Capitolo 6

Object Oriented in un chicco di grano

Buzzword Oriented Programming
BJARNE STROUSTRUP

Introduzione

Obiettivo del presente capitolo è introdurre brevemente le nozioni alla base del paradigma Object Oriented con particolare riguardo alla fase di disegno. Lungi dall'idea di voler affrontare l'argomento in maniera esauriente — esistono allo scopo diversi libri accademici, alcuni dei quali proposti nella bibliografia —, in questa sede presenteranno tematiche imprescindibili da un punto di vista più operativo.

Il consiglio per tutti coloro che ben conoscono il mondo Object Oriented, interessati unicamente allo Unified Modeling Language, è di procedere direttamente con la lettura del capitolo successivo.

Chiaramente lo UML non è l'Object Oriented, ma si tratta di un linguaggio grafico che permette di realizzare, documentare, ecc. modelli basati su tale paradigma… Anche se poi, nella realtà, visionare modelli realmente Object Oriented è una vera e propria chimera… Alcune persone sembrerebbero non comprendere che esiste qualche differenza tra inserire in un apposito diagramma qualche classe legata da relazioni e una vera e propria modellazione; così come non sempre è un modello valido il diagramma delle classi prodotto durante la fase di codifica. Tutti possono disegnare case o ponti — anche l'autore lo faceva fin dall'asilo — ma considerare questi "schizzi" come veri progetti è probabilmente un po' diverso…

Questo capitolo non è il risultato di una pianificazione iniziale, ma nasce da una necessità emersa nel corso della redazione dei capitoli successivi: un vero esempio di applicazione della tecnica del refactoring. Ci si è resi conto infatti di numerosi rimandi alle nozioni fondamentali dell'Object Oriented nonché di frequenti ripetizioni di concetti. Invece di continuare a citare brevemente le nozioni all'uopo, si è scelto di raggruppare tali concetti in un "mini" capitolo propedeutico che, per forza di cose, non può e non vuole considerarsi esaustivo e tanto meno a elevato grado di rigorosità. La speranza è che questo "Bignami", con tutte le limitazioni del caso, non risulti forviante per i lettori e magari possa risultare utile a coloro che sono interessati a rispolverare le nozioni basilari dell'Object Oriented da una prospettiva molto operativa... Teoria e pratica: in teoria non esiste alcuna differenza, in pratica sì.

Nozioni base

Oggetti e classi

Il percorso introduttivo delle nozioni dell'Object Oriented non poteva che prendere il via dai concetti di **oggetto** e **classe**. Il problema è che si tratta di nozioni che, se da un lato sono ben chiare a livello intuitivo, dall'altro sono ben più ardue da rendere rigorosamente. Come se non bastasse, si tratta di materia spesso abusata e fraintesa… Non a caso è stata citata la frase del maestro Bjarne Stroustrup ("Buzzword Oriented Programming").

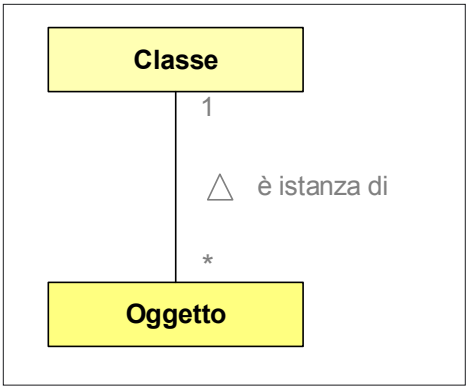
Una delle definizioni di oggetto più accurata è sicuramente una di quelle enunciate dal solito Booch: "un oggetto rappresenta un articolo (*item*), un'unità o un'entità individuale, identificabile, reale o astratta che sia, con un ruolo ben definito nel dominio del problema e un confine altrettanto ben stabilito".

Frequentemente in "elucubrazioni filosofiche", i termini di oggetto e classe tendono a essere utilizzati come sinonimi, sebbene i due concetti siano profondamente diversi: gli oggetti rappresentano istanze delle classi (fig. 6.1). Gli esperti, ovviamente, non hanno alcun problema nel discernere quando il termine "oggetto" viene utilizzato propriamente (si riferisce a una particolare istanza di una classe), e quando gli si attribuisce un significato più generale atto a indicare entità esistenti nello spazio del problema. Però, per i non addetti ai lavori, la mancanza di coerenza nella nomenclatura può generare qualche problema. Per esempio, i vari **diagrammi delle classi** che descrivono l'area business che il sistema dovrà automatizzare, sono denominati modelli a **oggetti** del dominio o del business (a seconda della versione presa in esame).

Benché le attività di analisi prevedano lo studio iniziale di entità presenti nel dominio, ossia degli oggetti "realmente esistenti o da esso traspiranti", l'obiettivo da conseguire è la sintesi, cioè la definizione dettagliata e formale che in termini Object Oriented è detta classe.

Quando si considera il problema di rappresentare formalmente il dominio o l'area business che il sistema da sviluppare dovrà in qualche misura risolvere, si focalizza l'at-

Figura 6.1 — *Diagramma delle classi utilizzato per mostrare la relazione esistente tra gli oggetti e le classi. A tempo di esecuzione esistono solo oggetti, ognuno dei quali è istanza di una e una sola classe. Durante la fase di sviluppo si definiscono le classi, che, a meno di particolari vincoli, originano diversi oggetti. In maniera* molto approssimata*, si può pensare al rapporto che intercorre tra un oggetto e la relativa classe come a quello esistente tra una variabile e il relativo tipo, in un qualsivoglia linguaggio di programmazione.*



tenzione sugli oggetti reali e sui concetti esistenti in tale mondo. L'obiettivo è rappresentare formalmente elementi del vocabolario, opportunamente relazionati, utilizzati nell'area oggetto di studio. Esempi tipici di classi di in un sistema bancario, sono le valute, i conti correnti, i clienti (spesso denominati *terze parti*), i contratti, le regole di pagamento (*settlement*), i messaggi, i *trade*, ecc.

Il concetto di valuta rappresenta una classe, mentre "Euro (EUR)", "Sterlina del Regno Unito (GBP)", "Dollaro americano (USD)", ne sono istanze.

Nel sistema di una generica agenzia viaggi, classi tipiche sono hotel, stanze, pacchetti turistici, crociere, voli aerei, clienti, vettori aerei, ecc. In un sistema per il commercio elettronico, classi specifiche potrebbero essere articoli, aziende produttrici, cataloghi, carrelli della spesa, utenti, carte di credito, e così via. Quindi, mentre per esempio l'entità volo aereo rappresenta la classe, il volo "AZ232 Roma-Londra" ne è un'istanza (magari complessa), così come il volo "BA546 Liverpool-Firenze". Gli oggetti menzionati rappresentano "cose" che "esistono" (più o meno tangibilmente) nell'area di business oggetto di analisi, e si differenziano, almeno dal punto di vista concettuale, da altri che invece sono istanze di classi che vivono nel modello di disegno, come per esempio `Vector`, `File`, `IOStream`, e così via. Queste ultime sono ancora classi, ma non vivono nello spazio del dominio, bensì sono utilizzate al fine di creare l'infrastruttura necessaria per realizzare i servizi richiesti al sistema. Bruce Eckel, nel magnifico libro *Thinking in Java*, afferma che "esiste una connessione stretta tra gli oggetti e i calcolatori: ogni oggetto somiglia, in

qualche modo, a un piccolo computer; possiede degli stati e delle operazioni che vi si possono invocare".

Una caratteristica particolarmente elegante dei linguaggi di programmazione Object Oriented consiste proprio nel permettere ai disegnatori di rappresentare il sistema in termini di "spazio del problema" piuttosto che in funzione di esigenze proprie dello spazio delle soluzioni, cosa che invece accade con il paradigma della programmazione procedurale. Quindi è possibile considerare una classe come un tool che permette di impacchettare insieme dati e funzionalità in "concetti", in modo che possano rappresentare appropriatamente un'idea appartenente allo spazio del problema, piuttosto che essere obbligati ad utilizzare forzatamente un idioma legato al mondo dei calcolatori. L'idea alla base dell'Object Oriented è far in modo che il disegno si conformi al problema reale attraverso l'introduzione di nuove specifiche classi, astrazione di concetti presenti nell'area di studio. Ciò permette di leggere il disegno — e quindi, in ultima analisi il codice — come se si trattasse della dichiarazione delle business rules presenti nel dominio del problema. Chiaramente nella realtà le cose non sono mai così lineari e ancora non è possibile leggere direttamente il disegno/codice come se fosse una dichiarazione del dominio del problema.

Chiaramente una stessa idea può essere rappresentata in modi diversi in funzione di molti aspetti e soprattutto dell'importanza posseduta nel contesto del dominio di studio. Per esempio, nella modellazione di un sistema di una generica organizzazione, l'entità città si presta a essere rappresentata attraverso poche informazioni come nome, codice di avviamento postale, nazione di appartenenza ecc., mentre in un sistema di informazioni turistiche, i dati di interesse potrebbero essere più articolati e legati a lingua parlata, valuta, percorsi turistici, alimentazione tipica, ecc. Ovviamente molte di queste ultime informazioni si presterebbero a essere rappresentate per mezzo di opportune relazioni con altre classi.

> Il problema nasce quando la stessa idea, nello stesso contesto, è astratta in modo diverso dipendentemente dal ruolo e delle capacità della persona. Per esempio, realizzando i modelli ad oggetti dell'area del dominio, molto spesso è difficile far comprendere al cliente le motivazioni alla base di relazioni di generalizzazione, o perché alcuni attributi siano presenti in una classe anziché in un'altra, ecc. Tipicamente per i clienti il concetto di classe coincide con i dati mostrati nelle schermate video. Quando questi problemi sono relegati al rapporto con i clienti la situazione è ancora gestibile, quando invece sono presenti anche nel team tecnico, allora la situazione si complica.

Un oggetto è quindi qualcosa che esiste (o "traspira") nel mondo concettuale e, come tale, se ne può parlare, eventualmente lo si può toccare o manipolare in qualche modo. "Ma cosa sono le *cose* del dominio del problema? Molte di queste cose probabilmente appartengono ad una delle seguenti cinque categorie: oggetti tangibili, ruoli, episodi, interazioni e specificazioni." [Booch].

Oltre all'insieme degli oggetti "strettamente" materiali, c'è poi tutta un'altra categoria che non esiste nel mondo reale, ma i cui elementi possono essere considerati come deriva-

ti da specifici oggetti tangibili, attraverso lo studio delle strutture e dei comportamenti. Si consideri per esempio un evento che interviene in un sistema e che è necessario gestire. Si consideri una transazione di un pagamento. È certamente un oggetto in termini informatici (più precisamente è un oggetto quando si verifica e una classe quando lo si modella), ma si tratta di un oggetto molto particolare. Di certo è difficile "toccarlo" eppure gli effetti generati possono avere importanti riflessi sulla vita dei titolari. Proprio per via della duplice natura degli oggetti — reale e "virtuale" — spesso ci si riferisce all'ambiente in cui "vivono" come il mondo concettuale, da cui la definizione che "un oggetto è un'entità che vive nel mondo concettuale".

Probabilmente il precursore del disegno Object Oriented è stato Aristotele, il quale, osservando le specie viventi, intuì il concetto di *classe*, ossia tutti gli oggetti sebbene unici, appartengono a determinati insiemi (classi) caratterizzati dal possedere stesse caratteristiche e comportamento. Tutti gli oggetti sono "istanze" di classi, ove per classe si intende un qualcosa che consente di descrivere formalmente proprietà e comportamento di tutta una categoria di oggetti simili. L'obiettivo è creare una corrispondenza biunivoca tra gli elementi del dominio del problema (oggetti che realmente esistono) e quelli dello spazio delle soluzioni: la difficoltà consiste proprio nel riuscire a descrivere formalmente, precisamente e completamente un'astrazione a partire dagli esempi delle relative istanze (corpo dell'astrazione). Quindi è necessario studiare un certo numero di oggetti simili al fine di individuarne le caratteristiche comuni che, in ultima analisi, sono attributi, operazioni e relazioni con altri oggetti. Per esempio il vocabolo "cane" (nome della classe) rappresenta un'astrazione molto potente di un "oggetto": un animale ben definito caratterizzato dall'avere proprietà strutturarli (quattro zampe, una coda, ecc.) e comportamentali (abbaiare, annusare, mordere l'osso, sotterrarlo, ecc.). Questa parola condensa in sé le caratteristiche (definizione della classe) condivise da tutta una categoria di elementi, accomunati dalle proprietà di avere quattro zampe, una coda, e così via (attributi) e di correre, abbaiare ecc. (metodi). La stessa classe `Cane` possiede diverse specializzazioni (`Alano`, `Dalmata`, `Levriero`, ecc.) che pur condividendo le medesime caratteristiche base, ne aggiungono altre specifiche (probabilmente i cani alani posseggono una eredità multipla: cani dalle dimensioni di cavalli). La cosa più importante è che di queste classi esistono delle istanze concrete, come `Scooby Doo`, `Oliver`, che sono istanze della specializzazione `Alano` della classe `Cane`.

> Da notare che nell'illustrazione dell'ultimo esempio si è volutamente nascosto il concetto di ereditarietà per non sovraccaricare la descrizione. In effetti l'animale cane appartiene alla gerarchia Essere Vivente > Animale > Mammifero > Cane > ...

Banalizzando estremamente e con tutte le inesattezze del caso, si può pensare alle classi come stampi per statue fittili e agli oggetti come le statue stesse realizzate attraverso le matrici. In questa banale similitudine, si trascura che anche l'impasto, il relativo colore, ecc. hanno il loro peso, ma l'importante è cercare di spiegarsi...

Elementi fondamentali di un oggetto

"Un oggetto possiede stato, comportamento e identità; la struttura e il comportamento di oggetti simili sono definiti nella loro classe comune; i termini di istanza e oggetto sono intercambiabili." [Grady Booch].

Stato

Gli oggetti, tipicamente, non vengono creati per permanere in un determinato stato; al contrario, durante il loro ciclo di vita transitano in una serie di fasi. Alcuni di essi sono vincolati a evolvere attraverso un insieme finito di fasi, mentre per altri è infinito oppure molto grande. Quindi, mentre per i primi (o meglio per la classe di cui sono istanza) può avere molto senso descrivere il diagramma degli stadi attraverso i quali i relativi oggetti possono transitare durante la propria vita, per gli altri l'esercizio è decisamente più complesso e non sempre fattibile e/o utile. Si consideri una classe che rappresenta una semplice lampadina, in questo caso l'insieme degli stati dei relativi oggetti prevede due soli elementi: acceso e spento. Si consideri ora una sua evoluzione, ossia una lampadina digitale con un numero ben definito di diverse intensità luminose. In questo caso l'insieme degli stati potrebbe prevedere: spenta, accesa intensità 1, accesa intensità 2, ..., accesa intensità max.

Altri oggetti, invece, durante l'arco della propria vita evolvono attraverso una serie di stati, che però non sono numerabili. Si consideri per esempio un sistema di illuminazione delle stanze la cui funzione sia accendere/spegnere i vari faretti in funzione del numero di persone presenti nelle stanze. Sebbene questo numero sia delimitato (il numero massimo di persone stipabili all'interno della stanza), non è conveniente indicare i vari stati dell'oggetto (una persona, due persone, tre persone, …).

Un'ultima categoria è costituita dagli oggetti il cui stato è (teoricamente) infinito. Per esempio, si consideri un'estensione del sistema precedente, in cui la decisione di accendere e/o spegnere l'illuminazione dipenda anche dal valore dell'intensità luminosa segnalata da un apposito oggetto (sensore). In questo caso, il dominio dei valori dei dati forniti sarebbe teoricamente infinito: si tratterebbe della trasposizione digitale di grandezze fisiche (segnali continui per definizione). In pratica la differenza tra due misure successive non è infinita, bensì è legata alla sensibilità del trasduttore (dispositivo atto a tradurre grandezze fisiche in segnali di altra natura, tipicamente, elettrica), al numero di bit del convertitore analogico/digitale, ecc.

Lo stato di un oggetto è molto importante poiché ne influenza il comportamento futuro. Gli oggetti, almeno loro, hanno una certa memoria storica. Tipicamente, sottoponendo opportuni stimoli a un oggetto (invocazione dei metodi, o invio di messaggi se si preferisce), questo tende a reagire, nella maggior parte dei casi, in funzione del suo stato interno. Si consideri l'esempio della lampadina elementare. Se una sua istanza si trova nello stato di accesa e ne viene richiesta nuovamente l'accensione (`turnOn()`), nulla accade, così come nella versione digitale a diverse luminosità, se è spenta e si tenta di variarne l'intensità

luminosa, nuovamente, nulla accade. Un ennesimo esempio è fornito dal lettore CD: se si preme il tasto di play senza aver inserito un CD, nulla accade (viene generata un'eccezione), mentre la pressione dello stesso tasto, con CD inserito, avvia il suono della musica. Pertanto il comportamento di molti oggetti è influenzato dal relativo stato. In queste circostanze l'ordine con cui ne vengono invocati i messaggi è, tipicamente, importante. Se si inserisce il CD e si preme il tasto play si assiste a uno specifico comportamento, che è diverso da quello generato dalla sequenza: pressione del tasto play e inserimento del CD.

Lo stato di un oggetto è un concetto dinamico e, in un preciso istante di tempo, è dato dal valore di tutti i suoi attributi e dalle relazioni instaurate con altri oggetti (che alla fine sono ancora particolari valori, indirizzi di memoria, attribuiti a specifici attributi).

Come si vedrà successivamente, è molto importante nascondere quando possibile lo stato di un oggetto al resto del mondo. Sicuramente deve esserne sempre nascosta l'implementazione (principio dell'*information hiding*) e quando possibile anche lo stato stesso (minimizzare l'accoppiamento di tipo). Quest'ultima possibilità dipende ovviamente dagli obiettivi (responsabilità) della classe. Se per esempio una classe rappresenta un sensore atto a valutare la temperatura del reattore nucleare di una centrale atomica, potrebbe aver senso comunicare all'esterno lo stato dei relativi oggetti.

Comportamento

Una volta studiato e formalizzato lo stato di un oggetto si è effettuato un passo in avanti nel processo di astrazione, ma non ancora è sufficiente per la completa descrizione dello stesso. Molto importante è analizzarne anche il comportamento.

Un oggetto non solo non viene creato per essere lasciato oziare in uno specifico stato, ma neanche per lasciarlo morire di solitudine. Tipicamente un oggetto interagisce con altri scambiando messaggi, ossia rispondendo agli stimoli provenienti da altri oggetti (richiesta di un servizio) e, a sua volta, inviandoli ad altri al fine di ottenere la fornitura di "sottoservizi" necessari per l'espletamento del proprio. Quindi, il comportamento di un oggetto è costituito dalle inerenti attività (operazioni) visibili e verificabili dall'esterno. Come visto poc'anzi, lo scambio di messaggi, generalmente, varia lo stato dell'oggetto stesso. In sintesi "il comportamento stabilisce come un oggetto agisce e reagisce, in termini di cambiamento del proprio stato e del transito dei messaggi." [Booch].

Un'operazione è una qualsiasi azione che un oggetto è in grado di richiedere a un altro al fine di ottenere la reazione desiderata. Per esempio, un oggetto `ContoCorrente` potrebbe richiedere l'inserimento di un credito in un altro oggetto variandone lo stato, così come potrebbe richiedere l'estratto conto, senza apportare modifiche allo stesso oggetto. È evidente che la relazione esistente tra stato di un oggetto e comportamento è di mutua dipendenza: è possibile considerare "lo stato di un oggetto, in un certo istante di tempo, come l'accumulazione dei risultati prodotti dal relativo comportamento", il quale, a sua volta, dipende dallo stato in cui si trovava l'oggetto all'atto dell'esecuzione del "comportamento".

Identità

L'identità di un oggetto è la caratteristica che lo contraddistingue da tutti gli altri. Spesso ciò è dato da un valore univoco. Per esempio un oggetto `ContoCorrente` è identificato dal relativo codice, da una persona, dal codice fiscale, e così via.

Tutti gli oggetti indossano un'interfaccia

> Nella consultazione dei seguenti paragrafi dedicati ai concetti di interfaccia si presti bene attenzione a non disorientarsi. In effetti con questo termine si indicano diversi concetti non solo nella comunità dell'Object Oriented ma anche in quella più vasta dell'informatica in generale. In particolare e brevemente, nel presente paragrafo si fa riferimento a una caratteristica intrinseca posseduta da tutte le classi, data dall'elenco dei metodi e attributi non privati che permette di definire responsabilità ed estensibilità degli oggetti istanza della classe. Nei paragrafi successivi si fa riferimento al concetto UML e Java di interfaccia ossia come definizione **pura** di tipo. Anche le classi permettono di definire un tipo, ma a differenza delle interfacce, vi definiscono anche l'implementazione.

Come visto nei paragrafi precedenti, una volta definita una nuova classe, a meno di vincoli particolari (vedi per esempio il pattern Singleton), è possibile dar luogo a quante istanze (oggetti) della classe si vuole. Chiaramente avrebbe ben poco senso dar vita a nuovi oggetti per poi lasciarli vivere isolatamente.

Sebbene ogni oggetto sia, potenzialmente, in grado di realizzare specifiche funzionalità dall'inizio alla fine, tipicamente, non lo fa per propria iniziativa, bensì come risposta a esplicite richieste da parte di altri oggetti (detti messaggi). Ciò è possibile poiché esiste un meccanismo che permette a un oggetto di richiedere a un altro di "fare qualcosa".

Ogni oggetto è in grado di soddisfare solo tipi ben definiti di richieste, specificati dalla propria "interfaccia". Pertanto in questo contesto, con il termine interfaccia non si fa riferimento a elementi come per esempio il costrutto *interface* del linguaggio Java, dotati esclusivamente della firma dei propri metodi, bensì alla caratteristica intrinseca posseduta da ogni classe: l'elenco di metodi e attributi non privati corredati dalla relativa firma (ciò che nella fig. 6.2 è stato rappresentato dalla classe `ProprietàComportamentali`).

> Per essere più precisi, in tutti i linguaggi Object Oriented che prevedono visibilità *protected* (indica metodi e attributi di una classe visibili solo dalle classi ereditanti), è possibile esprimersi in termini di due versioni di interfaccia: quella "visibile" a tutte le classi (elenco di metodi e attributi pubblici) e quella relativa alle sole classi ereditanti (costituita dai metodi e dagli attributi pubblici e protetti). Volendo si potrebbe complicare ulteriormente la situazione considerando anche la visibilità di tipo *package* che quindi genererebbe una terza versione di interfaccia contenente le due precedenti. Si ricordi che in UML un elemento con visibilità *package* (indicata con il carattere tilde ~) di una classe *X* è visibile

da tutte le classi appartenenti allo stesso package (o a uno da esso annidato qualsiasi livello) della classe *X*.

Volendo è possibile considerare l'interfaccia propria di una classe con una vera e propria "interface" non dichiarata, incorporata nella classe stessa. Pertanto le richieste che un oggetto può soddisfare sono specificate dalla propria interfaccia, o, meglio, dall'interfaccia della classe di appartenenza. L'interfaccia, anche se implicita, rappresenta un contratto stipulato tra gli oggetti fornitori e quelli utilizzatori di servizi: in essa vengono condensate tutte le assunzioni che gli oggetti client fanno circa quelli di cui utilizzano i servizi (server). Chiaramente l'interfaccia determina la o dipende dalla struttura interna degli oggetti fornitori di servizi.

Figura 6.2 — *In questo (meta) modello sono distinte più nettamente le caratteristiche a tempo di implementazione* (`Classe`, `ProprietàComportamentali` *e* `ProprietàStrutturali`) *da quelle a tempo di esecuzione* (`Oggetto`, `Identità` *e* `Stato`). *Le proprietà comportamentali rappresentano l'insieme dei metodi esposti da una classe e quindi invocabili da parte di oggetti istanze di altre classi, mentre quelle strutturali rappresentano l'insieme degli attributi. Per quanto concerne la notazione, per il momento si consideri il diamante pieno* (relazione di composizione) *come una relazione strutturale molto forte tra due entità, di cui le istanze della classe con il diamante rappresentano il concetto generale costituito dalle istanze delle altre classi associate.*

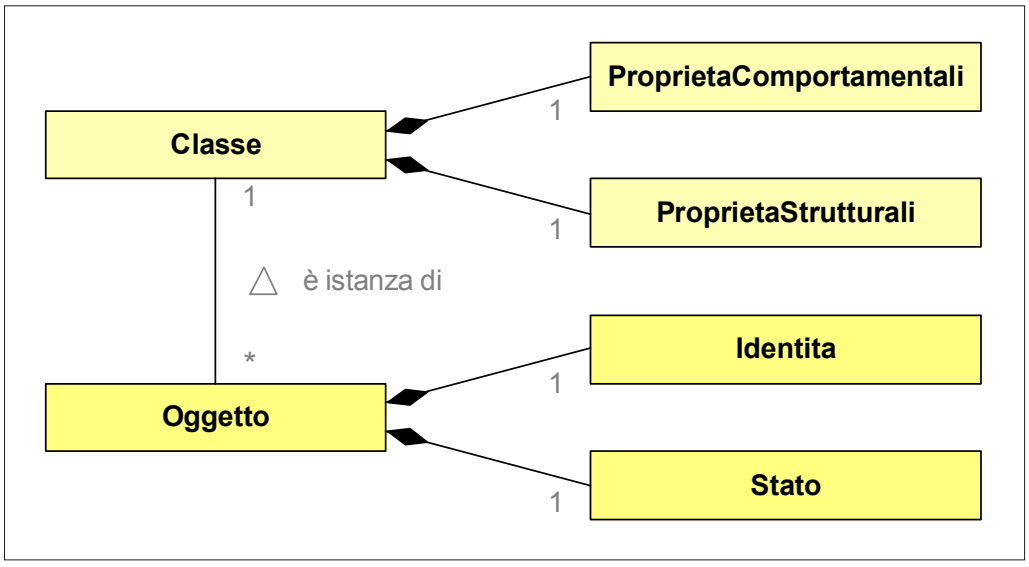
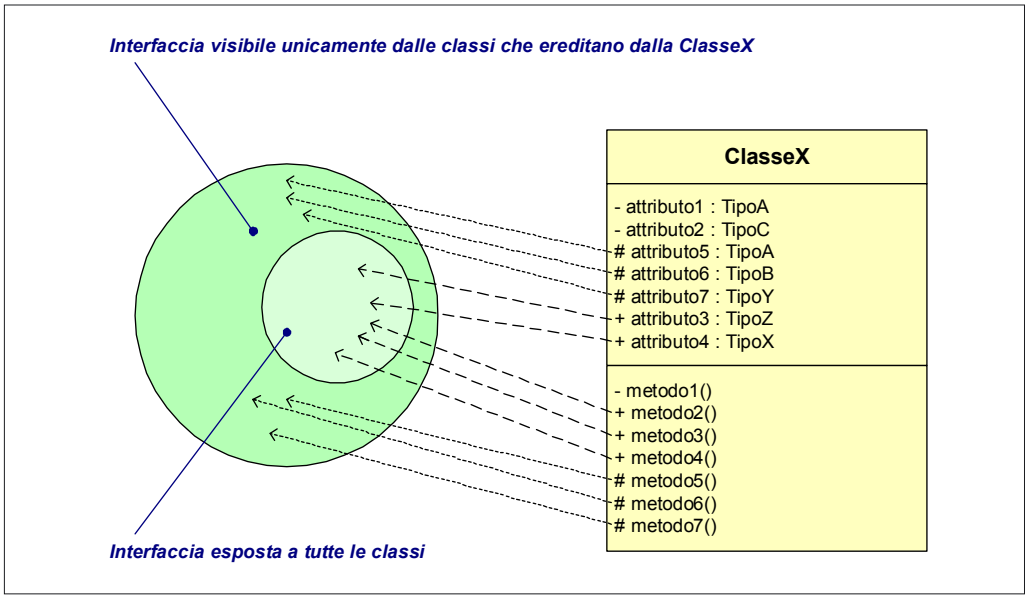


Figura 6.3 — *Le due versioni di un'interfaccia. Come da standard UML il segno meno (-) posto davanti a un metodo o attributo ne indica una visibilità privata e pertanto l'elemento non appartiene all'interfaccia esposta dalla classe. Il segno più (+) indica una visibilità pubblica e quindi l'elemento appartiene all'interfaccia visibile da tutti gli oggetti istanza di classi relazionate, in qualche maniera, a quella che possiede l'elemento. Il segno diesis (#) rappresenta una visibilità protetta e quindi i relativi elementi appartengono all'interfaccia per così dire di eredità.*



L'invocazione di un metodo di un oggetto (per motivazioni storiche derivanti dal linguaggio SmallTalk), tecnicamente, è considerato come l'invio di un apposito messaggio allo stesso oggetto.

Si consideri, per esempio, un sistema di riscaldamento di una casa, pilotato centralmente da un computer. Ogni radiatore potrebbe essere rappresentato da un'istanza di un'apposita classe (`Radiator`), i cui metodi eseguibili potrebbero essere:

`turnOn`	(avviamento)
`turnOff`	(spegnimento)
`turnFanOn`	(attivazione della ventola)
`turnFanOff`	(disattivazione della ventola)
`increasePower`	(aumento potenza)
`decreasePower`	(decremento potenza)

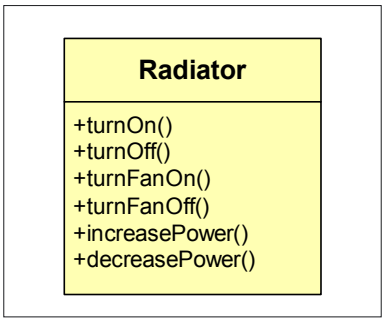
L'interfaccia di un oggetto è spesso definita *protocollo*. In effetti stabilisce i servizi offerti e la "sintassi" (firma dei vari metodi) con cui utilizzarli. Tipicamente, in oggetti non banali, il protocollo può essere ripartito in gruppi di comportamento, definiti ruoli. Questi rappresentano delle "maschere" che un oggetto può indossare ("Uno, Nessuno e Centomila", magari "nessuno" sarebbe un po' difficile) per stipulare contratti con altri oggetti. Per esempio i dipendenti di una certa organizzazione, potrebbero essere modellati attraverso una specifica classe denominata `Employee`. Ora una determinata istanza di questa classe, in una particolare relazione con i progetti, potrebbe recitare il ruolo di manager, caratterizzato da particolari proprietà, per questioni amministrative potrebbe recitare il generico ruolo del dipendente, e così via.

L'interfaccia propria di un oggetto è il luogo in cui sono specificate le assunzioni che oggetti cliente possono fare circa le istanze della classe.

Interfaccia (Java/UML)

In questo paragrafo con il termine interfaccia si fa riferimento al concetto a cui normalmente si è portati a pensare: ossia al costrutto che permette di definire un "tipo puro" (in quanto non direttamente istanziabile e quindi senza implementazione) attraverso la definizione di un insieme di operazioni identificato da un opportuno nome. In altre parole si fa riferimento al costrutto interface del linguaggio di programmazione Java e alla metaclasse estendente il classificatore nel metamodello UML. Per essere precisi, sebbene il concetto sia lo stesso, esiste una sottile differenza tra la versione Java e quella UML: nel linguaggio Java un'interfaccia può possedere attributi, sebbene questi siano automaticamente static e final (la versione Java del concetto di costante). Al fine di evitare ogni possibile confusione alla fonte, in questo paragrafo ci si riferirà al concetto di interfaccia con i termini di *interfaccia esplicita*.

Figura 6.4 — *Rappresentazione UML della classe* `Radiator`. *Indipendentemente dalla presenza o meno di un'interfaccia esplicita, la classe* `Radiator` *ne possiede una implicita data dalla lista di metodi e attributi non privati, corredati dalla relativa firma.*



Un'interfaccia è un insieme, identificato da un nome, di operazioni (corredate dalla firma) che caratterizzano il comportamento di un elemento. Si tratta di un meccanismo che rende possibile dichiarare esplicitamente le operazioni visibili dall'esterno di classi, componenti, sottosistemi, ecc. senza specificarne la struttura interna. L'attenzione è quindi focalizzata sulla struttura del servizio esposto e non sull'effettiva realizzazione (separazione tra la definizione di tipo e l'implementazione).

Per questa caratteristica, le interfacce si prestano a demarcare i confini del sistema o del componente a cui appartengono: espongono all'esterno servizi che poi altre classi (interne) hanno la responsabilità di realizzare fisicamente. Qualora una classe implementi un'interfaccia, deve necessariamente dichiarare (o eventualmente ereditare) tutte le operazioni definite da quest'ultima. Le interfacce non possiedono implementazione o stati: dispongono unicamente della dichiarazione di operazioni, definita *firma*, e possono essere connesse tra loro tramite relazioni di generalizzazione. Visibilità private dei relativi metodi avrebbero ben poco significato, sebbene sia sempre possibile imbattersi in programmatori intenti nella disperata impresa di dichiarare metodi privati e pertanto visibili solo all'interno — non esistente — di un'interfaccia. Grazie al cielo i compilatori non permettono la dichiarazione di visibilità non compatibili con il contesto.

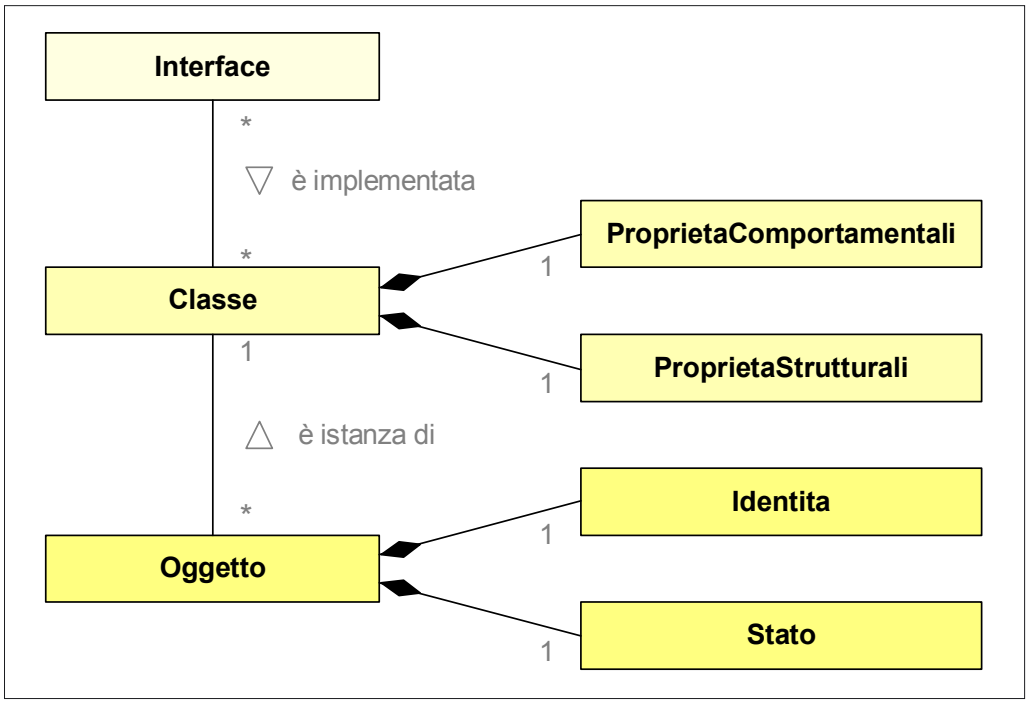
Una singola interfaccia dovrebbe dichiarare un comportamento circoscritto, ben definito, a elevata coesione e, quindi, elementi con comportamento complesso potrebbero realizzare diverse interfacce.

Il concetto di interfaccia rappresenta un'astrazione di estrema importanza per il disegno di modelli Object Oriented: è la chiave per la realizzazione di sistemi eleganti, flessibili, in grado di agevolare l'assorbimento dei famosi *change requirements* (variazione dei requisiti), ecc. La sua centralità è addirittura cresciuta esponenzialmente con l'introduzione dei sistemi component-based, caratterizzati da una ancora più netta separazione tra i servizi esposti da un componente e la relativa implementazione.

Il concetto di interfaccia rende possibile un insieme di meccanismi, come l'aggiunta (apparentemente) indolore di nuove funzionalità, la rimozione di altre, la sostituzione di componenti con altri più moderni (magari versioni più efficienti che rispondono meglio alle nuove richieste dei clienti, …), la realizzazione di Framework, ecc. Queste peculiarità derivano dal fatto che l'interfaccia costituisce un vero e proprio strato di indirezione tra le classi che la implementano e la restante parte del sistema. Quindi, gli oggetti che interagiscono con altri facendo riferimento alle relative interfacce delle classi di appartenenza vedono questi ultimi solo attraverso quanto dichiarato nell'interfaccia stessa, senza possedere alcuna conoscenza diretta delle classi che la implementano.

Una classica similitudine, spesso riportata per chiarire il concetto di interfaccia, è relativa agli slot di espansione dei computer. Gli slot possono essere visti come la definizione formale di un'interfaccia (volendo essere precisi gli slot sono più potenti: possiedono condizioni pre-, post- e durante l'utilizzo), il computer è il sistema software (qui lo sforzo di immaginazione richiesto è piuttosto ridotto) mentre le scheda rappresenta la classe (pro-

Figura 6.5 — *Nel modello illustrato sono presenti i tre livelli di astrazione di un oggetto: l'interfaccia che costituisce la specificazione, la classe che ne rappresenta l'implementazione e l'oggetto che ne rappresenta l'immagine a tempo di esecuzione. Come si può notare, una classe può implementare diverse interfacce così come un'interfaccia può essere implementata da diverse classi.*



babilmente sarebbe più opportuno esprimersi in termini di package) che realizza i servizi esposti dall'interfaccia. Il computer è in grado di utilizzare un certo numero di nuovi dispositivi, la cui interfaccia però deve uniformarsi a quella prevista dai relativi zoccoli degli slot. In caso di necessità è possibile sostituire una scheda con una più moderna (si consideri il caso di una scheda grafica): fintantoché l'interfaccia resta la stessa il tutto continua a funzionare perfettamente. È anche possibile rimuovere una scheda (un intero servizio) senza sostituirla e, nei limiti dettati dall'importanza del servizio rimosso, il tutto continua a funzionare. Sebbene ormai da tanto tempo esista il concetto del *Plug & Play* (ribattezzato dai tecnici esperti della materia *Plug & Pray*), inserire fisicamente una nuova interfaccia, sostituirne una con un'altra più moderna, o semplicemente eliminarla, senza dover eseguire ulteriori funzionalità aggiunte è qualcosa che nella pratica non accade quasi mai. Tipicamente è necessario eseguire qualche operazione di configurazione. Allo stes-

so modo, anche nei sistemi software, tipicamente è necessario eseguire alcune operazioni supplementari, come dichiarare l'esistenza in una nuova classe in un apposito file di configurazione, modificare un factory per la creazione delle relative istanze, ecc. In conclusione della metafora, si può pensare che ogni qualvolta nel disegno di un sistema si introduce un'interfaccia è come se si installasse uno zoccolo per schede di espansione in un PC e quindi, virtualmente, si realizza un punto di "plug-in" in cui è possibile cambiare funzionalità, aggiornare la scheda, aggiungere altri servizi ecc.

Sintetizzando, un'interfaccia è vista dal sistema come un protocollo predefinito da utilizzarsi per interagire con le classi che fisicamente la implementano. Ciò permette di realizzare concetti dell'ingegneria del software come minimo accoppiamento, i cui vantaggi sono ben noti e legati, principalmente, alla riduzione delle dipendenza tra le diverse componenti del sistema (sistemi flessibili, quindi più recettivi alle modifiche, maggiore possibilità di riutilizzo del codice, ecc.).

> Per l'attribuzione del nome alle interfacce, esistono diverse convenzioni. Alcuni preferiscono premettere la lettera maiuscola `I` mentre altri prediligono lasciare il nome inalterato all'interfaccia per poi aggiungere il suffisso `Impl` alla classe che la implementa (questa convenzione funziona bene solo nei contesti in cui ogni interfaccia esposta possegga esattamente una classe che la implementi). Le particelle grammaticali candidate per rappresentare il nome delle interfacce sono, come per le classi, i sostantivi. Talune volte si utilizzano anche i verbi, qualora la classe serva per definire in maniera astratta una o più operazioni. Il primo caso tende a implicare che ci si riferisce ad un accesso a dati (`ITicket`, `IHotel`), mentre un verbo rappresenta la richiesta di una computazione (`IBooking`, `ICheckAvailability`). Quest'ultima in generale non è un'ottima pratica, in quanto è sufficiente avere la necessità di aggiungere un ulteriore metodo che il nome dell'interfaccia rischia di diventare inconsistente. Pertanto è sempre opportuno utilizzare un sostantivo (`IBooker`, `IAvailabilityChecker`, ecc.).

Conversando con personale di varie organizzazioni, l'autore ha avuto modo di constatare come alcuni tecnici junior tendano a incontrare problemi nel far sposare il disegno Object Oriented dei sistemi con concetti quali estensibilità, flessibilità, adattamento (nel significato di "customizzazione"), ecc. Ciò non è completamente incomprensibile: la naturale tendenza del disegno Object Oriented è organizzare il sistema in una miriade di oggetti specializzati che interagiscono per realizzare vari servizi (sebbene non sia infrequente imbattersi in sistemi dotati di poche "macro" classi "tuttofare"). In altre parole ogni scenario di utilizzo è realizzato attraverso la collaborazione in stretta sequenza temporale di oggetti. In disegni privi di interfacce, si genera una mancanza di flessibilità dovuta al fatto che le classi che partecipano ad associazioni o che ricevono messaggi, sono dichiarate esplicitamente nel codice. In sostanza ogni oggetto si riferisce esplicitamente a un altro. In queste situazioni si capisce bene che aggiungere una nuova classe oppure modificare l'implementazione di un'altra può costituire un problema: è necessario individuare tutti i

riferimenti alla classe e quindi modificare il codice. Poiché poi la creatività di alcuni tecnici è sconfinata, spesso il problema viene arginato facendo ereditare la nuova da quella da sostituire. Ciò chiaramente non sempre risolve il problema… Cosa fare in caso di una nuova funzionalità che deve coesistere con quella predefinita? Oppure come risolvere il problema qualora si utilizzi un linguaggio come Java che non prevede ereditarietà multipla nel caso in cui la classe abbia necessità intrinseche di ereditare? Chiaramente la soluzione consiste nell'utilizzare le interfacce. Il livello di astrazione offerto permette di ignorare lo specifico oggetto con il quale si interagisce, fintantoché la relativa interfaccia sia rispettata.

L'utilizzo delle interfacce permette anche di arginare relativamente l'eterno problema del cambiamento continuo dei requisiti. Infatti, analizzando il disegno, è possibile evidenziare classi/packages la cui probabilità di subire modifiche sia elevata e isolarle, attraverso l'introduzione di opportune interfacce, dal resto del disegno. Questa tecnica, in molte situazioni, potrebbe fornire un buon meccanismo per assorbire le modifiche: la reingegnerizzazione del sistema potrebbe vertere, molto frequentemente, sulle classi isolate dalle relative interfacce. In questi casi, l'aggiornamento non inciderebbe sulla restante parte della struttura che continuerebbe a vedere unicamente l'interfaccia. In alcuni sistemi, considerata la frequenza del cambiamento delle idee da parte degli utenti, la tecnica potrebbe portare a circondare tutte le classi con diverse interfacce. Probabilmente si tratterebbe di un sistema molto flessibile ma con un costo di realizzazione di qualche ordine di grandezza superiore a quello iniziale.

A questo punto, considerati i vantaggi generati dall'uso delle interfacce, perché non disseminarle ovunque nel disegno del sistema? Le risposte sono semplici:

- il sistema necessiterebbe di tempi e quindi costi di sviluppo di qualche ordine di grandezza superiore (inserire interfacce tipicamente non è sufficiente: è necessario introdurre meccanismi generici per creare e gestire gli oggetti istanza delle classi protette dalle interfacce, come per esempio factory);

- spesso l'eccessiva flessibilità è sorgente di caos;

- in genere, il coefficiente di flessibilità di un sistema è inversamente proporzionale alla chiarezza e semplicità del relativo disegno (caratteristiche sempre molto apprezzate).

In merito al primo punto, non è infrequente analizzare modelli in cui siano state introdotte correttamente delle interfacce, salvo poi eliminarne tutti i vantaggi associando le classi direttamente a quelle che implementano l'interfaccia e non all'interfaccia stessa (fig. 6.6). Questo punto verrà dettagliato nel Capitolo

8. Per adesso basti pensare che il problema risiede nel fatto che sebbene le classi possano riferirsi in maniera astratta a interfacce, a tempo di esecuzione nel sistema esistono unicamente oggetti. Pertanto i riferimenti alle interfacce dovranno essere sostituiti da oggetti concreti istanza di classi che implementano le varie interfacce. Si pone quindi il problema di selezionare in modo "astratto" la particolare classe che implementa l'interfaccia di cui creare un'istanza, e quindi crearla. Si tratta ovviamente di un problema noto e risolto elegantemente dal sottoinsieme dei pattern detti creazionali (Creational Patterns [BIB04]).

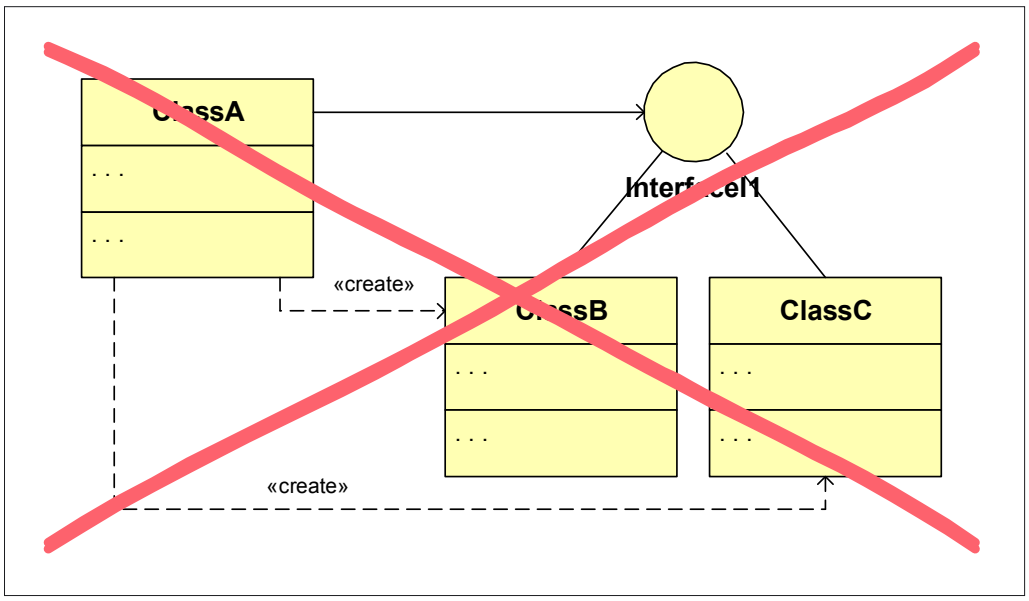
Astrazione

L'attitudine all'astrazione è una delle proprietà tipiche del cervello umano che permettono di minimizzarne le limitazioni nell'affrontare la complessità dei sistemi. Si tratta di una caratteristica che gli individui iniziano a sviluppare fin dai primi anni di vita e che poi continuano a migliorare con il passar del tempo (salvo molteplici eccezioni). Considerata le difficoltà della mente umana nel gestire la complessità dei sistemi nella loro interezza, non resta che rassegnarsi a ignorarne i dettagli per concentrarsi su un modello idealizzato e generalizzato. La qualità e quantità di dettagli da trascurare, chiaramente, dipende dal livello di dettaglio a cui si è interessati.

In generale l'atto dell'astrarre consiste nell'individuare similitudini condivise tra oggetti, processi ed eventi appartenenti alla vita reale e nella capacità di concentrarsi su queste, tralasciando momentaneamente le differenze. In sostanza si tratta di un meccanismo che permette di enfatizzare alcuni aspetti del sistema e di trascurarne altri in funzione degli obiettivi previsti e quindi del livello di dettaglio confacente. Chiaramente il punto critico consiste nel saper riconoscere gli aspetti significativi in una particolare fase, o per uno specifico modello, da quelli che invece possono essere in quel caso trascurati. Il criterio di distinzione, però, è soggettivo e dipende dai fini che si desidera raggiungere. Per esempio un acquirente medio pensa a un'automobile in termini di design, colore, optional, affidabilità, ecc.; un'istanza di Paperon de' Paperoni ne valuta i consumi, le tasse, ecc.; un meccanico è interessato al numero di cilindri, ai relativi litri, alla coppia, alla potenza, ecc.; un playboy vede la macchina in termini di possibili "abbordaggi", e così via. Pertanto ciascuno di questi personaggi, quando parla dello stesso concetto di autovettura, costruisce nel proprio cervello un modello completamente diverso della stessa entità: ciò che tipicamente viene definito proiezione.

L'obiettivo centrale del processo di astrazione è la produzione di un concetto (modello) che permetta di identificare in maniera univoca, completa e semplificata l'entità a cui si riferisce. Deve essere, pertanto, in grado di distinguere l'oggetto di riferimento da tutti gli altri, evidenziarne i confini, le caratteristiche proprie, ecc. A tal fine, il processo di astrazione è, intrinsecamente, focalizzato sulla vista esterna degli oggetti, allo scopo (special-

Figura 6.6 — *Esempio di una situazione di mancata astrazione nella creazione degli oggetti istanza di classi che implementano un'interfaccia. Da notare che non sempre ciò è un errore. Sebbene la classe* `ClassA` *preveda, in linea teorica, la possibilità di trattare in modo astratto oggetti istanza di classi che implementano l'interfaccia* `Interface1`*, questa capacità viene meno, giacché la* `ClassA` *crea direttamente le istanze delle classi* `ClassB` *e* `ClassC`*. Ciò inibisce diversi vantaggi offerti dall'utilizzo delle interfacce: l'inserimento e/o la rimozione di classi che implementano la medesima interfaccia non può avvenire senza modificare il codice della classe* `ClassA`*.*



mente nel contesto nel mondo Object Oriented) di separare il comportamento dalla relativa interfaccia che, in qualche modo, costituisce il confine tra l'astrazione e l'implementazione. Per esempio quando si pensa a un televisore, ciò che viene in mente è la scatola con lo schermo, la visualizzazione delle immagini, l'audio, lo "scettro del potere" che permette di cambiare programma e di certo non il tubo a raggi catodici o gli elettroni, i circuiti stampati, ecc.

Nella pratica quotidiana, più o meno consciamente, si dà luogo a tutta una serie di astrazioni relative a concetti completamente diversi tra loro. Per esempio si possono astrarre delle entità che effettivamente esistono nel dominio del problema o che traspaiono da esso (in un sistema bancario alcuni esempi sono le valute, i prodotti, i *trade*, ecc.), le azioni e/o gli eventi che si verificano (sempre nello stesso sistemi esempi di eventi sono la rivalutazione di fine giornata, la ricezione di un nuovo trade, ecc.), le operazioni, i servizi e così via.

Come illustrato nei paragrafi precedenti, focalizzarsi sulla proiezione esterna di un'entità, nel mondo Object Oriented porta alla definizione dell'interfaccia (implicita e/o esplicita), ossia il *contratto* che gli oggetti utilizzatori stipulano con quelli fornitori di servizi: il repository delle assunzioni effettuate dagli oggetti *client* circa i relativi fornitori. Il *protocollo*, ossia l'elenco delle operazioni visibili dagli oggetti *client* che costituiscono l'interfaccia degli oggetti fornitori. rientra negli argomenti trattati in dettaglio nei paragrafi dedicati all'*Abstract Data Type*.

Leggi fondamentali dell'Object Oriented in breve

Ogni qualvolta si parla di disegno o programmazione Object Oriented, le parole "magiche" che immediatamente vengono alla mente sono: ereditarietà (*inheritance*), incapsulamento (*encapsulation*) e polimorfismo (*polymorphism*), ossia le leggi fondamentali del disegno orientato agli oggetti.

Ereditarietà

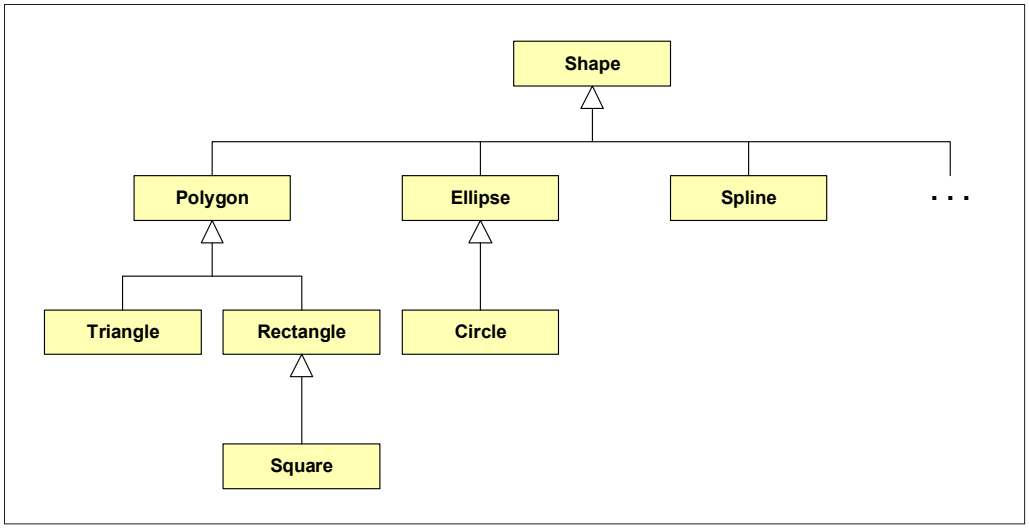
L'ereditarietà è indubbiamente la legge più nota del mondo Object Oriented, che, tanto per non volersi sempre ripetere, non è esente da diversi fraintendimenti. Si tratta di un meccanismo attraverso il quale un'entità più specifica incorpora struttura e comportamento definiti da entità più generali. Il fine cui si dovrebbe tendere attraverso l'utilizzo dell'ereditarietà è il "riutilizzo del tipo" (interfaccia implicita di un oggetto) e non dell'implementazione, sebbene questo sia un effetto ben desiderato. Il problema, come si vedrà, è tentare di riutilizzare unicamente l'implementazione per mezzo dell'ereditarietà.

Gli elementi da cui si eredita sono detti *genitori*, mentre quelli ereditanti sono detti *figli*. Proseguendo nella metafora, un elemento legato a uno di partenza, nella direzione del genitore attraverso più relazioni di ereditarietà (nonno, bisnonno, etc.), è detto antenato (*ancestor*). Mentre un elemento legato a uno di partenza, nella direzione del figlio (nipote, pronipote, ecc.), attraverso diverse relazioni di ereditarietà è detto discendente (*descendant*). Nel caso di diagrammi delle classi, gli elementi genitori sono anche detti superclassi (*superclass*), mentre i figli sono definiti sottoclassi (*subclass*).

L'albero visualizzato nella fig. 6.7 rappresenta una struttura di classificazione che si presta a diverse estensioni, mostrando per esempio le specializzazioni della figura piana triangolo (isoscele, rettangolo, equilatero e scaleno), il pentagono, l'esagono, ecc. La relazione tra elemento genitore e quello figlio in UML è detta generalizzazione, e spesso viene indicata come *is-a* (= "è un"). In effetti un "quadrato *è un* rettangolo", il quale, a sua volta "*è un* poligono", e così via. Per quanto concerne gli antenati e i discendenti, si può notare, per esempio, che la classe `Square` è discendente di quella `Polygon`, e quindi quest'ultima è sua antenata.

Gli elementi figli, ovviamente, sono completamente consistenti con quelli più generali da cui ereditano (ne possiedono tutte le proprietà, i membri e le relazioni) e in più possono specificare struttura e comportamento aggiuntivi. Gli elementi figli duplicano ed estendono l'interfaccia "implicita" dei genitori. L'ereditarietà non implica necessariamente l'inclusione di comportamento aggiuntivo nelle classi discendenti. Eventualmente, queste possono lasciare inalterata la definizione implicita dell'interfaccia del genitore, modificando "semplicemente" il comportamento della classe genitrice (ridefinizione di opportuni metodi). Per esempio la classe astratta `Shape` potrebbe ridefinire metodi come `draw()`, `getArea()`, la cui implementazione varia a seconda della specializzazione della classe (chiaramente il calcolo dell'area della circonferenza differisce da quella del rettangolo). Questo concetto, noto come polimorfismo (illustrato nei successivi paragrafi), è ottenuto grazie all'*overriding* dei metodi.

Figura 6.7 — *Esempio di classificazione. La definizione dei vari concetti è ottenuta combinando il meccanismo della condivisione del comportamento comune con la definizione incrementale dei concetti. La relazione di generalizzazione tra due classi è mostrata in UML attraverso una freccia collegante l'elemento figlio al proprio genitore, con un triangolo vuoto posto in prossimità di quest'ultimo. Su questo argomento si tornerà con maggior dettaglio nel prossimo capitolo. Da notare che sebbene fin dalle scuole superiori venga insegnato che la circonferenza è una "degenerazione" dell'ellisse, spesso si possono incontrare problemi nel rappresentare questa specializzazione. In effetti un'ellisse è definibile in termini di* fuochi *mentre per una circonferenza è sufficiente specificare le coordinate del centro e la misura del raggio. Questa apparente incompatibilità si risolve considerando le equazioni canoniche di queste due curve.*



Nel caso in cui gli elementi ereditanti si limitano a specializzare il comportamento dichiarato nelle classi genitrici, si parla di sostituzione pura e gli elementi ottenuti mostrano la stessa interfaccia delle classi genitrici (caso classico del polimorfismo). Le classi figlie però possono anche inibire specifici comportamenti e/o strutture di quelle genitrici. Anche se ciò non sempre rappresenta esattamente un buon disegno, spesso risulta una tecnica molto comoda per risolvere diversi problemi.

Si consideri la fig. 6.8 nella quale viene mostrata l'organizzazione gerarchica delle interfacce Java che consentono di utilizzare le collezioni di dati indipendentemente dai relativi dettagli implementativi. Come si può notare, al fine di rendere più facilmente gestibili le collezioni, gran parte del comportamento è dichiarato nelle interfacce antenate, anche in modo un po' artificioso. Ciò comporta che qualora si tenti di eseguire un metodo definito da un'interfaccia il cui oggetto implementante non ne prevede la realizzazione, l'oggetto stesso comunichi un'apposita eccezione (`UnsopportedOperation-Exception`).

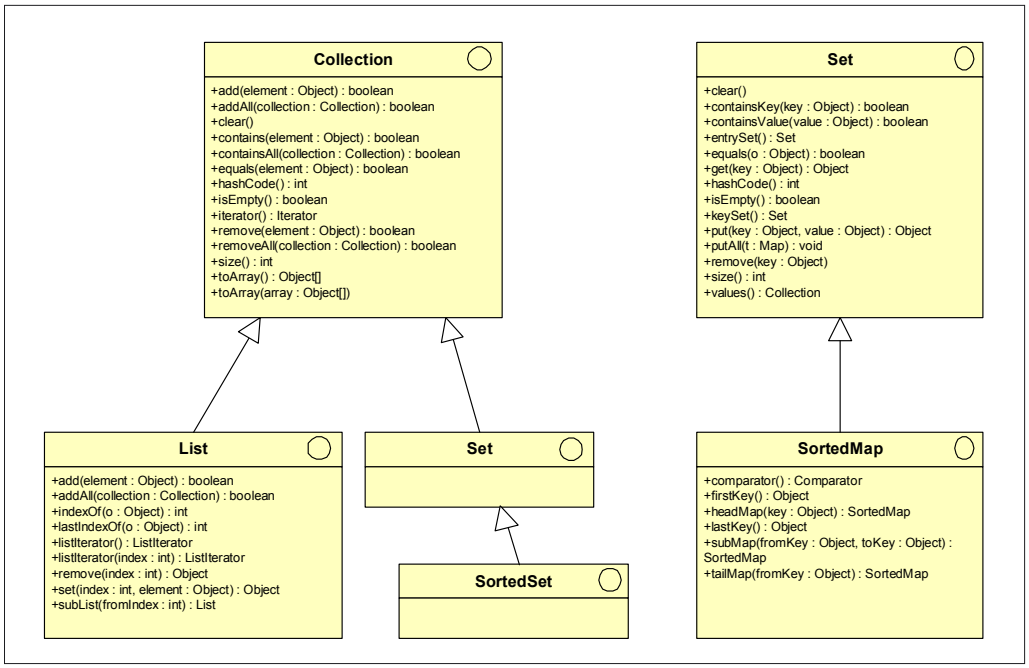
Con un semplice paragone, è possibile pensare all'ereditarietà come a un meccanismo in grado di prendere un elemento di partenza, clonarlo e di modificare e/o aggiungervi struttura e comportamento ulteriori. Un elemento definito per mezzo della relazione di generalizzazione è, a tutti gli effetti, un nuovo tipo che "eredita" dal genitore tutto ciò che è dichiarato come tale (in sostanza tutto tranne attributi e metodi dichiarati privati). Chiaramente vengono inibiti i riferimenti diretti agli elementi dichiarati privati, mentre quelli indiretti restano ancora possibilissimi (si pensi ai metodi `get/set`).

Da quanto detto è chiaro che l'ereditarietà presenta punti di contrasto con il principio dell'incapsulamento: la classe antenata deve esporre propri dettagli interni alla classe ereditante. Ciò, tipicamente, comporta che la comprensione del funzionamento di una classe discendente dipende dalla logica interna di quella antenata e quindi modifiche alle classi antenate tendono a ripercuotersi su quelle discendenti. Un altro effetto negativo è dovuto al fatto che l'ereditarietà (qualora non applicata al dominio del problema) può generare una certa difficoltà nel comprendere il disegno/codice: la cognizione esatta di comportamento e struttura di una classe figlia prevede la conoscenza delle classi antenate.

La capacità dei linguaggi di supportare *direttamente* l'ereditarietà, rappresenta la differenza tra linguaggi Object Oriented e quelli Object Based (come per esempio Java Script). L'ereditarietà, come si vedrà di seguito, può essere *simulata*: questo concetto, però, è diverso dal *supportare*.

L'ereditarietà è la trasposizione informatica dello strumento che permette di modellare i risultati dei processi di classificazione, in uso fin dai tempi antichi (probabilmente Aristotele fu il primo disegnatore Object Oriented di cui si abbia notizia…). Tale processo permette di organizzare gerarchicamente entità effettivamente esistenti nel mondo reale, realizzando una struttura in cui la descrizione di un elemento è fornita in maniera incrementale, attraverso la localizzazione del comportamento comune nelle classi progenitrici (si sposta più in alto possibile nella gerarchia), specializzandolo via via che si procede verso il basso, ossia nelle classi discendenti. Sebbene l'estrazione del comporta-

Figura 6.8 — *Diagrammi rappresentanti le interfacce del core Collection di Java. Brevemente, una collezione rappresenta un gruppo di oggetti, detti elementi. Alcune implementazioni prevedono elementi duplicati, altri no; alcune sono ordinate, altre mantengono l'ordine di inserimento. L'interfaccia* `Collection` *non viene direttamente implementata, bensì rappresenta l'elemento in comune (antenato) delle collezioni più specifiche. Un* `Set` *(insieme), come suggerisce il nome, è una particolare versione di collezione che non ammette elementi con stessa chiave (duplicati). La* `List` *(lista) rappresenta una collezione ordinata in grado di contenere elementi duplicati. L'interfaccia* `Map` *(mappa), permette di rappresentare delle coppie (chiave, valore) e quindi non ammette elementi con la stessa chiave.*



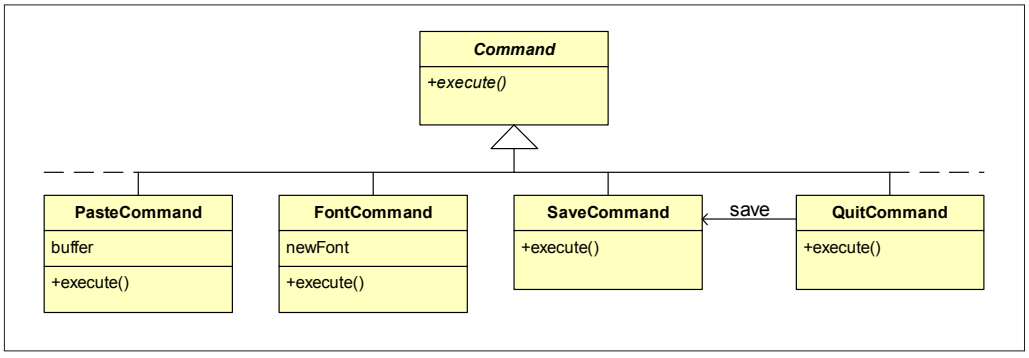
mento comune sia, in genere, una buona pratica, giacché favorisce il riutilizzo del codice da parte di tutte le classi discendenti, é opportuno non abusarne per tutta una serie di motivi, non ultimi la necessità di dovere inibire determinate parti nelle classi discendenti (situazione non sempre consigliabile), l'immutabilità delle relazioni disegnate, ecc.

L'ereditarietà è una tecnica molto potente e i vantaggi apportati sono la semplificazione della modellazione di sistemi reali, la riusabilità del codice, nonché il polimorfismo. La riusabilità del codice è dovuta al fatto che il comportamento comune è definito una sola volta nella classe progenitrice e poi utilizzato in tutte le classi discendenti. Ciò è molto importante anche perché, tipicamente, genera modelli più snelli e tende a neutralizzare

gli effetti generati da alcuni aggiornamenti. Infatti, se questi ultimi sono relativi unicamente alla porzione condivisa, è sufficiente modificare opportunamente la classe progenitrice e, immediatamente, tutte le discendenti ne rifletteranno le variazioni. Pertanto riduce la quantità di lavoro richiesta dalle operazioni di modifica e minimizza la possibilità di creare accidentali inconsistenze dovute a processi di modifica non eseguiti completamente.

Uno dei principi fondamentali impliciti nell'ereditarietà è la *sostituibilità*, definito formalmente da Barbara Liskov. In particolare questo principio afferma che "un'istanza di una classe discendente può sempre essere utilizzata in ogni posto ove è prevista un'istanza di una classe antenata". Ciò è abbastanza intuitivo considerando che le classi figlie ereditano comportamento e struttura di quelle genitrici e in più vi aggiungono comportamento specifico. Considerando l'esempio successivo, è possibile utilizzare, in ogni luogo in cui è previsto un oggetto di tipo `Command` (che tra l'altro non può esistere in quanto `Command` è una classe astratta), un'istanza di una sua specializzazione (`PasteCommand`, `FontCommand`, ecc.). In altre parole, è possibile, per esempio, disporre di un metodo in

Figura 6.9 — *Applicazione parziale del pattern* Command *definito nel libro della Gang Of Four (BIB04). In questo modello si è voluto mostrare un utilizzo più operativo della relazione di eredità. In particolare, invece di rappresentare relazioni tra oggetti esistenti nel dominio del problema, la si è utilizzata per mostrare parte dell'infrastruttura di un sistema. Come si può notare è possibile definire una classe generica* `Command` *dotata di un metodo astratto* `execute()` *e quindi tutta una serie di specializzazioni atte a definire il comportamento del metodo in funzione delle responsabilità dello specifico comando. Per esempio, nella classe* `PasteCommand`*, il metodo* `execute()` *si occupa di copiare quanto presente nel buffer nell'area selezionata; nella classe* `QuitCommand`*, il metodo ha l'incarico di terminare l'esecuzione del programma (eventualmente chiedendo conferma) e, nel caso in cui vi siano cambiamenti non ancora memorizzati, ha l'ulteriore responsabilità di richiedere se salvare o meno i cambiamenti prima di terminare l'esecuzione.*



grado di eseguire diversi comandi che preveda come parametro formale un oggetto di tipo `Command` (`executeCommand(c : Command)`) a cui fornire come argomenti attuali le istanze delle classi specializzanti. Ciò permette di eseguire, con lo stesso metodo, comandi di paste, di quit, e così via. Il vantaggio è che la parte di sistema che utilizza una classe antenata non cambia in funzione delle classi discendente e non ha alcuna visione di quale particolare specializzazione si tratti.

Come accennato in precedenza, l'ereditarietà favorisce il meccanismo del polimorfismo (al livello di operazioni). In particolare, questo meccanismo permette di dichiarare per una stessa operazione (metodo) definita in una classe antenata, diverse implementazioni ognuna localizzata in una delle classi discendenti. A tempo di esecuzione, la particolare versione da invocare è determinata dalla classe di cui sono istanze gli oggetti ai quali sono applicate, piuttosto che dallo stato del chiamante. In altre parole, una stessa classe può disporre di molteplici specializzazioni e ciascuna di queste può definire, in funzione delle relative esigenze, proprie versioni di un'operazione condivisa da tutte le altre classi (i fratelli).

Si consideri per esempio il modello di fig. 6.6. Si sarebbe potuto specificare in tutte le classi un metodo per il calcolo della superficie, per esempio `calculateArea()`, (nella superclasse `Shape` si sarebbe trattato di un metodo astratto) la cui implementazione sarebbe dovuta variare da classe a classe (il calcolo della superficie del quadrato è ben diversa da quella di un'ellisse). Questa tecnica offre la possibilità di trattare oggetti di una classe come se fossero istanze di una classe progenitrice (*upcasting*, ciò avviene automaticamente nei linguaggi di programmazione). In altre parole, l'oggetto non viene trattato come un tipo specifico, bensì come il genitore. Il vantaggio è definire codice indipendente dagli specifici tipi. Per esempio è possibile richiedere a una figura (`Shape`) di disegnarsi nel canvas, senza sapere a quale particolare tipo di figura si faccia riferimento (invocazione polimorfa). Un altro vantaggio è relativo alla possibilità di aggiungere nuove figure (o nuovi comandi) senza dover modificare il codice esistente e così via.

> Come si vedrà nell'apposita sezione del capitolo successivo, in UML, l'ereditarietà è mostrata attraverso la relazione di Generalizzazione. Si tratta di una relazione tassonomica, transitiva e antisimmetrica, tra un elemento generale e uno più specifico, in cui quest'ultimo risulta completamente consistente con il genitore e vi aggiunge comportamento supplementare. Pertanto un'istanza di un elemento figlio può sempre essere utilizzata in ogni posto in cui è previsto l'utilizzo dell'istanza padre.

La classificazione

La classificazione è un processo mentale che permette di organizzare la conoscenza. Nel mondo Object Oriented ciò si traduce raggruppando in un'opportuna organizzazione gerarchica le caratteristiche comuni di specifici oggetti. Questo permette di dar luogo

a modelli più leggeri e quindi più semplici da comprendere, sebbene, qualora utilizzata in maniera impropria, possa generare non pochi inconvenienti, come una visione distorta della realtà.

Il problema della classificazione "intelligente" è ovviamente presente in tutte le scienze e riuscire a individuare un'organizzazione opportuna ai propri scopi è un'attività cruciale, poiché favorisce il processo di comprensione tipicamente attuato dalla mente umana. In generale, la costruzione di modelli significativi di oggetti ed eventi osservati è spesso propedeutica allo sviluppo di nuove teorie scientifiche. Un esempio? La legge dell'ereditarietà di Mendel.

Più specificatamente, nel mondo dell'Object Oriented la classificazione è estremamente importante in ogni aspetto del disegno: permette di identificare generalizzazioni, specializzazioni, strutture gerarchiche e così via.

Chiaramente non esiste una regola o un solo metodo per classificare: tutto dipende dagli obiettivi che si intendono perseguire e ciò è ovviamente valido in ogni disciplina, da quelle a carattere meno formale a quelle più marcatamente scientifiche. Esempi celebri possono essere trovati nella classificazione del DNA, degli elementi chimici, delle specie viventi, ecc.

Il problema è analogo a scalare una montagna: una volta giunti sulla vetta tutto appare chiaro. Con ciò si intende dire che l'incognita è dare vita a una classificazione appropriata: una volta ottenuto tale risultato tutto diventa facilmente comprensibile. In generale, i disegni migliori sono quelli semplici — forse sarebbe opportuno dire meno complessi — ma la semplicità richiede tanta esperienza e soprattutto molto tempo di lavoro.

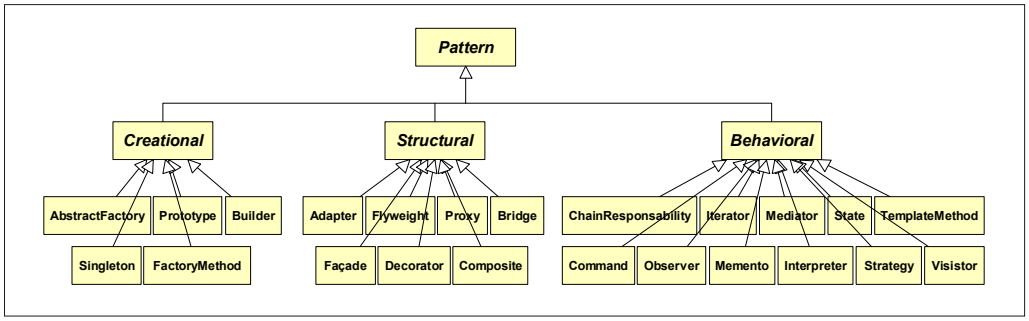
Il processo di classificazione, si presta ad essere applicato in maniera iterativa e incrementale. Si definisce una prima soluzione (in termini di una struttura di classi) che tipicamente è realizzata per risolvere uno specifico problema (in altre parole è realizzata *ad hoc*). Poi, si comincia a studiare la soluzione e ci si rende conto che essa, opportunamente astratta, si presta a risolvere diversi altri problemi e quindi si avanza con un processo atto ad aumentare il livello di generazione. L'attività di *disegnare* è molto opportuna anche per questo: il costo di tale refactoring al livello di disegno è molto contenuto (in fondo si tratta di spostare dei rettangolini nello schermo, di cambiarne eventualmente il nome e di muovere attributi e operazioni).

Nel processo di aumento del livello di astrazione bisogna fare attenzione a non giungere fino all'estremo opposto: il disegno delle soluzioni è così generale da risultare caotico.

I tranelli dell'ereditarietà

Come descritto in precedenza l'ereditarietà è probabilmente la legge più nota dell'Object Oriented ma, verosimilmente, anche quella di cui si fa maggiore abuso. Se da una parte è vero che l'identificazione del comportamento condiviso da più classi permette di accentrarne una versione generalizzata in un'apposita classe antenata, dando luogo a una migliore ristrutturazione gerarchica (le altre classi estendono e specializzano il comporta-

Figura 6.10 — *Esempio di classificazione relativa ai pattern presentati nel famosissimo libro della "combriccola dei quattro" (Gang of Four) [BIB04].*



mento in comune), dall'altro bisogna tenere in mente che l'eredità non è esente da controindicazioni. L'importante è considerare che l'obiettivo da perseguire è l'ereditarietà dell'interfaccia delle classi. I vantaggi derivanti da un'*intelligente* classificazione, in quest'ambito sono relativi all'incentivazione del riutilizzo del codice, alla razionalizzazione del modello — che tende a divenire più "leggero" — e così via. Come tutti gli strumenti però, possiede il proprio dominio di applicazione che, se non rispettato, non solo non aiuta a risolvere lo specifico problema, ma può addirittura generare tutta una serie di gravi anomalie nel sistema. Il problema, come al solito, non è tanto legato all'ereditarietà, quanto all'utilizzo forzato (la solita aspirina utilizzata per guarire un'ulcera) che spesso ne viene fatto. Nella pratica succede che disegnatori junior, non appena "fiutino" l'eventualità di un minimo comportamento condiviso — magari un paio di attributi e/o metodi — si affrettino a dar luogo a generalizzazioni le quali, ahimè, spesso risultano abbastanza stravaganti.

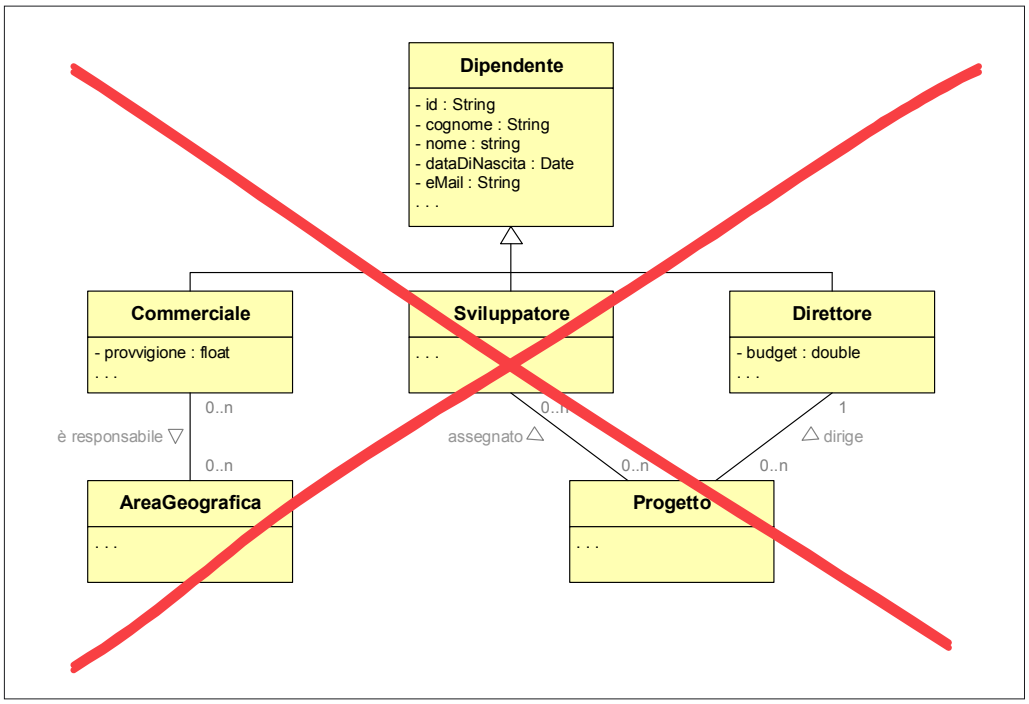
La "debolezza" dell'ereditarietà è intrinseca nella propria definizione: la staticità e la rigidità della struttura gerarchica. Una volta che una classe viene incastonata in questo tipo di organizzazione non ne può più uscire. Quindi, se un determinato oggetto nasce di un tipo, non può evolvere durante la propria vita: è inevitabilmente condannato a morire nello stesso tipo (in altre parole non può subire delle metamorfosi). Ciò è ovviamente vero per tutti gli oggetti. Qualora però un'istanza di una determinata classe abbia necessità, in qualche modo, di mutare "tipo" durante il proprio ciclo di vita, ecco che è necessario rappresentare questo comportamento per mezzo di opportune versioni della relazione di associazione (composizione) e non con legami di generalizzazione. In sostanza, l'informazione relativa al "tipo variante" va realizzata non attraverso una rigida relazione di ereditarietà, bensì tramite relazioni di composizione con altri oggetti, che rappresentano appunto il tipo. Trattandosi di relazioni di composizione è sempre possibile "staccare" l'associazione con un'istanza di una determinata classe e "attaccarla" (realizzarne una nuova) all'istanza di un'altra classe. Se queste classi destinazione dell'associazione model-

lano in qualche modo l'evoluzione dell'oggetto, ecco fornita una modalità per "trasmutare" gli oggetti in altri tipi… o meglio per simularne la trasmutazione.

Per chiarire quanto espresso, si consideri il seguente esempio relativo alla classificazione dei ruoli in un'organizzazione. Si tratta di una situazione molto frequente: in tutte le organizzazioni, gli attori umani del sistema (clienti, dipendenti, ecc.) necessitano di essere modellati in una struttura gerarchica di ruoli. Il problema opportunamente generalizzato, potrebbe essere ricondotto a un caso di ereditarietà: è possibile identificare una serie di oggetti che esibiscono segmenti comuni di comportamento e/o struttura (per esempio la classe `Persona`) al quale ognuno aggiunge ulteriori specializzazioni (`Cliente`, `Manager`, `Contabile`, ecc.). Sebbene la situazione, in prima analisi, possa essere considerata il più classico esempio di ereditarietà, le cose invece non stanno esattamente così. Si consideri il diagramma di fig. 6.11.

Cosa accade se un `Commerciale`, a un certo punto del suo ciclo di vita decide di diventare uno `Sviluppatore`? L'autore immagina che tale interrogazione possa aver

Figura 6.11 — *Esempio di errato utilizzo della relazione di ereditarietà. Da notare che con il termine di* sviluppatore *si intende far riferimento ai diversi ruoli implicati nella costruzione di sistemi (*architetti*,* programmatori*,* tester*, ecc.).*



generato qualche sorrisino malizioso sulla bocca di diversi lettori, ma in questo caso la domanda verte unicamente sulla qualità del modello (che triste lavorare in un mondo in cui la gente non crede più nei miracoli…). Ebbene, sarebbe necessario dar luogo a un'altra istanza, questa volta di tipo `Sviluppatore`, e quindi avere due oggetti diversi relativi allo stesso individuo con due identificatori distinti, con tutti i problemi derivanti. Ancora, cosa succederebbe se alcuni sviluppatori (per esempio Antonio Rotondi, Roberto Virgili), fossero così eccelsi da svolgere anche funzioni di `Direttore`? Queste due semplici domande sono sufficienti a dimostrare tutti i limiti dell'utilizzo della relazione di generalizzazione in contesti come questo. Chiaramente con ciò non si intende "offuscare il prestigio" della relazione di ereditarietà ma si vuole solo fornire un esempio del suo cattivo impiego.

Da queste brevi constatazioni è possibile enunciare un paio regole semplici ma efficaci:

- qualora in una struttura gerarchica un oggetto possa "trasmutare", evidentemente l'applicazione della relazione di estensione è inappropriata e quindi è opportuno ricorre alla composizione;

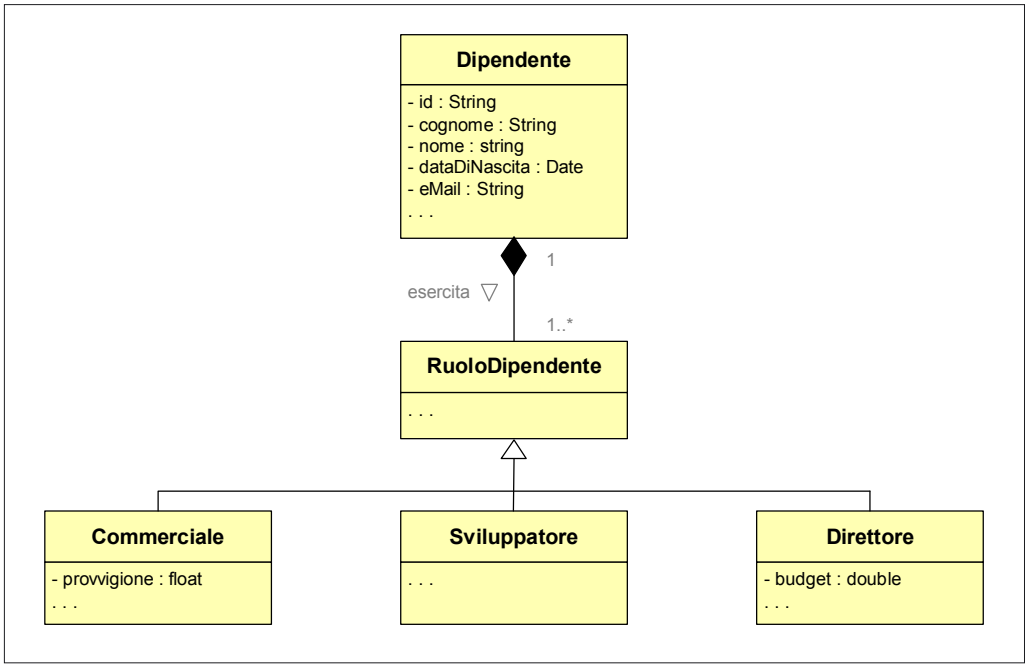
- ogniqualvolta in una struttura gerarchica un oggetto possa appartenere a più "tipi", nuovamente non è opportuno utilizzare la relazione di generalizzazione.

Per quanto possa sembrare strano, anche la relazione di composizione — si tratta di una versione della relazione di associazione, ma con una semantica più forte — si presta ad essere utilizzata, con le opportune cautele, per estendere le responsabilità degli oggetti delegando parte del lavoro ad altri. In parole semplici, la composizione è in grado di *simulare* la relazione di ereditarietà. Questa affermazione, tipicamente, genera le perplessità di tecnici ancora non molto esperti. In virtù di quanto detto, il diagramma si presta a essere rappresentato come riportato in fig. 6.12.

Questo modello è riconducile al pattern denominato *Actor-Participant* elaborato da Coad. Ciò che potrebbe lasciare perplessi è la molteplicità `1` della relazione `esercita` dal lato `Dipendente`. Essa sancisce che un oggetto `Dipendente` può possedere diversi ruoli, e che una specifica istanza della classe `RuoloDipendente` è relativa a un solo oggetto `Dipendente`. L'utilizzo della composizione — come si vedrà nel prossimo capitolo — sancisce una forte connotazione tutto-parte (*whole-part*) tra le classi associate ed enfatizza il controllo e il possesso da parte della classe tutto (`Dipendente`) nei confronti delle proprie parti (specializzazioni delle classe `RuoloDipendente`). Ciò implica, tra l'altro, che la creazione e distruzione degli oggetti parte debba avvenire sotto il controllo di quello tutto.

Il modello di fig. 6.12 è decisamente più flessibile ed è in grado di rappresentare tutte le situazioni in cui un dipendente cambi ruolo nell'arco della collaborazione con un'azienda, oppure abbia responsabilità di diversi ruoli, ecc. Tipicamente, in quasi la totalità dei mo-

Figura 6.12 — *Rappresentazione dei ruoli attraverso la relazione di composizione. Da notare che la classe* `RuoloDipendente` *non è strettamente necessaria.*



delli di analisi del dominio è possibile sostituire relazioni di generalizzazione con opportune composizioni, mentre, per quanto concerne il modello di disegno, con particolare riferimento alle classi di "infrastruttura" bisogna essere più cauti. Se da una parte è vero che la composizione permette di simulare l'ereditarietà, dall'altra bisogna tenere presente che non tutte le caratteristiche di quest'ultima possono essere riprodotte. Quando si eredita da una classe, ciò che viene ereditato non è solamente l'implementazione (simulabile con la composizione) ma l'interfaccia (in effetti questo dovrebbe essere l'obiettivo primario dell'ereditarietà), cosa che invece non avviene con la composizione. Chiaramente anche per scavalcare questo problema è possibile individuare diversi espedienti: si tratta comunque pur sempre di artefici.

Si consideri ora la limitazione Java legata all'impossibilità di realizzare ereditarietà multiple. Qualora se ne abbia la necessità il problema può essere risolto come riportato nella fig. 6.13.

La soluzione prevede di selezionare una delle due classi (per esempio `ClassB`) e quindi realizzare un'apposita interfaccia (`InterfaceB`). Ciò è importante per ottenere l'obiettivo principale dell'ereditarietà: il riutilizzo del tipo. Ove non fosse strettamente necessa-

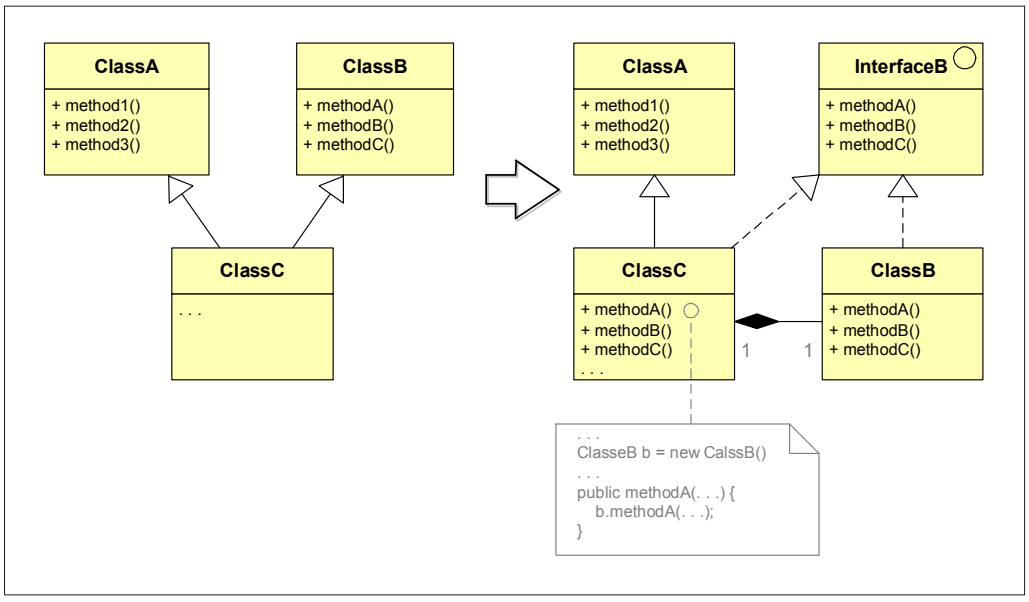
rio trattare le istanze di `ClassC` come se fossero istanze della classe `ClassB`, si potrebbe evitare di dar luogo all'`InterfaceB`. Nella soluzione canonica, `ClassC` eredita da `ClassA` e "implementa" `InterfaceB`. `ClassC` quindi deve implementare i metodi definiti nell'interfaccia: anche qualora non implementasse `InterfaceB`, dovrebbe comunque implementarli, altrimenti non si spiegherebbe la necessità di ereditare da `ClassB`. Ridefinire l'implementazione di tutti i metodi spesso non costituisce una buona idea. La soluzione consiste nel delegare, all'interno di `ClassC`, il comportamento dei metodi dichiarati nell'`InterfaceB` alla classe `ClassB` che a sua volta implementa `InterfaceB`.

Da notare che anche se si utilizza la relazione di composizione, è possibile che la classe `ClassB` reciti il ruolo di classe composta in altre classi. Come si vedrà nel capitolo successivo, la relazione di composizione impone che ogni *istanza* dell'elemento composto (`ClassB`) sia associata, in ogni istante di tempo, a un solo oggetto "tutto" (`ClassC`, e altri eventuali).

Altri problemi intrinseci nell'ereditarietà sono legati al forte legame di dipendenza che si instaura tra classe genitore e quella figlio. Logica conseguenza di ciò è una violazione dei principi dell'incapsulamento. Ciò è dovuto essenzialmente a due fattori:

- la sottoclasse deve conoscere diversi dettagli della superclasse;

Figura 6.13 — *Nel diagramma è illustrata la tecnica canonica che permette di simulare l'ereditarietà multipla: implementazione di interfacce e composizione.*



- ancora la classe ereditante non è protetta dai cambiamenti nella classe genitore. Pertanto, ogni variazione apportata in una superclasse, potenzialmente è motivo di revisione e aggiornamento di tutte le sottoclassi (questo fenomeno è noto con il nome di *changes ripple*, ossia propagazione delle variazioni).

Questo rischio si presta a essere minimizzato attraverso una corretta applicazione della relazione di ereditarietà. In particolare è necessario, nei limiti del possibile, strutturare le classificazioni con un basso grado di coesione. Qualora una particolare estensione di una classe genitore necessiti di inibire un comportamento definito in quest'ultima, è opportuno interrogarsi se ciò è veramente quello che si vuole o se magari sia necessario introdurre altre classi intermedie.

Soprattutto nei primi modelli, realizzati durante la fase dell'analisi dei requisiti, non è sempre consigliabile organizzare le classi gerarchicamente solo perché condividono del comportamento (per le ottimizzazioni c'é sempre tempo…). Ciò rende i diagrammi più difficilmente comprensibili perché si mostrano delle regole che non appartengono al dominio che si sta cercando di modellare, bensì si tratta di artefici introdotti solo per questioni tecniche.

Genitore unico

Con questo termine ci si riferisce all'ennesimo argomento molto dibattuto nella comunità Object Oriented soprattutto in passato, relativo alla necessità o meno che tutte le classi debbano discendere da uno stesso antenato. Ci sono linguaggi come il C++ in cui non esiste questa organizzazione gerarchica, e altri come Java, in cui tutti gli oggetti discendono dalla classe denominata `Object`. L'ereditarietà in questo caso è implicita per via del fatto che in Java non è possibile definire ereditarietà multiple (più genitori), quindi se tutti gli oggetti ereditassero esplicitamente da quello base `Object` non sarebbero in grado di ereditare da nessun altro oggetto, il che, di fatto, renderebbe impossibile l'utilizzo dell'ereditarietà. Tutte le classi definibili in Java quindi condividono un insieme seppur minimo di comportamento, ossia definiscono un piccolo segmento di interfaccia comune che permette, in qualche modo, di uniformali tutti allo stesso tipo fondamentale.

Il disporre di questa gerarchia permette di richiedere l'esecuzione di tutto un insieme di metodi a ogni classe, come per esempio `toString()`, permette di realizzare diverse classi di utility, come per esempio il `Vector`, l'`Hashtable`, ecc. nelle cui istanze è possibile memorizzare oggetti eterogenei trattandoli come istanze dell'antenato `Object`. In questi casi, l'inserimento di un oggetto richiede l'*upcasting* (implicito) alla classe `Object`, mentre il reperimento necessita il *downcasting* all'oggetto specifico. Quest'ultimo punto non è esattamente un vantaggio. In effetti, l'utilizzo di queste classi obbliga a eseguire un *casting* continuo non controllabile a tempo di compilazione — fonte di errori non facilmente identificabili — e quindi rende il linguaggio meno *type-checked*.

La tecnica della gerarchia con singola radice, inoltre, permette di semplificare la realizzazione di molti meccanismi, come per esempio il *garbage collector*; infatti è possibile inviare a ogni oggetto precisi messaggi sapendo che questo dispone dei metodi appropriati per trattarli. Anche il debugging è relativamente agevolato, è sempre possibile determinare l'identità di un oggetto, e così via. Ciò però non significa assolutamente che la presenza del genitore unico sia un requisito imprescindibile dei linguaggi Object Oriented.

Ereditarietà multipla

Fino a questo punto si è considerata la situazione in cui una classe erediti da un solo genitore (una sorta di autofecondazione); nel caso più generale — non previsto da tutti i linguaggi — è possibile che una stessa classe erediti da più genitori. In questi casi si parla di ereditarietà multipla.

La comunità Object Oriented è stata spesso divisa circa l'utilità di ricorrere a tale meccanismo. Ciò è dovuto, principalmente al fatto che disegni basati sull'eredità multipla, se non eseguiti accuratamente, possono essere fonte di tutta una serie di anomalie. Un'affermazione molto celebre di Booch è che "l'ereditarietà multipla è come un paracadute: non sempre se ne ha bisogno, ma quando viene la necessità, si è molto felici di averne uno a portata di mano".

L'autore del presente testo, limitatamente a questo argomento è solito utilizzare un approccio decisamente pragmatico. Quando si disegnano diagrammi delle classi precedenti a quello di disegno (dominio, business e analisi) — e quindi a forte connotazione descrittiva dell'area business e a minore impatto sull'implementazione — se la realtà si presta intrinsecamente a essere modellata attraverso l'eredità multipla, é appropriato utilizzarla. Si ricordi che l'obiettivo di questi modelli e descrivere nel modo più semplice e accurato possibile l'area business oggetto di studio. Quando poi si giunge al modello di disegno, che quindi dovrebbe essere quasi in corrispondenza biunivoca con l'implementazione (sebbene alla fine "solo il codice sia sincronizzato con sé stesso" [S. Ambler]) è possibile scomporre l'ereditarietà multipla secondo le tecniche descritte in precedenza.

Per l'autore questa tecnica è diventata obbligatoria nel corso degli ultimi anni, dal momento che i sistemi sviluppati sono esclusivamente basati sul linguaggio Java, che, come noto, non supporta l'ereditarietà multipla.

L'esempio classico citato da molti testi è legato alla classificazione dei veicoli. Infatti, in prima analisi li si potrebbe suddividere in terresti, nautici e aerei, salvo poi avere i mezzi anfibi che, chiaramente, hanno sia capacità tipiche dei mezzi terrestri sia di quelli nautici. Un altro esempio è relativo alla classe associazione (*association class*) che, come si vedrà nel capitolo successivo, nel metamodello UML è rappresentata attraverso un particolare elemento che ha caratteristiche sia di classe, sia di relazione e quindi eredita da entrambi gli elementi del metamodello. Un ulteriore esempio, di carattere più operativo, è mostrato nel paragrafo successivo.

Uno dei problemi legati all'ereditarietà multipla è che se una caratteristica è dichiarata esattamente nello stesso modo (per esempio, metodo con stessa firma) in due diverse classi, dalle quali erediti una terza che non specializza tale caratteristica, si crea un conflitto nel modello. In caso di invocazione/accesso alla caratteristica dichiarata indipendentemente dai due genitori, a quale far riferimento? *(Alcuni linguaggi, come il C++, permettono di risolvere questi tipi di conflitti attraverso la dichiarazione esplicita del metodo da utilizzare)*. Chiaramente lo UML non fornisce meccanismi per risolvere conflitti di questo tipo: il tutto è affidato al disegnatore.

Il problema del "diamante" nell'ereditarietà multipla

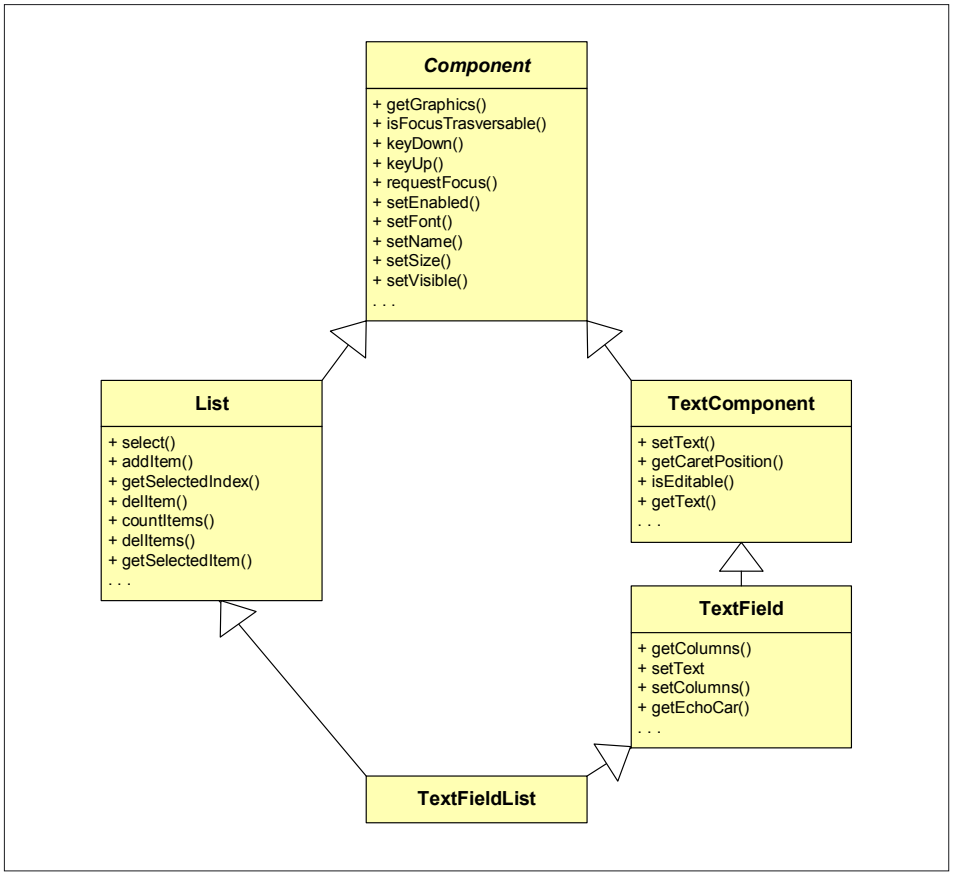
Scopo del presente paragrafo è fornire qualche indicazione relativa al famoso problema del diamante… Sebbene ciò possa portare a pensare a problemi connessi con l'eredità di un singolo bene materiale da parte di diversi eredi, in effetti le cose non sono esattamente così. Non si sta parlando neanche di regalo alla propria partner, sebbene anche… il diamante Java sia per sempre.

Nella comunità Object Oriented, con tale termine si fa riferimento a un noto problema relativo all'ambiguità insita nei modelli in cui una classe erediti da due altre (ereditarietà multipla), entrambe discendenti dal medesimo genitore. Si consideri l'esempio di voler realizzare un nuovo componente Java AWT che presenti caratteristiche di una normale lista (`List`) e che contemporaneamente permetta di inserire dinamicamente voci (item) aggiuntive attraverso i meccanismi di una normale `TextField`. Una buona idea, considerando per assurdo che il linguaggio Java lo consenta, potrebbe essere quella di ricorrere all'ereditarietà multipla: realizzare il nuovo componente `TextFieldList`, ereditando dai due precedenti (si consideri il diagramma della fig. 6.14). Come si può notare, questo disegno genera la congiuntura del diamante: sia `List`, sia `TextField` ereditano dalla classe astratta `Component`, e `TextFieldList` eredita da entrambe. Chiaramente una soluzione alternativa potrebbe essere realizzata utilizzando una delle tecniche descritte in precedenza.

Ora, disponendo di un'istanza del nuovo componente (`TextFieldList textFieldList : new TextFieldList()`), l'invocazione dei metodi ereditati da `Component` genererebbe situazioni di ambiguità (`textFieldList.keyUp()`). Chiaramente il problema nascerebbe anche qualora, nella nuova classe, si presentasse la necessità di riferirsi a un metodo della superclasse (il *super* non sarebbe molto di aiuto: quale genitore?) o si volesse accedere a un attributo dichiarato nella classe `Component`.

Come risolve il problema Java? Semplice: non realizzando alcun meccanismo diretto di ereditarietà multipla tra classi e demandando la soluzione all'uso di interfacce e/o composizioni. In C++ il problema viene tipicamente risolto attraverso il meccanismo delle *virtual base classes* [B. Stroustrup, *The C++ Programming Language*, 3rd edition – Addison Wesley – §15.2.4, *Virtual base classes*] e con lo *scooping* esplicito.

Figura 6.14 — *Esempio del problema del diamante. Il nome del pattern è dovuto alla forma che ricorda un rombo, spesso chiamato* diamond *(diamante) in inglese.*



Incapsulamento

L'incapsulamento è il meccanismo che rende possibile il famoso principio dell'*information hiding* (nascondere le informazioni). Con tale termine ci si riferisce alla capacità degli oggetti di celare al mondo esterno, la propria organizzazione in termini di struttura e logica interna: non si tratta pertanto di nascondere le informazioni ai colleghi, come molti sarebbero portati a credere… Il principio fondamentale è che nessuna parte di un sistema deve dipendere dai dettagli interni di una sua parte.

Non di rado accade di accorgersi, nella fase di test, che determinati oggetti causano un eccessivo consumo di memoria oppure eseguono specifiche operazioni con pessime per-

formance. In questi casi, se si è utilizzato con intelligenza il principio dell'incapsulamento, è possibile reingegnerizzare le classi da cui derivano tali oggetti, senza dover modificare altre parti del sistema. Pertanto è possibile sostituire sezioni di implementazione, o addirittura intere classi o componenti, senza dover apportare cambiamenti all'esterno. Sebbene uno dei vantaggi sbandierati dal paradigma Object Oriented prima, e dal Component Based poi, sia il riutilizzo del codice (che in realtà per mille motivi, non ultimi i continui cambiamenti dei requisiti utenti, avviene molto raramente), il vero punto di forza probabilmente consiste nella possibilità di sostituire — in maniera *quasi* indolore — porzioni di codice. (consultare Martin Fowler e il suo incommensurabile *Refactoring*, Addison Wesley).

Nelle fasi iniziali del disegno di modelli a oggetti, tipicamente, si è interessati a "scoprire" — e spesso a inventare — oggetti definendone formalmente il comportamento esterno: in altre parole è necessario eseguire un processo di astrazione. Poi, nella realizzazione del comportamento definito, è fondamentale seguire le norme dettate dal principio dell'incapsulamento che è maggiormente focalizzato nella fase di realizzazione delle astrazioni. In sintesi questi due concetti permettono di separare nettamente le due componenti fondamentali di ogni oggetto: l'interfaccia (definita attraverso l'attività dell'astrazione) e la relativa implementazione (sia delle astrazioni esposte al mondo esterno sia dei meccanismi necessari per realizzare tale comportamento).

Una delle motivazioni alla base dell'*information hiding* è data dalla necessità di creare uno strato di separazione tra gli oggetti clienti e quelli fornitori. In altre parole è necessario separare l'interfaccia propria di un oggetto dalla sua implementazione interna. In sostanza l'interfaccia (anche se implicita) rappresenta il contratto stipulato tra gli oggetti client e quelli server. Ciò è vantaggioso al fine di aumentare il riutilizzo del codice e di limitare gli effetti generati dalla variazione della struttura di un oggetto (questo argomento è trattato nella sezione successiva dedicata al *Design by Contract*).

In genere l'incapsulamento *standard* prevede che le classi non abbiano alcuna conoscenza della struttura interna delle altre, e in particolare di quelle di cui possiedono un riferimento, con la sola eccezione della firma dei metodi esposti nella relativa interfaccia. Ciò permette a ogni classe di modificare, aggiungere, rimuovere parte del proprio comportamento e della propria struttura interna senza generare alcun effetto sulle restanti classi. Questo è vero fintantoché le variazioni non abbiano come dominio metodi appartenenti all'interfaccia della classe: è sufficiente anche la variazione di un solo parametro della firma di un metodo dell'interfaccia per rendere necessaria la modifica delle classi client.

Da quanto riportato appare evidente che i princìpi dell'incapsulamento e dell'ereditarietà, per molti versi, presentano diversi punti di discordanza. Il nascondere il più possibile l'organizzazione della struttura delle classi, di fatto, limita o addirittura inibisce l'ereditarietà. Metodi e attributi privati non sono ereditati automaticamente, o meglio, sono ancora ereditati ma non accessibili, e quindi ridefinibili, dalla classe ereditante. In estrema sintesi si può asserire che la *privacy non aiuta l'ereditarietà* (al contrario di quanto avviene nella vita…).

Tipicamente, il principio dell'incapsulamento standard, in Object Oriented, si realizza rendendo privata la struttura interna della classe. Chiaramente una classe con tutti i metodi e gli attributi privati sarebbe di ben poco utilizzo (eternamente condannata alla ricerca della comunicabilità). Ciò che si desidera è, in definitiva, conferire una visibilità privata a quanta più parte di struttura e comportamento possibile (soprattutto agli attributi), limitandosi a esporre specifici metodi.

L'incapsulamento *totale* si ha quando ogni classe non dispone assolutamente di alcuna conoscenza delle altre, non solo per ciò che concerne il comportamento e la struttura interna, ma neanche in termini di esistenza.

Dipendenza dal tipo

Un modo appropriato di descrivere un oggetto consiste nel cercare di immaginarlo come una sorta di "scatola nera" dotata di un insieme di capacità ben definite. Ciò equivale ad affermare che è necessario focalizzare l'attenzione sui servizi che un oggetto è in grado di fornire e non sul come questi siano effettivamente realizzati (l'interfaccia). Il vantaggio è che se l'implementazione è "nascosta" è possibile variarla senza che ciò influenzi le classi clienti. Tipicamente, per fornire i servizi esposti, un oggetto necessita di memorizzare delle informazioni (attributi), spesso in modo permanente: tutti gli oggetti hanno la propria memoria, in genere stanziata per altri oggetti. Secondo un approccio Object Oriented purista, il modo con cui queste informazioni sono rappresentate all'interno dell'oggetto dovrebbe essere del tutto irrilevante. Contrariamente a molte convinzioni comuni, ciò non significa semplicemente che tutti gli attributi dell'oggetto debbano avere una visibilità privata ed essere esposti per mezzo di opportuni metodi `get` e `set`.

Sebbene questa sia la tecnica generalmente utilizzata — molto spesso inevitabile —, in ultima analisi si tratta di un modo prolisso di esporre direttamente gli attributi. Avere metodi di accesso agli attributi membro è molto utile, specie se il relativo aggiornamento può generare delle conseguenze (per esempio una variazione del colore di un oggetto grafico ne richiede il ridisegno), se l'insieme valori impostabili è condizionato da altri elementi, e così via. Nonostante ciò, anche con questi metodi non si fa altro che fornire l'accesso ai dati membro dell'oggetto. La filosofia Object Oriented specifica che le capacità di un oggetto siano esercitate attraverso scambio di "messaggi": un oggetto cliente richiede l'esecuzione di un servizio esposto da un altro oggetto inviandogli apposito messaggio. I metodi `get`, molte volte, non fanno altro che continuare a esporre i relativi attributi membro attraverso il valore restituito. Per esempio, dichiarare privato un determinato attributo `x` di tipo `y`, e poi realizzarne un metodo `y : getX()` che ne restituisce il valore secondo il tipo di dato (`y`) non è esattamente un esempio di incapsulamento totale. Infatti, una modifica del tipo di dato dell'attributo, richiede l'aggiornamento dell'implementazione delle classi clienti che utilizzano il metodo `getX()`. Quindi, se per qualche motivo varia la rappresentazione interna di un attributo membro di un oggetto, è necessario individuare tutti gli oggetti clienti ed eventualmente variarne il codice. Proba-

bilmente ciò non è completamente coerente con una delle promesse dell'Object Oriented che prevede "la possibilità di variare anche completamente l'implementazione di un oggetto senza variare il codice degli oggetti cliente".

Sempre secondo un approccio purista, invece di richiedere a un oggetto la fornitura di uno specifico dato, bisognerebbe chiedere all'oggetto stesso di eseguire sul dato in questione la funzione di cui si ha bisogno. Poi, quanto questo sia sempre fattibile è un altro argomento. Per esempio, in un'organizzazione, dopo aver definito un oggetto `Dipendente`, invece di accedere al nominativo dello stesso bisognerebbe definire metodi del tipo "stampa sulla busta paga il nominativo", "visualizza a video il nominativo", ecc. Considerazioni circa fattibilità e vantaggi offerti da tale approccio vengono demandati ai lettori…

Polimorfismo

Polimorfismo deriva dalle parole greche *polys* (= molto) e *morphé* (=forma): significa quindi "molte forme". Si tratta di una caratteristica fondamentale dell'Object Oriented, relativa alla capacità di supportare operazioni con la medesima firma e comportamenti diversi, situate in classi diverse ma derivanti da una stessa antenata. Chiaramente il corretto utilizzo del principio dell'ereditarietà — e anche l'implementazione di interfacce nel caso Java — è propedeutico al polimorfismo. Esempio tipico è il calcolo dell'area di una figura. Ciascuna classe espone la stessa firma del metodo (per esempio `getArea() : real`), ma l'algoritmo di calcolo varia in ognuna di esse in funzione della figura che rappresentano (per esempio nella classe `Rettangolo` si avrà `area = latoMinore * latoMaggiore`, in quella `Triangolo` `area = (base * altezza) / 2`, nella `Cerchio` `area = raggio * piGreco`^{2}, e così via).

In sostanza, il polimorfismo fornisce una diversa dimensione per separare l'interfaccia di una classe dalla relativa implementazione, ottenuta in termini di tipi. Si tratta di un meccanismo brillante che, opportunamente applicato, permette di migliorare il disegno del sistema nonché la flessibilità e l'estensibilità: è possibile accrescere la base conoscitiva del sistema e/o aumentarne l'organizzazione senza modifiche al codice. Anche se nella realtà spesso è necessario apportare qualche modifica, l'utilizzo intelligente del polimorfismo permette di minimizzarle. Nel caso delle figure, per esempio, si potrebbero aggiungere nuove classi, come il pentagono, l'esagono, ecc. e la logica di funzionamento sarebbe ancora in grado di comandarle, magari richiedendone il calcolo dell'area, senza averne una conoscenza diretta. Per poter espandere il sistema senza modificare il codice esistente, non è sufficiente disporre dei meccanismi del polimorfismo e dell'ereditarietà, ma sono necessarie tecniche che permettano di generare, in modo astratto, istanze di oggetti polimorfi. In parole povere, è necessario prevedere opportuni meccanismi di generazione di oggetti (pattern creazionali [BIB04]). Nel diagramma relativo alle figure piane, per esempio, è necessario poter generare istanze della figura desiderata (esagono, pentagono

ecc.) nel modo più astratto possibile, altrimenti si finirebbe unicamente per spostare le dipendenze e ciò finirebbe per limitare il plug-in indolore di nuove funzionalità.

La possibilità di attuare il polimorfismo richiede propedeuticamente la facoltà conoscere a priori una porzione dell'interfaccia di un gruppo di classi, o se si preferisce, di poter raggruppare insiemi di classi attraverso segmenti di interfaccia condivisa. Ciò, naturalmente, si ottiene attraverso l'utilizzo dell'ereditarietà: tutte le classi discendenti ereditano l'interfaccia di quella antenata e quindi è possibile trattare i relativi oggetti come se fossero istanze di uno stesso tipo (la classe antenata). L'utilizzo del meccanismo delle interfacce (nel senso di costrutto `public interface`) rende anch'esso possibile trattare in maniera astratta un opportuno insieme di classi (quelle che implementano l'interfaccia): in effetti l'implementazione può essere considerata una versione di ereditarietà (la classe che implementa l'interfaccia ne eredita il tipo definito).

Per poter realizzare il polimorfismo, i linguaggi di programmazione devono realizzare meccanismi di collegamento dinamico (*dynamic binding*, detto anche *late binding*, ossia collegamento ritardato). Con ciò si fa riferimento alla capacità di associare l'invocazione di un metodo alla relativa implementazione presente in un oggetto in tempo di esecuzione. Il motivo è abbastanza evidente: l'istanza della classe che deve eseguire il metodo è conosciuta solo durante l'esecuzione del programma in quanto, in diversi periodi dell'esecuzione, la stessa invocazione potrebbe essere soddisfatta da oggetti appartenenti ad istanze di classi diverse (che ereditano da una stessa classe o che implementano una comune interfaccia). Si consideri l'esempio del pattern Command: un metodo di una generica classe atto a eseguire uno specifico comando non è in grado di sapere a priori quale specifico comando dovrà eseguire fintantoché alla relativa istanza non venga fornito uno preciso oggetto istanza di una classe che eredita da `Command`. Lo stesso metodo, verosimilmente, si troverà a eseguire svariate tipologie di comandi, ossia a utilizzare diversi oggetti istanze di diverse specializzazioni della classe `Command`. Ciò determina l'impossibilità di associare l'invocazione di un metodo alla relativa implementazione staticamente a tempo di compilazione. In questi casi, il compilatore può unicamente verificare che l'operazione invocata esista con una firma compatibile a quella specificata dall'invocazione stessa.

In un'organizzazione gerarchica ottenuta per mezzo dell'ereditarietà, si effettua un *upcasting* ogniqualvolta un oggetto di una classe discendente viene trattato (*casted*) come se fosse un'istanza della classe progenitrice (ciò avviene in maniera implicita). Per esempio, quando un'istanza di una classe che specializza la classe *Command* è trattata come se fosse di quest'ultimo tipo.

Il termine deriva dal conformare un'istanza alla forma (*casting into a mold*) di una classe superiore (*up*). Mentre, con il termine di *downcasting*, si intende l'operazione opposta, ossia si tenta di trattare un riferimento di un tipo antenato come un'istanza di uno specifico discendente. Per esempio, quando si inserisce un oggetto in una struttura dati come il `Vector`, quest'ultimo effettua, implicitamente un'operazione di *upcasting*: l'og-

getto fornito come parametro viene trattato come se fosse un'istanza della classe `Object`. Mentre, quando lo si preleva, il `Vector` restituisce un'istanza della classe comune `Object` ed è quindi necessario effettuare il *downcasting* alla classe originaria dell'oggetto per poterne utilizzare tutte le caratteristiche.

La variante mostrata fino a questo punto, viene denominata da Booch *monopolimorfismo* per differenziarla dal *polimorfismo multiplo* (*multiple polymorphism*). Questo caso si ottiene combinando l'*overriding* con l'*overloading* dei metodi. Per esempio, si supponga di dotare tutti gli oggetti di tipo `Shape` di un apposito metodo denominato `draw`. Si supponga ancora di prevedere diverse modalità di disegno delle immagini: una normale (`draw()`) e una ridotta (`draw(width : int, heigh: int)`), da utilizzarsi per servizi di anteprima. In questo caso, si potrebbe sia specializzare il metodo `draw` per le varie figure (*overriding*), sia prevederne diverse versioni, con differenti firme per quella ridotta (*overloading*).

Overloading e Overriding

L'autore del libro ha sempre sognato di scrivere un paio di righe su questi due semplici concetti… Si suggerisce comunque di usare questi "paroloni" con moderazione: parlando con presunti programmatori Object Oriented si potrebbe essere tacciati di utilizzare un gergo difficile… Figuriamoci poi se si utilizzassero parole come *up-* e *downcasting*: "Eretico! Eretico! Al rogo!".

Il termine *overriding* è intimamente legato al polimorfismo: quando in una classe discendente si ridefinisce l'implementazione di un metodo, in gergo si dice che se ne è effettuato l'overriding. In sostanza si crea una nuova definizione della funzione polimorfica nella classe discendente. L'esempio è dato dal diagramma relativo al pattern Command in cui la funzione `execute()`, è ridefinita nelle varie classi discendenti da quella astratta `Command`.

Il termine *overloading* ha a che fare con la definizione di diversi metodi con nome uguale, ma firma diversa (non esattamente, visto che la variazione del solo tipo di ritorno non costituisce un *overloading*). Per l'utilizzo di questa tecnica non è necessario dar luogo a particolari legami di ereditarietà.

Nel disegno dei modelli a oggetti è molto importante tentare di utilizzare nomi quanto più chiari e rispondenti alla realtà argomento di studio. Ciò sia per rendere il sistema più facilmente leggibile e quindi mantenibile, sia per tentare di simulare quanto più realisticamente possibile il sistema reale. La scelta dei nomi più adatti non deve essere circoscritta alle classi, ma anche ai metodi: d'altronde rappresentano i servizi forniti. Ora gli stessi servizi possono essere erogati su tipi diversi e con diversi livelli di genericità (leggasi diversi parametri formali) e quindi risulterebbe piuttosto innaturale e decisamente poco chiaro definire metodi diversi per stessi servizi su diversi domini di dati. In ultima analisi, il concetto dell'*overloading* appartiene alla natura umana. Si è abituati a dire "leggi il libro", "leggi il giornale", e così via, non di certo "leggiLibro" o tantomeno "leggiGiornale".

Per esempio, in Java la classe astratta `Component` (da cui deriva la quasi totalità degli oggetti grafici), definisce il metodo `repaint` (il cui significato è evidente) con una serie di overloading, come:

```
repaint() : void
repaint(i : int, j :int, k : int: int) : void
repaint(l : long) : void
repaint(l : long, i : int, j :int, k : int: int) : void
```

A questo punto in cui tutto è chiaro circa l'overriding e l'overloading, si pone il seguente quesito… Il metodo `add(index : int, element : Object) : void`, presente nell'interfaccia `List`, è un esempio di overriding o di overloading? La risposta al quesito è inserita nella sezione "Ricapitolando…" in fondo al capitolo.

Massima coesione e minimo accoppiamento

Con i termini di massima coesione e minimo accoppiamento ci si riferisce a due celeberrimi principi della Computer Science, i cui nomi sono destinati a rimanere indissolubilmente legati. Si tratta di leggi inizialmente formulate per la programmazione strutturata, che poi sono state inglobate tra i princìpi fondamentali del disegno (e quindi della programmazione) Object Oriented.

Per quanto concerne la coesione, Booch afferma che: "un modulo presenta un'elevata coesione quando tutti i componenti collaborano fra loro per fornire un ben preciso comportamento". Da tener presente che il concetto di coesione può essere applicato a diversi livelli di dettaglio (metodi, attributi, classi, package, componenti, ecc.): per esempio, un metodo presenta un elevato grado di coesione quando svolge una sola funzione ben definita. In questo paragrafo si focalizza l'attenzione sull'applicazione a livello di classi.

Da questo punto di vista è possibile definire la coesione come la misura della correlatività delle proprietà strutturali (attributi e relazioni con le altre classi) e comportamentali (metodi) di una classe. Chiaramente si desidera un valore di coesione più elevato possibile (i vari attributi e metodi sono fortemente correlati tra loro). Non è infrequente anche il caso in cui il grado di coesione di una classe venga analizzando limitando l'attenzione alle responsabilità. Si tratta di un livello di astrazione superiore, in quanto, in ultima analisi, sia le proprietà strutturali, sia quelle comportamentali sono disegnate al fine di permettere a una classe di assolvere specifiche responsabilità.

Per esempio, se una classe dovesse rappresentare un utente del sistema e si avessero attributi del tipo `nominativo`, `sesso`, `dataDiNascita`, ecc. combinati con altri del tipo `valuta`, `valore`, ecc. sarebbe piuttosto evidente che i due insiemi di attributi risulterebbero ben poco correlati tra loro (sembra un esempio bizzarro eh?). In termini di database relazionali, il concetto è equivalente alla presenza di dipendenze funzionali di-

verse degli attributi. Ancora, se in una classe rappresentante il carrello della spesa degli utenti di un sistema per il commercio elettronico, si trovassero dei metodi del tipo `svuotaCarrello`, `aggiungiItem`, `verificaValiditaContenuto`, ecc. ed altri del tipo `effettuaOrdine`, `reperisciUtente`, ecc. evidentemente qualche problema di coesione potrebbe esistere anche in questo caso.

Come spesso accade, benché dal punto di vista intuitivo il concetto sia chiaro, si tratta di un criterio non facilmente quantificabile la cui valutazione dipende da diversi fattori non tutti facilmente esprimibili. Ciò che invece risulta possibile è individuare una serie di segnali di allarme in grado di evidenziare disegni non esattamente a elevata qualità per problemi legati a uno scarso grado di coesione. Il più evidente è connesso alle dimensioni delle classi. Qualora una classe contenga troppi attributi, oppure troppe relazioni con altre classi, oppure un numero eccessivo di metodi, molto probabilmente il livello di coesione di questi elementi non dovrebbe essere molto elevato e presumibilmente si ha a che fare con una classe che ne ingloba altre. I problemi generati da classi di questo tipo (troppo prolisse) sono relative alla difficoltà di comprensione del codice e quindi di manutenzione, di laboriosità nell'eseguire i vari test, inabilità nel riutilizzo, difficoltà di isolare elementi soggetti a variazioni, ecc. Un altro segnale chiarissimo si ha qualora non si riesca a identificare un nome preciso per una classe oppure questo risulti troppo generico. Ancora una volta ciò potrebbe essere dovuto al fatto che una classe possiede troppe responsabilità (fa troppe cose). In effetti questo principio ha una sua *ratio* facilmente riscontrabile anche nella vita quotidiana. Se, per esempio, il disegnatore capo deve occuparsi anche del management del progetto, del processo di sviluppo del software, della definizione del disegno di dettaglio, della codifica, sarà ben difficile per lui riuscire ad assolvere a tutti questi impegni garantendo un sommo grado di qualità. In certi casi c'è da essere felici constatando la mancata richiesta di provvedere alle pulizie degli uffici a fine giornata.

I concetti citati per le classi sono facilmente estendibili ad altri elementi, come i package, per esempio. In questo ambito, un buon criterio da seguire per organizzare i modelli di disegni in package è raggruppare le classi al fine di massimizzare la coesione interna del package e minimizzarne l'accoppiamento con gli altri.

Dovrebbe essere ormai chiaro, ma si ritiene opportuno presentare un breve riepilogo dei vantaggi legati a "componenti" software con elevata coesione. In particolare:

- si aumenta il grado di riusabilità dei componenti: disporre di componenti con un insieme ben definito e circoscritto di responsabilità, evidentemente ne aumenta la probabilità di riutilizzo;

- robustezza: un componente con responsabilità ben definite è più facile da comprendere e da verificare, e quindi rende l'intero sistema più robusto.

> In merito al primo punto, molti autori credono che la tanto agognata riutilizzabilità vantata dall'Object Oriented, nella pratica sia più una chimera che una realtà, per tutta una serie di motivi imputabili anche ai disegnatori/programmatori. Ciò che invece è sicuramente dimostrato è che il paradigma dell'Object Oriented semplifica la sostituzione di parti di codice, magari per via del cambiamento dei requisiti, perché si ha bisogno di codice più efficiente, ecc. La facilità di sostituire componenti software non è assolutamente un aspetto meno importante: in ultima analisi accresce la manutenibilità dell'intero sistema che incide notevolmente (circa il 60%) sul ciclo di vita dei sistemi. Qualunque sia la visione, un elevato grado di coesione favorisce sia la riusabilità, sia la sostituibilità dei componenti software.

Nel mondo della costruzione di sistemi informatici, un altro principio di importanza fondamentale è relativo al minimo accoppiamento. Come si vedrà tra breve — limitatamente al settore dell'informatica — tale proprietà è particolarmente ricercata in quanto capace di generare tutta una serie di vantaggi, del tutto equivalenti a quelli generati alla massima coesione: aumento della probabilità di riutilizzo del codice, semplificazione delle attività di manutenzione, ecc. La proprietà di minimo accoppiamento — caso abbastanza raro — possiede sia una definizione, sia un metro di verifica meno problematici.

Per quanto concerne la definizione, si tratta della misura della dipendenza tra componenti software (classi, package, componenti veri e propri, ecc.) di cui è composto il sistema.

Si ha una dipendenza tra due elementi, per esempio classi, quando un elemento (client) per espletare le proprie responsabilità ha bisogno di accedere alle proprietà comportamentali (metodi) e/o strutturali (attributi) dell'altro (server). Chiaramente quest'ultimo non dipende dai client, mentre è vero il contrario. Ciò comporta che un cambiamento all'elemento che fornisce i servizi genera la necessità di revisionare ed eventualmente aggiornare degli elementi client. In sintesi, la dipendenza di un componente da un altro, implica che il funzionamento del componente stesso dipende dal corretto funzionamento di altri componenti. Come si vedrà nel capitolo successivo, la relazione con minore livello di accoppiamento è la dipendenza (nelle varie forme), per poi passare all'associazione, all'aggregazione, terminando con la composizione e l'ereditarietà.

Disponendo di un diagramma delle classi, è possibile verificare il grado di accoppiamento di ogni classe in maniera visiva: è sufficiente contare il numero di relazioni "non entranti" nella classe stessa. In sostanza relazioni per le quali la classe oggetto di studio non svolga un ruolo di fornitore puro di servizi. Con i termini "non entranti" si intende sottolineare che si è interessati unicamente a relazioni di dipendenza in cui la classe recita il ruolo di classe dipendente (client, la classe è attaccata alla coda della freccia) e relazioni in cui la classe all'altra estremità del segmento risulti navigabile. Come si vedrà nel prossimo capitolo, asserire che, relativamente a una specifica relazione, una classe sia navigabile implica che gli oggetti istanza dell'altra (client) devono memorizzare il riferimento

agli oggetti istanza della classe server, al fine di poter accedere ad alcune proprietà strutturali e/o comportamentali della stessa.

> Nella relazione di associazione che lega l'utente alle password, si può notare che la classe `User` non è navigabile. Ciò implica che tale classe deve prevedere un attributo (una lista) al fine di permettere alle proprie istanze di memorizzare i riferimenti agli oggetti `Password` ad esse associate, mentre non deve avvenire il contrario. Le istanze della classe `Password` non devono memorizzare riferimenti agli oggetti `User` a cui appartengono. Pertanto dagli oggetti della classe `User` si può navigare nei corrispondenti della classe `Password`, mentre non è vero il contrario. Dal punto di vista dell'accoppiamento è evidente che la classe `User` è accoppiata a quella `Password`, da cui dipende, mentre la classe `Password` non dipende da quella `User` (*cfr* fig. 6.15).

Un accoppiamento elevato non è desiderabile per una serie di motivi, tra i quali i più importanti sono:

- la variazione di un componente genera a cascata la necessità di verificare ed eventualmente aggiornare i componenti dipendenti;

- qualora si volesse riutilizzare uno specifico componente, è necessario riutilizzare (o comunque portarsi dietro) tutti i componenti da cui questo dipende; ecc.

In merito all'ultimo punto, un elevato accoppiamento può creare un pericoloso ciclo vizioso. Per esempio volendo riutilizzare una classe A è necessario copiare anche la classe B, da cui A dipende, poi la classe C e D utilizzate da B e così via.

L'obiettivo da perseguire è minimizzare il grado di accoppiamento. Chiaramente eliminarlo non avrebbe molto senso: si potrebbe correre il rischio di generare la situazione opposta ovvero classi mastodontiche che non sono accoppiate perché fanno tutto da sole. Si genererebbe quindi un problema di minima coesione…

Spesso nella progettazione di sistemi Object Oriented complessi si "sorvola" qualora le classi presenti all'interno di un package non presentino esattamente un accoppiamento ridotto al minimo: ciò su cui però bisogna porre molta attenzione è che l'accoppiamento tra package sia veramente ridotto al minimo indispensabile. I package dovrebbero raggruppare classi molto correlate tra loro e quindi scarsamente accoppiate con il resto del sistema. Qualora si commetta un errore nel selezionare il package di appartenenza di una classe, questo inconveniente dovrebbe venir immediatamente evidenziato da un corrispondente aumento dell'accoppiamento dei diversi package legato a uno di scarso livello della classe stessa con le restanti presenti nel package di appartenenza.

Per comprendere più scrupolosamente il concetto di accoppiamento lo si esamini a un livello di dettaglio inferiore: al livello dei metodi delle classi.

In prima analisi, i metodi di ciascuna classe possono essere suddivisi in tre macrocategorie:

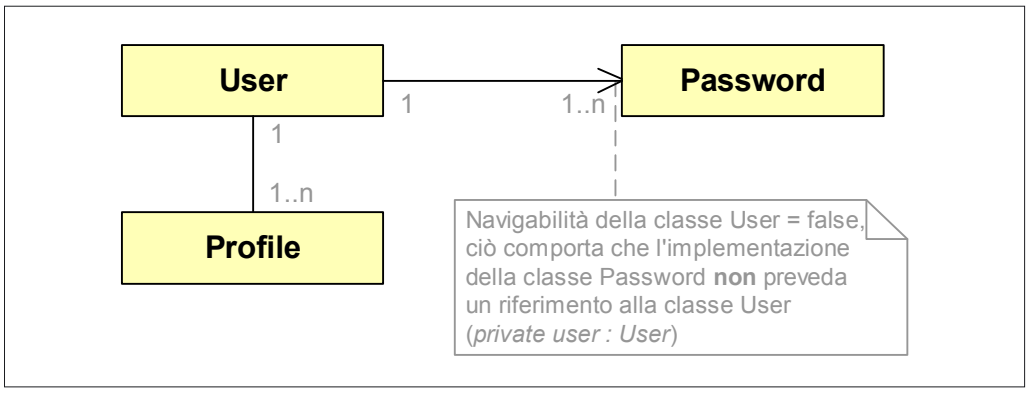
1. metodi che forniscono un servizio semplicemente elaborando i dati di input senza ricorrere all'utilizzo di altri dati e tanto meno senza utilizzare lo stato dell'oggetto (per esempio in Java i metodi della classe `java.lang.Math`, come `Math.abs()`);

2. metodi che comunicano una porzione dello stato interno di un oggetto, oppure elaborano risultati dipendenti da esso, senza modificarli (per esempio i metodi `getX()`);

3. metodi che aggiornano lo stato interno di un oggetto (per esempio i metodi `setX()`).

Per quanto concerne la prima tipologia di metodi, essi presentano un accoppiamento minimo quando risultano privi di effetti collaterali — i famigerati *side effects* — e operano dunque esclusivamente sui parametri di input. Qualora questi metodi utilizzino altri dati, magari privati all'oggetto, di cui però si abbia strettamente bisogno, si ha ancora un accoppiamento contenuto, ma non più minimo. Per quanto concerne i risultati generati, il metodo deve produrre unicamente un dato atomico o eventualmente un altro oggetto di cui viene fornito il riferimento in memoria. Per mantenere un accoppiamento minimo, il

Figura 6.15 — *Frammento di diagramma delle classi relativo agli elementi* `User`, `Profile` *e* `Password`. *In particolare un utente è associato con diverse parole chiave di cui una sola è quella attiva (ciò non si evince da questo particolare diagramma): per esempio è importante memorizzare quelle utilizzate in passato da ogni singolo utente per evitare che le riutilizzi. Per ciò che concerne il legame con i profili, si tratta esattamente della stessa situazione: un utente dispone di diversi profili, ma solo uno è quello attivo.*



metodo, durante la propria esecuzione, non deve poi delegare ad altri parte del proprio processo (non deve invocare altri metodi). Da quanto riportato risulta evidente che non sempre un accoppiamento minimo è assolutamente indispensabile e desiderabile. Anzi spesso, sono accettabilissimi alcuni compromessi, magari al fine di soddisfare altri requisiti di qualità del software, come rendere i metodi più leggibili, manutenibili, riusabili, per esempio, magari derogando parte del loro lavoro ad altri metodi.

Nel caso di metodi del secondo tipo, si ha un accoppiamento minimo quando il metodo, per generare i risultati della propria elaborazione, utilizza i parametri di input ed accede ai soli attributi e metodi della classe, sia statici che non. Ancora una volta restituisce un valore atomico o un riferimento a un apposito grafo di oggetti o, eventualmente, genera un'eccezione per comunicare uno stato di errore. Metodi di questo tipo, pertanto, accedono allo stato dell'oggetto senza però modificarlo e utilizzano esclusivamente proprietà (metodi e attributi) della classe o dell'oggetto stesso.

Per i metodi dell'ultimo tipo, la materia non varia di molto. La differenza è che la propria esecuzione altera lo stato dell'oggetto. Chiaramente un accoppiamento minimo non prevede la variazione dello stato di altri oggetti.

Premesso ciò, è possibile formulare una definizione, per così dire *induttiva*, di minimo accoppiamento a livello di classe, ossia "una classe prevede un livello minimo di accoppiamento quando tutti i rispettivi metodi presentano un livello minimo di accoppiamento".

Da tenere presente che limitare la propria attenzione ai soli metodi sarebbe un errore piuttosto grossolano: un forte accoppiamento tra classi si instaura con relazioni di ereditarietà. In questo caso si ha accoppiamento tra classi/interfacce quando:

- una classe o interfaccia sottoclasse discende (o eredita) dall'altra;
- una classe implementa un'interfaccia;

Abstract Data Type (tipo di dato astratto)

Tutti coloro che si sono formati nel settore dell'Object Oriented attraverso lo studio dei "testi sacri" ricorderanno — forse — con un certa nostalgia la teoria relativa all'ADT. In effetti molti autori si riferiscono alla progettazione Object Oriented come la pianificazione di tipi di dati astratti e delle relazioni tra essi. Ciò è del tutto consistente, specie se si ricorda che in linguaggi puramente Object Oriented (quali per esempio Small Talk) ogni entità è un oggetto, anche quelle relative a elementi base come numeri interi, caratteri, e così via. Chiaramente in questi paragrafi non è possibile illustrare tutti i dettagli dell'ADT, però si è ritenuto comunque importante presentarne le basi formali corredate da qualche esempio: si cercherà di evitare quello inflazionatissimo dei tipi di dato relativi ai numeri complessi.

Come riportato nei paragrafi precedenti, una delle prerogative del paradigma Object Oriented consiste nel realizzare sistemi attraverso la descrizione dello spazio del proble-

ma, in sostanza il sistema dovrebbe essere composto dalla trasposizione Object Oriented del vocabolario dell'area business. Quindi, in teoria, la lettura di un sistema realizzato secondo i principi dell'Object Oriented dovrebbe coincidere con la lettura delle business rules del dominio che il sistema dovrà, in qualche modo, automatizzare. Nella pratica, poi, ciò non è quasi mai possibile per una serie di motivi, non ultima la necessità di realizzare l'infrastruttura informatica che, ovviamente, non vive nello spazio del problema…

In ogni modo, uno dei primi problemi che bisogna affrontare consiste nel realizzare un modello dell'area business. Per coloro che hanno avuto il piacere di studiare la teoria di Booch, si tratta di dare una forma alle varie nuvolette identificate. A tal fine è necessario comprendere l'area business oggetto di studio, riuscire a distinguere i dettagli trascurabili da quelli oggetto di interesse, rappresentare questi ultimi formalmente ecc. Occorre quindi applicare il processo dell'astrazione al fine di rappresentare una vista del mondo reale confacente ai fini preposti. Per esempio, dovendo definire un modello per un sistema di *back office* di una *investment bank*, il modo di lavorare tipico dei *broker*, il relativo linguaggio utilizzato, le informazioni consultate ecc., sebbene possano essere intrinsecamente molto interessanti, lo sono molto di meno dal punto di vista del modello da realizzare. In tale contesto si sa che giungono informazioni relative ai *trade* stipulati nel *front office* di cui bisogna farsi carico, mentre non si ha alcun interesse nel sapere come questi vengono contrattati.

Una volta terminato il processo d'astrazione — capita di visionare modelli che potrebbero far invidia ai capolavori di Dalì — dovrebbero emergere due categorie di informazioni:

- dati trattati nel sistema, opportunamente raggruppati e relazionati tra loro;
- funzioni che, agendo sui dati stessi, realizzano specifici servizi.

Si supponga che sia necessario rappresentare le valute coinvolte nel sistema bancario. Una lista di attributi individuabili potrebbe essere:

- codice standard (ISO);
- nome in inglese;
- nome in lingua originale;
- descrizione;
- data di istituzione;
- simbolo della valuta;

- numero di cifre decimali significative;
- elenco dei formati (tagli) previsti;
- nome delle frazioni;

e così via.

Chiaramente, nell'ambito del sistema da realizzare, non si è sempre interessati a tutti gli attributi. Alcuni di essi, tipicamente, diventano trascurabili poiché non sono specifici del problema, mentre magari sarebbero di notevole interesse in contesti diversi.

La selezione degli elementi di interesse costituisce un primo passo verso la modellazione. In effetti, si realizza un modello della valuta con tutti e solo le informazioni che soddisfano i requisiti del sistema.

La descrizione dei dati richiesti è sicuramente un'attività molto importante, ma decisamente non sufficiente: è necessario definire anche i servizi a cui si è interessati. In altre parole è necessario definire rigorosamente le operazioni eseguibili sull'ADT e definire la visibilità delle stesse.

Il processo di astrazione permette di rappresentare il "nebuloso" spazio del problema in una ben definita struttura di entità corredate di dati e operazioni che nel paradigma Object Oriented devono essere considerate come caratteristiche inscindibili.

Evidentemente ciò che fino a questo punto si è definito entità non è altro che il concetto ben noto di classe: ciascuna di esse definisce la struttura dei dati corredata dalle operazioni con le quali è possibile accedere ad esse. L'insieme delle operazioni rappresenta l'interfaccia esposta dall'entità. Un'entità che rispetti quanto detto finora rappresenta un tipo di dato astratto che comunemente viene detto "classe".

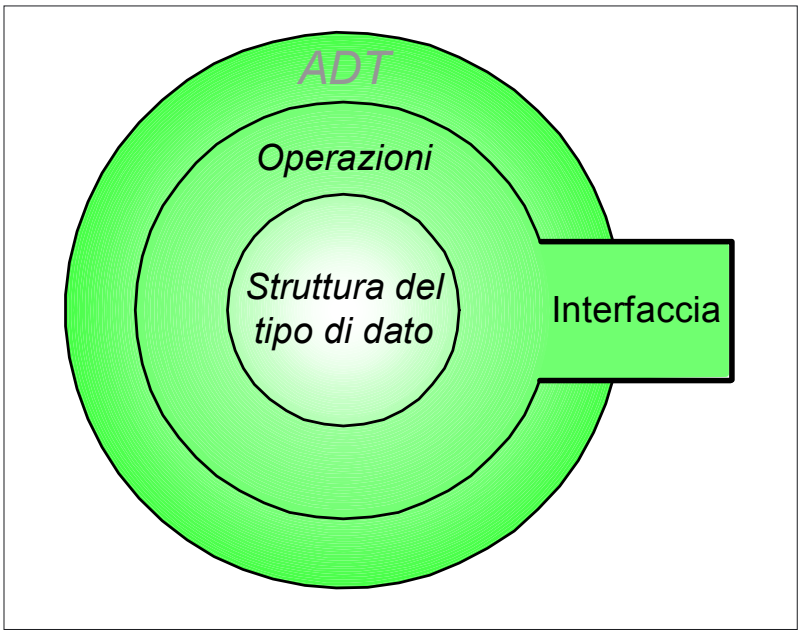
Una volta che un ADT è creato, e la struttura di dati viene riempita con gli opportuni valori, si ha quella che viene definita un'istanza. A meno di vincoli particolari, è possibile generare quante istanze si desidera di ogni ADT. Considerando il caso della valuta, si potrebbero avere, per esempio, istanze relative all'euro (EUR), al dollaro americano (USD), alla sterlina del regno unito (GBP) e così via.

Proprietà

Le proprietà che caratterizzano un ADT sono:

1. il tipo che esportano;
2. l'interfaccia, ossia le operazioni esposte che rappresentano l'unico meccanismo utilizzabile per accedere ai dati;

Figura 6.16 — *Rappresentazione schematica del tipo di dato astratto (Abstract Data Type).*



3. assiomi e condizioni dettate dallo spazio del problema.

Per quanto concerne la prima proprietà, c'é ben poco da dire. Una volta definito un nuovo tipo di dato, utilizzando apposite operazioni (costruttori), è possibile generare quante istanze del tipo si desiderano. Dietro il secondo punto si nasconde ovviamente la legge dell'incapsulamento descritta nell'apposito paragrafo. Da tener presente che in questo contesto il principio è molto importante anche perché si vuole definire una serie di ADT non vincolati a uno specifico linguaggio di programmazione — prescindendo dalla particolare implementazione — la cui logica interna può variare senza che ciò abbai ripercussioni sulla restante parte del sistema che la utilizza. Per quanto concerne poi l'ultimo punto, è evidente che oltre a definire ogni funzione, in base ai princìpi che regolano l'ADT, è necessario evidenziare opportune condizioni. Per esempio, al fine di ottenere un elemento da una tabella è necessario fornire un valore non vuoto per la chiave, se si desidera effettuare il *pop* su uno *stack*, è necessario che questo non sia vuoto, e così via.

Una volta definito un ADT, a meno di particolari vincoli, è possibile definire quante istanze si desiderano. Non è infrequente il caso in cui la definizione di un ADT sia basata su quella di altri o li includa, al fine di fornire i servizi esposti nell'interfaccia. L'esempio

classico è dato dalle collezioni. In effetti è possibile definire liste, pile, code, hashtable, ecc. che per essere utili devono operare necessariamente su altri dati. Per esempio liste di interi, liste di stringhe, ecc. Si tratta ancora degli stessi elementi, ma ciò che varia è il tipo di dato su cui si opera. In sostanza si realizzano tipi di dato astratti parametrici denominati tipo di dato astratto generico (*Generic Abstract Data Type*), le cui "specializzazioni" si ottengono dichiarando il tipo di dato su cui deve operare.

> Per chi ha esperienza del mondo C++, si tratta dei famosi *template* che, vista la potenza, sembrerebbero dover confluire nella prossima revisione del linguaggio Java a totale carico del compilatore, cioè senza modifiche alle specifiche della VM.

Per esempio, si supponga di aver definito un elemento di tipo lista; la versione relativa alle stringhe sarà:

```
List<string> ListOfString
```

dove, l'elemento racchiuso tra parentesi angolari rappresenta la specifica variante del tipo di dato astratto generico. Il nuovo elemento condivide la stessa interfaccia di quello generico (`List`) solo che le relative operazioni sono eseguite su un tipo stringa.

Notazione

Con l'avvento dello UML potrebbe essere del tutto legittimo utilizzare il formalismo dei diagrammi delle classi per descrivere gli ADT (rimarrebbe però il problema di specificare la logica delle operazioni). Ciò nonostante, la notazione storica è di carattere più descrittivo e meno grafico. In ogni modo si tratta di una notazione astratta e quindi non legata a un singolo linguaggio. Da quanto riportato fino a questo punto, dovrebbe essere abbastanza chiaro che le parti costituenti sono due:

- **dati**: in questa sezione è riportata la descrizione della struttura utilizzata dal particolare ADT. La descrizione può essere effettuata con diverse notazioni, sebbene non sia infrequente il caso in cui se ne seleziona una informale, specie per descrivere strutture non banali. La sezione dati potrebbe essere omessa (espressione massima di incapsulamento) se non fosse per necessità relative alla definizione formale dei metodi;

- **operazioni** (l'interfaccia): anche in questo caso è possibile selezionare diversi livelli di formalità, sebbene sia opportuno tentare cercare di essere più rigorosi possibili. Una descrizione Object Oriented dei metodi dovrebbe prevedere la dichiarazione esplicita degli argomenti dei metodi (firma) e delle relative condizioni (pre-, post- e durante l'utilizzo). Queste ultime informazioni, come si vedrà nel prossimo para-

grafo, sono la base della tecnica nota con il nome di "disegno per contratto" (*Design by Contract*). La lista delle operazioni si presta a essere specializzata in quattro sottocategorie:

1. costruttori: descrivono le azioni da eseguire per creare un'istanza del tipo di dato;
2. distruttori: si tratta delle operazioni speculari alle precedenti e descrivono le azioni da compiere prima di distruggere l'istanza;
3. selettori: questi permettono di ottenere informazioni relative a un'istanza senza modificarne lo stato;
4. modificatori, si tratta di metodi che variano lo stato interno dell'istanza dell'ADT.

Per cui la notazione potrebbe essere:

```
ADT <nome>
DATA
        <descrizione della struttura dei dati>
OPERATIONS
        constructor:
                <definizione del costruttore>
        destructor:
                <definizione del distruttore>
        <operazione>
                <descrizione dell'operazione>
        ...
END
```

Per quanto concerne la definizione formale, un ADT è costituito dalla tripla <D, F, E> dove:

- D è l'insieme dei possibili domini $\{D_1, D_2, \ldots, D_n\}$, all'interno del quale è possibile individuare quello di interesse per l'ADT che si sta definendo;
- F è l'interfaccia, ossia l'insieme delle possibili funzioni $\{F_1, F_2, \ldots, F_m\}$ che possono essere eseguite sulla struttura di dati dell'ADT. Ciascuna di esse è vincolata ad avere dominio e codominio (banalizzando, rispettivamente insieme dei valori di input e quelli di output) appartenenti a D e inoltre, il dominio di interesse deve essere o il codominio della funzione, oppure essere presente tra gli insiemi che ne definiscono il relativo dominio;
- E è l'insieme degli elementi che denotano valori di particolare importanza.

Probabilmente a questo punto si è riusciti a rendere complicato un concetto che per sua natura non lo sarebbe. Al fine di eliminare questo effetto si consideri l'originalissimo esempio dello *stack*. Si tratta della versione di lista governata dalla regola: *Last In First Out* (l'ultimo arrivato è il primo a essere prelevato), in cui un nuovo elemento viene inserito nella cima della pila (*TOS*, *Top Of Stack*, cima della pila) e quindi è il primo a poter essere eliminato.

La versione presa in esame fa riferimento a uno stack con capacità di memorizzazione infinita, in altre parole non contempla lo stato `Pieno`.

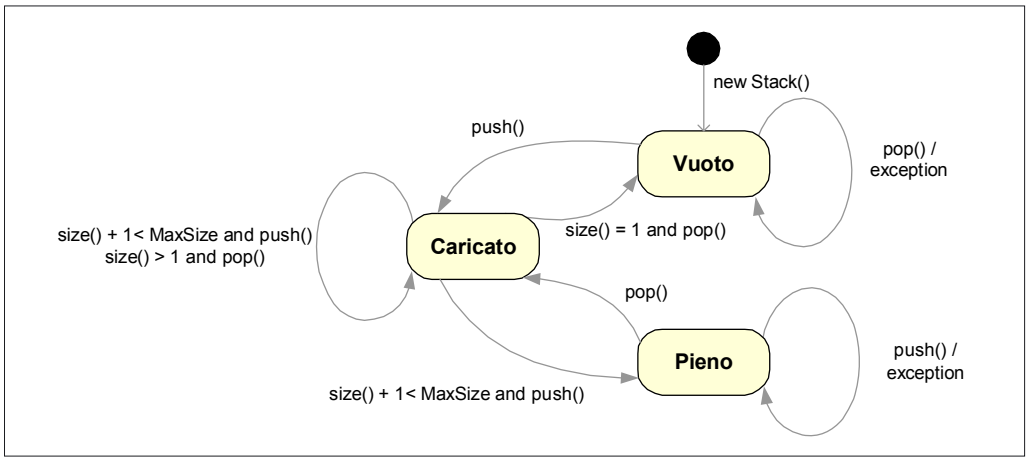
In questo ambito gli elementi `<D, F, E>` sono:

- `D` è l'insieme dei domini, ossia `{stack, tipo_base, boolean}`;
- `F` è l'insieme delle operazioni applicabili sull'ADT;
- `E` è l'insieme delle costanti che in questo caso contempla un unico elemento: `EmptyStack` (ossia pila vuota).

Prima di definire le operazioni è necessario fissare le convenzioni base: `S` è un'istanza dello Stack e `x` è un particolare valore del `tipo_base`; a questo punto le operazioni di interesse sono:

- `new(): →Stack`. Genera un nuovo stack vuoto (costruttore).
- `reset(S):Stack→Stack`. Restituisce una particolare pila, ossia quella vuota (`EmptyStack`);
- `isEmpty(S):Stack→boolean`. Restituisce `true` se la pila è vuota, `false` altrimenti;
- `push(S,x):(Stack,tipo_base)→Stack`. Restituisce un nuovo `Stack` ottenuto da quello di input, aggiungendo il carattere `x` in cima. Se lo `Stack` di input è formato dai valori $\{a_n, a_{n-1}, \ldots, a_1, a_0\}$ (in cui a_n è l'ultimo elemento inserito), quello ottenuto come risultato della funzione `push(S,`x_0`)` è $\{x0, a_n, a_{n-1}, \ldots, a_1, a_0\}$.
- `pop(S):Stack→Stack`. Questa funzione è l'opposta della precedente; è dato uno `Stack` di input, ne produce uno di output, ottenuto eliminando l'elemento in cima alla lista dallo stack di input. Quindi se la pila di ingresso è $\{a_n, a_{n-1}, \ldots, a_1, a_0\}$, il risultato della funzione `pop(S)` è $\{a_{n-1}, \ldots, a_1, a_0\}$. Questa funzione è applicabile qualora la pila di ingresso non sia vuota (precondizione).

Figura 6.17 — *Diagramma degli stadi di uno Stack.*



- `tos(S):Stack→tipo_base`. Qualora la pila di input non sia vuota, restituisce l'elemento in cima.

Da notare che queste funzioni si prestano ad essere descritte formalmente, e indipendentemente da un linguaggio di programmazione, attraverso l'OCL.

La rappresentazione formale prevederebbe:

```
ADT Stack
DATA
        Array pila di n elementi del tipo base;
        top intero indicante l'elemento in cima alla pila
OPERATIONS
        constructor:
                new()
                post: top = -1, result = EMPTY_STACK
        selectors:
                isEmpty(stack : Stack)
                        post: result (stack.top == -1)
                tos(stack : Stack) : BasicType
                        pre : stack.isEmpty() == false
                        post : result = pila[top]
        modifiers:
                push(stack : Stack, x : BasicType)
                        post: pila[++top] = x
                pop(stack : Stack)
                        pre : stack.isEmpty() == false
                        post : top—
END
```

Tre parole sul Design by Contract (DbC, disegno per contratto)

Introduzione

Una delle caratteristiche imprescindibili di ogni sistema ingegneristico, e non solo, dovrebbe essere l'*affidabilità* (di nuovo la vocina interna dell'autore diviene incontrollabile). In effetti nessuno acquisterebbe un'autovettura che in curva sia incline a capovolgersi o installerebbe un ascensore che tenda a bloccarsi, o tanto meno costruirebbe un ponte che oscilli spaventosamente, e così via (chissà poi come mai per ciascuna delle precedenti asserzioni esiste una controprova). Anche nei sistemi software chiaramente si è interessati a questa caratteristica (ancora una volta raramente soddisfatta… Purtroppo alcuni celebri errori in sistemi informatici, sono tragicamente passati alla storia.)

Nell'ambito dei sistemi software, il termine di affidabilità implica due altri concetti:

- correttezza: il sistema realizza le funzioni per il quale è stato progettato. Quindi a stimoli corretti di ingresso produce gli effetti di output previsti;

- robustezza: il sistema è in grado di rilevare e gestire situazioni anomale (dati di ingresso non corretti, parti del sistema non funzionanti, ecc.).

In parole povere l'affidabilità è sintetizzabile con ciò che sembra essere un miraggio informatico: la realizzazione di sistemi privi di errori (bug). Se poi si considera che un altro sogno dell'Object Oriented (enfatizzato dai sistemi component-based) è la riutilizzabilità del codice, si comprende come questa caratteristica assuma ancora più rilevanza.

Il problema da porsi è come mai sia così difficile realizzare sistemi affidabili, o meglio ancora, quale tecnica utilizzare per costruire sistemi affidabili. Chiaramente non esiste una risposta univoca e tanto meno semplice. Esistono tuttavia una serie di comportamenti e di *best practices* che permettono di aumentare la qualità e l'affidabilità del software. Un approccio particolarmente apprezzato è il famoso *Design by Contract* (disegno per contratto, tecnica ideata da Bertrand Mayer nei laboratori del linguaggio Eiffel). Come si vedrà, si tratta di una metodologia particolarmente valida per via della sua sistematicità.

> Sebbene si tratti di una tecnica la cui validità è universalmente accettata, il suo successo sembra essere concretamente inferiore alle attese. L'utilizzo è principalmente relegato ad ambienti accademici. Ciò probabilmente è dovuto sia al grande quantitativo di tempo necessario per la formulazione matematica delle varie condizioni, sia a lacune presenti "nativamente" in alcuni linguaggi di programmazione, sebbene siano poi disponibili dei pre-processori per diversi linguaggi di programmazione.

Il punto di partenza è che il sistema può essere immaginato come una serie di componenti (eventualmente oggetti, d'altronde Eiffel è uno dei linguaggi orientati agli oggetti più formali) comunicanti, la cui interazione è basata su precise obbligazioni tra oggetti richiedenti i servizi (*client*) e quelli che invece li forniscono (*supplier*). Tali obbligazioni costituiscono quello che viene definito *contratto*. In quest'ambito si fa riferimento ad accordi tra due parti (probabilmente sarebbe più opportuno esprimersi in termini di tipologie di parti) e pertanto alcuni obblighi risultano a carico del richiedente mentre altre sono di responsabilità del fornitore. Ovviamente ciascuna delle parti si aspetta di ottenere dei benefici dal contratto e per questo accetta, come contropartita, alcune obbligazioni. In ultima analisi, un contratto stabilisce, in maniera formale, l'elenco dei benefici e delle obbligazioni che ogni parte contrae. Da una parte, ogni oggetto cliente si fa carico di rispettare le condizioni pattuite nel richiedere i servizi, mentre dall'altra i server assicurano il rispetto di determinate condizioni relative la fornitura dei servizi. Qualora un cliente non rispetti le condizioni pattuite, ovviamente il contratto perde di validità e quindi l'altra parte non è più vincolata a rispettare le obbligazioni pattuite.

Chiaramente il concetto di contratto è un qualcosa che appartiene alla vita quotidiana e viene utilizzato sistematicamente, più o meno consapevolmente, da ogni persona. I contratti si applicano ogniqualvolta ci sia uno scambio di prodotti (e/o servizi) tra un cliente e un fornitore. Per esempio, quando si utilizza la carta di credito per effettuare degli acquisiti, implicitamente si utilizza un servizio governato da un apposito contratto. Una semplificazione di questo contratto è riportato nella tab. 6.1.

Un contratto è quindi vantaggioso e protegge entrambi i contraenti sancendo, da una parte, natura e qualità del servizio (postcondizioni) e, dall'altra, sollevando il fornitore (precondizioni) da ogni imputabilità qualora il cliente richieda la fornitura del servizio non rispettando gli accordi.

Si consideri ora un esempio di contratto applicabile a una classe di un sistema software. A tal fine si consideri l'inserimento di un oggetto in un "dizionario" (map). Queste strutture permettono di gestire un insieme di coppie (chiave, valore), pertanto vi è una corrispondenza biunivoca tra le chiavi e i relativi valori. Ciò implica che gli elementi inseriti siano reperibili specificando il valore della relativa chiave (tab. 6.2).

Come visto anche nella presentazione dell'Abstract Data Type, il contratto governa le interazioni tra ogni metodo di un oggetto e i potenziali clienti. Pertanto, per ogni metodo, definisce informazioni di vitale importanza: le assicurazioni che ogni parte deve garantire per una corretta fruizione del servizio (invocazione del metodo) e il conseguimento del risultato pattuito.

Nuovamente anche questo argomento, probabilmente, non è privo di malintesi. Diversi autori sostengono che il soddisfacimento di una precondizione implica che il cliente "debba garantire lo stato dell'oggetto server prima dell'invocazione di un suo servizio". L'autore del presente testo non condivide questo punto di vista. Sembra addirittura un

Tabella 6.1 — *Esempio semplificato del contratto d'uso di una carta di credito*

Tabella 6.2 — *Contratto di utilizzo di un dizionario*

PARTE	OBBLIGAZIONI	BENEFICI
Cliente	*(Rispettare le pre-condizioni)* Assicurare che il valore della chiave non sia nullo	*(Ottenuti dalle post-condizioni)* Memorizzare la coppia chiave valore.
Fornitore	*(Assicurare le post-condizioni)* Inserire il valore nel dizionario	*(Assunzione delle pre-condizioni)* Non deve eseguire alcuna operazione qualora la chiave sia nulla

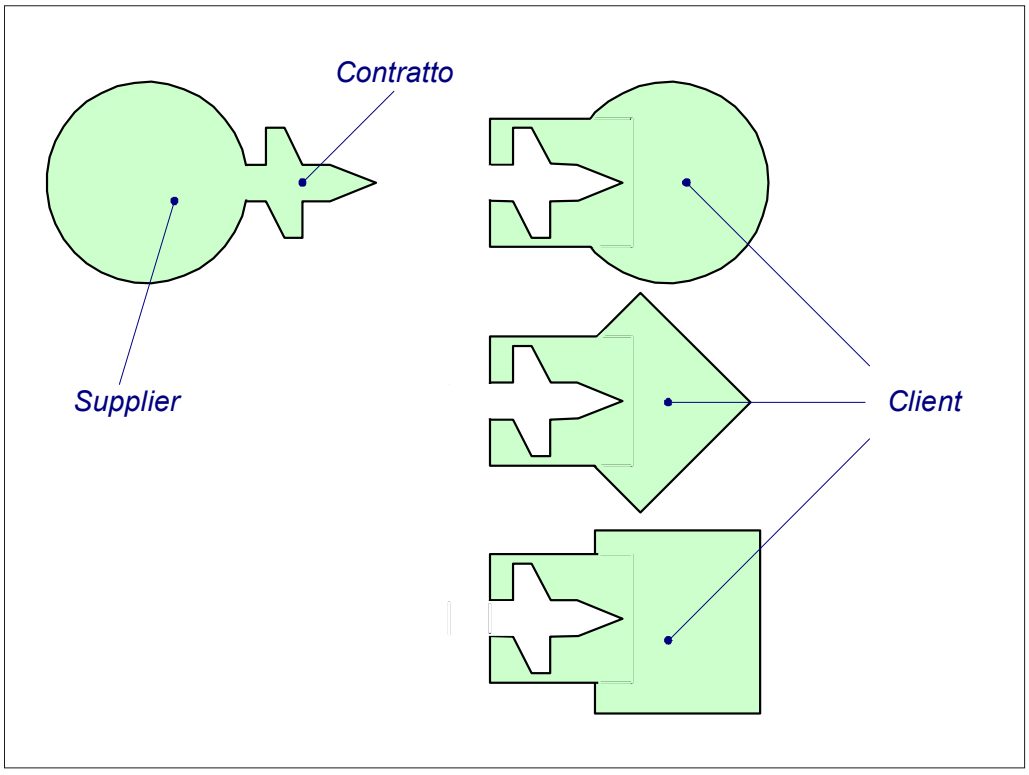
controsenso che un oggetto debba garantire lo stato interno di un altro che fornisce il sevizio. Ciò che invece appare più verosimile è che il cliente possa garantire alcune condizioni sui parametri di un servizio. Per esempio, dovendo invocare una funzione che calcoli la radice di un numero, l'oggetto cliente può e deve farsi carico di verificare che l'argomento sia un numero non negativo, oppure nel richiedere l'autorizzazione di una transazione commerciale, verificare che l'importo sia diverso da zero, ecc. Il fatto che alcuni obblighi siano a carico dell'oggetto client non significa poi che nell'oggetto server i controlli non siano eseguiti. In sostanza si applica ancora una volta il famoso concetto della fiducia con verifica. Un esempio apparentemente semplice ma che invece presenta delle anomalie è relativo al metodo di `pop` di uno stack. Verosimilmente si potrebbe dichiarare qualcosa del genere:

```
precondition:   this.isEmpty() == false;
postcondition:  return this.getTOS();
```

Riflettendo bene, benché sia strettamente necessario eseguire la verifica sulla condizione di stack vuoto, probabilmente non è corretto esprimersi in termini di precondizione. Ciò perché forzerebbe il cliente a verificare lo stato interno dell'oggetto server. Se poi ci si trovasse in un sistema concorrente, allora tra una invocazione del metodo `isEmpty` e `pop` potrebbe avvenire di tutto (anche passaggio dello stack da non vuoto a vuoto) e quindi come potrebbe un cliente essere sicuro dello stato dell'oggetto client? Chiaramente nessuno vorrebbe inserire il concetto di lock nel Design by Contract.

Fino a questo punto si è visto come pre- e postcondizioni si applicano al livello di servizio, o metodo se si vuole. Esiste tuttavia un'altra serie di asserzioni il cui dominio invece è la classe. Tali condizioni sono dette *invarianti* (queste sono state presentate brevemente nel capitolo panoramico). Si tratta di condizioni valide per tutte le istanze di una specifica classe, detta per l'appunto classe invariante. Per esempio, la valutazione del metodo `isOfAge()` (è maggiorenne) per tutte le persone che posseggono di una carta di

Figura 6.18 — *L'immagine rappresenta un'amabile schematizzazione del Design by Contract. Si tratta della rielaborazione di una figura presente nel libro [BIB16].*



credito deve restituire il valore `true`. La dichiarazione di questi vincoli è molto importante sia per l'implementazione della classe stessa, sia per lo sviluppo dei test poiché stabilisce dei criteri che devono essere sempre soddisfatti dalle istanze della classe, indipendentemente dalla relativa evoluzione.

A questo punto l'idea del contratto alla base della relativa tecnica dovrebbe assumere una forma più compiuta: esso è definito in termini di condizioni, ognuna delle quali appartiene a una delle seguenti tipologie:

- **precondizioni**: vincoli che gli oggetti client devono rispettare per poter fruire di un servizio;
- **postcondizioni**: garanzie assicurate dal fornitore del servizio, qualora le precondizioni siano rispettate;
- **invarianti**: condizioni sempre valide.

Le varie condizioni possono essere considerate delle asserzioni (espressioni booleane relative allo stato dell'oggetto) la cui valutazione a tempo di esecuzione deve restituire un valore `true` per il corretto proseguimento. Il caso in cui la valutazione restituisca un valore `false` costituirebbe la segnalazione della violazione di una clausola del contratto. Si tratta quindi di *check point* (punti di controllo) inseriti in luoghi precisi del sistema. In particolare è necessario verificare le precondizioni immediatamente prima di eseguire un metodo (all'entrata), le postcondizioni subito dopo la fine dell'esecuzione o meglio poco prima della fornitura dei risultati (all'uscita), mentre le invarianti dovrebbero poter essere valutate in ogni momento in quanto definiscono condizioni che dovrebbero sempre essere verificate.

Dovrebbe risultare abbastanza chiaro che l'Object Constraint Language è un formalismo che si presta facilmente per specificare rigorosamente i vincoli e le condizioni del disegno per contratto. Per essere precisi, è molto più potente.

Il Design by Contract è stato inizialmente ideato per essere utilizzato in fasi ad elevato dettaglio tecnico, per quanto, recentemente, diversi autori ritengano che il suo utilizzo possa includere le prime fasi del ciclo di vita di un sistema software, caratterizzate da un maggior livello di astrazione rispetto a dettagli tecnici. Sebbene si tratti di una rappresentazione a livello di ADT e quindi non legata a uno specifico linguaggio di programmazione, probabilmente un utilizzo durante le fasi di analisi potrebbe creare qualche difficoltà, legata sia al tempo necessario per una loro corretta formulazione, sia alla difficoltà che riscontrerebbero gli utenti medi a fruirne i relativi contenuti.

Rimanendo in un ambito puramente implementativo, è possibile notare come il luogo ideale per la definizione formale dei contratti dovrebbe essere dato dalle interfacce che espongono servizi forniti dalle relative classe. Il problema è che ciò non sempre è possibile.

Per esempio, linguaggi come Java non consentono (nativamente) di specificare più della firma dei metodi esposti da una classe (per essere precisi permettono di specificare anche valori costanti), mentre non è possibile specificare nulla relativamente al comportamento.

> Ad onore del vero esistono vari strumenti non standard in grado di estendere la grammatica Java abilitando questo linguaggio all'utilizzo della tecnica del disegno per contratto. Uno dei più celebri è indubbiamente *iContract* realizzato da Reto Kramer. Si tratta di uno strumento molto intelligente che permette di dichiarare le varie condizioni (pre-, post- e invarianti) nel commento iniziale, sia dei metodi (@pre, @post), sia della classe (@inv), attraverso opportuni tag, in modo del tutto analogo a JavaDoc. Ciò fa sì che il codice, processato da un opportuno pre-compilatore, sia in grado di utilizzare le peculiarità del Design by Contract, grazie alla generazione automatica di tutta una serie di classi utilizzate per verificare il rispetto delle condizioni, mentre compilato direttamente con javac dia luogo alla generazione del codice atteso. La prima modalità potrebbe rilevarsi molto utile in fase di test, mentre la seconda, essenzialmente per questioni di performance, dovrebbe essere preferita per la produzione. La grammatica prevista per la specifica delle condizioni è un sottoinsieme del linguaggio OCL.

Il sottocontratto

Nell'ambito della progettazione dei sistemi Object Oriented, un aspetto di particolare importanza è dato dalla combinazione della metodologia del *Design by Contract* con l'ereditarietà. In altre parole è necessario esaminare come un contratto pattuito da una classe antenata sia ereditato da una classe discendente. Il problema è che non si ha a che fare unicamente con l'ereditarietà, bensì anche con concetti quali il polimorfismo: non è infrequente il caso di una classe discendente che ridefinisce un metodo dichiarato in una antenata. Questa situazione costituisce sicuramente una fonte di potenziale pericolo. La soluzione a questo problema è data dal concetto del *subcontratto*, il quale stabilisce una serie di norme derivanti essenzialmente dal principio di sostituibilità (ogni istanza di una classe discendente deve poter essere utilizzata in ogni punto in cui è previsto un oggetto di una classe antenata).

In particolare, una classe ereditante, per ogni metodo, può:

- mantenere le stesse precondizioni definite in quella antenata oppure definirne altre più deboli;
- mantenere le stesse postcondizioni definite in quella antenata oppure definirne altre più forti;

Queste regole rappresentano un sottocontratto definito onesto, il cui non rispetto può generare non pochi problemi. Da tenere presente che in linguaggi come Java, il concetto

del sottocontratto diviene importante per classi figlie, implementanti un'interfaccia, annidate e per interfacce estendenti altre.

Eccezioni

Il concetto della gestione delle eccezioni, sebbene, non sia di per sé un aspetto primario della teoria del Design by Contract, in effetti rappresenta un aspetto di tipo implementativo, mentre la teoria resta pur sempre un fattore con il quale prima o poi bisogna confrontarsi. Come visto in precedenza ogni oggetto possiede un'interfaccia, alcuni la espongono esplicitamente per mezzo di appositi elementi, mentre in altri si esaurisce in quella implicita. Come logica conseguenza, ciò permette di affermare che ogni oggetto possiede un contratto e nuovamente ci sono oggetti che lo dichiarano apertamente, mentre per altri è più implicito (quest'ultima condizione tende ad aumentare il carico di lavoro in fase di documentazione).

Un fallimento nel rispetto delle condizioni sancite da un contratto porta all'impossibilità da parte della classe server di fornire il servizio richiesto e, in ultima analisi, a un'eccezione che necessita di essere gestita; chiaramente non è mai opportuno lasciare il sistema in uno stato indesiderato, che in questo contesto potrebbe significare violare le invarianti di un oggetto. Il guaio è che le eccezioni possono insorgere anche durante la fornitura di un servizio per cause indipendenti dalle parti, come per esempio un problema dovuto a un malfunzionamento dell'hardware. In ogni modo ciò che è necessario svolgere, pragmaticamente, indipendentemente dalla tipologia dell'eccezione, consiste nel riportare l'oggetto server in uno stato consistente e quindi notificare l'anomalia.

Vantaggi

Come degna conclusione del paragrafo si è deciso di specificare formalmente i vantaggi della metodologia del Design by Contract:

- fornisce una tecnica sistematica per la realizzazione di sistemi senza errori;
- realizza un framework per il debugging e il test delle diverse parti del sistema;
- permette di documentare più agevolmente i sistemi software (come si è visto, si tratta di una tecnica derivante direttamente dall'Abstract Data Type);
- agevola la rilevazione e gestione di condizioni anomale, ecc.

Il Design by Contract è a tutti gli effetti un'evoluzione del disegno Object Oriented e pertanto ne enfatizza le caratteristiche peculiari. Permette di realizzare sistemi di miglio-

re qualità, però ciò è pur sempre realizzato attraverso altro codice (alla fine le varie asserzioni vengono rese attraverso una programmazione, magari in OCL) e come tale, è soggetto a sua volta a errori.

Classi ben disegnate

Il presente capitolo non poteva che terminare cercando di rispondere a uno degli interrogativi sui quali più frequentemente si arrovella la mente dei neofiti del mondo Object Oriented: "Quando una classe può essere definita ben disegnata?". Si tratta di una bella domanda per la quale, caso raro, è possibile formulare una buona risposta, tenendo presente sia quanto riportato nei paragrafi precedenti, sia gli insegnamenti dei maestri dell'informatica, con particolare riferimento al solito Booch ([BIB13]). Tipicamente, la precedente domanda tende a evocarne un'altra decisamente più insidiosa: "Quando un modello di disegno si può considerare ben progettato?", alla quale si tenterà di dare una risposta nel capitolo ottavo.

Tornando al primo quesito, sicuramente una classe ben disegnata deve rispettare i principi della massima coesione e del minimo accoppiamento. Quindi, riassumendo quanto riportato nei precedenti paragrafi, i relativi elementi devono essere molto correlati tra loro, la classe deve possedere il minor numero possibile di dipendenze con le altre classi, e queste devono essere le più deboli realizzabili.

Una classe deve essere in grado di rappresentare correttamente la relativa astrazione e soprattutto deve strutturarne sufficienti caratteristiche al fine di permettere un'interazione valida ed efficiente con le classi client. Nel caso in cui tale proprietà non sia rispettata, la classe stessa potrebbe considerarsi, per ovvi motivi, inutilizzabile. Questa caratteristica è stata denominata da Booch *sufficiency* (sufficienza). Violazioni della proprietà di sufficienza sono facilmente riscontrabili: durante la progettazione delle dinamiche interne del sistema: ci si accorge che in una o più iterazioni l'interfaccia di una o più classi non prevede i servizi necessari. Ciò comporta l'impossibilità di dar luogo all'iterazione desiderata o di ottenerla comunque attraverso un percorso tortuoso che coinvolge molti più oggetti del necessario. Come è possibile individuare la classe a cui appartiene l'operazione mancante? Il metodo mancante dovrebbe essere membro della classe in cui sono presenti gli elementi che presentano una forte correlazione con il metodo stesso. In altre parole, la classe prescelta, dopo l'inserimento del metodo, deve continuare a mantenere un elevato grado di coesione e un minimo accoppiamento (ciò perché il metodo potrebbe delegare parte delle proprie responsabilità ad altri metodi presenti in diverse classi).

Per esempio, una classe la cui responsabilità sia memorizzare un elenco di oggetti individuabili attraverso un identificatore univoco (una sorta di *hashtable* specializzata), presenterebbe un'interfaccia non sufficiente, qualora non preveda un metodo atto verificare la presenza di uno specifico elemento: un risultato `null` di un metodo di `Object : getElement(key : Object)`, non permetterebbe di discernere il caso in cui la

chiave non sia stata inserita da quello in cui sia stata volutamente accoppiata con un valore `null`.

La sufficienza è una caratteristica, per così dire, necessaria ma non sufficiente. In effetti, si vuole che l'interfaccia della classe sia anche completa, ossia l'interfaccia catturi tutte le caratteristiche significative della relativa astrazione. Questa caratteristica è tipicamente indicata con il nome di *completeness* (completezza). Quindi, mentre la proprietà di sufficienza comporta un'interfaccia minima, quella di completezza implica un'interfaccia che copra tutti gli aspetti dell'astrazione. Una classe è completa quando la relativa interfaccia è così generale da poter essere utilizzata da ogni classe cliente. In questo caso si ha a che fare con un criterio per molti versi soggettivo. Ancora una volta però, una violazione del principio di completezza, nel contesto di uno specifico disegno, è facilmente ravvisabile: è possibile individuare una classe cliente che abbia bisogno di un determinato servizio da un'altra fornitore non presente nella relativa interfaccia e non ottenibile attraverso la combinazione dell'invocazione di diversi metodi. Quest'ultimo concetto è molto importante e si collega al principio della *primitiveness* (primitività). Una classe oltremodo ingegnerizzata di certo non può definirsi ben disegnata e quindi, a meno di motivazioni importanti, si vuole evitare che nell'interfaccia di una classe siano presenti metodi non primitivi, ossia ottenibili dall'invocazione di altri. In generale, una classe possiede la caratteristica di primitività qualora non siano esposti nell'interfaccia metodi che possano essere ottenuti dall'esecuzione combinata di altri. Sebbene, in prima analisi, il concetto di primitività possa sembrare in antitesi a quello di completezza, riflettendo bene ciò è vero molto parzialmente. La primitività non richiede di omettere nell'interfaccia metodi non utilizzati, bensì di "valutare attentamente" l'opportunità di inserire metodi in grado di fornire servizi ottenibili attraverso l'invocazione di più metodi primitivi. La primitività non sempre è una caratteristica da applicare alla lettera. Per esempio, in diverse classi tipo vettore, collezione, ecc., sono previsti dei metodi che invece di aggiungere un singolo elemento aggiungono delle liste. Volendo essere puntigliosi, si potrebbe affermare che metodi di questo tipo invalidano la proprietà di primitività della classe. I vantaggi generati sono però tali da rendere del tutto accettabile una riduzione del livello di tale proprietà.

Altra caratteristica indispensabile che deve possedere una classe è l'incapsulamento, pertanto la classe non deve esporre all'esterno i propri dettagli implementativi ed, eventualmente, neanche il proprio stato (salvo che non rappresenti un'astrazione deputata a tale fine).

Oltre alle caratteristiche formali che deve possedere una classe per considerarsi ben disegnata, esistono una serie di segnali che possono aiutare a capire situazioni anomale. Uno dei più semplici si ha qualora non si riesca ad attribuire un "bel" nome significativo a una classe. Ciò potrebbe essere il risultato di uno scarso livello di coesione, magari perché si sono inglobate più classi in una sola, oppure di un elevato accoppiamento tra due o più classi: si è scissa un'entità in più classi, e così via. Un altro segnale potrebbe

venire da classi con troppe responsabilità o, se si gradisce, di dimensioni elevate. Ciò potrebbe essere frutto nuovamente di una classe con scarso livello di coesione.

Ancora, una classe con un'interfaccia particolarmente prolissa potrebbe essere risultato o di uno scarso livello di coesione, oppure di un processo di over-ingegnerizzazione (violazione ingiustificata della proprietà di primitività).

Qualora si abbiano le idee confuse circa il buon disegno di una classe, la prova del nove potrebbe essere fornita dalla progettazione formale del relativo tipo di dato astratto. Da tenere presente che tale progettazione non è sempre indolore: per classi complesse tende a innescarne una serie preliminare atta a progettare gli ADT relativi alle classi da cui dipende quella oggetto di studio. In ogni modo, qualora sia possibile dar luogo ad un corretto ADT relativo alla classe, si può tranquillamente asserire che la stessa sia ben disegnata; negli altri casi evidentemente è presente qualche anomalia.

Ricapitolando, le proprietà di una classe ben disegnata ([BIB13]) sono:

- massima coesione;
- minimo accoppiamento;
- sufficienza;
- completezza;
- primitività.

Ricapitolando…

Nel presente capitolo sono state introdotte brevemente le nozioni basilari dell'Object Oriented. L'intento dell'autore è stato presentare le nozioni utilizzate nei capitoli successivi, senza avere la presunzione di trattare in maniera approfondita un argomento così importante e vasto, per il quale si rimanda a testi più specifici.

Il modo migliore per iniziare è indubbiamente la definizione di classi e oggetti enunciata da Booch: "un oggetto rappresenta un articolo (item), unità o entità individuale, identificabile, reale o astratta che sia, con un ruolo ben definito nel dominio del problema e con un confine ben stabilito".

Gli oggetti presenti in ogni sistema, molto genericamente, possono essere suddivisi in due categorie: "cose" che "esistono" (più o meno tangibilmente) nell'area di business oggetto di analisi, e *oggetti* che invece vivono nel modello di disegno e vengono introdotti per realizzare l'infrastruttura informatica, come per esempio `Vector`, `File`, `IOStream`, e così via. Per quanto concerne gli oggetti appartenenti alla prima tipologia, si può dire che tali entità sono un qualcosa che esiste (o "traspira") nel mondo concettuale e, come tali, se ne può parlare, eventualmente li si può toccare o manipolare in qualche modo. "Ma cosa sono le *cose* del dominio del problema? Molte di queste cose probabilmente apparten-

gono a una delle seguenti cinque categorie: oggetti tangibili, ruoli, episodi, interazioni e specificazioni" [Booch]. Bruce Eckel, nel libro *Thinking in Java*, afferma che esiste una connessione stretta tra gli oggetti e i calcolatori: "ogni oggetto assomiglia, in qualche modo, a un piccolo computer; possiede degli stati e delle operazioni che vi si possono invocare". Un oggetto può anche essere considerato come un tool che permette di impacchettare insieme strutture dati e funzionalità per concetti, in modo da far loro rappresentare appropriatamente un'idea appartenente allo spazio del problema piuttosto che essere forzata a utilizzare un idioma del calcolatore sottostante.

Tutti gli oggetti sono "istanze" di classi, ove per classe si intende un qualcosa che consente di descrivere formalmente proprietà e comportamento di tutta una categoria di oggetti simili. L'obiettivo è creare una corrispondenza biunivoca tra gli elementi del dominio del problema (oggetti che realmente esistono) e quelli dello spazio delle soluzioni. La difficoltà consiste nel riuscire a descrivere formalmente, precisamente e completamente un'astrazione a partire dagli esempi delle relative istanze (corpo dell'astrazione).

Si può pensare al rapporto che intercorre tra un oggetto e la relativa classe come a quello esistente tra una variabile e il relativo tipo, in un qualsivoglia linguaggio di programmazione. Sempre secondo Booch, "un oggetto possiede stato, comportamento e identità; la struttura e il comportamento di oggetti simili sono definiti nella loro classe comune; i termini di istanza e oggetto sono intercambiabili".

Lo stato di un oggetto è un concetto dinamico e, in un preciso istante di tempo, è dato dal valore di tutti i suoi attributi e dalle relazioni instaurate con altri oggetti. Si tratta di un concetto molto importante poiché influenza il comportamento futuro dell'oggetto. Tipicamente, sottoponendo opportuni stimoli a un oggetto (invocazione dei metodi), questo tende a reagire, nella maggior parte dei casi, in funzione del suo stato interno.

Il comportamento di un oggetto è costituito dalle inerenti attività (operazioni) visibili e verificabili dall'esterno. Lo scambio di messaggi tra oggetti (invocazione di metodi), generalmente, varia lo stato dell'oggetto stesso. "Il comportamento stabilisce come un oggetto agisce e reagisce, in termini di cambiamento del proprio stato e del transito dei messaggi" [Booch].

Da quanto emerso, è evidente che la relazione esistente tra lo stato di un oggetto e il comportamento è di mutua dipendenza: è possibile considerare "lo stato di un oggetto come l'accumulazione dei risultati prodotti dal relativo comportamento", il quale, a sua volta, dipende dallo stato in cui si trova l'oggetto.

L'identità di un oggetto è la caratteristica che lo contraddistingue da tutti gli altri. Spesso ciò è dato da un valore univoco. Per esempio, un oggetto "conto corrente" è identificato dal relativo codice, una persona cittadina italiana dal codice fiscale, e così via.

Ogni oggetto è dotato di una propria interfaccia (si faccia attenzione a non confondersi con il concetto di interfaccia UML/Java) scaturita dall'elenco dei metodi esposti agli altri oggetti corredati dalla relativa firma. Quindi le richieste che un oggetto può soddisfare sono specificate dall'interfaccia della classe di cui è istanza. Questa, anche se implicita, rappresenta un contratto stipulato tra gli oggetti fornitori e quelli utilizzatori di servizi. Nell'interfaccia sono condensate tutte le assunzioni che gli oggetti client fanno circa quelli di cui utilizzano i servizi (server). L'interfaccia di un oggetto viene anche definita *protocollo*, poiché stabilisce i servizi offerti e la "sintassi" (firma dei vari metodi) con cui utilizzarli.

Al concetto di interfaccia implicita si aggiunge quello di interfaccia esplicita (quella a cui si è portati comunemente a pensare). In UML un'interfaccia è definita come un insieme di operazioni, identificato

da un nome, che caratterizza il comportamento di un elemento. Si tratta di un meccanismo che rende possibile dichiarare esplicitamente, in appositi costrutti esterni, le operazioni implementate dalle classi. L'attenzione è quindi focalizzata sulla struttura del servizio esposto e non sull'effettiva realizzazione. Le interfacce non possiedono implementazione, attributi o stati; dispongono unicamente della dichiarazione di operazioni (definita firma) e possono essere connesse tra loro tramite relazioni di generalizzazione. Visibilità private dei relativi metodi avrebbero ben poco significato e quindi non sono ammesse.

Il meccanismo delle interfacce rende possibile tutto un insieme di meccanismi, come l'aggiunta quasi indolore di nuove funzionalità, la rimozione di altre, la sostituzione di componenti con altri più moderni (magari resi più efficienti, maggiormente rispondenti alle nuove richieste dei clienti) ecc. Gli oggetti che interagiscono con altri attraverso interfacce vedono questi ultimi solo attraverso quanto dichiarato nella relativa interfaccia senza necessità di possedere conoscenza diretta delle classi che implementano l'interfaccia. Queste ultime quindi si possono considerare come punti di *plug-in* nel disegno, in cui è possibile cambiare funzionalità, aggiornare la scheda, aggiungere altri servizi ecc..

Nella progettazione di sistemi Object Based, un'attitudine particolarmente richiesta è quella dell'astrazione. Si tratta di una proprietà tipica del cervello umano che permette di far fronte alle limitazioni nell'affrontare la complessità dei sistemi. In generale l'atto dell'astrarre consiste nell'individuare similitudini condivise tra oggetti, processi ed eventi appartenenti alla vita reale e nella capacità di concentrarsi su queste, tralasciando momentaneamente le differenze. Chiaramente, molto importante è anche il concetto di livello di astrazione che permette di enfatizzare alcuni aspetti del sistema e di trascurarne altri.

Ogni qualvolta si parla di disegno o programmazione Object Oriented, le parole magiche che immediatamente vengono alla mente sono: ereditarietà, incapsulamento e polimorfismo, ossia le leggi fondamentali del disegno orientato agli oggetti.

L'ereditarietà è indubbiamente la legge più nota del Object Oriented. Si tratta di un meccanismo attraverso il quale un'entità più specifica incorpora struttura e comportamento definiti in entità più generali. Tipicamente, quest'ultimi sono detti genitori, mentre gli elementi ereditanti sono detti figli. Questi, ovviamente, sono completamente consistenti con quelli più generali da cui ereditano (ne possiedono tutte le proprietà, i membri e le relazioni) e in più possono specificare struttura e comportamento aggiuntivi.

Da un punto di vista intuitivo, è possibile pensare all'ereditarietà come a un meccanismo in grado di prendere un elemento di partenza, clonarlo e di aggiungervi ulteriore comportamento. Un elemento definito per mezzo della relazione di generalizzazione è, a tutti gli effetti, un nuovo tipo che eredita dal genitore tutto ciò che è dichiarato come tale. L'ereditarietà è una tecnica molto potente e i vantaggi apportati sono la semplificazione della modellazione di sistemi reali, la riusabilità del codice, nonché il polimorfismo. Uno dei principi più importanti dell'ereditarietà afferma che "un'istanza di una classe discendente può sempre essere utilizzata in ogni posto ove è prevista un'istanza di una classe antenata" (principio della sostituibilità di Liskov).

Attraverso l'ereditarietà è possibile rappresentare formalmente i risultati dei processi di classificazione, ossia del processo mentale che permette di organizzare la conoscenza. Nel mondo Object Oriented ciò si traduce raggruppando, in un'opportuna organizzazione gerarchica, le caratteristiche comuni di specifici oggetti. Ciò permette di dar luogo a modelli più piccoli e quindi più semplici da comprendere,

sebbene, qualora non utilizzata propriamente, possa generare diverse anomalie. Come tutti gli strumenti, anche l'ereditarietà ha un proprio dominio di applicazione che, se non rispettato, non solo non fornisce la formulazione di valide soluzioni, ma addirittura può generare gravi anomalie. Molto spesso disegnatori junior, non appena "fiutano" l'eventualità di un minimo comportamento condiviso (magari un paio di attributi e/o metodi), non si lasciano sfuggire l'occasione e danno luogo a generalizzazioni abbastanza stravaganti. Sebbene l'ereditarietà sia uno dei principi fondamentali dell'Object Oriented e offra moltissimi vantaggi se utilizzata appropriatamente, ciò non significa che si tratta dell'acqua santa. L'idea alla base della generalizzazione è il riutilizzo del tipo e non del codice. Il "limite" della generalizzazione è che una volta realizzata non può più variare: si tratta di una versione di soluzioni *hard-coded*. I problemi, come sempre, nascono quando si utilizza la generalizzazione in maniera forzata. L'esempio classico è relativo alle persone che possono essere imbarcate in un volo. Le tipologie sono, essenzialmente tre: piloti, assistenti di volo e passeggeri. In prima analisi potrebbe sembrare una classica situazione di ereditarietà salvo poi accorgersi che sia i piloti, sia il personale viaggiante possono — spesso e volentieri viste le tariffe agevolate — recitare il ruolo di passeggeri. Allora cosa fare? Le opzioni sono:

a. ignorare il problema e dar luogo a copie di oggetti ogni qualvolta sia necessario: ciò comporta che ogni qualvolta sia necessario aggiornare i dati di un oggetto bisogna inventarsi un meccanismo impossibile per trovare eventuali copie;

b. dare luogo a ulteriori specializzazioni: "pilota-passeggero" e "personale di volo-passeggero", creando classificazioni assurde ad enorme ridondanza di codice;

c. sostituire l'ereditarietà con un'aggregazione, in modo da avere un nocciolo di comportamento e struttura comune a tutti gli oggetti e una tipologia variabile nel tempo.

Questo problema è noto con il nome di "anomalie dei ruoli".

Nell'Object Oriented non esiste unicamente il caso in cui una classe erediti da un solo genitore, bensì è possibile che una stessa classe erediti da diversi genitori: ereditarietà multipla. Questa tecnica, in diversi linguaggi (come per esempio Java) non è direttamente implementabile, pertanto è consigliabile non utilizzarla nel modello di disegno che dovrebbe essere quanto più possibile vicino al codice, mentre non esistono particolari controindicazioni, qualora ricorrano le condizioni, nell'utilizzarla nei modelli precedenti il cui scopo principale è ancora descrivere l'area business oggetto di studio.

L'incapsulamento è il meccanismo che rende possibile il famoso principio dell' *information hiding* (nascondere le informazioni) che, come suggerito dal nome, consiste nel nascondere i dettagli della struttura interna di una classe al resto del sistema. Il principio fondamentale è che nessuna parte di un sistema complesso debba dipendere dai dettagli interni di un'altra. A tal fine è necessario creare uno strato di separazione tra gli oggetti clienti e quelli fornitore, ottenuto separando l'interfaccia propria di un oggetto dalla sua implementazione interna.

In generale l'incapsulamento viene suddiviso in due grandi categorie: quello standard che prevede che le classi non abbiano alcuna conoscenza della struttura interna delle altre, e in particolare di quelle

di cui possiedano un riferimento, con la sola eccezione della firma dei metodi esposti nella relativa interfaccia. L'incapsulamento totale si ha quando ogni classe non dispone assolutamente di alcuna conoscenza delle altre, non solo per ciò che concerne il comportamento e la struttura interna, ma, neanche in termini di esistenza.

Da quanto riportato appare evidente che i princìpi dell'incapsulamento e dell'ereditarietà, per molti versi, presentano diversi punti di discordanza. Il nascondere il più possibile l'organizzazione della struttura degli oggetti, di fatto, limita o, addirittura inibisce l'ereditarietà.

Secondo un approccio purista Object Oriented, il modo con cui le informazioni sono rappresentate all'interno dell'oggetto dovrebbe essere del tutto irrilevante. Contrariamente a molte convinzioni comuni, ciò non significa semplicemente che tutti gli attributi dell'oggetto debbano avere una visibilità privata ed essere esposti per mezzo di opportuni metodi `get` e `set`. Questi metodi `get`, molte volte, non fanno altro che continuare a esporre i relativi attributi membro attraverso il valore restituito. Ciò implica che, se per qualche motivo varia la rappresentazione interna di un attributo membro di un oggetto, è necessario individuare tutti gli oggetti clienti e variarne conseguentemente il codice.

Polimorfismo deriva dalle parole greche *polys* (= molto) e *morphé* (=forma): significa quindi "molte forme". Si tratta di una caratteristica fondamentale dell'Object Oriented, relativa alla capacità di supportare operazioni con la medesima firma site in classi diverse con comportamenti diversi (derivanti però da una stessa antenata). La possibilità di poter attuare il polimorfismo, richiede, propedeuticamente, la facoltà di conoscere a priori una porzione dell'interfaccia di un insieme di classi, o se si preferisce, di poter raggruppare un insieme di classi attraverso segmenti di interfaccia condivisa. Ciò, naturalmente, si ottiene attraverso l'utilizzo dell'ereditarietà: tutte le classi discendenti ereditano l'interfaccia di quella antenata e quindi è possibile trattare i relativi oggetti come se fossero istanze di uno stesso tipo (la classe antenata). L'utilizzo del meccanismo delle interfacce (nel senso di costrutto `public interface`), rende anch'esso possibile trattare in maniera astratta un opportuno insieme di classi (quelle che implementano l'interfaccia).

In un'organizzazione gerarchica ottenuta per mezzo dell'ereditarietà, si effettua un *upcasting* ogniqualvolta un oggetto di una classe discendente viene trattato (*casted*) come se fosse un'istanza della classe progenitrice, mentre con il termine di *downcasting* si intende l'operazione opposta.

Il termine *overriding* è intimamente legato al polimorfismo: quando in una classe discendente si ridefinisce l'implementazione di un metodo, in gergo si dice che se ne è effettuato l'*overriding*. In sostanza si crea una nuova definizione della funzione polimorfica nella classe discendente.

Il termine *overloading* ha a che fare con la definizione di diversi metodi con lo stesso nome, ma diversa firma.

Due principi fondamentali della Computer Science sono quelli noti con i termini di *massima coesione* e *minimo accoppiamento*. Per quanto concerne la coesione, Booch afferma che: "un modulo presenta un'elevata coesione quando tutti i componenti collaborano fra loro per fornire un ben preciso comportamento". Il concetto di coesione può essere applicato a diversi livelli di dettaglio (metodi, attributi, classi, package, componenti, ecc.): per esempio, un metodo presenta un elevato grado di coesione quando svolge una sola funzione ben definita. Al livello di classe la "coesione è la misura della correlatività delle proprietà strutturali (attributi e relazioni con le altre classi) e comportamentali di una classe (meto-

di)". Chiaramente si desidera avere un valore di coesione che sia il più elevato possibile (i vari attributi e metodi sono fortemente correlati tra loro).

Disegni i cui elementi presentano scarsi livelli di coesione possono generare una serie di anomalie. Per esempio rappresentano un problema le classi "mastodontiche" dotate di un numero eccessivo di attributi, relazioni con altre classi e/o metodi. Ciò non è auspicabile per una serie di motivi: si complica la comprensione del codice e quindi la relativa manutenzione, diviene laborioso eseguire il test, il riutilizzo della classe diviene complesso, si rende difficile isolare elementi soggetti a variazioni, ecc. Questo problema è tipicamente generato quando in un'unica classe ne vengono inglobate più d'una.

I vantaggi generati da "componenti" software a elevata coesione sono:

- aumento del grado di *riusabilità*. Disporre di componenti con un insieme ben definito e circoscritto di responsabilità, evidentemente ne aumenta la probabilità di riutilizzo;

- *robustezza*. Un componente con responsabilità ben definite è più facile da comprendere, da verificare e quindi l'intero sistema risulta più robusto.

La proprietà di minimo accoppiamento è definita come la "misura della dipendenza tra componenti software (classi, packages, componenti veri e propri, ecc.) di cui è composto il sistema". Si ha una dipendenza tra due elementi, per esempio classi, quando un elemento (client) per espletare le proprie responsabilità ha bisogno di accedere alle proprietà comportamentali (metodi) e/o strutturali (attributi) dell'altro (server). Chiaramente quest'ultimo non dipende dai client, mentre è vero il contrario. Ciò comporta che un cambiamento all'elemento che fornisce i servizi genera la necessità di revisionare ed eventualmente aggiornare degli elementi client. In sintesi, la dipendenza di un componente da un altro implica che il funzionamento del componente stesso dipende dal corretto funzionamento di altri componenti.

Un accoppiamento elevato non è desiderabile per una serie di motivi, tra i quali i più importanti sono:

- la variazione di un componente genera a cascata la necessità di verificare ed eventualmente aggiornare i componenti dipendenti;

- qualora si volesse riutilizzare uno specifico componente è necessario riutilizzare (o comunque portarsi dietro) tutti i componenti dipendenti; ecc.

L'obiettivo da perseguire è minimizzare il grado di accoppiamento. Chiaramente eliminarlo non avrebbe molto senso: si potrebbe correre il rischio di generare la situazione opposta: classi mastodontiche che non sono accoppiate perché fanno tutto da sole e quindi si genererebbe un problema di minima coesione…

Spesso nella progettazione di sistemi Object Oriented complessi si "sorvola" qualora le classi presenti all'interno di un package non presentino esattamente un accoppiamento minimo: ciò su cui però bisogna porre molta attenzione è che l'accoppiamento tra package sia veramente ridotto al minimo indispensabile.

L'*Abstract Data Type* (tipo di dato astratto) è un linguaggio "matematico" che permette di specificare collezioni di entità atte a manipolare informazioni attraverso operazioni di generazione, distruzione,

accesso, modifica delle relative istanze. Poiché queste collezioni sono definite attraverso un linguaggio ad alto livello di tipo matematico, ne segue che si è interessati più alla specifica dell'interfaccia che al metodo con cui l'ADT viene implementato e di come ne vengono trattati i dati.

Le proprietà che caratterizzano un ADT sono: il tipo che esportano, l'interfaccia (ossia le operazioni esposte che rappresentano l'unico meccanismo utilizzabile per accedere ai dati), gli assiomi e condizioni dettate dallo spazio del problema.

Le parti costituenti un ADT sono, essenzialmente, due: quella relativa ai dati, in cui è riportata la descrizione della struttura utilizzata dal particolare ADT e quella relativa ai servizi forniti. Per quanto concerne la prima, potrebbe essere omessa (espressione massima di incapsulamento) se non fosse per necessità relative alla definizione formale dei metodi. Anche per ciò che concerne l'altra sezione, quella dedicata alle operazioni (l'interfaccia), è possibile selezionare diversi livelli di formalità, sebbene sia opportuno tentare cercare di essere più rigorosi possibili. Una descrizione Object Oriented dei metodi dovrebbe prevedere la dichiarazione esplicita degli argomenti dei metodi e delle relative condizioni (pre-, post- e durante l'utilizzo). Le operazioni si possono suddividere nelle categorie costruttori, distruttori, selettori e modificatori.

Una caratteristica fondamentale importanza per ogni sistema è rappresentata dall'affidabilità. Nell'ambito dei sistemi software, con tale termine si includono due altri concetti: *correttezza* (il sistema realizza le funzioni per il quale è stato progettato) e *robustezza* (il sistema è in grado di rilevare e gestire situazione anomale). Ciò in pratica si traduce nel realizzare sistemi senza errori.

La tecnica del *Design by Contract* (disegno per contratto), ideata da Bertrand Mayer nei laboratori del linguaggio Eiffel, grazie alla metodicità e formalità, rappresenta un ottimo strumento per realizzare sistemi (più) affidabili.

La tecnica, come suggerito dal nome è basata sul concetto di contratto che stabilisce precise obbligazioni tra oggetti client e quelli fornitore. Da una parte, ogni oggetto cliente si fa carico di rispettare le condizioni pattuite nel richiedere i servizi, mentre dall'altra, i server assicurano il rispetto di determinate condizioni relative la fornitura dei servizi. Qualora un cliente non rispetti le condizioni pattuite, ovviamente il contratto perde di validità. Un contratto è quindi vantaggioso e protegge entrambi i contraenti sancendo, da una parte, la natura e qualità del servizio (*postcondizioni*) e, dall'altra, sollevando il fornitore (*precondizioni*) da ogni imputabilità qualora il cliente richieda la fornitura del servizio non rispettando gli accordi.

Il contratto, in ultima analisi, governa le relazioni tra ogni metodo di un oggetto e i potenziali clienti. Pertanto definisce le assicurazioni che ogni parte deve garantire per una corretta fruizione del servizio (invocazione del metodo) e il risultato pattuito.

Oltre alle pre- e postcondizioni, esiste un'altra serie di asserzioni il cui dominio invece è la classe. Tali condizioni sono dette invarianti poiché la relativa valutazione deve essere vera per tutte le istanze della specifica classe detta invariante.

Quindi, le condizioni che costituiscono un contratto sono:

- **precondizioni**: vincoli che gli oggetti client devono rispettare per fruire di un servizio;
- **postcondizioni**: garanzie assicurate dal fornitore del servizio, qualora le precondizioni siano rispettate;

- *invarianti*: condizioni sempre valide.

Un aspetto di particolare importanza è dato dalla combinazione della metodologia del Design by Contract con l'ereditarietà. In altre parole è necessario esaminare come un contratto pattuito da una classe antenata venga ereditato da una classe discendente. La soluzione a questo problema è data dal concetto del *sottocontratto*, il quale stabilisce una serie di norme dovute essenzialmente al principio di sostituibilità, quindi la classe estendente deve mantenere le stesse precondizioni definite in quella antenata oppure definirne altre più deboli e mantenere le stesse postcondizioni definite in quella antenata oppure definirne altre più forti.

I vantaggi della metodologia del Design by Contract sono legati alla sua sistematicità che agevola per la realizzazione di sistemi senza errori, alla realizzazione di un framework per il debugging e il test delle diverse parti del sistema, all'aumento della qualità della documentazione, all'agevolazione della rilevazione e gestione di condizioni anomale, ecc.

Il presente capitolo non poteva che terminare rispondendo ad una delle domande che più frequentemente sono poste dai neofiti del mondo Object Oriented: "Quando una classe può essere definita ben disegnata?". Alla domanda è possibile rispondere, tenendo presente sia quanto riportato nei paragrafi precedenti, sia gli insegnamenti dei maestri dell'informatica.

In particolare, una classe è ben disegnata quando rispetta le caratteristiche di:

- massima coesione (tutti i relativi elementi presentano un elevato grado di correlazione);
- minimo accoppiamento (la classe possiede il minor numero possibile di relazioni con le altre classi);
- sufficienza (rappresenta correttamente la relativa astrazione e soprattutto modella sufficienti caratteristiche che permettono un'interazione valida ed efficiente con le classi client);
- completezza (l'interfaccia della classe cattura tutte le caratteristiche significative della relativa astrazione);
- primitività (non ci sono metodi esposti nell'interfaccia che possono essere ottenuti dalla combinazione dell'esecuzione di altri).

RISPOSTA ALLA DOMANDA

Il metodo `add(index : int, element : Object) : boolean`, presente nell'interfaccia `List`, è un esempio di overloading. Ebbene sì. In effetti, poiché l'interfaccia `List` estende `Collection`, automaticamente eredita il metodo `add(element : Object) : boolean`, e quindi il metodo in questione ne rappresenta una versione. Ragion per cui si tratta di un meccanismo di overloading.