

APPUNTI DI PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Rel. 1.0.1 b





INDICE GENERALE

CAPITOLO 1

1.0 INTRODUZIONE

1.1 PROCESSO DI PROGETTAZIONE DEL SOFTWARE

1.2 VERIFICA E CONVALIDA

1.3 PROVE DI INTEGRAZIONE

1.4 LA MANUTENZIONE DEL SOFTWARE

1.5 CHE COS'E' UN LINGUAGGIO DI PROGRAMMAZIONE ORIENTATO AGLI OGGETTI (O.O.P.).

CAPITOLO 2

2.0 PREMESSA

2.1 ASTRAZIONE DEI DATI

2.2 L'INCAPSULAMENTO O INFORMATION HIDING

2.3 IL POLIMORFISMO

2.4 EREDITARIETA'

2.5 ASSOCIAZIONE DINAMICA

2.6 VANTAGGI DELLA PROGRAMMAZIONE AD OGGETTI



CAPITOLO 3

3.0 PREMESSA

3.1 CREARE LE CLASSI

3.2 CONTROLLO DELL'ACCESSO ALLA CLASSE

3.3 COMPORTAMENTO DEI MEMBRI PROTECTED (EREDITARIETA' MULTIPLA)

3.4 EREDITARIETA' DA PIU' CLASSI

3.5 COSTRUTTORI E DISTRUTTORI

3.6 FUNZIONI FRIEND

3.7 CLASSI FRIEND

CAPITOLO 4

4.0 PREMESSA

4.1 FUNZIONI GENERICHE

4.2 UNA FUNZIONE TEMPLATE CON DUE O PIU' TIPI GENERICI

4.3 CLASSI GENERICHE

4.4 LA POTENZA DEI TEMPLATE

RINGRAZIAMENTI

BIBLIOGRAFIA



NOTA DELL' AUTORE

Questa documentazione vuole essere una semplice guida per chi si inoltra, per la prima volta, nel mondo della programmazione orientata agli oggetti (O.O.P.).

Questa documentazione è stata creata prendendo il meglio degli argomenti contenuti in vari libri di testo universitari scritti in varie lingue. L'opera dell'autore sta nell'aver tradotto alcuni paragrafi e nell'aver cercato di armonizzare tra loro questi argomenti utilizzando un linguaggio alla portata di tutti (la dove è stato possibile), senza alterarne l'importanza ed il loro significato.

Con la presente nota si vuole mettere in evidenza il fatto che gli argomenti trattati non sono esposti nella loro completezza didattica, in quanto, per la totale comprensione, si richiederebbe un livello formativo molto tecnico¹.

Questa documentazione non vuole essere considerata come una documentazione esaustiva. Si rimanda alla nota bibliografica, per chi volesse saperne di più.

Ing. Giovanni Iozzelli

¹ Questa documentazione sarà soggetta a future revisioni.



Copia per uso personale

DITEMI QUELLO CHE PENSATE

Come lettori di questa documentazione, siete i più importanti critici e commentatori.

Inviatemi la vostra opinione attraverso una e-mail². Inoltre, mi piacerebbe conoscere se la documentazione vi è utile così com'è o cosa dovrei aggiungere per migliorarla.

Come autore di questa dispensa vi assicuro che ogni sorta di commento è ben gradito.

Mi scuso anticipatamente se non sarò in grado di rispondere a tutti.

Grazie

² Per inviare una e-mail scrivete a: g.iozzelli@tidestudio.com



Copia per uso personale



CAPITOLO 1

1.0 INTRODUZIONE

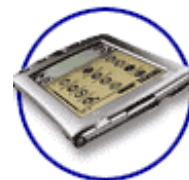
1.1 PROCESSO DI PROGETTAZIONE DEL SOFTWARE

1.2 VERIFICA E CONVALIDA

1.3 PROVE DI INTEGRAZIONE

1.4 LA MANUTENZIONE DEL SOFTWARE

1.5 CHE COS'E' UN LINGUAGGIO DI PROGRAMMAZIONE ORIENTATO AGLI OGGETTI (O.O.P.).





37993.4441



Copia per uso personale

1.0 INTRODUZIONE

Il 1960 ha dato i natali alla programmazione strutturata. L'impiego dei linguaggi strutturati ha reso possibile la realizzazione di programmi piuttosto complessi con una discreta facilità. I linguaggi strutturati sono caratterizzati dal supporto di subroutine o funzioni indipendenti, di variabili locali e costrutti di controllo avanzati e dal fatto di non impiegare l'istruzione GOTO.

Tuttavia, anche utilizzando metodi di programmazione strutturata, un progetto può diventare incontrollabile una volta che raggiunga determinati dimensioni.

Si consideri questo fatto: ad ogni punto di svolta nel campo della programmazione, sono stati creati strumenti e tecniche che consentivano al programmatore di realizzare programmi più complessi. Ogni passo in questo percorso consisteva nell'utilizzo dei migliori elementi dei metodi precedenti e nel loro sviluppo.

Prima dell'invenzione della programmazione orientata agli oggetti (O.O.P.), molti progetti raggiungevano o superavano il punto in cui l'approccio strutturato non poteva essere più adottato. La programmazione ad oggetti è nata con lo scopo di superare questa barriera.

La programmazione ad oggetti ha preso le migliori idee della programmazione strutturata e le ha combinate con nuovi concetti. Il risultato è un'organizzazione completamente nuova dei programmi.

In genere un programma può essere realizzato in due modi: ponendo al centro dell'analisi "il codice" (cioè quello che accade) o ponendo al centro dell'analisi "i dati" (gli elementi che si utilizzeranno per effettuare le elaborazioni all'interno del programma).

Utilizzando le tecniche della programmazione strutturata, i programmi vengono tipicamente organizzati attorno al codice. I programmi ad oggetti seguono un altro approccio: questi sono organizzati attorno ai dati e si basano sul fatto che sono i dati a controllare l'accesso al codice. In un linguaggio ad oggetti si definiscono i dati e le routine che sono autorizzate ad agire su tali dati. Pertanto sono i dati a stabilire quali sono le operazioni che possono essere eseguite.

I linguaggi che consentono di attuare i principi della programmazione orientata agli oggetti hanno tre fattori in comuni: l'incapsulamento, il polimorfismo e l'ereditarietà³.

³ Il loro significato sarà spiegato dettagliatamente più avanti.



37993.4441



Copia per uso personale

1.1 PROCESSO DI PROGETTAZIONE DEL SOFTWARE

Per comprendere le innovazioni introdotte dalla programmazione orientata agli oggetti, iniziamo con il descrivere le varie fasi che caratterizzano lo sviluppo di un software.

Il processo attuale di sviluppo del software può essere suddiviso in un certo numero di fasi:

- *Analisi delle richieste del cliente;*
- *Specifiche delle richieste;*
- *Progettazione del sistema;*
- *Progettazione dettagliata*
- *Programmazione.*

Ciascuna di queste fasi corrisponde ad un'attività distinta che si conclude con la generazione di uno o più documenti che costituiscono la base di lavoro della fase successiva.

Analisi delle richieste del cliente.

In generale, il cliente fornisce un documento nel quale descrive le sue aspettative. La qualità dei contenuti di questo documento varia considerevolmente: i clienti meno esperti di tecnologie informatiche, di specifiche e di sistemi software, produrranno documenti molto succinti e astratti che, probabilmente, potranno essere utilizzati per acquisire suggerimenti generici sulle loro richieste. I clienti più esperti, invece, potrebbero fornire dei documenti voluminosi che descrivono dettagliatamente i vari aspetti del sistema da sviluppare.

Specifiche delle richieste.

Il documento con le richieste del cliente viene inviato al progettista del software che, poi, procederà all'analisi di tali richieste. Per svolgere questo compito, gli analisti del gruppo di progettazione studiano i sistemi esistenti che dovranno essere sostituiti dal sistema proposto, intervistando il cliente che ha prodotto il documento delle richieste, e avviano "l'analisi tecnica" delle richieste.

Quando il processo di analisi delle richieste del cliente è ultimato, gli analisti producono un documento per descrivere le caratteristiche del sistema da sviluppare.



37993.4441



Copia per uso personale

Questo processo costituisce la fase nota come “specificazione delle richieste” e il documento risultante è chiamato “*specificazione delle richieste*”.

Il documento contiene una descrizione dettagliata di tutti i compiti che il nuovo sistema dovrà svolgere, di eventuali limitazioni (quali l'utilizzazione della memoria o i tempi di risposta).

La specificazione delle richieste è un documento chiave nella progettazione del software per varie ragioni. In primo luogo, questo documento rappresenta l'input principale per la successiva fase del processo di sviluppo: la progettazione del sistema. In secondo luogo, esso è utilizzato nella sua versione definitiva o, più realisticamente, in quella provvisoria dal project manager per stimare le risorse richieste dal progetto. In terzo luogo, tale documento serve al gruppo di controllo della qualità per sviluppare una serie di test che potranno convalidare il corretto funzionamento del sistema. In fine, il documento in esame viene consultato dalle persone che dovranno realizzare i test o i manuali descrittivi del sistema.

Progettazione del sistema.

Questa fase consiste nel definire l'insieme delle soluzioni tecniche (architettura del sistema) che realizzano le funzioni contenute nella specificazione delle richieste, rispettando le eventuali limitazioni specificate in tale documento. L'architettura del sistema è espressa in termini di gruppi indipendenti di istruzioni di programma chiamati “moduli”.

L'architettura del sistema può essere definita utilizzando una serie di notazioni. Probabilmente, la classe di notazioni più comunemente adottata è quella grafica denominata flow-chart.

Progettazione dettagliata.

Durante la “progettazione del sistema”, le persone responsabili dello sviluppo di questa fase hanno descritto le operazioni associate ai vari moduli del sistema. Il compito dei “progettisti dei dettagli” è quello di esaminare questa descrizione e definire tutte le operazioni dei singoli moduli. In pratica, questo lavoro consiste nel preparare un'apposita notazione per i moduli, come flow-chart o le istruzioni in un linguaggio di programmazione.

La fase di progettazione dettagliata è facoltativa. Molti sviluppatori di software preferiscono ometterla e programmare direttamente dalla progettazione del sistema.

La progettazione dettagliata è utilizzata soprattutto dagli sviluppatori che sono interessati alla versatilità, in particolare se essi intendono realizzare più versioni di un sistema che impiega vari linguaggi di programmazione.



La progettazione dettagliata dovrebbe essere una descrizione indipendente dal linguaggio di programmazione delle funzioni del sistema: questa descrizione deve essere facilmente traducibile in qualsiasi linguaggio di programmazione, sia in assembler sia in un linguaggio ad alto livello.

Programmazione.

In questa fase il programmatore esamina le descrizioni dei moduli estratte dalla progettazione dettagliata (se è stato prodotto il relativo documento) o dalla specifica del sistema.

Se la fase di progettazione dettagliata è stata svolta, il processo di programmazione è molto semplice: tutto ciò che occorre fare è tradurre le specifiche dei moduli nelle istruzioni del linguaggio di programmazione che si vuole adottare. D'altra parte in assenza della progettazione progettazione dettagliata, la programmazione diventa un compito intellettuale più impegnativo che richiede la scelta di un algoritmo che svolga le operazioni previste nella progettazione del sistema.

In questo modo, si conclude il processo di sviluppo di un progetto software.. Parallelamente alle fasi precedentemente descritte, si svolge un'applicazione continua di una serie di "test" per verificare che i progettisti stiano sviluppando correttamente le funzioni del software e che il sistema in esame soddisfi le richieste del cliente.

Queste attività sono globalmente chiamate "*verifica e convalida*" del sistema.



37993.4441



Copia per uso personale

1.2 VERIFICA E CONVALIDA

L'attività di "verifica" consistono nel controllare che un determinato compito sia stato svolto correttamente (per esempio, verificare che un programmatore abbia implementato correttamente la progettazione dettagliata del software).

Per l'attività di "convalida" s'intende una serie di controlli che verificano se il sistema soddisfa le richieste del cliente (per esempio, provare il sistema: eseguire il software per controllare che esso soddisfi le richieste del cliente).

Durante il processo di sviluppo del software si svolgono varie attività di verifica e di convalida.

Durante "l'analisi delle richieste" e "le specifica delle richieste", si svolgono due attività di convalida e una viene soltanto avviata.

Le prime due attività sono:

- *la revisione delle specifiche delle richieste*
- *realizzazione di un sistema prototipo*

la terza attività è:

- *la preparazione dei test* del sistema e dei test di accettazione

La revisione delle specifiche delle richieste

E' svolta da un certo numero di persone che spendono un po' di tempo (non più di due ore) nella lettura di un documento o di un segmento di programma per verificarne la correttezza. La revisione è un processo molto efficace: le persone che partecipano a questo processo tendono a concentrarsi su un problema e sono in grado di controllare un documento o le istruzioni di un programma con una maggiore accuratezza. Molti programmatori, analisti e progettisti trovano difficile distaccarsi da quanto hanno prodotto, pertanto non riescono a scoprire tutti i loro errori. Il processo di revisione consente di superare questo problema, in quanto le persone che lo attuano sono libere di vagliare senza alcun pregiudizio un particolare documento o prodotto software.

La revisione è un processo di rilevazione degli errori, non di correzione; di conseguenza, c'è poco da discutere sulle azioni da svolgere per migliorare la specifica. Il risultato del lavoro di revisione è "*l'elenco degli errori*" che sono stati scoperti e che necessitano di un successivo esame.



37993.4441



Copia per uso personale

Realizzazione di un sistema prototipo.

Questa attività consiste nella produzione di una prima versione del sistema che può essere utilizzata come strumento di addestramento sia del progettista che del cliente.

La realizzazione del prototipo è un processo molto efficiente per svolgere l'analisi delle richieste. Le specifiche dei sistemi sono diventate così complesse ed estese che i clienti e gli sviluppatori incontrano più difficoltà a comprenderle. Un prototipo rappresenta la manifestazione più concreta di un sistema software che è possibile realizzare e che fornisce al cliente un'idea esatta e completa di ciò che riceverà.

Preparazione dei test.

Quest'attività si svolge verso la fine della specifica delle richieste. Consiste nella preparazione dei test del sistema e dei test di accettazione. Questi test costituiscono i controlli finali su un sistema per verificare se esso soddisfa le richieste del cliente.

I test vengono effettuati alla fine del processo di sviluppo di un sistema, dopo che tutti i moduli corretti sono stati riuniti. I test di accettazione vengono svolti nello stesso ambiente in cui il sistema sarà installato; ad ogni test assiste il cliente o un suo rappresentante. Questi test sono cruciali in quanto, se il sistema non supera una prova di accettazione, il cliente ha il diritto di rifiutare il sistema, e, quindi, di pagarlo. Inoltre, il fallimento di una prova di accettazione spesso richiede un ulteriore lavoro per ridefinire il sistema e modificare le specifiche. Poiché i test di accettazione sono così importanti, gli sviluppatori di software effettuano una serie di prove preliminari nel loro posto di lavoro.



1.3 PROVE DI INTEGRAZIONE

Un'altra importante attività, che viene avviata durante la progettazione del sistema, è la definizione delle *"prove di integrazione"*.

Nelle fasi finali del processo di progettazione, gli sviluppatori stabiliscono una strategia di prove di integrazione: il sistema viene assemblato utilizzando pochi moduli per volta; dopo aver aggiunto nuovi moduli, effettuano una serie di test per verificare che le interfacce tra i moduli integrati e il sistema che stanno realizzando siano correttamente implementate.

I progettisti devono prendere alcune importanti decisioni sul processo di integrazione e sulle relative prove. Innanzi tutto, dovranno stabilire la strategia generale da adottare.

Esistono tre tipi fondamentali di integrazione:

- integrazione top down;
- integrazione bottom up;
- integrazione inside out.

Ogni tipo ha i suoi vantaggi e svantaggi.

Ad esempio, l'integrazione top-down consente di realizzare un'immediata, anche se parziale, versione del sistema, ma non è molto efficace per rilevare errori nei moduli finali del sistema.

Un'altra decisione riguarda il livello di dettaglio del processo di integrazione: il numero di moduli che devono essere di volta in volta aggiunti a quelli esistenti. I progettisti potrebbero decidere di aggiungere un numero costante di moduli (per esempio, due moduli alla volta) o un numero di variabile di moduli (per esempio, potrebbero integrare un solo modulo se questo è molto complesso o venti moduli contemporaneamente se questi sono semplici).

In generale, una buona norma da seguire è: *se è possibile controllare qualcosa durante le fasi di sviluppo del progetto, ciò deve essere fatto.*

In tal modo sarà possibile risparmiare i costi extra di revisione e di riprogettazione del sistema che, necessariamente, dovranno essere sostenuti se i controlli vengono effettuati in ritardo.



1.4 LA MANUTENZIONE DEL SOFTWARE

Nei precedenti parafi sono state descritte le operazioni svolte da un gruppo ben organizzato di progettisti per produrre un sistema software. Esistono, tuttavia, ancora alcuni problemi da esaminare: quelli legati alla *manutenzione* del software.

Per manutenzione del software si intende il processo di modifica di un sistema durante il suo funzionamento.

Sulla manutenzione permangono due concetti errati:

- il primo è che essa consiste soltanto nel correggere gli errori commessi in fase di sviluppo del software;
- il secondo è di non ritenere la manutenzione del software un'attività importante. Le attività di manutenzione possono raggiungere fino al 70-75% dei costi di sviluppo del software.

Le operazioni di manutenzione possono essere suddivise in tre distinte categorie:

- *manutenzione correttiva*;
- *manutenzione migliorativa*;
- *manutenzione di adattamento*

Manutenzione correttiva.

La manutenzione correttiva corrisponde al concetto più comune del processo di manutenzione: identificare e correggere gli errori commessi durante lo sviluppo del software

Manutenzione migliorativa.

E' il processo che tende a perfezionare in qualche misura il sistema software, per esempio sostituendo un algoritmo con uno più efficace, o modificando l'hardware in modo che il software possa essere eseguito da un computer più potente.

Manutenzione di adattamento.

E' il processo di modifica del sistema in seguito alle variazioni delle richieste del cliente. Queste richieste potrebbero cambiare per vari motivi.



Dato un sistema più o meno complesso, quando si applicano delle modifiche queste si possono ripercuotere indirettamente anche su parti diverse del sistema, complicando e allungando le operazioni di aggiornamento. La manifestazione più evidente di questo problema è un fenomeno di modifiche a catena.

Questo fenomeno, noto come *system ripple*, si verifica quando viene apportata una modifica ad un sistema dalla modesta architettura. La modifica di un modulo comporta la modifica dei moduli ad esso correlati; questi moduli, a loro volta, determinano la modifica dei moduli ad essi correlati; queste modifiche richiedono ulteriori cambiamenti in altri moduli, e così via. L'effetto più deleterio del *system ripple* consiste nel fatto che anche una piccola modifica può richiedere una considerevole quantità di risorse.

A tal punto sorge un'esigenza di trovare un sistema di programmazione che agevoli le operazioni di manutenzione, che consenta il riutilizzo anche di codici sorgenti complessi scritti per altri sistemi.

La soluzione a queste esigenze i programmatori l' hanno trovata nelle tecniche di sviluppo e nei linguaggi di programmazione orientati agli oggetti.

Con queste tecniche sarà possibile creare delle librerie le cui componenti richiederanno un modestissimo lavoro di adattamento per essere riutilizzate in un nuovo sistema.



1.5 CHE COS'E' UN LINGUAGGIO DI PROGRAMMAZIONE ORIENTATO AGLI OGGETTI (O.O.P.).

I linguaggi di programmazione orientati agli oggetti sono linguaggi che consentono di implementare degli “oggetti”, assicurando che l’implementazione stessa possa essere nascosta al programmatore permettendo, così, di raggiungere un elevato livello di reimpiego del codice sorgente.

Questi linguaggi permettono ad un programmatore di produrre un’entità nota come “*classe*”, che descrive le strutture dei dati e dei moduli.

Da ciò si evince che la prima caratteristica di un linguaggio di programmazione ad oggetti è quella di disporre di uno strumento per mezzo del quale limitare l’accesso ad una struttura di dati. Tale caratteristica è nota come “*information hiding*”⁴ o *incapsulamento*.

Un’altra caratteristica di un linguaggio ad oggetti è il “polimorfismo” o, come a volte viene chiamata, *generalità*. Questa caratteristica consente al progettista del software di definire i dati e le operazioni che implementano un oggetto attraverso il meccanismo della classe, e di predisporre la classe per una varietà di componenti nella parte dei dati della classe.

Il polimorfismo permette ad un programmatore di creare una libreria di oggetti riutilizzabili e contribuisce enormemente allo sviluppo di un software capace di essere reimpiegato in altri sistemi.

Un’altra caratteristica dei linguaggi ad oggetti è “*l’ereditarietà*”. Con questo termine si intende la possibilità per il programmatore di definire una nuova classe che erediti molte delle caratteristiche di una classe esistente.

Esiste un’ampia gamma di linguaggi che potrebbero essere classificati come linguaggi di programmazione orientata agli oggetti, a partire dal linguaggio ADA che dispone di un modesto numero di caratteristiche fino al linguaggio SMALLTALK che ha come sua caratteristica di base l’oggetto e dove ogni altra caratteristica del linguaggio è integrata con il concetto di oggetto, sino ad arrivare al linguaggio oggi più affermato il C++.

⁴ Le istruzioni di una struttura di dati sono nascoste dietro una serie di operazioni che permettono di accedere ai dati solamente se si specificano appositi parametri.



Copia per uso personale

L'ingegneria del software ha una storia di appena 35 anni. Nonostante ciò, essa è durata abbastanza per indicarci che nello sviluppo del software non si verificano cambiamenti radicali, ma che i cambiamenti, mitigati dal fatto che esiste un'enorme mole di manutenzione del software, sono processi evolutivi che avvengono per gradi sulla base delle tecnologie, linguaggi di programmazione e tecniche di sviluppo esistenti.

**Caratteristiche
fondamentali di un
linguaggio O.O.P.**

l'astrazione dei dati;
incapsulamento o information hiding;
il polimorfismo;
l'ereditarietà;
l'associazione dinamica



Copia per uso personale



CAPITOLO 2

2.0 PREMESSA

2.1 ASTRAZIONE DEI DATI

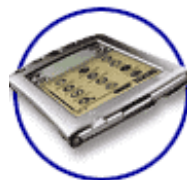
2.2 L'INCAPSULAMENTO O INFORMATION HIDING

2.3 IL POLIMORFISMO

2.4 EREDITARIETA'

2.5 ASSOCIAZIONE DINAMICA

2.6 VANTAGGI DELLA PROGRAMMAZIONE AD OGGETTI





2.0 PREMESSA

La programmazione orientata agli oggetti è associata alle *nuove tecniche di sviluppo che pongono i dati, anziché le funzioni, al centro del processo di sviluppo di un sistema.*

I concetti principali sui quali è basato il software orientato agli oggetti sono:

- l'astrazione dei dati;
- incapsulamento o information hiding;
- il polimorfismo;
- l'ereditarietà;
- l'associazione dinamica.

Lo scopo di questo capitolo è quello di descrivere questi concetti.

2.1 ASTRAZIONE DEI DATI

Il concetto primario sul quale si basa la programmazione orientata agli oggetti è l'astrazione dei dati. Una categoria astratta di dati è una raccolta di dati e di operazioni che devono essere svolte su tali dati.

La specifica di una categoria astratta di dati s'incentra sulle operazioni che dovranno essere applicate ai dati, non sulla struttura dei dati.

Esempio:

Gli ingegneri del software che progettano in termini di categorie astratte, come ad esempio uno stack⁵, non sono molto interessati al tipo di struttura dello stack, se questo debba essere costituito da un array o da una lista, ma bensì all'effetto delle operazioni sullo stack. All'ingegnere del software interessa che quando un elemento viene registrato nello stack (funzione push) e successivamente ripristinato (funzione pop), lo stack rimanga inalterato.

⁵ Uno stack è una pila di elementi messi uno sopra l'altro. Il funzionamento di uno stack può essere di tipo LIFO (last in first out) o di tipo FIFO (first in first out).



Il meccanismo che consente di realizzare tutto questo è la **classe**.

Una classe costituisce lo schema per descrivere i dati e le operazioni da svolgere sui dati. Dalla classe si genera l'oggetto. La struttura reale di una classe, intesa come sintassi, varia in funzione del linguaggio di programmazione che viene utilizzato.

Qui di seguito viene riportato un esempio di classe utilizzando la sintassi del C++:

```
class stack {  
    private:  
        double depot [100];  
        int i;  
    public:  
        double push (double);  
        double pop (double);  
        void print (double) ;  
};  
  
stack buffer ;    // Creazione di uno oggetto di nome  
                  « buffer » di classe ( o tipo) « stack »
```

Esempio di dichiarazione di una classe in C++.

Le categorie astratte di dati sono considerate oggi come il fondamento dei linguaggi di programmazione orientati agli oggetti, al punto che molti esperti ritengono che lo sviluppo di un sistema software basato su tali linguaggi sia una mera manipolazione di categorie astratte di dati, realizzata grazie alle altre caratteristiche di supporto a tale processo di sviluppo.



2.2 L'INCAPSULAMENTO O INFORMATION HIDING

Il modo, descritto nel paragrafo precedente di dichiarare gli oggetti presenta un vantaggio principale: le istruzioni del programma che usano gli oggetti definiti dalle classi non dovranno contenere riferimenti alle istruzioni interne dei dati delle classi. Questo significa quando un sistema che usa le classi deve essere modificato (per esempio durante la manutenzione), tutte le modifiche tendono ad essere limitate alle classi.

Questo concetto, noto come *“information hiding”*, è il punto cruciale per realizzare dei sistemi che siano semplici da mantenere⁶.

L'incapsulamento, o *information hiding*, è il meccanismo che riunisce insieme il codice e i dati da esso manipolati e che mette entrambi al sicuro da interferenze o errati utilizzi.

In un linguaggio a oggetti, il codice ed i dati possono essere raggruppati in modo da creare una sorta di “scatola nera”. Quando il codice e i dati vengono raggruppati in questo modo, si crea un oggetto. In altre parole, un oggetto è un “dispositivo” che supporta l'incapsulamento o *information hiding*.

All'interno di un oggetto, il codice, i dati o entrambi possono essere privati di tale oggetto o pubblici (vedere esempio riportato nella pagina precedente). Il codice o i dati privati sono noti e accessibili solo da parte degli elementi dell'oggetto stesso. Questo significa che il codice e i dati privati non risultano accessibili da parte di elementi del programma che si trovano all'esterno dell'oggetto. Se il codice o i dati sono pubblici, risulteranno accessibili anche da altre parti del programma che non sono definite all'interno dell'oggetto.

Generalmente le parti pubbliche di un oggetto sono utilizzate per fornire un'interfaccia di controllo agli elementi privati dell'oggetto.

Un oggetto è in tutto per tutto una variabile di un tipo definito dall'utente.

Può sembrare strano pensare a un oggetto, che contiene codici e dati, come a una variabile. Tuttavia nella programmazione a oggetti, avviene proprio così. Ogni volta che si definisce un nuovo tipo di oggetto, si crea implicitamente un nuovo tipo di dati.

⁶ Sebbene il concetto sia stato sviluppato indipendentemente dai linguaggi di programmazione orientati agli oggetti, tuttavia esso è stato giustamente, acquisito dai progettisti di questi linguaggi.



2.3 IL POLIMORFISMO

Quando avrete sviluppato un certo numero di sistemi software, potrete notare l'alto grado di comunanza di caratteristiche tra le categorie astratte di dati che sono state utilizzate. Per esempio, virtualmente gran parte dei sistemi software richiedono l'uso di tabelle. Gli elementi di una tabella possono essere differenti: nei sistemi che elaborano i dati, gli elementi tipici potrebbero essere i prezzi dei prodotti di una azienda, mentre nei sistemi operativi, gli elementi più caratteristici sono i files e le procedure. Nonostante ciò, nella maggior parte del software, le strutture che gestiscono i dati si replicano da un'applicazione all'altra.

I linguaggi di programmazione orientata agli oggetti consentono alle classi di descrivere gli oggetti generici che possono essere orientati verso una particolare applicazione.

Questa caratteristica è nota come *polimorfismo* o *genericità*.

Il polimorfismo aiuta a ridurre la complessità del programma consentendo di utilizzare la stessa interfaccia per accedere a una classe generale di azioni.

2.4 EREDITARIETA'

Una delle principali caratteristiche dei linguaggi di programmazione orientati agli oggetti è l'ereditarietà (inheritance). Questo aspetto consente ad un programmatore di definire una classe base che contiene i dati e le operazioni comuni e di sviluppare nuove classi che riflettano le diverse categorie, ereditando molte proprietà della classe base e introducendo quelle appropriate alle nuove categorie.

Se si prova a riflettere, la maggior parte della conoscenza è resa più gestibile da classificazioni gerarchiche. Ad esempio, una mela gialla Renetta appartiene alla classificazione "mela" che a sua volta appartiene alla classe "frutta" che a sua volta si trova nella classe "cibo".

Senza l'uso dell'ereditarietà, ogni oggetto dovrebbe essere definito esplicitamente con tutte le proprie caratteristiche.

L'uso della classificazione consente di definire un oggetto sulla base delle qualità che lo rendono unico all'interno della propria classe. Sarà il meccanismo di ereditarietà a rendere possibile per un oggetto di essere una specifica istanza di un caso più generale.



37993.4441

Copia per uso personale



2.5 ASSOCIAZIONE DINAMICA

L'ultima caratteristica di cui dispone un linguaggio di programmazione orientato agli oggetti è "l'associazione dinamica" (dynamic binding). Con questo termine s'intende la capacità di un linguaggio di programmazione di determinare la forma dinamica di un oggetto durante l'esecuzione di un programma.

Per esempio, si supponga un sistema di gestione del personale descritto da due classi:

- *dipendenti*, che descrive la totalità dei dipendenti di una società;
- *manager*, che descrive i dipendenti che guidano la società.

Si supponga, inoltre, che ciascuna di queste due classi abbia una distinta procedura pubblica (o funzione) , individuata con il nome *print_attribute()*, che stampa i dettagli degli oggetti descritti dalle rispettive classi.

La seguente serie di istruzioni serve a dimostrare il concetto di associazione dinamica:

```
...
dipendente e;
manager m;

...

...

...

e:= m;
e.print_attribute ();
```

Quale versione della funzione *print_attribute ()* dovrà essere utilizzata?

Un linguaggio di programmazione orientato agli oggetti deve essere in grado di "capire" che la variabile *e*, anziché contenere un oggetto della classe *dipendente*, contiene un oggetto della classe *manager* e , pertanto, deve applicare automaticamente la funzione associata a questo oggetto.

Questa capacità di riconoscere il tipo di oggetto è una caratteristica cruciale. Che consente ai linguaggi di programmazione orientati agli oggetti di ridurre notevolmente il lavoro di programmazione richiesto nelle applicazioni software più complesse.



2.6 VANTAGGI DELLA PROGRAMMAZIONE AD OGGETTI

Vi sono due vantaggi ad utilizzare un linguaggio di programmazione ad oggetti.

In primo luogo, le categorie astratte dei dati favoriscono lo sviluppo di sistemi software che, per essere modificati, richiedono soltanto delle modifiche a parti relativamente piccole.

In secondo luogo, le caratteristiche di polimorfismo, ereditarietà e associazione dinamica consentono al progettista del software di creare una libreria di classi base che possono essere riutilizzate, abbastanza facilmente, in nuove applicazioni.

Una delle considerazioni più importanti da fare sulla programmazione orientata agli oggetti, sui linguaggi orientati agli oggetti e sull'ingegneria del software orientato agli oggetti è il cambiamento radicale dell'approccio dei progettisti di software al concetto di sviluppo del sistema. Aniché concentrarsi sulla funzionalità (ciò che un sistema dovrà fare), i progettisti dovranno concentrarsi sui dati che stanno alla base del sistema.



Copia per uso personale



CAPITOLO 3

3.0 PREMESSA

3.1 CREARE LE CLASSI

3.2 CONTROLLO DELL'ACCESSO ALLA CLASSE

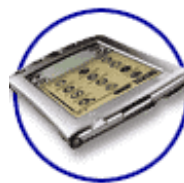
3.3 COMPORTAMENTO DEI MEMBRI PROTECTED (EREDITARIETA' MULTIPLA)

3.4 EREDITARIETA' DA PIU' CLASSI

3.5 COSTRUTTORI E DISTRUTTORI

3.6 FUNZIONI FRIEND

3.7 CLASSI FRIEND





3.0 PREMESSA

I linguaggi di programmazione orientati agli oggetti sono linguaggi che consentono agli sviluppatori di software di definire gli *oggetti* attraverso un meccanismo chiamato “**classe**”.

Scopo di questo capitolo è quello di introdurre i formalismi che governano le classe.

La sintassi utilizzata negli esempi è quella del linguaggio C++.

3.1 CREARE LE CLASSI

Una cosa importante da tenere sempre presente è che *la dichiarazione di una classe definisce un nuovo tipo di dato che racchiude in se sia del codice che dei dati.*

Questo nuovo tipo di dati verrà utilizzato dal programmatore per dichiarare *oggetti* appartenenti a tale classe.

Definizione:

Una classe è un’astrazione logica, mentre un oggetto ha esistenza fisica.

Tecnicamente si definisce un oggetto come *un’istanza* di una classe.

Di seguito viene presentata la forma generale completa della dichiarazione di una classe che non erediti proprietà da altre classi:

```
class nome_classe{  
    specificatori di accesso:  
        dati e funzioni  
    specificatori di accesso:  
        dati e funzioni  
    specificatori di accesso:  
        dati e funzioni  
} elenco oggetti;
```



Copia per uso personale

Esaminiamo in dettaglio la dichiarazione della classe sopra scritta:

class : parola riservata atta a far comprendere al compilatore che ciò che segue si riferisce alla definizione di una struttura nuova di dato

nome_classe : nome proprio che identifica il nuovo tipo di dato strutturato.

Specificatori di accesso:

Gli specificatori di accesso sono di tre tipi identificati dalle seguenti parole chiave⁷:

- public
- private
- protected

Le funzioni e i dati dichiarati all'interno di una classe sono *normalmente privati* di tale classe e possono essere utilizzati solo dagli altri membri della classe.

Utilizzando lo specificatore **public** si consente però anche ad altre parti del programma di accedere alle funzioni o ai dati della classe.

Lo specificatore di accesso **protected** è richiesto solo in caso di ereditarietà⁸.

Una volta utilizzato, uno specificatore d'accesso rimane attivo finché non viene indicato un altro specificatore di accesso o finché non viene raggiunta la fine della dichiarazione della classe.

Dati: dati della classe, per esempio variabili e costanti. Le variabili che sono elementi di una classe sono chiamate "*variabili membro*" o "*dati membro*".

Funzioni: funzioni dichiarate all'interno di una classe sono chiamate "*funzioni membro*". Le funzioni membro possono accedere a tutti gli elementi della classe di cui fanno parte e quindi anche agli elementi privati.

elenco oggetti : istanze della classe. L'oggetto è un'entità reale che occupa un quantitativo di memoria. L'elenco oggetti è opzionale.

⁷ Si fa riferimento al linguaggio C++

⁸ Sarà spiegato dettagliatamente nel prossimo paragrafo.



Copia per uso personale

Sono poche le restrizioni applicabili ai membri di una classe:

- Nessun membro può essere un oggetto della classe dichiarata⁹;
- Nessun membro può essere dichiarato come : *auto extern* o *register*¹⁰;

In generale, si dovranno rendere tutti i dati membri di una classe privati di tale classe. Questo consente di mantenere l'incapsulamento dei dati.

Tuttavia vi possono essere situazioni in cui si devono rendere pubbliche una o più variabili. Quando una variabile è pubblica, è possibile accedere ad essa direttamente da qualsiasi punto del programma.

La sintassi di accesso a dati membri pubblici è la stessa di una chiamata a una funzione membro: si deve specificare il nome dell'oggetto, il punto (operatore d'azione di campo) e il nome della variabile.

Esempio:

```
...  
class combo{  
    private:  
        int i, j;  
    public:  
        float k;  
    } primo, secondo;  
...  
...  
primo.k = 123;  
secondo.k = 30;
```

⁹ Anche se un membro può essere un puntatore alla classe dichiarata.

¹⁰ Specificatore di classe di memorizzazione, dicono al compilatore il modo in cui memorizzare le variabili.



3.2 CONTROLLO DELL'ACCESSO ALLA CLASSE

L'ereditarietà consente di creare classificazioni gerarchiche. Utilizzando l'ereditarietà, è possibile creare una classe generale che definisce le caratteristiche comuni a una serie di oggetti correlati. Questa classe può in seguito essere ereditata da una o più classi, ognuna delle quali aggiunge alla classe ereditata solo elementi specifici.

Per conservare la terminologia standard, la classe ereditata viene chiamata "classe base". La classe che "riceve" l'eredità è detta "classe derivata". A sua volta, una classe derivata può fungere da classe base per un'altra classe derivata.

Quando una classe ne eredita un'altra, i membri della classe base divengono membri della classe derivata.

L'ereditarietà delle classi ha la seguente forma generale:

```
class nome_classe_derivata : specificatore_di_accesso nome_classe_base {  
                                                                    //corpo della classe  
};
```

esempio:

```
class mela : public frutto {  
                                                                    //corpo della classe  
};
```

Il tipo di `specificatore_di_accesso` utilizzato determina la natura dell'accesso della classe derivata sui membri della classe base. Lo `specificatore_di_accesso` alla classe base può essere: `public`, `private` o `protected`.

Se non si indica uno specificatore d'accesso, si possono verificare i seguenti casi:

- se la classe derivata è una **class**, lo specificatore d'accesso sarà `private`;
- se la classe derivata è una **struct**, lo specificatore standard sarà `public`;



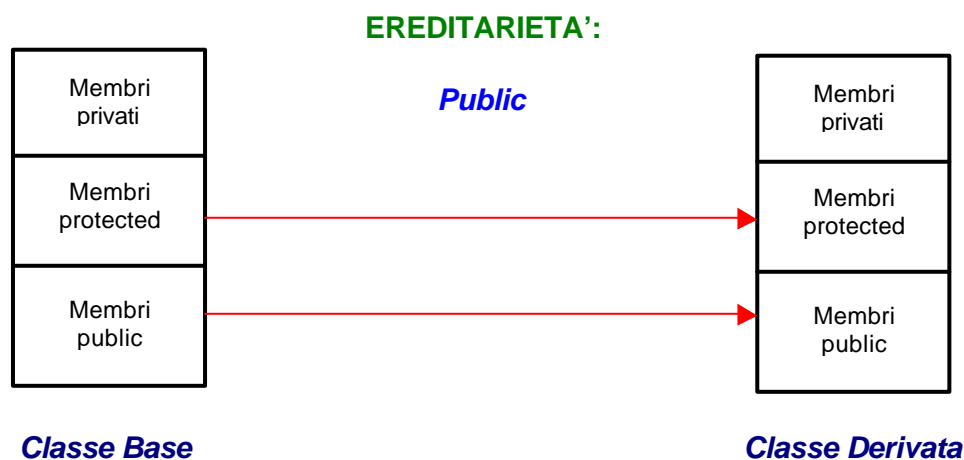
Copia per uso personale

Ora vediamo le implicazioni dell'uso degli specificatori di accesso:

Public:

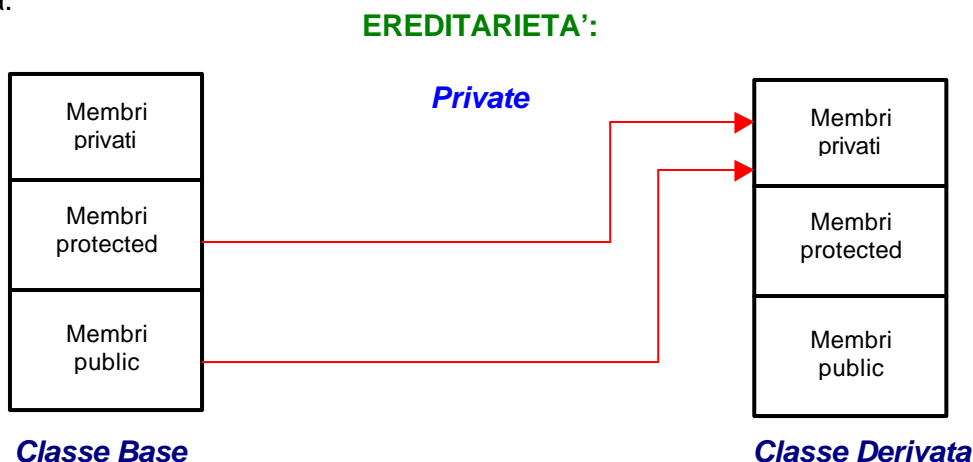
Quando lo specificatore di accesso alla classe base è *public*, tutti i membri pubblici della classe base diverranno membri pubblici della classe derivata e tutti i membri protetti della classe base diverranno membri protetti anche della classe derivata.

In tutti i casi, gli elementi privati della classe base rimarranno privati della classe base e non saranno pertanto accessibili da parte dei membri della classe derivata.



Private:

Quando la classe base è ereditata tramite lo specificatore d'accesso *private*, tutti i membri pubblici e protetti della classe base diverranno membri privati della classe derivata. Questo significa che rimarranno accessibili da parte dei membri della classe derivata e inaccessibili da parte di altri punti del programma che non siano membri della classe base o della classe derivata.

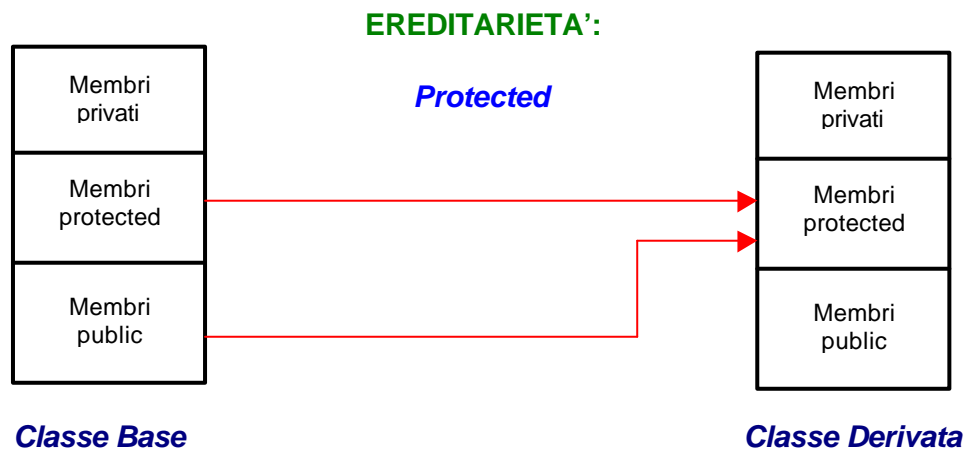




Copia per uso personale

Protected:

Se una classe base è **ereditata come protected**: in tal caso tutti i membri public e protected della classe base diverranno membri protected della classe derivata.





37993.4441



Copia per uso personale

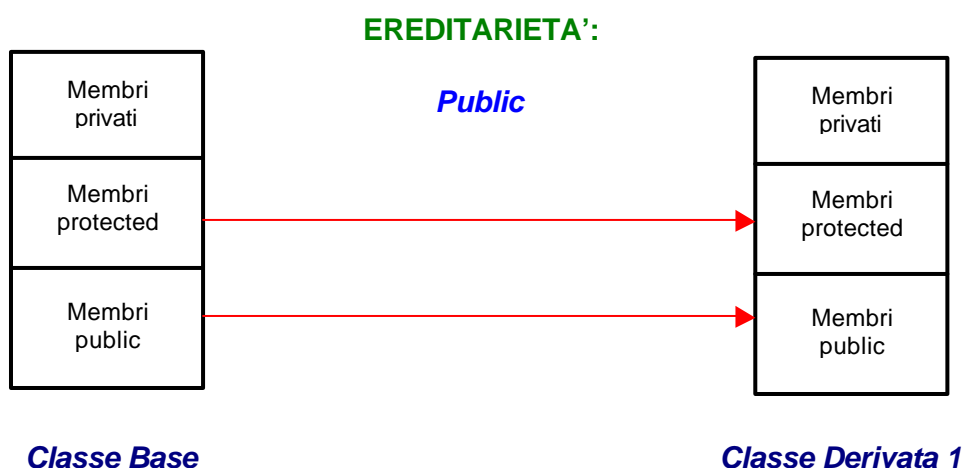
3.3 COMPORTAMENTO DEI MEMBRI PROTECTED (EREDITARIETA' MULTIPLA)

Lo specificatore *protected* è stato introdotto per fornire maggiore flessibilità nel meccanismo di ereditarietà. Quando un membro di una classe è dichiarato *protected*, tale membro non sarà accessibile da parte di altri elementi non membri della classe stessa (come visto per i membri *private*). L'unica eccezione si verifica quando viene ereditato un membro *protected*. In questo caso, un membro *protected* è molto diverso da un membro *private*.

Come si è scritto precedentemente, un membro *private* di una classe base non è accessibile da altre parti del programma, incluse tutte le classi derivate.

I membri *protected* si comportano in modo differente:

- se la classe base è **ereditata come public**: i membri *protected* della classe base diverranno membri *protected* della classe derivata e saranno pertanto accessibili alla classe derivata stessa. In altre parole, utilizzando la parola *protected*, un programmatore può creare membri della classe che siano privati di tale classe ma che possono essere ereditati e manipolati dalle classi derivate.



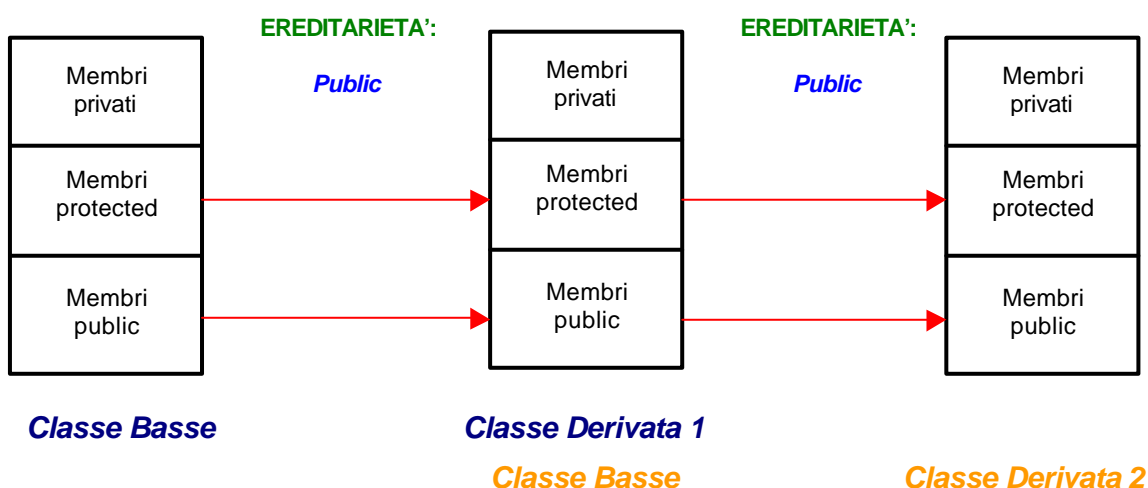


37993.4441

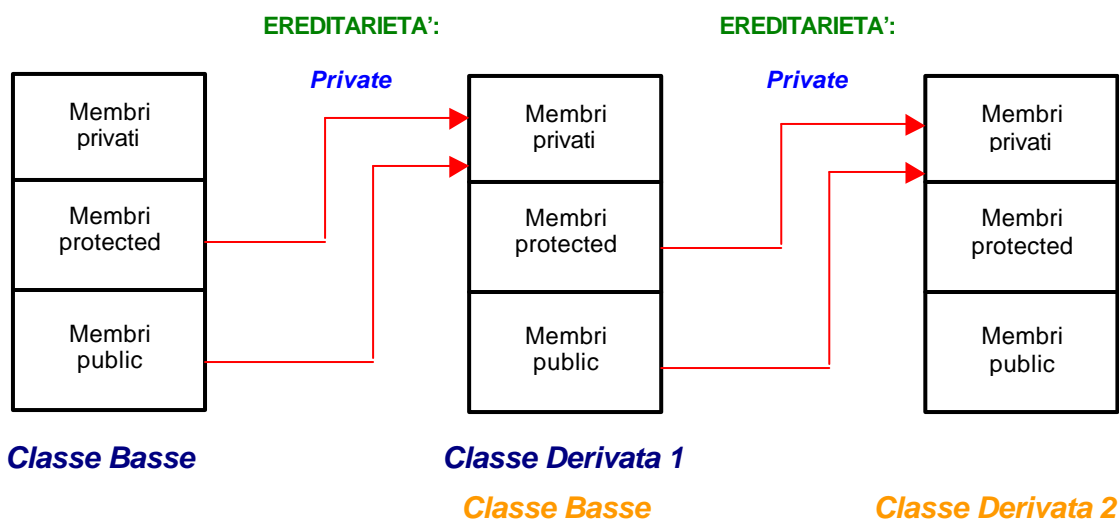
Copia per uso personale



Quando una classe derivata è utilizzata come classe base di un'altra classe derivata, tutti i membri *protected* della classe base iniziale ereditati (come *public*) della prima classe derivata potranno essere ereditati nuovamente come *protected* anche dalla seconda classe derivata.



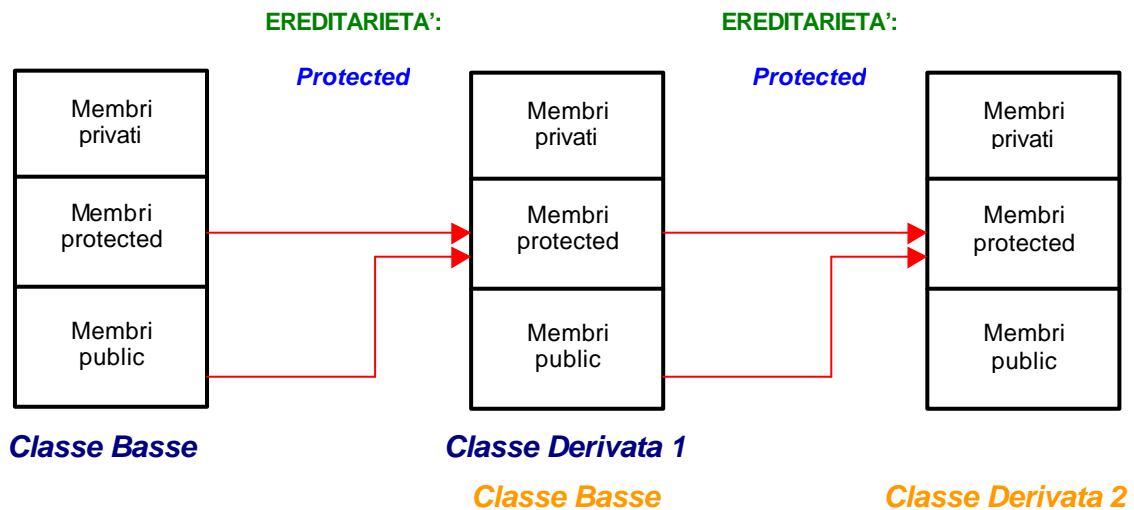
- Se la classe base è **ereditata come private**: tutti i membri della classe base diventano membri privati della classe derivata e pertanto risulterebbero inaccessibili ad un secondo grado di ereditarietà.





Copia per uso personale

- Se la base è **ereditata come protected**: tutti i membri public e protected della classe base diventano protected della classe derivata, quindi trasportabili in un secondo grado di ereditarietà.





Copia per uso personale

3.4 EREDITARIETA' DA PIU' CLASSI

Una classe derivata può ricevere in eredità elementi di due o più classi base.

Per ereditare si deve utilizzare un elenco di classi separati da virgole, utilizzando un o specificatore di accesso per ogni classe.

```
class nome_classe : specificatore_di accesso classe_Base1 , ... , specificatore_di accesso classe_Base N {  
    //Corpo della classe  
};
```

Esempio:

```
class mela : public frutto , public vegetale , private molecola {  
    ...  
    ...  
};
```



3.5 COSTRUTTORI E DISTRUTTORI

E' molto comune che una parte di un oggetto debba essere inizializzata prima dell'uso.

I linguaggi di programmazione orientata agli oggetti hanno degli strumenti che consentono di inizializzare gli oggetti al momento della creazione. Questa inizializzazione automatica è ottenuta grazie all'impiego di una funzione "costruttore".

La funzione costruttore è una particolare funzione membro di una classe che porta lo stesso nome della classe:

```
class complex {  
    int real;  
    int imag;  
    public:  
        complex() ;    // costruttore  
        void print () ;  
};  
  
...  
...  
//definizione della funzione costruttore  
complex::complex()  
{  
    real =0;  
    imag=0;  
}  
...  
...
```

Si noti che per il costruttore non viene specificato il tipo restituito¹¹.

Il costruttore di un oggetto viene richiamato automaticamente nel momento in cui deve essere creato l'oggetto. Questo significa che viene richiamata al momento della dichiarazione dell'oggetto.

¹¹ In C++ le funzioni costruttore non possono restituire valori e pertanto non si deve specificare il tipo restituito.



L'operazione complementare del costruttore è il “*distruttore*”.

Gli oggetti locali vengono costruiti al momento in cui si entra nel blocco in cui si trovano e vengono distrutti all'uscita del blocco. Gli oggetti globali vengono distrutti nel momento in cui termina il programma.

Quando viene distrutto un oggetto, viene automaticamente richiamato il relativo distruttore.

Vi sono molti casi in cui è necessario utilizzare una funzione distruttore. Ad esempio, potrebbe essere necessario deallocare la memoria precedentemente allocata dall'oggetto oppure potrebbe essere necessario chiudere un file aperto.

Il distruttore ha il compito di gestire gli eventi di disattivazione.

Il *distruttore* ha lo stesso nome della classe ma è preceduto dal carattere ~ (tilde)¹².

```
class complex {
    int real;
    int imag;
public:
    complex() ;    // costruttore
    ~complex () ;  // distruttore
    void print () ;
};

...

...

//definizione della funzione distruttore
complex::~~complex()
{
    printf("oggetto distrutto");
}

...

...
```

¹² Il carattere tilde lo si ottiene con la combinazione di tasti ALT+126



37993.4441

Copia per uso personale



3.6 FUNZIONI FRIEND

Una *funzione friend* o “ amica” può accedere a tutti i membri *private* e *protected* della classe per la quale è dichiarata friend.

Per dichiarare una funzione friend, se ne deve includere il prototipo nella classe¹³.

```
class complex {
    int real;
    int imag;
public:
    complex () ;           // costruttore
    ~complex ();           // distruttore
    friend int sum (complex y); // funzione friend
    void print () ;
};
...
...
int sum (complex y)        // sum() non è funzione membro della classe
{
    return y.real + y.imag ; // accede ai termini privati perché è funzione friend
}
```

E' importante notare che la funzione friend, in questo caso sum(), viene definita normalmente e verrà utilizzata senza l'ausilio dell'operatore punto, in quanto non è una funzione membro della classe.

L'utilità di questa tipologia di funzione la si trova quando si deve:

- eseguire l'overloading di funzioni;
- alcuni overloading di operatori;
- semplificazioni di funzioni di I/O
- quando due o più classi contengono membri correlati con altre parti del programma.

¹³ In C++ le si fa precedere dalla parola riservata *friend*.



3.7 CLASSI FRIEND

E' anche possibile che un'intera classe sia *friend* di un'altra classe. In questo caso, la classe friend e tutte le sue funzioni membro avranno accesso ai membri privati definiti all'interno dell'altra classe.

```
class complex {  
    int real;  
    int imag;  
public:  
    complex () ;           // costruttore  
    ~complex ();           // distruttore  
    friend int sum (complex y); // funzione friend  
    friend class part_real ; // classe friend  
    void print () ;  
};  
  
// definizione classe part_real  
class part_real  
{  
public:  
    int r ( complex k);  
};  
...  
...  
  
int part_real:: r (complex k)  
{  
    return k.real;  
}
```

E' importante ricordare che quando una classe è *friend* di un'altra, ha solamente accesso ai nomi definiti nell'altra classe ma non eredita le caratteristiche dell'altra classe.



Copia per uso personale



CAPITOLO 4

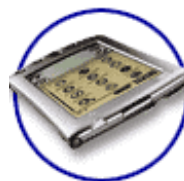
4.0 PREMESSA

4.1 FUNZIONI GENERICHE

4.2 UNA FUNZIONE TEMPLATE CON DUE O PIU' TIPI GENERICI

4.3 CLASSI GENERICHE

4.4 LA POTENZA DEI TEMPLATE





37993.4441



Copia per uso personale

4.0 PREMESSA

I **template**, o modelli, sono una delle funzionalità più sofisticate e potenti¹⁴. Con un template è possibile creare *funzioni* e *classi generiche*. In una funzione o classe generica, il tipo dei dati su cui la funzione o la classe operano viene specificato come parametro. Pertanto, si potrà utilizzare una funzione o una classe con vari tipi di dati senza dover ricodificare esplicitamente una diversa versione specifica per ogni tipo di dati.

4.1 FUNZIONI GENERICHE

Una funzione generica definisce una serie di operazioni generali applicabili a vari tipi di dati.

Il tipo dei dati su cui si troverà ad operare viene passato tramite un parametro.

Molti algoritmi sono infatti logicamente identici indipendentemente dal tipo di dati su cui operano.

Creando una “funzione generica” è possibile definire, indipendentemente dai dati, la natura dell'algoritmo. In questo modo, il compilatore genererà il codice corretto per il tipo di dati effettivamente utilizzato per richiamare la funzione. In pratica, quando si crea una funzione generica, si crea una funzione in grado di eseguire automaticamente l'overloading di sé stessa.

Per creare una funzione generica si utilizza un opportuno costrutto¹⁵ che fa capire al compilatore che ciò che si è definito è una funzione generica (o template):

```
template <typename16 tipo> tipo_restituito nome_funzione (elenco parametri formali)
{
    //corpo della funzione
}
```

¹⁴ Ci si riferisce in particolar modo al linguaggio C++.

¹⁵ Ciò che segue fa riferimento alla creazione di un template realizzato in linguaggio C++.

¹⁶ Al posto della parola riservata *typename* si può utilizzare anche la parola *class*. (n.d.a.: A mio modesto parere l'utilizzo della parola *class* potrebbe indurre a confusione.



Copia per uso personale

esempio:

...

```
template <typename X> void swapval (X a, X b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

...

```
int main ()
{
    int i = 20, s = 30;
    float w = 2.345, r = 34,5786;
    char a = 'q', b = 'm';
    ...
    ...
    swapval (i,s);    // scambia due interi
    swapval (w,r);    // scambia due float
    swapval (a,b);    // scambia due char
    ...
    ...
}
```

Osservando attentamente il listato, si può notare che la riga:

```
template <typename X> void swapval (X a, X b)
```

dice al compilatore le cose:

- 1) che si deve creare una funzione template e che sta per iniziare una definizione generica. Qui **X** è un tipo generico utilizzato come segnaposto. Dopo la posizione template viene dichiarata la funzione `swapval()` utilizzando **X** per definire i tipi di dati che devono essere scambiati
- 2) Poiché `swapval()` è una funzione generica, il compilatore crea automaticamente le tre versioni di funzioni: per `int`, per `float` e per `char`.



Ecco, qui di seguito, delle terminologie tecniche che si usano:

Funzione template: E' una funzione generica (ovvero la definizione di una funzione preceduta dall'istruzione *template*).

Funzione generata: Quando il compilatore crea una versione specifica di tale funzione. L'atto di generazione di una funzione viene chiamato *istanziamento*.

4.2 UNA FUNZIONE TEMPLATE CON DUE O PIU' TIPI GENERICI

Con l'istruzione *template* è possibile definire anche più di un tipo di dati generico, utilizzando un elenco separato da virgole.

```
template <typename X, typename Y> void transfert (X a, Y b)
{
    X buffer;
    Y convert;
    convert = buffer;
}
```

Quando si crea una funzione *template*, in realtà si chiede al compilatore di generare tutte le versioni di tale funzione necessarie per gestire tutte le situazioni nelle quali il programma deve impiegare tale funzione.



4.3 CLASSI GENERICHE

Oltre alle funzioni generiche è anche possibile definire una classe generica.

In tal caso si crea una classe che definisce tutti gli algoritmi utilizzati in tale classe ma nella quale il tipo di dati da manipolare viene specificato come parametro nel momento in cui vengono creati gli oggetti della classe.

Le classi generiche sono utili nel caso in cui una classe contenga elementi logici generalizzabili. Ad esempio, lo stesso algoritmo che esegue la gestione di una coda di interi funzionerà anche per una coda di caratteri.

La forma generale della dichiarazione di una classe generica è:

```
template <typename17 tipo> class nome_classe
{
    //corpo della classe
}
```

Qui “*tipo*” è un segnaposto a cui dovrà essere sostituito il nome del tipo al momento dell’istanziamento della classe. Se necessario, si può definire più di un tipo di dati generico utilizzando un elenco separato da virgole.

Dopo aver creato una classe generica, sarà possibile creare una specifica istanza di tale classe utilizzando la forma generale:

```
nome_classe < tipo > oggetto;
```

Qui “*tipo*” è il nome del tipo dei dati su cui la classe si troverà ad operare.

Le funzioni membro di una classe generica sono anch’esse (automaticamente) generiche.

Non è necessario usare “*template*” per specificarle come tali.

¹⁷ Al posto della parola riservata *typename* si può utilizzare anche la parola *class*. (n.d.a.: A mio modesto parere l’utilizzo della parola *class* potrebbe indurre a confusione)



37993.4441



Copia per uso personale

Esempio:

...

```
template <typename Stacktype> class stack
{
    Stacktype stck[100];
    int tos;           // indice della cima dello stack
public:
    stack() { tos = 0; }; // costruttore
    void push (StackType ob);
    Stacktype pop();
};
```

// definizione delle funzioni membro

// FUNZIONE PUSH

```
template <typename StackType> void stack <Stacktype> :: push (StackType ob)
```

```
{
    if (tos==100)
    { printf("Lo spazio è pieno.");
      return;
    }
    stck[tos] = ob;
    tos++;
}
```

// FUNZIONE POP

```
template <typename StackType> void stack <Stacktype> :: pop (StackType ob)
```

```
{
    if (tos== 0)
    { printf("Stack vuoto.");
      return 0;
    }
    tos--;
    return stck[tos];
}
```



Copia per uso personale

```
...  
...  
int main()  
{  
...  
...  
    stack <char> s1, s2 ;    // crea due stack di tipo carattere  
    s1.push("a");  
...  
    stack <double> d1, d2;  // crea due stack di tipo carattere  
    d1.push (13.3426);  
...  
...  
}
```



4.4 LA POTENZA DEI TEMPLATE

I *template* aiutano a raggiungere uno degli obiettivi più difficili della programmazione: creare codice riutilizzabile.

Le funzioni e le classi generiche costituiscono strumenti molto potenti per amplificare i propri sforzi di programmazione. Dopo aver scritto e corretto una *classe template*, si sarà creato in solido componente software utilizzabile con sicurezza in varie situazioni.

Dunque non sarà più necessario creare implementazioni distinte per ciascun tipo di dati cui dovrà essere applicata la classe.

Le *funzioni* e *classi template* stanno diventando sempre più comuni in programmazione¹⁸. Anche se i template aggiungono un livello di astrazione al programma, producono comunque codice oggetto ad alte prestazioni.

¹⁸ Ad esempio, la libreria STL (Standard Template Library) definita in C++ è costruita sui template.



Copia per uso personale

RINGRAZIAMENTI

Con la presente nota voglio ringraziare tutti coloro che utilizzano questa documentazione, che segnalano imperfezioni od errori o che inviano consigli per modificarla¹⁹.

Grazie per il vostro aiuto.

Si ringrazia in particolar modo:

*Giorgi M. T.
Parsipal
Perino Andrea*

¹⁹ Per l'invio di segnalazioni inerenti a questa documentazione scrivere all'indirizzo mail g.iozzelli@tidestudio.com. Grazie.



BIBLIOGRAFIA

(in fase di aggiornamento)

Hekmatpour, S. H. e Ince, D. C. (1988) *Software Prototyping, Formal Method and VDM*, Addison-Wesley, Reading, Mass.

Meyer, B. (1988) *Object-oriented Software Construction*, Prentice-Hall, Englewood Cliff, NJ.

Darrel Ince (1992) *Object-oriented software engineering with C++*, McGraw Hill, UK.

Lamb, D.A. (1988) *Software Engineering*, Prentice-Hall, Englewood Cliffs, NJ

Herbert Schildt (1998) *C++: The complete Reference*, Osbornw McGraw Hill Inc.



37993.4441



Copia per uso personale

Pagina bianca preposta a contenere annotazioni.



37993.4441



Copia per uso personale

Pagina bianca preposta a contenere annotazioni.



37993.4441



Copia per uso personale

Pagina bianca preposta a contenere annotazioni.