Appunti di Metodi Avanzati di Programmazione

Federico Calò

Indice

1 Premessa

Appunti dell'esame di Metodi Avanzati di Programmazione che tratta argomenti di OOP e Java. Se trovate degli errori siete pregati di segnalarli. Trovate tutte le informazioni sul mio sito www.federicocalo.it

2 Introduzione

La scienza del software studia il passaggio dal **"cosa"** al **"come"**. L'estremo **"cosa"** raccoglie gli obiettivi, i desideri e i bisogni dell'utente, la sua conoscenza del dominio del problema, espressi nel linguaggio naturale. L'estremo **"come"**, invece, è l'hardware che può raggiungere quegli obiettivi e soddisfare i bisogni e i desideri in modo estremamente procedurale e rigorosamente sequenziale. Per plasmare il processo di analisi, progettazione e sviluppo di un programma sono nati diversi paradigmi di programmazione. Questi modelli concettuali strutturano il pensiero in quanto determinano la forma di programmi validi. Essi influenzano il modo in cui pensiamo e formuliamo le soluzioni, arrivando a condizionare perfino la possibilità di trovare una soluzione.

Ogni linguaggio di programmazione supporta un particolare paradigma fornendo:

- Le **primitive** di quel paradigma
- I **metodi di composizione** di quel paradigma
- Un **linguaggio utente appropriato** che rende chiari i programmi scritti secondo quel paradigma
- Una **esecuzione efficiente** di programma scritti in quello stile.

I linguaggi di programmazione sono dotati di opportuni costrutti linguistici che riflettono i modelli concettuali di un paradigma, al fine di facilitare l'espressione di una soluzione definita attraverso i modelli concettuali del paradigma. In realtà, i linguaggi di programmazione possono supportare più di un paradigma.

I paradigmi che supportano la programmazione ad alto livello possono essere raggruppati in base al loro approccio alla soluzione de problemi:

- **Approccio operazionale**: descrive per passi come costruire una soluzione
- **Approccio dimostrazionale**: è una variante del precedente che illustra la soluzione in modo operativo per esempi specifici e lasca al sistema il compito di generalizzare queste soluzioni di esempi ad altri casi
- **Approccio definizionale**: esso stabilisce delle proprietà della soluzione in modo da vincolarla senza per questo descrivere come viene calcolata.

Nessun paradigma è appropriato per tutti i problemi. Le applicazioni complesse necessitano l'adozione di più paradigmi di programmazione. Ogni singolo paradigma di programmazione è caratterizzato da:

- Un diverso **potere di elisione**, cioè capacità di esprimere qualcosa in modo conciso.
- Una diversa **invarianza rispetto ai cambiamenti** apportati nella strategia di soluzione di un problema.

Occorre sempre bilanciare i costi dovuti all'uniformità di paradigma con i costi determinati dall'uso di diversi paradigmi per un medesimo problema. I costi sono:

- Costi iniziali di apprendimento
- Costi continuati di debugging
- Costi di variazione dovuti all'evoluzione del programma
- Costi di esecuzione dell'applicazione

I **paradigmi operazionali** sono caratterizzati da sequenze computazionali passo-passo. E' difficile stabilire se l'insieme dei valori calcolati operativamente è proprio l'insieme dei valori soluzione. Le tecniche di **verifica** e di **debugging** cercano di superare questo problema di programmazione. Questi paradigmi operazionali si distinguono in:

- **Side-effecting**: procedono modificando ripetutamente la loro rappresentazione dei dati
- **Non-side-effecting**: procedono cerando continuamente nuovi dati. Questi paradigmi includono quelli che tradizionalmente sono detti funzionali.

Inoltre i paradigmi operazionali con effetti collaterali si distinguono in:

- Imperativi
- Orientati agli oggetti

Inoltre in base al tipo di flusso di controllo, possiamo distinguere i paradigmi in:

- Sequenziale: il flusso di controllo è unico
- Concorrente o parallelo: quando sono ammessi più flussi di controllo

3 Astrazione nella progettazione

Astrarre deriva dal latino e significa "tirare via". Nella progettazione generalmente l'astrazione riguarda il cambiamento della rappresentazione di un problema, con l'obiettivo di concentrarsi su aspetti rilevanti, dimenticando gli elementi incidentali. Sia ben chiaro, non si tratta di **omettere** parti della rappresentazione di un problema, ma di **riformulare** lo stesso concentrando la propria attenzione su idee generali piuttosto che su manifestazioni specifiche di quelle idee, tenendo conto della prospettiva di un osservatore. Quindi l'astrazione si focalizza sulle caratteristiche essenziali di un oggetto rispetto alla prospettiva di colui che lo osserva.

Il termine attrazione sotto-intende:

- **Un processo**: l'estrazione delle informazioni essenziali e rilevanti per un particolare scopo, ignorando il resto dell'informazione.
- **Un entità**: una descrizione semplificata di un sistema che enfatizza alcun dei dettagli o proprietà trascurandone altri.

Nell'analisi dei sistemi un'astrazione è una descrizione semplificata del sistema, che enfatizza alcuni dettagli o proprietà essenziali del sistema mentre ne ignora altri estranei.

Nella programmazione l'astrazione allude alla distinzione che si fa tra:

- **cosa** fa un pezzo di codice
- **come** esso è implementato

Per l'utente del codice l'essenziale è cosa fa il codice mentre non è interessato ai dettagli dell'implementazione.

I sistemi software diventano sempre più complessi. Per padroneggiare la complessità è necessario concentrarsi solo sui pochi aspetti che più interessano in un certo contesto ed ignorare i restanti. L'**astrazione** permette ai progettisti di sistemi software di risolvere problemi complessi in modo organizzato e gestibile.

L'**astrazione funzionale** si riferisce alla progettazione del software, e in particolare alla possibilità di specificare un modulo software che trasforma dei dati di input in dati di output nascondendo i dettagli algoritmici della trasformazione. Quindi il consumatore conosce solo le corrette convenzioni i chiamata, **specifica sintattica**, e cosa fa il modulo, **specifica semantica**, inoltre deve fidarsi del risultato.

La specifica sintattica indica il nome del modulo, il tipo di dato che riceve in input e il tipo di risultato, in moo da permettere la corretta chiamata del modulo. Ad esempio: $diff(intero, intero) \rightarrow float$

La specifica semantica indica la trasformazione operata, cioè la funzione calcolata:

$$diff(a,b) = \begin{cases} \frac{a}{b} & b > 1 \\ Impossibile & Altrimenti \end{cases}$$

Come la trasformazione è calcolata non è noto al fruitore.

Per specificare la semantica di un modulo i può esprimere attraverso due predicati la relazione che lega i dati di ingresso ai dati di uscita. Se il primo predicato, chiamato **precondizione**, è vero sui dati di ingresso e se il programma termina su quei dati, allora il secondo predicato, chiamato **post-condizione** è vero sui dati di uscita. Queste specifiche semantiche sono dette assiomatiche.

Lo **Stepwise Refinement** o **Top-down Programming** è una metodologia che mira a costruire i programmi progredendo dal generale al particolare. Per risolvere un problema K si procede come segue:

1. Si decompone il compito K in sotto compiti $K_1, K_2, \ldots, K_n$
2. Si ipotizza di disporre di moduli $M_1, M_2, \ldots, M_n$ che effettuano le trasformazioni richieste rispettivamente da $K_1, K_2, \ldots, K_n$
3. Si compone un modulo M che assolve al compito P usando i moduli $M_1, M_2, \ldots, M_n$
4. Si applica ricorsivamente la metodologia ai sotto compiti $K_1, K_2, \ldots, K_n$ al fine di definire la realizzazione di $M_1, M_2, \ldots, M_n$ fino a quando non si ottengono sotto compiti considerati elementari o non ulteriormente decomponibili.

Secondo questa metodologia il programmatore è libero di assumere l'esistenza di qualsiasi modulo che si può applicare al particolare sotto compito e di cui fornisce una specifica, salvo dover poi specificare come quel modulo va realizzato.

L'astrazione funzionale presenta diversi limiti. I dettagli relativi alla rappresentazione dei dati di input e output devono essere conosciuti da chi poi andrà a realizzare il modulo. La rappresentazione è solitamente condivisa fra diversi moduli, per cuui i cambiamenti apportati alla rappresentazione dei dati in input/output a un modulo si possono ripercuotere su moli altri moduli. L'astrazione funzionale non permette di sviluppare soluzioni invarianti ai cambiamenti nei dati. Questo rende difficoltosa la manutenzione delle soluzioni progettate.

4 Astrazione dati e Specifiche

Alla base dell'**Astrazione dati** c'è il principio che non si può accedere direttamente alla rappresentazione di un dato, qualunque esso sia, ma solo attraverso un insieme di operazioni considerate lecite. Un enorme vantaggio di questo principi è il fatto che un cambiamento nella rappresentazione del dato si ripercuoterà solo sulle operazioni lecite che potrebbero subire delle modifiche, mentre non inficerà il codice che utilizza il dato astratto.

In generale un principio i astrazione suggerisce di **occultare l'informazione** (information hiding) sulla rappresentazione del dato, sia perchè non necessaria al fruitore dell'entità astratta, sia perchè la sua rivelazione creerebbe delle inutili dipendenze che comprometterebbero l'invarianza ai cambiamenti.

L'incapsulamento è una tecnica di progettazione consistente nell'impacchettare una collezione di entità, creandone una barriera concettuale. Come l'strazione, l'incapsulamento sottointende:

- **un processo**: l'impacchettamento
- **una entità**: il pacchetto ottenuto

L'incapsulameto non indica come devono essere le pareti del pacchetto, che potranno essere:

- **trasparenti**: permettendo di vedere tutto ciò che è stato impacchettato;
- **traslucide**: permettendo di vedere in modo parziale il contenuto
- **opache**: nascondendo tutto il contenuto del pacchetto

Facciamo un piccolo confronto tra l'astrazione dati e l'astrazione funzionale. L'astrazione dati ricalca ed estende quella funzionale. L'astrazione funzionale stimola gli sforzi per evidenziare operazioni ricorrenti o comunque ben caratterizzate all'interno della soluzione di un dato problema. L'astrazione di dati stimola di più gli sforzi per individuare le organizzazioni dei dati più consone alla soluzione del problema.

Quando si esegue l'astrazione e l'incapsulamento dei dati, l'isolamento dei moduli non può essere totale. La specifica descrive come si può interagire con un dato astratto. L'astrazione deve dividere quelli che sono gli interessi degli utenti, interessati a cosa si astrae, e gli interessi degli implementatori, interessati a come si realizza. Per questa ragione una definizione di astrazione ha sempre due componenti: una specifica e una realizzazione.

Per descrivere una specifica occorre ricorrere a dei linguaggi di specifica, che sono diversi dai linguaggi usati per descrivere la realizzazione delle astrazioni. La specifica potrà essere:

- **sintattica**: stabilisce quali identificatori sono associati all'astrazione

- **semantica**: definisce il risultato della computazione inclusa nell'estrazione.

L'efficacia di un'astrazione può essere migliorata mediante l'uso di parametri per la comunicazione con l'ambiente esterno. Quando un'astrazione è chiamata, ciascun parametro formale verrà associato in qualche modo al corrispondente argomento. I meccanismi utilizzati per realizzare queste associazioni sono fondamentalmente due:

- **meccanismi di copia**: copiano il valore da passare
- **meccanismi definizionali**: legano direttamente il parametro formale alla definizione dell'argomento passato.

Anche un'astrazione dati, come qualunque astrazione, è costituita da una specifica e una realizzazione:

- La **specifica** consente di descrivere un nuovo dato e gli operatori che ad esso sono applicabili
- La **realizzazione** stabilisce come il nuovo dato e i nuovi operatori vengono ricondotti ai dati e agli operatori già disponibili.

Chi utilizzerà questi dati con i nuovi operatori nella scrittura dei suoi programmi sarà tenuto a conoscere la specifica dell'astrazione dei dati, ma potrà astrarre dalle tecniche utilizzate per la realizzazione.

I linguaggi di specifica per le astrazioni dati più noti sono le **specifiche assiomatiche**, che utilizzano un linguaggio logico-matematico usato nelle asserzioni, e le **specifiche algebriche**, che utilizzano il linguaggio dell'algebra usato nelle equazioni definite fra gli operatori specificati nel dato astratto.

Le specifiche assiomatiche dei dati astratti sono costituite da asserzioni, costituite da una specifica sintattica detta anche **segnatura**, la quale fornisce l'elenco dei nomi dei domini e delle operazioni specifiche del tipo e i domini di partenza e di arrivo per ogni nome di operatore, e una specifica **semantica**, la quale associa un insieme ad ogni nome di tipo introdotto nella specifica sintattica e una funzione ad ogni nome di operatore, esplicitando le condizioni sui domini di partenza e di arrivo.

Il metodo di specifica assiomatica è preciso nella definizione della specifica sintattica, mentre è piuttosto informale per gli altri aspetti, pertanto non consente di caratterizzare precisamente un tipo astratto. Inoltre non consente di definire i valori che possono essere generati mediante l'applicazione di operatori e non consente di stabilire quando l'applicazione di diverse sequenze di operatori porta al medesimo valore.

Le specifiche algebriche si basano sull'algebra, piuttosto che sulla logica. Essenzialmente definiscono un dato astratto come un'algebra eterogenea, ovvero come una collezione di diversi insiemi su cui sono definite diverse operazioni. Una specifica algebrica consiste in tre parti:

1. **Sintattica**: elenca i nomi del tipo, le sue operazioni e il tipo degli argomenti delle operazioni. Se un'operazione è una funzione allora è specificato il codominio della funzione.
2. **Semantica**: consiste in un insieme di equazioni algebriche che descrivono in modo indipendente dalla rappresentazione le proprietà delle operazioni.
3. **Di restrizione**: stabilisce varie condizioni che devono essere soddisfatte o prma che siano applicate le operazioni o dopo che esse siano state completate.

Uno desgli aspetti più interessanti delle specifiche algebriche è la semplicità del linguaggio di specifica rispetto ai linguaggi di programmazione procedurale.

Scrivere delle specifiche semantiche complete, consistenti e non ridondanti può non essere semplice. Per facilitare il compito si è introdotta una metodologia che si basa sulla distinzione degli operatori di un dato astratto in **costruttori**, che creano o istanziano il dato, e **osservazioni**, che ritrovano informazioni sul dato astratto.

Il comportamento di un'astrazione dati può essere specificata riportando il valore di ciascuna osservazione applicata a ciascun costruttore. Questa informazione è organizzata in modo naturale in una matrice, con i costruttori lungo una dimensione e le osservazioni lungo l'altra.

Definiamo inoltre **osservazioni unarie** che osservano un singolo valore del dato astratto. Alcune volte è necessario creare delle osservazioni più complesse che osservano due istanze del dato astratto per confrontare due valori. In questo caso il valore dell'osservazione deve essere definito per tutte le combinazioni di costruttori possibili per i valori astratti che si devono confrontare.

In una osservazione, per definire costruttori si utilizza generalmente il criterio di minimalità, cioè l'insieme dei costruttori deve essere il più piccolo insieme di operatori necessario a costruire tutti i possibili valori per un certo dato astratto. Inoltre gli argomenti di un costruttore non devono essere tutti dello stesso sort del dato astratto.

5 Astrazione nella programmazione

L'applicazione dei principi di astrazione nella fase progettuale sarebbe poco efficace se poi nella fase di realizzazione non fosse possibile continuare a nascondere l'informazione. Pertanto è indispensabile che i linguaggi di programmazione, che costituiscono l principale strumento utilizzato in fase di realizzazione, supportino l'applicazione dei principi di astrazione al dine di agevolare lo sviluppo e la manutenzione di sistemi software sempre più complessi.

In linea di principio, ogni **linguaggio simbolico** offre di per se una forma di astrazione dalla particolare macchina. Il principio dell'astrazione applicato nel progetto dei linguaggi assemblativi fu quello di nascondere i dettagli di codifica di istruzioni e indirizzi di memoria. I **linguaggi ad alto livello** sono caratterizzati dalla possibilità di trattare particolari domini di valori indipendentemente dalla loro rappresentazione in memoria centrale, introducendo strutture di controllo astratte e dei nuovi operatori non previsti a livello di linguaggio macchina.

E' possibile costruire astrazioni su una qualunque classe sintattica, purchè le frasi di quella classe specifichino un qualche tipo di computazione. Possiamo avere diversi tipi di astrazione:

1. astrazione di funzione, che include un'espressione da valutare
2. astrazione di procedura, include un comando da eseguire
3. astrazione di controllo, include delle espressioni di controllo dell'orine di esecuzione delle istruzioni
4. astrazione di selettore, include il calcolo dell'accesso ad una variabile
5. astrazione de tipo, include un gruppo di operatori che definiscono implicitamente un insieme i valori
6. astrazione generica, include una frase che sarà elaborata per produrre legami.

Un'**astrazione di funzione** include un'espressione da valutare e che darà un valore come risultato quando viene chiamata. La definizione di funzione è del tipo:

$$function \quad I(FP_1; \ldots; FP_n) \quad is \quad E$$

Dove I è un identificatore, $FP_1; \ldots; FP_n$ sono parametri formali e E è l'espressione da valutare. In questo modo si lega I a un'entità, l'astrazione di funzione, che ha la proprietà di dare un risultato ogni qualvolta è chiamata con argomenti appropriati. Una chiamata di funzione ha due punti di vista:

- Punto di vista dell'**utente**: la chiamata i una funzioni trasforma gli argomenti in un risultato;

- Punto di vista dell'**implementatore**: la chiamata valuta E, avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti.

In alcuni linguaggi di programmazione, le funzioni sono di **terza classe**, cioè possono essere solo chiamate, in altri sono di **seconda classe**, cioè possono essere passate come argometi, mentre i altri linguaggi di programmazione esse sono di **prima classe** in quanto possono essere anche restituite come risultato della chiamata di altre funzioni o possono essere assegnate come valore a una variabile.

In generale, in programmazione una qualunque entità si dice **cittadino di prima classe** quando non è soggetta a restrizioni nel suo utilizzo. I valori possono essere **denotabili**, se possono essere associati ad un nome, **esprimibili**, se possono essere il risultato di un'espressione complessa, o **memorizzabili**, se possono essere memorizzati in una variabile.

Un'**astrazione di procedura** include un comando da eseguire e quando chiamata aggiornerà le variabili che rappresentano lo stato del sistema. E' definita mediante una definizione del tipo:

$$procedure \quad I(FP_1; \ldots; FP_n) \quad is \quad C$$

Dove I è un identificatore, $FP_1; \ldots; FP_n$ sono parametri formali e C è il blocco di comandi da eseguire. L'astrazione di procedura gode della proprietà di cambiare lo stato del sistema quando chiamata con argomenti appropriati. Dal punto di vista dell'utente, la chiamata aggiornerà lo stato del sistema in modo dipendente dai parametri, mentre il punto di vista dell'implementatore è che la chiamata consentirà l'esecuzione del corpo di procedura C, avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti.

L'astrazione di funzione e di procedura sono due tecniche di programmazione di supporto all'**astrazione funzionale**, intesa come tecnica di progettazione del software secondo la quale occorre distinguere la specifica d un operatore dalla sua realizzazione.

Quale sia la tecnica di programmazione più opportuna da adottare dipende sia dal tipo di operatore progettato, sia ai limiti imposti dal linguaggio di programmazione.

Le strutture di controllo definiscono l'ordine in cui le singole istruzioni o i gruppi di istruzioni devono essere eseguiti. A queste strutture di controllo si applica l'**astrazione di controllo**. Il linguaggio macchina fornisce due semplici meccanismi per gestire il flusso di controllo delle istruzioni: il salto e l'elaborazione in sequenza. Nei linguaggi di alto livello sono state introdotte altre strutture di controllo, quali la selezione, l'iterazione, la ricorsione e meccanismi per gestire situazioni eccezionali.

Possiamo specificare un'astrazione di controllo come segue:

$$control \quad I(FP_1; \ldots; FP_n) \quad is \quad S$$

Dove I è un identificatore, $FP_1; \ldots; FP_n$ sono parametri formali e S è un'espressione di controllo che definisce un ordine di esecuzione.

I linguaggi di programmazione sequenziale utilizzando la sequenza, la selezione e la ripartizione per definire un ordinamento totale sull'esecuzione dei comandi in un programma, mentre i linguaggi di programmazione parallela utilizzano altri costruttori del flusso di controllo come il fork, cobegin o cicli for paralleli, per introdurre in modo esplicito un ordinamento parziale sull'esecuzione dei comandi.

Si parla di **astrazione di selettore** quando un linguaggio di programmazione fornisce metodi per accedere alle variabili, strutturate e non. Possiamo definire un'astrazione di selettore come segue:

$$selector \quad I(FP_1; \ldots; FP_n) \quad is \quad A$$

dove A è un'espressione che restituisce un accesso a una variabile.

L'**espressione di tipo** è il costrutto con cui alcuni linguaggi di programmazione consentono di definire un nuovo tipo. Un'**astrazione di tipo** ha un corpo costituito da un'espressione di tipo. Quando è valutata, l'astrazione di tipo stabilisce sia una rappresentazione per un insieme di valori e ia le operazioni ad essi applicabili. L'astrazione di tipo può essere specificata come:

$$type \quad I(FP_1; \ldots; FP_n) \quad is \quad T$$

dove I è un identificatore del nuovo tipo, $FP_1; \ldots; FP_n$ sono i parametri formali, e T è un'espressione di tipo che specificherà la rappresentazione dei dati di tipo I e le operazioni a esso applicabili.

Nei linguaggi imperativi, i valori di un tipo astratto venono trattati alla stessa maniera dei tipi concreti, sono entrambi **cittadini di prima classe**. Al contrario i valori rappresentati mediante gli oggetti sono trattati come cittadini di terza classe i quanto una procedura non può restituire l'istanza di un generic package e non è possibile creare dinamicamente degli oggetti. I tipi astratt sono utili in tutti i paradigmi di programmazione, mentre gli oggetti, essendo variabili aggiornabili, si adattano solo a un paradigma di programmazione side-effecting.

Uno svantaggio dei tipi astratti sono la **scarsa estendibilità**, in quanto l'aggiunta di un nuovo costruttore comporta dei cambiamenti intrusivi nelle implementazioni esistenti degli operatori.

L'**astrazione generica** è un'astrazione su una dichiarazione, pertanto il corpo della dichiarazione di un'astrazione generica è a sua volta una dichiarazione. La chiamata di un'astrazione generica è detta **istanziazione** e produce dei legami elaborando la dichiarazione contenuta nel corpo dell'astrazione generica. L'astrazione generica può essere specificata come:

$$generic \quad I(FP_1; \ldots; FP_n) \quad is \quad D$$

dove D è una dichiarazione che quando elaborata produrrà dei legami. La dichiarazione può essere di diverso tipo: dichiarazione di un tipo, di un modulo, di una funzione, di una procedura. L'astrazione generica è d supporto all'astrazione dati, in quanto permette di definire delle classi che sono invarianti ad alcuni tipi di dati necessari per definirle. Mediante i parametri di tipo, l'astrazione è applicabile a tipi astratti che possono essere così ugualmente svincolati dalla necessità di specificare il tipo degli elementi sui quali operare. L'alternativa sarebbe quella di scrivere una definizione separata per ciascuno dei tipi, con conseguente codice duplicato, sforzo di programmazione ridondante e manutenzione complicata.

6 Il Paradigma OOP

Nella programmazione imperativa le variabili globali sono accessibili da ogni parte del programma. I grandi programmi che non presentano una disciplina di accesso alle variabili globali risultano ingestibili perchè nessun modulo che accede ad una variabile globale può essere sviluppato e compreso indipendentemente da altri moduli che accedono alla medesima variabile. Agli inizi degli anni '70 si iniziò ad utilizzare l'information hiding come soluzione a questo problema, incapsulando in un modulo le variabili globali e le operazioni per accedervi. Gli altri moduli possono accedere alle variabili globali utilizzando esclusivamente le operazioni per accedervi. Questo rappresenta un primo passo verso la definizione degli oggetti.

Nella programmazione imperativa è possibile definire oggetti, il cui utilizzo non è forzato ed è legato alla auto consapevolezza degli sviluppatori, inoltre non sono cittadini di prima classe. Possiamo quindi dire che il paradigma OO, rispetto al paradigma imperativo, costituisce un'evoluzione in quanto gli oggetti sono considerati cittadini i prima classe, e una rivoluzione, in quanto gli oggetti assumono un ruolo fondamentale nella progettazione e nella programmazione.

Gli **oggetti** incapsulano uno stato, identificato dal contenuto di una certa area di memoria, e un comportamento, definito da una collezione di procedure e funzioni che possono operare sulla rappresentazione dell'area di memoria associata all'oggetto. All'interno del progetto, gli oggetti hanno la finalità di modellare le entità presenti nel dominio dell'applicazione. Ogni oggetto è costituito da un identificatore di oggetto che lo identifica univocamente. Tale identificatore è immutabile.

Una **classe** è la descrizione di una famiglia di oggetti che condividono la stessa struttura e il medesimo comportamento. Nella programmazione Object Oriented, ogni oggetto è un'istanza di una classe, cioè un oggetto no può essere ottenuto se non si definisce la sua classe di appartenenza. Analogamente nella modellazione OO le istanze esistono in quanto ci sono le loro astrazioni.

I dettagli di realizzazione della classe sono normalmente nascosti. Ogni classe ha una doppia componente: una **componente statica**, ovvero i dati, costituita da campi o attributi dotati di nome, che contengono un valore e caratterizzano lo stato degli oggetti durante l'esecuzione del programma, e una **componente dinamica**, ovvero i metodi, che rappresentano il comportamento comune degli oggetti appartenenti alla classe.

In base all'ambito d'azione o **scope**, gli attributi si possono distinguere in:

- **Attributi d'istanza** che sono associati ad un'istanza e hanno un tempo di vita pari a quello dell'istanza alla quale sono associati
- **Attributi di classe** che sono associati alle classi e condivisi da tutte le istanze della classe.

I metodi si possono classificare come:

- **Metodi costruttori**, invocati per creare gli oggetti e inizializzarli,
- **Metodi di accesso**, che restituiscono astrazioni significative dello stato di un oggetto
- **Metodi di trasformazione**, che modificano lo stato di un oggetto,
- **Metodi distruttori**, invocati quando si rimuovono gli oggetti dalla memoria.

Un concetto fondamentale nel paradigma OO riguarda la visibilità di attributi e metodi, che soddisfa il concetto di incapsulamento citato precedentemente. Vi sono diversi livelli di **visibilità**:

- **pubblica**, quando i metodi e gli attributi sono visibili da altre classi;
- **privata**, quando i metodi e gli attributi sono visibili solo all'interno della classe di appartenenza;
- **protetta**, quando i metodi e gli attributi sono visibili all'interno del package i cui è dichiarata la classe e dalle discendenti di questa anche se si trovano in altri package;
- **package**, quando i metodi e gli attributi sono visibili sono agli elementi del package in cui la classe è definita.

Si definisce **molteplicità il classe** il numero di istanze che una classe può avere. Generalmente non si pone un limite, anche se in alcuni casi è necessario indicare che la classe può avere una sola istanza, e in questo caso viene definita **classe singoletto**, o comunque un numero ben definito di istanze.

In una buona modellazione Object Oriented di un sistema software è necessario stabilire le responsabilità da attribuire a ciascuna classe individuata. Al fine di rendere il sistema altamente riutilizzabile, il lavoro di individuazione delle classi deve essere condotto in modo preciso. Alcuni consigli pratici per raggiungere tale scopo sono:

- identificare gli elementi che gli utenti usano per descrivere il problema
- per ogni astrazione individuata è necessario identificare un insieme di responsabilità
- fornire ad ogni classe gli attributi e le operazioni di cui ha bisogno per eseguire tali responsabilità.

Definiamo la relazione **instance of**, definita come relazione tra un oggetto e una classe per indicare che l'oggetto è istanza di quella classe. La relazione fondamentale tra le classi è la **relazione di ereditarietà**. Partiamo dal presupposto che ogni classe è un insieme di conoscenze a partire dal quale si può definire classi più specifiche, che completano le conoscenze della class madre. Una sottoclasse è dunque una specializzazione di una classe detta super classe, che eredita tutte le informazioni e le amplia, evitando in questo modo la duplicazione di codice. Esistono diverse forme di ereditarietà.

Nell'**ereditarietà per estensione** la sottoclasse introduce delle caratteristiche non presenti nella super classe e non applicabili a istanze della super classe. La visibilità degli attributi e delle operazioni ereditate dalla super classe non è modificata. La specializzazione per estensione permette di sviluppare del codice estendibile. Quando si dovranno aggiungere ulteriori funzioni, occorrerà individuare le classi interessate e derivare da queste nuove classi alle quali verranno aggiunti gli attributi e i metodi necessari per implementare le nuove funzioni.

Nell'**ereditarietà per variazione funzionale** si ridefiniscono alcune caratteristiche della super classe quando quelle ereditate si rivelano inadeguate per l'insieme di oggetti descritti dalla sottoclasse. La ridefinizione del metodo ereditato (**overriding**) riguarda solo l'implementazione e non la segnatura. Ogni richiesta di esecuzione del metodo ridefinito da parte di un oggetto della sotto classe, farà riferimento alla nuova implementazione fornita nella sottoclasse.

La ridefinizione non è incrementale, quindi i cambiamenti nel metodo originale devono essere riportati anche nei metodi ridefiniti. Purtroppo non c'è alcuna garanzia che questo accada e si possono introdurre degli errori. Molti esperti vedono nella ridefinizione una potenziale fonte di errori e ne sconsigliano l'uso. I metodi dovrebbero essere ereditati completamente senza sovrascrittura, altrimenti non andrebbero specificati affatto. Per mitigare gli effetti di questo problema, si può adottare qualche accorgimento nella realizzazione dei metodi per i quali si riconosce già in fase di progetto una incrementalità al cambiamento. Inoltre nell'ereditarietà per variazione funzionale, la visibilità degli attributi e delle operazioni ereditate dalla super classe non è modificata. Inoltre la variazione funzionale attiene solo le operazioni di accesso e trasformazione di una classe e non i costruttori degli oggetti.

Nell'**ereditarietà per restrizione** le istanza di una sottoclasse soddisfano vincoli che non sono necessariamente soddisfatti da istanze della super classe. L'ereditarietà per restrizione non chiede la modifica della visibilità degli attributi e delle segnature delle operazioni ereditate dalla super classe.

Un principio fondamentale del paradigma OOP è il **principio di sostituibilità**, il quale definisce che data una dichiarazione di una variabile o di un parametro il cui tipo è dichiarato come X, una qualunque istanza di una classe che è discendente di X può essere usato come valore effettivo senza violare la semantica della dichiarazione e il suo uso.

Ciò significa che l'istanza di un discendente può essere sostituita all'istanza di un ascendente. Questo principio è fortemente legato al polimorfismo di inclusione, nella programmazione orientata a oggetti. La conseguenza del principio di sostituibilità è che una sottoclasse non può rimuovere o rinunciare a proprietà/metodi della super classe. Altrimenti un'istanza della sottoclasse non sarà sostituibile in una situazione in cui si dichiara l'uso di istanze della super classe. In questo modo, preservando la visibilità degli attribuiti e dei metodi ereditati, si garantisce che gli oggetti della super classe. Quindi il principio di sostituibilità è compatibile con l'ereditarietà per estensione, variazione funzionale e restrizione.

Nelle varie forme di ereditarietà (estensione, funzionale e restrizione) la relazione di ereditarietà fra classi

corrisponde a una **relazione di generalizzazione** (o "is-a"). Ogni istanza di una classe derivata da una classe base va considerata come un'istanza della classe base.

Nell'**ereditarietà di implementazione** la sotto classe utilizza il codice della super classe per implementare l'astrazione associata. In questo tipo di implementazione comporta la modifica alla visibilità delle caratteristiche ereditate, inoltre non è compatibile con il principio di sostituibilità. Quindi se la classe Y eredita solo l'implementazione della classe X, non si può riutilizzare su istanze di Y tutto il codice in cui si dichiarano e utilizzano dati di classe X, in quanto no vale il polimorfismo di inclusione. Però questo tipo di implementazione permette un riuso parziale del codice.

Nella progettazione di una classe si possono combinare diverse forme di ereditarietà. La rappresentazione della relazione di generalizzazione fra un insieme di classi definisce un grafo di ereditarietà che è un grafo orientato aciclico, inoltre la realizzazione di generalizzazione è transitiva e antisimmetrica. La transitività comporta che le caratteristiche delle classi superiori sono ereditate dalle classi inferiori. Invece l'antisimmetria definisce una direzione di attraversamento del grafo di ereditarietà che porta dalla sottoclasse alla super classe.

Nell'**ereditarietà singola** il grado è un albero in cui ogni classe ha una sola super classe diretta. In questo caso il grafo di ereditarietà di una classe C è una catena di antenati. Gli elementi della catena sono ordinati secondo una relazione d'ordine totale. Mentre nell'**ereditarietà multipla** una classe può avere più classi. In questo caso il grafo di ereditarietà non è più un albero, ma un grafo aciclico orientato, inoltre l'ordine fra le classi è parziale. Nel definire che A deriva dalle due super classi occorrerà elencale in un qualche ordine. La relazione di ereditarietà introduce un ulteriore livello di visibilità: quella protetta. Una caratteristica con visibilità protetta può essere vista solo da una classe e da tutte le classi del package e delle classi discendenti.

Se in programmazione no può esister un oggetto senza che sia stata creata la classe di appartenenza, è invece possibile che esistano classi per le quali non è possibile generare delle istanze definite **classi astratte**. Una classe astratta può essere una classe non completamente specificata. In particolare, non è definito il metodo corrispondente a un'operazione, si parla di metodo astratto. Una classe astratta non può essere istanziata perchè il comportamento dei suoi oggetti non sarebbe completamente definito. La relazione di ereditarietà riguarda anche le classi astratte. Le classi astratte sono strumenti per fattorizzare proprietà comuni tra classi simili e poterle organizzare in una gerarchia di ereditarietà. Non potremo mai creare oggetti a partire da una classe astratta, ma possiamo servircene per dare una radice comune a un insieme i classi che condividono le stesse proprietà e poter quini sfruttare il polimorfismo di inclusione il binding dinamico. L classi astratte fungono da serbatoi di ereditarietà.

Una classe è definita **classe finale** o **classe foglia**, quando non può essere ulteriormente specializzata, e quindi non può essere modificata. Si definisce quna classe foglia quando il comportamento della classe deve essere ben stabilito per ragioni di affidabilità. La dichiarazione di una classe foglia permette anche la generazione di codice ottimizzato in quanto facilita l'espansione in linea del codice.

Si definisce **interfaccia** la descrizione del comportamento degli oggetti senza specificarne una implementazione. Essa è una collezione di operazioni, cioè di servizi che possono essere richiesti, priva di informazioni sulle implementazioni dei servizi. Similmente a una classe, un'interfaccia può avere un qualsiasi numero di operazioni. Al contempo non specifica una struttura e non fornisce un'implementazione. Un'interfaccia quindi assomiglia a una classe astratta i cui metodi sono tutti astratti e non dispone di attributi. Una o più classi possono implementare/realizzare le operazioni contenute in un interfaccia. La relazione che si stabilisce tra un'interfaccia e una classe che la implementa è detta relazione di realizzazione. Le interfacce servono a disaccoppiare al definizione delle operazioni dalla loro implementazione. Per poter usare uncerto oggetto è sufficiente conoscere la sua interfaccia, non serve conoscere l'implementazione. Anche le interfacce possono ereditare da altre interfacce attraverso una relazione di generalizzazione "is-a". Inoltre, poichè le interfacce non implementano le operazioni, l'ereditarietà multipla non pone problemi di conflitto. Inoltre per evitare problemi di conflitto, è permesso ad una classe di implementare più interfacce, che non siano però correlate da una relazione di generalizzazione. Più classi possono implementare la stessa interfaccia.

L'ereditarietà offre molti vantaggi, ma non tutti gli oggetti si ottengono bene derivandoli da altri oggetti. Spesso un oggetto è ottenuto aggregando altri oggetti. L'ereditarietà di implementazione permette il riuso della sola implementazione ma non dei comportamenti. Una composizione di oggetti può essere rappresentata permettendo alle variabili di istanza di una classe di puntare a oggetti di altre classi, in quanto lo stato di un oggetto può anche contenere il riferimento ad un altro oggetto. Si possono stabilire legami con più istanze di una classe che descrive un componente. La relazione che si stabilisce in questo modo fra le classi è detta **aggregazione** o **composizione**. Essa è quindi un'altra possibile relazioni fra le classi, diversa dall'ereditarietà. Una classe A si dice essere in relazione di aggregazione con una classe B quando alcune istanze di B contribuiscono a formare una parte delle istanze di A. Quindi l'aggregazione è una relazione asimmetrica. La molteplicità consente di indicare quanti oggetti possono essere aggregati. L'aggregazione può essere utilizzata in diverse situazioni, quali:

- Contenimento fisico
- Appartenenza
- Composizione funzionale

Le aggregazioni sono associazioni deboli fra parti e intero. Questo significa che le parti possono esistere senza l'intero. Un'associazione forte fra parti e intero è detta **composizione** e comporta una dipendenza esistenziale, in quanto le parti non esistono senza il contenitore. Quindi questo implica che la creazione e la distruzione delle parti avvengano nel costruttore, e che i componenti non siano parti di altri oggetti. La *regola di non condivisione* consiste nel fatto che una classe può essere componente di molte altre classi, ma ogni sua istanza può essere componente di un solo oggetto.

In molti casi è possibile associare due classi mediante ereditarietà o aggregazione/composizione e la scelta no è immediata. Nell'ereditarietà ogni oggetto della classe contiene tutti i campi definiti nelle classi tipo di quel-

l'oggetto e può accedervi direttamente. Nell'aggregazione/composizione, ogni oggetto o della classe contiene due campi di tipo di un altro oggetto e per accedere all'informazione sul giorno occorrerà invocare un metodo sull'oggetto D. Se l'ereditarietà corrisponde a una relazione di generalizzazione (is-a) vale il polimorfismo di inclusione, mentre questo non è possibile nel caso della composizione. Il meccanismo di aggregazione/composizione è generalmente usato quando si vogliono utilizzare i servizi di una classe predefinita ma non la sua interfaccia. L'ereditarietà di implementazione, qualora non dovesse essere permessa da un linguaggio di programmazione, potrebbe essere resa da una relazione di aggregazione/composizione.

La mole di classi aumenta in modo considerevole all'aumentare della complessità del sistema da modellare. E' importante organizzare tali classi in gruppi separati al fine di rendere più facilmente individuabili e accessibili le varie parti del sistema. Questo meccanismo prende il nome di **package**. Un package definisce un namespace per i suoi elementi. Questo significa che ogni classe di un package deve are un nome distinto all'interno del package che la racchiude. Il nome completo della classe sarà ottenuto indicando il nome del package che la contiene mediante una notazione ::. In generale è possibile avee nomi uguali per elementi della stesa specie se sono inseriti in package differenti, non è possibile avere all'interno di uno stesso package due elementi della stessa specie con lo stesso nome. I package possono essere innestati senza alcun limite di profondità.

Si può specificare la visibilità degli elementi di un package:

- Public (+) sono visibili ad altri elementi del package stesso, a uno dei package innestati o a package che li importano.
- Private non sono visibili all'esterno del package.
- Protected rende l'elemento visibile solo alle altre classi che appartengono allo stesso package

Una classe interna o inner class è una classe la cui dichiarazione si trova all'interno di un'altra classe ospite. Le classi interne sono identificate da un nome al pari delle classi top level. Una inner class può essere privata e in tal caso no è visibile all'esterno della classe ospite. La inner class può accedere a tutti i metodi e i campi della classe ospitante, mentre la classe ospitante può vedere solo la parte pubblica della inner class. Un oggetto della inner class non può esistere se non esiste un oggetto della classe ospitante.

Con il termine polimorfismo si vuole indicare la possibilità di associare a una operazione diverse realizzazioni. Vi sono principalmente due tipi di polimorfismo:

- polimorfismo parametrico, ottenuto quando una funzione lavora uniformemente su una gamma di tipi. Questi tipi normalmente esibiscono una Qualche Struttura comune
- polimorfismo ad hoc, ottenuto quando una funzione lavora su tipi differenti e potrebbe comportarsi in modo scorrelato per ciascuno di essi.

Si possono definire altri tipi di polimorfismo, ad esempio quello per inclusione al fine di modellare i concetti di sottotipo e di ereditarietà, e il polimorfismo universale che contrasta il polimorfismo ad hoc.

La coercizione è il meccanismo di conversione implicita operata da un compilatore per applicare un operatore definito per gli oggetti di tipo T1 anche a oggetti di tipo T2. Le coercizioni possono essere stabilite staticamente, inserendole automaticamente fra gli argomenti e le funzioni al momento della compilazione, oppure potrebbero essere determinate dinamicamente da test al run-time sugli argomenti. La coercizione è la forma di polimorfismo più semplice.

Si ha il polimorfismo per overloading quando si usa lo stesso identificatore per metodi differenti e si ricorre a informazione di contesto per decidere quale metdo è denotato da una particolare occorrenza dell'identificatore. La disambiguazione necessaria per una coretta compilazione si basa sul tipo degli argomenti del metodo o sulla classe dell'oggetto a cui si richiede il servizio. Nel paradigma a oggetti si ha overloading anche nel caso di funzioni con medesimo nome ma definite in classi non correlate gerarchicamente.

Nel polimorfismo parametrico, una funzione polimorfa ha un parametro di tipo esplicito o implicito che determina il tipo dell'argomento per ciascuna applicazione della funzione. Le funzioni che esibiscono il polimorfismo parametrico sono anche dette funzioni generiche, che lavora su argomenti di molti tipi, generalmente esibendo lo stesso comportamento.