
                int getLarghezza();
                int getAltezza();
                void visualizzaVolume();
};

//*********************************
// file cubo.cpp
//*********************************
#include <iostream>
#include "cubo.h"
// Implementazione dei Metodi "Mutator" e "Accessor"
void Cubo::setLunghezza(int lun)
{
        lunghezza = lun;
}
void Cubo::setLarghezza(int lar)
{
        larghezza = lar;
}
void Cubo::setAltezza(int alt)
{
        altezza = alt;
}
int Cubo::getLunghezza()
{
        return lunghezza;
}
int Cubo::getLarghezza()
{
        return larghezza;
}
int Cubo::getAltezza()
{
        return altezza;
```

```
}

// Metodo pubblico che visualizza il volume del cubo, usando le proprietà
// interne della classe
void Cubo::visualizzaVolume()
{
         int vol = lunghezza * larghezza * altezza;
         cout << "Volume del cubo: " << vol <<endl;
}
```

Esempio: Implementare l'incapsulamento in VB.NET

```
Public Class Cubo
    private lunghezza as Integer
    private larghezza as Integer
    private altezza as Integer
    Public Property Lunghezza as Integer
        Get
            Return lunghezza
        End Get
        Set(ByVal Value as Integer)
            lunghezza = Value
        End Set
    End Property
    Public Property Larghezza as Integer
        Get
            Return larghezza
        End Get
        Set(ByVal Value as Integer)
            larghezza = Value
        End Set
    End Property
    Public Property Altezza as Integer
        Get
            Return altezza
        End Get
        Set(ByVal Value as Integer)
            altezza = Value
        End Set
    End Property
    // Metodo pubblico che visualizza il volume del cubo, usando le proprietà
    // interne della classe
    Public Sub visualizzaVolume()
        System.Console.WriteLine("Volume: " + lunghezza * larghezza *   altezza)
    End Sub
End Class
```

C3 EREDITARIETA' (il secondo principio fondamentale paradigma OOP)

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).

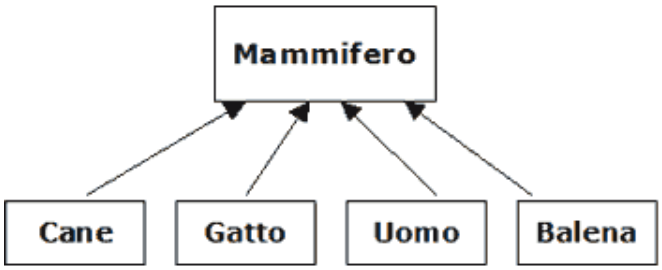
È semplice poter osservare esempi di ereditarietà nel mondo reale.

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 6. 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.

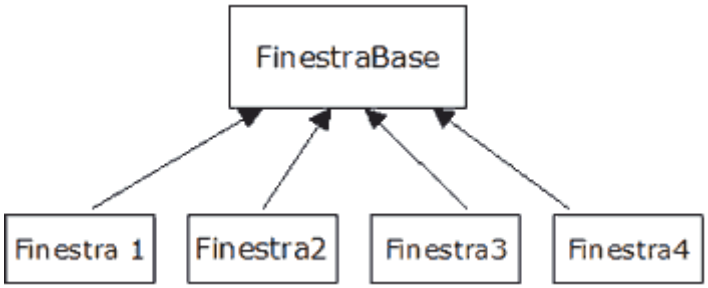
Ad esempio, se i mammiferi diventassero improvvisamente (e anche inverosimilmente!) degli animali a sangue freddo, soltanto l'oggetto padre mammifero necessiterà di tale variazione. Il gatto il cane, l'uomo, la balena, e tutti gli altri oggetti figli erediterebbero automaticamente la caratteristica di 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 7. 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**.

Esempio: Implementare l'ereditarietà in C++

```
//*********************************
// file contatore.h
//*********************************
class Contatore
{
    public:
        Contatore();
        Contatore(int valoreIniziale);
        int getValoreContatore();
        void incrementaContatore();
    private:
        int valoreContatore;
};

//*********************************
// file contatore.cpp
//*********************************
#include "contatore.h"
Contatore::Contatore()
{
    valoreContatore = 0;
}
Contatore::Contatore(int valoreIniziale)
{
    valoreContatore = valoreIniziale;
}
int Contatore::getValoreContatore()
{
    return valoreContatore;
}
void Contatore::incrementaContatore()
{
    valoreContatore++;
}

//*********************************
// file contatoredoppio.h
//*********************************
#include "contatore.h"
class ContatoreDoppio : public Contatore
{
    public:
        ContatoreDoppio(int valoreIniziale);
        void incrementaContatore();
};

//*********************************
// file contatoredoppio.cpp
//*********************************
#include "contatoredoppio.h"
// Si noti la chiamata al costruttore della classe padre, dopo i due punti
ContatoreDoppio::ContatoreDoppio(int valoreIniziale) : Contatore(valoreIniziale)
{
}
void ContatoreDoppio::incrementaContatore ()
{
    // Viene chiamato il metodo incrementaContatore della classe padre
    // utilizzando lo scope operator ::
    for (int i =0; i < 2; i++)
    {
        Contatore::incrementaContatore();
```

```
    }
}

//*********************************
// file main.cpp
//*********************************
#include "contatoredoppio.h"
#include < iostream>
main()
{
    Contatore cont1(3);
    ContatoreDoppio cont2(3);
     cout << cont1.getValue() << endl;
     cout << cont2.getValue() << endl;
     cont1.incrementaContatore();
     cont2.incrementaContatore();
     cout << cont1.getValue() << endl;
     cout << cont2.getValue() << endl;
}
```

Esempio: Implementare l'ereditarietà in VB.NET

```
Public Class Impiegato
    private nome as String
    private salario as Double
    private matricola as String
    private anniDiServizio as Integer
    Public Sub New(n As String, s as Double, m as String, ads as Integer)
        nome = n
        salario = s
        matricola = m
        anniDiServizio = ads
    End Sub
    Public Sub incrementaSalario(double percentuale)
        salario *= 1 + percentuale / 100
    End Sub
    Public Sub stampaInfo()
        System.Console.WriteLine (nome + " " + salario + " " + matricola)
    End Sub
    Public ReadOnly Property Nome as String
        Get
         return nome
        End Get
    End Property
    Public AnniServizio as Integer
        Get
         return anniDiServizio
        End Get
    End Property
End Class

Public Class Manager Inherits Impiegato
    Private nomeSegretaria as String
    Public Sub New(n as String, s as Double, m asString, ads as Integer)
        MyBase.New(n, s, m, ads)
        nomeSegretaria = String.empty
    End Sub
    Public Sub incrementaSalario(percentuale as Double)
        ' Aggiunge alla percentuale lo 0.5% per ogni anno di servizio
        Dim bonus as Double = 0.5 * AnniServizio
        MyBase.incrementaSalario(percentuale + bonus)
    End Sub
    Public Property Segretaria as String
```

```
        Get
            return nomeSegretaria
        End Get
        Set(ByValue Value as String)
           nomeSegretaria = Value
        End Set
    End Property
End Class
```

C.4 IL POLIMORFISMO (il terzo principio fondamentale paradigma OOP)

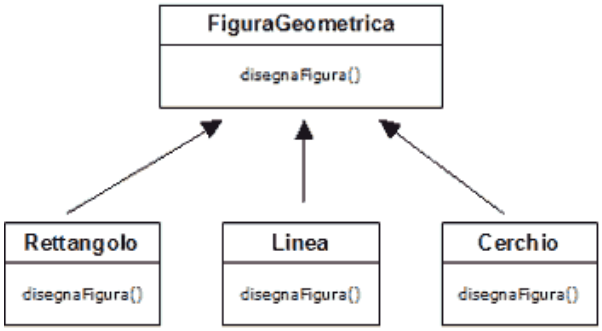
Il terzo elemento fondamentale della programmazione ad Oggetti è il polimorfismo Letteralmente, la parola polimorfismo indica la possibilità per uno stesso oggetto di assumere più forme.

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 8. 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.

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.

Esempio: Implementare il polimorfismo in C++

```
class Albero
{
    protected:
        virtual void cresce();
        {
            cout << "Metodo cresce della classe Albero";
        }
};
class Abete : public Albero
{
    public:
        void cresce()
        {
             cout << "Metodo cresce della classe Abete";
        }
};
main()
{
    Albero* al = new Abete();
    al->cresce();
    delete al;
}
```

Esempio: Implementare il polimorfismo in VB.NET

```
Public class Albero
    Public Overridable Sub cresce()
        Console.WriteLine("Metodo cresce della classe Albero")
    End Sub
End Class
Public class Abete : Inherits Albero
    Public Overrides Sub cresce()
         Console.WriteLine("Metodo cresce della classe Abete")
    End Sub
End Class
```

```
Module Module1
    Sub Main()
        Dim alb As Albero
        Dim ab As Abete
        alb = New Albero()
        ab = New Abete()
        alb.cresce() ' output --> "Metodo cresce della classe Albero"
        ab.cresce()  ' output --> "Metodo cresce della classe Abete"
        alb = New Abete()
        alb.cresce() 'output --> "Metodo cresce della classe Abete"
    End Sub
End Module
```

C.5 ASTRAZIONE E CLASSI ASTRATTE

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.

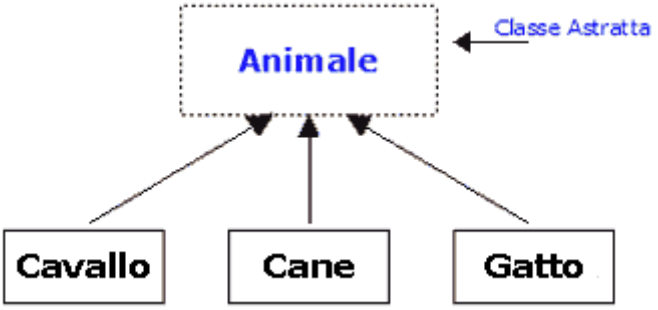
Una delle definizioni migliori sul concetto di Astrazione dei Dati è quella di Booch: «Un'astrazione deve denotare le caratteristiche essenziali di un oggetto contraddistinguendolo da tutti gli altri oggetti e fornendo, in tal modo, dei confini concettuali ben precisi relativamente alla prospettiva dell'osservatore»

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 9. 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.

Esempio: Implementare l'astrazione dei dati in C++

```
class Frutta
{
    public:
        virtual void siSbuccia() = 0;   //Metodo virtuale puro
        .....
};
class Mela : public Frutta
{
    private:
        bool buccia = true;
    public:
        bool siSbuccia()
        {
            return buccia;
        }
};
```

Esempio: Implementare l'astrazione dei dati in VB.NET

```
Public MustInherit Class A
   Public MustOverride Sub F()
End Class
Public MustInherit Class B : Inherits A
    Public Sub G()
       ..........
    End Sub
End Class
Public Class C : Inherits B
   public Overrides Sub F()
      ' Implementazione del metodo F
      ..........
   End Sub
End Class
```

C.6 LA VISIBILITA'

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.

C.7 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.

Lo studente 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.

Esempio: Implementare il singleton in C++

```
// Singleton.h
class Singleton
{
    public:
        static Singleton* Instance();
        void helloWorld();
    protected:
        Singleton();
    private:
        static Singleton* instanza;
};
// Singleton.cpp
#include < iostream>
#include "Singleton.h"
Singleton* Singleton::istanza = 0;
Singleton* Singleton::Instance ()
{
    if (istanza == 0)
        istanza = new Singleton;
    return istanza;
}
void Singleton::HelloWorld()
{
    cout << "Hello World!";
}
int main()
{
    Singleton *p = Singleton::Instance();
    p->HelloWorld();
    delete p;
}
```

Esempio: Implementare il singleton in VB.NET

```
Imports system
Class Singleton
    Public Shared istanza As Singleton
    Private Sub new()
    End Sub
     Public Shared Property Instance as Singleton
        Get
            If (istanza = Nothing) Then
                istanza = new Singleton()
            End If
            return istanza
        End Get
    End Property
    Public Sub helloWorld()
        Console.WriteLine("Hello World")
    End Sub
End Class
Module UsaSingleton
    Sub Main()
        Singleton.Instance.helloWorld()
    End Sub
End Module
```

D) NOZIONI AVANZATE

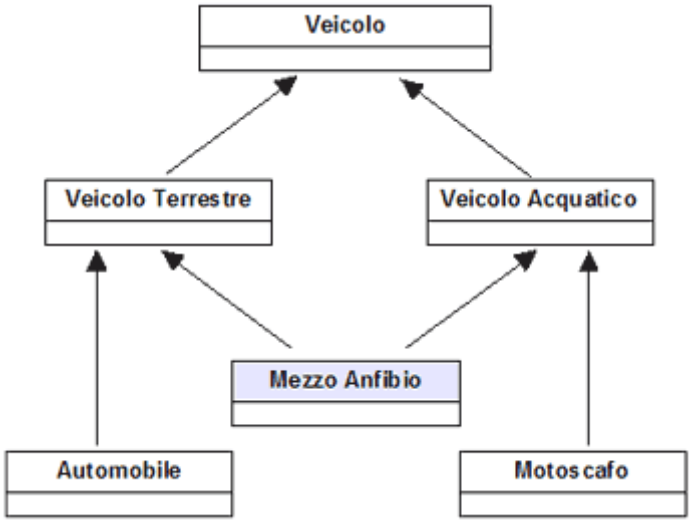
D1 EREDITARIETA' MULTIPLA

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 10. 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.

N.B. 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.

Esempio: Implementare l'ereditarietà multipla in C++

```
class Poligono
{
  protected:
    int larghezza, altezza;
  public:
    void setValori (int lar, int alt)
    {
       larghezza = lar;
       altezza = alt;
    }
};
class Output
{
  public:
     void output (int i)
     {
        cout << i
     }
};
class Rettangolo: public Poligono, public Output
{
  public:
    int area ()
    {
        return (larghezza * altezza);
    }
    void visualizzaArea()
    {
        output(area());
    }
};
```

D2 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.

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.

D3 COESIONE ED 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) PROCESSO DI SVILUPPO 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.

E1 UML cenni iniziali

Nel mondo costantemente in fermento ed in evoluzione quale e' quello dello sviluppo di applicazioni Object Oriented, diviene sempre più difficile costruire e gestire applicazioni di alta qualità in tempi brevi.

Come risultato di tale difficoltà e dalla esigenza di avere un linguaggio universale per modellare gli oggetti che potesse essere utilizzato da ogni industria produttrice di software, fu inventato il linguaggio **UML**, la cui sigla sta per **U**nified **M**odeling **L**anguage.

In termini pratici, potremmo dire che il linguaggio UML rappresenta, in qualche modo, una cianografia, un **progetto di ingegneria edile**, riportato nel mondo dell'Information Technology.

L'UML è, dunque, un metodo per descrivere l'architettura di un sistema in dettaglio.

Come è facile intuire, utilizzando una sorta di cianografia sarà molto più facile costruire o manutenere un sistema e assicurarsi che il sistema stesso si presterà senza troppe difficoltà a futuri cambiamenti dei requisiti.

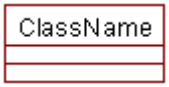
La forza dell'Unified Modeling Language consiste nel fatto che il processo di disegno del sistema può essere effettuata in modo tale che i clienti, gli analisti, i programmatori e chiunque altro sia coinvolto nel sistema di sviluppo possa capire ed esaminare in modo efficiente il sistema e prendere parte alla sua costruzione in modo attivo.

E2 UML: CLASS DIAGRAM

Si è detto precedentemente nella guida che gli oggetti da rappresentare di un qualsiasi sistema possono essere suddivisi in categorie e, quindi, in classi.

Il **Class Diagram** del linguaggio UML consiste di svariate classi connesse tra di loro tramite delle relazioni.

Prima di tutto, però, è importante definire graficamente una classe in UML:

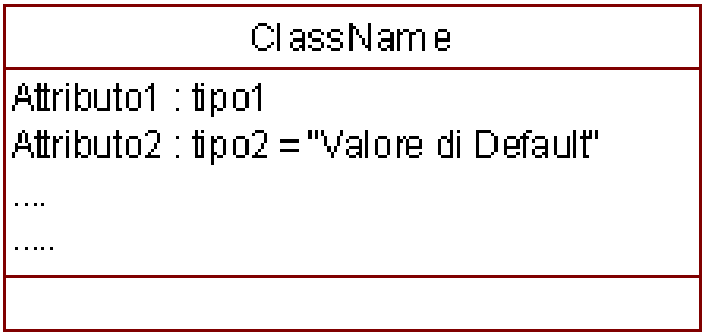


Una classe viene rappresentata da un rettangolo. Il nome della classe, per convenzione, è una parola con l'iniziale maiuscola ed appare vicino alla sommità del rettangolo. Se il nome della classe definita consiste di una parola composta a sua volta da più parole allora viene utilizzata la notazione in cui tutte le iniziali di ogni parola sono scritte in maiuscolo.

Definizione generica di una classe
Un attributo rappresenta una proprietà di una classe. Esso descrive un insieme di valori che la proprietà può avere quando vengono istanziati oggetti di quella determinata classe. Una classe può avere zero o più attributi.

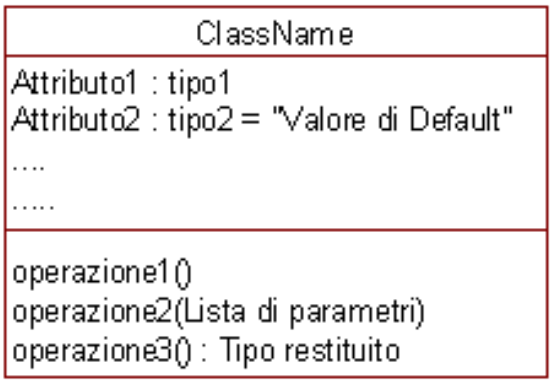
Un **Attributo** il cui nome è costituito da una sola parola viene scritto sempre in caratteri minuscoli. Se, invece, il nome dell'attributo consiste di più parole (es: Informazioni-Cliente) allora il nome dell'attributo verrà scritto unendo tutte le parole che ne costituiscono il nome stesso con la particolarità che la prima parola verrà scritta in minuscolo mentre le successive avranno la loro prima lettera in maiuscolo. Nell'esempio appena visto l'attributo sarà identificato dal termine: informazioniCliente.

La lista degli attributi di una classe viene separata graficamente dal nome della classe a cui appartiene tramite una linea orizzontale.



Nell'icona della classe, come si vede nella figura precedente, è possibile specificare un tipo in relazione ad ogni attributo (string, float, int, bool, ecc.). E' anche possibile specificare un valore di default che un attributo può avere.

Un'**Operazione** è un'azione che gli oggetti di una certa classe possono compiere. Analogamente al nome degli attributi, il nome di un'operazione viene scritto con caratteri minuscoli. Anche qui, se il nome dell'operazione consiste di più parole, allora tali parole vengono unite tra di loro ed ognuna di esse, eccetto la prima, viene scritta con il primo carattere maiuscolo. La lista delle operazioni (metodi) viene rappresentata graficamente sotto la lista degli attributi e separata da questa tramite una linea orizzontale.



Anche I metodi possono avere delle informazioni addizionali. Nelle parentesi che seguono il nome di un'operazione, infatti, è possibile mostrare gli eventuali parametri necessari al metodo insieme al loro tipo. Infine, se il metodo rappresenta una funzione è necessario anche specificare il tipo restituito.

Altre informazioni addizionali che possono essere unite agli attributi di una classe sono le "Constraints" e le Note.

Le "Constraints" sono delle caselle di testo racchiuse tra parentesi. All'interno delle parentesi viene specificata una o più regole che la classe è tenuta a seguire obbligatoriamente.

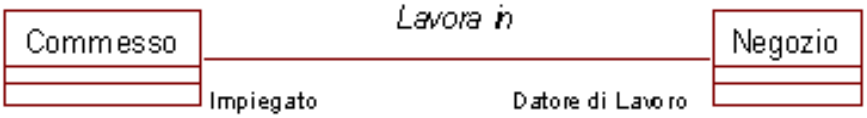
Le Note solitamente sono associate con gli attributi e/o con i metodi. Esse forniscono una informazione aggiuntiva ad una classe. Una nota può contenere sia elementi grafici che elementi di testo.

Come si può riuscire a discernere le classi da utilizzare dalla intervista con il cliente?
E' necessario, a tal fine, prestare particolare attenzione ad i nomi che i clienti usano per descrivere le entità del loro business. Tali nomi saranno ottimi candidati per diventare delle classi nel modello UML. Si deve prestare, altresì, attenzione ai verbi che vengono pronunciati dai clienti. Questi costituiranno, con molta probabilità, i metodi (le operazioni) nelle classi definite. Gli attributi di una classe verranno stabiliti con le analoghe modalità utilizzate per i nomi delle classi.

E3 Associazioni

Quando più classi sono connesse l'una con l'altra da un punto di vista concettuale, tale connessione viene denominata associazione.

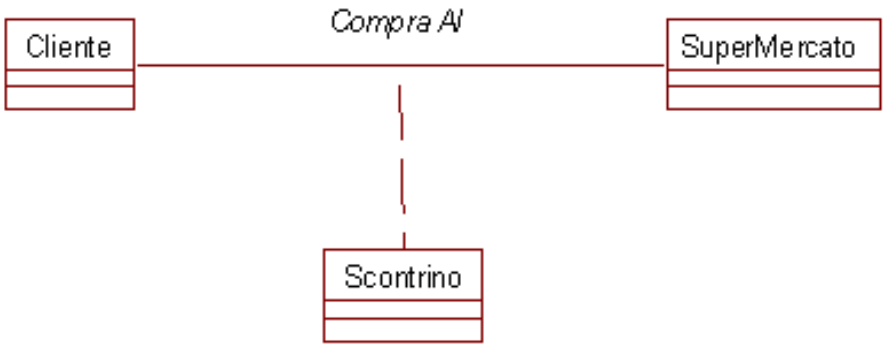
Graficamente, un'associazione viene rappresentata con una linea che connette due classi, con il nome dell'associazione appena sopra la linea stessa.



Quando una classe si associa con un'altra, ognuna di esse gioca un ruolo all'interno dell'associazione. E' possibile mostrare questi ruoli sul diagramma, scrivendoli vicino la linea orizzontale dalla parte della classe che svolge un determinato ruolo, come si vede nella figura precedente.

Un'associazione può essere più complessa del concetto di connettere una classe ad un'altra. E' possibile, infatti che più classi possano connettersi ad una singola classe.

Qualche volta un'associazione tra due classi deve seguire una regola. Tale regola si indica inserendo una "constraint" vicino la linea che rappresenta l'associazione.



Esattamente come una classe, un'associazione può avere attributi ed operazioni. In questo caso si parla di una Classe Associazione (vedasi figura precedente).

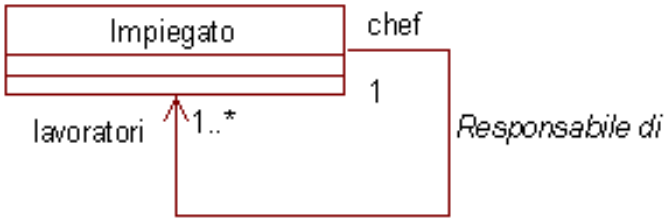
Le classi associazione vengono visualizzate allo stesso modo con cui si mostra una classe normale. Si utilizza una linea tratteggiata per connettere una Classe Associazione alla linea di associazione.

La molteplicità è un tipo speciale di associazione in cui si mostra il numero di oggetti appartenenti ad una classe che interagisce con il numero di oggetti della classe associata. Una classe, in generale, può essere correlata ad una altra nei seguenti modi:

- Uno ad uno
- Uno a molti
- Uno ad uno o più
- Uno a zero o uno
- Uno ad un intervallo limitato (es.: 1 a 2 – 20)
- Uno ad un numero esatto n
- Uno ad un insieme di scelte (es.: 1 a 5 o 8)

Il linguaggio UML utilizza un asterisco (*) per rappresentare le opzioni "molti" e "più".

Qualche volta, una classe può trovarsi in associazione con se stessa. Ciò si verifica quando una classe contiene oggetti che possono giocare svariati ruoli. Tali associazioni sono chiamate: "Associazioni Riflessive".



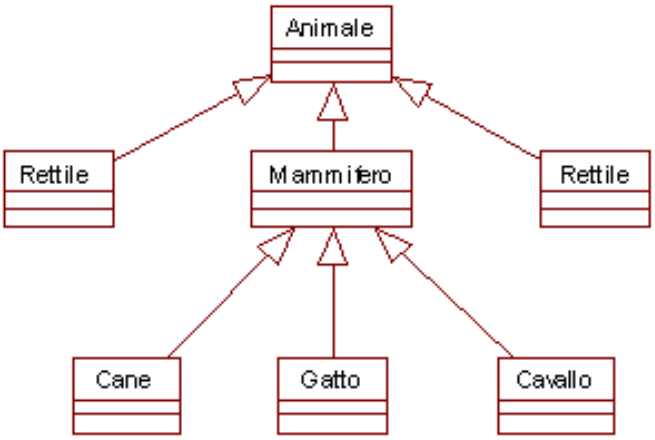
Associazione Riflessiva (o ricorsiva)
Senza le relazioni, un modello di classi sarebbe soltanto una lista di rettangoli che rappresentano un "vocabolario" del sistema. Le relazioni mostrano come i termini del vocabolario si connettono tra di loro per fornire un quadro della fetta di sistema che si sta modellando. L'associazione rappresenta, dunque, la connessione concettuale fondamentale tra le classi in cui ogni classe, come detto, gioca un ruolo specifico.

E4 EREDITARIETA' E GENERALIZZAZIONE

Se si conosce qualcosa riguardo ad una categoria di cose, automaticamente si conosce qualcosa che è possibile trasferire ad altre categorie che, in qualche modo, discendono dalla categoria che conosciamo. Chiariamo meglio questo concetto.

Ad esempio, se si conosce qualcosa su un animale generico (ad esempio si sa che un animale mangia, dorme, é nato, si sposta da un luogo ad un altro, ecc.), potremo dire che tutte le sottocategorie di animali (rettili, anfibi, mammiferi, ecc.) erediteranno le stesse caratteristiche. Tale meccanismo, in Analisi Object Oriented, viene definito come:

Ereditarietà



Un esempio di ereditarietà
Una classe figlia (o sottoclasse) può ereditare gli attributi e le operazioni da un'altra classe (che viene definita classe padre o super classe) che sarà sempre più generica della classe figlia.

Nella generalizzazione, una classe figlia può rappresentare un valido sostituto della classe padre. Cioè, in qualunque posto appaia la classe padre, sarebbe possibile far apparire la classe figlia. Il viceversa non è, invece, vero.

In UML l'ereditarietà viene rappresentata con una linea che connette la classe padre alla classe discendente e dalla parte della classe padre si inserisce un triangolo (una freccia).

Se ragioniamo dal punto di vista dell'associazione, l'ereditarietà può essere vista come un tipo di associazione.
Come devono fare gli analisti per scoprire l'ereditarietà di una o più classi? L'analista deve far si che gli attributi e le operazioni per una classe siano generali e si applichino a parecchie altre classi (le sottoclassi) che, a loro volta, potranno "specializzare" la classe padre aggiungendo attributi e operazioni propri. Un'altra possibilità è che l'analista noti che due o più classi hanno un numero di attributi ed operazioni in comune.

Le classi che non permettono di istanziare nessun tipo di oggetto sono dette classi astratte. In UML una classe astratta si indica scrivendo il suo nome in corsivo.

Riassumendo si può dire che una classe può ereditare attributi ed operazioni da un'altra classe. La classe che eredita è figlia della classe (padre) da cui prende gli attributi e le operazioni. Le classi astratte sono utilizzate soltanto come classi base per l'ereditarietà e non forniscono alcun oggetto implementabile.

E5 AGGREGAZIONE

Qualche volta una classe può rappresentare il risultato di un insieme di altre classi che la compongono.

Questo è un tipo speciale di relazione denominata **aggregazione**.

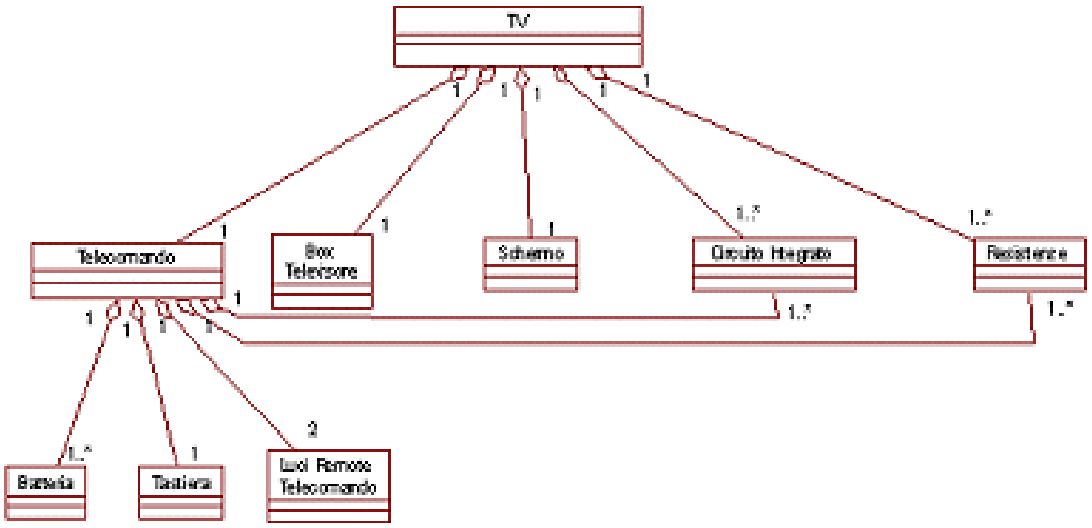
Le classi che costituiscono i componenti e la classe finale sono in una relazione particolare del tipo: **parte – intero (part-whole)**

Un'aggregazione è rappresentata come una gerarchia in cui l' "intero" si trova in cima e i componenti ("parte") al di sotto. Una linea unisce "l'intero" ad un componente con un rombo raffigurato sulla linea stessa vicino all' "intero".

Un esempio di aggregazione

Esaminiamo le "parti" che costituiscono un Televisore. Ogni TV ha un involucro esterno (Box), uno schermo, degli altoparlanti, delle resistenze, dei transistors, un circuito integrato e un telecomando (oltre, naturalmente, ad altri tantissimi componenti!). Il telecomando può, a sua volta, contenere le seguenti parti: resistenze, transitors, batterie, tastiera e luci remote.

Il Class Diagram che ne deriva sarà il seguente:

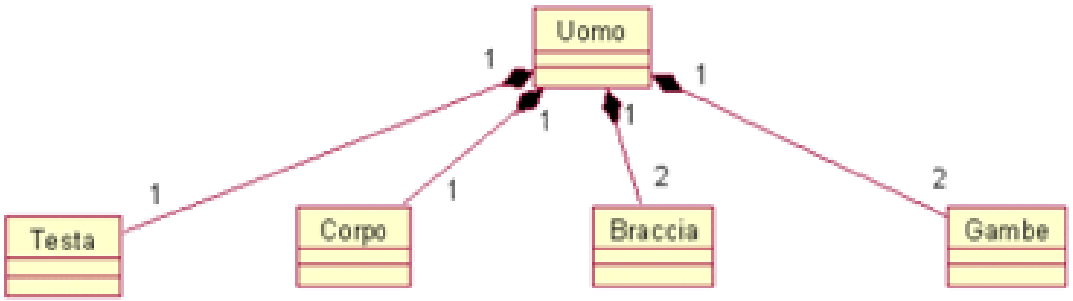


Qualche volta l'insieme di possibili componenti di un'aggregazione si identifica in una relazione OR. Per modellare tale situazione, è necessario utilizzare una constraint – la parola OR all'interno di parentesi graffe su una linea tratteggiata che connette le due linee parte-intero.

Una **composizione** è una tipo più forte di aggregazione.

Ogni componente in una composizione può appartenere soltanto ad un "intero". Il simbolo utilizzato per una composizione è lo stesso utilizzato per un'aggregazione eccetto il fatto che il rombo è colorato di nero.

Se, ad esempio, si esamina l'aspetto esterno degli uomini, si evincerà che ogni persona ha (tra le altre cose): una testa, un corpo, due braccia. e due gambe. Tale concetto viene rappresentato nel seguente grafico:



Una associazione composita. In tale associazione ogni componente appartiene esattamente ad un intero

Riepilogando diciamo che un'aggregazione specifica un'associazione di tipo "parte – intero" in cui una classe che rappresenta l' "intero" è costituita da più classi che la compongono.
Una composizione è una forma più forte di aggregazione in cui un componente può appartenere soltanto ad un "intero". Le aggregazioni e le composizioni sono rappresentate come linee che uniscono l' "intero" e il componente singolo tramite, rispettivamente, un rombo aperto e un rombo riempito di nero, entrambi disegnati dalla parte dell' "intero".

E 6 INTERFACCE E REALIZZAZIONI

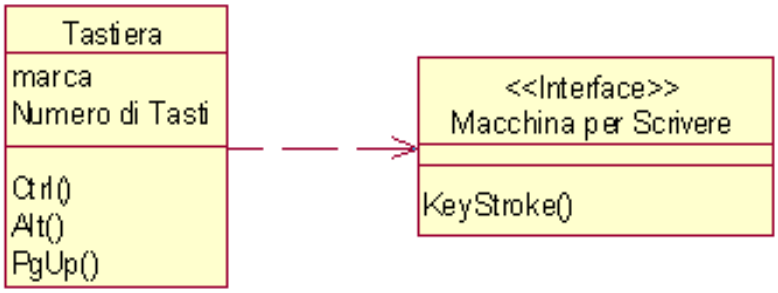
Finora abbiamo detto che è importante definire le classi in base all'intervista fatta con il cliente, e far sì che tali classi vengano messe nelle corrette relazioni tra di loro. Tuttavia, è possibile che alcune classi non siano correlate ad una particolare classe padre e che il loro comportamento possa includere alcune operazioni tutte recanti lo stesso nome. In tal caso esiste la possibilità di definire tali operazioni per una classe e riutilizzarle per le altre: è proprio quello che avviene tramite l'utilizzo delle interfacce.

Un'interfaccia è un insieme di operazioni che specifica alcuni aspetti del comportamento di una classe; in altre parole, si può dire che un'interfaccia rappresenta un insieme di operazioni che una classe offre ad altre classi.

Definizione: Un'interfaccia viene modellata allo stesso modo in cui viene modellato il comportamento una classe e rappresenta un insieme di operazioni che una classe offre ad altre classi.

Per modellare un'interfaccia si utilizza lo stesso modo utilizzato per modellare una classe, con un rettangolo. La differenza consiste nel fatto che un'interfaccia non ha attributi ma soltanto operazioni (metodi). Un altro modo utilizzato in UML per rappresentare le interfacce è quello che utilizza un piccolo cerchio che si unisce tramite una linea alle classi che implementano l'interfaccia stessa.

La tastiera del computer è un tipico esempio di interfaccia riutilizzabile. La pressione di un tasto (KeyStroke) rappresenta un'operazione che è stata riutilizzata dalla macchina per scrivere. La collocazione dei tasti è la stessa della macchina per scrivere, ma il punto cruciale è che la pressione di un tasto è stata trasferita da un sistema (i due sistemi, come si evince facilmente, nel nostro caso sono la macchina per scrivere ed il computer) ad un altro. D'altro canto, sulla tastiera dei computer si trovano un insieme di operazioni che non si trovano sulla macchina per scrivere (Ctrl, Alt, PageUp, PageDown, ecc.)



Una interfaccia è una collezione di operazioni che una classe esegue

Per distinguere le interface dalle classi, dal punto di vista UML si utilizza la scrittura **interface** all'inizio del nome di ogni interfaccia, come mostrato nella figura precendente.

La relazione tra una classe ed un'interfaccia viene definita realizzazione. Tale relazione è visualizzata nel modello da una linea tratteggiata con un triangolo largo aperto costruito sul lato dell'interfaccia.

E 7 VISIBILITA'

La Visibilità si applica ad attributi o operazioni e specifica la possibilità che hanno le classi di usare gli attributi e le operazioni di un'altra classe.

Sono consentiti tre livelli di visibilità:

- Livello pubblico: L'utilizzo viene esteso a tutte le classi
- Livello protetto: L'utilizzo è consentito soltanto alle classi che derivano dalla classe originale
- Livello privato: Soltanto la classe originale può utilizzare gli attributi e le operazioni definite come tali.

A livello grafico vengono utilizzati, generalmente, i seguenti simboli per distinguere i tre livelli:

- Livello pubblico: +
- Livello protetto: #
- Livello privato: -

Il Rational Rose, invece, utilizza i seguenti simboli, per le operazioni:

- Livello pubblico:
- Livello protetto: 
- Livello privato: 

E gli stessi, con il colore differente, per gli attributi:

- Livello pubblico: 
- Livello protetto: 
- Livello privato: 

E 8 CLASS DIAGRAM: un esempio pratico

Prendiamo come esempio una applicazione classica: quella che permette di scrivere un documento di testo. Cerchiamo di descrivere i passi che costituiscono tale applicazione.

Supponiamo che si stia digitando un documento di testo, utilizzando qualche famoso editor di testi, come Microsoft Word, ad esempio. L'utente ha due possibilità iniziali:

- Cominciare a scrivere un nuovo documento
- Aprire un documento esistente

Il testo da scrivere, ovviamente, verrà digitato tramite l'utilizzo della tastiera.

Ogni documento è composto di svariate pagine, e ogni pagina, a sua volta, è composta da una testata, dal corpo del documento e da un piè di pagina. Nella intestazione e nel piè di pagina è possibile aggiungere la data, l'ora, il numero di pagina, la collocazione del file, ecc.

Il corpo del documento è formato da frasi. Le frasi, a loro volta, sono composte da parole e segni di punteggiatura. Le parole sono formate da lettere, numeri e/o caratteri speciali. Inoltre,. Vi è la possibilità di aggiungere delle immagini e delle tabelle nel documento. Le tabelle sono costituite da righe e colonne ed ogni cella di una tabella può contenere del testo o delle immagini.

Dopo aver terminato il documento, l'utente può scegliere di salvarlo o stamparlo.

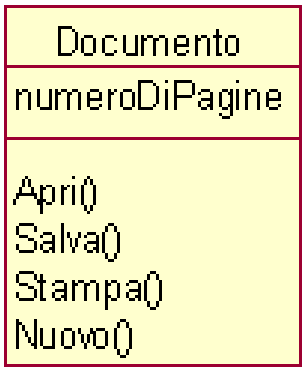
Abbiamo in tal modo descritto, in maniera semplificata ma congruente, il processo di creazione di un documento tramite Microsoft Word. Rifacendoci a quanto già detto nella guida circa l'intervista con il cliente, siamo in grado, allora, di estrarre la seguente lista di parole chiave:

documento , editor di documenti, Microsoft Word, **testo** , tastiera, **intestazione, piè pagina**, corpo del documento, **data, ora, numero di pagina, collocazione del file, pagina, frase, parola, segno di punteggiatura, lettera, numero, carattere speciale, immagine, tabella, riga, colonna, cella, utente**.

Potremmo dire che le parole chiave evidenziate in blu sono sicuramente candidate a diventare classi o attributi per il modello da sviluppare.

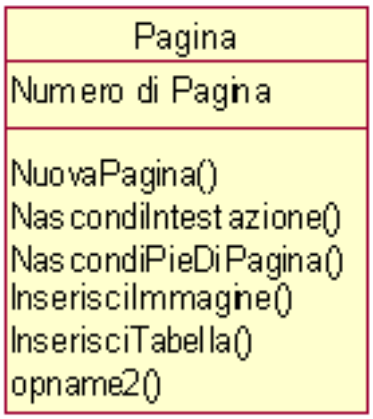
Come è possibile osservare, nell'esempio descritto l'oggetto attorno a cui ruoto un po' tutto il discorso è il **Documento**. Per tale ragione, sarà una buona idea identificare nel Documento la classe centrale del nostro Class Diagram.

Un documento, come detto, può avere svariate pagine e, quindi, potremo definire un attributo: numeroDiPagine che descrive tale caratteristica. Invece, per le operazioni, sarà bene definire i metodi:
Apri( ), Salva( ), Stampa( ),Nuovo( ).



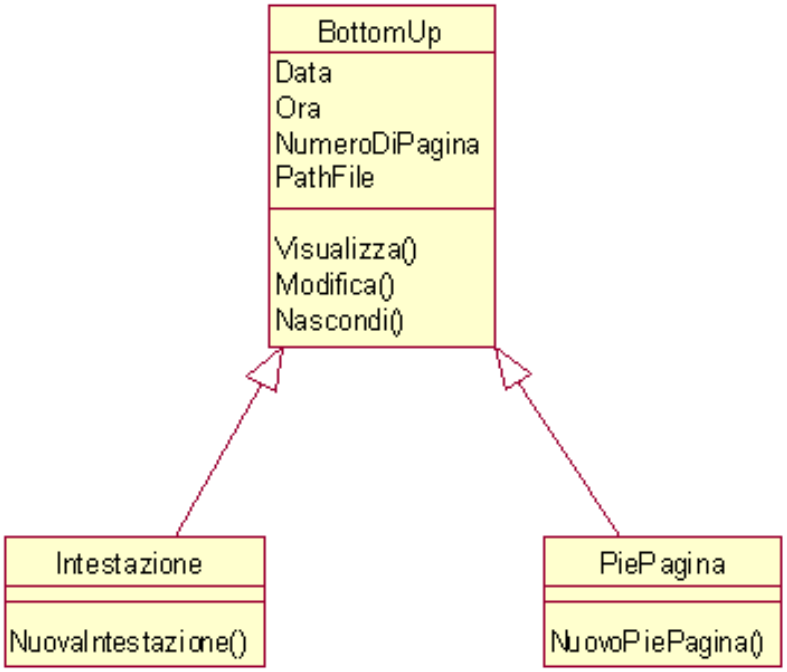
Si è detto, poi, che ogni documento è composto di pagine. La Pagina sarà, dunque, un buon candidato per essere una classe del nostro diagramma.

La Classe Pagina conterrà un attributo : numeroDiPagina, che identificherà il numero della pagina dell'oggetto, e le seguenti operazioni: nuovaPagina(), nascondiIntestazione() e nascondiPièdiPagina().



I metodi per la testata e il piè di pagina ci fanno intendere che è bene definire altre due classi: **Intestazione e PiePagina**.

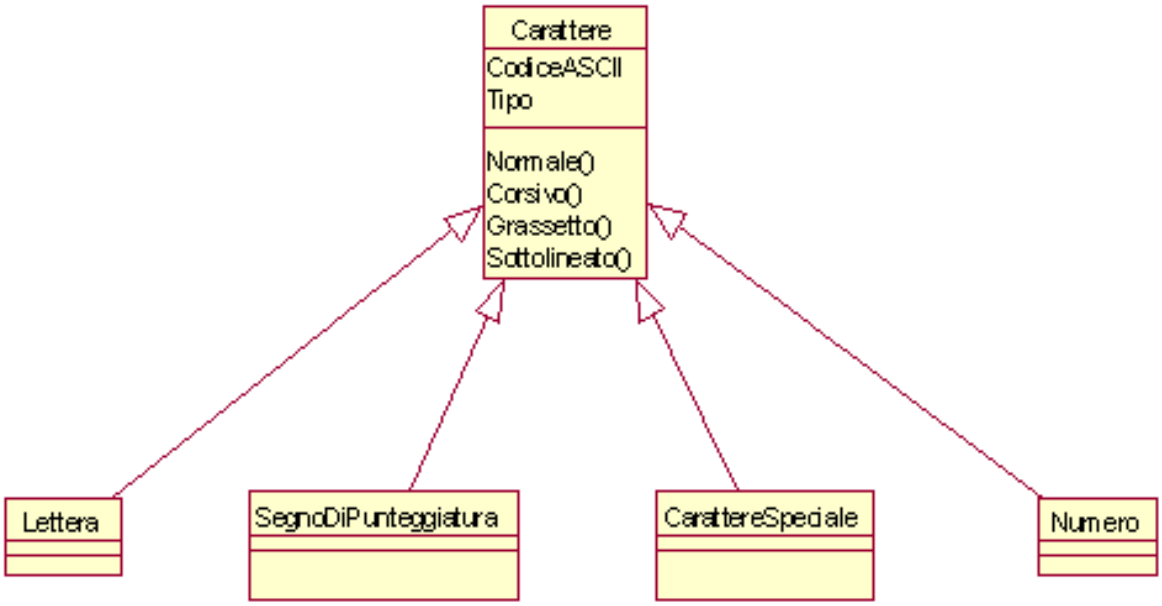
Tali classi hanno i seguenti attributi in comune: data, ora, numeroDiPagina e pathFile. Tali attributi sono opzionali per ogni intestazione o pie' di pagina e l'utente potrà configurarli a suo piacimento.
La somiglianza tra Intestazione e PièPagina ci spinge a definire ancora una nuova classe che sia in comune alle due appena descritte. è qui che utilizziamo il concetto di ereditarietà che abbiamo definito in precedenza.
La classe padre sarà **BottomUp** (questo nome è scelto perché le intestazioni e i piè di pagina appaiono, rispettivamente, nella parte alta e bassa di ogni pagina del documento) e conterrà gli attributi che sono in comune alle classi Intestazione e PiePagina oltre alle operazioni (anch'esse in comune) : visualizza(), modifica() e nascondi().
Le classi Intestazione e PiePagina (figlie di BottomUp) avranno la necessità, quindi, di definire, rispettivamente, soltanto le operazioni: nuovaIntestazione() e nuovoPiePagina().



Prima di passare ad esaminare il corpo del documento o il testo, diamo uno sguardo alla definizione dei componenti di un testo. Come detto, il testo del documento è composto di frasi. Le frasi, a loro volta, sono composte di parole e le parole sono formate da caratteri. Se le parole sono, dunque, array di caratteri e le frasi sono identificabili come array di parole, allora una frase è anche (per una sorta di proprietà transitiva) un array di caratteri.

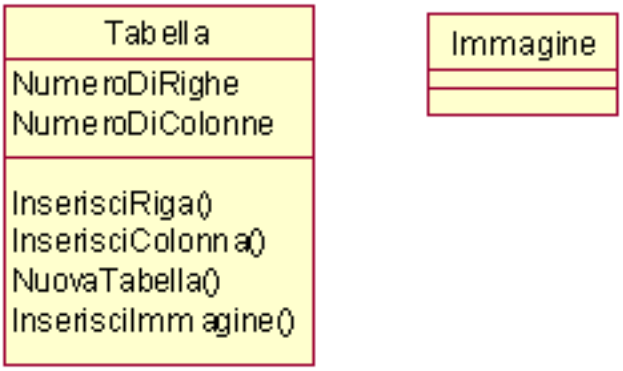
Quindi, il corpo di un documento può essere descritto come un array di caratteri. Per tale ragione, per descrivere il testo di un documento faremo uso della classe **Carattere** con alcune sottoclassi.
La classe Carattere avrà gli attributi: codiceASCII e tipo (il tipo ci informa sulla tipologia di carattere, ovvero se esso è in formato normale, corsivo, grassetto o sottolineato) e definirà le operazioni: Normale(), Corsivo(), Grassetto() e Sottolineato().

Come discendenti della classe Carattere definiremo poi: **Lettera, SegnoDiPunteggiatura, CarattereSpeciale e Numero**. Inoltre, nel corpo del documento, come detto, possono comparire tabelle o immagini. Entrambe costituiranno altre due nuove classi nel nostro diagramma.

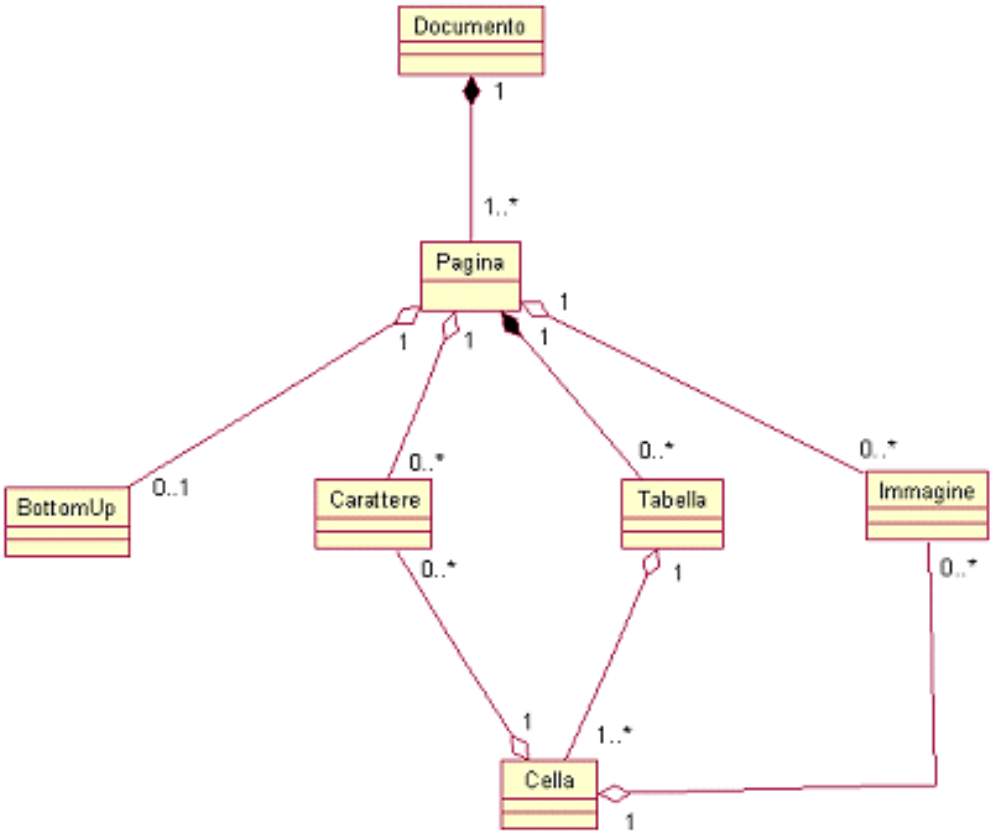


La Classe **Tabella** contiene gli attributi numeroDiRighe e numeroDiColonne e definisce le operazioni inserisciRiga( ) e inserisciColonna( ).

Ogni tabella consiste di una o più celle e, come già detto, in ogni cella è possibile che compaiano testo o immagini.



Dovremmo essere ad un buon punto per iniziare a mettere insieme tutti i tasselli finora definiti. Utilizzando le associazioni tra le classi che abbiamo descritto otterremo dunque il seguente Class Diagram che descrive il nostro esempio:



è importante rendersi conto che il diagramma precedente non costituisce la massima descrizione in dettaglio per un processo di scrittura di un documento di testo.

Il modello descritto potrà essere specializzato sempre di più e, quindi, crescere in modo appropriato per poter consentire un appropriato lavoro di analisi e disegno.

VISUAL BASIC

Visual Basic della piattaforma .NET framework di Microsoft (VB NET) è un **linguaggio di programmazione pseudo compilato**

Riassumendo in sintesi:

- linguaggio compilato (es. C, C++, FORTRAN, PASCAL, etc.): è un linguaggio di programmazione di alto livello per il quale la trasformazione dal codice sorgente al codice macchina (programma eseguibile) avviene mediante programmi, chiamati *compilatori*, che traducono tutte le istruzioni del programma sorgente in codice binario in un'unica volta.
Quindi traduzione ed esecuzione sono fatte in tempi differenti;

- linguaggio interpretato (es BASIC, JAVASCRIPT, etc.): è un linguaggio di programmazione di alto livello per il quale la trasformazione dal codice sorgente al codice macchina eseguibile avviene mediante programmi, chiamati *interpreti*, che traducono ed eseguono le istruzioni del programma sorgente una alla volta.
Quindi traduzione ed esecuzione procedono contemporaneamente una di seguito all'altro alternandosi.

- linguaggio pseudocompilato (es JAVA, VB NET, etc.) : è un linguaggio di programmazione di alto livello per il quale il programma sorgente viene dapprima compilato in codice intermedio simile all'assembler e successivamente viene eseguito da un interprete. Siamo quindi in una situazione intermedia tra la compilazione e l'interpretazione o meglio metà lavoro viene svolto dal compilatore e metà dall'interprete.

Visual Basic è anche un linguaggio di **programmazione ad eventi o *even-driven***

La **programmazione a eventi** è un paradigma di programmazione dell'informatica. Mentre in un programma tradizionale l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore, nei programmi scritti utilizzando la tecnica *a eventi* il flusso del programma è largamente determinato dal verificarsi di eventi esterni (quali ad esempio la pressione di un tasto della tastiera oppure un click del mouse oppure ancora l'apertura di una finestra etc.).

I programmi che utilizzano la programmazione a eventi sono composti tipicamente da diversi brevi sotto-programmi (routine) , chiamati **gestori degli eventi** (*event handlers*), che sono eseguiti in risposta agli eventi esterni, e da un **dispatcher**, che effettua materialmente la chiamata, spesso utilizzando una *coda degli eventi* che contiene l'elenco degli eventi già verificatisi, ma non ancora "processati".

In molti casi i gestori degli eventi possono, al loro interno, innescare (*"trigger"*) altri eventi, producendo una cascata di eventi.

VISUAL BASIC e la programmazione ad oggetti

Un programma Visual Basic è formato da un insieme di classi.

In questo linguaggio di programmazione i due tipi di classi più comuni sono i **Form** e le **Classi utente.**
La principale differenza tra le due tipologie è che rispetto ad una Classe utente i Form contengono già incorporata un'interfaccia grafica utente

L'**interfaccia grafica utente (GUI o** *Graphic User Interface***)** consente all'utente di interagire con il computer manipolando graficamente degli oggetti in maniera molto accattivante ed user-friendly, al contrario di quanto avviene sulla cosiddetta *riga di comando* di una **CLI** (*Command Line Interface*), in cui l'esecuzione del programma viene guidata da istruzioni o comandi impartiti dall'utente tramite tastiera.

L'elemento visuale principale di un'applicazione Visual Basic di tipo Windows Form (applicazione con interfaccia Windows) è la **finestra** (o **Form**) sulla quale possiamo disporre i **controlli grafici** ai quali è associato il codice scritto in Visual Basic.
Il codice Visual Basic associato ad un controllo viene eseguito quando si verifica un **evento**.

I principali elementi sui quali poggia la programmazione visuale sono quelli illustrati di seguito:

Oggetti: rappresentano gli oggetti della programmazione visuale come ad esempio le finestre (o Form) ed i controlli grafici (caselle di testo, pulsanti, caselle combinate, etc.).
Ciascun oggetto è identificato da un nome (proprietà *name*) ed è l'istanza di una classe.

Proprietà: ciascun oggetto possiede alcune proprietà che ne specificano le caratteristiche (nome, dimensioni, colore, posizione nel Form, etc.) che possono essere impostate o direttamente nella sezione apposita prevista oppure utilizzando nel codice la dot notation.

Eventi: ciascun oggetto riconosce alcune azioni specifiche prodotte o dall'utente o dal sistema perativo, chiamate eventi, in risposta alle quali il programmatore può predisporre la relativa routine (sottoprogramma) di evento che verrà eseguita al verificarsi di quello specifico evento.

Metodi: rappresentano le azioni eseguibili sugli oggetti. Si tratta di Function o Sub associate ai controlli e possono essere eseguite secondo la dot notation.

Speciale laboratorio

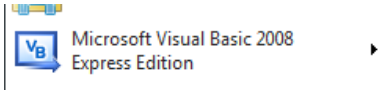
VISUAL BASIC 2008 Express Edition : c

a) Creare la prima applicazione Windows Form

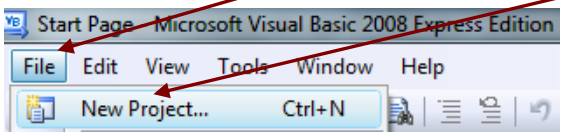
Ora impareremo come realizzare un programma che consente di utilizzare i controlli tipici di Windows, quali, per esempio, i pulsanti, le caselle di tetso, etc.

Per creare un nuovo progetto di modello Applicazione Windows Form dobbiamo:

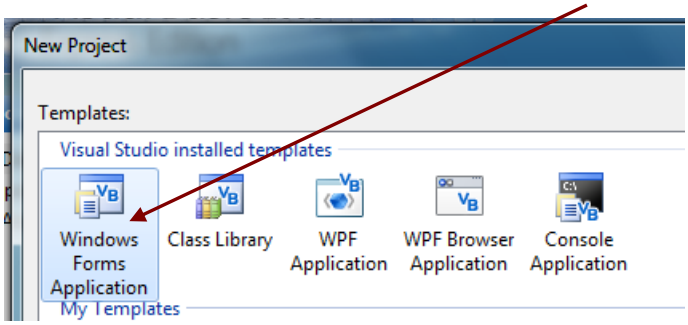
1) Prima di tutto aprire l'ambiente di sviluppo Visual Basic 2008 Express Edition facendo doppio click sull'icona relativa



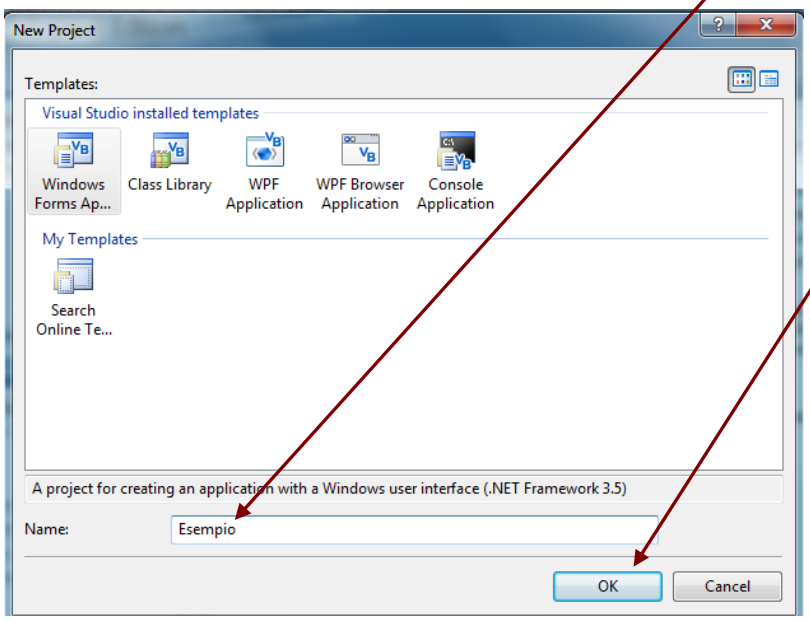
poi dopo avere fatto click su **File** ed avere selezionato **Nuovo Progetto…**



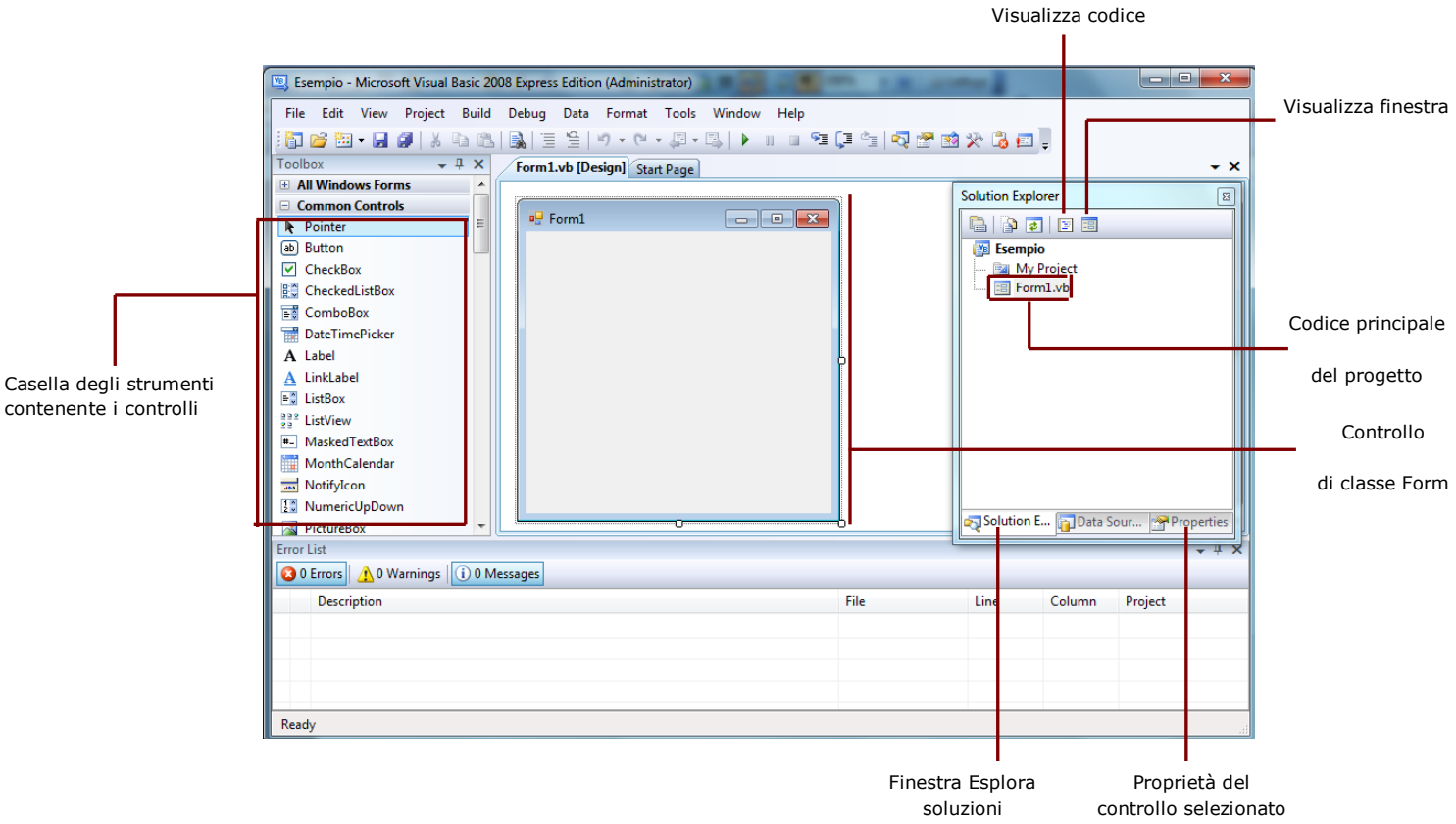
scegliere la prima icona, denominata **Applicazione Windows Form**



2) Dopo avere digitato il nome del progetto (in questo caso *Esempio*) e cliccato su OK

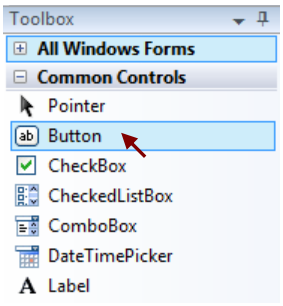


si ottiene la schermata iniziale dell'IDE suddivisa in diverse sezioni, come mostrato dalla figura seguente



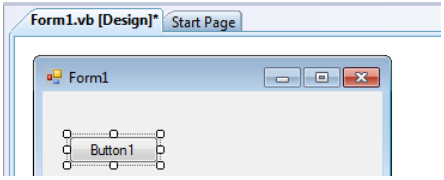
La finestra *Esplora soluzioni* elenca tutti i Form, i moduli e le classi del progetto selezionabili con il mouse.
Attraverso le icone *Visualizza finestra progettazione* e *Visualizza Codice* della finestra *Esplora soluzioni* si passa dalla visualizzazione del "disegnatore" del Form al codice Visual Basic associato e viceversa.

3) Passiamo all'inserimento di un controllo all'interno del Form1. Proviamo ad immettere un pulsante (controllo *Button*).
Per far ciò occorre selezionare il controllo desiderato con il mouse, come mostrato a lato, all'interno della casella degli strumenti



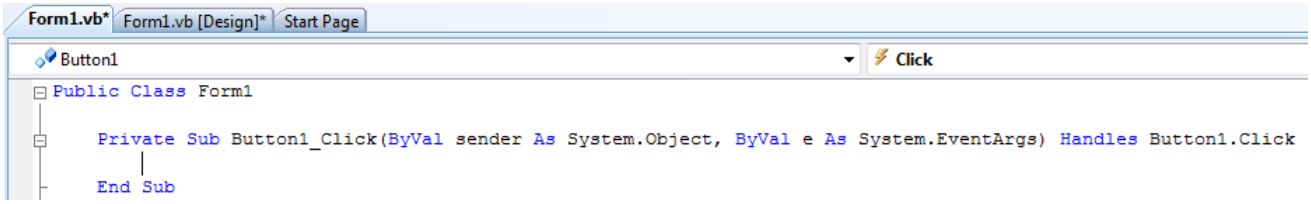
4) Per disegnare un nuovo pulsante nel Form, dopo averlo selezionato nella casella degli strumenti, occorre trascinare il mouse fino a raggiungere la dimensione desiderata

5) A questo punto una volta lasciato il mouse, appare il disegno che raffigura il controllo, in questo caso un pulsante di classe *Button* di nome *Button1*

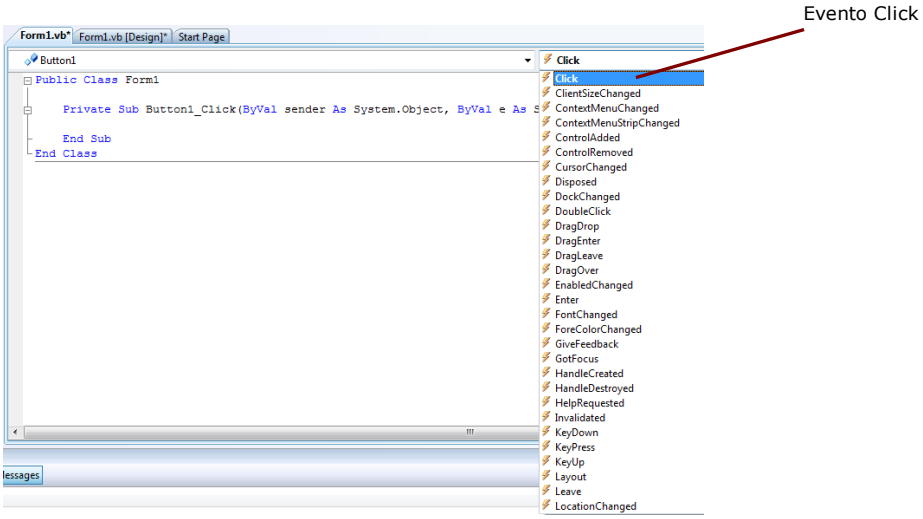


6) E' possibile associare del codice Visual Basic al controllo facendo doppio click sul pulsante per creare una routine di risposta al click

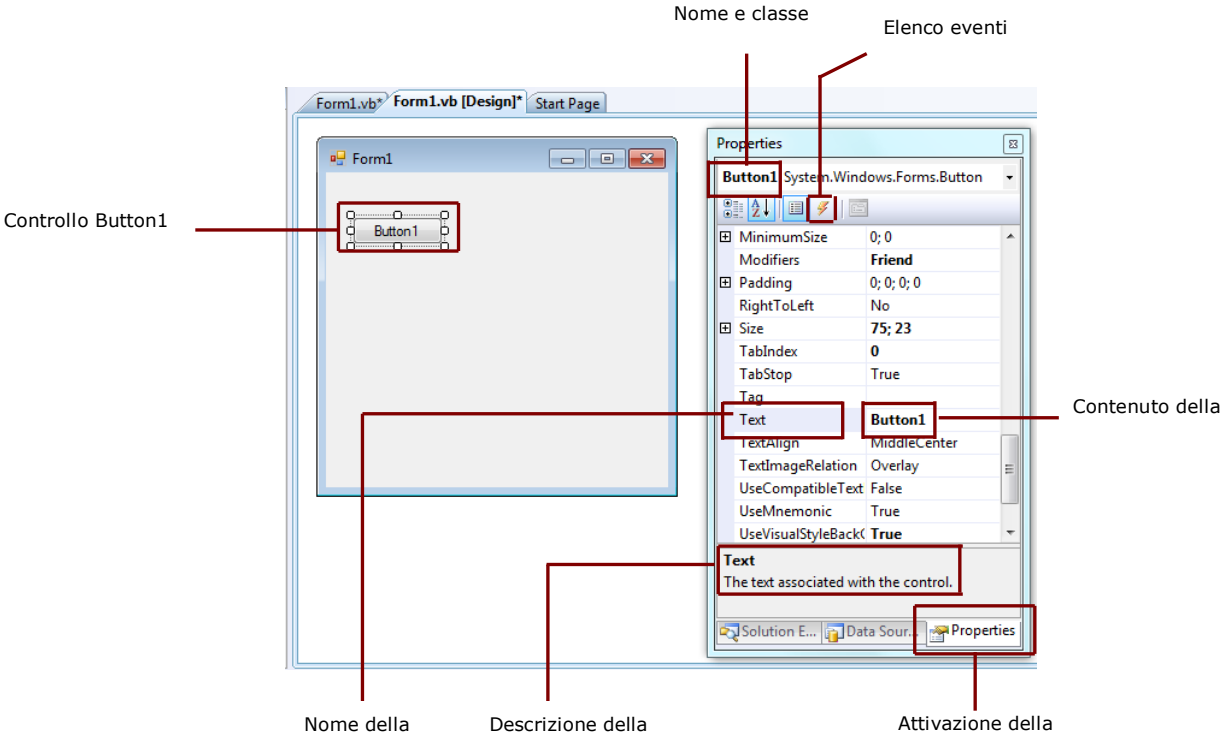
7) Si apre automaticamente una nuova Sub con lo stesso nome del controllo e dell'evento separato dal carattere underscore (*Button1_Click*)



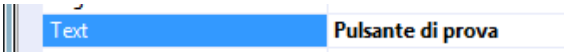
8) Nella finestra del codice (Form1.vb) sono presenti due caselle a tendina.
La prima a sinistra, contiene l'elenco degli oggetti presenti nel progetto;
la seconda sulla destra contiene l'elenco degli eventi attivabile da quel particolare oggetto selezionato nella casella associato.
In questo caso si può notare che l'evento di default è *Click*.



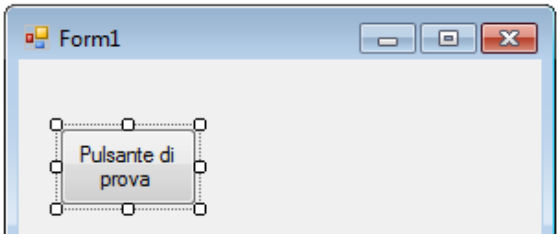
9) Un oggetto possiede **proprietà** che possono essere modificate attraverso la finestra *Proprietà* come mostrato nella figura seguente



10) Si può notare che tra le proprietà vi è il testo mostrato sopra il pulsante (proprietà *Text*): per modificarlo è sufficiente fare click all'interno della casella di testo posta accanto (contenuto della proprietà *Text*), assegnandogli il testo seguente



11) Il pulsante ora ha cambiato aspetto



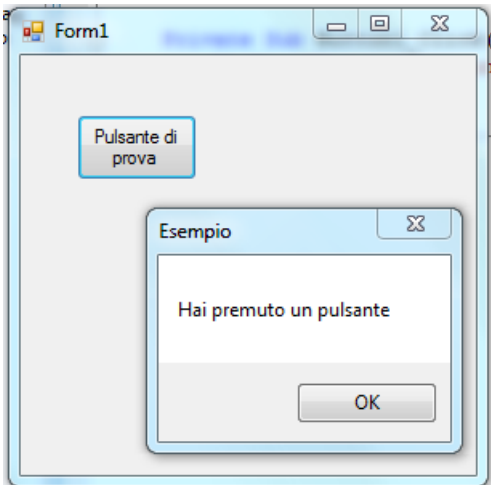
N.B. Occorre non confondere la proprietà *Text* con la proprietà *Name*: sul pulsante appare il testo "Pulsante di prova", tuttavia il nome dell'oggetto è rimasto *Button1* ed è proprio questo il nome che occorre utilizzare per identificarlo nel codice.

12) E' possibile ora passare alla scrittura del codice da eseguire al click sul pulsante.
Basta fare doppio click sul pulsante oppure selezionare l'evento *Click* dal menù a tendina posto in alto.
si ottiene La seguente procedura all'interno della quale è possibile scrivere il seguente codice da eseguire (*MsgBox*)

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        MsgBox("Hai premuto un pulsante")
    End Sub
End Class
```

13) Una volta eseguito il programma si ottiene la visualizzazione del Form1: il click sul pulsante Button1 determina l'attivazione della finestra che segue:



b) L'output tramite tastiera: sintassi completa dell'istruzione MsgBox