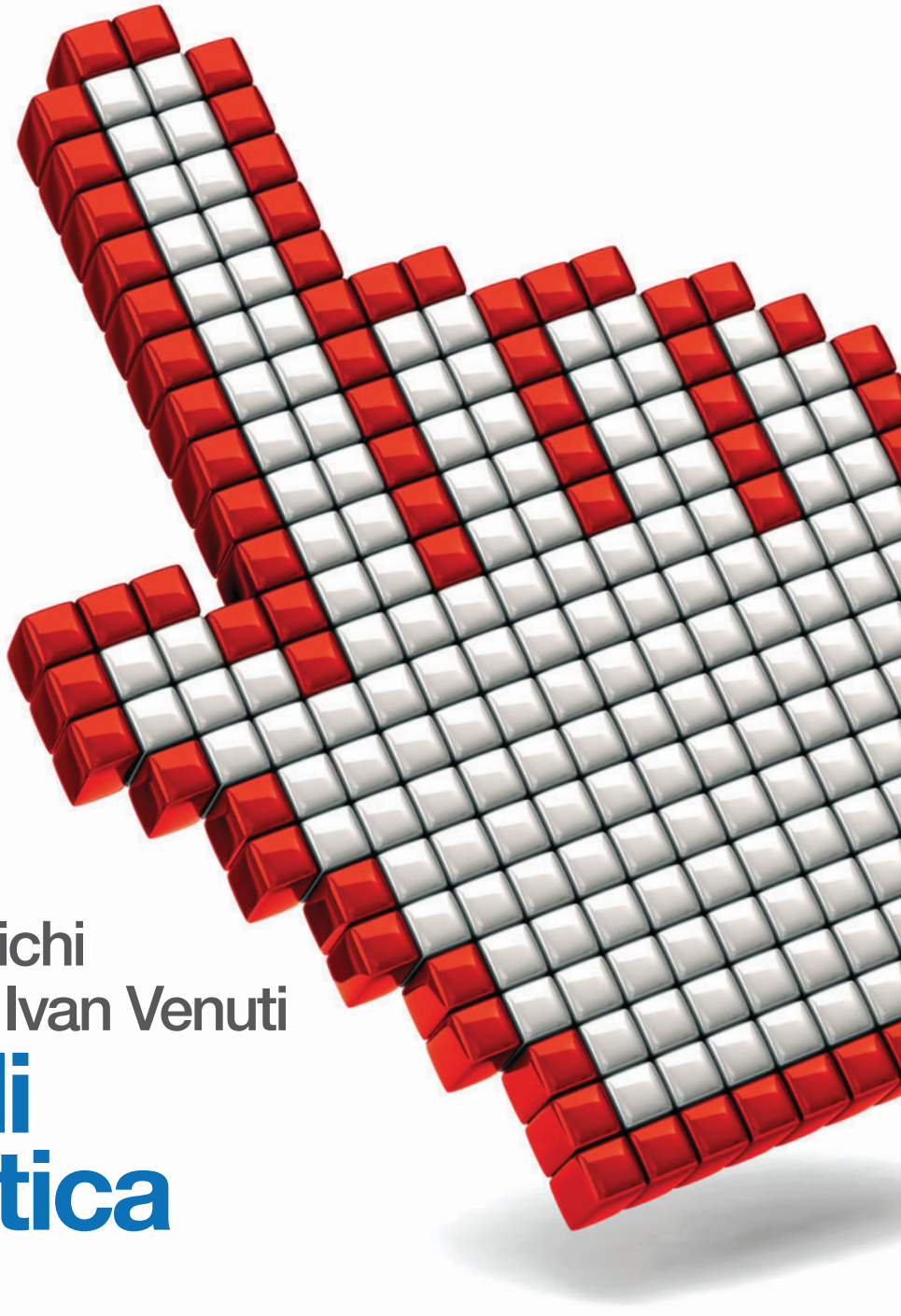
1 2 3 Idee per il tuo futuro



Fiorenzo Formichi
Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

TECNOLOGIA ZANICHELLI

Fiorenzo Formichi
Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

Realizzazione editoriale:
– Coordinamento redazionale: Matteo Fornesi
– Segreteria di redazione: Deborah Lorenzini
– Progetto grafico: Editta Gelsomini
– Collaborazione redazionale, impaginazione, disegni e indice analitico: Stilgraf, Bologna

Contributi:
– I capitoli della sezione B sono a cura di Ivan Venuti
– Rilettura dei testi in inglese: Roger Loughney

Copertina:
– Progetto grafico: Miguel Sal & C., Bologna
– Realizzazione: Roberto Marchetti
– Immagine di copertina: valdis torms/Shutterstock; Artwork Miguel Sal & C., Bologna

Prima edizione: gennaio 2012

L'impegno a mantenere invariato il contenuto di questo volume per un quinquennio (art. 5 legge n. 169/2008) è comunicato nel catalogo Zanichelli, disponibile anche online sul sito www.zanichelli.it, ai sensi del DM 41 dell'8 aprile 2009, All. 1/B.

File per diversamente abili
L'editore mette a disposizione degli studenti non vedenti, ipovedenti, disabili motori o con disturbi specifici di apprendimento i file pdf in cui sono memorizzate le pagine di questo libro. Il formato del file permette l'ingrandimento dei caratteri del testo e la lettura mediante software screen reader. Le informazioni su come ottenere i file sono sul sito www.zanichelli.it/diversamenteabili

Suggerimenti e segnalazione degli errori
Realizzare un libro è un'operazione complessa, che richiede numerosi controlli: sul testo, sulle immagini e sulle relazioni che si stabiliscono tra essi. L'esperienza suggerisce che è praticamente impossibile pubblicare un libro privo di errori. Saremo quindi grati ai lettori che vorranno segnalarceli.
Per segnalazioni o suggerimenti relativi a questo libro scrivere al seguente indirizzo:

lineazeta@zanichelli.it

Le correzioni di eventuali errori presenti nel testo sono pubblicati nel sito www.zanichelli.it/aggiornamenti

Zanichelli editore S.p.A. opera con sistema qualità
certificato CertiCarGraf n. 477
secondo la norma UNI EN ISO 9001:2008

Fiorenzo Formichi
Giorgio Meini Ivan Venuti

Corso di informatica

per Informatica

Programmazione orientata agli oggetti e linguaggio Java
Pagine web con JavaScript

TECNOLOGIA ZANICHELLI

Indice

SEZIONE A

Programmazione orientata agli oggetti e linguaggio Java

A1 Introduzione alla programmazione e alla progettazione orientate agli oggetti

A2 Il linguaggio di programmazione Java

A3 La programmazione orientata agli oggetti in Java

A4 Strutture dati

A5 Ereditarietà e polimorfismo

A6 Tipi generici e collezioni nel linguaggio Java

A7 Introduzione alle *Graphic User Interface* in Java

A8 Ambiente di sviluppo *NetBeans* e applicazioni Java con GUI *Swing*

A9 Gestione della concorrenza nel linguaggio Java

SEZIONE B

Pagine web con JavaScript

B1 Il linguaggio JavaScript

B2 JavaScript e il DOM, jQuery e Google *Maps*

B3 Strumenti per lo sviluppo di pagine web con JavaScript

I capitoli affiancati da questa icona sono disponibili, con chiave di attivazione, all'indirizzo:
www.online.zanichelli.it/formichimeinicorsoinformatica

SEZIONE

A Programmazione orientata agli oggetti e linguaggio Java

A1 Introduzione alla programmazione e alla progettazione orientate agli oggetti

Informatizzare una realtà significa crearne un modello astratto finalizzato alla gestione dei flussi informativi che la caratterizzano. L'ambiente circostante è costituito da sistemi spesso molto articolati: i mezzi di comunicazione, quelli di trasporto e le stesse realtà urbane in cui viviamo mostrano scenari in continua trasformazione, dove vari tipi di soggetti (persone, veicoli, mezzi di trasmissione, computer, …) interagiscono tra di loro a diversi livelli di complessità. Studiare il comportamento di questi sistemi e delle loro interazioni per crearne un modello informatico richiede l'utilizzo di tecniche che siano al tempo stesso efficaci ed efficienti. Secondo una visione ormai condivisa e consolidata, un primo passo in questa direzione è rappresentato dall'individuazione e astrazione delle entità di riferimento per tali contesti.
Consideriamo a questo proposito un oggetto di uso quotidiano come un televisore, allo scopo di volerne creare un modello per descriverne sia gli aspetti statici (non soggetti a cambiamento) sia quelli dinamici (soggetti a cambiamento). Alcuni dati (attributi) significativi sono:

- la dimensione in pollici dello schermo;
- il tipo di schermo (LCD, LED, …);
- il colore esterno;
- il canale attualmente selezionato;
- il livello del volume impostato;
- il livello della luminosità regolato;
- …

Il contenuto informativo che essi descrivono è relativo sia alle **caratteristiche** del televisore (colore esterno, dimensione in pollici, …), sia allo **stato** in cui esso si trova in un determinato momento (acceso/spento, livello del volume, …). Ma per una completa modellizzazione dobbiamo considerare anche alcuni suoi **comportamenti** modificabili mediante i controlli esterni (il telecomando e/o i pulsanti):

- accenditi/spegniti;
- cambia canale;
- alza/abbassa il volume;
- modifica la luminosità;
- …

Object Oriented Programming

L'interesse e la definitiva diffusione dell'approccio OOP sono dovuti al passaggio, avvenuto nel decennio 1980-1990, dalle precedenti architetture informatiche monopiattaforma a quelle multipiattaforma distribuite.
Questa evoluzione ha comportato alcuni problemi da risolvere, in particolare la questione di come distribuire i dati e le funzioni che costituivano le precedenti applicazioni monolitiche su una pluralità di sistemi cooperanti, secondo quale logica e con quale modalità di colloquio tra diversi moduli applicativi.
L'OOP ha fornito a tali domande risposte innovative e tecnicamente adeguate, in particolare sotto il profilo tecnico-organizzativo nello sviluppo del software.
Un'applicazione a oggetti è costituita da un insieme di moduli software logicamente indipendenti in cui le classi definiscono modelli astratti che incapsulano dati e operazioni sui dati stessi (risolvendo la tradizionale distinzione tra dati e funzionalità) e che interagiscono tra loro tramite scambio di «messaggi».

In pratica, per definire il modello del nostro oggetto, abbiamo effettuato:

- una astrazione sui dati mediante le sue caratteristiche (attributi);
- una astrazione funzionale individuando le azioni che può compiere (operazioni).

Queste due astrazioni sono alla base dell'**approccio orientato agli oggetti** (*Object Oriented* o **OO** in breve), sia in relazione alla **progettazione** (**OOD**, *Object Oriented Design*) sia alla **programmazione** (**OOP**, *Object Oriented Programming*).

OSSERVAZIONE Il livello di complessità dell'architettura interna di un televisore (vista come insieme dei suoi componenti) non è banale; ciò nonostante, la maggior parte di noi sa come impiegarli – almeno nelle funzionalità di base – utilizzando pochi comandi che possono essere attivati mediante un telecomando o i pulsanti presenti al suo esterno. In pratica non interessa (e non deve interessare) sapere come è fatto dentro per poterlo utilizzare. D'altra parte non è pensabile comandarlo interagendo direttamente con i componenti elettronici che costituiscono la sua struttura interna, utilizzando strumenti come le pinze o i cacciavite, per ragioni sia di praticità sia di sicurezza: potremmo danneggiare il televisore o rimanere fulminati da una scarica elettrica!

La precedente osservazione è inerente a uno dei fondamenti dell'approccio OO: il **principio di *information hiding*** (**occultamento dell'informazione**), che stabilisce la netta separazione tra l'implementazione (struttura interna) e l'interfaccia (comandi per l'utilizzo) di un oggetto.

1 Tipi di dato astratto e principio di *information hiding*

Quando un informatico si riferisce ai tipi di dato viene immediatamente da pensare ai tipi elementari come i numeri interi, i numeri decimali, le stringhe di caratteri, spesso utilizzati nel contesto di un programma.

Ma un tipo di dato è caratterizzato da due aspetti fondamentali: un **insieme di valori** e un **insieme di operazioni** che possono essere a esso applicate.

ESEMPIO

Per i valori numerici interi compresi in un intervallo finito sono definite le classiche operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione, …).
Le stringhe di testo sono sequenze finite di caratteri per le quali sono definite operazioni che consentono, per esempio, di determinare o modificare un singolo carattere, di concatenare due stringhe, di estrarre una sottostringa e così via.

OSSERVAZIONE In matematica i valori e le operazioni di uno specifico tipo di dato sono definiti mediante equazioni, assiomi o altre descrizioni formali. Un'algebra in prima approssimazione viene definita come un insieme di valori associato alle operazioni definite su di esso.

Con l'espressione **tipo di dato astratto** (**ADT**, *Abstract Data Type*) ci si riferisce a un tipo di dato completamente specificato, ma indipendentemente da una sua particolare implementazione. L'implementazione di un ADT è demandata allo sviluppatore del software sia per il contenuto informativo (i **dati**, o **attributi**), che per le operazioni specifiche associate al tipo di dato (**metodi**).

Un ADT è caratterizzato dalle seguenti proprietà caratteristiche:

- la definizione di un nuovo tipo di dato;
- la definizione di un insieme di operazioni sul tipo di dato che costituiscono la sua interfaccia;
- le operazioni previste dall'interfaccia sono l'unico meccanismo tramite il quale è possibile interagire con il dato;
- il comportamento delle operazioni è definito mediante condizioni prestabilite.

OSSERVAZIONE Nella programmazione orientata agli oggetti l'elemento centrale è il dato: l'idea alla base dei tipi di dato astratti è infatti quella di definire una struttura di dati con associate le operazioni definite su di essa. Quindi è il tipo di dato che esporta le procedure necessarie alla sua gestione.

ESEMPIO

La maggior parte dei linguaggi di programmazione non possiede tra i propri tipi di dato elementari le frazioni. Un ADT che definisca come nuovo tipo una frazione avrà come attributi i valori del numeratore e del denominatore della frazione e come operazioni le usuali operazioni aritmetiche: somma, sottrazione, moltiplicazione, divisione, … realizzate mediante gli algoritmi che consentono di ottenere una frazione come risultato di un'operazione che interessa due frazioni come operandi.

Il principio fondamentale di progettazione degli ADT è quello dell'*information hiding* o **incapsulamento** che stabilisce la netta separazione tra l'interfaccia esposta da un ADT all'esterno per essere utilizzato e la sua implementazione interna che deve invece rimanere nascosta. Un obiettivo importante del principio di *information hiding* è rendere invisibili all'esterno di un ADT le scelte implementative che possono essere soggette a modifiche, proteggendo in questo modo il codice che utilizza l'ADT dalle conseguenze dovute all'eventuale cambiamento di tali scelte.

ESEMPIO

Il segno di una frazione definita mediante un ADT può essere rappresentato in vari modi: in modo separato dai valori del numeratore e del denominatore (considerati in questo caso sempre positivi), come segno del solo numeratore (considerando in questo caso il denominatore sempre positivo), come combinazione algebrica dei segni del numeratore e del denominatore). L'eventuale variazione della modalità di rappresentazione del segno internamente all'ADT non dovrebbe in nessun modo alterarne l'interfaccia esterna.

L'incapsulamento, inoltre, garantisce un corretto utilizzo del tipo di dato prevenendo eventuali errori nella sua gestione da parte del codice che lo utilizza.

ESEMPIO

Un ADT che definisce una frazione deve impedire che il denominatore sia erroneamente impostato al valore zero: l'impossibilità di accedere dall'esterno direttamente ai valori dei dati (attributi) consente allo sviluppatore che implementa le operazioni (metodi) di evitare questa condizione di errore.

Ovviamente è necessario, sia in fase di progettazione sia di implementazione di un ADT, porre molta cura nella definizione della sua interfaccia esterna. Il corretto funzionamento del codice che usa un ADT è garantito dal rispetto delle specifiche dell'interfaccia e dalla corretta utilizzazione delle operazioni che l'interfaccia stessa rende disponibili.

OSSERVAZIONE Dal momento che la correttezza dell'accesso e della variazione dei valori memorizzati da un ADT sono garantiti dalle operazioni dell'interfaccia e che quest'ultima costituisce l'unico meccanismo di accesso alla rappresentazione interna dei dati, nello sviluppo del software il programmatore può utilizzare gli ADT come componenti «pronti all'uso» preoccupandosi esclusivamente della logica generale dell'applicazione in relazione alle funzionalità che vuole realizzare.

Riguardo alla modularità delle applicazioni possiamo affermare che il principio di *information hiding* è fondamentale nella progettazione e realizzazione di software: la suddivisione in moduli e la corretta specificazione delle loro interfacce ne permette lo sviluppo e la verifica indipendente, in modo che ogni eventuale errore rimanga relativo a un singolo modulo (a meno che non si tratti di un errore di natura più complessa relativo all'errata modellazione dell'interazione tra moduli).

La correzione di un errore implica quindi la modifica del solo modulo che lo contiene, senza effetti collaterali sugli altri moduli nell'ottica di una filosofia di progetto per cui, come afferma Grady Booch[1], «nessuna parte di un sistema complesso dovrebbe dipendere dai dettagli interni di qualsiasi altra parte».

1. Grady Booch è un noto progettista e metodologo nel campo dell'ingegneria del software *object oriented*: uno dei suoi principali contributi ha consistito nella partecipazione alla definizione dello *Unified Modeling Language* (UML) insieme a James Rambaugh e Ivar Jacobson (universalmente conosciuti come *los tres amigos*).

Nella pratica della programmazione il principio di *information hiding* viene realizzato ricorrendo a una tecnica che, in generale, prevede la classificazione degli attributi e dei metodi in **pubblici** o **privati**. Un elemento pubblico sarà visibile nell'interfaccia che un oggetto espone verso l'esterno, mentre un elemento privato è confinato all'interno e per l'utente dell'oggetto esso risulterà invisibile.

ESEMPIO

Il concetto dell'*information hiding* è rappresentato graficamente nella FIGURA **1** dove la struttura privata interna di un oggetto che modella un televisore è incapsulata e protetta dall'interfaccia che espone all'esterno le sole operazioni pubbliche che permettono di interagire con l'oggetto stesso.

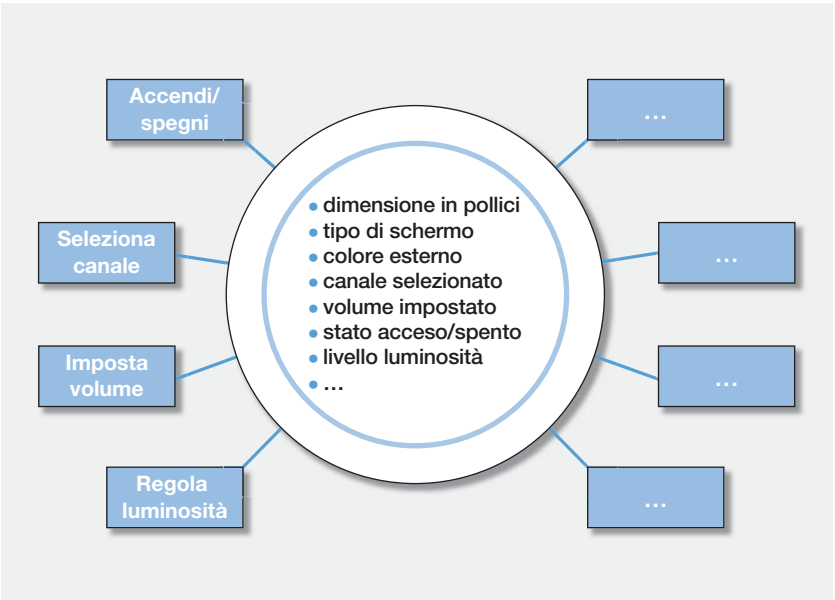


FIGURA **1**

OSSERVAZIONE Pur essendo possibile definire attributi di tipo pubblico, una buona prassi di progettazione e programmazione sconsiglia questa pratica; lasciando gli attributi privati, essi risultano accessibili esclusivamente mediante i metodi pubblici che l'oggetto espone nella sua interfaccia. Stabilendo, per esempio, che il livello di volume del nostro televisore è impostabile in un intervallo compreso tra i valori 0 e 100, è inopportuno lasciare al codice che utilizza l'oggetto la possibilità di modificare senza alcun controllo tale valore: sarebbe infatti possibile impostare dei valori fuori dell'intervallo definito causando un errore. È preferibile permettere la gestione del livello di volume esclusivamente mediante specifici metodi pubblici – come *alzaVolume* e *abbassaVolume* – che nella loro implementazione dovranno impedire che il valore impostato fuoriesca dall'intervallo predefinito.

2 Classi e oggetti, attributi e metodi nei diagrammi UML

Il concetto di **classe** è alla base della progettazione e della programmazione orientate agli oggetti: una classe rappresenta la descrizione di una specifica categoria di oggetti individuati da comportamenti e caratteristiche simili.

ES. La classe *Televisore* può essere usata per definire gli oggetti *televisori*.

Come abbiamo già visto i comportamenti (le funzionalità) sono denominati **metodi** o **operazioni**, mentre le componenti informative (i dati) sono denominate **proprietà** o **attributi**.

▶ Nel contesto della OOD/OOP, una **classe** rappresenta un modello formale per la descrizione di un certo tipo di oggetti definendone gli attributi, i metodi e le caratteristiche dell'interfaccia.

Nel codice di un qualsiasi linguaggio di programmazione OO, a partire dalla definizione di una classe, è possibile creare (il termine tecnico è **istanziare**) oggetti simili che condividono lo stesso insieme di attributi, lo stesso insieme di metodi e la stessa interfaccia. Ogni oggetto (**istanza** della classe) è però un'entità autonoma con propri valori per gli attributi.

ESEMPIO A partire dalla classe *Televisore* è possibile istanziare due oggetti distinti *televisore_soggiorno* e *televisore_cucina* dove il valore dell'attributo che definisce la dimensione dello schermo in pollici può essere uguale a 32 per *televisore_soggiorno* e a 24 per *televisore_cucina*.

Nella programmazione OO la creazione di un oggetto ha come conseguenza due azioni consequenziali:

- allocare un'area di memoria per la memorizzazione dell'oggetto stesso;
- inizializzare i valori degli attributi che costituiscono la componente informativa dell'oggetto.

La seconda azione viene espletata dal **costruttore** della classe, ossia da uno speciale metodo, normalmente denominato con lo stesso nome della classe.

Nella progettazione OO, tra le varie proposte che si sono succedute nel tempo per la definizione di un formalismo che permettesse la rappresentazione delle classi, **UML** (*Unified Modeling Language*) è sicuramente quella che ha

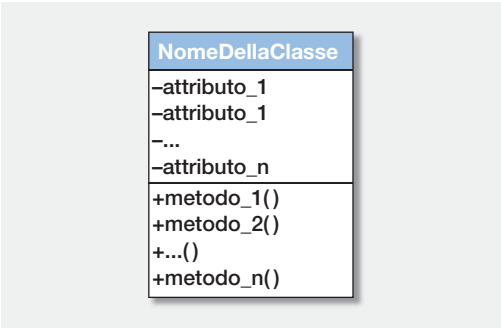
Unified Modeling Language

L'acronimo UML (*Unified Modeling Language*) identifica, nell'ambito dell'ingegneria del software, un insieme di linguaggi grafici di modellazione basato sul paradigma OO.
Esso fu proposto nel 1996 da Grady Booch, Jim Rumbaugh e Ivar Jacobson con la supervisione dello *Object Management Group*, il consorzio che tuttora ne gestisce lo standard.
UML nacque con l'intenzione di unificare gli approcci precedenti, raccogliendone gli aspetti migliori per la definizione di uno standard industriale unificato che comprendesse diversi formalismi grafici, tra cui: il diagramma delle classi, il diagramma degli oggetti, il diagramma dei casi d'uso, il diagramma di sequenza, il diagramma degli stati, il diagramma delle attività, il diagramma delle comunicazioni, il diagramma di *deployment*.
Essendo un set di linguaggi grafici, l'UML svolge, nella comunità della programmazione a oggetti, un'importante funzione per la progettazione del software, indipendentemente dall'effettivo linguaggio di programmazione utilizzato.
È ormai uso comune utilizzare i diagrammi UML per descrivere modelli di analisi e soluzioni progettuali in modo sintetico, comprensibile e rigoroso.
L'ultima versione del linguaggio, la 2.0, è stata consolidata nel 2004 e ufficializzata da OMG nel 2005.

Classi parametriche

Una **classe parametrica** è una classe definita utilizzando dei tipi generici che dovranno essere specificati come tipi reali da parte del codice che ne istanzierà gli oggetti: in questo modo è possibile generalizzare la funzionalità di una classe.
Per esempio, una classe che implementa una lista di elementi può essere facilmente definita in modo indipendente dal tipo di elementi che vi potranno essere memorizzati: il tipo specifico degli elementi potrà essere indicato al momento della creazione di una lista e la stessa classe consentirà di gestire liste di elementi di tipo diverso.

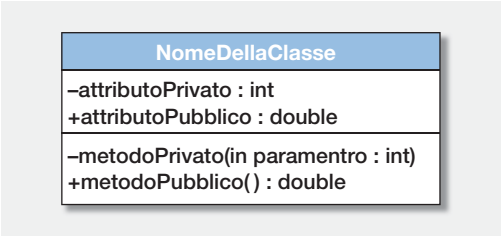
riscosso il maggior successo. Tra i vari formalismi grafici che UML mette a disposizione per descrivere i diversi aspetti di un sistema software a molteplici livelli di dettaglio, faremo riferimento al **diagramma delle classi** (*class diagram*) che, come è deducibile dal nome, fornisce una notazione grafica per formalizzare le classi e le relazioni che intercorrono tra di esse. Definire una classe utilizzando un *class diagram* UML è piuttosto semplice; lo schema generale del diagramma relativo a una singola classe è infatti il seguente:



dove:

- nella prima sezione a partire dall'alto si inserisce il nome della classe;
- la seconda sezione è relativa alla definizione degli attributi (proprietà);
- la terza e ultima sezione è relativa alla definizione delle operazioni (metodi).

È inoltre possibile classificare le componenti private e quelle pubbliche premettendo ai singoli nomi rispettivamente i simboli «–» e «+» :



OSSERVAZIONE Si noti che per ogni metodo, oltre al suo livello di visibilità (pubblica o privata) è necessario definire:

- i nomi e i tipi degli eventuali parametri e il loro ruolo (in/out/in-out);
- il tipo dell'eventuale valore restituito.

ESEMPIO

Il diagramma UML di FIGURA **2** definisce la classe *Televisore*.
In relazione a questo esempio si noti che:

- tutti gli attributi sono privati e tutti i metodi sono pubblici;
- per accedere agli attributi privati sono stati definiti metodi convenzionalmente noti come *getter/setter*: un metodo il cui nome inizia con il prefisso *get*, seguito dal nome di un attributo della classe, restituisce il valore dell'attributo; un

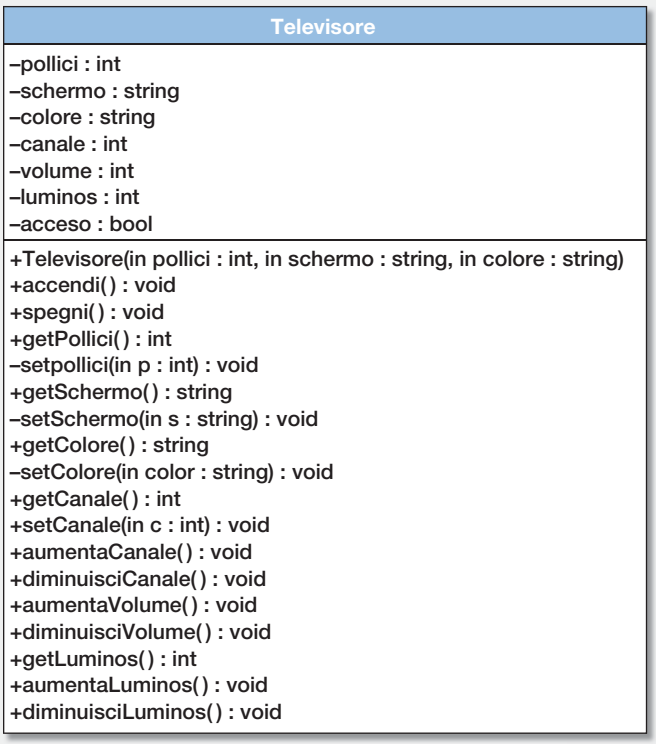


FIGURA **2**

metodo il cui nome inizia con il prefisso *set*, seguito dal nome di un attributo, ha lo scopo di impostare un nuovo valore la cui congruità viene controllata dal metodo stesso;

- non per tutti gli attributi sono stati definiti dei metodi *setter*: in particolare alcuni attributi di un televisore non sono modificabili nel corso della sua esistenza e possono essere impostati solo dal metodo costruttore che assume il nome della stessa classe;
- i metodi che consentono di modificare gli attributi aumentandone o diminuendone il valore devono essere implementati in modo che non possano impostare valori inferiori al minimo valore predefinito, o superiori al massimo valore predefinito per ogni attributo.

3 Interazione tra oggetti e diagrammi UML di sequenza

In un sistema software gli oggetti interagiscono tra di loro scambiandosi **messaggi**: un messaggio è un'informazione che viene scambiata tra due oggetti.

OSSERVAZIONE In un linguaggio di programmazione OO lo scambio di messaggi avviene mediante invocazione dei metodi degli oggetti istanziati a partire dalle classi definite.

Design pattern

I *design pattern* sono soluzioni generali predefinite di progettazione applicabili a problemi ricorrenti nella progettazione del software in contesti eterogenei.

Questo tipo di approccio è stato proposto da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, ormai universalmente noti a tutti gli informatici come la *Gang of Four* o mediante l'acronimo collettivo GoF.

L'idea di base della proposta è piuttosto semplice: dal momento che la qualità della progettazione software deriva dall'esperienza del progettista – un progettista esperto nel risolvere nuovi problemi anche in nuovi contesti utilizza comunque schemi di soluzioni già sperimentate che hanno fornito buoni risultati in precedenza – è possibile individuare, formalizzare (spesso utilizzando diagrammi UML) e pubblicare le soluzioni progettuali che risolvono i problemi che si presentano più frequentemente in modo indipendente dal contesto.

In questo modo si mettono a disposizione della comunità degli sviluppatori software privi della necessaria esperienza soluzioni predefinite di qualità almeno per i problemi progettuali comuni.

Il messaggio può essere:

- **sincrono**: l'emittente rimane in attesa di una risposta;
- **asincrono**: l'emittente non rimane in attesa di una risposta che può eventualmente essere ricevuta in un momento successivo.

Un messaggio generato in risposta a un precedente messaggio, e al quale eventualmente fa riferimento anche in relazione al suo contenuto informativo, è detto **messaggio di risposta**.

È possibile rappresentare graficamente lo scambio di messaggi tra oggetti utilizzando un **diagramma UML di sequenza** (*sequence diagram*). Un diagramma di sequenza è un tipo di diagramma dello standard UML ideato per descrivere una determinata sequenza di interazioni.

ESEMPIO

La FIGURA **3** rappresenta il diagramma di sequenza che descrive l'acquisto di un oggetto in un negozio da parte di un cliente.

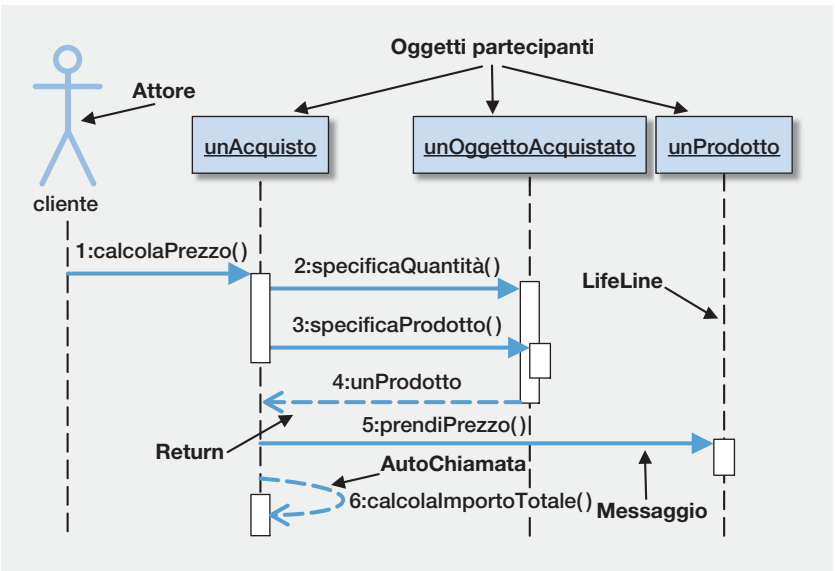


FIGURA **3**

In un diagramma di sequenza si distinguono i seguenti elementi.

- **Istanze di classi (oggetti)**: sono rappresentate da rettangoli riportanti il nome della classe e l'identificatore dell'oggetto (con la notazione: ***oggetto : classe***), o più semplicemente un nome dal quale si desume che si tratta di un'istanza della classe (*unAcquisto*, *unProdotto*, …).
- **Attori**: se presenti, sono riportati sulla sinistra con le frecce di interazione rivolte verso gli oggetti del sistema.
- **Messaggi**: sono rappresentati come frecce da un attore a un oggetto, o tra due oggetti; i messaggi di ritorno, se riportati, hanno la linea della freccia tratteggiata. I messaggi sincroni terminano con una freccia triangolare piena, mentre i messaggi asincroni terminano con una freccia aperta semplice. Un messaggio può anche insistere sullo stesso oggetto

che lo invia. L'ordine dei messaggi – dall'alto verso il basso – rispetta la sequenza temporale con cui sono scambiati; per maggiore chiarezza essi possono essere preceduti da un numero che ne indica l'esatta successione.

- **Linee di vita (*life-line*)**: sono linee tratteggiate verticali che partono dal rettangolo rappresentativo dell'oggetto; indicano il periodo temporale di vita dell'oggetto, dalla sua costruzione alla sua distruzione.
- **Barre di attivazione (*activation box*)**: sono rappresentate da un rettangolo colorato sovrapposto alla linea di vita di un oggetto che riceve un messaggio: rappresentano convenzionalmente il tempo di elaborazione della richiesta giunta all'oggetto.

I diagrammi UML di sequenza sono utilizzati in diverse fasi del ciclo di vita di sviluppo di un'applicazione software.

- Nella fase di analisi un *sequence diagram* può rappresentare graficamente uno scenario di un caso d'uso: in questo caso il ruolo degli oggetti sarà recitato da un generico oggetto denominato **sistema**, saranno presenti altri oggetti (indicanti, per esempio, alcuni componenti architetturali quali *server*, *database*, …) solo se rappresentano elementi vincolati dall'analisi.
- Nella prima parte della fase di progettazione i *sequence diagram* descrivono scenari di casi d'uso in cui come oggetti compaiono istanze delle classi individuate e come messaggi compaiono le operazioni riportate nel diagramma delle classi.
- Nella successiva progettazione di dettaglio i *sequence diagram* descrivono realizzazioni di metodi complessi, dove come oggetti compaiono istanze delle classi del diagramma delle classi del progetto di dettaglio e come messaggi compaiono le invocazioni dei metodi degli oggetti. In questo caso al posto degli elementi architetturali devono comparire oggetti istanza delle classi che ne consentono l'accesso (per esempio, invece che *database* dovrebbe comparire l'oggetto che consente di interagire con il database).

OSSERVAZIONE Per la descrizione di algoritmi, i *sequence diagram* – che sono stati specificatamente ideati per descrivere le interazioni – non sono lo strumento adatto: UML dispone allo scopo dei formalismi grafici *activity diagram* e *state-chart diagram*.

ESEMPIO

Un sistema distributore automatico di bevande in bottiglia può essere verosimilmente scomposto in quattro oggetti di base:

- il **cruscotto esterno**, ovvero l'interfaccia che la macchina presenta all'utente;
- il **ricevitore delle monete**, cioè il componente in cui vengono introdotte e raccolte le monete;
- il **controllo**, ovvero il dispositivo che gestisce l'intera macchina;
- il **contenitore delle bottiglie**, il sottosistema che contiene i prodotti acquistabili dagli utenti.

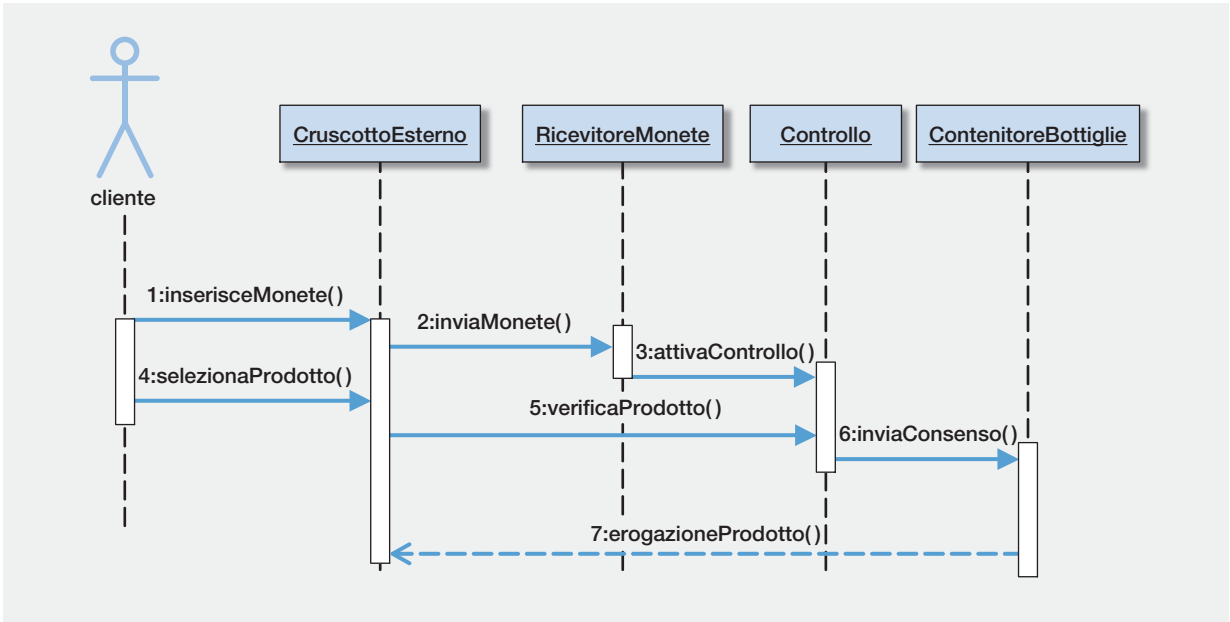


FIGURA **4**

Nel diagramma di sequenza possiamo individuare le seguenti azioni (FIGURA **4**):

- il cliente inserisce le monete nella macchina;
- il cliente esegue la selezione del prodotto desiderato;
- le monete vengono acquisite dal ricevitore;
- il dispositivo di controllo verifica se il prodotto richiesto è disponibile;
- il ricevitore delle monete aggiorna il suo contenuto;
- il dispositivo di controllo abilita il contenitore delle bottiglie all'erogazione del prodotto richiesto.

ESEMPIO

Un sistema che rappresenta un televisore con un servizio *pay-per-view* può essere rappresentato da tre oggetti:

- il **televisore**, cioè l'interfaccia tramite la quale opera l'utente (ovviamente dotato di dispositivo *pay-per-view* per l'introduzione della scheda di pagamento);
- il **controllo remoto**, che effettua il controllo della scheda di pagamento e del suo credito, da cui verrà detratto il costo del programma prescelto;
- il **fornitore del servizio**, cioè il sottosistema che memorizza i programmi televisivi che vengono acquistati dall'utente.

Nel diagramma di sequenza possiamo individuare le seguenti azioni (FIGURA **5**):

- l'utente inserisce la scheda prepagata nel televisore;
- l'utente esegue la selezione del programma desiderato;
- il controllo remoto verifica se il credito è sufficiente per pagare il servizio richiesto;
- il credito della scheda viene diminuito del costo previsto per la fruizione del programma;
- il dispositivo di controllo informa il sistema di erogazione del servizio che può rendere disponibile il programma prescelto.

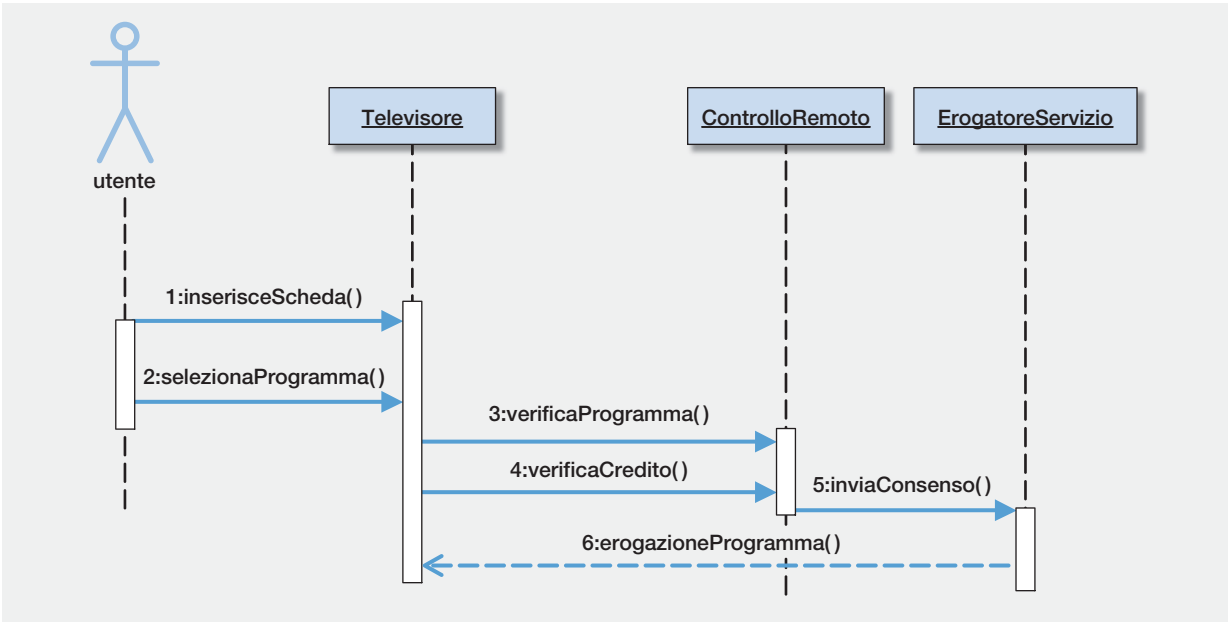


FIGURA **5**

4 Ereditarietà e polimorfismo

Un concetto fondamentale dell'OOD/OOP è quello di **ereditarietà**: utilizzando questo meccanismo è infatti possibile in un linguaggio OO creare nuove classi a partire da classi già esistenti ereditandone le caratteristiche (attributi e/o metodi), aggiungendone di nuove o ridefinendone alcune. L'ereditarietà è finalizzata alla creazione di gerarchie di classi e grazie a essa si estende la possibilità di riutilizzare componenti comuni a più classi della stessa gerarchia.

Per mezzo dell'ereditarietà tra classi è possibile definire una classe generale (**classe base** o **superclasse**) avente la funzione di definire le caratteristiche comuni a uno specifico insieme di oggetti. Le caratteristiche della classe generale potranno quindi essere ereditate o estese da altre classi (**classi derivate** o **sottoclassi**) che integreranno le caratteristiche della classe base con elementi specifici. Ma le classi derivate, a loro volta, potranno configurarsi come superclassi per nuove classi, realizzando in questo modo la gerarchia.

Una classe derivata da una classe base mantiene le proprietà (attributi) e le operazioni (metodi) della classe base da cui eredita e a questi aggiunge i propri attributi e i propri metodi specifici.

OSSERVAZIONE Il meccanismo dell'ereditarietà è particolarmente vantaggioso in un contesto dove si intende riutilizzare in ambiti diversi il codice già sviluppato: per mezzo di esso, infatti, è possibile definire, a partire da una classe data, nuove classi, specificando semplicemente le differenze tra queste e la classe di partenza.

ESEMPIO

Avendo definito la classe *Veicoli_a_motore*, essa sarà una classe base per le classi derivate *Auto* e *Motocicli*, che erediteranno da essa proprietà e operazioni. Pur con le proprie particolarità, infatti, sia le automobili sia i motocicli sono comunque entrambi dei veicoli a motore, in quanto dotati di caratteristiche comuni come la cilindrata, la targa, il numero di telaio, …). Procedendo nella costruzione della gerarchia, le due sottoclassi potranno a loro volta essere superclassi per altri tipi di automobili (*Tradizionali*, *Cabriolet*, *Suv*, …) o di motocicli (*Moto_Stradali*, *Moto_Fuoristrada*, …).

In un diagramma UML delle classi l'ereditarietà – che in questo contesto assume la denominazione di **generalizzazione** – viene rappresentata in modo intuitivo mediante frecce che collegano le classi derivate alle classi base (le estremità delle frecce sono triangolari, ma vuote).

ESEMPIO

Il diagramma di FIGURA **6** illustra la gerarchia delle classi ipotizzata nell'esempio precedente.

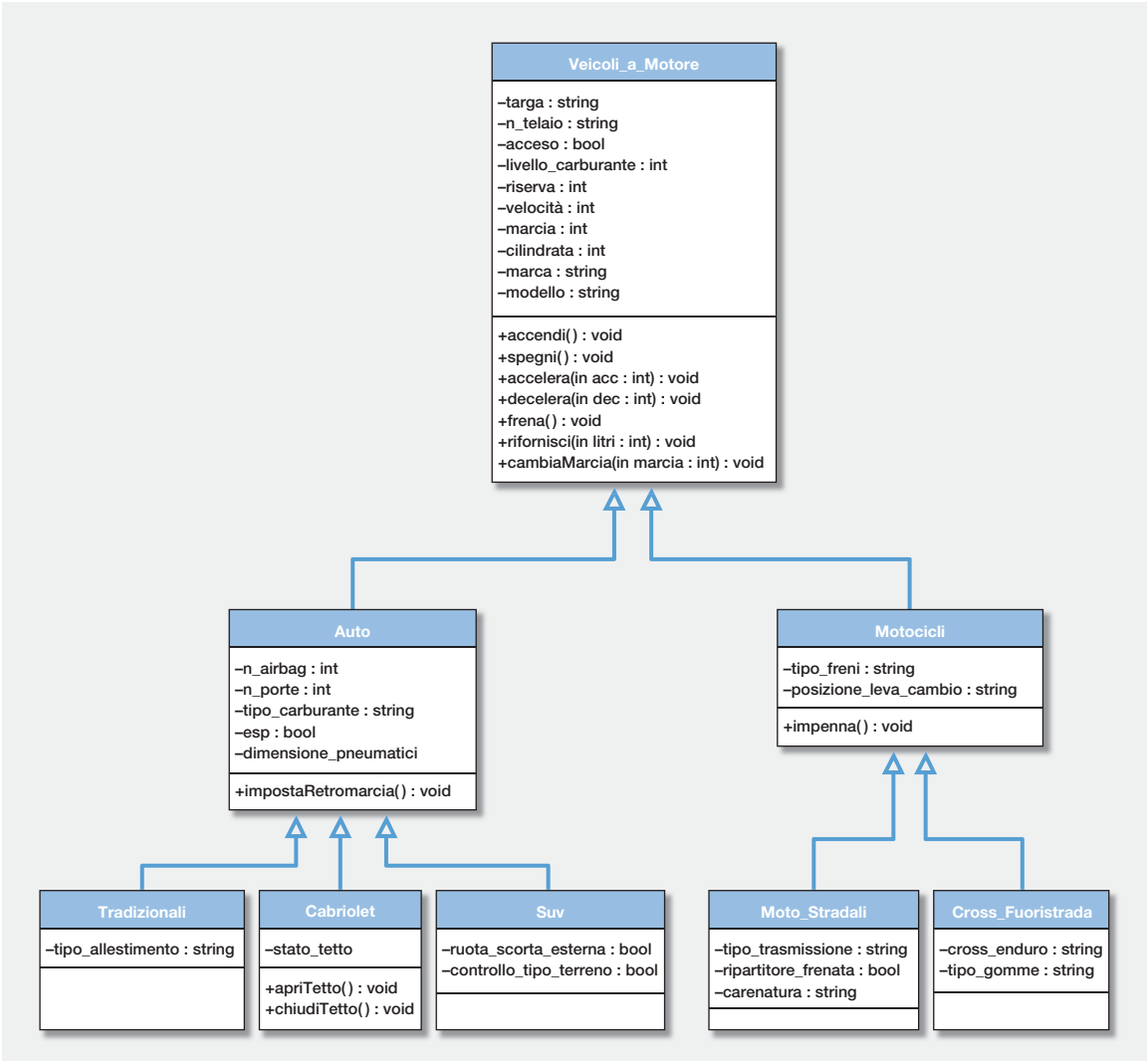


FIGURA **6**

Il tipo di ereditarietà presentato è noto come **ereditarietà singola**: in questo caso una sottoclasse può derivare al più da una superclasse. Esistono però situazioni in cui una sottoclasse dovrebbe derivare da due o più superclassi: in questo caso il tipo di ereditarietà è definito **ereditarietà multipla**.

ESEMPIO

Dovendo progettare una gerarchia di classi per rappresentare i veicoli in funzione dell'ambiente in cui questi sono capaci di spostarsi, il diagramma UML potrebbe essere strutturato nel modo illustrato da FIGURA **7**.
In questo caso i veicoli anfibi ereditano sia le caratteristiche dei veicoli terrestri sia quelle dei veicoli acquatici.

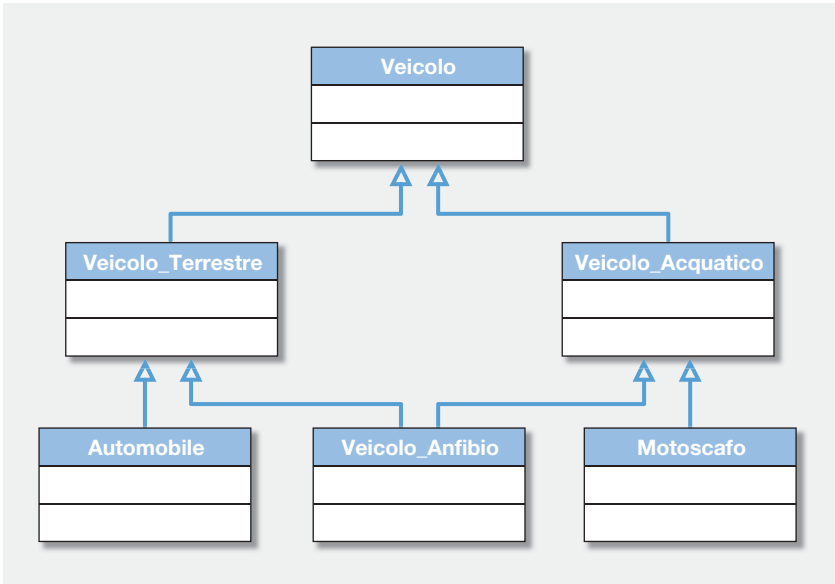


FIGURA **7**

OSSERVAZIONE Tutti i linguaggi di programmazione orientati agli oggetti supportano l'ereditarietà singola, ma solo alcuni consentono di realizzare l'eredità multipla (C++ è tra questi, ma Java, per esempio, la consente solo in una forma molto limitata tramite le cosiddette «interfacce»).

4.1 Polimorfismo

Il concetto di **polimorfismo** degli oggetti, tipico della programmazione OO, è strettamente collegato all'ereditarietà tra classi. Utilizzando il polimorfismo è possibile ottenere comportamenti e risultati diversi invocando gli stessi metodi a carico di oggetti diversi.

Il polimorfismo è fondato sulla possibilità di ridefinire nelle classi derivate i comportamenti originali dei metodi ereditati dalla classe base; in questo modo è possibile ottenere metodi in grado di operare in modo appropriato

su oggetti di diversa tipologia: sarà compito dell'ambiente di esecuzione decidere di volta in volta quale è il metodo corretto da utilizzare sulla base della specifica classe di cui l'oggetto è istanza.

Mediante il polimorfismo è possibile trattare gli oggetti istanza delle classi di una gerarchia derivata da una classe base come se fossero istanze della classe base stessa.

ESEMPIO

Dovendo scrivere del codice in un linguaggio di programmazione orientato agli oggetti per calcolare l'area di varie figure geometriche, è sensato definire una classe generale *Figura_Geometrica* dalla quale derivare le classi che rappresentano le particolari figure geometriche (*Quadrato*, *Rettangolo*, *Trapezio*, …). Il diagramma UML delle classi è quindi quello di FIGURA **8**.

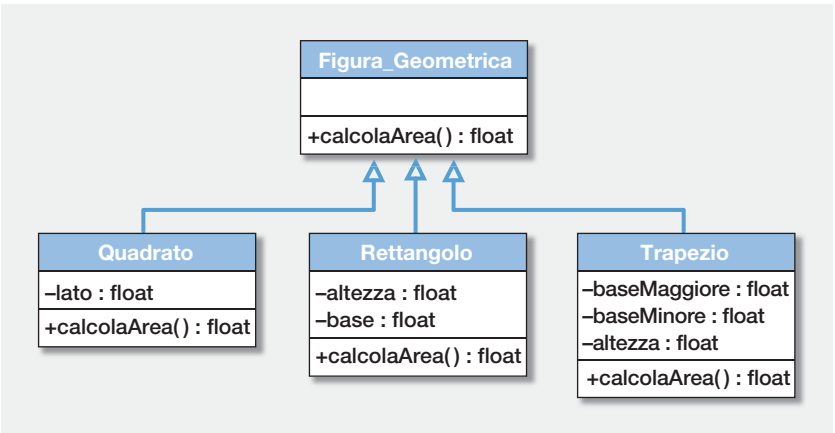


FIGURA **8**

Il metodo *calcolaArea* della classe *Figura_Geometrica* è ridefinito in tutte le classi derivate: questo significa che ogni classe implementa uno specifico algoritmo di calcolo dell'area mantenendo solo la firma del metodo ereditato. Per calcolare l'area di una figura geometrica disponendo di un oggetto *figura* istanza di una qualsiasi classe della gerarchia è quindi sufficiente invocare questo metodo:

```
…
area = figura.calcolaArea();
…
```

Applicando il polimorfismo il codice è in grado di determinare il metodo da invocare (quello definito nella classe *Quadrato*, oppure nella classe *Rettangolo*, o nella classe *Trapezio*) in funzione della classe di cui l'oggetto figura è istanza. Utilizzando un linguaggio di programmazione non orientato agli oggetti, e quindi privo del polimorfismo, il codice sarebbe invece risultato simile al seguente (l'operatore `isA` determina in questo caso se un oggetto è o meno istanza di una classe):

```
…
if (figura isA Quadrato)
   area = figura.calcolaAreaQuadrato();
else
   {
```

▶

```
      if (figura isA Rettangolo)
        area = figura.calcolaAreaRettangolo();
      else
        area = figura.calcolaAreaTrapezio();
    }
…
```

Il confronto evidenzia tutta la potenza del polimorfismo, che consente a oggetti istanza di sottoclassi diverse, derivate dalla stessa classe base, di avere la flessibilità di rispondere in modo differenziato allo stesso tipo di messaggio.

OSSERVAZIONE Il maggiore beneficio del polimorfismo è dato dalla conseguente facilità di manutenzione del codice. In riferimento all'esempio precedente, volendo aggiungere la figura geometrica triangolo, è sufficiente definire la nuova classe *Triangolo* derivandola da *Figura_Geometrica*, ma – se il metodo *calcolaArea* viene correttamente ridefinito nella classe *Triangolo* – non è in alcun modo necessario modificare il codice che lo invoca. Non disponendo invece del polimorfismo, per ogni singola invocazione del metodo di calcolo dell'area di una qualsiasi figura geometrica dovrebbe essere aggiunta un'ulteriore selezione per invocare il metodo corretto *calcolaAreaTriangolo*.

4.2 Metodi astratti e classi astratte

Il funzionamento dell'ereditarietà e del polimorfismo nei linguaggi di programmazione OO prevede esplicitamente la ridefinizione nelle sottoclassi derivate e i metodi definiti in una superclasse; in alcuni casi l'implementazione di un metodo in una classe base diviene puramente formale.

ESEMPIO

Il metodo *calcolaArea* della classe *Figura_Geometrica* dell'esempio precedente non è in grado di calcolare l'area di nessuna figura in particolare e probabilmente dovrà limitarsi a restituire un valore fittizio o nullo.

Una classe base può limitarsi a definire la firma di alcuni metodi (l'**interfaccia**) tralasciandone l'implementazione che riserva alle classi derivate. Metodi con questa caratteristica prendono il nome di **metodi astratti**. Una classe in cui uno o più metodi sono astratti **fattorizza** le operazioni comuni a tutte le sottoclassi dichiarandone i metodi senza implementarli; a partire da una classe di questo tipo non è possibile istanziare oggetti, perché risulterebbero indefiniti rispetto all'eventuale invocazione dei metodi astratti. Classi che presentano uno o più metodi astratti sono definite **classi astratte**: una classe astratta non viene definita allo scopo di istanziare oggetti, ma esclusivamente per derivarne sottoclassi che specificheranno (implementandole) le operazioni che essa dichiarate.
Nei diagrammi UML una classe astratta viene indicata scrivendone la denominazione in carattere corsivo.

Interfacce

Alcuni linguaggi di programmazione orientati agli oggetti consentono di definire, oltre alle classi, anche le interfacce: una **interfaccia** è analoga a una classe astratta, ma costituisce una pura specifica formale dei comportamenti, limitandosi a dichiarare i propri metodi senza implementarli (tutti i metodi di un'interfaccia sono quindi implicitamente astratti); inoltre non può prevedere attributi che non siano costanti.

Le interfacce possono essere estese – il termine tecnico è **implementate** – per ereditarietà, esattamente come le classi, ma possono essere generalmente strutturate in gerarchie di ereditarietà multipla anche nei linguaggi di programmazione, come Java, in cui questo meccanismo non è disponibile per le classi.

ESEMPIO

Sviluppando l'esempio relativo alla gerarchia dei veicoli a motore, possiamo affermare che:

- ogni veicolo ha un particolare tipo di propulsore;
- ogni veicolo si sposta in una specifica modalità;
- ogni veicolo si muove in un determinato ambiente (terraferma, acqua, aria, …).

Possiamo quindi affermare che non esiste un veicolo astratto generale, ma solo veicoli reali concreti; per esempio:

- l'automobile che ha un motore a scoppio, si sposta sulle ruote e si muove sulla terraferma;
- la nave che ha un motore marino, si sposta navigando e si muove sull'acqua;
- l'aereo che ha un motore a reazione, si sposta volando e si muove nell'aria.

L'introduzione di una classe base astratta *Veicolo* permette di fattorizzare – definire cioè una sola volta per tutte le classi derivate – gli aspetti comuni a tutti gli oggetti. Nel caso specifico possiamo stabilire che, per descrivere un veicolo reale, è in ogni caso necessario definirne il tipo di propulsore, la modalità di spostamento e l'ambiente in cui si muove. Per ogni specifico tipo di veicolo sarà possibile aggiungere le proprietà caratteristiche, ma in ogni caso non potremo prescindere dal definire almeno i tre aspetti fattorizzati per un veicolo generico (FIGURA **9**).

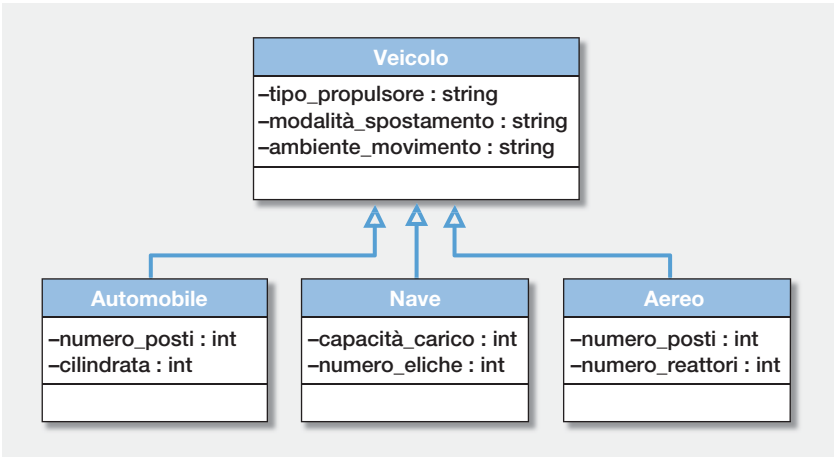


FIGURA **9**

5 Associazioni tra classi

Uno dei punti fondamentali della progettazione software OO consiste nel determinare come gli oggetti creati a partire dalle classi definite collaborino tra di loro. Nella programmazione OO gli oggetti interagiscono tra loro scambiando i messaggi che richiedono l'esecuzione di specifici metodi. Oltre al fatto che gli attributi di una classe possono essere del tipo definito da un'altra classe (associazione statica), sono gli scambi di messaggi che veicolano con i loro parametri le relazioni tra classi (associazioni dinamiche) la cui classificazione e documentazione è fondamentale nella fase di progettazione del software.
Pur esistendo la possibilità di rappresentare associazioni multiple tra classi, il caso più frequente è quello delle associazioni che sussistono tra coppie di classi (associazioni binarie) di cui prendiamo in esame le tipologie più significative:

- dipendenza;
- generalizzazione;
- composizione;
- aggregazione.

OSSERVAZIONE Nelle associazioni binarie tra classi si definisce **molteplicità** (o **cardinalità**) dell'associazione il numero di oggetti che per ogni classe partecipano all'associazione stessa. I valori possibili sono:

- molti (*);
- uno (1);
- zero o più (0..*);
- zero o più, fino al massimo di N (0..N);
- uno o più (1..*);
- uno o più, fino al massimo di N (1..N);
- da N a M (N..M).

5.1 Dipendenza

Nei diagrammi UML delle classi la **dipendenza**, o **associazione d'uso**, è un tipo di associazione in cui le eventuali variazioni a un elemento (il **fornitore**) possono avere influenza sull'altro elemento (il **cliente**). Questo accade, per esempio se una classe ha come tipo dei propri attributi un'altra classe, oppure se un metodo di una classe ha parametri del tipo definito da un'altra classe. La simbologia UML adottata per questo tipo di associazione è quella di una freccia tratteggiata orientata dalla classe cliente a quella fornitore.

ESEMPIO

Una ipotetica classe *Viaggio* può dipendere dalla classe *Automobile*, se una delle sue operazioni ha un parametro di tipo *Automobile*. Un'associazione di questo tipo si potrebbe interpretare come «*Viaggio* **utilizza** *Automobile*», oppure «*Viaggio* **dipende da** *Automobile*»: la classe *Viaggio* è l'entità cliente, mentre la classe *Automobile* è l'entità fornitore.
Nel diagramma UML delle classi l'associazione di dipendenza è una freccia tratteggiata che parte dalla classe *Viaggio* e arriva alla classe *Automobile* (FIGURA **10**).

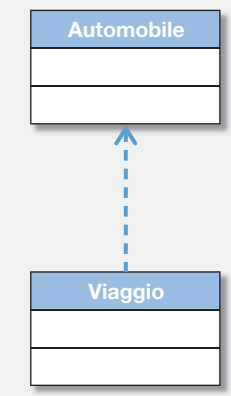


FIGURA **10**

OSSERVAZIONE L'associazione di dipendenza dell'esempio precedente indica che una eventuale modifica della classe *Automobile* potrebbe comportare una modifica della classe *Viaggio*. Alcuni tipi di modifiche alla classe *Automobile*, però, non comporteranno modifiche alla classe *Viaggio*, in particolare quelle relative alla parte privata, oppure l'aggiunta di nuove operazioni. Come sempre un attento ricorso al principio di *information hiding* consente di limitare i vincoli fornendo un'ampia possibilità di modifica dell'implementazione privata se l'interfaccia pubblica viene mantenuta inalterata.

5.2 Generalizzazione

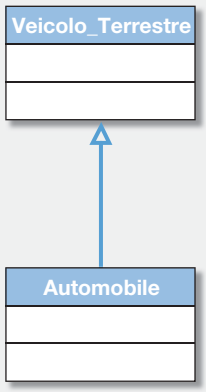


FIGURA **11**

L'ereditarietà tra classi realizzata dai linguaggi di programmazione OO è una associazione comune e nel contesto dei diagrammi UML è definita **generalizzazione**. La rappresentazione UML della generalizzazione consiste in una freccia con estremità triangolare che unisce la classe derivata alla classe base.

OSSERVAZIONE L'associazione tra classi in cui una generalizza l'altra è nota come relazione **is-A**: un oggetto istanza di una sottoclasse è infatti, grazie al polimorfismo, anche istanza della relativa superclasse.

ESEMPIO

Il diagramma UML di FIGURA **11** rappresenta la classe *Veicolo_Terrestre* come generalizzazione della classe *Automobile* (la classe *Automobile* è di conseguenza una «specializzazione» della classe *Veicolo_Terrestre*).

5.3 Composizione

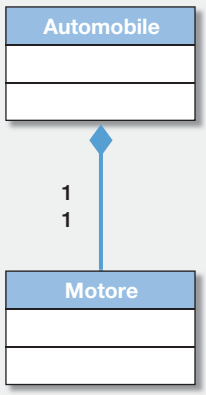


FIGURA **12**

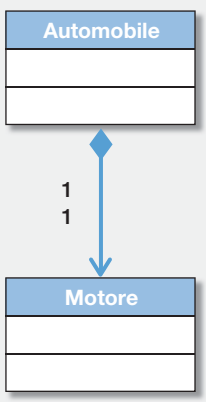


FIGURA **13**

Un tipo di associazione interessante è la **composizione**. Essa è un'associazione tutto-parti in cui la classe **tutto** governa il periodo di vita delle classi **parti**, che esistono esclusivamente come componenti del tutto. Questa associazione è nota come **has-A**, perché un oggetto (componente) è parte di un altro oggetto di tipo composito. La notazione UML per le composizioni è un segmento che collega le due classi in cui l'estremo rivolto alla classe componente (parte) è semplice, mentre l'estremo rivolto alla classe composita (tutto) è caratterizzato da un piccolo rombo pieno.

Agli estremi del collegamento può essere specificata la molteplicità dell'associazione, ovvero il numero di istanze della classe associate a ciascuna delle istanze della classe che si trova all'estremo opposto.

ESEMPIO

Il diagramma delle classi UML di FIGURA **12** rappresenta un'associazione di tipo compositivo dove viene stabilito che un oggetto di tipo *Automobile* **ha** un oggetto di tipo *Motore*.
La molteplicità 1 a 1 tra automobile e motore è data dal fatto che normalmente ogni automobile ha un motore e che ogni motore è di una sola automobile.

OSSERVAZIONE Se il collegamento verso la classe componente è rappresentato da una freccia con l'estremità aperta, esso rappresenta un indicatore di «navigabilità» che indica la possibilità di accedere la/e istanza/e della classe puntata dalla freccia a partire dall'istanza della classe che si trova all'altro estremo della freccia (questo normalmente significa che nella classe da cui parte la freccia ci sono uno o più attributi del tipo della classe a cui punta la freccia) (FIGURA **13**).

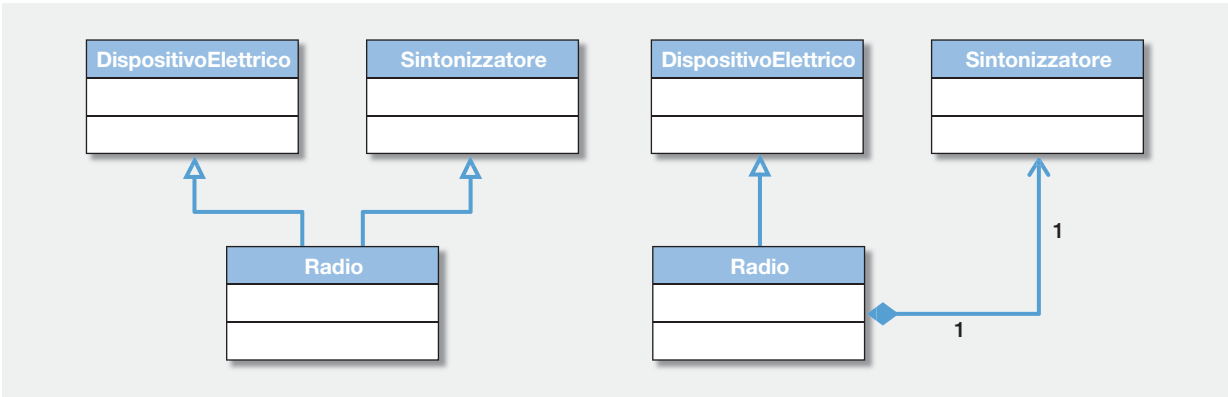


FIGURA **14**

OSSERVAZIONE Non sempre è facile capire se utilizzare un'associazione di tipo «is-A» (generalizzazione) o di tipo «has-A» (composizione). Per esempio, dovendo progettare una classe *Radio*, a partire dalle classi esistenti *Sintonizzatore* e *DispositivoElettrico*, è abbastanza ovvio che *Radio* debba derivare da *DispositivoElettrico*; ma non è altrettanto ovvio se essa debba anche derivare da *Sintonizzatore*, oppure avere come attributo un'istanza di *Sintonizzatore* (FIGURA **14**).
Una prima decisione dipende dalla molteplicità dell'associazione tra le due classi: se non è 1 a 1 – cioè se una radio può avere più di un sintonizzatore – è sicuramente da escludere la derivazione; in caso contrario, la scelta in genere è determinata dal contesto d'uso della classe, e in particolare dal vantaggio/svantaggio derivante dal fatto che un oggetto di classe *Radio* sia anche – mediante il ricorso al polimorfismo – un oggetto di classe *Sintonizzatore*.

5.4 Aggregazione

L'associazione di **aggregazione** è un'associazione tutto-parti, ma meno forte della composizione: una parte, infatti, può appartenere a più di una classe tutto, che può esistere anche indipendentemente dalle parti. La notazione UML per le aggregazioni è un segmento che collega le due classi in cui l'estremo rivolto alla classe aggregata (parte) è semplice, mentre l'estremo rivolto alla classe aggregante (tutto) è caratterizzato da un piccolo rombo vuoto.

ESEMPIO

Il diagramma delle classi UML di FIGURA **15** rappresenta un'associazione di tipo aggregativo dove la classe *Parcheggio* aggrega la classe *Automobile*, perché un parcheggio (il tutto) aggrega più automobili (le parti), pur esistendo indipendentemente da esse.
La molteplicità indicata per l'associazione – un parcheggio (1) per zero o più automobili (0..*) – riflette il fatto che il parcheggio può essere vuoto, o contenere una o più automobili. Potrebbe essere esplicitato il segnalatore di navigabilità ponendo una freccia aperta sul lato del collegamento che punta alla classe *Automobile* per indicare che la classe *Parcheggio* ha come attributo una «collezione» di oggetti di tipo *Automobile*.

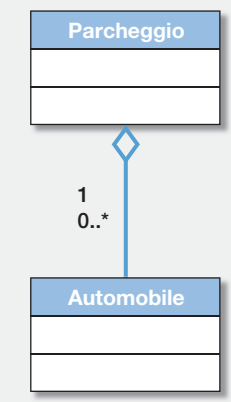


FIGURA **15**

OSSERVAZIONE Non sempre è semplice distinguere correttamente un'associazione di composizione da una di aggregazione: alcune situazioni possono infatti trarre in inganno. Un possibile criterio per distinguere l'aggregazione dalla composizione può essere quello di verificare se l'eliminazione di un oggetto composto implica naturalmente anche l'eliminazione dei suoi componenti: in caso affermativo si tratta di composizione, altrimenti di aggregazione.

Sintesi

OOP (*Object Oriented Programming*). La programmazione orientata agli oggetti è un paradigma di programmazione che prevede la definizione di oggetti software che interagiscono tra di loro attraverso lo scambio di messaggi.

ADT (*Abstract Data Type*). L'espressione *tipo di dato astratto* si riferisce a un tipo di dato completamente specificato, ma indipendente da una sua particolare implementazione; l'implementazione di un ADT è demandata allo sviluppatore del software sia per il contenuto informativo (i *dati* o *attributi*), sia per le operazioni specifiche associate al tipo di dato (*metodi*). Un ADT è caratterizzato dalla definizione di un nuovo tipo di dato, dalla definizione di un insieme di operazioni sul tipo di dato il cui comportamento è definito mediante condizioni prestabilite e che costituiscono la sua interfaccia, essendo l'unico meccanismo tramite il quale è possibile interagire con il dato stesso.

Information hiding. Il principio fondamentale di progettazione degli ADT è quello dell'**incapsulamento**, che stabilisce la netta separazione tra l'interfaccia esposta da un ADT all'esterno per essere utilizzato e la sua implementazione interna che deve invece rimanere nascosta. Un obiettivo importante del principio di *information hiding* è rendere invisibili all'esterno di un ADT le scelte implementative che possono essere soggette a modifiche, proteggendo in questo modo il codice che utilizza l'ADT dalle conseguenze dovute all'eventuale cambiamento di tali scelte. L'incapsulamento, inoltre, garantisce un corretto utilizzo del tipo di dato, prevenendo eventuali errori nella sua gestione da parte del codice che lo utilizza. Nella pratica della programmazione il principio di *information hiding* viene implementato ricorrendo a una tecnica che, in generale, prevede la classificazione degli attributi e dei metodi in *pubblici* o *privati*.

Classe. Nel contesto della OOP una classe rappresenta un modello formale per la descrizione di un certo tipo di oggetti definendone gli attributi, i metodi e le caratteristiche della sua interfaccia. A partire dalla definizione di una classe è possibile creare (*istanziare*) oggetti simili che condividono lo stesso insieme di attributi, lo stesso insieme di metodi e la stessa interfaccia.

Oggetto. Istanza di una classe che rappresenta un'entità autonoma con propri valori per gli attributi.

Costruttore. Metodo speciale mediante il quale si istanziano oggetti della classe.

UML (*Unified Modeling Language*). Formalismo grafico per la progettazione e la documentazione del software che rende disponibili diversi tipi di diagrammi per descrivere i vari aspetti di un sistema OO (diagramma delle classi, diagramma degli oggetti, diagramma dei casi d'uso, diagramma di sequenza, …).

Diagramma delle classi. Diagramma UML che consente di rappresentare graficamente una o più classi con le loro componenti (attributi e metodi) e le relazioni che intercorrono tra le classi.

Diagramma di sequenza. Diagramma UML che descrive uno scenario di interazione tra