Fondamenti della Tecnologia ad Oggetti

Serafino Cicerone

Dipartimento di Ingegneria Elettrica
Università degli Studi dell'Aquila
I-67040 Monteluco di Roio, L'Aquila, Italy
cicerone@ing.univaq.it

Sommario

Dispensa per il corso di *Programmazione ad Oggetti – AA 2003/04* (il materiale presente in questa dispensa proviene principalmente dal volume Progettazione a oggetti con UML, Meilir Page-Jones. Apogeo, Gennaio 2002)

Indice

Introduzione

L'espressione orientato agli oggetti in sé non significa nulla: oggetto è una delle parole più generali che si possano trovare in una lingua come l'italiano o l'inglese. Basta cercarla su un dizionario per rendersene conto; ecco infatti una delle definizioni che si pu trovare:

Definizione 1 (Oggetto) *Per oggetto si intende una cosa presentata o in grado di essere presentata ai sensi.*

In altre parole, un oggetto può essere praticamente qualunque cosa. Nemmeno la parola orientato è di molto aiuto: definita come diretto verso o indirizzato verso, in genere trasforma il termine orientato agli oggetti in un aggettivo. Pertanto abbiamo questa definizione:

Definizione 2 (Orientato agli oggetti) *"Orientato agli oggetti" equivale a "indirizzato verso quasi qualunque cosa cui si possa pensare".*

Non c'è quindi da meravigliarsi che l'industria del software abbia sempre avuto dei problemi a trovare una definizione ampiamente condivisa di orientato agli oggetti, e nemmeno che questa mancanza di chiarezza un tempo consentisse facilmente di poter dire che un certo software fosse orientato agli oggetti.

Se si chiedesse ad esperti del mondo della programmazione di produrre una definizione di orientamento agli oggetti, ognuno di loro troverebbe serie difficoltà e probabilmente ripiegherebbe nel produrre l'elenco delle proprietà considerate indispensabili di un ambiente orientato agli oggetti. La proprietà che probabilmente farebbe parte di ogni elenco è quella di incapsulamento. Insieme all'incapsulamento, le altre proprietà che probabilmente farebbero parte della maggioranza degli elenchi sono le seguenti:

1. Incapsulamento
2. Occultamento delle informazioni e dell'implementazione
3. Conservazione dello stato
4. Identità degli oggetti
5. Messaggi
6. Classi (e loro Composizione ed Aggregazione)
7. Ereditarietà
8. Polimorfismo
9. Genericità

Il modo migliore per dare una certa chiarezza al significato alla base di questi termini è ricorrere a un piccolo esempio di codice a oggetti. Con il progredire dell'analisi del codice nel corso del documento, è possibile scoprire che il gergo dell'orientamento agli oggetti è meno ostico nel significato di quanto possa apparire. E anzi probabile che si conoscano già molte delle nozioni tipiche dell'orientamento agli oggetti, anche se con nomi diversi, grazie alla probabile precedente esperienza nel mondo del software. Tre osservazioni sono comunque doverose prima di iniziare.

Innanzi tutto, il codice che viene presentato è una parte di una semplicissima applicazione a oggetti che visualizza una sorta di robot in miniatura che si sposta su una griglia sullo schermo (il genere di entità che è possibile vedere in un videogame). Sebbene l'orientamento agli oggetti non sia certamente un approccio limitato alle applicazioni grafiche, un'applicazione di questo tipo fornisce un eccellente esempio operativo.

In secondo luogo, dal momento che non ci soffermeremo troppo sulla sintassi o la semantica del codice, non c'è da preoccuparsi se inizialmente il codice non sembrerà perfetto: nel corso della spiegazione della terminologia caratteristica dell'orientamento agli oggetti, chiariremo anche i dettagli del codice stesso. L'algoritmo è stato scritto sotto forma di pseudocodice a oggetti, la cui sintassi è il risultato di una media ricavata dalla sintassi dei principali linguaggi a oggetti, come C++, Java, Eiffel e Smalltalk. Per inciso, segnaliamo che il costrutto `repeat...until...endrepeat` non ha nulla a che fare con l'orientamento agli oggetti: si tratta di un costrutto di pura programmazione strutturata che prevede la verifica della validità della condizione di ciclo in un punto interno al ciclo stesso.

In terzo luogo, sebbene la progettazione delle due classi non sia perfetta, esse sono sufficienti per gli scopi che ci prefiggiamo in questo capitolo (il difetto di progettazione che la caratterizza prende il nome di coesione a dominio misto e non verrà analizzato in queste dispense).

Analizziamo ora l'applicazione gettando uno sguardo a una possibile comunicazione inviata dalla direzione di un gruppo di sviluppo software ai membri del gruppo stesso.

MEMORANDUM

Da: Minali Koad, Direzione sviluppo software
A: Gruppo di sviluppo per il software Robot
Oggetto: Software di controllo Robot (V2.0)

Mi è appena giunta voce dagli uffici delle alte sfere che abbiamo vinto la gara per il contratto relativo al controllo dell'hardware del robot. Questa volta dobbiamo fare un buon lavoro, per riscattarci dal fiasco del robot precedente che è finito sotto una schiacciasassi. Infatti i clienti desiderano una dimostrazione del software su uno schermo prima di dare il via alla realizzazione pratica dell'hardware. La dimostrazione in questione è fissata per lunedì prossimo.

Nella versione 1.0 del software il robot dovrà semplicemente spostarsi su un percorso lineare con qualche curva, come quello illustrato nella Figura 1. La cosa potrebbe essere tradotta in una serie di blocchi quadrati collocati in modo da produrre un percorso della larghezza di un blocco che si stende da un quadrato INIZIO (I) a un quadrato ARRIVO (A). Ogni svolta lungo il percorso sarà ad angolo retto.

Un singolo avanzamento da parte del robot lo porta esattamente avanti di un quadrato (rispetto alla sua direzione corrente). È importante che il robot tocchi tutti i quadrati del percorso da INIZIO ad ARRIVO, ed è ancora più importante che non tocchi le pareti, perché a quel punto faremmo la figura degli stupidi e non ci permetteranno di installare il software nell'hardware vero e proprio.

Fortunatamente la nostra libreria orientata agli oggetti contiene già quattro classi che potrebbero rivelarsi adatte: Robot, Griglia, Posizione e Direzione. Pertanto per lunedì non dovete fare altro che scrivere il codice a oggetti che utilizza le operazioni di queste classi.
In caso di domande potete contattarmi per email. Buon weekend!.

P.S. Includo delle brevi specifiche per le classi di interesse contenute nella libreria.

```
Interfaccia pubblica della classe Griglia {
```

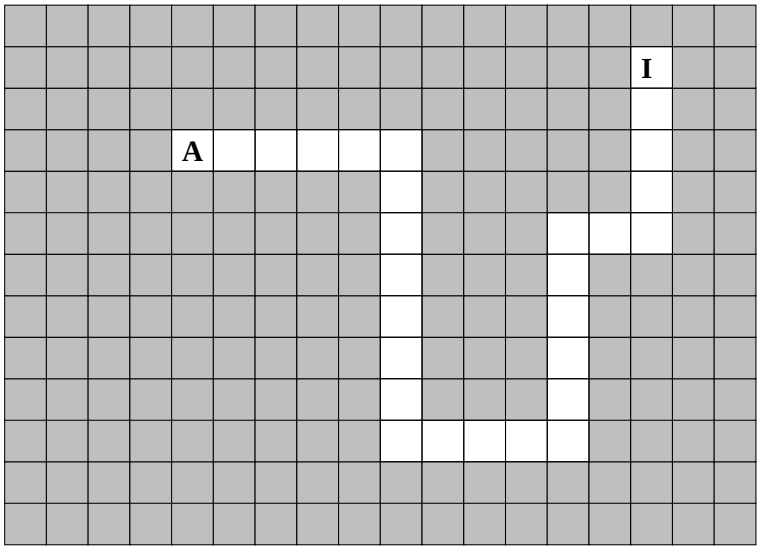


Figura 1: *Un percorso lineare che attraversa la griglia su cui è posto il robot.*

```
// costruttore; la posizione \‘e quella di inizio percorso; gli interi
// rappresentano altezza e larghezza della griglia
   New(\_posizione:Posizione, \_altezza=10: Integer, \_larghezza=10: Integer): Griglia

// rende true se e sole se la posizione in input \‘e nella griglia
   inGriglia( \_posizione: Posizione ): Boolean

// rende true se e solo se la posizione in input \‘e nella griglia ma non
// sul bordo della stessa
   internoInGriglia( \_posizione: Posizione ): Boolean

// rende true se e sole se la posizione in input \‘e nel percorso
   inPercorso( \_posizione: Posizione ): Boolean

// inserisce il robot nella posizione specificata; rende true se e solo
// se l’inserimento \‘e avvenuto in una posizione appartenente al percorso
   inserisciRobot( \_robot:Robot, \_posizione: Posizione ): Boolean

// restituisce la posizione di inizio del percorso
   getInizio(): Posizione

// restituisce la posizione di arrivo del percorso
   getArrivo(): Posizione

// disegna la griglia evidenziando le posizioni di inizio e di arrivo e, se presente,
// il robot (mediante un carattere che n\’e evidenzia anche la direzione)
   disegna()
```

```
}


Interfaccia pubblica della classe Robot {

// costruttore; genera il robot in posizione [0,0] e direzione "NORD"
   New(): Robot

// ruota di 90 a destra la direzione del robot
   giraASinistra()

// ruota di 90 a sinistra la direzione del robot
   giraADestra()

// avanza di una unit\‘a la posizione del robot verso la direzione corrente del robot
   avanza()

// restituisce la posizione corrente del robot
   getPosizione(): Posizione

// modifica la posizione corrente del robot
   setPosizione(\_posizione: Posizione)

// restituisce la direzione corrente del robot
   getDirezione(): Direzione

// modifica la direzione corrente del robot
   setDirezione(\_direzione: Direzione)
}


Interfaccia pubblica della classe Posizione {

// costruttore
   New( \_x=0: Integer, \_y=0: Integer): Posizione

// restituisce la prima coordinata  della posizione
   getX(): Integer

// modifica la prima coordinata  della posizione
   setX( \_x: Integer )

// restituisce la seconda coordinata  della posizione
   getY(): Integer

// modifica la seconda coordinata  della posizione
   setY( \_y: Integer )
```

```
// modifica la posizione passando ad adiacente (secondo la direzione in input)
   moveTo( \_direzione: Direzione )

// confronto di minoranza tra due posizioni
   operator< ( \_posizione: Posizione ): Boolean

// confronto di uguaglianza tra due posizioni
   operator== ( \_posizione: Posizione ): Boolean

// stampa la posizione nel formato [x,y]
   stampa()

 }


Interfaccia pubblica della classe Direzione {

// costruttore
   New( \_direzione="NORD": String ): Direzione

// restituisce la direzione corrente
   getDirezione(): String

// modifica la direzione corrente
   bool setDirezione (string \_direzione);

// modifica la direzione imponendo la direzione successiva a quella
// corrente secondo il senso orario; usare come post-incremento
   operator++( )

// modifica la direzione imponendo la direzione successiva a quella
// corrente secondo il senso anti-orario; usare come post-decremento
   operator--( )

}
```

Il seguente codice ad oggetti permette la creazione di una griglia, di un robot, di inserire il robot nella griglia e poi di fare effettuare un singolo passo sul percorso al robot.

```
//-------- codice per l'avanzamento del robot di una singola posizione
//-------- nel caso il robot sia diretto verso il percorso

var p: Posizione := Posizione.New(5,5);  // creazione di un oggetto
```

```
                                        // di tipo Posizione con
                                        // stato [5,5]

var g: Griglia := Griglia.New(p,20,30);  // creazione di un oggetto di
                                        // tipo Griglia con dimensioni
                                        // 20 x 30.

var a1: Robot := Robot.New();  // creazione di un oggetto di
                               // tipo Robot

var esitoInserimento: Boolean := Boolean.New();
var pareteDiFronte: Boolean := Boolean.New();
var unaPosizione: Posizione := Posizione.New();

unaPosizione := g.getInizio();
esitoInserimento := g.inserisciRobot(a1,unaPosizione);


if not esitoInserimento
then "annulla tutto"
endif;

g.disegna();

// ------- fine fase di creazione oggetti

unaPosizione := a1.getPosizione();
unaPosizione.moveTo( a1.getDirezione() );
pareteDiFronte := g.inPercorso( unaPosizione );

if pareteDiFronte
 then  a1.avanza();
endif;

// ------------------------------------- fine codice
```

Esercizio 1 *A partire da questo codice, scrivere un programma (C++ oppure Java) che permette al robot di attraversare tutte le posizioni del percorso (dalla posizione di inizio fino a quella di arrivo).*

Sfruttando il codice del robot come esempio illustrativo, riprendiamo ora le nove proprietà dell'orientamento agli oggetti indicate in precedenza. E la prima da considerare e senza dubbio quella che apparirebbe sugli elenchi di tutti gli esperti: l'incapsulamento.

1 Incapsulamento

Definizione 3 (Incapsulamento) *L'incapsulamento è il raggruppamento di idee correlate in un'unica unità cui è possibile fare riferimento con un singolo nome.*

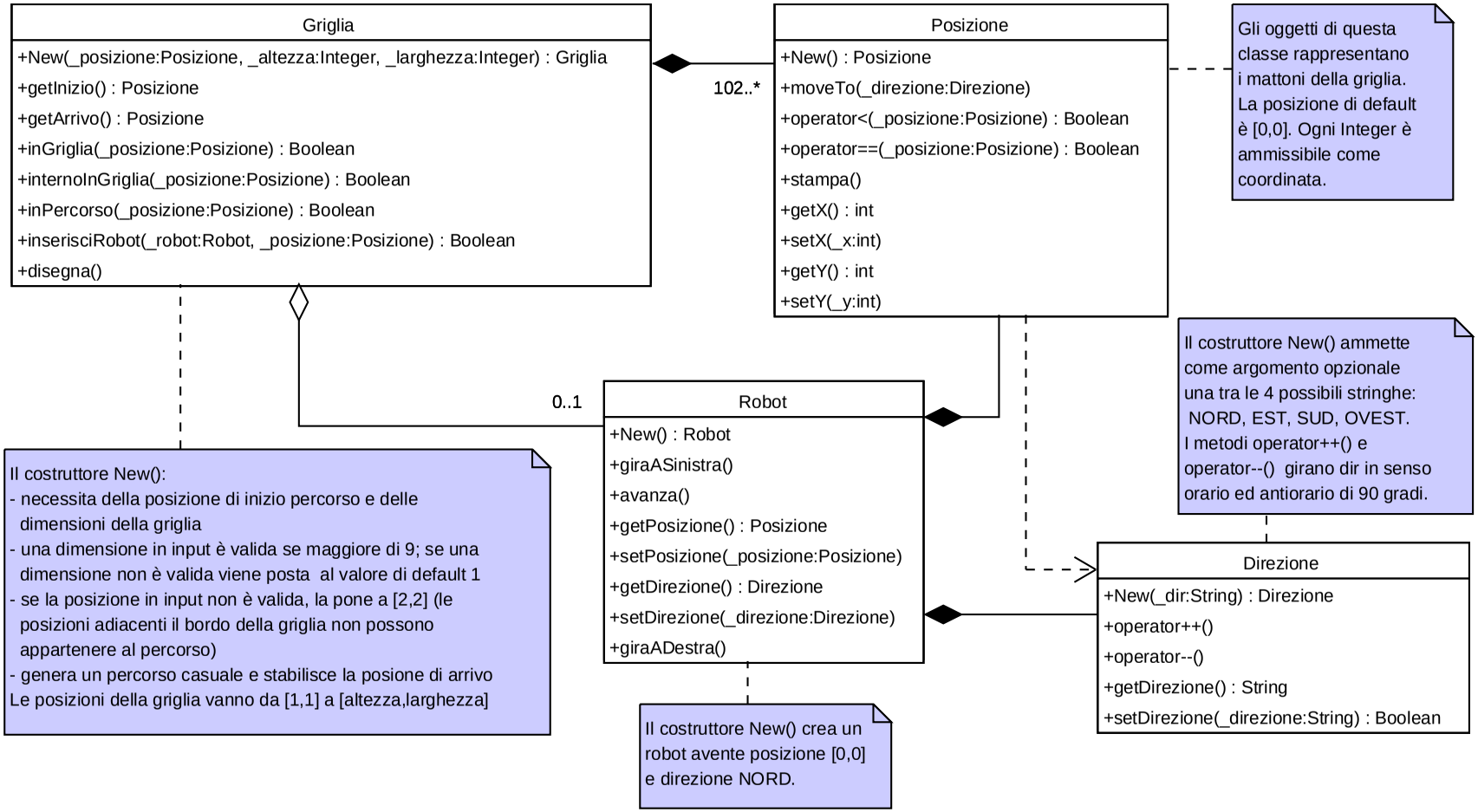


Figura 2: *Diagramma delle classi del progetto Robot (per dettagli sulle relazione tra le classi, vedere sezione 6.1).*

L'incapsulamento del software è un concetto vecchio quasi quanto il software stesso. Già negli anni '40 i programmatori si erano accorti che accadeva spesso che uno stesso schema di istruzioni comparisse diverse volte all'interno dello stesso programma. Alcuni presto si resero conto che tale schema ripetitivo poteva essere conservato in un angolo del programma e richiamato con un solo nome da molti punti diversi del programma principale.

Nacque così il sottoprogramma, come venne denominato questo incapsulamento di istruzioni. Esso era chiaramente un ottimo modo per risparmiare memoria del computer, una risorsa molto preziosa a quei tempi. In seguito tuttavia alcuni si resero conto che il sottoprogramma permetteva di risparmiare anche memoria umana, in quanto rappresentava un'unità concettuale che una persona poteva (nell'uso pratico) considerare e manipolare come una singola linea.

L'incapsulamento nella programmazione a oggetti ha una finalità simile a quella del sottoprogramma, anche se strutturalmente è più sofisticato.

Definizione 4 (Incapsulamento orientato agli oggetti) *L'incapsulamento orientato agli oggetti si ha quando le operazioni e gli attributi che rappresentano lo stato vengono raccolti in un tipo di oggetto in modo che lo stato sia accessibile o modificabile solo attraverso l'interfaccia fornita dall'incapsulamento.*

Un oggetto è costituito da un insieme di operazioni e da un insieme di attributi, come mostrato nelle Figure 3 e 4. Per esempio, un oggetto di tipo Robot ha delle operazioni come:

- `giraASinistra()` , che ruota a sinistra l'oggetto robot di 90 gradi a sinistra
- `avanza()` , che sposta in avanti di una posizione il robot

Ogni operazione è una procedura o funzione che normalmente risulta visibile agli altri oggetti, il che significa che può essere richiamata da altri oggetti.

Gli attributi rappresentano le informazioni che un oggetto si ricorda, e solo le operazioni di un oggetto possono accedere a essi e aggiornarli. In altre parole, nessun altro oggetto può accedere a un attributo accedendo direttamente alle variabili sottostanti che lo implementano; esso dovrà, infatti, ricorrere a una delle operazioni messe a disposizione dall'oggetto.

Dal momento che solo le operazioni dell'oggetto possono leggere e aggiornare gli attributi di quest'ultimo, queste operazioni formano un anello di protezione attorno al nucleo centrale delle variabili implementate all'interno dell'oggetto (rappresentazione circolare della Figura 3).
Per esempio, l'operazione `getDirezione()` offre ai clienti esterni di un oggetto della classe `Direzione` la possibilità di ottenere (nella forma di una stringa) la direzione di quest'ultimo: non esiste alcun modo per accedere direttamente alla struttura dati usata dall'oggetto per memorizzare questa informazione.

La struttura di un oggetto pertanto assomiglia a una città medievale, circondata da mura difensive con delle porte ben sorvegliate che servivano a regolare l'ingresso e l'uscita dalla città.

E' opportuno precisare che molti linguaggi a oggetti consentono ai programmatori di designare ciascun attributo e operazione come pubblico (e pertanto visibile agli altri oggetti) o privato (visibile solo all'interno dell'oggetto).

2 Occultamento delle informazioni e dell'implementazione

Un'unità incapsulata può essere analizzata sia dall'esterno (vista pubblica) che dall'interno (vista privata); la ricompensa di un buon incapsulamento è l'eliminazione, nella vista pubblica, del-

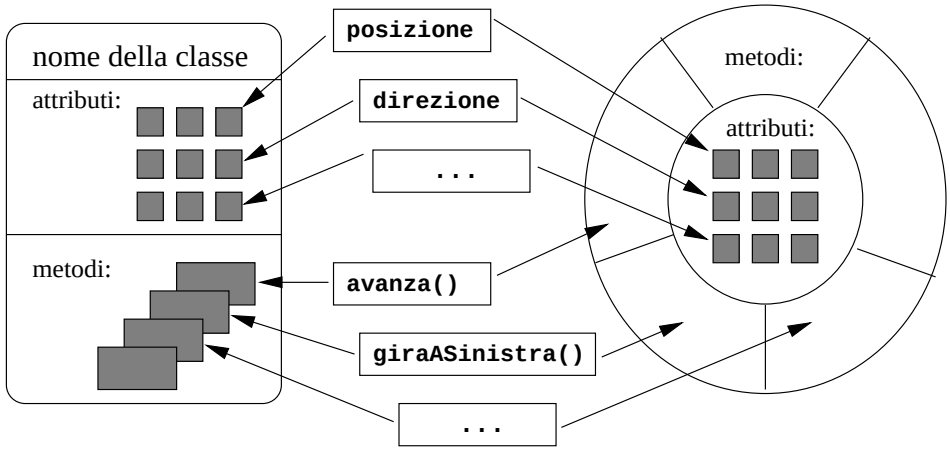


Figura 3: *Le operazioni e gli attributi di un oggetto Robot (notazione informale, la seconda figura evidenzia il concetto di incapsulamento).*

Robot
-posizione : Posizione -direzione : Direzione
+New() : Robot +giraASinistra() +avanza() +getPosizione() : Posizione +setPosizione(_posizione:Posizione) +getDirezione() : Direzione +setDirezione(_direzione:Direzione) +giraADestra()

Figura 4: *Le operazioni e gli attributi di un oggetto Robot (notazione UML).*

la miriade di dettagli che caratterizzano la vista privata. Questa eliminazione assume due forme: occultamento delle informazioni (information hiding) e occultamento dell'implementazione (implementation hiding).

L'espressione occultamento delle informazioni significa che le informazioni all'interno dell'unità non possono essere percepite all'esterno di quest'ultima, mentre l'espressione occultamento dell'implementazione significa che i dettagli dell'implementazione interni dell'unità non possono essere colti dall'esterno.

Definizione 5 (Occultamento dell'informazione) *L'occultamento delle informazioni e/o dell'implementazione è l'uso dell'incapsulamento per limitare la visibilità esterna di certe informazioni o decisioni di implementazione che sono interne alla struttura di incapsulamento.*

Un oggetto di tipo `Robot` offre un esempio di occultamento delle informazioni per il fatto che contiene alcune informazioni private inaccessibili dall'esterno, per esempio la direzione in cui il robot è rivolto. Dall'esterno dell'oggetto possiamo modificare queste informazioni (magari attraverso l'operazione `giraAsinistra()` o attraverso l'operazione `setDirezione()`), ma non esiste un meccanismo per poter accedere direttamente al loro.

Tuttavia l'espressione occultamento delle informazioni indica solo parte di ciò che un buon incapsulamento può nascondere: questo infatti spesso rivela le informazioni ma occulta l'implementazione. Questo aspetto è fondamentale per l'orientamento agli oggetti: la variabile all'interno di un oggetto che mantiene le informazioni fornite da un attributo non deve necessariamente essere implementata nello stesso modo dell'attributo, che è disponibile per altri oggetti.

Per esempio, sebbene l'operazione `stampa()` offra ai clienti esterni di un oggetto della classe `Posizione` la possibilità di produrre in output le coordinate di quest'ultimo (nella forma `[x,y]`), non sappiamo come esso memorizzi internamente queste informazioni. Potrebbero essere memorizzate sotto forma di coppia di coordinate cartesiane `(xCoord, yCoord)` , ma anche invertendo gli assi x e y come `(yCoord, xCoord)` , o come coordinate polari, o per mezzo di un ingegnoso meccanismo concepito dal progettista del software nel corso di una notte insonne. A patto che l'oggetto esporti la sua posizione in una forma accettabile da noi come clienti, non ci dobbiamo preoccupare del modo in cui esso memorizza questa informazione.

Pertanto gli oggetti dela classe `Posizione` rappresentano un esempio di occultamento sia delle informazioni sia dell'implementazione. Come ulteriore esempio, non sappiamo se l'informazione mantenuta internamente dagli oggetti di tipo `Direzione` sia detenuta come angolo numerico (con valori da O a 359 gradi), come singolo carattere (con i valori N, E, S e O), o come `percentualeDirezione` , che esprime la direzione in cui è rivolto il robot come percentuale di un cerchio completo da O a 99,999.

In una futura riprogettazione potremmo decidere di rivelare le informazioni di direzione e di fornire un'operazione per esportare l'attributo direzione ad altri oggetti: anche in questo caso tuttavia manterremmo nascosta l'implementazione perché continueremo a non avere alcuna necessità di sapere se l'implementazione interna all'oggetto è la stessa dell'informazione pubblica.

Per esempio, potremmo decidere che l'oggetto direzione deve detenere l'informazione interna sotto forma di caratteri e, dopo averlo convertito, esportarlo pubblicamente sotto forma di angolo. In altre parole, l'operazione che fornisce il valore di questo attributo potrebbe convertirlo da una rappresentazione interna arbitraria a un angolo espresso in forma numerica, che risulta accettabile alla maggior parte delle persone che desiderano accedere all'attributo direzione.

L'occultamento delle informazioni e dell'implementazione è una tecnica molto efficace per mantenere il controllo sulla complessità del software. Ciò significa che un oggetto appare come una scatola nera all'osservatore esterno, il quale, in altre parole, ha una conoscenza completa di ciò che

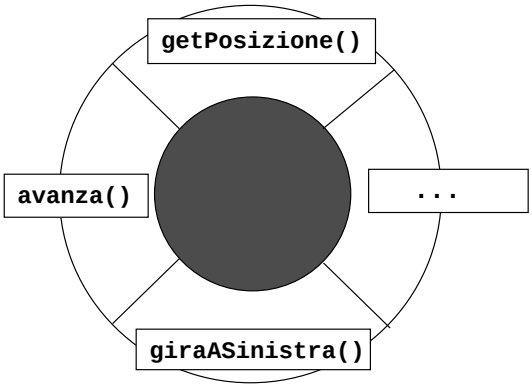


Figura 5: *L'oggetto robot visto come una scatola nera (per semplicità, vengono presentate solo alcune delle operazioni della classe* `Robot` *.*

l'oggetto può fare, ma non sa assolutamente nulla di come l'oggetto possa farlo o di come esso sia costruito internamente. La Figura 5 è un'illustrazione schematica di quanto appena detto.

La Figura 6 mostra invece una "vista privata" delle quattro classi coinvolte nel progetto "Robot". Questa vista delle classi è possibile solo per il progettista (colui che ha progettato le strautture dati interne delle classi); l'utilizzatore delle classi avrà a disposizione solamente la "vista pubblica" (cf. Figura 2).

La possibilità di occultare le informazioni e l'implementazione presenta due vantaggi principali:

1. Localizza le decisioni di progettazione. Le decisioni di progettazione private (quelle all'interno di un oggetto) hanno scarso impatto, se mai ne hanno, sul resto del sistema, pertanto possono essere prese e modificate influendo solo minimamente sul sistema nel suo insieme. Ciò limita l'effetto domino delle modifiche.

2. Disaccoppia il contenuto delle informazioni dalla loro forma di rappresentazione. In tal modo, nessun utente di informazioni che siano esterne a un oggetto può essere legato a un qualunque formato interno particolare usato per tali informazioni. Ciò evita che gli utenti esterni di un oggetto, per esempio gli altri programmatori, possano mettere le mani al suo interno. Inoltre, impedisce a programmatori con cattive intenzioni di introdurre connessioni instabili a un oggetto, che dipendono dalle particolarità del formato.

3 Conservazione dello stato

La terza astrazione dell'orientamento agli oggetti riguarda la capacità dell'oggetto di conservare il proprio stato. Quando un modulo procedurale tradizionale (funzione, sottoprogramma, procedura e così via) restituisce il controllo al proprio chiamante senza alcun effetto collaterale, il modulo cessa di esistere, lasciando solo il suo risultato come lascito. Quando lo stesso modulo viene nuovamente richiamato, è come se nascesse per la prima volta, perché non ha alcuna memoria di qualunque cosa gli sia accaduta nelle vite precedenti; anzi, non ha la benché minima idea di aver avuto un'esistenza precedente[1].

Un oggetto come il robot, invece, è consapevole del suo passato: mantiene le informazioni al suo interno per un periodo di tempo indefinito. Per esempio, potrebbe accadere che il chiamante

[1]Qui non si considerano meccanismi particolari come le variabili `static` del C++

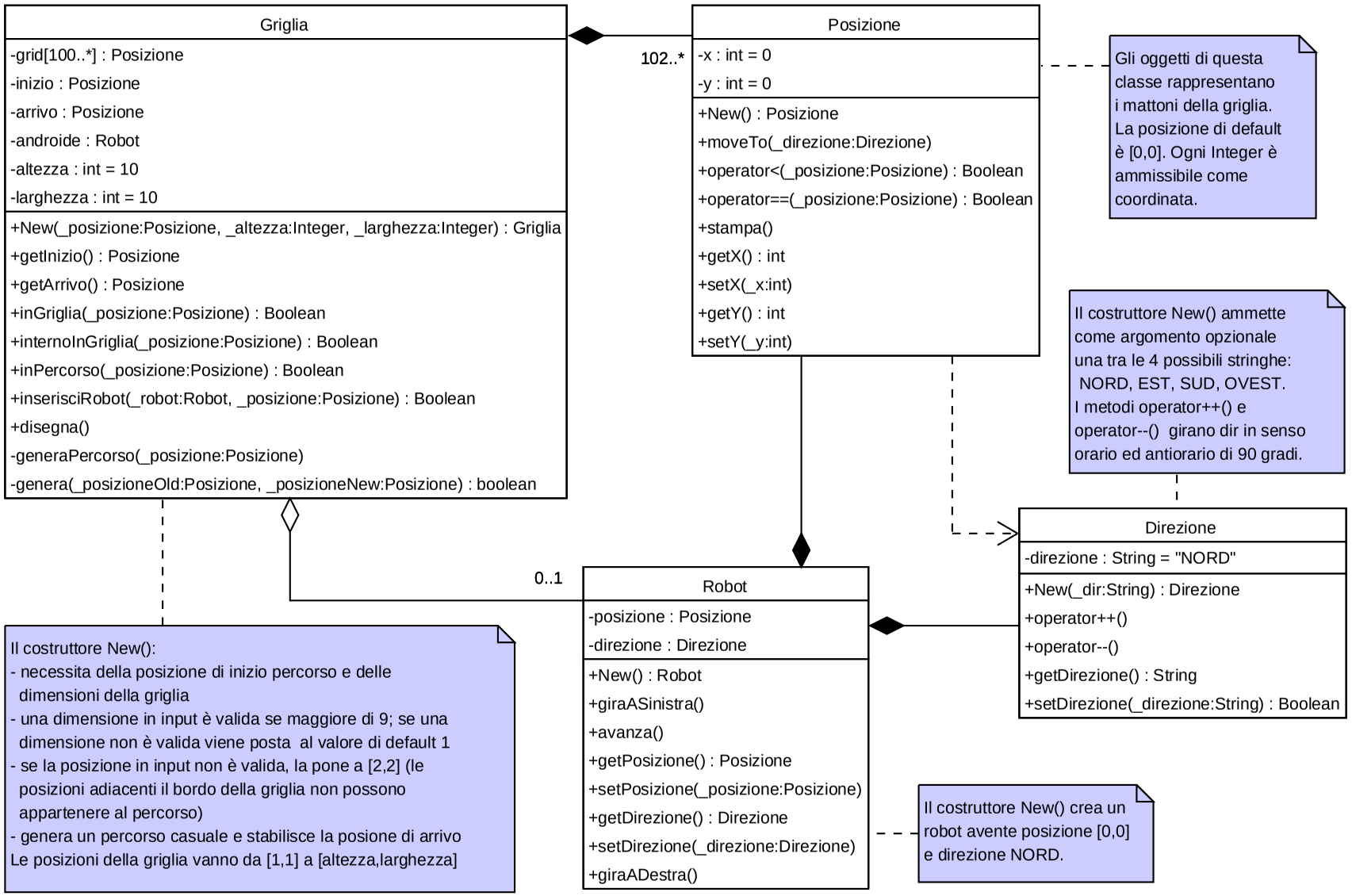


Figura 6: *"Vista privata" delle classi del progetto Robot. Sono visibili i dettagli interni definiti dal progettista delle classi.*

di un oggetto dia a quest'ultimo una particolare informazione e che poi lo stesso chiamante, o un altro, chieda all'oggetto di offrire nuovamente tale informazione. In altre parole, un oggetto non muore al termine della sua esecuzione, ma rimane in attesa di tornare nuovamente in azione.

Con un'espressione tecnica diciamo pertanto che un oggetto conserva il proprio stato (stato in effetti significa l'insieme dei valori detenuti internamente da un oggetto tramite i propri attributi). Per esempio, il robot conserva indefinitamente la conoscenza della posizione su cui si trova e della direzione in cui è rivolto: nelle sezioni refsec:incapsulamento e 2 abbiamo visto però che il modo in cui l'oggetto sceglie di conservare queste conoscenze è solo un suo problema interno.

L'incapsulamento, l'occultamento delle informazioni e dell'implementazione e la conservazione dello stato costituiscono il nucleo fondamentale dell'orientamento agli oggetti, pur non essendo di certo idee nuove: sono molti anni che in tutto il mondo esperti di informatica studiano questi concetti con il termine di tipi di dato astratti (TDA). Tuttavia l'orientamento agli oggetti va molto più in là dei TDA, come rivelano le altre sei proprietà della tecnologia a oggetti descritte nelle prossime sezioni.

4 Identità degli oggetti

La prima proprietà dell'orientamento agli oggetti che va oltre il concetto di tipo di dato astratto è di importanza fondamentale: tutti gli oggetti hanno la propria identità.

Definizione 6 (Identità degli oggetti) *L'identità degli oggetti è la proprietà per cui ciascun oggetto (a prescindere dalla sua classe o dallo stato corrente) può essere identificato e trattato come una distinta entità software.*

Ogni oggetto ha una caratteristica unica che lo distingue dai suoi simili: questa caratteristica viene offerta dal meccanismo della maniglia (handle) dell'oggetto. La maniglia è nota formalmente come identificatore di oggetto (OID, Object IDentifier); la maggior parte degli ambienti a oggetti crea automaticamente questo OID. La creazione di un oggetto può essere spiegata analizzando la seguente linea di codice:

```
var a1: Robot := Robot.new();  // creazione di un oggetto di tipo Robot
```

La parte destra di questa istruzione crea un nuovo oggetto (della classe `Robot`), visibile nella Figura 7. Osservate la maniglia all'oggetto, che nell'esempio della figura è il numero 602237: si tratta di un identificatore (OID) collegato a un oggetto al momento della sua creazione. Sono due le regole che si applicano alle maniglie:

1. Un oggetto mantiene la stessa maniglia per tutta la sua vita, indipendentemente da ciò che può accadere all'oggetto in questo tempo.

2. Due oggetti non possono avere la stessa maniglia. Ogni volta che il sistema crea un nuovo oggetto in fase di esecuzione, esso gli assegna una maniglia diversa da tutte le altre maniglie passate, presenti e future. Pertanto è sempre possibile distinguere due oggetti, anche se sono identici in struttura e nelle informazioni che contengono, perché avranno maniglie diverse.

La parte sinistra della linea di codice è la dichiarazione `var a1: Robot` . Si tratta di una normale dichiarazione che assegna un nome che abbia un significato per il programmatore (in questo caso `a1`) a un'insieme di locazioni di memoria che, per esempio, possono detenere un valore. Il termine

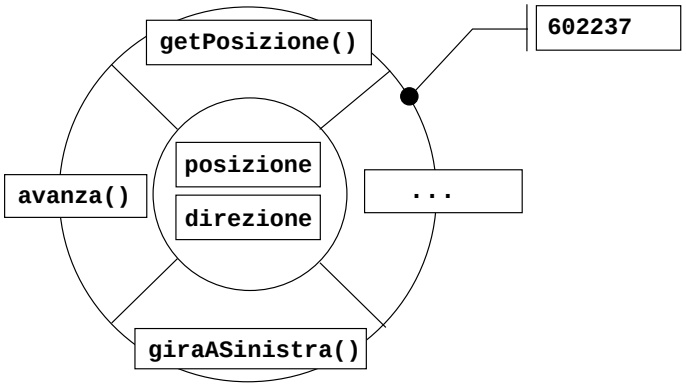


Figura 7: *Un oggetto con la sua maniglia.*

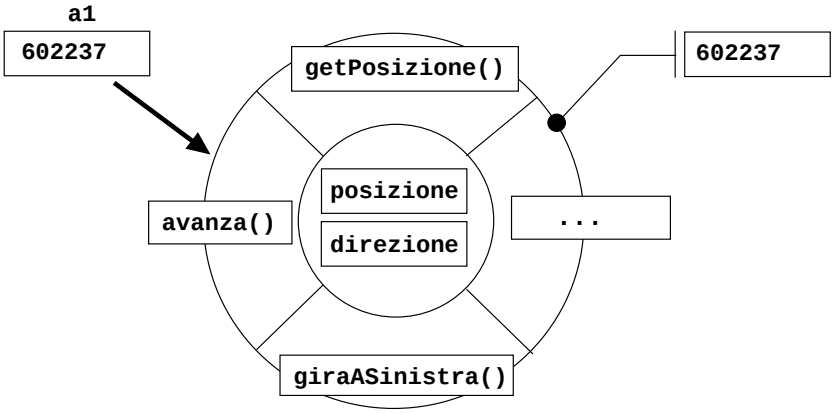


Figura 8: *`a1` punta all'oggetto con maniglia 602237.*

`Robot` in questo caso è il nome della classe di `a1` , argomento che tratteremo nella sezione 6. Il codice precedente fa in modo che la variabile a1 contenga la maniglia dell'oggetto creato.

Nessuno (programmatore, utente o chiunque altro) vedrà mai effettivamente la maniglia del nuovo oggetto (602237), a meno che non frughi nella memoria con un debugger. Il programmatore invece accederà all'oggetto tramite la variabile a1, alla quale egli stesso ha dato il nome. In altre parole, a1 punta all'oggetto che ha come maniglia 602237, come illustrato nella Figura 8.

Alcuni ambienti orientati agli oggetti utilizzano l'indirizzo di memoria fisico dell'oggetto come sua maniglia. Pur trattandosi di una soluzione semplice, potrebbe rivelarsi un grande errore qualora l'oggetto venga spostato nella memoria o nel caso in cui venga scaricato nell'area del disco dedicata alla memoria virtuale. E' pertanto meglio che una maniglia sia un valore senza alcun significato particolare, casuale ma unico. Supponiamo ora di dover eseguire anche la seguente linea di codice:

```
var a2:  Robot := Robot.New();
```

Questa linea creerà un altro oggetto (anch'esso della classe Robot) con una maniglia, per esempio, avente il valore 142857, per poi memorizzarla nella variabile a2 (Figura 9).

Per maggiore chiarezza, scriveremo un'altra istruzione di assegnamento:

```
a2 := a1;
```

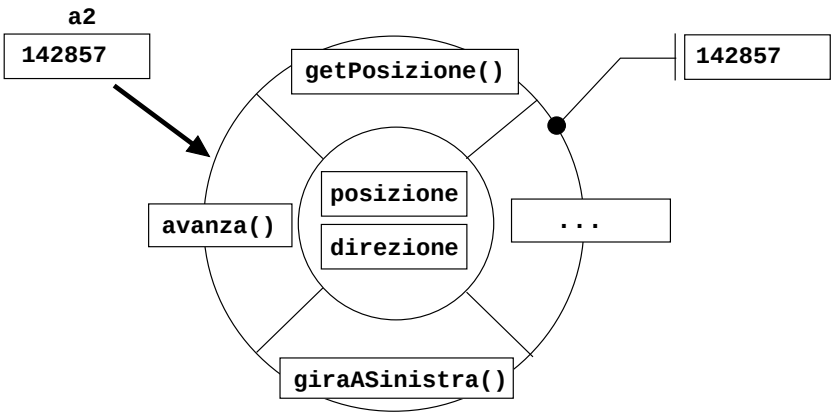


Figura 9: *a2 punta all'oggetto con maniglia 142857.*

Ora le variabili `a1` e `a2` puntano entrambe allo stesso oggetto (il primo che abbiamo creato, l'istanza di `Robot` con la maniglia 602237): fate riferimento alla Figura 10. Avere due variabili che puntano allo stesso oggetto è di solito inutile; peggio ancora, ora non abbiamo alcun modo di raggiungere il secondo oggetto (quello che ha la maniglia 142857). Di fatto, pertanto, quell'oggetto è scomparso, proprio come se fosse caduto in un buco nero. In pratica però esso non scompare realmente. la maggior parte degli ambienti orientati agli oggetti in questa circostanza utilizzerebbe un garbage collector per rimuovere l'oggetto dalla memoria.

L'idea di attribuire a tutti gli oggetti una loro identità per mezzo di una maniglia sembra innocua, anche nella peggiore delle ipotesi. Sorprendentemente, tuttavia, questa semplice idea provoca un profondo cambiamento nel modo in cui progettiamo e costruiamo il software a oggetti, come vedremo nella prossima sezione.

5 Messaggi

Un oggetto chiede a un altro oggetto di svolgere un'attività tramite un messaggio. Inoltre, molti messaggi trasferiscono informazioni da un oggetto a un altro. La maggior parte degli elenchi citati all'inizio del documento tratterebbero certamente i messaggi come una delle proprietà fondamentali.

Definizione 7 (Messaggio) *Un messaggio è il veicolo tramite il quale un oggetto mittente `ogg1` recapita a un oggetto destinatario `ogg2` una richiesta affinché quest'ultimo applichi uno dei suoi metodi.*

In questa sezione vengone descritte l'anatomia di un messaggio, le caratteristiche degli argomenti dei messaggi, il ruolo di un oggetto che invia un messaggio, il ruolo di un oggetto che riceve un messaggio e tre tipologie di messaggi.

5.1 Struttura dei messaggi

Un messaggio è costituito da diversi elementi sintattici, ognuno dei quali è importante di per sé nella progettazione a oggetti. Affinché l'oggetto `ogg1` invii un messaggio sensato all'oggetto `ogg2` , l'oggetto `ogg1` deve conoscere tre cose:

1. La maniglia di `ogg2` (ovviamente quando inviate un messaggio, dovete sapere a chi sta andando).

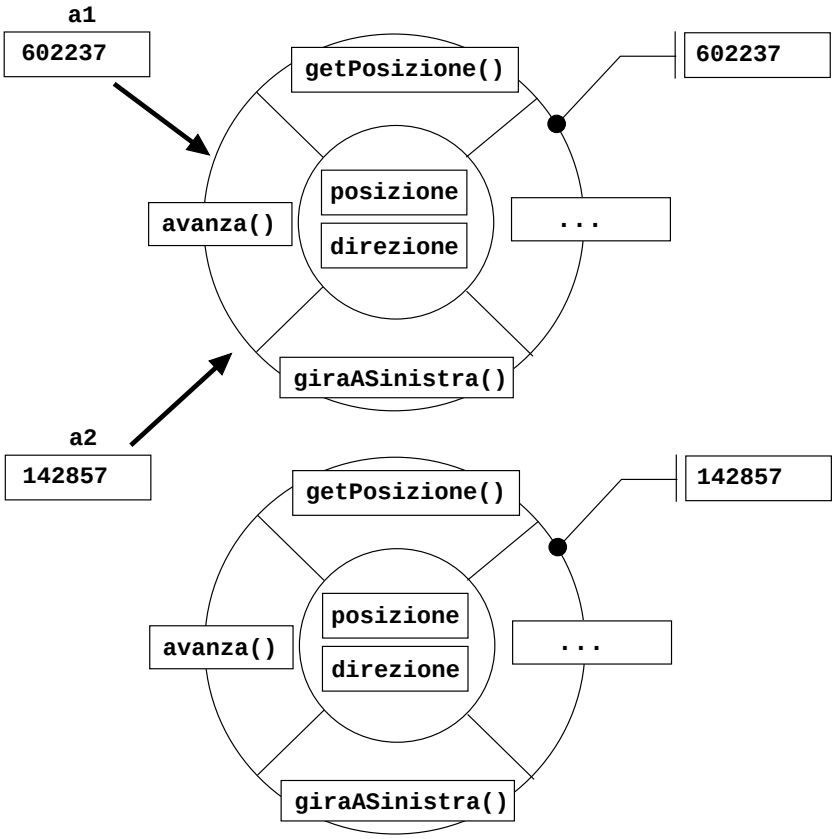


Figura 10: *a1 e a2 puntano entrambe allo stesso oggetto e l'altro oggetto non è più raggiungibile.*

2. Il nome dell'operazione di `ogg2` che `ogg1` desidera eseguire.

3. Qualunque informazione supplementare (parametro) che sarà necessaria a `ogg2` per eseguire la sua operazione.

L'oggetto che invia il messaggio (`ogg1` nell'esempio precedente) viene chiamato mittente (*sender*) mentre l'oggetto che riceve il marcatore (`ogg2`) prende il nome di destinatario (*target*) – altri termini utilizzati per mittente e destinatario sono rispettivamente *client* e *server*. Il software del robot offre diversi esempi di messaggi, uno dei quali è il seguente:

```
a1.avanza();
```

In questa istruzione, `a1` punta all'oggetto destinatario del messaggio (ossia ne contiene la maniglia); se vi ricordate, la maniglia è stata assegnata ad a1 con l'istruzione di assegnamento `var a1 : Robot = Robot.New()` , mentre `avanza()` è il nome dell'operazione (appartenente all'oggetto destinatario) che deve essere eseguita (questo messaggio non ha bisogno di parametri, perché `avanza()` sposta il robot sempre di una posizione).

L'invio di un messaggio equivale alla tradizionale chiamata a funzione o procedura; per esempio, in un linguaggio procedurale avremmo potuto scrivere

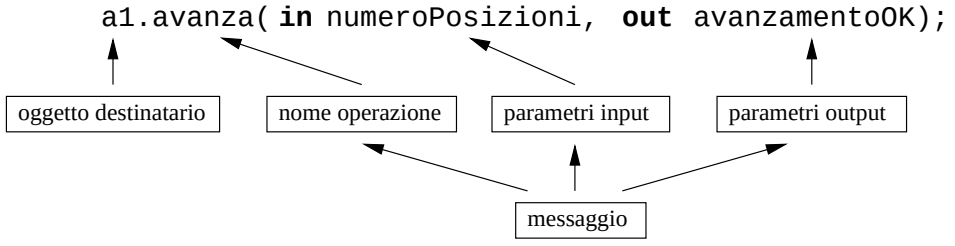
```
call avanza (a1);
```

osservate però l'inversione: con le tecniche software tradizionali ricorriamo a un'entità procedurale e le forniamo l'oggetto sul quale agire; con l'impostazione a oggetti, invece, ricorriamo a un oggetto, che poi esegue una delle sue entità procedurali.

A questo punto questa distinzione sembra semplicemente sintattica, o quanto meno filosofica. Tuttavia, quando parleremo (nella sezione 8) di polimorfismo, sovraccarico e legame dinamico, vedremo che questa enfasi sul concetto l'oggetto per primo, la procedura per seconda porta a un'importante differenza pratica tra la struttura a oggetti e la struttura tradizionale. Ciò avviene perché diverse classi di oggetti possono utilizzare lo stesso nome di operazione per operazioni che mettono in atto diversi comportamenti specifici della classe o comportamenti simili ma con nomi diversi.

5.2 Parametri dei messaggi

Come avveniva con i vecchi sottoprogrammi, la maggior parte dei messaggi passa parametri avanti e indietro. Per esempio, se modificassimo l'operazione `avanza()` facendo in modo che l'operazione restituisca un indicatore (flag) contenente il risultato dell'avanzamento, allora avremmo:



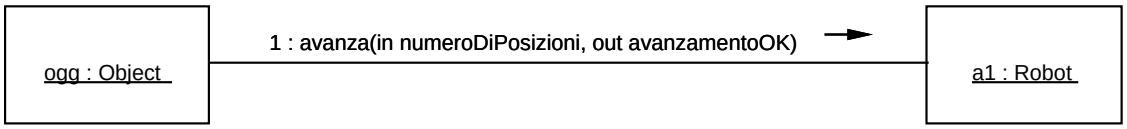


Figura 11: *Il messaggio a1.avanza(in numeroPosizioni, out avanzamentoOK), espresso con la notazione UML.*

Pertanto la struttura di un messaggio a un oggetto destinatario viene definita dalla **segnatura** (o **prototipo**) dell'operazione del destinatario da invocare, che è costituita da tre parti: il nome dell'operazione, l'elenco dei parametri di input (con prefisso `in`) e l'elenco degli parametri di output (con prefisso `out`). Entrambi gli elenchi di parametri potrebbero essere vuoti; per brevità, normalmente si omette la parola chiave `in` , considerandola predefinita.

I parametri di un messaggio riflettono un altro fondamentale contrasto tra il software orientato a oggetti e quello tradizionale. In un ambiente a oggetti puro (come quello di Smalltalk), i parametri del messaggio non sono dati, bensì maniglie di oggetti. Gli argomenti dei messaggi sono oggetti vivi!

Per esempio, la Figura 11 illustra un messaggio tratto dal programma del robot, espresso con la notazione UML.

Se dovessimo fare un'istantanea del programma del robot nel momento in cui sta eseguendo questo messaggio ed estrarre i valori dei parametri del messaggio, troveremmo qualcosa di inatteso. Per esempio, potremmo trovare:

- `numeroDiOuadrati` impostato a 123432

- `avanzamentoOK` impostato a 664730

Perché questi strani numeri? Perché 123432 potrebbe essere la maniglia dell'oggetto (della classe `Integer`) che potremmo considerare come l'intero 2, mentre 664730 potrebbe essere la maniglia dell'oggetto (della classe `Boolean`) che normalmente considereremmo come valore logico `true` .

Per fare un altro esempio, se dovessimo eseguire un sistema di gestione del personale orientato agli oggetti e facessimo la stessa cosa, potremmo trovare l'argomento `impDelMese` impostato a 441523, che potrebbe essere la maniglia dell'oggetto (della classe `Impiegato`) che rappresenta Giacomo Colombo.

5.3 I ruoli degli oggetti nei messaggi

In questa sezione analizzo i tre ruoli possibili che abbiamo indicato per un sistema orientato agli oggetti. Un oggetto può essere:

1. mittente di un messaggio;

2. destinatario di un messaggio;

3. riferito da una variabile all'interno di un altro oggetto.

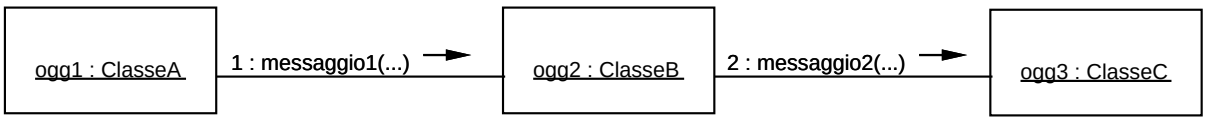


Figura 12: *Due messaggi tra due coppie di oggetti.*

Un dato oggetto può assumere uno o più di questi ruoli nel corso della sua esistenza. Dalla Figura 12 si evince chiaramente che un oggetto non è mai un mittente nato o un destinatario nato. Infatti, per messaggiol(), `ogg1` è il mittente e `ogg2` è il destinatario; per `messaggio2()` , invece, `ogg2` è il mittente e `ogg1` è il destinatario. Vediamo pertanto che in diversi momenti lo stesso oggetto può recitare entrambe le parti. I termini mittente e destinatario sono pertanto relativi a un dato messaggio e non sono proprietà fisse degli oggetti stessi.
Un ambiente orientato agli oggetti puro contiene solo oggetti che giocano uno o più dei tre ruoli visti sopra. Nella tecnologia a oggetti pura non c'è alcun bisogno di dati, perché gli oggetti possono fare tutto il lavoro software necessario ai dati; e in Smalltalk (un linguaggio a oggetti molto puro), non c'è proprio alcun dato! Al momento dell'esecuzione ci sono solo oggetti che puntano ad altri oggetti (tramite variabili) e comunicano reciprocamente passandosi avanti e indietro maniglie di altri oggetti ancora.

Invece, in C++ (che è un linguaggio misto, per il fatto che utilizza dati e funzioni tradizionali insieme agli oggetti), gli argomenti possono essere puntatori a qualunque cosa. Se il codice C++ è puro come in Smalltalk, tutti i parametri saranno puntatori a oggetti. Se invece si mescolano oggetti e dati in un programma, alcuni dei parametri possono essere semplicemente dei dati (o puntatori a dati). Un commento simile è valido per il codice Java, sebbene quest'ultimo sia un linguaggio molto meno a ruota libera rispetto al C++.

5.4 Tipi di messaggi

Un oggetto può ricevere tre tipi di messaggi: informativi, interrogativi e imperativi. In questa sezione definiamo brevemente e offriamo un esempio di ogni tipo di messaggio, ancora una volta ricorrendo a codice relativo al controllo del robot.

Definizione 8 (Messaggio informativo) *Un messaggio informativo è un messaggio a un oggetto che fornisce a quest'ultimo delle informazioni per aggiornarsi (è noto anche come messaggio di aggiornamento, di inoltro o push). E un messaggio orientato al passato per il fatto che in genere informa l'oggetto di ciò che è già avvenuto altrove.*

Un esempio di messaggio informativo è:

```
impiegato.sposato(dataMatrimonio:  Data)
```

questo messaggio dice a un oggetto, che rappresenta un dipendente, che il dipendente in questione si è sposato in una certa data. In generale, un messaggio informativo comunica a un oggetto qualcosa che è accaduto nella parte di mondo reale rappresentata da quell'oggetto.

Definizione 9 (Messaggio interrogativo) *Un messaggio interrogativo è un messaggio a un oggetto che richiede a quest'ultimo di rivelare alcune informazioni su di sé (è noto anche come messaggio di lettura, retrospettiva o pulI). È un messaggio orientato al presente per il fatto che chiede all'oggetto di comunicare informazioni correnti.*

Un esempio di messaggio interrogativo è:

```
a1.posizione()
```

che chiede al robot di comunicare la sua posizione corrente sulla griglia. Questo tipo di messaggio non modifica nulla, essendo in genere un'interrogazione relativa a quella parte di mondo rappresentata dall'oggetto destinatario.

I metodi invocati tramite messaggi informativi e interrogativi sono di solito rispettivamente definiti mediante la seguente sintassi:

```
getNomeattributo():  Classe
setNomeattributo ( _nomeattributo:  Classe )
```

dove `Nomeattributo` è il nome di un attributo (appartenente alla classe `Classe`). In genere, per un attributo possono essere definiti entrambi i metodi, e vengono brevemente indicati come metodi `set` e metodi `get` .

Definizione 10 (Messaggio imperativo) *Un messaggio imperativo è un messaggio a un oggetto che richiede a quest'ultimo di portare a termine qualche azione su se stesso, su un altro oggetto o persino sull'ambiente attorno al sistema (è noto anche come messaggio di forzatura o di azione). È un messaggio orientato al futuro per il fatto che chiede all'oggetto di compiere qualche azione nell'immediato futuro.*

Un esempio di messaggio imperativo è

```
a1.avanza()
```

che provoca lo spostamento in avanti del robot. Questo tipo di messaggio spesso si traduce nell'esecuzione da parte dell'oggetto destinatario di qualche algoritmo che gli consenta di fare quanto richiesto.

I sistemi in tempo reale orientati agli oggetti, nei quali gli oggetti controllano componenti hardware, spesso contengono molti messaggi imperativi. Questi messaggi illustrano chiaramente lo spirito rivolto al futuro di un messaggio imperativo. Considerate questo esempio tratto dal mondo della robotica:

```
manoSinistraRobot.vaiAPosizione (x,y,z:  Lunghezza, thetal,theta2,theta3:  Angolo)
```

Questo messaggio stabilisce la posizione e l'orientamento nello spazio della mano sinistra di un robot. L'algoritmo potrebbe richiedere lo spostamento della mano del robot, del suo braccio e/o quello del robot stesso. I sei argomenti rappresentano i sei gradi di libertà della mano, un'entità tridimensionale nello spazio. A questo punto ci spostiamo dai messaggi a un'altra proprietà indiscutibilmente fondamentale per l'orientamento agli oggetti, la classe di oggetti.

6 Classi

Nel software di controllo del robot abbiamo creato un oggetto (che rappresenta un robot) eseguendo `Robot.New()` . La classe `Robot` è servita da modello per creare gli oggetti robot (come quello con maniglia 602237). Ogni volta che eseguiamo l'istruzione `Robot.New()` , istanziamo un oggetto che è strutturalmente identico a tutti gli altri oggetti creati dall'istruzione `Robot.New()` , dove per

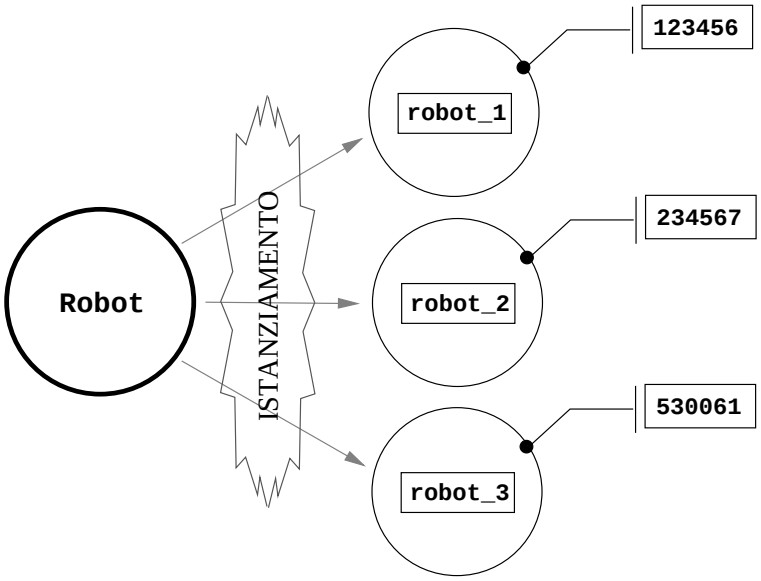


Figura 13: *Tre oggetti istanziati dalla stessa classe (* `Robot` *).*

strutturalmente identico intendiamo dire che ogni oggetto robot ha le stesse operazioni e variabili, in particolare quelle che il programmatore ha codificato quando ha scritto la classe `Robot` (Figura 13).

Definizione 11 (Classe) *Una classe è la sagoma a partire dalla quale vengono creati (istanziati) gli oggetti. Ogni oggetto ha la stessa struttura e comportamento della classe dalla quale è istanziato. Se l'oggetto* `ogg` *appartiene alla classe* `C`*, diciamo che* `ogg` *è un'istanza di* `C`*.*

Vi sono due differenze tra gli oggetti di una stessa classe: ogni oggetto ha una maniglia diversa e, in qualunque momento particolare, ogni oggetto probabilmente avrà uno stato diverso (il che significa diversi valori memorizzati nelle sue variabili). Dato che inizialmente la distinzione tra classe e oggetto può creare una certa confusione, suggeriamo queste semplici definizioni, che possono aiutare a fare un po' di chiarezza.

- Una classe è ciò che progettate e programmate.
- Gli oggetti sono ciò che create (a partire da una classe) in fase di esecuzione.

Pacchetti software molto noti presentano una stretta analogia con classi e oggetti. Supponiamo che acquistiate un programma per foglio di calcolo chiamato Visigoth 5.0 prodotto dalla Wallisoft Corp (fondata dallo stesso Wally Soft). Il pacchetto in sé sarebbe un analogo della classe, mentre i fogli di calcolo veri e propri che create a partire da esso sarebbero simili agli oggetti. Ogni foglio di calcolo ha a disposizione tutti i meccanismi di foglio di calcolo in quanto istanza della classe Visigoth.

In fase di esecuzione, una classe come `Robot` può generare 3, 300 o 3.000 oggetti (ossia istanze di `Robot`). Una classe pertanto assomiglia a una sagoma: una volta ritagliata, da essa è possibile ricavare la stessa forma migliaia di volte, ottenendo oggetti tutti identici tra loro e, ovviamente, uguali alla forma della sagoma originale. Per essere ancora più chiari, analizziamo più da vicino la popolazione di oggetti generati da una singola classe: come abbiamo visto, tutti gli oggetti di una classe hanno la stessa struttura, ossia lo stesso insieme di operazioni e di attributi. Pertanto

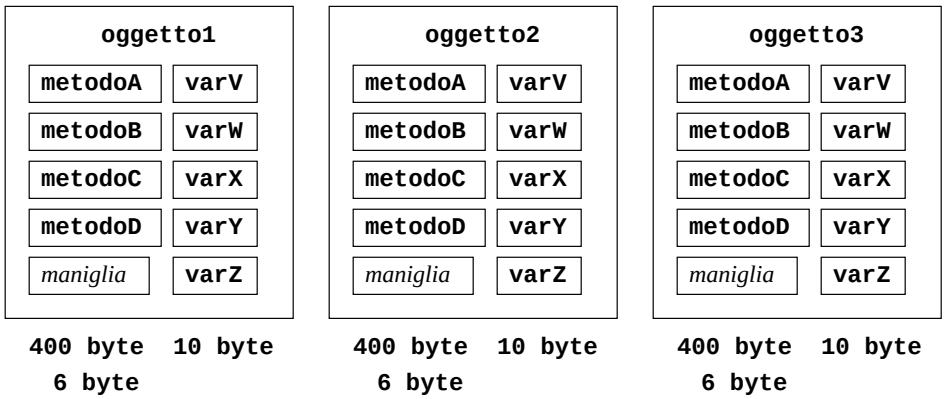


Figura 14: *I metodi, le variabili e le maniglie per tre oggetti della stessa classe, insieme con i requisiti di memoria degli oggetti (approccio con spreco di risorse).*

ogni oggetto (istanza) di una classe ha la propria copia dell'insieme di metodi che gli occorrono per implementare le operazioni e l'insieme di variabili necessarie per implementare gli attributi. In linea di principio, in un dato momento ci sono tante copie dei metodi e delle variabili (0, 3, 300,...) quanti sono gli oggetti istanziati in quel momento (Figura 14).

Al fine di spiegare ulteriormente la reale struttura di un insieme di oggetti della stessa classe, che chiameremo `C`, analizziamo brevemente i dettagli dell'implementazione. Supponiamo che ogni metodo che implementa una delle operazioni della Figura 14 occupi 100 byte, che ogni variabile occupi 2 byte e che per ogni maniglia occorrano 6 byte. In base a ciò, oggetto i occuperà 416 byte di memoria (ossia 100 * 4 + 5 * 2 + 6 * 1). I tre oggetti insieme pertanto occuperanno 1.248 byte di memoria (ossia 3 * 416).

Questo approccio all'allocazione della memoria per gli oggetti provocherebbe però un notevole spreco di risorse, perché ognuno dei tre insiemi di metodi dei tre oggetti è identico; e dal momento che ciascun insieme di metodi contiene solo codice procedurale, tutti gli oggetti possono condividere un solo insieme. Quindi, sebbene in linea di principio ogni oggetto abbia il proprio insieme di metodi per l'implementazione delle operazioni, in pratica (per risparmiare spazio) condividono tutti la stessa copia fisica.

Di converso, sebbene le maniglie e le variabili di ciascun oggetto siano identiche in struttura da un oggetto all'altro, esse non possono essere condivise tra più oggetti per l'ovvio motivo che devono contenere diversi valori in fase di esecuzione.

Così, sebbene gli oggetti della classe `C` condividano tutti lo stesso insieme di operazioni, la memoria totale consumata dai tre oggetti di `C` sarà effettivamente di 448 byte (400 byte per il singolo insieme di metodi, 30 byte per i 3 set di variabili e 18 byte per le 3 variabili). Questo consumo di soli 448 byte è inferiore ai 1.248 byte della soluzione nativa e rappresenta il modo normale in cui gli ambienti a oggetti allocano la memoria per gli oggetti (Figura 15). Ovviamente il risparmio cresce con il crescere del numero di oggetti istanziati: se con soli tre oggetti si consuma il 35,9% della memoria (ovvero 448 byte rispetto ai previsti 1.248 totali), con 300 oggetti il consumo scende al 4,1% (5.200 byte rispetto a 124.800).

Quasi tutte le operazioni e gli attributi analizzati in questa sezione appartengono a singoli oggetti e prendono il nome di **operazioni di istanza** e **attributi di istanza**. Tuttavia esistono anche operazioni di classe e attributi di classe. Per definizione esiste sempre esattamente un insieme di operazioni di classe e di attributi di classe per una data classe, indipendentemente dalla quantità di oggetti di quella classe che possono essere stati istanziati. Le operazioni e gli attributi di classe

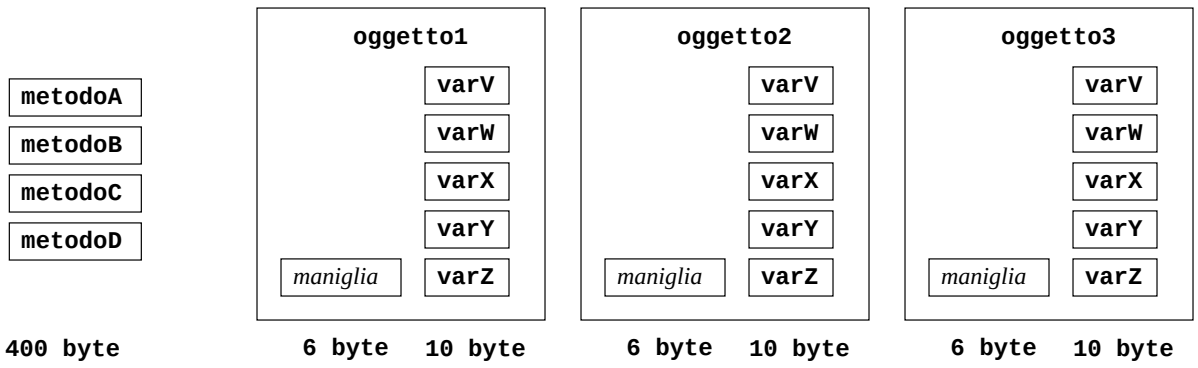


Figura 15: *Illustrazione schematica della memoria effettiva (448 byte) utilizzata da 3 oggetti della stessa classe.*

sono necessari per far fronte alle situazioni che non possono essere responsabilità di un qualunque oggetto singolo: l'esempio più famoso di un'operazione di classe è `New()` , che istanzia un nuovo oggetto di una data classe.

Il messaggio `New()` non potrebbe mai essere inviato a un singolo oggetto. Supponiamo, per esempio, di avere tre oggetti della classe `ClienteBanca` , che rappresentano tre effettivi clienti di una banca (faremo riferimento a questi oggetti con i nomi `bob` , `carol` e `ted`). Supponiamo inoltre che desideriamo istanziare un nuovo oggetto dalla classe `ClienteBanca` (per esempio `alice`). A quale oggetto invieremmo il messaggio `New()` ? Non ci sarebbe nessun particolare motivo per inviarlo a `bob` piuttosto che a `carol` o a `ted` ; peggio ancora, non avremmo mai potuto istanziare il primo cliente della banca, in quanto inizialmente non ci sarebbe stato nessun oggetto della classe `ClienteBanca` al quale inviare il messaggio `New()` .

Quindi, `New()` è un messaggio che deve essere inviato a una classe, invece che a un singolo oggetto. L'esempio presente nel gioco del robot era `Robot.New()` , un messaggio di classe inviato alla classe `Robot` per ottenere l'esecuzione della sua operazione di classe `New()` e creare così un nuovo oggetto, una nuova istanza della classe `Robot` .

Un esempio di attributo di classe potrebbe essere `numeroDiRobotCreati: Integer` , che verrebbe incrementato da `New()` della classe `Robot` a ogni esecuzione di questa operazione. Indipendentemente dal numero di oggetti robot, ci sarebbe solo una copia di questo attributo di classe. Potreste progettare un'operazione di classe per offrire al mondo esterno la possibilità di accesso a questo attributo di classe.

La Figura 16 illustra la struttura della memoria nel caso in cui la classe `C` abbia due operazioni di classe (i metodi delle quali occupano 100 byte ciascuno) e tre attributi di classe (le variabili dei quali occupano 2 byte ciascuna). Il numero di byte per il meccanismo di classe (206 in questo esempio) rimarrà costante a prescindere dal numero di oggetti che `C` ha istanziato. Con l'aggiunta di questo meccanismo di classe, `C` e il suo gregge di dodici oggetti ora occupa un totale di 798 (ossia 206 + 592) byte di memoria.

Osservate che, sia in linea di principio sia in pratica, c'è un solo insieme di metodi di classe per classe: questo è in contrasto con i metodi di istanza, dove in linea di principio ogni oggetto ha il proprio insieme (solo per risparmiare memoria facciamo in modo che gli oggetti condividano lo stesso insieme di metodi per le loro operazioni). La distinzione tra variabili di classe e variabili di istanza è quindi più chiara: ogni classe ha solo un insieme di variabili di classe, mentre esiste un insieme di variabili di istanza per ogni oggetto della classe, sia in linea di principio sia di fatto.

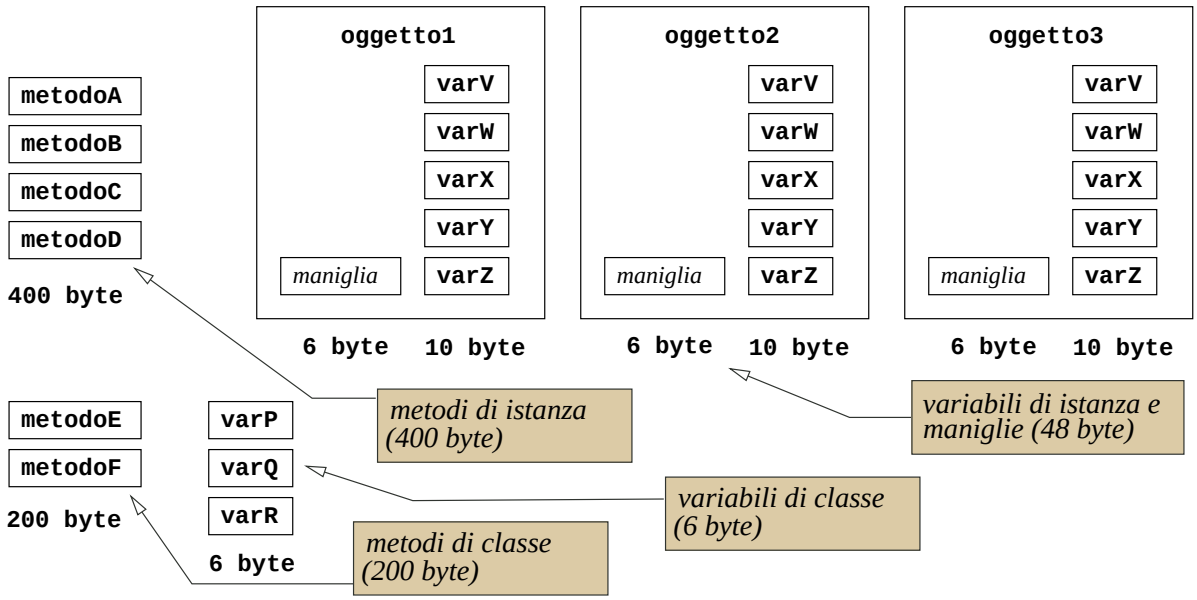


Figura 16: *Illustrazione schematica della memoria effettiva (846 byte) utilizzata da 3 oggetti e dal meccanismo di classe*

Quale è allora la differenza tra una classe e un TDA? La risposta è che un TDA descrive un'interfaccia: è la facciata che dichiara cosa verrà fornito agli utenti del TDA, senza però dire alcunché sul modo in cui il TDA verrà implementato. Una classe è una cosa in carne e ossa (o almeno dotata di disegno interno e codice) che implementa un TDA. In effetti per un dato TDA si potrebbero progettare e costruire diverse classi: per esempio, una di queste classi potrebbe produrre oggetti molto efficienti in esecuzione, mentre da un'altra si potrebbero ottenere oggetti che occupano poca memoria.

6.1 Composizione ed Aggregazione di classi

Quando in una classe si includono oggetti appartenenti ad altre classi si realizza il concetto di associazione di tipo tutto/parte tra classi. Due sono i tipici modi di realizzare questa associazione: composizione e aggregazione. Nel primo caso l'oggetto interno esiste solo per poter realizzare la parte dell'oggetto esterno, mentre nel secondo caso l'oggetto interno esiste indipendentemente del suo ruolo di parte. Spieghiamo in dettaglio, analizzando brevemente anche la notazione UML che permette di rappresentare le due situazioni.

Composizione: La composizione è una struttura comune nei sistemi software, siano essi o meno orientati agli oggetti, perché nella vita di ogni giorno abbiamo a che fare con molti oggetti composti. Per esempio, un messaggio di posta elettronica è un composto che contiene un'intestazione e alcuni paragrafi di testo. A sua volta, l'intestazione è composta dal nome del mittente, da quello del destinatario, dall'oggetto del messaggio e da altre informazioni specifiche del sistema di trasmissione elettronica dei messaggi. Ora però, prima di continuare, è necessario fare qualche precisazione terminologica. L'associazione tutto/parte prende il nome di composizione, dove il tutto viene chiamato (oggetto) composto, mentre la parte prende il nome di (oggetto) componente. Ecco le tre caratteristiche più importanti della composizione.

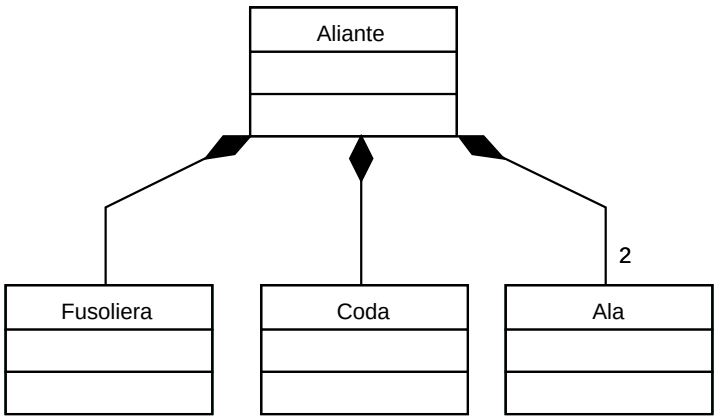


Figura 17: *Un oggetto composto ed i suoi componenti.*

1. L'oggetto composto non esiste senza i suoi componenti. Per esempio, rimuovete le setole, il manico e le piccole parti in gomma da uno spazzolino da denti, e non avrete più uno spazzolino da denti. In effetti è sufficiente rimuovete le setole da uno spazzolino da denti perché diventi difficile qualificarlo come tale. In questo senso diciamo che la vita di un oggetto composto non può andare oltre quella dei suoi componenti.

2. In qualunque momento, ciascun oggetto componente dato può essere parte di un solo composto.

3. La composizione tipicamente è eterogenea, nel senso che è molto probabile che i com- ponenti siano di tipi misti: alcune ruote, alcuni assi, dei pezzi di legno... ed ecco un carro.

L'oggetto composto della Figura 17 rappresenta un aliante semplificato formato da quattro componenti: una fusoliera, una coda, un'ala sinistra e un'ala destra. Come ulteriore esempio, vedi la Figura 2, dove un oggetto robot è progettato come aggregato di una `Posizione` e di una `Direzione` .
Analizziamo questa figura per capire il funzionamento di UML.

1. Un'associazione tra l'oggetto composto e ciascuno dei suoi componenti appare sul diagramma come una linea di associazione, con un piccolo rombo nero collocato all'estremità accanto all'oggetto composto.

2. La classe del composto, `Aliante` , appare a un'estremità della linea di associazione, mentre all'altra estremità appare la classe di ciascun componente, `Fusoliera` , `Coda` e `Ala` . Osservate che un componente come `Ala` può limitarsi a comparire una sola volta nel diagramma; il numero componenti di tipo `Ala` (la sua molteplicità) compare numericamente sulla linea di associazione, vicino alla classe contenuta.

3. E' necessario indicare la molteplicità all'estremità del componente di ciascuna linea di associazione. Se la molteplicità all'estremità del composto non viene indicata, allora si presuppone che sia esattamente 1.

4. La linea di associazione non ha nome, che è la norma sia per la composizione sia per l'aggregazione. Il motivo è che raramente, nel caso di un'associazione di composizione, un nome

aggiunge un significato che vada oltre quello di tutto/parte già indicato dalla simbologia. Le forme verbali come ha, comprende, consiste e così via non aggiungono nulla al modello.

Per quanto concerne l'implementazione, all'interno della classe `Aliante` possono essere dichiarate le seguenti variabili:

```
fusoliera: Fusoliera;
coda: Coda;
alaSinistra: Ala;
alaDestra: Ala;
```

Quando un oggetto della classe `Aliante` , chiamiamolo `aliante1` , viene istanziato e inizializzato, la variabile `coda` punterà a un oggetto che rappresenta la coda di `aliante1` . Analogamente, le variabili `fusoliera` , `alaSinistra` e `alaDestra` contengono le maniglie degli altri componenti di un oggetto della classe `Aliante` . Questa implementazione supporta la navigabilità da un oggetto composto ai suoi oggetti componenti.

La composizione spesso è legata alla propagazione dei messaggi. Per esempio, per spostare un simbolo di rettangolo su uno schermo, potreste chiedere all'oggetto rettangolo di spostare se stesso. A sua volta, il rettangolo potrebbe inviare un messaggio a ciascuno dei suoi segmenti componenti e dire loro di muoversi. Analogamente, per trovare il peso di una sedia, potreste inviare un messaggio a ciascun componente della sedia richiedendone il peso.

Aggregazione: Come la composizione, l'aggregazione è un costrutto ben noto, per mezzo del quale i sistemi rappresentano strutture tratte dal mondo reale. Per esempio, una città è un aggregato di case, una foresta è un aggregato di alberi e un gregge è un aggregato di pecore. In altre parole, l'aggregazione e un associazione gruppo/membri. Ancora una volta occorre qualche precisazione terminologica. L'associazione prende il nome di aggregazione, dove il tutto viene chiamato (oggetto) aggregato, mentre la parte prende il nome di (oggetto) costituente. Le tre caratteristiche più importanti dell'aggregazione sono indicate di seguito.

1. L'oggetto aggregato potenzialmente può esistere senza i suoi oggetti costituenti. Per esempio, un dipartimento continua a esistere anche nel caso in cui vengano licenziati tutti i suoi dipendenti.

2. In qualunque momento, ciascun oggetto può essere costituente di più di un aggregato. Ancora una volta un aggregato reale potrebbe anche non avvalersi di questa proprietà.

3. L'aggregazione tende a essere omogenea, il che significa che gli oggetti costituenti di un tipico aggregato apparterranno alla stessa classe. Per esempio, i costituenti di un paragrafo sono le frasi, mentre i costituenti di una foresta sono tutti gli alberi.

Vediamo ora la notazione UML per rappresentare il costrutto di aggregazione. La Figura 18 mostra una cassa formata da bottiglie di birra (per un ulteriore esempio, vedi anche la relazione tra le classi `Griglia` e `Robot` nella Figura 2). Su questa figura è possibile fare le seguenti osservazioni.

1. Un'associazione tra un aggregato e i suoi costituenti viene indicata da un piccolo rombo vuoto sulla linea di associazione all'estremità dell'aggregato.

2. Le classi dell'aggregato (`Cassa`) e dei costituenti (`BottiglieBirra`) appaiono alle rispettive estremità della linea di associazione.

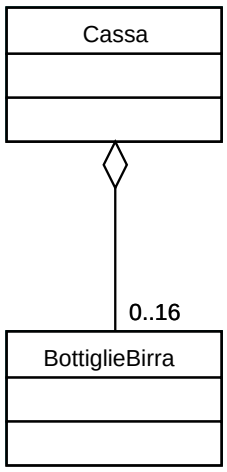


Figura 18: *Un oggetto aggregato ed i suoi costituenti.*

3. Con l'aggregazione è necessario indicare la molteplicità a entrambe le estremità della linea di associazione, perché non è mai possibile ipotizzarla, come invece avviene nel caso della composizione. La molteplicità all'estremità dell'aggregato della Figura 18 è 1, il che significa che una bottiglia di birra può appartenere ad una cassa, oppure non appartenere a nessuna cassa. La molteplicità all'estremità del costituente è 0..16, il che significa che una cassa può includere fino a 16 bottiglie di birra, o anche nessuno (nel caso di una cassa vuota).

Come avviene con la composizione, è possibile implementare l'aggregazione per mezzo di variabili. Per la navigabilità dall'aggregato ai costituenti, una variabile nell'aggregato punterà ai costituenti. Per esempio, la classe `Cassa` può contenere la seguente dichiarazione:

```
bottiglie: BottiglieBirra[16];
```

che nel nostro pseudo-linguaggio potrebbe denotare una variabile array di 16 componenti di tipo `BottiglieBirra` .

7 Ereditarietà

Cosa fare nel caso fosse necessario scrivere il codice di una classe `C` e, allo stesso tempo, fosse disponobile una classe `D` quasi identica a `C` tranne per alcuni attributi e operazioni aggiuntivi? La semplice soluzione di duplicare tutti gli attributi e le operazioni di `C` per collocarli in `D` , oltre a comportare del lavoro aggiuntivo, renderebbe seccante il lavoro di manutenzione. Una soluzione migliore è far sì che la classe `D` in qualche modo, chieda di utilizzare le operazioni della classe `C` : questa soluzione prende il nome di ereditarietà.

Definizione 12 (Ereditarietà) *L'ereditarietà (di una classe `D` da una classe `C`) è il meccanismo tramite il quale `D` ha implicitamente definito su di essa ciascuno degli attributi e delle operazioni della classe `C` come se tali attributi e operazioni fossero stati definiti per `D` stessa. `C` viene definita superclasse di `D` , mentre `D` è una sottoclasse di `C` .*

In altre parole, attraverso l'ereditarietà gli oggetti della classe `D` possono utilizzare attributi e operazioni che altrimenti sarebbero disponibili solo agli oggetti della classe `C` . L'ereditarietà

rappresenta un'altra delle caratteristiche principali grazie alle quali la tecnologia a oggetti si distacca dagli approcci dei sistemi tradizionali; essa infatti permette effettivamente di costruire il software in modo incrementale distinguendo due fasi fondamentali.

Fase 1: Innanzi tutto si costruiscono le classi destinate a far fronte alle situazioni più generali.

Fase 2: Poi, per affrontare i casi particolari, si aggiungono classi più specializzate che ereditano dalle classi generali. Una classe specializzata avrà pertanto il diritto di utilizzare tutte le operazioni e gli attributi (operazioni e attributi sia di classe sia di istanza) della classe originale.

Un esempio può essere di aiuto per illustrare questo principio. Supponiamo di avere, in un'applicazione aeronautica, una classe `Velivolo` che può aver definita un'operazione di istanza denominata `vira()` e un attributo di istanza di nome `rotta` .

La classe `Velivolo` ha a che fare con tutta l'attività e le informazioni pertinenti a qualunque tipo di apparecchio volante; tuttavia esistono tipi speciali di velivolo che svolgono speciali attività e pertanto richiedono informazioni particolari. Per esempio, un aliante svolge attività speciali (come sganciare il cavo di rimorchio) e potrebbe dover registrare speciali informazioni (per esempio, se è attaccato a un cavo di rimorchio).

Ecco allora che possiamo definire un'altra classe, `Aliante` , che eredita da `Velivolo` e avrà un'operazione di istanza chiamata `sganciacavoRimorchio()` e un attributo di istanza di nome `seCavoRimorchioAttaccato` (di classe `Boolean`). Questo ci dà la struttura mostrata nella Figura 19, nella quale la freccia bianca denota l'ereditarietà.

Ora analizziamo i meccanismi dell'ereditarietà immaginando un po' di codice a oggetti che crea oggetti delle classi `Velivolo` e `Aliante` e in seguito gli invia dei messaggi. Il codice è seguito da un'analisi delle quattro istruzioni contrassegnate da (1) a (4).

```
var v: Velivolo :=  Velivolo.New();
var a: Aliante  :=  Aliante.New();
v.vira(nuovaRotta, out viraoK);      (1)
a.sganciaCavoRimorchio();            (2)
a.vira(nuovaRotta, out viraoK);      (3)
v.sganciaCavoRimorchio();            (4)
```

(1) L'oggetto cui punta `v` riceve il messaggio `vira(nuovaRotta, out viraOK)` , che fa sì che esso applichi l'operazione `vira()` (con i parametri opportuni). Dal momento che `v` è un'istanza di `Velivolo` , esso utilizzerà semplicemente l'operazione `vira()` che è stata definita nella classe `Velivolo` .

(2) L'oggetto cui punta `a` riceve il messaggio `sganciaCavoRimorchio()` , che fa si che esso applichi l'operazione `sganciaCavoRimorchio()` (che non richiede argomenti). Dal momento che `a` è un'istanza di `Aliante` , esso utilizzerà semplicemente l'operazione `sganciaCavoRimorchio()` che è stata definita nella classe `Aliante` .

(3) L'oggetto cui punta `a` riceve il messaggio `vira(nuovaRotta, out viraOK)` , che fa sì che esso applichi l'operazione `vira()` (con gli argomenti opportuni). Senza ereditarietà, questo messaggio provocherebbe un errore in fase di esecuzione (come `vira(): operazione non definita`) perché `a` è un'istanza di `Aliante` , che non ha alcuna operazione denominata

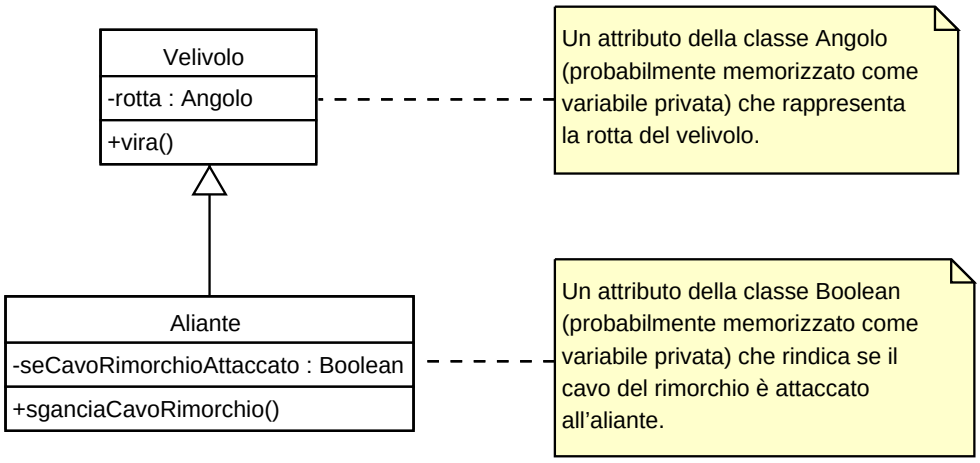


Figura 19: *`Aliante` è una sottoclasse che eredita dalla sua superclasse, `Velivolo` .*

`vira()` . Tuttavia, dato che `Velivolo` è una superclasse di `Aliante` , anche l'oggetto `a` può utilizzare a pieno titolo qualunque operazione di `Velivolo` (se `Velivolo` avesse una superclasse `OggettoVolante` , `a` potrebbe anche utilizzare qualunque operazione di questa classe). Pertanto la linea di codice contrassegnata con (3) funzionerà senza problemi e l'operazione `vira()` , così come è stata definita per `Velivolo` , verrà eseguita.

(4) Questo non funzionerà! `v` si riferisce a un'istanza di `Velivolo` , che non ha alcuna operazione denominata `sganciaCavoRimorchio()` . L'ereditarietà non è di alcun aiuto in questo caso, dal momento che `Aliante` è l'unica classe che definisce al suo interno l'operazione `sganciaCavoRimorchio()` e `Aliante` è una sottoclasse di `Velivolo` . Dato che l'ereditarietà non funziona in questa direzione, il sistema si blocca e segnala un errore di esecuzione. Se ci pensiamo bene, comunque, ciò ha un senso, perché `v` potrebbe puntare a un oggetto che rappresenta un grande aereo a reazione, per il quale `sganciaCavoRimorchio()` non avrebbe significato.

Nella sezione 6 abbiamo visto la distinzione tra classe e oggetto; ora invece vedremo che esiste anche una sottile distinzione tra oggetto e istanza. Sebbene finora abbiamo utilizzato i termini oggetto e istanza quasi come sinonimi, vedremo che l'ereditarietà in un certo senso permette a un singolo oggetto di essere contemporaneamente un'istanza di più di una classe. Ciò corrisponde bene a quanto accade nel mondo reale. Se possedete un aliante, avete esattamente un oggetto con un'identificazione (maniglia). Tuttavia questo aliante è (ovviamente) un esempio di aliante e allo stesso tempo un esempio di velivolo. Dal punto di vista concettuale, quindi, l'oggetto che rappresenta la cosa che possedete è un'istanza di `Aliante` e un'istanza di `Velivolo` .

Effettivamente l'esempio precedente costituisce un test rivelatore per un valido uso dell'ereditarietà, che prende il nome di test è un (is-a). Se potete dire: un `D` è un `C` , allora `D` quasi certamente deve essere una sottoclasse di `C` . Quindi, dato che un aliante è un velivolo, la classe `Aliante` deve essere una sottoclasse di `Velivolo` . Analizziamo ulteriormente la questione gettando uno sguardo a ciò che avviene dietro le quinte dell'ereditarietà. L'oggetto cui fa riferimento `a` sarà rappresentato in fase di esecuzione dall'unione di due parti: una parte saranno le operazioni e gli attributi di istanza definiti per `Aliante` , l'altra le operazioni e gli attributi di istanza definiti per `Velivolo` . Nella maggior parte dei linguaggi, la sottoclasse che eredita riceve tutto ciò che la superclasse ha da offrire, senza scegliere cosa ereditare. Esistono tuttavia alcuni accorgimenti che permettono a una

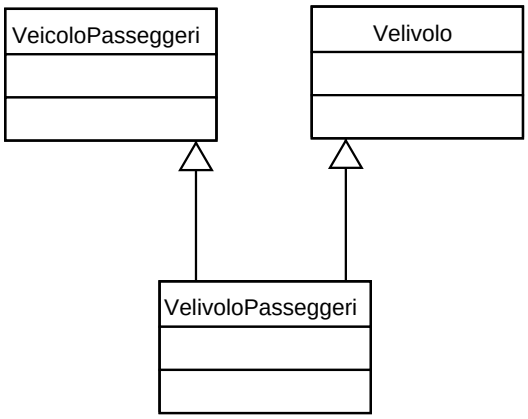


Figura 20: *Ereditarietà multipla: una sottoclasse con più superclassi.*

sottoclasse di sovrascrivere (ossia modificare) le operazioni ereditate, come vedremo nella sezione 8. Il codice per implementare effettivamente l'ereditarietà in un buon linguaggio a oggetti è semplice: è sufficiente dichiarare la superclasse nella definizione di ogni sottoclasse che deve ereditare da essa. Per esempio,

```
class Aliante inherits from Velivolo;
```

L'esempio presentato in questa sezione è di ereditarietà singola, il che significa che ogni classe ha al massimo una superclasse diretta; vi sono anche casi di ereditarietà multipla, nel qual caso ogni classe può avere un numero arbitrario di superclassi dirette. L'ereditarietà multipla trasforma la struttura ad albero dell'ereditarietà singola in un reticolo di ereditarietà, come mostrato nella Figura 20.
L'ereditarietà multipla introduce alcune difficoltà di progettazione, compresa la possibilità che una sottoclasse erediti dai suoi progenitori operazioni o attributi in contrasto tra loro (le operazioni in contrasto hanno lo stesso nome e la sottoclasse che eredita non riesce a decidere facilmente quale ereditare). Difficoltà come il problema del contrasto di nomi hanno dato all'ereditarietà multipla una cattiva reputazione. Nel corso degli anni sia la condanna sia la difesa dell'ereditarietà multipla hanno raggiunto livelli parossistici. Comunque, dal momento che l'ereditarietà multipla può creare strutture complesse e poco comprensibili, deve essere utilizzata sensatamente, ancora di più di quanto avviene con l'ereditarietà singola. Attualmente due dei principali linguaggi a oggetti (C++ e Eiffel) permettono l'ereditarietà multipla, mentre gli altri due (Java e Smalltalk) non la accettano.

8 Polimorfismo

La parola polimorfismo deriva da due termini greci che significano rispettivamente molti e forme: una cosa polimorfa pertanto ha la proprietà di assumere molte forme. I manuali di programmazione a oggetti contengono due definizioni di polimorfismo, contrassegnate con (A) e (B) nella nota sottostante. Si tratta di due definizioni valide, ed entrambe le proprietà del polimorfismo agiscono di concerto per fornire una grande potenza alla tecnologia a oggetti. Più avanti in questa sezione analizzeremo in modo più approfondito queste due definizioni.

Definizione 13 (Polimorfismo)

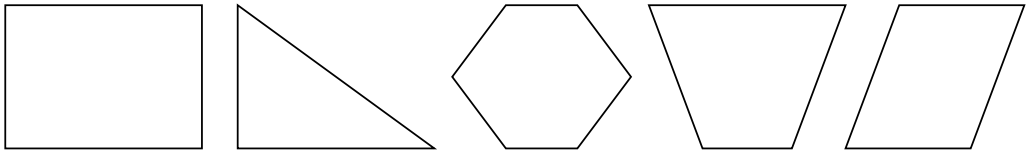


Figura 21: *Alcuni poligoni piani.*

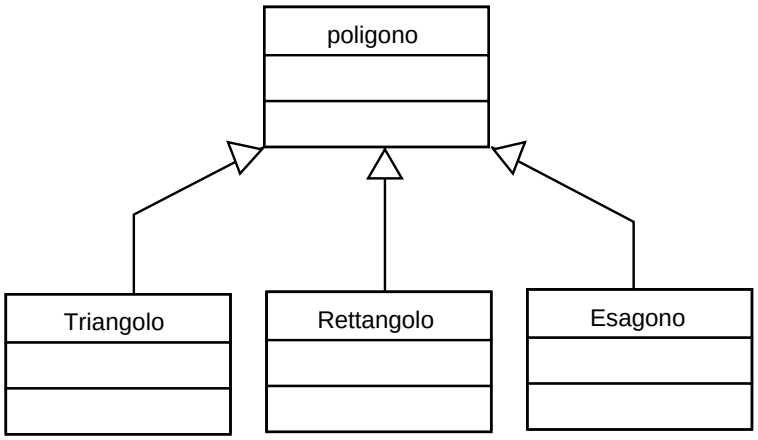


Figura 22: *`Poligono` e le sue tre sottoclassi.*

(a) Il polimorfismo è il meccanismo grazie al quale un singolo nome di operazione o di attributo può essere definito per più di una classe e può assumere diverse implementazioni in ciascuna di quelle classi.

(b) Il polimorfismo è la proprietà per mezzo della quale un attributo o una variabile può puntare a (detenere la maniglia di) oggetti di diverse classi in diversi momenti.

Supponiamo di avere una classe `Poligono` che rappresenta il tipo di forme bidimensionali illustrato dalla Figura 21. Potremmo definire un'operazione denominata `calcolaArea()` per `Poligono`, che restituisce il valore dell'area di un oggetto di tipo `Poligono` (osservate che area è un attributo definito per `Poligono` e, tramite ereditarietà, per le sue sottoclassi). L'operazione `calcolaArea()` ha bisogno di un algoritmo abbastanza sofisticato perché deve preoccuparsi dei vari poligoni di forma strana della Figura 21.

Ora aggiungiamo alcune classi, per esempio `Triangolo`, `Rettangolo` ed `Esagono`, che sono sottoclassi di `Poligono` e pertanto ereditano da essa. Ciò è accettabile perché un triangolo è un poligono, un rettangolo è un poligono e così via (Figura 22).

Osservate che nella Figura 22 anche le classi `Triangolo` e `Rettangolo` hanno delle operazioni denominate `calcolaArea()`, che svolgono lo stesso compito della versione di `calcolaArea()` presente nella classe `Poligono`, ossia calcolare l'area della superficie racchiusa dalla forma.

Il progettista/programmatore del codice che implementa `calcolaArea()` per `Rettangolo` tuttavia scriverebbe il codice in modo molto diverso da quello dell'operazione `calcolaArea()` di `Poligono`. Perché? Dato che è semplice calcolare l'area di un rettangolo (base altezza), il codice dell'operazione `calcolaArea()` di `Rettangolo` è di conseguenza semplice ed efficiente. Dal momento invece che l'algoritmo per calcolare l'area di un poligono arbitrario complesso è complicato e meno efficiente, non desideriamo utilizzarlo per calcolare l'area di un rettangolo.

Se pertanto scrivessimo del codice che invia il seguente messaggio a un oggetto riferito da `formaBidimensionale` :

```
formaBidimensionale.calcolaArea();
```

potremmo non sapere quale algoritmo per il calcolo dell'area verrà eseguito. Il motivo è che potremmo non sapere esattamente a quale classe appartiene `formaBidimensionale` . Ci sono cinque possibilità:

1. `formaBidimensionale` è un'istanza di `Triangolo` e verrà eseguita l'operazione `calcolaArea()` nella forma definita per `Triangolo` .
2. `formaBidimensionale` è un'istanza di `Rettangolo` e verrà eseguita l'operazione `calcolaArea()` nella forma definita per `Rettangolo` .
3. `formaBidimensionale` e un istanza di `Esagono` , ma dato che a questa classe manca un'operazione di nome `calcolaArea()` , tramite ereditarietà verrà eseguita l'operazione `calcolaArea()` nella forma definita per `Poligono` .
4. `formaBidimensionale` è un'istanza generale di forma arbitraria della classe `Poligono` e verrà eseguita l'operazione `calcolaArea()` nella forma definita per `Poligono` .
5. `formaBidimensionale` e un istanza di una classe `C` diversa da tutte le classi precedenti. Dal momento che probabilmente per `C` non è stata definita alcuna operazione denominata `calcolaArea()` , l'invio del messaggio `calcolaArea()` provocherà un errore di compilazione o esecuzione.

Anche se è possibile trovare strano che un oggetto possa non conoscere con precisione l'esatta classe dell'oggetto destinatario cui sta inviando il messaggio, questa situazione è abbastanza comune. Per esempio, nella linea finale del codice riportato qui sotto, al momento della compilazione non è possibile sapere a quale classe di oggetti punterà `p` in fase di esecuzione: l'oggetto effettivamente puntato sarà determinato da una scelta effettuata dall'utente all'ultimo momento (verificata dall'istruzione `if`).

```
var p: Poligono := Poligono.New();
var t: Triangolo := Triangolo.New();
var h: Esagono := Esagono.New();
...

if   "l'utente dice OK"
then   p := t
else   p := h
endif;
...

p.calcolaArea();  // qui p pu\'o fare riferimento a un oggetto
                  // Triangolo o Esagono

...
```

In questo frammento di codice a oggetti non abbiamo bisogno di un test attorno a `p.calcolaArea()` per determinare quale versione di `calcolaArea()` eseguire. Questo è un esempio di occultamento dell'implementazione molto pratico, che permette di aggiungere una nuova sottoclasse di `Poligono` (per esempio `Ottagono`) senza modificare in alcun modo il codice precedente. Per ricorrere a una metafora, l'oggetto destinatario sa come calcolare la propria area e pertanto il mittente non si deve preoccupare.

Osservate anche la dichiarazione `var p: Poligono` si tratta di una limitazione di sicurezza al polimorfismo della variabile `p` . Nella sintassi di programmazione utilizzata qui significa che a `p` è consentito puntare solo a oggetti della classe `Poligono` (o ad oggetti di una delle sottoclassi di `Poligono`). Se a `p` venisse assegnata la maniglia di un oggetto `Cliente` o `Cavallo` , il programma si interromperebbe e segnalerebbe un errore di esecuzione.

L'operazione `calcolaArea()` , essendo definita per diverse classi, offre un valido esempio di polimorfismo, così come nella definizione 13(a); la variabile `p` invece, essendo in grado di puntare a oggetti di molte classi diverse (per esempio `Triangolo` ed `Esagono`), è un buon esempio della definizione 13(b). L'esempio nel complesso mostra come i due aspetti del polimorfismo operino di concerto per agevolare il compito del programmatore.

Un ambiente a oggetti spesso implementa il polimorfismo attraverso il legame dinamico: l'ambiente verifica la classe effettiva dell'oggetto destinatario di un messaggio all'ultimo momento possibile, ossia in fase di esecuzione, quando il messaggio viene inviato.

Definizione 14 (Dynamic binding) *Il dynamic binding (legame dinamico o legame in esecuzione o legame tardivo) è la tecnica per cui l'esatto codice da eseguire viene determinato solo al momento dell'esecuzione e non in fase di compilazione.*

L'esempio precedente, in cui l'operazione `calcolaArea()` viene definita per `Poligono` e `Triangolo` , dimostra anche il concetto di sovrascrittura (overriding).

Definizione 15 (Overriding) *L'overriding (sovrascrittura) è la ridefinizione di un metodo definito per una classe `C` in una della sottoclassi di `C` .*

L'operazione `calcolaArea()` , che in origine è stata definita per `Poligono` , viene sovrascritta in `Triangolo` da un'operazione che ha lo stesso nome ma un algoritmo diverso. Occasionalmente è possibile utilizzare le tecniche di sovrascrittura per neutralizzare un'operazione definita per una classe `C` in una della sottoclassi di `C` . Ciò è possibile semplicemente ridefinendo l'operazione in modo che restituisca un errore. Se tuttavia vi trovate ad affidarvi molto a queste azioni di neutralizzazione, è probabile che siate partiti con una gerarchia di superclassi/sottoclassi traballante. Correlato al concetto di polimorfismo c'è quello di overloading (sovraccarico), che non va confuso con la sovrascrittura.

Definizione 16 (Overloading) *L'overloading (sovraccarico) di un nome o simbolo avviene quando diverse operazioni (o operatori) definite per la stessa classe hanno quel nome o simbolo. In questo caso diciamo che un nome o simbolo è sovraccaricato.*

Polimorfismo e overloading spesso richiedono che l'operazione specifica da eseguire venga scelta in fase di esecuzione. Come abbiamo visto nel codice dell'esempio precedente, il motivo e che la classe esatta dell'oggetto destinatario, e pertanto l'implementazione specifica dell'oggetto da eseguire, può non essere nota fino al momento dell'esecuzione.

La normale distinzione tra polimorfismo e overloading è che il primo consente di definire lo stesso nome di operazione in modo diverso per classi diverse, mentre con il secondo è possibile definire lo stesso nome di operazione in modo diverso per più volte all'interno della stessa classe.

L'operazione polimorfa che verrà selezionata dipende solo dalla classe dell'oggetto destinatario cui il messaggio è indirizzato. Ma nel caso di un'operazione soggetta a overloading, come avviene il legame del frammento di codice corretto con il nome dell'operazione al momento dell'esecuzione? Tramite la segnatura, ossia il numero e la classe degli argomenti, del messaggio. Ecco due esempi:

```
1a. prodotto1.ribassa()
1b. prodotto1.ribassa(altaPercentuale)
2a. matrice1 * i
2b. matrice1 * matrice2
```

Nel primo esempio, il prezzo di un prodotto è ridotto dall'operazione ribassa. Se questa operazione viene richiamata senza argomenti (come in 1a), l'operazione utilizza una percentuale di sconto standard; se invece ribassa viene invocata con un argomento (l'argomento `altaPercentuale` di 1b), allora l'operazione applica il valore fornito da `altaPercentuale` .

Nel secondo esempio l'operatore di moltiplicazione `*` è sovraccaricato. Se il secondo operando è un intero (come in 2a), allora l'operatore `*` indica una moltiplicazione scalare, mentre nel caso in cui sia un'altra matrice (come in 2b) indica una moltiplicazione tra matrici.

9 Genericità

Definizione 17 (Genericità) *La genericità è la costruzione di una classe* `C` *in modo tale che una o più delle classi che essa utilizza internamente venga fornita solo in fase di esecuzione (al momento in cui un oggetto della classe* `C` *viene istanziato).*

Illustriamo il concetto di genericità mediante un esempio. Supponiamo di dover progettare e programmare un albero binario di ricerca bilanciato contenente numeri interi (Figura 23). Un albero binario è detto di ricerca se le informazioni contenute nei nodi appartengono ad un insieme ordinato I ed inoltre, per ogni nodo x dell'albero, valgono le seguenti proprietà:

- tutte le informazioni contenute nel sottoalbero sinistro di x *precedono* (secondo l'ordinamento definito in I) l'informazione contenuta in x;
- tutte le informazioni contenute nel sottoalbero destro di x *seguono* (secondo l'ordinamento definito in I) l'informazione contenuta in x.

Inoltre, un generico albero è detto bilanciato se, per ogni nodo, la differenza fra le profondità[2] dei suoi due sottoalberi è al più 1.

Tutto ciò è molto semplice, almeno fino a quando non si deve inserire un altro intero nella struttura (per esempio 5, come nella Figura 24). A questo punto l'albero può sbilanciarsi e si è costretti a eseguire delle operazioni contorte e faticose per far riguadagnare all'albero il suo equilibrio. Queste operazioni faticose possono costituire un algoritmo di ribilanciamento. Se fosse necessario, per altri motivi, mantenere degli elenchi aggiornati di clienti e prodotti, si potrebbe recuperare il codice prodotto per la gestione di alberi di interi e copiarlo due volte, sostituendo in una copia `Integer` con `IDCliente` e nell'altra `Integer` con `IDprodotto` .

[2]La profondità di un albero è definita come la massima distanza tra la radice dell'albero ed una delle sue foglie

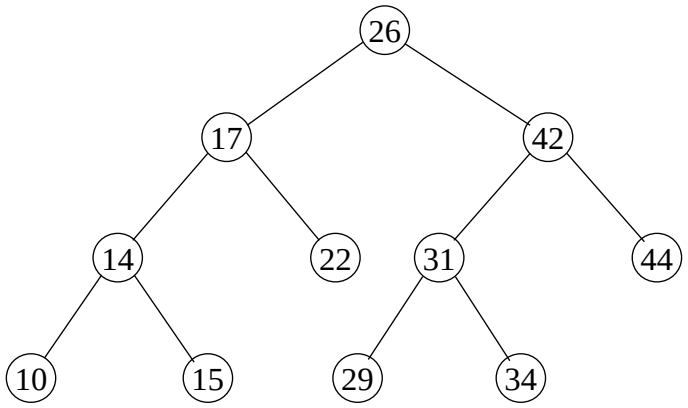


Figura 23: *Un albero binario di ricerca bilanciato (contenente numeri interi).*

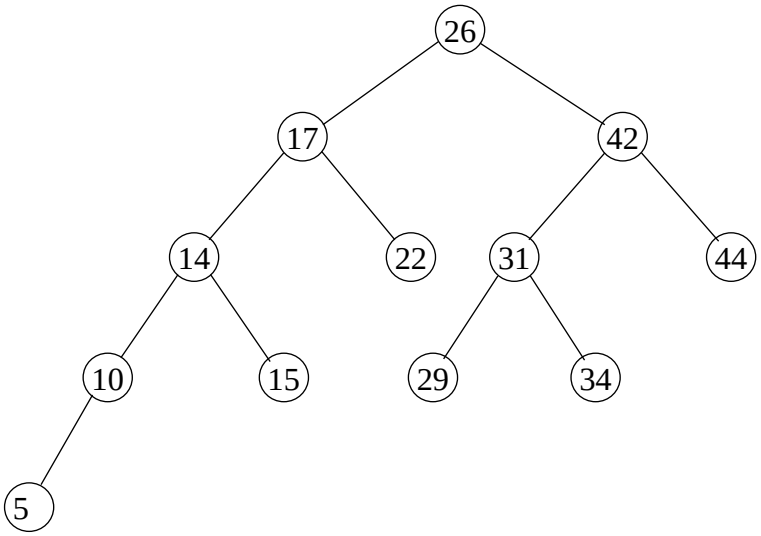


Figura 24: *L'albero di figura 23 dopo l'inserimento del numero 5. L'albero ottenuto non è più bilanciato (cf. profondità dei sottoalberi del nodo contenente il numero 17).*

Questa *donazione* del vecchio codice può aumentare notevolmente la produttività, anche se questo approccio presenta un significativo pericolo di donazione: si può essere costretti a mantenere tre copie di codice quasi identico.

Ciò significa che, nel caso in cui venga progettato un algoritmo migliore per il bilanciamento dell'albero, è necessario rivedere tre parti di codice; questa gestione di tre versioni pertanto comporta non solo del lavoro aggiuntivo, ma risulta anche complicata (a meno di non riuscire a realizzare rapidamente una procedura automatizzata per la modifica). Ciò che occorre è un modo per scrivere una sola volta la struttura fondamentale dell'algoritmo di bilanciamento dell'albero per poi poterla applicare tutte le volte che né abbiamo necessità ad interi, clienti, prodotti o qualunque altro tipo di dato.

In un caso come questo, la genericità fornisce un notevole aiuto. Se definiamo `AlberoBilanciato` come classe parametrica (in C++ le classi parametriche sono note come *template*), significa che almeno una delle classi utilizzate all'interno di `AlberoBilanciato` non deve necessariamente essere assegnata fino al momento dell'esecuzione. Questa presumibilmente sarebbe la classe degli elemen-

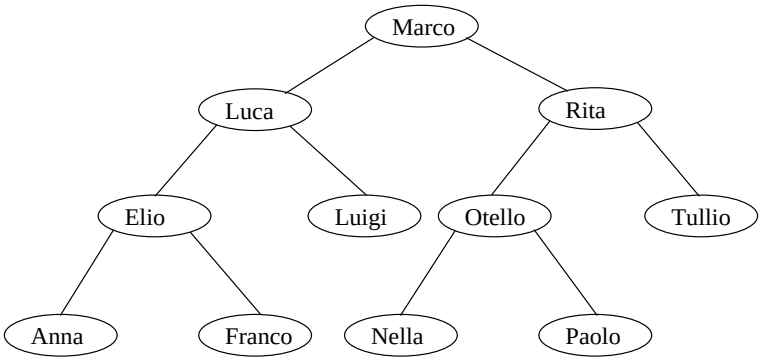


Figura 25: *Un albero binario bilanciato di oggetti Cliente (i clienti contenuti nei nodi sono rappresentati con i nomi propri ed ordinati rispetto a tali nomi). Questo albero è l'oggetto al quale punta la variabile* `alberoClienti` .

ti da memorizzare nei nodi del particolare oggetto albero bilanciato che istanziamo. E' pertanto possibile scrivere la classe `AlberoBilanciato` nel modo seguente:

```
class AlberoBilanciato< ClasseElementoNodo >;
...
var nodoCorrente: ClasseElementoNodo := ClasseElementoNodo.New();
...
nodoCorrente.print();
...
```

Osservate l'argomento `ClasseElementoNodo` della classe parametrica. Questo è un argomento formale, il cui effettivo valore verrà fornito solo in fase di compilazione/esecuzione. Per esempio, quando istanziamo un nuovo oggetto della classe `AlberoBilanciato` , forniamo un nome di classe reale come argomento, come indicato di seguito:

```
var alberoClienti: AlberoBilanciato := AlberoBilanciato.New( <Cliente> );
var alberoProd: AlberoBilanciato := AlberoBilanciato.New( <Prodotto> );
```

Pertanto `alberoClienti` ora punta a un oggetto (un'istanza di `AlberoBilanciato`) che mantiene istanze della classe `Cliente` nei suoi nodi, come mostrato nella Figura 25 (lo stesso vale per `alberoProd` , ovviamente).

E' come se avessimo donato due volte la prima parte del codice (una volta per `Cliente` , una volta per `Prodotto`) per ottenere:.

```
class AlberoClientieBilanciato;
...
var nodoCorrente: Cliente := Cliente.New();
...
nodoCorrente.print();
...
class AlberoProdottiBilanciato;
...
var nodoCorrente: Prodotto :=  Prodotto.New();
```

```
...
nodoCorrente.print();
```

Infine, osservate l'istruzione `nodoCorrente.print()` . Si tratta di un raffinato esempio di polimorfismo, perché quando scriviamo questa istruzione nella classe parametrica `AlberoBilanciato` , non sappiamo quale potrebbe essere la classe di `nodoCorrente` . Pertanto chiunque istanzi un oggetto della classe `AlberoBilanciato` deve preoccuparsi che l'operazione print venga effettivamente definita per qualunque classe che venga usata per istanziare un particolare albero.

Per fare un altro esempio, se progettate una classe parametrica `TabellaHash<C>` , dovreste far rilevare che per qualunque classe (come `Simbolo`) fornita come effettivo argomento a `C` dev'essere definita l'operazione hash.

Forse vi siete resi conto che c'è un modo per scrivere `AlberoBilanciato` senza ricorrere alla genericità e senza donazione: potremmo lasciare che i nodi di un albero bilanciato accettino un oggetto della classe di livello più alto nella gerarchia superclasse/sottoclassi. Se denominassimo `Oggetto` questa classe, allora il codice sarebbe

```
class AlberoBilanciato;
...
var nodoCorrente: Oggetto :=  Oggetto.New();
...
nodoCorrente.print();
...
```

Ora ogni nodo di un albero bilanciato accetterebbe l'inserimento di un oggetto assolutamente di qualunque tipo: in particolare, potremmo mescolare clienti, interi, prodotti, poligoni, cowboy e cavalli sullo stesso albero. Questo quasi certamente non avrebbe alcun senso; ancora peggio, sarebbe molto improbabile che tutte queste classi diverse di oggetti comprendano il messaggio `print()` .

`AlberoBilanciato` e `TabellaHash` sono entrambi esempi di classe contenitore (vedi le classi contenitore della Standard Template Library (STL) in C++), un tipo di classe che serve a contenere oggetti in qualche struttura (spesso sofisticata). La genericità viene spesso impiegata nella programmazione a oggetti per progettare queste classi contenitore. Sebbene non sia strettamente necessaria per scrivere il codice riutilizzabile delle classi contenitore, è certamente superiore al codice donato o ad una progettazione fragile in cui oggetti di classi arbitrarie vengono mescolati nello stesso contenitore.

10 Riepilogo

Dato che l'espressione orientamento agli oggetti manca di un significato a priori, storicamente c'è stato poco consenso sull'insieme di proprietà che caratterizzano questo paradigma. Vengono in genere considerate centrali le seguenti proprietà: incapsulamento, occultamento delle informazioni e dell'implementazione, conservazione dello stato, identità degli oggetti, messaggi, classi, ereditarietà, polimorfismo e genericità.

L'incapsulamento orientato agli oggetti produce una struttura software (un oggetto) costituita da un anello di operazioni protettive attorno agli attributi che rappresentano lo stato dell'oggetto; in termini di implementazione, i metodi delle operazioni manipolano delle variabili che mantengono lo stato dell'oggetto. Tale incapsulamento garantisce che qualunque cambiamento (o accesso) alle informazioni memorizzate internamente dall'oggetto avviene tramite le operazioni di quest'ultimo.

L'occultamento delle informazioni e dell'implementazione è la ricompensa di un buon incapsulamento: questo infatti permette di proteggere le informazioni locali di un oggetto e le decisioni di progettazione relative all'implementazione dell'oggetto dagli sguardi e dalle intromissioni dall'esterno.

La conservazione dello stato è la proprietà per cui un oggetto può conservare indefinitamente le informazioni, anche negli intervalli tra le diverse attivazioni delle sue operazioni.

L'identità degli oggetti offre a tutti gli oggetti un'identità esclusiva e permanente che è indipendente dal suo stato corrente. La maniglia di un oggetto (OID, Object IDentifier) è il meccanismo utilizzato generalmente per assegnare un'identità a un oggetto e deve essere noto a qualunque oggetto che desideri inviare messaggi a quell'oggetto particolare. Un messaggio è costituito dal nome di un'operazione definito per l'oggetto destinatario, insieme a qualunque parametro che debba essere passato all'operazione o da essa; i parametri possono essere dati o puntatori a dati, tuttavia in un ambiente a oggetti essi fanno riferimento o puntano soltanto a oggetti.

Gli oggetti derivati dalla stessa classe condividono la stessa struttura e il medesimo comportamento: una classe è una sagoma disegnata e programmata per essere la struttura dalla quale le istanze della classe, gli oggetti, sono fabbricate al momento dell'esecuzione. Una classe può avere un set di operazioni di classe e di attributi di classe.

In linea di principio, ogni oggetto ha il proprio insieme di metodi per implementare le sue operazioni di istanza e il proprio insieme di variabili per implementare i suoi attributi di istanza; ma in pratica gli oggetti della stessa classe normalmente risparmiano l'uso della memoria condividendo la stessa copia di ciascuno dei propri metodi di istanza.

Le classi possono formare gerarchie di ereditarietà formate da superclassi e sottoclassi. L'ereditarietà permette agli oggetti di una classe di utilizzare gli strumenti di qualunque superclasse a essa relativa: tuttavia in una sottoclasse è possibile ridefinire o sovrascrivere (tramite l'overriding) selettivamente le operazioni ereditate da una classe.

Il polimorfismo è il meccanismo grazie al quale un singolo nome di operazione può essere definito per più di una classe e può assumere diverse implementazioni in ciascuna di quelle classi. In alternativa, il polimorfismo è la proprietà per mezzo della quale un attributo può puntare a (detenere la maniglia di) oggetti di diverse classi in diversi momenti. Il polimorfismo imprime una nuova svolta all'occultamento dell'implementazione e dà alla tecnologia a oggetti buona parte della sua potenza. Per esempio, un oggetto mittente può inviare un messaggio senza conoscere la classe esatta dell'oggetto destinatario: a patto che il progettista sappia che tutte le possibili classi cui il destinatario può appartenere hanno accesso a un'operazione con il nome e la segnatura corretti, la selezione della particolare operazione può essere lasciata all'ambiente di esecuzione.

L'overloading è un concetto simile a quello di polimorfismo, per il fatto che in fase di esecuzione viene scelta una tra le molte possibili implementazioni di un'operazione verificando il numero e/o le classi degli argomenti di un messaggio. Sia il polimorfismo sia il sovraccarico tipicamente richiedono il legame dinamico.

La genericità permette a una classe parametrica di accettare una classe come argomento ogni volta che viene istanziato un oggetto: ciò consente una facile creazione di classi contenitore generiche con la funzione di scheletro, su cui vengono aggiunti particolari specifici durante l'esecuzione. Le classi parametriche offrono i vantaggi del codice donato senza il lavoro aggiuntivo dovuto alla manutenzione replicata.

11 Esercizi

1. Un oggetto conosce la sua maniglia? E il tal caso, in che modo un oggetto fa riferimento a essa?

2. Un'operazione di istanza può fare riferimento a una variabile di classe. Tuttavia, in un ambiente a oggetti vero e proprio, un'operazione di classe non può fare direttamente riferimento a una variabile di istanza all'interno di un oggetto. Perché?

3. Quando abbiamo eseguito `Aliante.New()` nella sezione refsec:ereditarieta, quanti oggetti abbiamo creato?

4. Come viene avviato un programma a oggetti?

5. Cosa accade a tutti i vostri oggetti quando spegnete il computer?

6. Cosa accade a tutte le vostre classi quando spegnere il computer?

7. Riuscite a trovare un modo semplice per aggirare il robusto meccanismo dell'incapsulamento della tecnologia a oggetti in un linguaggio come il C++?

12 Risposte agli esercizi

1. Si. Un oggetto ha una variabile (in realtà una costante) che non è necessario dichiarare e che contiene la propria maniglia. Il nome di questa variabile è la parola chiave self, this, this o Current (rispettivamente in Smalltalk, C++, Java o Eiffel).

2. La difficoltà è attributo di quale oggetto? Ricorderete che per una data classe possono esserci migliaia di oggetti di quella classe in fase di esecuzione. L'unico modo che la classe ha per raggiungere l'oggetto desiderato è avere la maniglia di quell'oggetto e inviargli un messaggio, che a questo punto può restituire le informazioni sull'attributo alla classe che ha inviato il messaggio.

3. Uno (ogni volta che eseguiamo l'operazione di classe `New()` , creiamo sempre esattamente un oggetto con, ovviamente, una maniglia). Tuttavia questo oggetto, che abbiamo denominato a, è un istanza di `Aliante` e anche, tramite il meccanismo dell'ereditarietà, di `Velivolo` .

4. L'avvio di un programma a oggetti dipende sia dal linguaggio sia dal sistema operativo utilizzato. Comunque sia sono tre i modi più comuni in cui un sistema operativo può trasferire il controllo al programma applicativo:

   - Iniziare l'esecuzione da una funzione principale (come main() in C++) che, come in un normale programma procedurale, acquisisce il controllo all'avvio del programma.
   - Istanziare automaticamente un oggetto di una classe `Radice` definita dal programmatore il quale, al suo avvio, dà inizio all'esecuzione dell'intero programma.
   - Avviare un ambiente di browsing a oggetti, in genere con un'interfaccia grafica. l'utente-programmatore a questo punto può interattivamente inviare un messaggio a una classe appropriata (o ad un oggetto) per far funzionare le cose.

5. Quando spegnere il computer, perdete tutti gli oggetti presenti nella memoria volatile insieme alle informazioni in essi contenute. Se questo è un problema, allora dovete memorizzare le informazioni su disco prima che il programma a oggetti termini. Con un sistema di gestione database a oggetti (ODBMS, Object-Oriented Database-Management System), potete memorizzare gli oggetti in modo più o meno diretto. Se invece state utilizzando, per esempio, un database relazionale, dovete trasformare le informazioni degli oggetti in un formato tabellare normalizzato prima di poterle memorizzare. In mancanza di database, potete sempre ricorrere a file testuali.

6. Quando spegnete il computer, non accade nulla alle vostre classi (almeno nella maggior parte degli ambienti), perché sono codice che è stato compilato e memorizzato in modo permanente sul disco. Ciò illustra ancora una volta la differenza tra le classi, che sono forme permanenti, e gli oggetti, che sono unità sempre mutevoli legate all'esecuzione.

7. In alcuni linguaggi, un programmatore malintenzionato può stabilire dei modi che consentono agli esterni di scalare le mura di un oggetto e gettare scompiglio direttamente tra le sue variabili. Uno di questi modi è rappresentato dal meccanismo delle funzioni amichevoli (friend) del C++. Come quasi tutti gli altri peccati che riguardano la progettazione, anche questo viene in genere commesso nel nome della grande dea Efficienza.

Riferimenti bibliografici

[1] Meilir Page-Jones. *Progettazione a oggetti con UML.* Apogeo, Gennaio 2002.

[2] Roger Pressman. *Principi di Ingegneria del software.* Mc Graw Hill, Luglio 2000.

[3] Leszek A. Maciaszek. *Sviluppo di sistemi informativi con UML - Analisi dei requisiti e progetto di sistema.* Addison Wesley, Maggio 2002.

Indice analitico