



Gregorys.it  
**I corsi on-line**



# **Corso di Object Oriented Programming**

# Indice

<b>II Software</b>	<b>3</b>
Definizione di Software	3
Ingegneria del Software	3
Qualità del prodotto	4
<b>Cicli di vita</b>	<b>5</b>
Modello in Cascata o Strutturale	5
Exploring Programming	7
Prototyping	8
Differenze tra i vari modelli	8
Modello a Spirale	9
<b>Modelli Entità-Relazione</b>	<b>10</b>
<b>Progettazione del software</b>	<b>12</b>
Processo della Progettazione	12
<b>Strategie di Progetto</b>	<b>13</b>
<b>Qualità della Progettazione</b>	<b>14</b>
<b>FOD - Function Oriented Design</b>	<b>15</b>
<b>OOD - Object Oriented Design</b>	<b>16</b>
Object Oriented Design – Caratteristiche e Vantaggi	16
OOD e OOP – Design e Programming	17
<b>OOP - Object Oriented Programming</b>	<b>18</b>
Il Riutilizzo Object-Oriented	18

# Il Software

## Definizione di Software

Il software può essere definito nei seguenti modi:

- 1) L'insieme delle istruzioni che eseguite forniscono le funzioni e le prestazioni desiderate;
- 2) L'insieme delle strutture dati che consentono ai programmi di manipolare adeguatamente le informazioni;
- 3) L'insieme della documentazione che descrive l'operabilità e l'uso dei programmi.

La prima definizione è abbastanza intuitiva, fa infatti riferimento ad un programma per un elaboratore, a differenza della seconda che lo vede dal punto di vista dell'insieme delle strutture dati. La terza invece basandosi sulla documentazione, ne descrive l'operabilità e le istruzioni d'uso per quelle che saranno le strutture dati.

Come si può notare non esiste una vera e propria definizione di software, tutte vanno bene, bisogna solamente sapere rispetto a cosa viene data.

## Ingegneria del Software

Dopo aver visto come possiamo identificare un software dobbiamo trovare gli strumenti per poterlo realizzare in maniera professionale. Possiamo quindi introdurre la definizione di Ingegneria del Software, ovvero *l'insieme dei principi e delle metodologie ingegneristiche che offrono i mezzi per realizzare un sistema software conveniente, affidabile ed efficiente, in poche parole di qualità.*

Dalla definizione, un sistema software deve obbligatoriamente soddisfare una serie di caratteristiche che lo rendano un **Software di Qualità**: *Caratteristiche Esplicite ed Implicite.*

**Caratteristiche Esplicite:** sono tutte le specifiche accordate con il cliente e si dividono in **Requisiti Funzionali** e **Requisiti Prestazionali**. I primi rappresentano i requisiti che il sistema deve avere sulla base delle richieste del cliente; i secondi invece rappresentano i requisiti degli standard di sviluppo.

**Caratteristiche Implicite:** sono tutte le specifiche sottintese/scontate che un software professionale deve possedere.

*Esempio:*

*Se si acquista uno stereo ci si aspetta che nel momento in cui abbia un guasto lo si possa riparare; oppure se si acquista un computer ci si aspetta che in futuro lo si possa potenziare con degli upgrade.*

Quindi si può notare come un sistema non deve solamente soddisfare le richieste del cliente, ma anche tutte quelle caratteristiche racchiuse all'interno della sua realizzazione. In conclusione per quanto buono possa essere il risultato di realizzazione delle caratteristiche esplicite, se non si rispettano anche quelle implicite, nel caso in cui un sistema vada in tilt l'unica cosa da fare è buttarlo.

## Qualità del prodotto

Come detto in precedenza, affinché un prodotto software si possa definire di qualità, deve soddisfare delle determinate caratteristiche. Entriamo più in dettaglio ed elenchiamole una x una:

- **Correttezza:** quanto più un sistema raggiunge gli obiettivi prefissati dal cliente, tanto più è corretto. Tuttavia raggiungere una correttezza al 100% è quasi impossibile.
- **Affidabilità:** indica la misura di quanto ci si può aspettare che un programma svolga le proprie funzioni con la precisione desiderata.
- **Robustezza:** un programma è robusto se si comporta "ragionevolmente" anche in circostanze impreviste.
- **Efficienza:** determina il grado di complessità del programma legato alle risorse di controllo e al codice.
- **Amichevolezza:** rappresenta il grado di facilità d'uso col quale il programma si presenta all'utente.
- **Verificabilità:** un software è verificabile se risulta documentato procedura per procedura.
- **Manutenibilità:** indica lo sforzo, in termini umani e di tempo, richiesto per modificare il prodotto software dopo la prima release.
  - o *Manutenzione Correttiva:* manutenzione dovuta ad un errore non previsto dal sistema. Questo si verifica quando non è stato testato bene.
  - o *Manutenzione Adattiva:* manutenzione dovuta a possibili modifiche. Melo legato è all'ambiente, al dato e all'hardware, tanto più è flessibile.
  - o *Manutenzione Perfettiva:* manutenzione che tenta di perfezionare un sistema già esistente.
- **Riparabilità:** un sistema è riparabile se consente la correzione dei suoi difetti con una piccola quantità di lavoro.
- **Riusabilità:** è il grado di riutilizzo del prodotto o di una sua parte in altre applicazioni.
- **Portabilità:** indica lo sforzo richiesto per trasferire un prodotto da un ambiente software/hardware ad un altro.
- **Interoperabilità:** determina lo sforzo richiesto per collegare e far cooperare un prodotto software con un altro.

## Qualità che agiscono sul successo del prodotto

- **Puntualità:** un sistema è puntuale se produce il risultato nei tempi prestabiliti. La puntualità può influenzare la correttezza.
- **Trasparenza:** indica quanto un processo è visibile attraverso i suoi passi. Questa qualità è legata alla verificabilità del prodotto finale.
- **Produttività:** è la qualità che porta ad una maggiore efficienza del prodotto.

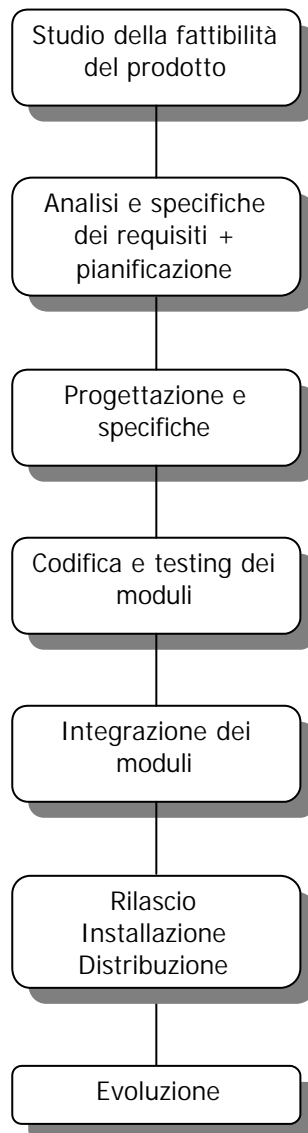
# Cicli di vita

Un prodotto software è visto come un'entità che ha un suo concepimento ed un vero e proprio ciclo di vita. Il concepimento avviene quando l'utente richiede un determinato tipo di prodotto e stabilisce un accordo con colui il costruttore, mentre la conclusione dipende dal modello che è stato adottato per la sua realizzazione.

Vediamo quali sono i modelli più utilizzati:

## Modello in Cascata o Strutturale

Questo modello è uno tra i più noti cicli di vita del software nonostante le sue pecche. E' detto in "cascata" o "strutturale" poiché i vari passi si susseguono l'un l'altro.



Quando si usa un modello in cascata la conclusione di ciascuna fase consiste nella stesura di un documento con specifici.

Lo studio della fattibilità è il primo passo per verificare se esistono i mezzi hardware, umani ed economici affinché lo si possa creare. Si studiano pertanto tutte le informazioni in modo da rendersi conto dei rischi di fondo a cui si va incontro. La stesura del documento introduce la fase per la stipula del contratto col cliente. Nelle fasi successive, come riportato dal grafico, si analizzano le specifiche, i requisiti che il prodotto deve soddisfare e il progetto di massima. E' importante dettagliare la documentazione in quanto finisce per essere l'unico mezzo di comunicazione tra le fasi individuate nel modello in cascata. La codifica, o implementazione, è la fase in cui si realizza quello che in seguito sarà il sistema software ed anticipa lo stadio di **V&V** ovvero di **Verification** e **Validation** comunemente chiamata *fase di testing*. Lo scopo della Verification è quello di controllare se ciò che si sta costruendo è effettivamente ciò che si era prefissato. Mentre quello della Validation è controllare che il sistema non riporti alcun tipo di errore.

Bisogna fare attenzione con questo modello perché in realtà non è tanto in cascata dato che al verificarsi di errori si è costretti a ritornare sui propri passi. E' buona norma effettuare una fase di testing al termine di ogni fase per testarne la correttezza. Ricordiamo che più tardi viene scoperto l'errore più è alto il costo di manutenzione.

Una volta conclusa la fase di testing si passa all'integrazione tra i moduli e alla verifica del corretto funzionamento. All'atto del rilascio il prodotto software continua la sua esistenza nell'ambito operativo, verrà infatti mantenuto ed eventualmente evoluto.

Possiamo riassumere il modello in cascata nel seguente modo:

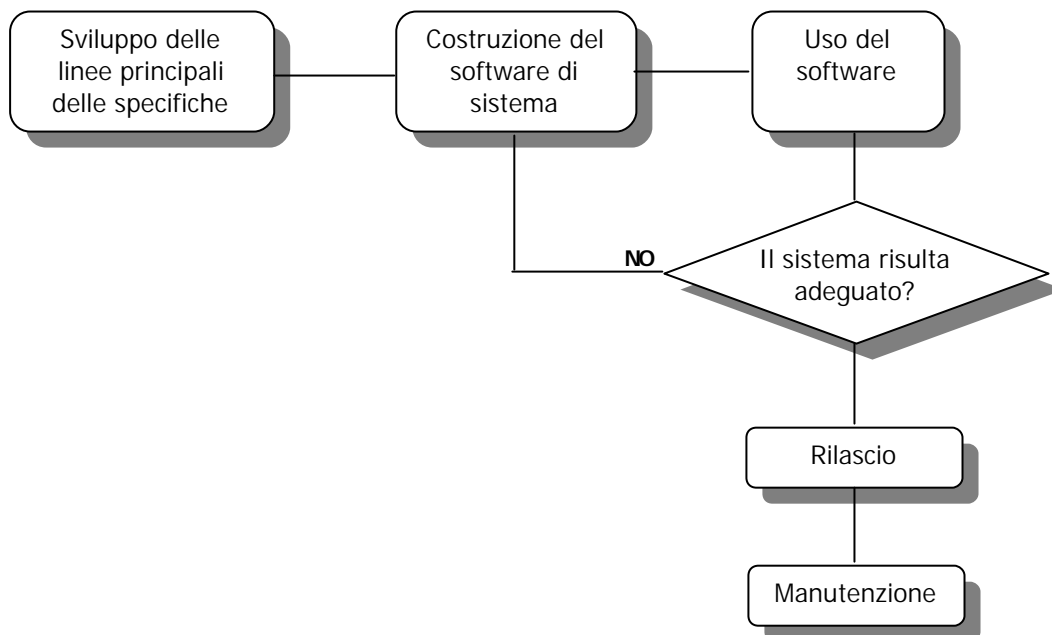
- Analisi delle richieste → I e II fase
- Progettazione del sistema → III fase
- Implementazione e Testing → IV fase
- Integrazione tra i moduli → V fase

### **Vantaggi e svantaggi**

L'avere i moduli in cascata ha il grosso vantaggio di poter suddividere i team, di avere quindi persone specializzate per ciascuna fase, mentre ha lo svantaggio di non essere effettivamente reale, in quanto non esiste progetto che segue rigidamente la cascata senza tornare indietro.

## Explorating Programming

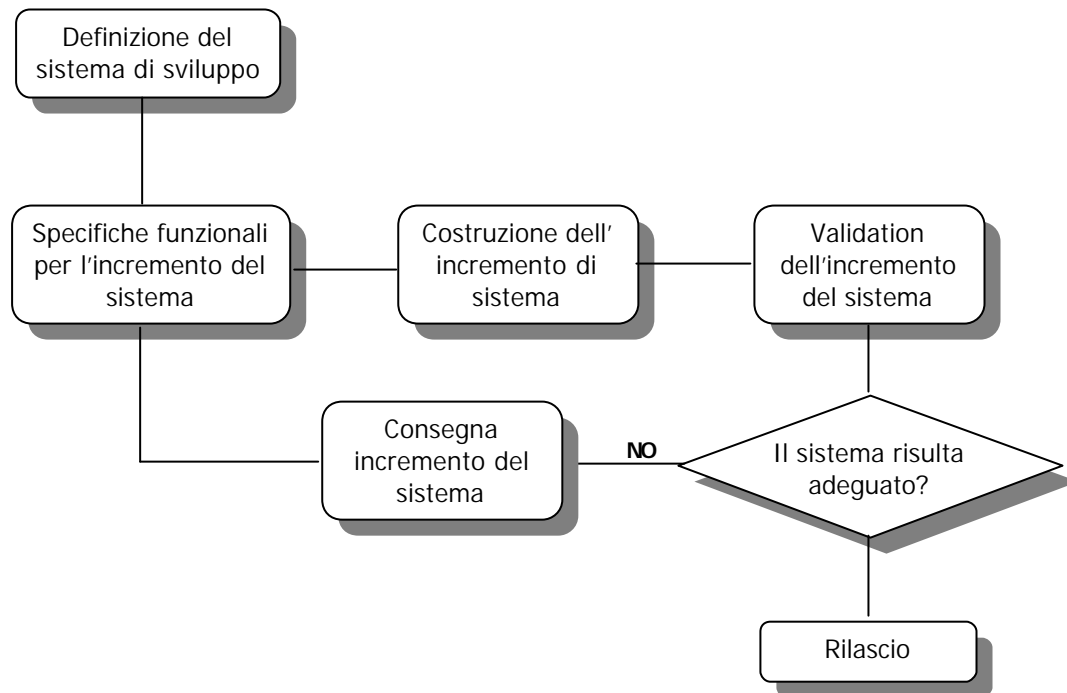
Questo tipo di approccio è usato per un tipo di software per il quale non è possibile scrivere le specifiche, ad esempio quelli implementati per l'intelligenza artificiale. In tali casi infatti si fa uno studio del sistema immaginando quale tipo di sistema si voglia realizzare. Una bozza del progetto viene presentata al cliente e sottoposta alle prime modifiche. Logicamente i costi sono molto alti in quanto questa fase può continuare per molto tempo, finché non si trova un punto d'accordo.



La prima fase del progetto consiste nel mettere insieme quel "qualcosa" che il cliente richiede, la seconda nel mettere mano direttamente sul codice e la terza nel suo utilizzo controllando eventuali errori. Quest'ultima fase può essere vista come una fase di testing che ne verifica l'adeguatezza. Non si può infatti parlare di correttezza o di verification, poiché non è presente alcuna specifica. Rilascio e manutenzione concludono il ciclo di vita.

## Prototyping

L'approccio di questo modello è simile all'exploring programming e prevede la realizzazione di un prototipo durante la fase di realizzazione dell'intero sistema. Se tale prototipo viene accettato allora si può passare alla fase successiva nella quale si aggiungono tutti i componenti necessari per avere il sistema finale. Teniamo presente infatti che nelle prime fasi non si tiene conto degli aspetti più marginali, bensì ci si preoccupa della realizzazione degli aspetti concreti.



Il vantaggio del prototyping è quello di essere in grado di fornire un sistema funzionante già durante lo sviluppo, anche se non completo.

## Differenze tra i vari modelli

La differenza tra il prototyping e l'exploring programming consiste nel fatto che nel primo si hanno delle specifiche e si costruisce un prototipo funzionante simile al sistema finale, mentre nel secondo non esiste alcuna specifica.

La differenza tra prototyping e modello in cascata consiste nel fatto che per quest'ultimo il prodotto non può essere visto durante la sua fase di realizzazione.



## Modello a Spirale

Abbiamo visto che il modello in cascata è quello più utilizzato nei progetti, ma non è il più preciso, in quanto in alcuni casi non è realizzabile. Analogamente per gli altri modelli visti: exploring programming e prototyping. Tuttavia ognuno di essi possiede delle caratteristiche che li privilegiano e che li differenziano, sarebbe interessante quindi proporre la realizzazione di un modello che prenda le parti più significative e li fonda tutti insieme. Ciò significa che nella realizzazione del ciclo di vita del software, si individuano delle fasi che sono anelli di una spirale e ogni anello rappresenta un processo del ciclo. Questo modello riassuntivo verrà chiamato **Modello a Spirale**.

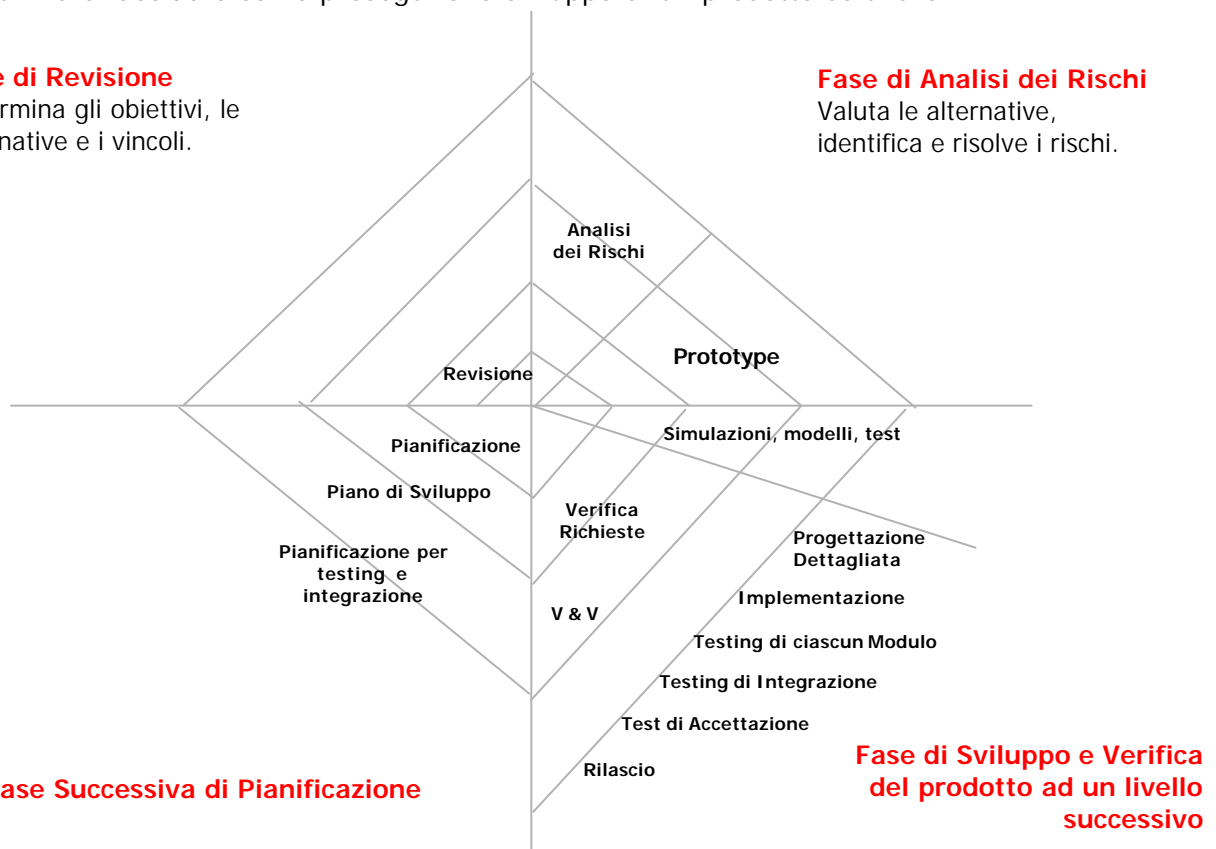
Il modello è realizzato su un sistema di assi cartesiani che suddivide una spirale in quattro settori, ognuno dei quali distingue una fase. Ogni ciclo di spirale parte da una fase di revisione e lungo tutto il suo percorso si propone di valutare, analizzare ed effettuare una analisi dei rischi prima di decidere come proseguire lo sviluppo di un prodotto software.

### Fase di Revisione

Determina gli obiettivi, le alternative e i vincoli.

### Fase di Analisi dei Rischi

Valuta le alternative, identifica e risolve i rischi.



- Il primo quadrante (in alto a sx) determina gli obiettivi, le alternative ed i vincoli (Fase di Revisione).
- Il secondo (alto a dx) è il quadrante che effettua l'analisi dei rischi, dove si analizzano tutta una serie di casi che comprometterebbero il prodotto. Anche se l'analisi comporta un rallentamento dello sviluppo finale del progetto, non lo si può trascurare in quanto è proprio da qui che si decide il modello da applicare.
- Il terzo quadrante rappresenta l'effettivo sviluppo e verifica del progetto. In questa fase si introducono i concetti iniziali e le specifiche funzionali. Ci si domanda quindi cosa deve fare il sistema.
- Nell'ultimo quadrante si passa alla fase di pianificazione della fase successiva. Il ciclo continua seguendo il tracciato della spirale fino ad arrivare al momento del rilascio del prodotto.

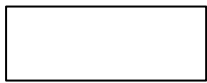
La differenza tra questo modello e gli altri sta nel fatto che in ciascuna delle spirali la scelta dei passi può essere effettuata secondo diversi modelli, in quanto inglobati al suo interno, avendo perciò il vantaggio di utilizzare un modello diverso ad ogni passo.

# Modelli Entità-Relazione

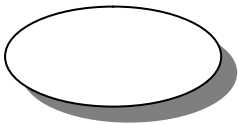
Un aspetto fondamentale per la realizzazione di prodotto software di qualità sono le *Specifiche delle Richieste*, ovvero come evidenziare le funzionalità in maniera più dettagliata. Attenzione perché il documento che verrà prodotto da questo studio sarà la base per la stipula del contratto tra cliente e produttore.

Per mostrare il modello logico dei dati, introdurremo il modello Entità-Relazione in modo tale da controllare gli oggetti che partecipano al prodotto software ed il modo in cui questi oggetti sono relazionati.

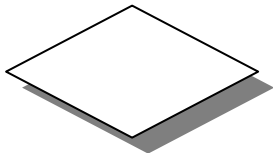
Per rappresentare graficamente un modello E/R useremo i seguenti simboli:



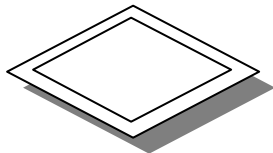
Con il rettangolo indicheremo le Entità



Con gli ovali gli Attributi

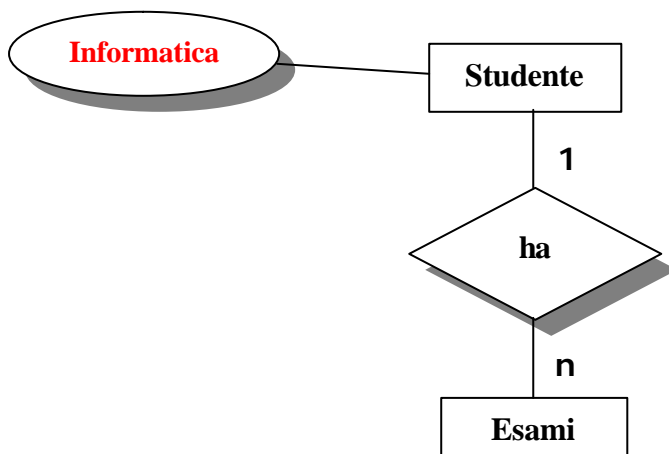


Con il rombo le Relazioni tra le entità



Con il doppio rombo le Relazioni con Ereditarietà

## Esempio



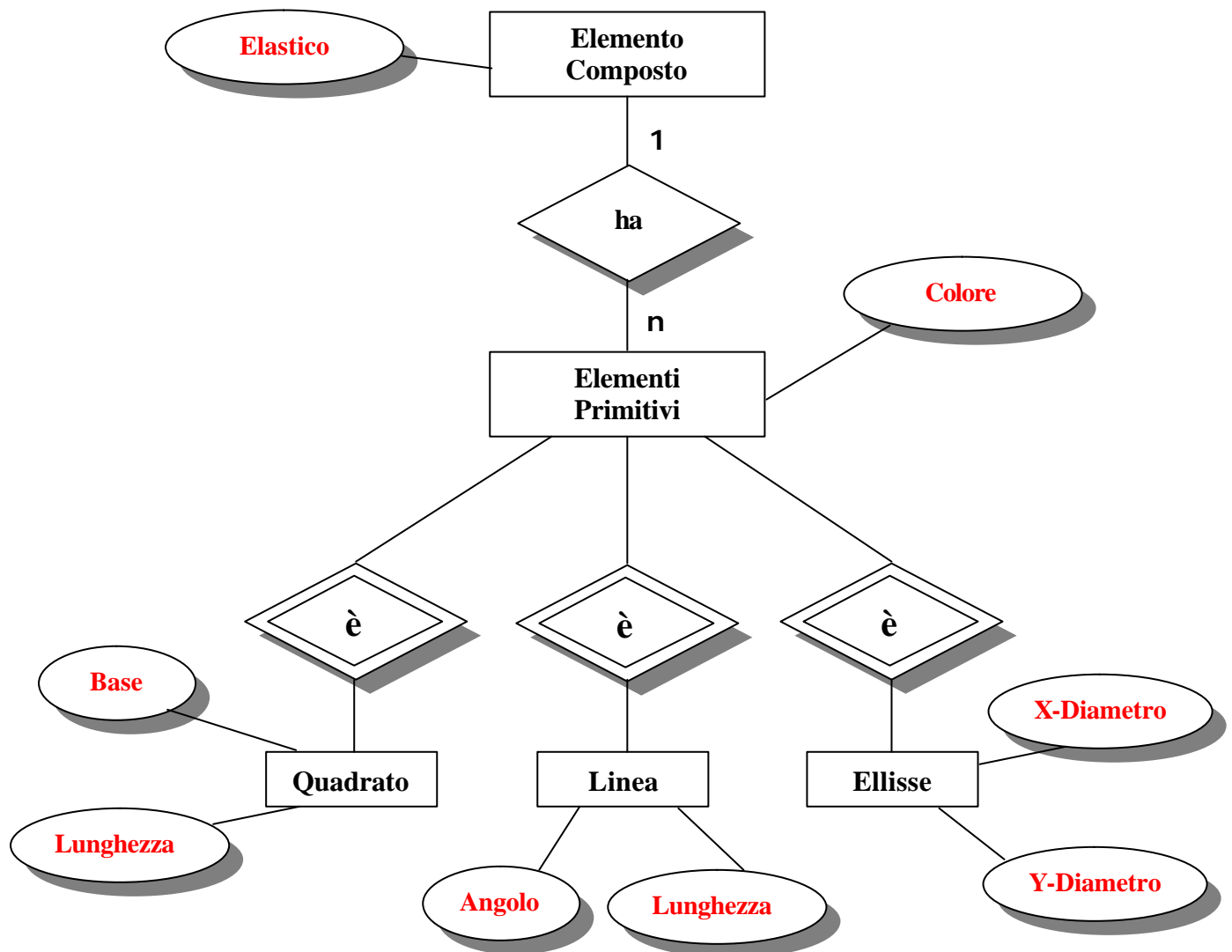
Supponiamo di essere all'università, ciascun studente iscritto ad una facoltà ha  $n$  esami da superare. Ne consegue una relazione tra l'entità "Studente" e l'entità "Esami" del tipo 1 a  $N$ . Possiamo aggiungere a questo punto, ad ogni entità, un attributo come quello in figura, quindi ogni studente avrà la caratteristica di essere iscritto alla facoltà di Informatica.

L'attributo descrive la caratteristica che lo studente è di Informatica e non di Storia.

Per quanto riguarda le relazioni di ereditarietà, se due entità sono in relazione, la seconda ne eredita tutti gli attributi senza la necessità di riportarli.

Esempio:

Il seguente modello descrive un oggetto composto da un editor grafico, il quale è in grado di costruire simboli mettendo insieme quadrati, linee ed ellissi.



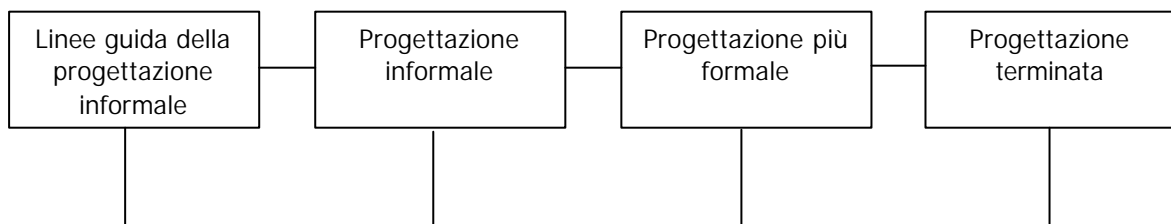
# Progettazione del software

La progettazione è un processo creativo che richiede esperienza e qualche furbizia da parte del progettista. Per passare a questa fase, dobbiamo aver concluso la fase delle specifiche delle richieste, cioè aver concluso il dialogo con l'utente.

La progettazione del software include un numero di sottofasi diverse:

- 1) Studiare e capire il problema: Senza questa conoscenza, la progettazione effettiva del software non può essere effettuata. Il problema "dovrebbe" essere esaminato da angoli diversi, in modo tale da avere più punti di osservazione verso le richieste di progettazione.
- 2) Identificare le caratteristiche evidenti di almeno una possibile soluzione: E' spesso utile avere un discreto numero di soluzioni ed identificare ognuna di esse. La scelta della soluzione dipende: dall'esperienza del progettista che tende a scegliere quella più familiare o già esistente; oppure dalla disponibilità di componenti riusabili e dalla semplicità delle soluzioni derivate. Ricordiamo che **a parità di fattori la soluzione più semplice è quella da prendere in considerazione**.
- 3) Descrivere ogni astrazione usata nella soluzione: Prima di scrivere la documentazione formale, il progettista può ritenere opportuno costruire una descrizione informale del progetto e farne il debug per svilupparlo in più dettagli. Così gli errori e le omissioni nella progettazione ad alto livello possono essere corretti prima che venga documentata la progettazione stessa.

## Processo della Progettazione



Vediamo ora le attività essenziali nella progettazione:

- **Progettazione Architettuale**: Descrive un sistema con vari sottosistemi, cioè con moduli, senza occuparsi di cosa è contenuto in ognuno di essi.
- **Specifiche Astratta**: E' fornita per dare informazioni sui moduli del sistema, dando i vincoli al di sotto dei quali operare.
- **Progettazione dell'Interfaccia**: Descrive come i vari moduli dialoghino tra loro e con l'utente.
- **Progettazione delle componenti**: Si inizia dando una bozza di progettazione utilizzando le varie strutture dati e successivamente si progetta l'algoritmo.
- **Progettazioni delle Strutture Dati**: Le strutture dati usate nell'implementazione del sistema vengono descritte nel dettaglio e specificate.
- **Progettazione dell'Algoritmo**: Gli algoritmi usati per fornire i servizi sono progettati in dettaglio e specificati.

Un approccio ampiamente raccomandato è un approccio TOP-DOWN dove il problema è partizionato ricorsivamente in sottoproblemi.

La tecnica Top-Down è un valido approccio dove i componenti della progettazione sono uniti strettamente. Tuttavia con una progettazione orientata agli oggetti tale tecnica è poco utile.

## Strategie di Progetto

Nel corso degli anni due tecniche sono state portate avanti tra le strategie di progettazione e possono essere riassunte nel modo seguente:

- Progettazione Funzionale;

Il sistema è progettato da un punto di vista funzionale, cioè partendo da una visione ad alto livello e raffinandola progressivamente in un progetto più dettagliato.

- Progettazione Orientata agli Oggetti;

Il sistema è visto come una collezione di oggetti, piuttosto che di funzioni e ciascun oggetto gestisce le proprie informazioni. Infatti gli oggetti hanno un insieme di attributi che definiscono il loro stato e le operazioni che agiscono su questi attributi.

Una delle differenze sostanziali tra le due tecniche sta nel fatto che la prima è stata molto usata in tempi passati per progetti sia di piccola che di grande scala, in diverse aree di applicazione; la seconda invece ha avuto uno sviluppo molto più recente e che incoraggia la produzione di sistemi composti da componenti indipendenti/interagenti.

Questi due approcci non devono tuttavia essere visti come tecniche concorrenziali, bensì complementari, ognuna delle quali può infatti essere applicata alle diverse fasi del processo di progettazione.

***“L’ingegnere software sceglie l’approccio più appropriato per ciascuna fase del processo di progettazione”.***

# Qualità della Progettazione

La qualità di una buona progettazione nasce da quattro principi fondamentali:

- **Coesione:**

E' la capacità di costruire una componente del sistema in modo tale che gli elementi che la costituiscono siano messi insieme in base a dei criteri e non siano ripetuti in altre componenti;

- **Accoppiamento:**

E' realizzato con scambi di messaggi fortemente dipendenti l'uno dall'altro. I moduli sono fortemente accoppiati se essi fanno uso di variabili condivise o se essi si scambiano l'informazione di controllo. L'accoppiamento debole è ottenuto laddove l'informazione è mantenuta in una componente e che la sua interfaccia dati con le altre unità avvenga attraverso una lista di parametri.

- **Comprensibilità:**

Per una buona comprensibilità del progetto, da parte di una persona che non abbia mai avuto a che fare con lo stesso, bisogna che ci sia:

- o un'alta *coesione*, in modo tale che la componente possa essere compresa senza far riferimento ad altre componenti
- o una buona *nominazione*, assegnando nella componente dei nomi significativi;
- o una chiara e dettagliata *documentazione*;
- o una più bassa possibile *complessità* degli algoritmi usati per l'implementazione della componente.

- **Adattabilità:**

Se un progetto deve essere mantenuto, esso deve essere prontamente adattabile; vale a dire che le sue componenti devono avere un basso accoppiamento. Nei sistemi OO l'adattabilità è uno dei vantaggi principali.

# FOD - Function Oriented Design

## (Progettazione Orientata alle Funzioni)

*L'approccio ad una Progettazione Orientata alle Funzioni fa sì che il progetto venga suddiviso in un insieme di unità interagenti che hanno una funzione chiaramente definita.*

Anche se risulta essere un metodo alquanto antiquato, molte organizzazioni hanno ancora standard e metodi di sviluppo basati sulla decomposizione funzionale e non sono certamente d'accordo ad una sua migrazione verso l'object oriented. Ecco perché continuerà ad essere largamente praticato.

Un approccio alla progettazione orientata alle funzioni è il cosiddetto **Progetto Strutturato** definito nel modo seguente:

- **DFD** (Data Flow Diagram):

I diagrammi di flusso mostrano come l'input dei dati è trasformato in output attraverso una sequenza di trasformazioni funzionali. Questi diagrammi di solito non includono informazioni di controllo, ma documentano solo le trasformazioni dei dati.

- **Structure Charts:**

Le tabelle di struttura sono un modo grafico per mostrare la struttura gerarchica delle componenti di un sistema.

- **Data Dictionaries:**

Rappresentano i dizionari dei dati, utili nel processo di progettazione, sono un modo per poter descrivere i links e le descrizioni diagrammatiche del progetto. Ciascuna entità "dovrebbe" essere presente all'interno del dizionario dei dati e dare informazioni circa: il suo tipo, la sua funzione e una spiegazione della sua inclusione. Il data dictionary riduce i rischi di riutilizzo dei nomi e della loro correttezza, inoltre permette al lettore una chiara visione del progetto e del modo di pensare del progettista.

*Chiedo scusa ai lettori di questo corso se sulla FOD non mi sono dilungato più di tanto e non ho chiarito dei concetti chiave che invece vengono solamente annunciati, ma il nostro intento è quello di dare una visione un po' più ampia della Progettazione Object Oriented.*

# OOD - Object Oriented Design

## (Progettazione Orientata agli Oggetti)

### Information Hiding

Prima di dare la definizione di OOD vorrei introdurre il concetto di Information Hiding.

*"Nascondere l'informazione" è una strategia adottata per permettere che ogni informazione venga nascosta all'interno della progettazione delle componenti del progetto.*

In tal modo si eliminano alcuni vincoli della logica di controllo e della struttura dati delle implementazioni, e le comunicazioni attraverso le condivisioni delle variabili globali è ridotto al minimo incrementando così la comprensibilità del progetto.

### Object Oriented Design – Caratteristiche e Vantaggi

La *progettazione orientata agli oggetti* si basa sulla strategia dell'Information Hiding (nascondere l'informazione) e differisce dall'approccio precedente in quanto vede un sistema software come un insieme di oggetti interagenti con il proprio stato privato e non come un insieme di funzioni.

***L'Object-Oriented è un modello computazionale che, se inteso ed applicato in modo formale, può assistere il progettista software nell'applicazione delle più attuali tecnologie ad oggetti che su di esso si fondano.<sup>1</sup>***

Le caratteristiche di un progetto object oriented sono:

- Eliminazione delle aree di dati condivise: gli oggetti comunicano scambiandosi messaggi piuttosto che condividere variabili, riducendo così il grado di accoppiamento dell'intero sistema.
- Gli oggetti sono entità indipendenti: essi possono essere prontamente cambiati poiché l'informazione di tutto lo stato è contenuta all'interno dello stesso oggetto.
- Gli oggetti possono essere distribuiti e possono essere eseguiti sia in modo sequenziale che parallelo.

Tali caratteristiche comportano una serie di vantaggi:

- Il sistema sviluppato potrebbe essere più facile da mantenere in quanto gli oggetti sono entità autonome.
- Gli oggetti sono componenti riusabili e quindi i progetti possono essere creati utilizzando oggetti creati in progetti precedenti.
- Per alcune classi del sistema c'è un chiaro mapping fra l'entità reali e i loro oggetti di controllo migliorando la comprensibilità del progetto.

---

<sup>1</sup> Ed Seidewitz, Mike stark – "Reliable Object-Oriented Software – Applying Analysis and Design", Advances in Object Technology, SIGS Books, 1995



Tuttavia lo svantaggio sta nella difficoltà di individuazione di opportuni oggetti del sistema.

## OOD e OOP – Design e Programming

La **Progettazione** orientata agli oggetti è spesso confusa con la **Programmazione** orientata agli oggetti.

- La *Programmazione* è un linguaggio che permette la diretta implementazione degli oggetti.
- La *Progettazione* invece è una strategia del prodotto e non dipende assolutamente da qualche specifico linguaggio.

Quindi il compito dell'Ingegnere o Progettista Software che utilizza l'Object-Oriented, sta nel descrivere il "cosa" (l'entità che si vuole creare) senza necessariamente indicare il "come" (i meccanismi implementativi legati al linguaggio di programmazione da utilizzare).

# OOP - Object Oriented Programming

## (Programmazione Orientata agli Oggetti)

### Il Riuso Object-Oriented

La programmazione orientata agli oggetti rappresenta una nuova frontiera della programmazione. Questo nuovo modo di programmare permette di creare programmi molto complessi in poco tempo grazie alla facilità con la quale parti di programmi già esistenti possono essere riutilizzati.

Andiamoci cauti tuttavia con l'uso della parola "riutilizzare", infatti *"...la prima cosa che un buon progettista software ad oggetti deve imparare è che il riuso non si ottiene gratis, in quanto non deriva dal semplice atto di utilizzare un ambiente di sviluppo object-oriented. In realtà, il riuso è un concetto molto più ampio nel cui contesto il livello di codifica è sicuramente quello meno produttivo..."*<sup>2</sup>.

A volte capita che per poter riutilizzare oggetti creati in precedenza, affinché si integrino al meglio con il nuovo contesto, bisogna apportare piccole modifiche, a patto che si abbiano i sorgenti di tali oggetti. Ciò non è sempre immediato. La soluzione sarebbe quella di progettare interfacce stabili per progetti futuri, ma purtroppo ciò non avviene in quanto non si possono sempre prevedere i contesti e le necessità dei progetti che verranno. Quindi per risolvere il problema dovremmo o modificare i vecchi oggetti e adattarli alle nostre esigenze o viceversa lasciare intatti i vecchi oggetti e adeguare il nostro progetto.

La scelta non è semplice:

- 1) nel primo caso modificando gli oggetti già creati, potremmo complicare le cose dando inizio ad una serie di errori non presi in considerazione al momento della creazione degli stessi oggetti in quanto le esigenze non avevano dato modo che emergessero;
- 2) nel secondo caso adeguando il nostro progetto, potremmo ottenere come risultato un aumento del livello di accoppiamento tra i vari componenti dell'applicazione, perdendo i benefici delle caratteristiche dell'OOD, come ad esempio l'information hiding e l'inheritance.

Quando progettiamo i componenti di un software dobbiamo sempre tenere a mente i principi dei concetti di "uso" e "riuso", ovvero:

- USO: Funzionalità, Efficienza e Facilità d'uso
- RIUSO: Generalità e Modificabilità

in tal modo un progettista software sa a priori il fine dell'oggetto che sta creando.

Tuttavia nel momento del riuso, per ogni minima modifica effettuata, sia che si adotti la prima o la seconda delle precedenti scelte, non è sufficiente basarsi sulla correttezza di uno strumento automatico di test come nello *unit test*, in quanto quest'ultimo verifica solamente porzioni di codice a prescindere dal contesto nel quale l'applicazione verrà eseguita. Sarà necessario eseguire almeno un *integration test*, progettare un insieme completo di test di regressione e testare l'intera applicazione nel suo insieme (*system test*). Tali verifiche anche se comportano dei costi aggiuntivi, sono necessarie altrimenti non si potrà ottenere il livello di qualità che il modello object-oriented augura.

---

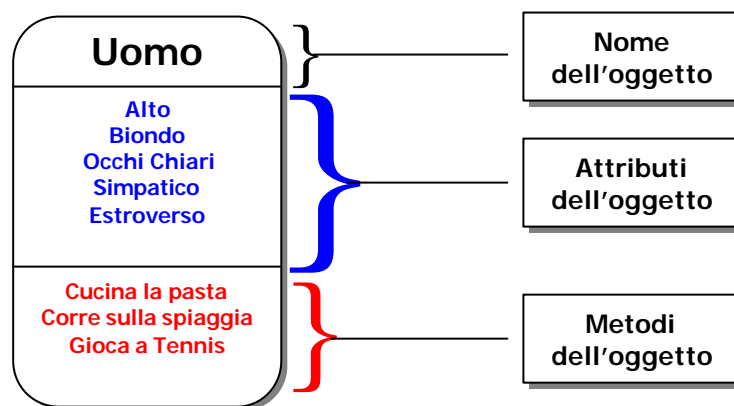
<sup>2</sup> Scott Amber – "A Realistic Look at Object-Oriented Reuse", MSDN periodic library, 1998

## La Programmazione Orientata agli Oggetti

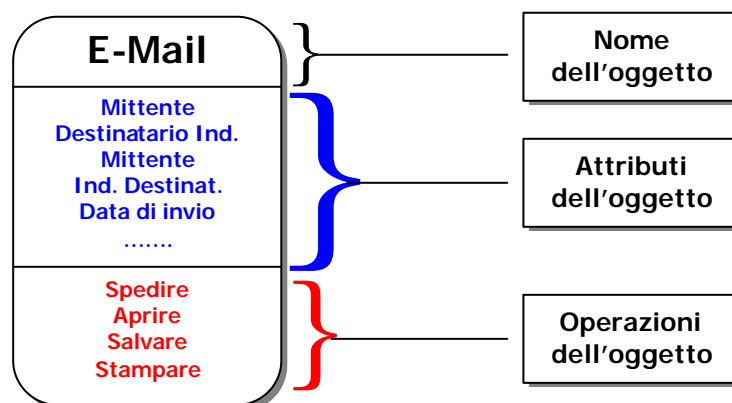
Programmare per oggetti significa definire oggetti stabilendone *Proprietà* e *Comportamenti*.

### Ma cosa sono questi oggetti?

Iniziamo con l'immaginare un programma secondo l'OOP, come un intero mondo circondato da tanti oggetti. Ogni oggetto (o *istanza*) ha delle caratteristiche fisiche e comportamentali. Fisiche come ad esempio quelle che può avere un uomo: alto, basso, biondo, scuro, simpatico, ecc.; e comportamentali come ad esempio cucina la pasta, corre sulla spiaggia, gioca a tennis. Ebbene le caratteristiche fisiche le chiameremo **Proprietà o Attributi**, mentre le quelle comportamentali le chiameremo **Metodi o Operazioni**.

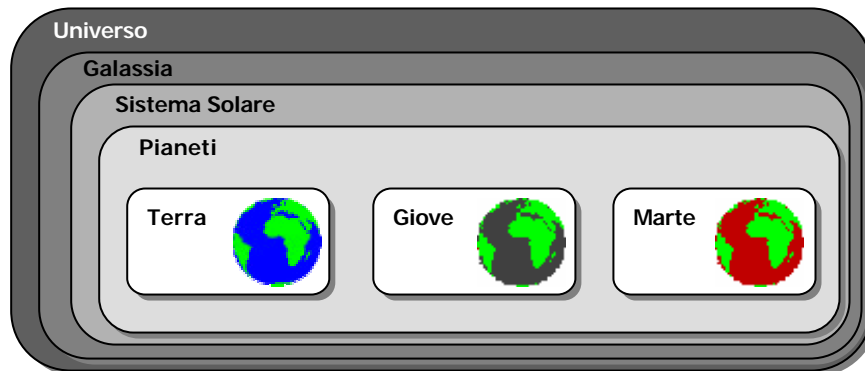


Vediamo un altro esempio prendendo in esame un Messaggio di Posta Elettronica. La descrizione dell'oggetto messaggio sarà la seguente:



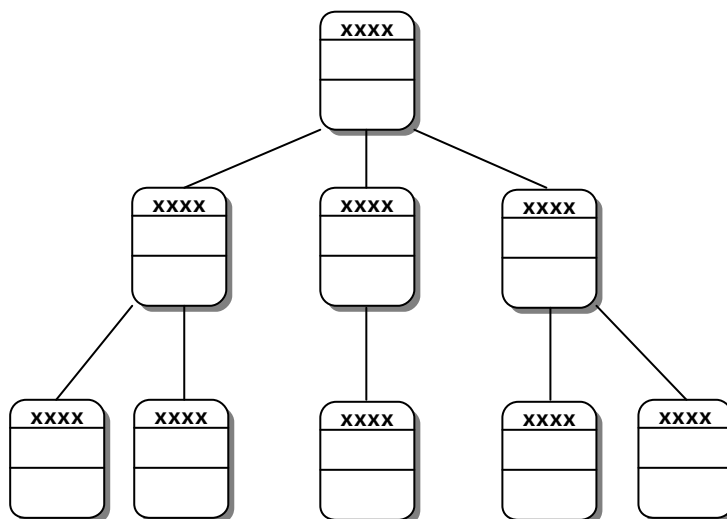
### Che cosa è una classe?

Una classe è banalmente un contenitore astratto per oggetti che hanno in comune delle proprietà e dei metodi. Vediamo l'esempio grafico di seguito riportato:



*Terra, Giove e Marte* sono *istanze* (o *oggetti*) della *Classe Pianeti*, che a sua volta fa parte della *Classe Sistema Solare*, contenuta nella *Classe Galassia* e così via. In particolare chiameremo **Sottoclasse** una classe contenuta in un'altra e **Superclasse** una classe che ne contiene altre. Nel nostro caso diremo che la *Classe Sistema Solare* è la **superclasse** della *Classe Pianeti* e la *Classe Pianeti* è una **sottoclasse** della *Classe Sistema Solare*.

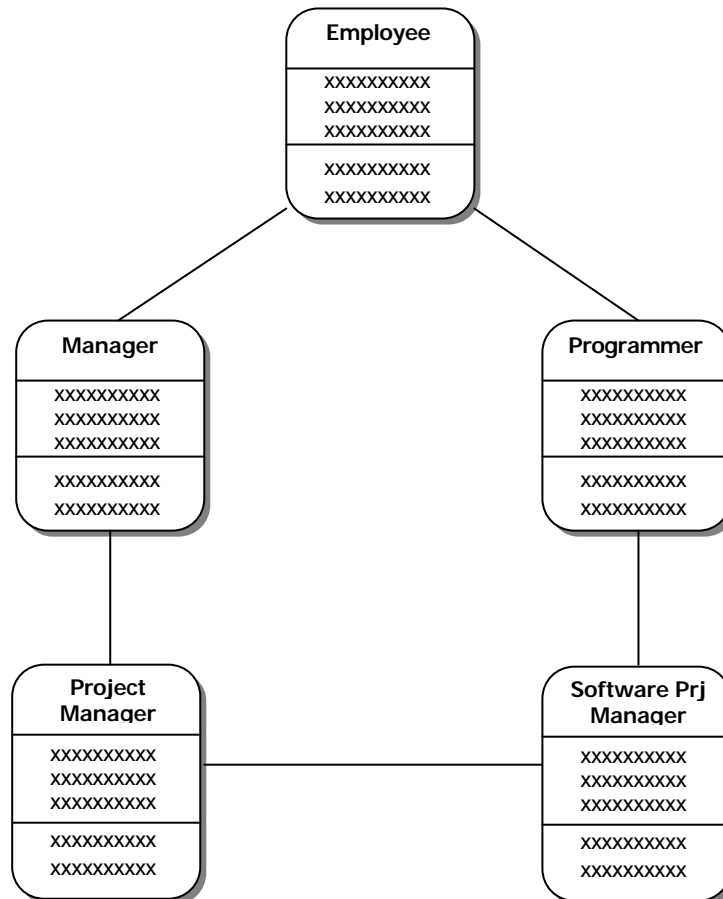
Quando una classe viene creata all'interno di una superclasse, la sottoclasse eredita tutti gli attributi e i metodi della sua superclasse. Tale processo viene chiamato **Ereditarietà** (Inheritance).



Una **Classe Astratta** è una classe che non contiene istanze e definisce solo in parte gli attributi e i metodi dei propri oggetti. Questi ultimi a loro volta, sono sottoclassi che possono contenere delle istanze completando la loro descrizione. Nel nostro caso le classi *Universo*, *Galassia* e *Sistema Solare* sono classi astratte.

Le *interfacce* sono l'estremizzazione di una classe astratta e contengono solo la dichiarazione dei metodi. Sono spesso utilizzate per specificare API standard.

Quando una sottoclasse eredita da più superclassi si ha una *Ereditarietà Multipla* creando così una rete di ereditarietà piuttosto che un albero:



In questo esempio è illustrata una situazione in cui un programmatore è stato nominato project manager. Gli attributi della classe Software Project Manager sono ereditati da entrambi le classi Manager e Programmer. Fare attenzione in questi casi ai nomi comuni ereditati da superclassi differenti, in tal caso la soluzione sta nel permettere la rinomina di alcuni attributi.

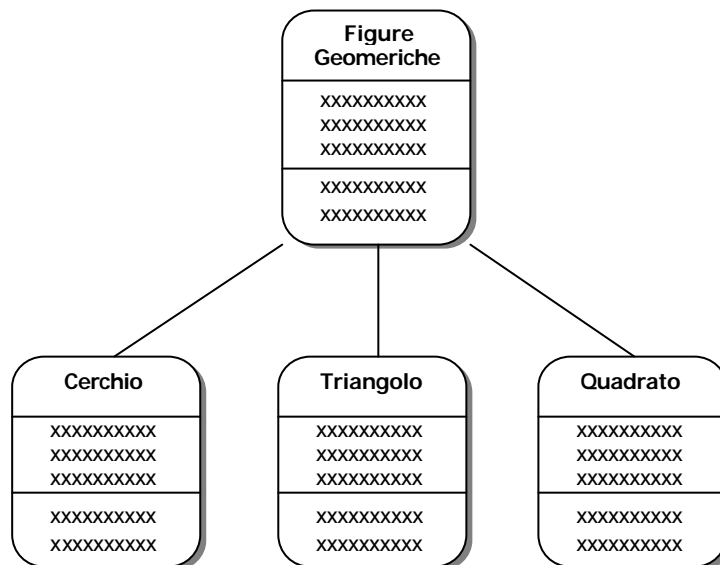
Un OOL (Object Oriented Language – Linguaggio di programmazione orientato agli oggetti) è un linguaggio che permette la diretta implementazione degli oggetti, delle classi e quindi supporta la loro ereditarietà.

## Polimorfismo

Il polimorfismo è una caratteristica essenziale dei linguaggi orientati agli oggetti e permette di trattare in modo uniforme tipi "simili" di oggetti, ma rispettandone le differenze. In altre parole in base alla posizione nella gerarchia di un oggetto, stabilita dall'ereditarietà, tale oggetto muta il metodo ereditato (utilizzando lo stesso nome) in base al proprio comportamento.

Ad esempio adottiamo la classe delle figure geometriche e le rispettive istanze. Ebbene ereditano la caratteristica di essere figure geometriche, ma la mutano in base al proprio comportamento (metodo).

Ricordiamo che gli oggetti di una stessa classe hanno metodi uguali, ma dati diversi.



La classe Figure Geometriche dichiara la funzione, mentre le altre sottoclassi ne forniscono le diverse implementazioni.

## Meccanismo fondamentale di computazione della OOP

Le istruzioni di un OOL dipendono dalla sintassi del linguaggio che si decide da usare e sono differenti tra loro. Tuttavia la regola standard che si ritrova nella maggior parte dei linguaggi per indicare ad un metodo su quale oggetto agire è la seguente:

