Capitolo 7

Gli oggetti: una questione di classe

Lo UML diventerà il nuovo linguaggio di programmazione.
Non esistono barriere tecniche, solo politiche.
IVAR JACOBSON

Le episodiche volte in cui un mio modello viene realizzato nella sua interezza senza modifiche, provo quasi un senso di commosso smarrimento.
LVT

Introduzione

Obiettivo del presente capitolo è illustrare il formalismo dei diagrammi delle classi che, verosimilmente, rappresenta uno dei più affascinanti e noti tra quelli definiti dallo UML. Non si tratta di una notazione completamente inedita, come del resto avviene anche per gli altri formalismi grafici, ma le sue origini possono essere individuate nei corrispondenti diagrammi dalla metodologia di Booch, che a loro volta discendono dai famosi diagrammi "entità–relazione" (*entity–relationship diagram*).

Benché non si tratti di un formalismo eccessivamente complesso in quanto prevede un numero piuttosto contenuto di elementi e regole, realizzare validi modelli orientati agli oggetti è un'attività tutt'altro che banale. Con l'ampliamento dell'esperienza si riesce a comprende sempre più intimamente quanto sia complesso realizzare modelli validi; basti considerare la fase di disegno, una delle più complesse e critiche dell'intero ciclo di vita del software: è necessario prendere un numero di decisioni verosimilmente superiore a ogni altra fase dell'intero processo. La costruzione dei modelli a oggetti, oltre a richiedere

tutti i talenti necessari per la realizzazione di un qualsivoglia modello (capacità di astrarre, di rappresentare rigorosamente concetti presenti nel mondo reale, e così via), esige ulteriori competenze, sia per l'elevato grado di formalità richiesto (i diagrammi delle classi sono l'espressione ultima delle leggi sancite nei "testi sacri" del paradigma OO), sia per via delle problematiche cui bisogna far fronte nella progettazione di un sistema(capacità di "prevedere" le evoluzioni future del sistema, di riuscire a circoscrivere le aree più vulnerabili, di rendere l'architettura flessibile, scalabile ed efficace, di saper coniugare i requisiti relativi alle prestazioni con altri non funzionali, quali la sicurezza per esempio, e così via). Queste capacità si conseguono costruendo sistemi informatici che funzionano.

Capacità di astrazione sono principalmente richieste nel disegno dei primi modelli a oggetti (dominio e business), in cui l'obiettivo primario è descrivere formalmente l'area business argomento di studio, mentre quelle più tecniche diventano indispensabili nella realizzazione di modelli di analisi e disegno in cui l'obiettivo primario diventa costruire il sistema.

La stessa nomenclatura in vigore non sempre aiuta e, in alcuni casi, concorre ad aumentare il livello di caos. Per esempio, in molte occasioni si utilizzano i concetti di oggetti e classi come sinonimi, altre volte i diagrammi delle classi sono indicati con il nome di "modelli a oggetti" (per esempio "modello a oggetti del dominio"), e via dicendo. Sebbene ciò risulti del tutto naturale per il personale esperto, in genere lo è di meno per quello coinvolto nello sviluppo di sistemi con non molta esperienza in materia. Lo stesso nome "diagramma delle classi", quantunque ormai di uso comunissimo, probabilmente non è il migliore cui si potesse pensare. Un nome più appropriato, come dichiarato anche nel documento ufficiale delle specifiche, potrebbe essere "diagramma della struttura statica".

> Gran parte dei diagrammi delle classi illustrati nel presente capitolo sono esempi di modelli a oggetti del dominio (*domain object model*). Le motivazioni vanno ricercate sia nella semplicità di comprensione dei concetti esposti (trattandosi della descrizione del dominio, nulla è dato per scontato), sia nell'elevato livello di astrazione rispetto a dettagli tecnici. Il modello a oggetti del dominio e quello business vengono realizzati durante le prime iterazioni del processo di sviluppo del software. L'obiettivo è realizzare un modello che permetta di descrivere formalmente gli oggetti che esistono nell'area business oggetto di studio e che il sistema, in qualche misura, dovrà implementare (su questo argomento si tornerà più approfonditamente nel capitolo successivo). Logica conseguenza è che dovrebbero essere rappresentate esclusivamente classi a carattere *business*, mentre quelle più strettamente tecniche dovrebbero comparire in modelli successivi (analisi e disegno). Qualora la corrispondenza tra entità dell'area business e classi presenti nel modello del dominio non sia del tutto immediata, è comunque possibile individuare regole, eventi, e così via, vigenti nel dominio del problema, che hanno dato origine ad essa. Per queste sue caratteristiche peculiari, il modello a oggetti del dominio concorre a diminuire il gap

esistente tra lo spazio del problema e quello delle soluzioni; in altre parole supporta la descrizione formale del vocabolario in vigore nell'area business che il sistema dovrà realizzare. Ciò dovrebbe permettere di avviare l'edificazione del ponte comunicativo tra personale tecnico e quello esperto dell'area business oggetto di studio. Le predette caratteristiche, rendono il modello a oggetti del dominio uno strumento formidabile per la produzione di esempi: la sua comprensione non richiede conoscenze specifiche di un linguaggio o di una tecnologia particolare, e tanto meno è necessario conoscere approfonditamente il dominio oggetto di studio, giacché è descritto dal modello stesso. Nonostante questi vantaggi, bisogna ricordare che, in ultima analisi, i sistemi devono pur essere costruiti (ebbene capitano anche queste iatture…) e il modello in cui il formalismo delle classi raggiunge la sua massima espressione è sicuramente quello di disegno.

Diagrammi delle classi

I diagrammi delle classi mostrano la struttura statica di un modello i cui elementi sono classi, interfacce, relazioni ecc. e il cui significato varia a seconda della fase del processo di sviluppo a cui appartiene il modello. Tecnicamente un diagramma delle classi è definito come un grafo di elementi (i famosi *classificatori*) connessi attraverso opportune relazioni. Pertanto vi albergano elementi come classi, interfacce, package, tipi di dati, e così via. Si tratta di elementi che nel metamodello UML specializzano la metaclasse astratta `Classifier`. Anche per questo motivo, probabilmente, un nome più appropriato per i diagrammi delle classi dovrebbe essere "diagramma della struttura statica".

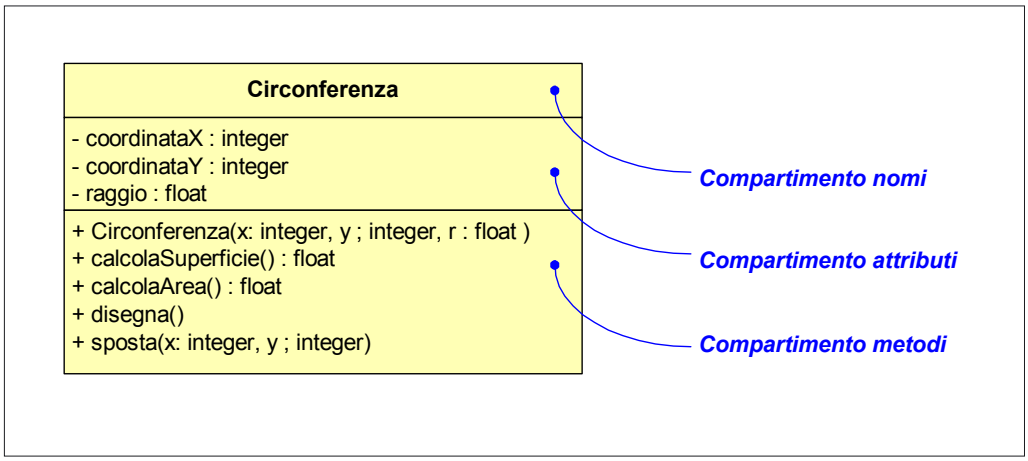
Molto importante è sottolineare come questi diagrammi mostrino unicamente aspetti statici del modello. Con sommo dispiacere di alcuni tecnici, non permettono di rappresentare comportamenti temporali. Questi aspetti dinamici si prestano a essere modellati attraverso altre notazioni, come i diagrammi di interazione, degli stati e delle attività (*cfr* Capitoli 9 e 10).

Classi in UML

Nell'ambito della progettazione di sistemi OO, con particolare riferimento ai diagrammi di classe, non ci si dovrebbe stupire del fatto che concetti come interfacce, classi e relative relazioni rappresentino gli elementi fondamentali. In particolare, le classi e gli oggetti permettono di descrivere cosa c'è all'interno di sistemi OO o component based, mentre le relazioni tra i vari oggetti consentono di evidenziarne l'organizzazione.

Una classe è una descrizione di un insieme di oggetti che condividono struttura (attributi), comportamento (operazioni) e relazioni. La rappresentazione grafica prevede un rettangolo, tipicamente suddiviso in tre sezioni dedicate rispettivamente al **nome**, agli

Figura 7.1 — *Rappresentazione grafica di una classe.*

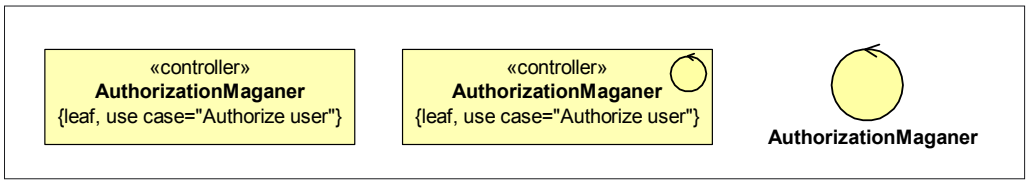


attributi e alle **operazioni**. Di queste, solo la prima è obbligatoria. Da tener presente che l'omissione di un compartimento non permette di effettuare alcuna deduzione. In altre parole se la sezione degli attributi non viene visualizzata, ciò non significa assolutamente che la classe ne sia sprovvista. Tale condizione è rappresentata mostrando il compartimento degli attributi vuoto.

Il nome di una classe può essere riportato nella forma semplice, ossia solo il nome, oppure corredato dalla lista dei package che ne descrive il percorso (*path name*). Da notare che gli elementi della lista vengono separati da una coppia del carattere "due punti" (`::`). Esempi di nomi estesi potrebbero essere: `java::util::Vector` oppure `com::lvt::system::dialoglayer::MainServlet`. Generalmente, di una classe viene mostrato unicamente il nome. Ciò implica che essa sia definita nel package oggetto di disegno, mentre qualora sia importata da uno differente, sotto il nome sarebbe opportuno riportare il path del package di appartenenza. Per esempio, nome della classe (`MainServlet`) e package di provenienza (`com::lvt::system::dialoglayer`). Il nome di una classe dovrebbe essere un sostantivo oppure un'espressione appartenente al vocabolario del sistema che si sta modellando.

Qualora la classe rappresenti un'istanza di un determinato stereotipo, è possibile specificarne il nome al di sopra di quello della classe, racchiuso dalle famose parentesi angolari (`<<entity>>`, `<<metaclass>>`, `<<singleton>>`). Una notazione grafica alternativa prevede di accostare al nome dello stereotipo una piccola icona, rappresentante lo stereotipo stesso, nell'angolo in alto a destra o, addirittura, di mostrare l'intera classe con la forma dell'icona (*cfr* fig. 7.2). Chiaramente deve sussistere una relazione biunivoca tra il nome dello stereotipo e la relativa icona.

Figura 7.2 — *Rappresentazione del compartimento del nome di una classe. Lo stereotipo "controller" è tipicamente utilizzato nel modello di analisi che, come si vedrà nel prossimo capitolo, si usa per rappresentare formalmente i requisiti funzionali, catturati attraverso i vari diagrammi dei casi d'uso, e per dar luogo a una primissima versione del modello di disegno. Spesso, lo use case a cui si riferisce la classe è mostrato attraverso l'introduzione del tagged value* `UseCase`*, che, chiaramente, non ha alcun significato per lo UML, ma è utile al lettore: in sistemi di dimensioni medie e grandi agevola il tracciamento* (tracking) *della provenienza degli elementi controller. Infine l'attributo* `leaf` *appartiene al metamodello e viene utilizzato per indicare che la classe non può essere ulteriormente specializzata. Equivale alla parola chiave* `final` *in Java.*



Subito sotto il nome della classe, è possibile specificare una lista di stringhe riportanti o appositi valori etichettati (*tagged value*), o attributi del metamodello. Per esempio classi astratte possono essere evidenziate sia riportando il relativo nome in carattere *corsivo*, sia visualizzando l'attributo del metamodello `{Abstract = true}`. Qualora si tratti di un campo booleano, la presenza della relativa parola chiave rappresenta automaticamente un valore `true`, mentre l'assenza equivale a un valore `false`. Quindi `{Abstract = true}` e `{Abstract}` sono equivalenti.

> La realizzazione dei vari modelli dello UML, e in particolare di quelli facenti riferimento a classi e oggetti, andrebbero realizzati utilizzando la lingua inglese. A questo punto già si possono sentire le urla di dolore di coloro che con tale idioma non hanno una "corrispondenza di amorosi sensi". Ciò è indispensabile in contesti di aziende multinazionali, ed è comunque utile per consolidare i vantaggi propri dello UML (come facilità di circolazione e riutilizzo dei modelli, ecc.). Infine serve per evitare storpiature dovute alla derivante lingua "mista", come metodi getIndirizzo o setDataFineValidita.

La notazione per la rappresentazione grafica delle classi prevede le seguenti regole:

- il nome in **grassetto** centrato riportato nell'apposita sezione;
- le altre parole chiave (come il nome dello stereotipo o eventuali tagged value) riportati in carattere tondo sempre nel compartimento dedicato al nome della classe;

- il nome della classe segue la convenzione generalmente accettata di riportare la prima lettera in maiuscolo e le restanti in minuscolo (`Articolo`, `Fattura`, `Utente`, ecc.), salvo utilizzare nuovamente una lettera maiuscola per distinguere nomi composti (`CarrelloDellaSpesa`, `LineaFattura`, `IntestazioneFattura`, sempre in maiuscolo la prima lettera di eventuali altri nomi);

- gli attributi e le operazioni riportati a caratteri tondi allineati a sinistra;

- la convenzione dei nomi rispetta quella specificata per le classi, con l'eccezione della primissima lettera riportata in minuscolo anziché maiuscolo;

- metodi astratti e nome delle classi astratte vanno riportati in *corsivo*;

- metodi e attributi statici sottolineati.

Quanto riportato di seguito, sebbene per molti lettori possa risultare piuttosto insolito, corrisponde a verità. La notazione dello UML prevede (tool permettendo) la possibilità di definire compartimenti supplementari specificati dall'utente, da utilizzarsi per indicare ulteriori proprietà del modello. Per esempio si potrebbero utilizzare compartimenti supplementari per evidenziare concetti quali le regole del business, responsabilità, eventi gestiti, eccezioni scatenate, ecc. Queste informazioni supplementari potrebbero essere rappresentate sia attraverso una lista semplice, sia per mezzo di formati più complessi messi a disposizione dallo UML. Per esempio un compartimento dedicato alle responsabilità (*cfr* fig. 7.3) di una classe potrebbe essere utile sia per agevolare il lavoro dei programmatori, sia per applicare metodologie derivate dalle CRC Cards (metodologia che sarà descritta nel Capitolo 8).

Qualora si dia luogo a compartimenti supplementari, è molto importante riportare il nome che li identifica, sia per fornire maggiori informazioni, sia per distinguerli chiaramente da quelli standard.

I compartimenti sono identificati dal nome che va riportato in alto al centro del compartimento stesso. Per questioni di comodità i nomi dei compartimenti predefiniti non vengono visualizzati.

Un **attributo** è una proprietà della classe, identificata da un nome, che descrive un intervallo di valori che le sue istanze possono assumere. Il concetto è del tutto analogo a quello di variabile di un qualsiasi linguaggio di programmazione. Una classe può non avere attributi o averne un numero qualsiasi. Quando però gli attributi superano l'ordine della decina, può essere il caso di verificare che non si siano inglobate più classi in una sola. Esempi di attributi sono il colore di un oggetto grafico, le coordinate, la data di nascita di un impiegato, le dimensioni di una ruota, il simbolo di una valuta, e così via. Un attributo è, a sua volta, un'astrazione di un tipo di dato o di stato che un oggetto di una classe può assumere.

Figura 7.3 — *Esempio di utilizzo di un compartimento relativo alle responsabilità di una classe. La classe potrebbe essere parte di un sistema per il commercio elettronico, e in particolare potrebbe appartenere a un componente in grado di gestire le informazioni necessarie a un servizio di "allarme". Si tratta di un agente in grado di segnalare all'utente il verificarsi di una precisa condizione (per esempio "prodotto in sconto"). La realizzazione del servizio, chiaramente, richiede la collaborazione di diverse classi non visualizzate. La classe da sola potrebbe rappresentare piuttosto un'interfaccia, magari di un componente EJB, che una classe vera e propria.*

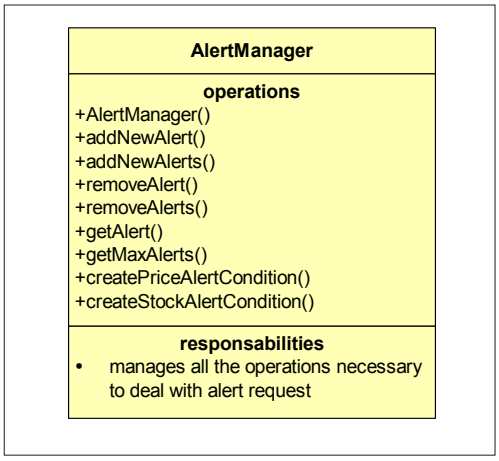


Figura 7.4 — *Esempio di utilizzo di un compartimento dedicato alle eccezioni scatenabili dalla classe. In genere però, si preferisce una notazione alternativa che prevede di rappresentare le varie eccezioni attraverso apposite classi connesse alla classe in grado di scatenarle attraverso relazione di dipendenza, come mostrato di seguito.*

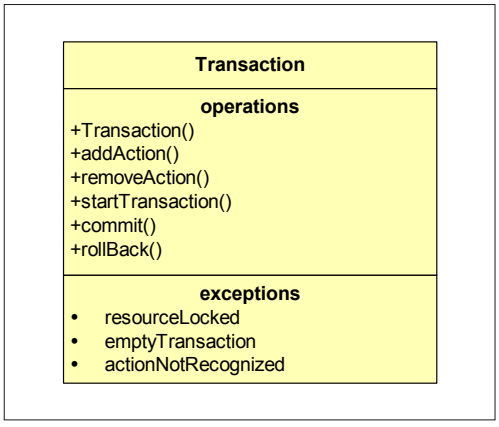
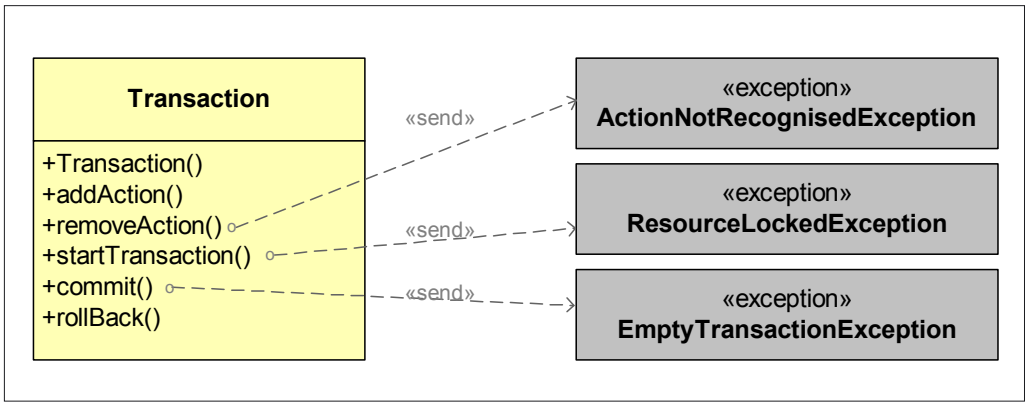


Figura 7.5 — *Rappresentazione delle eccezioni scatenabili da una classe. L'utilizzo dello stereotipo* `<<send>>` *delle relazioni di dipendenza è del tutto legittimo se si considera che, nel metamodello UML, le eccezioni rappresentano una particolare versione di segnali. Come si può notare, la rappresentazione in fig. è più precisa poiché permette di specificare esattamente quali sono i metodi in grado di generare le eccezioni.*



Nel metamodello UML (*cfr* fig. 7.6), la metaclasse Attributo è definita come specializzazione della caratteristica strutturale (StructuralFeature) associabile alla metaclasse Classifier (classificatore) di cui la metaclasse Class rappresenta una specializzazione. In altre parole, gli elementi che specializzano i classificatori (classi, attori, componenti, ecc.), a meno di particolari vincoli, posso dichiarare una serie di attributi.

Di un attributo è possibile indicare solo il nome, oppure il nome e il tipo, oppure la precedente coppia corredata da un valore iniziale. Eventualmente è possibile specificarne lo stereotipo. I seguenti sono validi esempi dello UML: `- nome:string`, `-numeroPagine:integer = 0`, `<<EJBCompField>> customerId:String`. Le convenzioni relative agli attributi sono specificate poco più avanti.

Un'**operazione** può essere considerata come la trasposizione, in termini informatici, di un servizio che può essere richiesto ad ogni istanza di una classe per modificare lo stato del sistema od ottenere un servizio. Pertanto, un'operazione (livello di specifica dei metodi) è un'astrazione di qualcosa che può essere eseguito su tutti gli oggetti di una stessa classe.

Una classe può anche non disporre di alcun metodo, sebbene ciò sia piuttosto improbabile, oppure averne un numero qualsiasi. Come nel caso degli attributi, qualora il numero dei metodi presente nella classe superi l'ordine delle decine, è consigliabile verificare se sia il caso di suddividerla in più classi di dimensioni ridotte, specializzate per compiti con un

grado superiore di coesione. Ciò, entro i limiti della razionalità, favorisce la riusabilità del codice.

Per ciò che concerne le convenzioni sul nome dei metodi, essi dovrebbero essere costituiti da brevi frasi contenenti dei verbi che rappresentano qualche comportamento o funzionalità della classe di appartenenza. La convenzione sul modo di scriverli rispecchia pienamente quella degli attributi: la prima lettera in minuscolo e le prime lettere delle parole successive in maiuscolo. Alcuni esempi di metodi sono `+isEmpty():boolean`, `+move(p:Point)`, `+addElement()`.

> Nel metamodello UML (*cfr* fig. 7.6), la metaclasse metodo (`Method`) è una specializzazione della metaclasse caratteristica comportamentale (`BehavioralFeature`) che a sua volta è specializzazione della metaclasse `Feature` da cui eredita anche la superclasse di attributo. Anche `Method` è associata alla metaclasse `Classifier`. Ciò equivale a dire che, a meno di particolari restrizioni, gli elementi dello UML che specializzano il classificatore possono dichiarare un insieme di metodi.

Figura 7.6 — *Attributi e metodi nel metamodello UML. Il diagramma mostrato in figura presenta un livello di complessità eccessivamente elevato per coloro con non molta esperienza del formalismo dei diagrammi delle classi. Il consiglio è di riesaminarlo al termine dello studio del presente capitolo. Ogni classificatore è costituito da diverse caratteristiche (*`Feature`*); queste sono di due tipologie: comportamentali (*`BehavioralFeature`*) e strutturali (*`StructuralBehavior`*). Quest'ultima è data dagli attributi. Le caratteristiche comportamentali sono espresse attraverso la descrizione delle operazioni (livello di specifica), ognuna delle quali è implementata da uno o più metodi. Come si può notare, le classi* `Classifier`, `Feature`, `BehavioralFeature` e `StructuralBehavior` *sono astratte.*

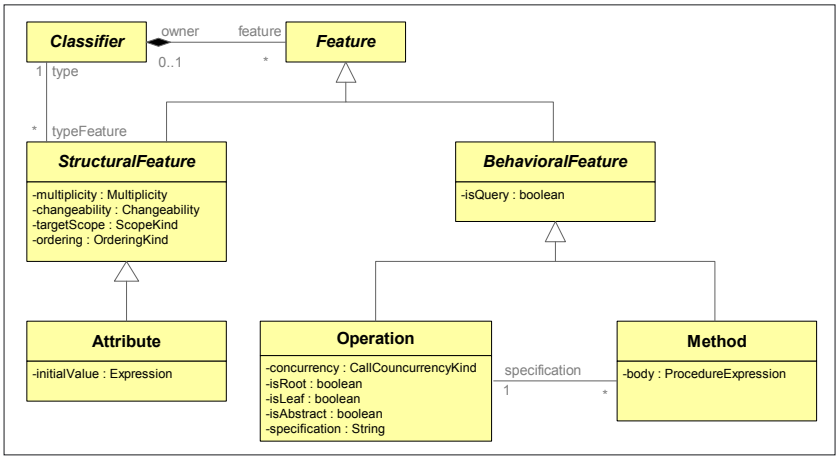
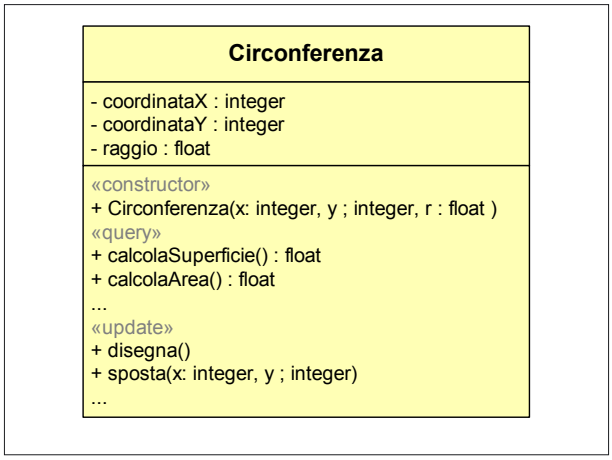


Figura 7.7 — *Organizzazione della lista dei metodi per mezzo dell'uso degli stereotipi. Purtroppo, ben pochi tool commerciali permettono di realizzare un effetto come quello di figura.*



Quando si disegna una classe non è assolutamente necessario riportarne tutti gli attributi e/o tutti i metodi; molto spesso essi sono in numero così alto che non ci sarebbe neanche lo spazio fisico per visualizzarli tutti. Qualora non si vogliano specificare tutti gli elementi di una sezione è consigliabile riportare quelli ritenuti più significativi nel contesto oggetto di studio e riportare i punti di sospensione (. . .), in fondo alla lista, al fine di evitare ogni possibile ambiguità. Un'altra possibilità offerta dallo UML per organizzare lunghi elenchi (proprietà o metodi) consiste nell'organizzarli introducendo opportuni stereotipi.

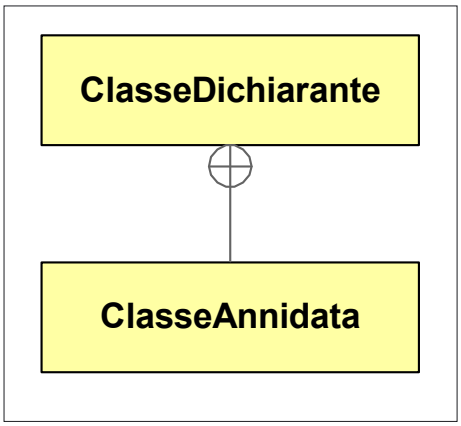
È appena il caso di ricordare come, nel caso degli attributi, la visibilità pubblica vada assolutamente utilizzata con parsimonia. Facendo riferimento ai princìpi dell'*information hiding* e dell'incapsulamento, si ricorda che è necessario che gli attributi siano manipolati dalle classi di essi "proprietarie", giacché dovrebbero essere le uniche a conoscerne la logica e le eventuali operazioni causate dai valori assunti. Sempre valide sono poi le regole di massima coesione, minimo accoppiamento, ecc.

Classi annidate

Classi dichiarate all'interno di altre sono definite annidate (*nested* o *inner*). Una classe annidata appartiene all'ambito di denominazione (*namespace*) della classe dove è definita e quindi è utilizzabile unicamente al suo interno.

Con la versione 1.4 dello UML è possibile mostrare l'annidamento di una classe all'interno di un'altra attraverso un apposito formalismo: si connettono le due classi con un segmento riportante un apposito simbolo detto *anchor* (ancora) all'estremità della classe dichiarante. L'*anchor* è una croce inscritta in una circonferenza come mostrato nella fig. 7.8.

Figura 7.8 — *Rappresentazione di classi annidate.*

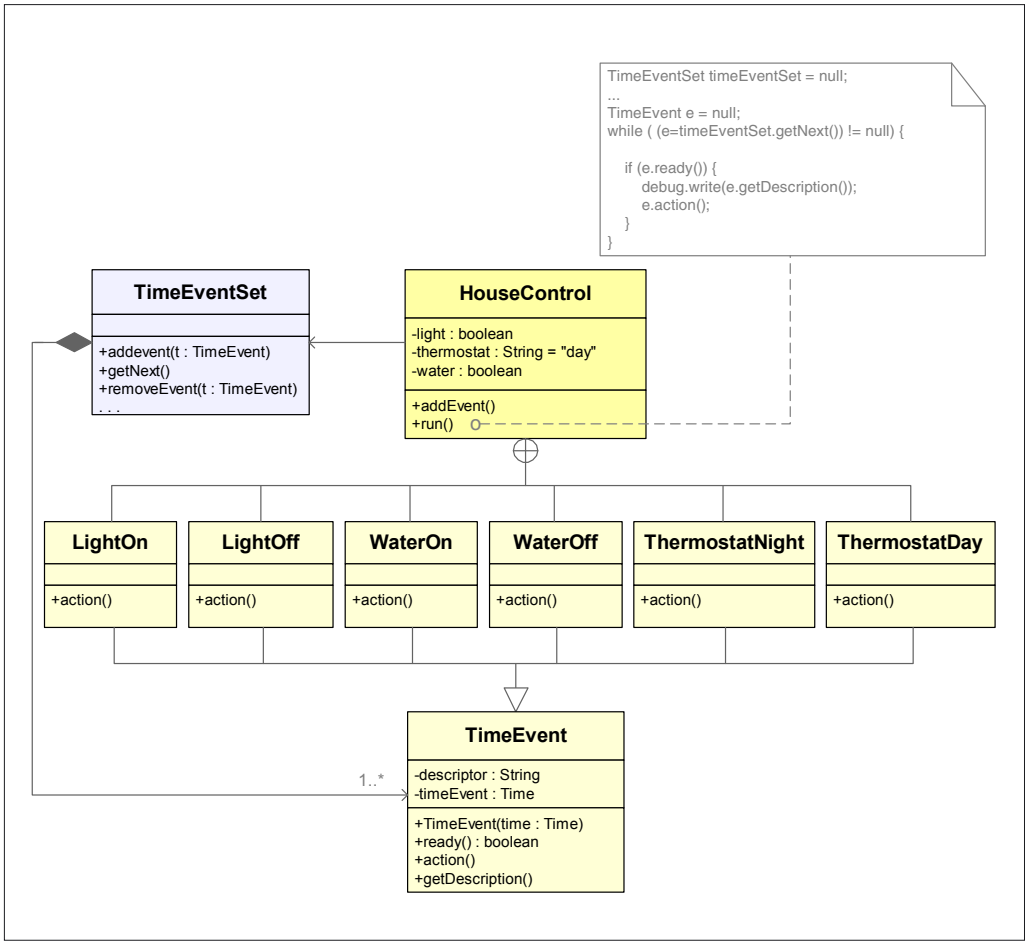


In questa sezione, prendendo spunto da quanto esposto in uno dei migliori libri scritti su Java, ormai assurto al ruolo di "classico", ovvero *Thinking in Java* di Bruce Eckel [BIB34], si mostra un esempio relativo alle classi annidate. Il concetto di annidamento tra classi risulta molto utile nel contesto dei framework di controllo. Una trattazione adeguata esula comunque dagli obiettivi del presente libro: esistono volumi interi dedicati all'argomento.

Tanto per cambiare, anche il termine di *framework* appartiene all'insieme di quelli abusati, utilizzati per indicare tanti concetti, spesso anche discordanti. In questo contesto il termine è utilizzato per indicare un insieme di classi opportunamente disegnate e organizzate per risolvere categorie di problemi simili. La logica conseguenza è che il *framework*, per risolvere una specifica istanza di dominio, necessita di un processo preliminare di "personalizzazione". Tale processo, in genere, consiste nel definire l'implementazione di interfacce predefinite o di specializzare opportunamente il comportamento di determinate classi. Il principio di funzionamento si basa sulla possibilità di disporre di classi che espongono un "comportamento" esterno predefinito — o stabilito in specifiche interfacce da implementare o in classi da cui ereditare, tipicamente dichiarate astratte — perciò conosciuto dalla logica interna del *framework*. Quest'ultima è quindi in grado di riferirsi alle nuove classi in modo astratto. La specializzazione del comportamento (*ridefinizione* o *override* di opportuni metodi) permette poi di risolvere i compiti specifici.

Il *framework di controllo* è particolarmente efficace nei contesti di sistemi guidati dai messaggi (*event driven system*), ossia in sistemi che devono rispondere alla ricezione di segnali, generalmente, in tempi molto brevi. Un esempio tipico è la gestione dell'interfaccia utente in cui le azioni eseguite dall'utente (pressione di un bottone, inserimento del testo in un apposito campo, ecc.) corrispondono a eventi che il sistema deve gestire.

Figura 7.9 — *Esempio leggermente semplificato del framework control proposto nel libro* Thinking in Java.



Un semplice esempio è riportato nella fig. 7.9. Per tutti coloro che non hanno molta padronanza dei diagrammi delle classi può risultare decisamente complesso e, verosimilmente, diverrà più chiaro con il procedere del capitolo.

In ogni modo si tratta di un semplice prototipo di un sistema di controllo di una casa in grado di gestire automaticamente determinate operazioni opportunamente "schedulate", quali accendere e spegnere le luci, aprire e chiudere l'acqua degli annaffiatoi, e così via. In questo contesto, le azioni sono simulate dall'impostazione del valore di precise variabili mentre, in un sistema reale,

potrebbero essere sostituite da un opportuno impulso inviato a un servomeccanismo opportunamente collegato a un computer.

La classe in cui avviene il controllo è HouseControl, che dichiara al suo interno una serie di altre classi annidate, quali: LightOn, LightOff, WaterOn, WaterOff, ThermostatNight e ThermostatDay. Queste rappresentano eventi particolari: ereditano tutte dalla stessa classe TimeEvent. Come si vedrà in seguito, ciò significa che ne ereditano e specializzano il comportamento. Il discendere da una classe comune, e quindi disporre del medesimo "aspetto", consente alla classe HouseControl di conoscerne la porzione di comportamento necessario per controllarle.

La classe HouseControl si avvale della collaborazione della classe TimeEventSet, per memorizzare gli eventi da gestire. In particolare, ciascuno di essi è caratterizzato dall'orario in cui deve essere eseguito e da un descrittore. Per esempio, l'acqua degli annaffiatoi deve essere aperta alle 18 per poi essere chiusa alle 20. Ciò è rappresentato dalla presenza di due istanze di evento nella classe TimeEventSet. Tali istanze sono la specializzazione WaterOn e WaterOff.

Il funzionamento prevede la configurazione iniziale in cui si creano tutte le istanze degli eventi da inserire nell'istanza della classe TimeEventSet, e quindi l'invocazione del metodo run(). In effetti non si tratta del modo migliore di realizzare tale metodo... Verosimilmente, così congegnato, costituisce uno dei quei metodi in grado di monopolizzare la CPU. In ogni modo il suo compito è cercare all'interno della lista degli eventi per individuarne qualcuno che abbia raggiunto l'orario della propria esecuzione (l'invocazione del metodo ready() restituisce un valore true), al fine di invocarne il metodo action(). Chiaramente la logica di controllo non ha la minima idea di quali azioni poi il metodo esegua. Nel diagramma in questione, ciascuna specializzazione della classe TimeEvent si limita ad agire sulla propria variabile dichiarata nella classe HouseControl. Per esempio, il metodo action() della classe LightOn si occupa di inizializzare la variabile booleana light al valore true.

Come si può notare, la struttura della classe HouseControl non è molto dissimile da quella di una classe utilizzata per gestire gli eventi originati da un particolare oggetto GUI: si dichiarano opportune classi annidate, che implementano interfacce listener predefinite, per potervi veder eseguito il corrispondente del metodo action() qualora si verifichi il relativo evento.

Interfacce

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 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. 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 realizzano fisicamente. Qualora una classe concreta implementi un'interfaccia deve necessariamente dichiarare tutte le operazioni definite da quest'ultima. Le interfacce non possiedono implementazione, attributi o stati; dispongono unicamente della dichiarazione di operazioni (firma) e possono essere connesse tra loro tramite relazioni di generalizzazione. Visibilità private dei relativi metodi possiedono ben poco significato.

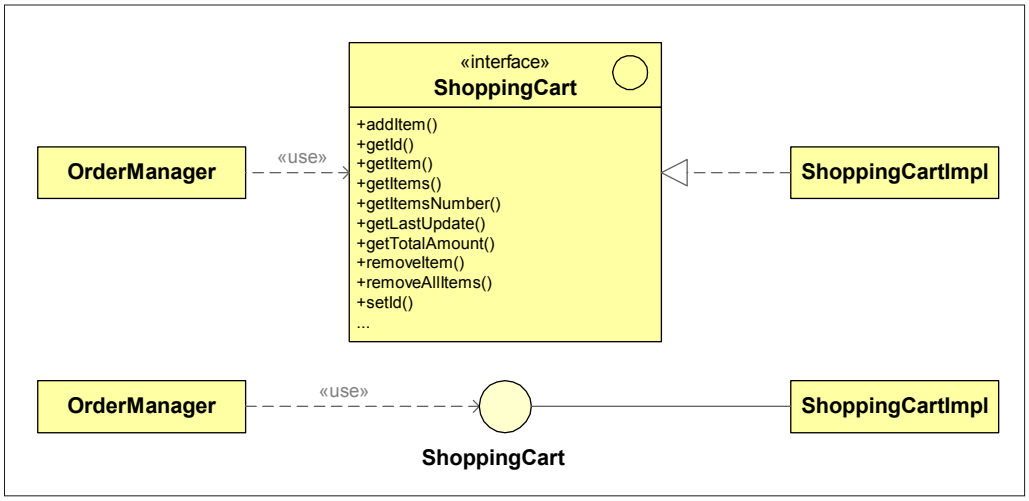
Tipicamente una singola interfaccia dichiara un comportamento circoscritto, ben definito, ad elevata coesione; pertanto elementi con comportamento complesso possono realizzare diverse interfacce.

Nel metamodello UML le interfacce sono definite come specializzazioni della metaclasse `Classifier`, quindi possono essere rappresentate graficamente attraverso il classico formalismo del rettangolo diviso in compartimenti (come mostrato nel paragrafo relativo alle classi). In questo caso però è necessario riportare la parola chiave `<<interface>>` sopra al nome dell'interfaccia, proprio per evidenziare la particolare semantica di questo classificatore. Il compartimento dedicato agli attributi è solitamente omesso perché sempre vuoto, mentre la lista delle operazioni dichiarate dall'interfaccia si riporta nel relativo compartimento. Eventualmente è possibile rappresentare graficamente le interfacce attraverso le apposite notazioni alternative: nel compartimento del nome aggiungere, allineata a destra, l'icona corrispondente all'interfaccia (cerchio), oppure mostrare direttamente l'icona al posto del rettangolo con specificato unicamente il nome della stessa. Qualora si decida di utilizzare quest'ultima notazione, il cerchio è associato al classificatore che l'implementa attraverso una linea continua. Il significato è abbastanza intuitivo: il classificatore definisce tutte le operazioni dichiarate dall'interfaccia. Un classificatore può implementare diverse interfacce e quindi ciascuna dichiara esplicitamente una sezione del relativo comportamento.

> Sebbene l'obiettivo della presente spiegazione sia l'interfaccia, nel contesto dei diagrammi delle classi si utilizza spesso la denominazione generale di *classificatore*, poiché quanto asserito mantiene la propria validità in tutti i contesti in cui è possibile utilizzare il concetto di interfaccia: casi d'uso, diagramma delle classi, diagramma dei componenti ecc. Si ricordi che `Classifier` è la metaclasse astratta le cui specializzazioni sono: classi, interfacce, nodi, tipi di dati, componenti, casi d'uso, ecc.

La notazione basata sull'esclusiva visualizzazione dell'icona offre il vantaggio di mostrare più elegantemente le interfacce, di evidenziare nettamente la separazione della dichiarazione dall'implementazione degli elementi presenti in un diagramma, e così via. Però, a fronte di questi vantaggi, esiste il limite, per altro eluso da diversi tool, di non poter visualizzare la lista delle operazioni. Ciò fa sì che molto spesso si preferisca utilizzare la notazione classica del classificatore (rettangolo) al fine di poter visualizzare l'elenco delle operazioni. In tal caso, l'interfaccia viene associata al classificatore che la realizza attraver-

Figura 7.10 — *Notazioni grafiche utilizzabili per visualizzare istanze di interfacce. Nell'immagine in alto l'interfaccia è visualizzata utilizzando la notazione classica del classificatore, mentre in quella in basso si utilizza la corrispondente icona. In questo esempio l'interfaccia permette di separare nettamente la dichiarazione di un carrello della spesa dalla relativa implementazione. Questo permette alla classe di gestione degli ordini di ottenere le informazioni desiderate dagli oggetti "carrello della spesa" senza dover necessariamente far riferimento a un'implementazione specifica, o conoscerne l'implementazione della classe.*



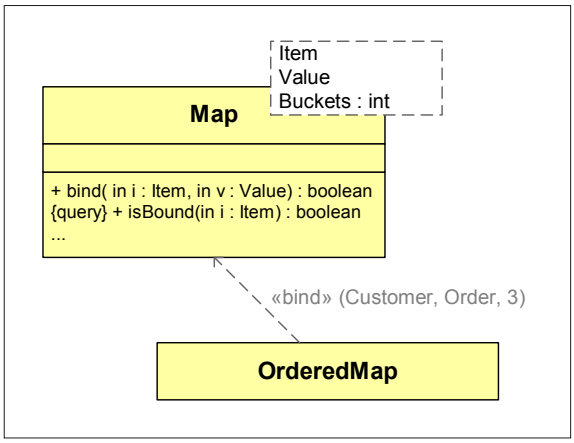
so un segmento tratteggiato culminante con il triangolino (una sorta di relazione di generalizzazione con segmento tratteggiato) rivolto nella direzione dell'interfaccia. La relazione tra gli elementi che utilizzano l'interfaccia per pilotare in modo astratto i classificatori che implementano l'interfaccia stessa e questi ultimi, a seconda del contesto, può essere visualizzata sia attraverso una relazione di dipendenza sia per mezzo di opportune versioni della relazioni di associazione. In quest'ultimo caso è opportuno indicare la navigabilità (freccia diretta verso l'interfaccia) per evitare ogni possibile confusione di notazione.

Classi parametrizzate: i famosi template

Per coloro che non possono vantare una certa esperienza di linguaggio di programmazione C++ il concetto di template potrebbe risultare piuttosto inconsueto. In effetti non tutti i linguaggi di programmazione (come per esempio Java) lo prevedono.

Un template è un descrittore di una classe con uno o più parametri. Pertanto un template non definisce una singola classe, bensì una famiglia, in cui ciascuna istanza è caratterizzata

Figura 7.11 — *Notazione grafica utilizzata per le classi parametrizzate.*



dal valore assunto da tali parametri. Questi valori diversificano le varie classi condizionando il comportamento di specifici metodi.

Un template non è direttamente utilizzabile in quanto la presenza dei parametri non permette di definire una singola classe. Quando poi il valore dei parametri viene precisato, si ha la specializzazione del template e quindi si dispone di una classe ben definita e utilizzabile.

Un template può partecipare a diverse relazioni con altre classi. Per esempio può ereditare da una classe ordinaria. Ciò equivale a dire che tutte le classi ottenute dal template, specificando opportunamente i relativi parametri, ereditano dalla stessa classe genitore.

La notazione grafica di un template prevede di rappresentare normalmente la classe, aggiungendo un rettangolo tratteggiato nell'angolo in alto a destra destinato a ospitare la dichiarazione della lista dei parametri (fig. 7.11).

> Il template è un meccanismo utilizzato da molti linguaggi di programmazione per poter definire collezioni omogenee di oggetti. In sostanza questo meccanismo permette di definire classi equivalenti alle collezioni Java (`Vector`, `Hashtable`, `ArrayList`, ecc.). Poiché in C++ non esiste il concetto della gerarchia di classi che ereditano da un progenitore comune (quello che in Java è la classe `Object`), il template rappresenta l'unico meccanismo per realizzare classi generiche da specializzare, in grado di gestire collezioni omogenee di oggetti. La differenza più evidente tra le collezioni Java e i template, risiede nel fatto che questi ultimi permettono di definire classi che dichiarano il tipo di dato che il template tratta. Ciò permette di avere maggiore controllo degli elementi che l'oggetto è in grado di memorizzare e di evitare i noiosissimi, e virtualmente pericolosi, *downcast*.

Il tipo enumerato (*enumeration*)

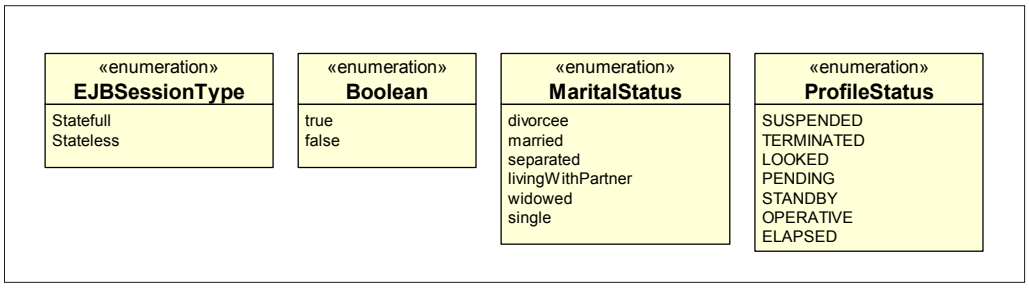
Il tipo enumerato rappresenta un tipo di dato definito dall'utente le cui istanze sono costituite da insiemi di "nomi", sempre definiti dall'utente. Questi nomi sono caratterizzati dal possedere un determinato ordine, ma nessuna algebra. Un tipo enumerato, generalmente, viene associato a un attributo per vincolare l'insieme dei valori che può assumere. Così come definito, il tipo enumerato non ha molta attinenza con i concetti di classe, però si è deciso ugualmente di introdurlo in questa sezione sia perché a questo si faranno parecchi riferimenti, specie più avanti, sia perché la relativa notazione presenta molte similitudini con quella delle classi. In ultima analisi si tratta di un altro discendente della metaclasse `Classifier`. Ciò, tra l'altro, permette di rappresentare i tipi enumerati attraverso il classico rettangolo con il nome inserito nell'apposito compartimento e la parola chiave `<<enumeration>>` per evidenziarne la natura. Nel compartimento centrale (quello tipicamente riservato agli attributi) si specificano i valori: uno per ogni linea, mentre in quello finale eventualmente è possibile specificare alcune operazioni inerenti il tipo enumerato. Per esempio, per il tipo enumerato `Boolean` è possibile specificare le seguenti operazioni:

```
and(with:Boolean)  : Boolean
or(with:Boolean)   : Boolean
xor(with:Boolean)  : Boolean
not()              : Boolean
```

Definizione formale di attributi e metodi

Nel presente paragrafo viene illustrata la definizione rigorosa del concetto di attributo e metodo nello UML. Data la sua natura si tratta di un paragrafo dedicato a un pubblico amante delle definizioni formali: per questo può essere trascurato o comunque letto molto rapidamente da tutti gli altri lettori.

Figura 7.12 — *Esempi di tipi enumerati.*



Si cominci con l'esaminare la definizione formale di attributo riportata di seguito:

```
visibilità nome   :  tipo-espressione [molteplicità ordinamento]
                  =  valore-iniziale {stringa-proprietà}
```

La visibilità specifica le regole secondo le quali il relativo attributo è accessibile da parte di altri oggetti. In particolare, le tipologie di visibilità previste sono:

- **pubblica**: l'attributo è accessibile da qualsiasi altro oggetto dotato di riferimento all'oggetto che contiene l'attributo in questione;
- **privata**: l'attributo è accessibile solo all'interno della classe di appartenenza (dichiarante);
- **protetta**: l'attributo è accessibile da tutte le istanze delle classi che "ereditano" da quella in cui l'attributo è definito;
- **package**: l'attributo è accessibile da qualsiasi altro oggetto istanza di classi appartenenti allo stesso package o in un altro ad esso annidato a qualsiasi livello.

Per evidenti motivi di praticità, i nomi delle visibilità sono sostituiti con i simboli standard:

\+ indica la visibilità pubblica
– indica la visibilità privata
\# indica la visibilità protetta
~ indica la visibilità package.

Nulla vieta però di utilizzare icone più accattivanti, come avviene in diversi tool commerciali.

La visibilità è un campo obbligatorio, sebbene la visualizzazione possa essere omessa. Qualora ciò avvenga, non esiste alcun default: in altre parole, se non è esplicitamente espresso, non è corretto assumere che la visibilità sia di tipo pubblica o privata o checchessia. Infine, è possibile aggiungere ulteriori visibilità dipendenti dal particolare linguaggio di programmazione utilizzato, come per esempio la visibilità "implementativa" (*implementation*) prevista dal C++.

Per quanto concerne il `nome`, si tratta di un identificatore rappresentante appunto il nome dell'attributo. Le convenzioni seguono quelle universalmente accettate dai linguaggi di programmazione: deve necessariamente iniziare con un carattere alfabetico o un sottolineato (*underscore*, _), essere seguito da una qualsiasi ripetizione di caratteri e nu-

meri ecc. Le convenzioni tipografiche prevedono che la prima lettera sia riportata in minuscolo. In alcuni linguaggi di programmazione si utilizza la convenzione di premettere il carattere sottolineato per distinguere variabili membro di una classe (attributi) da comuni variabili utilizzate per realizzare i vari metodi. In linguaggi quali Java, ciò non è necessario in quanto, qualora si abbia la necessità di evidenziare un attributo membro di una classe, è sufficiente premettere all'attributo stesso la parola chiave `this`.

Il campo `tipo-espressione` può essere il nome di un'altra classe, interfaccia ecc., oppure una stringa, sintassi e semantica delle quali dipendono dal particolare linguaggio di programmazione scelto, che però deve possedere una corrispondenza con il `ProgrammingLanguageDataType` (tipi di dato del linguaggio di programmazione).

> `ProgrammingLanguageDataType` è una metaclasse del metamodello introdotta con la versione 1.4 dello UML. Appartiene al package `Core` ed eredita dalla metaclasse `DataType`. Si tratta di un meccanismo che permette di specificare tipi di dato dipendenti dal particolare linguaggio di programmazione utilizzato. A tal fine è disponibile l'unico attributo `expression`, che rappresenta appunto un'espressione la cui sintassi e semantica dipendono dello specifico linguaggio di programmazione.

Da notare che, sebbene sia possibile specificare come tipo di un attributo un'altra classe, tipicamente si preferisce rappresentare tali informazioni per mezzo di apposite relazioni. Qualora, nel modello di disegno, un tipo sia uno di quelli definiti dal linguaggio di programmazione (`Date`, `Time`, ecc.) o da una nuova classe di utilità (e quindi a scarso valore semantico), si preferisce riportare un apposito attributo senza evidenziare tutte le relazioni per migliorare la leggibilità e l'eleganza dei diagrammi (si evitano molte linee — associazioni — di scarsa importanza).

I campi `molteplicità` e `ordinamento` sono facoltativi (presenza di parentesi quadre). Qualora omessi, la molteplicità viene assunta uguale a uno (`1..1`) e quindi il campo `ordinamento` perde di significato: in genere non ha molto senso definire l'ordine dell'unico elemento di una lista. La molteplicità è espressa specificando il limite inferiore seguito da due caratteri punto (`.`) e dal limite superiore (`indice-inferiore..indice-superiore`). Qualora il numero di elementi sia superiore all'unità, può avere senso definire l'ordinamento. Valori tipici sono: `ordered` (ordinato) o `unordered` (non ordinato). Per default si assume l'assenza di ordinamento.

La molteplicità può essere associata a diversi elementi dello UML; qualora sia associata a un attributo indica quanti valori del tipo specificato possono essere gestiti dall'oggetto in cui l'attributo viene dichiarato. Qualora il limite inferiore e quello superiore coincidano (l'intervallo degenera in un singolo valore) è sufficiente riportare il valore stesso (`3..3 <=> 3`), mentre se il limite superiore può essere virtualmente infinito si specifica il carattere asterisco (`*`) che equivale ad affermare una molteplicità del tipo "`0..*`".

Per esempio in una classe `Persona` potrebbe prevedere i seguenti attributi:

```
firstName          : string
middleName         : string
surname            : string
gender             : Gender
eMail[0..3]        : string
height             : float
dateOfBirth        : Date
eyeColor[1..2]     : EyeColor
...
```

Mentre la classe `Date` appartiene al linguaggio di programmazione, `Gender` potrebbe essere stata introdotta per indicare il sesso della persona e `EyeColor` per rappresentare il colore degli occhi. L'attributo relativo a quest'ultima caratteristica somatica prevede uno o due valori. Il primo caso potrebbe essere utilizzato per persone con entrambi gli occhi dello stesso colore, mentre il secondo potrebbe essere utilizzato per persone con occhi di colore d diverso: sebbene molto raro, è comunque un caso possibile che potrebbe essere di interesse per l'anagrafica di un centro di ricerca.

Il `valore-iniziale` è un'espressione dipendente dal linguaggio di programmazione prescelto, il cui valore, esito della valutazione della relativa espressione, rappresenta il valore dell'attributo nel momento in cui l'oggetto dichiarante viene creato.

Infine il campo `stringa-proprietà`, del tutto opzionale, indica valori di proprietà riferibili all'attributo. La presenza delle parentesi graffe potrebbe creare qualche confusione. In questo contesto indica che un attributo può avere un numero qualsiasi di `proprietà-stringa` (0 o diverse), le quali, tipicamente, sono racchiuse tra parentesi graffe, appunto.

Un'altra informazione molto importante di attributi e metodi è il campo d'azione (*scope*). In particolare sono previsti due tipi:

- **istanza** (*instance*): si tratta di quella di default, in cui ogni istanza della classe gestisce una propria copia dell'attributo;

- **classe** (*classifier*): esiste un solo valore dell'attributo che viene condiviso da tutte le istanze della classe (attributo statico).

> Tipicamente, quando si disegna una classe, si descrive la struttura e il comportamento dei relativi oggetti. Chiaramente non è possibile utilizzare queste caratteristiche fintantoché non sono create istanze della classe. Questo è vero per un campo d'azione di tipo istanza. Le cose variano quando si ha a che fare con campi d'azione statici. In tal caso gli attributi, i metodi e/o le relazioni, dichiarati come tali non sono legati a un singolo oggetto, bensì sono condivisi da tutte le istanze. Proprio per questo motivo si dice che lo scope è a livello di classificatore. In questo caso è possibile accedere agli elementi dichiarati statici anche senza aver generato un'istanza della classe che li contiene.

Gli attributi con scope a livello di istanza sono rappresentati normalmente, mentre a quelli statici viene aggiunta un'apposita sottolineatura. Infine, per default il valore degli attributi è modificabile nel tempo (`{changeable}`), qualora invece ciò non sia vero è possibile specificare diversamente. A tal fine sono disponibili due opzioni:

- `{frozen}` (congelato): i valori assunti dall'attributo non possono essere modificati dopo che l'oggetto di appartenenza è stato istanziato e i relativi attributi sono stati inizializzati. Qualora l'attributo sia una collezione, non è possibile aggiungere altri valori dopo l'inizializzazione.

- `{addOnly}` (solo aggiunta): chiaramente questo vincolo ha senso solo qualora un attributo individui un insieme di valori (`Array`, `Vector`, `HashTable`, ecc.). Esso sancisce che, una volta assegnati dei valori all'attributo, questi non possono essere più variati, mentre è sempre possibile aggiungerne degli altri.

> I vincoli e le proprietà specificabili per gli attributi si materializzano nel metamodello UML come attributi della metaclasse StructuralFeature da cui Attribute eredita (ordering, changeability, multiplicity e targetScope, *cfr* fig. 7.6).

Per ciò che concerne i metodi, la definizione formale è la seguente:

```
visibilità nome(lista-parametri):tipo-espressione-di-ritorno {stringa-proprietà}
```

Per quanto attiene al campo `visibilità`, vale quanto detto per gli attributi. Ovviamente, il frammento di frase "l'attributo è accessibile", va modificato con "il metodo è invocabile".

Il campo `nome`, a sua volta, segue le stesse regole riportate per gli attributi, così come il campo `tipo-espressione-di-ritorno` è completamente equivalente a quello `tipo-espressione` visto in precedenza.

Per quanto concerne `lista-parametri`, si tratta di un insieme di parametri formali separati dal carattere virgola (`,`), ciascuno dei quali rispetta la seguente sintassi:

```
Tipologia nome:tipo-espressione = valore-di-default
```

Il campo `Tipologia` prevede le seguenti alternative: `in`, `out`, `inout`, con significati piuttosto naturali: i parametri `in` sono veri e propri parametri di input e quindi non modificabili all'interno del metodo (per essere precisi, possono anche essere modificati, ma queste modifiche vengono perse poco prima che il metodo termini), quelli di `out` invece sono generati per comunicare un'informazione al metodo chiamante, mentre `inout` sono parametri di input modificabili. Per default, un parametro è di tipo `in`. Da tener presente che non tutti i linguaggi di programmazione permettono di definire parametri

modificabili all'interno del metodo: un esempio è Java in cui il passaggio dei parametri è unicamente per valore (salvo poi copiare il riferimento invece che il valore per questioni di prestazioni).

I campi `nome` e `tipo-espressione` hanno il medesimo significato visto in precedenza, mentre `valore-di-default` è del tutto assimilabile a quanto riportato per il campo `valore-inziale` associato a un attributo.

Anche per quanto riguarda i metodi, quelli statici (scope a livello di classe) vengono evidenziati attraverso apposita sottolineatura.

Metodi che non generano cambiamento di stato del sistema (in altre parole esenti da effetti collaterali) possono essere evidenziati per mezzo della proprietà `{query}`. L'eventuale tipologia di concorrenza prevista da un metodo è evidenziabile per mezzo della proprietà `{concurrency=nome}`, dove il `nome` può assumere i seguenti valori: `sequential` (sequenziale), `guarded` (controllata), `concurrent` (concorrente).

Un metodo astratto, ossia del quale viene dichiarata unicamente la "firma" (*signature*), in modo del tutto consistente con quanto avviene per le classi, può essere visualizzato in carattere *corsivo* oppure associandovi la proprietà `{abstract}`.

I metodi preposti per l'accettazione dei segnali, in oggetti in grado di operare ricevendo e rispondendo a precisi stimoli esterni provenienti sotto forma di segnali, possono essere enfatizzati con la parola chiave `<<signal>>`.

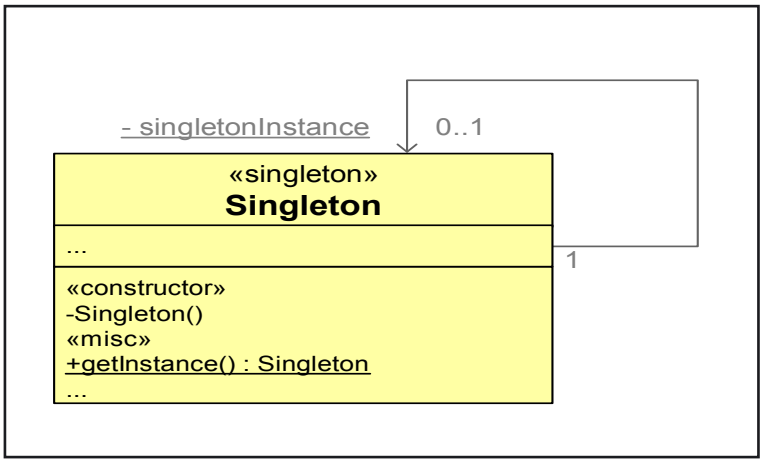
Alcuni esempi di metodi sono:

```
# setId(id:long)
+ getId():long
+ setName(name:String)
+ repaint() {guarded}
```

A conclusione del paragrafo, viene riportato un esempio ricorrente: il famoso *design pattern* (modello di disegno) denominato Singleton. La particolarità consiste nel fatto che l'unica classe di cui è costituito dà luogo, in fase di esecuzione, a una sola istanza. Questa peculiarità lo rende indispensabile ogni qualvolta sia necessario gestire centralmente un insieme di risorse: pool di connessioni al database, pool di thread, elenco dei parametri di inizializzazione di un sistema (i parametri che venivano specificati nei file `.ini`), ecc. La struttura generale del *Singleton* prevede che il costruttore abbia visibilità privata: quindi nessun client può richiedere la generazione di istanze.

Poiché almeno una volta questo metodo deve pur essere invocato, ossia almeno un'istanza deve essere creata, è necessario dichiarare all'uopo un opportuno metodo statico, `+ getInstance()`. Questo metodo viene invocato dalle classi client per ricevere il riferimento all'unica istanza della classe. Qualora ancora non sia stata ancora creata l'unica istanza prevista, il metodo è comunque invocabile in quanto statico, `getInstance()` e, in tal caso, si deve occupare di creare l'unica istanza e quindi restituirne il riferimento. Poiché diversi client potrebbero invocare lo stesso metodo contemporaneamente la prima

Figura 7.13 — *Struttura del design pattern Singleton.*



volta, la creazione dell'istanza dell'oggetto deve essere controllata, altrimenti si correrebbe il rischio di creare più istanze del Singleton! Per controllare la presenza o meno dell'unica istanza della classe, è necessario dichiarare un attributo (statico ovviamente) di tipo della stessa classe.

Relazioni

Le "entità" coinvolte in un diagramma delle classi raramente sono destinate a rimanere isolate; qualora ciò avvenga, potrebbe essere il caso di rivedere molto accuratamente quanto modellato. In situazioni ordinarie tutte le classi rappresentate sono connesse con altre secondo precisi criteri. La modellazione di un sistema, dunque, non solo richiede di identificare le classi di cui è composto, ma anche di individuare i legami che le interconnettono.

Nel mondo OO, tutte le relazioni si riconducono ai tipi fondamentali **dipendenza**, **generalizzazione** e **associazione**, da cui derivano tutte le altre.

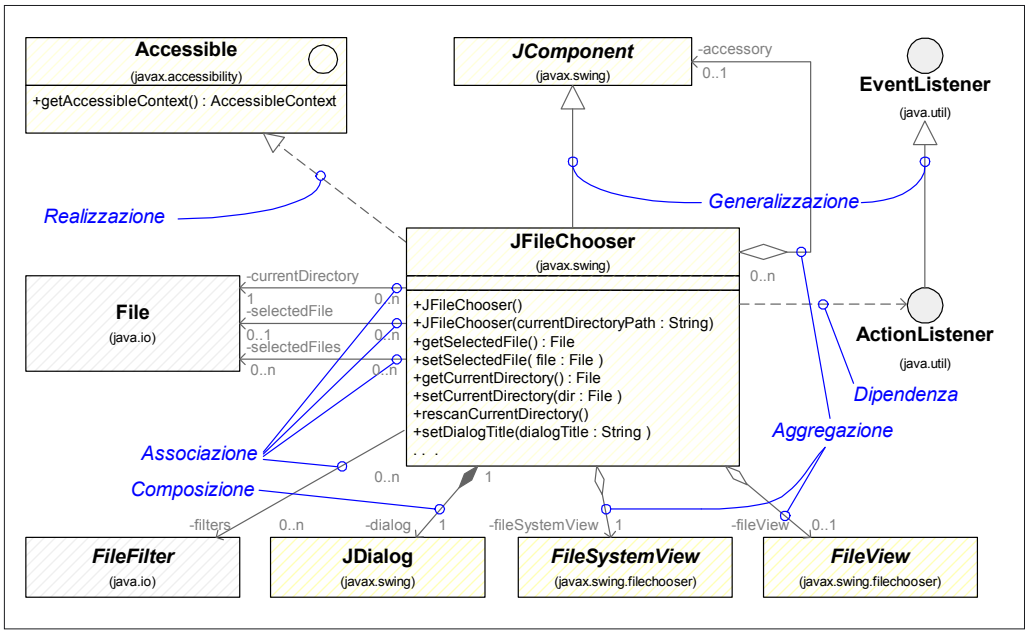
In fig. 7.14 è riprodotta una porzione della struttura statica del componente `JFileChooser` appartenente alla libreria Swing Java. Si tratta del widget che permette di selezionare uno o più file attraverso una finestra simile a quella che compare quando si seleziona l'opzione "apri file" o "salva con nome" in un generico programma. Si tenga presente che il diagramma è stato ottenuto esaminando il codice e quindi diverse interpretazioni possono risultare opinabili.

Ogni relazione fornisce un criterio e quindi una semantica diversa per organizzare le proprie astrazioni. Una relazione di generalizzazione rappresenta un legame tra una classe generale, denominata "superclasse" o "genitore" (*parent*), e una sua versione più specifi-

ca, denominata "sottoclasse" o "figlio" (*child*). Per esempio, tutti i widget Swing estendono la classe astratta `JComponent` (che a sua volta estende la classe `java.awt.Container`, a sua volta specializzazione di quella `java.awt.Component`). Come si può notare, una relazione di generalizzazione può interessare anche le interfacce: `ActionListener` estende `EventListener`. Le classi che intendono ricevere gli eventi generati dalla finestra di dialogo, devono sia implementare l'interfaccia, sia registrarsi come "ascoltatori" presso il componente stesso. In sostanza queste classi devono implementare il metodo `actionPerformed(ActionEvent e)` che gli consente di ricevere l'evento scaturito. La stessa relazione di realizzazione (legame semantico tra elementi in cui uno specifica un contratto che l'altro elemento si impegna a rispettare) non sono altro che particolari versioni della relazione di ereditarietà. In questo caso si eredita unicamente l'interfaccia (il tipo) senza l'implementazione.

Da notare che, nel diagramma in fig. 7.14, le interfacce presenti sono state rappresentate con due diverse notazioni: quella tipica delle classi (`Accessible`) e quella propria delle interfacce (`ActionListener` ed `EventListener`). La seconda è più intuitiva e accattivante, mentre la prima viene preferita qualora si intenda visualizzare i metodi dichiarati nell'interfaccia (nulla vieta di definire un ulteriore stereotipo che unisca i due vantaggi).

Figura 7.14 — *Relazioni dello UML*



Il componente `JFileChooser` è poi legato con una relazione di dipendenza all'interfaccia `ActionListener`. Si tratta di una relazione semantica tra due classificatori, in cui una variazione dell'elemento indipendente (fornitore) può comportare aggiornamenti di quello dipendente (cliente). Tale legame di dipendenza potrebbe apparire errato: ci si attenderebbe che la classe `JFileChooser` memorizzi l'elenco degli oggetti ascoltatori e quindi, in ultima analisi, che sussista una relazione più forte. L'arcano è spiegato dal fatto che la classe che effettivamente memorizza questa lista è `JComponent`, mentre `JFileChooser` si limita a reindirizzare le richieste.

Una relazione di associazione indica un legame strutturale tra oggetti. Di questa relazione esistono diverse versioni, tra cui l'aggregazione e la composizione. Una relazione di associazione tra due classi indica che — a meno di ulteriori vincoli come la navigabilità — è possibile navigare dagli oggetti di una classe a quelli dell'altra e viceversa. Per esempio la classe `JFileChooser` ha tre relazioni di associazione con la classe `File`. Queste sono utilizzate, rispettivamente, per tenere traccia della directory corrente, dell'eventuale file selezionato e degli eventuali file selezionati (nel caso in cui la multiselezione sia abilitata). Come si può notare, è presente un vincolo di navigabilità (freccia riportata in una parte dell'associazione). Questo stabilisce che, dagli oggetti `JFileChooser`, è possibile navigare in quelli di tipo `File` associati, mentre non è possibile il contrario. In sostanza gli oggetti di tipo `File` (legittimamente) non memorizzano un riferimento all'oggetto che li utilizza.

Lo stesso widget può essere associato alla classe astratta `FileFilter`, le cui specializzazioni permettono di definire dei criteri atti a stabilire quali file visualizzare e quali no (per esempio solo i file con estensione `.java`, quelli con estensione `.java`, `.jar`, e così via).

Nel diagramma sono presentate due versioni con semantica più forte della relazione di associazione: aggregazione e composizione. Come si vedrà di seguito, entrambe rappresentano relazioni strutturali tra elementi in cui uno rappresenta il tutto e gli altri le parti. Le composizione, oltre a specificare una relazione di tipo "tutto/parte", impone vincoli aggiuntivi, quali per esempio il fatto che la mancanza di una parte fa perdere di significato al "tutto", che le parti non possano essere condivise, ecc. Per esempio il componente `JFileChooser` è basato su una finestra di dialogo e quindi non si potrebbe avere neanche la classe `JFileChooser` senza la relativa classe (`JDialog`).

Per la `FyleSystemView` potrebbe valere un discorso analogo: se non si disponesse di un oggetto in grado di interpretare le informazioni lette dal file system, non sarebbe possibile presentare molte informazioni all'utente. In questo caso però si è preferito selezionare una relazione di aggregazione. Ciò perché le classi client che utilizzano il widget `JFileChooser` possono impostare (metodo `set`) l'istanza della specifica classe estensione di `FyleSystemView`, in tal caso nulla assicura che la medesima istanza non venga condivisa con altri oggetti. La non condivisibilità delle parti (nello stesso istante di tempo) è un vincolo fondamentale per una relazione di composizione.

La relazione di aggregazione con la classe `JComponent` nel ruolo `Accessory`, permette di associarvi altri componenti dedicati alla realizzazione di compiti particolari, come la visualizzazione dell'"anteprima" dei file.

Dipendenza

La dipendenza è una relazione semantica tra due elementi (o insiemi di elementi) utilizzata per evidenziare una condizione in cui la definizione di un'entità o il relativo funzionamento sono subordinati a quelli di un'altra. Ciò implica che una variazione dell'elemento indipendente (fornitore, *supplier*) genera la necessità di revisionare l'elemento dipendente (cliente, *client*). Ciò al fine di verificare l'eventuale necessità di modificare l'elemento cliente come conseguenza degli aggiornamenti apportati a quello fornitore.

Esempi tipici di relazioni di dipendenza si hanno quando, in assenza di una relazione più "forte", un metodo di un elemento prevede come parametri formali istanze di altre classi, quando un oggetto invoca un metodo di un altro, e così via. In tutti questi casi, esiste un'evidente dipendenza tra le classi coinvolte, sebbene esse non siano necessariamente legate da un'associazione esplicita. Tipicamente, si ricorre a una relazione di dipendenza qualora non ne esistano altre più appropriate; in altre parole, la relazione di dipendenza è la più debole di quelle previste. Un altro esempio molto ricorrente in cui si usa la relazione di dipendenza si ha quando una classe necessiti di utilizzarne altre in modo astratto attraverso opportune interfacce (fig. 7.10). In questi casi, evidentemente, eventuali variazioni dell'interfaccia si ripercuotono anche nelle classi utilizzanti e pertanto ha senso associare queste ultime alle interfacce utilizzate per mezzo della relazione di dipendenza (sempre nel caso in cui non sia presente una relazione più forte).

Dalla definizione della relazione di dipendenza consegue che non si tratta di un elemento di esclusiva proprietà del formalismo dei diagrammi delle classi, ma può essere utilizzata in altri contesti, come per esempio il diagramma dei componenti.

Tipicamente, relazioni di dipendenza non dovrebbero comparire in diagrammi a oggetti del dominio e di business (in tali contesti ha senso evidenziare legami semantici e strutturali tra le varie entità, che quindi sono di natura più forte), e pertanto, nell'ambito dei modelli a oggetti, sono relegate essenzialmente a quello disegno.

La definizione del metamodello UML prevede che, nella relazione di dipendenza, qualsiasi `ModelElement` (elemento del modello) possa svolgere il ruolo di *supplier* e *client*; si tratta della metaclasse astratta da cui derivano tutti gli altri elementi dello UML, come per esempio parametri, attributi, metodi, classi, interfacce, nodi, use case, attori ecc. Logica conseguenza di ciò è che tutti gli elementi dello UML possono essere relazionati tra loro per mezzo di una relazione di dipendenza. Ciò è vero a meno di particolari vincoli, definiti in specifiche specializzazioni della relazione di dipendenza, volti a limitarne il dominio di applicabilità. Per esempio la specializzazione Binding utilizzata per "specializzare" i template (illustrata di seguito) prevede i seguenti vincoli: un `ModelElement` può partecipare come client di una sola relazione di Binding, la relazione dispone di un solo supplier e molti

Figura 7.15 — *Rappresentazione del pattern Data Access Object (DAO). Il* `BusinessObject` *(tipicamente si tratta di un Session EJB, ma può anche trattarsi di una Servlet o di un JavaBean) è il client della situazione, ossia l'oggetto che necessita di accedere a una sorgente di dati (un RDMS o un LDAP, una parte di wrapper di un Legacy System, ecc.) per reperire quelli di sua pertinenza. A tal fine crea un'istanza della classe* `DataAccessObject`*, il cui scopo è quello di incapsulare l'appropriata* `DataSource` *per renderne trasparente l'utilizzo alla classe client. Quindi il* `DataAccessObject` *si occupa di interagire con la relativa sorgente dati e di creare un apposito grafo di oggetti (rappresentato per semplicità da un* `ValueObject`*) da restituire all'oggetto client (l'istanza del* `BusinessObject`*).*

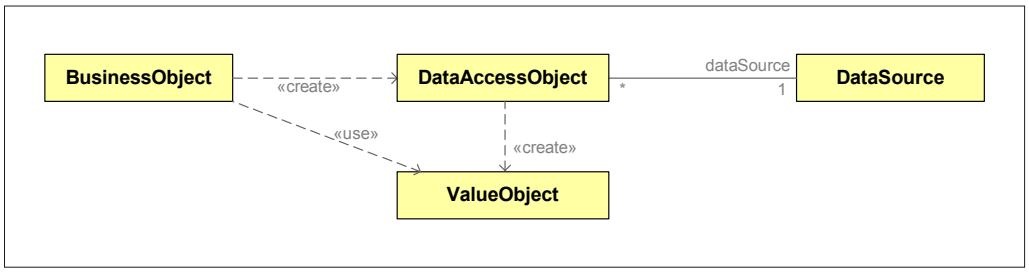
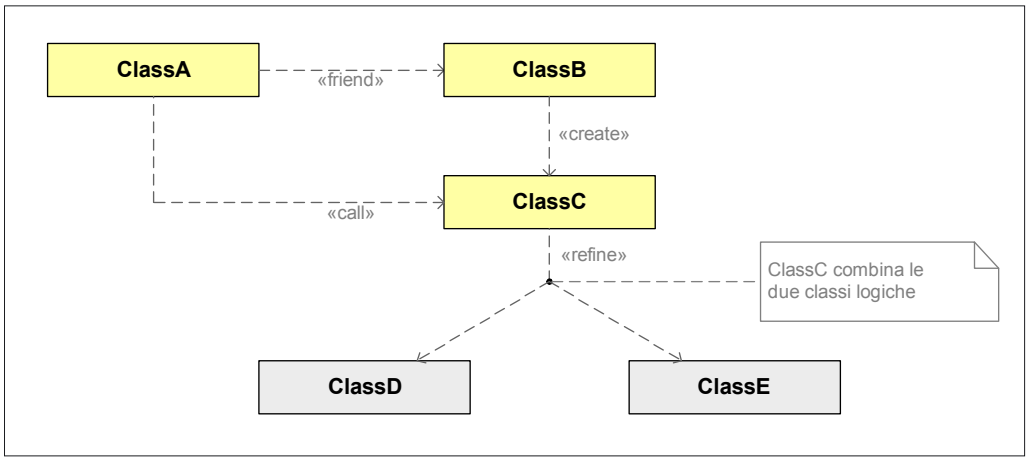


Figura 7.16 — *Esempio (tratto dalle specifiche della versione UML 1.4) di utilizzo di varie specializzazioni della relazione di dipendenza.*



client, il numero degli argomenti deve essere equivalente a quello dei parametri, ogni argomento del `ModelElement` supplier deve essere dello stesso tipo, o di un tipo discendente, del corrispondente parametro del `ModelElement` client, ecc.

Graficamente la relazione di dipendenza è rappresentata attraverso una freccia tratteggiata collegante due elementi, quello dipendente dal lato della coda e quello indipendente dal lato della freccia.

Qualora si verifichi il caso in cui un elemento dipenda da un insieme di altri elementi, la notazione grafica utilizzata prevede il ricorso a una serie di frecce tratteggiate, ciascuna puntante a un elemento indipendente, unite in coda alla freccia associata all'elemento indipendente. È inoltre necessario associare una nota all'associazione proprio nel luogo di congiunzione delle frecce, luogo che eventualmente può essere evidenziato da un apposito punto. La situazione opposta, un insieme di elementi dipendente da uno solo, si presta a essere rappresentata nella medesima maniera, avendo però l'accortezza di invertire l'orientamento delle frecce.

Nel diagramma di fig. 7.16 le istanze di `ClassA` fruiscono dei servizi esposti da oggetti di tipo `ClassB`; l'invocazione è resa possibile dalla visibilità di tipo *friend* di alcuni metodi. Per poter disporre di oggetti di tipo `ClassC`, le istanze di `ClassA` devono invocare opportuni metodi esposti dalle istanze di `ClassB` che si occupano di generare oggetti di tipo `ClassC` da restituire agli oggetti `ClassA`. Da notare che la definizione di `ClassC` combina quella di due classi logiche (`ClassD` e `ClassE`) che come tali vivono in modelli di altre fasi a maggiore livello di astrazione.

Specializzazioni e stereotipi della relazione di dipendenza

La relazione di dipendenza prevede diverse specializzazioni atte a enfatizzare la semantica dell'utilizzo che ne viene fatto (tabb. 7.1, 7.2, 7.3). In particolare, nel metamodello è predefinita una serie di specializzazioni standard (specificate per mezzo delle relative metaclassi: `Binding`, `Abstraction`, `Usage` e `Permission`) che nella pratica sono utilizzate molto raramente in quanto ancora eccessivamente generiche. Al loro posto si utilizzano specifici stereotipi in grado di rappresentare semantica specializzata. Per esempio la metaclasse `Usage`, pur specializzando la relazione di dipendenza, fornisce ancora indicazioni piuttosto generiche relative al proprio utilizzo.

Al fine di restringere ulteriormente la semantica e quindi fornire informazioni di maggior dettaglio, si preferisce adottare gli opportuni stereotipi (`call`, `create`, `instantiate` e `send`). La metaclasse `Usage` è comunque molto importante perché definisce le linee generali della relazione: quali sono i possibili elementi sorgenti, quali quelli destinatari, la semantica generale, e così via, che la contraddistinguono dalle altre famiglie di specializzazioni. Per esempio la metaclasse `Abstraction`, pur essendo una specializzazione della relazione di dipendenza, è profondamente diversa dalla relazione di `Usage`: `Abstraction` indica che gli elementi a cui si applica rappresentano concetti equivalenti a diversi livelli di astrazione.

Tabella 7.1 — *Stereotipi della metaclasse* `Abstraction`*, specializzazione della relazione di dipendenza (metaclasse* `Dependency`*).*

Parola chiave	Nome	Descrizione
abstraction	Abstraction	È utilizzata per relazionare due elementi (o insiemi) che rappresentano lo stesso concetto ma a diversi livelli di astrazione.
derive	Derivation	Si tratta di uno stereotipo della metaclasse *Abstraction* che, come suggerisce il nome, specifica una relazione di derivazione tra elementi del modello. Tipicamente, si tratta di elementi dello stesso tipo. Un elemento derivato, in quanto tale, può essere sempre ricavato dall'esecuzione di qualche operazione definita nell'elemento base. Sebbene ridondante, l'elemento derivato viene comunque definito o per rendere più chiaro il modello o per questioni di efficienza.
realize	Realization	Lo stereotipo *realize* evidenzia una relazione tra elementi del modello, in cui i supplier sono utilizzati per specificare mentre i client sono impiegati per implementare i relativi concetti. Gli elementi implementativi devono necessariamente supportare tutte le operazioni e/o ricevere tutti i segnali specificati nell'elemento dichiarativo. La relazione rappresenta appunto il mapping tra i concetti dichiarati e le relative implementazioni. Questa realizzazione si presta a essere utilizzata per modellare processi di raffinamento, ottimizzazioni, trasformazioni, template, sintesi, framework, composizioni, ecc. Chiaramente non va confusa con la relazione di generalizzazione: in questo contesto ci si riferisce a un livello superiore di astrazione (è pur sempre uno stereotipo della relazione di *Abstraction*).
refine	Refinement	La relazione di *refine* associa elementi del modello a diversi livelli di semantica, come per esempio classi a livello di analisi con le corrispondenti a livello di disegno. Il mapping può essere unidirezionale o bidirezionale, ed è utilizzato per modellare le trasformazioni di elementi presenti in artifact a differenti livelli semantici
trace	Trace	Come suggerito dal nome, lo stereotipo di "tracciamento" è utilizzato per illustrare relazioni tra elementi che rappresentano lo stesso concetto in differenti modelli. Questa relazione risulta molto utile per evidenziare gli elementi generati dai requisiti e le variazioni nei vari modelli. Tipicamente, si tratta di relazioni bidirezionali con significato informale e quindi non computabile.

Tabella 7.2 — *Stereotipi delle metaclassi* `Permission` *e* `Binding`*, specializzazione della relazione di dipendenza (metaclasse* `Dependency`*).*

Parola chiave	Nome	Descrizione
bind	Binding	La relazione di *binding* (legame) rappresenta una relazione tra un elemento *Template*, che in questo caso rappresenta il supplier, e l'elemento del modello generato dal *Template* (il client). Poiché la generazione del nuovo elemento avviene per mezzo dell'associazione dei parametri, ne segue che la relazione *Binding* include i parametri attuali necessari per sostituire quelli formali dichiarati dal *Template*. L'associazione dei parametri (binding appunto) produce la clonazione dell'elemento *Template*, durante la quale avviene la sostituzione dei parametri. Il risultato consiste in un altro elemento del modello che si comporta come se fosse direttamente parte del modello stesso.
permission	Permission	La relazione di "permesso" serve a evidenziare che un elemento del modello (client) è abilitato ad accedere a quelli presenti in un differente *NameSpace* (spazio dei nomi, appartenente al supplier). Per esempio una classe può eseguire i metodi di un'altra, un package può accedere agli elementi pubblici di un altro, ecc. Questa metaclasse prevede tre stereotipi standard: *access*, *import* e *friend*.
access	Access	Si tratta dello stereotipo della specializzazione *Permission*, utilizzato per evidenziare che un package ha il permesso di referenziare gli elementi pubblici presenti in un altro package.
friend	Friend	*Friend* è uno stereotipo della specializzazione *Permission*, i cui partecipanti sono elementi del tipo operazioni, classi, package. In particolare, la relazione sancisce che, l'elemento sorgente (client) può accedere a quello destinatario (supplier), indipendentemente dalle regole di visibilità dichiarate. Pertanto la relazione di "amicizia" estende la visibilità del supplier al fine di consentire all'elemento client di accedervi.
import	Import	La relazione di "importazione" rappresenta uno stereotipo della specializzazione *Permission*. In particolare definisce che lo spazio dei nomi supplier viene aggiunto a quello del client. In altre parole, il contenuto dichiarato pubblico in un package viene inserito in quello di un altro.

Tabella 7.3 — *Stereotipi della metaclasse* `Usage`*, specializzazione della relazione di dipendenza (metaclasse* `Dependency`*).*

Parola chiave	Nome	Descrizione
use	Usage	La relazione di utilizzo è una specializzazione di quella di dipendenza, utilizzata per evidenziare la situazione in cui un elemento client richiede un altro (o un insieme, supplier) per la completa implementazione od operatività. Non si tratta di una relazione utilizzata per tener "traccia" dell'evoluzione di un modello, bensì per di una necessità di collaborazione pendente. Ciò implica che i due elementi coinvolti devono necessariamente appartenere allo stesso modello. La classe si presta ad essere ulteriormente stereotipizzata al fine di rappresentare più accuratamente la natura della collaborazione, come per esempio una classe invoca un metodo di un'altra, un metodo possiede parametri del tipo di un'altra classe, una classe istanzia un'altra e così via.
call	Call	La relazione di "chiamata" è uno stereotipo di quella *Usage*, in cui sia il sorgente (supplier), sia il destinatario (client) devono essere necessariamente operazioni. Eventualmente, la relazione può essere estesa alle classi che definiscono i metodi, con un significato più generale: esiste almeno un metodo delle classi a cui la relazione è applicata. Questa forma risulta molto utile, al fine di ridurre il numero di relazioni da visualizzare, qualora la relazione preveda che diversi metodi della classe chiamante invochino uno o più della classe chiamata.
create	Creation	La relazione di *create* è uno stereotipo di *Usage*, il cui significato è del tutto inequivocabile: l'elemento client genera istanze dell'elemento supplier.
instatiate	Instatiation	Si tratta di uno stereotipo della relazione *Usage*, molto simile a quello denominato *Create*. La differenza risiede nel fatto che in questo è indicata l'operazione dell'elemento client che genera un'istanza dell'elemento supplier.
send	Send	La relazione *send* è uno stereotipo di *Usage*. È utilizzata per evidenziare l'operazione della classe client che genera istanze del segnale dichiarato dalla classe supplier.

Le tre tabelle mostrate in precedenza vanno considerate come se fossero una sola e sono suddivise per ragioni di impaginazione. Lo sfondo grigio indica le specializzazioni della relazione di dipendenza specificate per mezzo di apposite metaclassi nel metamodello UML, seguite dalle relative specializzazioni.

Generalizzazione

La generalizzazione è la relazione definita nello UML per visualizzare legami di ereditarietà tra classificatori e quindi anche tra classi. La definizione formale sancisce che si tratta di una relazione tassonomica tra un elemento più generale (detto padre) ed uno più specifico (detto figlio). Quest'ultimo è completamente consistente con quello più generale e, tipicamente, definisce ulteriori informazioni, in termini di struttura e comportamento. Pertanto un'istanza di un elemento figlio può essere utilizzata in ogni parte in cui è previsto un oggetto di tipo dell'elemento padre. Nel caso di relazione di generalizzazione tra classi, alla nomenclatura "ecclesiale" (padre e figlio) si preferisce quella più specifica di superclasse e sottoclasse (*superclass* e *subclass*).

Si tratta della stessa relazione già definita nel contesto dei casi d'uso, non a caso, nel metamodello UML, la relazione di *Generalizzazione* prevede come dominio di applicazione i classificatori, di cui classi, interfacce, attori, use case, ecc. sono specializzazioni.

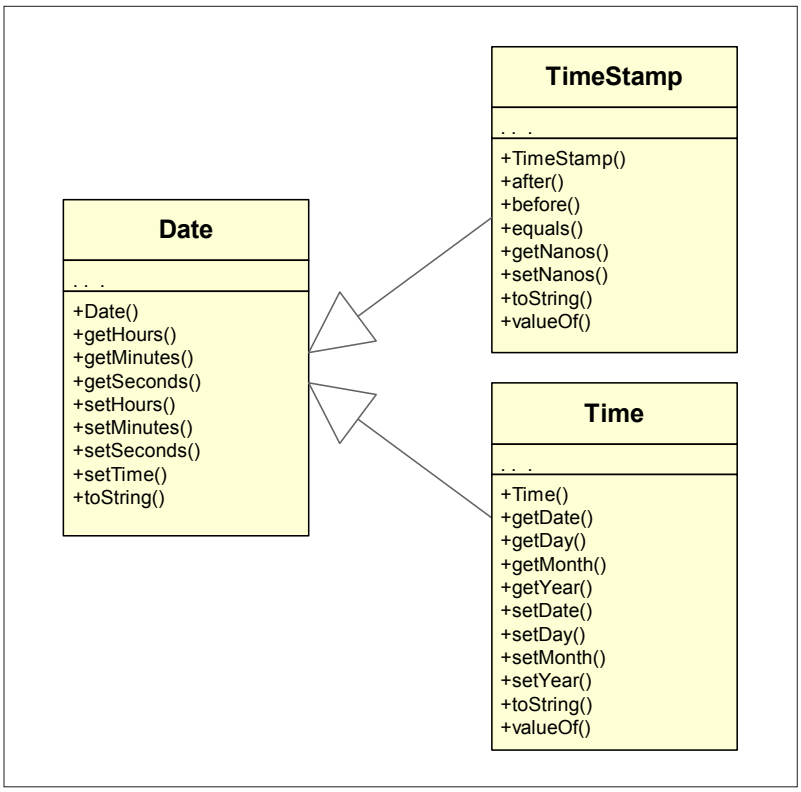
La generalizzazione è una relazione tassonomica, transitiva e antisimmetrica. La proprietà transitiva implica che se una Classe C generalizza (ossia eredita) una Classe B, la quale a sua volta generalizza Classe A, ne segue che C generalizza anche A. In parole semplici, un discendente (nell'esempio un nipote C) eredita a sua volta quanto il genitore (B) ha ereditato dai suoi antenati (a meno che il genitore non dissipi tutto il patrimonio…). La proprietà antisimmetrica sancisce che se A eredita da B, non è assolutamente vero (anzi è impossibile) il contrario.

Nel disegno di sistemi orientati agli oggetti (o component based), tipicamente l'identificazione del comportamento condiviso da diverse entità permette di estrarne e centralizzarne una versione generalizzata, di organizzare le varie classi similari secondo un'opportuna gerarchia, di realizzare il polimorfismo e il trattamento astratto. Questo approccio, se correttamente applicato, permette di realizzare modelli migliori e architetture più semplici. Come al solito però, il principio reca con sé tutta una serie di trabocchetti nei quali è facile incorrere. Lungi dal ripetere per l'ennesima volta le varie problematiche (per le quali si rimanda al Capitolo 6), si tenga presente la limitazione dell'utilizzo dell'ereditarietà: una volta stabilita una gerarchia, in qualche modo si "cementano" le varie classi in tale organizzazione.

La relazione di generalizzazione viene visualizzata attraverso un segmento congiungente la sottoclasse alla superclasse, con un triangolo vuoto posto all'estremità di quest'ultimo elemento.

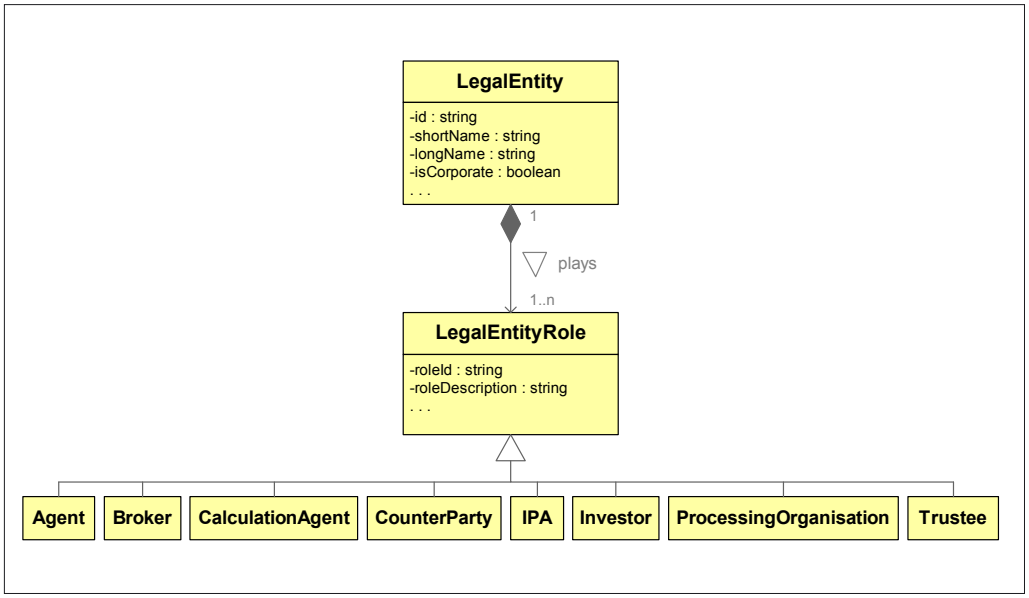
Si consideri un tipico sistema bancario con particolare riferimento all'area Treasury. In essa sono presenti diverse entità rappresentanti gli "attori" del business bancario. Per

Figura 7.17 — *Struttura gerarchica di* `Date`*,* `Time` *e* `Timestamp` *del package* `java.sql`*. Come si può ben notare sia* `Time`*, sia* `Timestamp` *sono realizzate specializzando la classe* `Date`*.*



esempio ci sono le *Counter Party* (terze parti), ossia i clienti della banca ai quali vendere (o meglio, con i quali scambiare) i prodotti finanziari, le *Processing Organisation* (organizzazioni di processo), che sono i dipartimenti di una banca demandati alla gestione dei trade stipulati con i clienti (Counter Party), i *Broker* (agenti) che si occupano di facilitare il commercio (principalmente relativo a scambi di valuta) tra la banca e i clienti, ecc. Tutte queste entità hanno un certo insieme di comportamento e struttura comune, con in più peculiari specializzazioni. Per esempio le Processing Organisation gestiscono i *Book* utili per organizzare i *Trade*, i quali contengono tutta una serie di informazioni molto importanti: il cliente, l'organizzazione di processo, il broker (il quale pretende di ricevere la provvigione, nota come *brokerage*, per la stipula del contratto), ecc. Pertanto, a prima vista si potrebbe trattare del classico esempio di generalizzazione: vi sono una *Legal Entity* che rappresenta il comportamento generico e tante specializzazioni, una per ogni ruolo.

Figura 7.18 — *Modello mostrante i diversi ruoli "interpretabili" da un'entità legale nel sistema business di una banca.*



Pur trattandosi di un modello molto semplice e facilmente comprensibile, è sfortunatamente del tutto inadeguato per diversi motivi. Per esempio, alcune *Legal Entity* possono assumere più ruoli: in qualche modo le relative istanze devono trasmutare durante il loro ciclo di vita. Situazioni di questo tipo si prestano a essere modellate sostituendo l'ereditarietà con un'apposita composizione. Resta il fatto che comunque è opportuno specializzare i ruoli, perché alcuni di essi partecipano a particolari relazioni con comportamenti specifici (*cfr* fig 7.18). In fig. 7.19 è mostrato un esempio tratto dal package `java.security`.

La classe astratta `Permission`, rappresenta la base predefinita per la gestione degli accessi a una risorsa del sistema. Tale classe implementa le interfacce `Serializable` e `Guard`. La prima è necessaria per consentire alle istanze delle specializzazioni di `Permission` la persistenza del proprio stato. Si può notare che l'interfaccia `Serializable` non dichiara alcun metodo. Ciò potrebbe sembrare abbastanza bizzarro: in realtà si tratta di un meccanismo utilizzato per raggruppare le classi il cui stato deve persistere (per questo motivo anche le discendenti devono implementare l'interfaccia `Serializable`).

L'interfaccia `Guard`, come suggerito dal nome, rappresenta una guardia le cui istanze sono oggetti utilizzati per proteggere l'accesso ad altri. A tal fine viene definito un unico metodo: `checkGuard`.

La classe `Permission`, in quanto astratta, non può essere istanziata direttamente (include dichiarazioni dei metodi astratti `equals`, `hashCode`, `implies` le cui specializzazioni devono definirne l'implementazione). A tal fine sono presenti le classi `UnresolvedPermission` (permessi non risolti all'atto dell'inizializzazione della policy), `AllPermission` (implica tutti gli altri permessi ossia viene disabilitata la sicurezza) e `BasicPermission` (Permessi basilari). Anche quest'ultima classe è astratta e pertanto valgono gli stessi discorsi della classe `Permission`. In questo caso la classe specializzante è `SecurityPermission`.

Tutti i metodi `newPermissionCollection` possono restituire una lista di permessi (in realtà la versione della classe `Permission` restituisce un valore `null`) per una determinata risorsa e quindi vi è un'evidente dipendenza dalla classe `PermissionCollection`. A dire il vero, poiché questa raccoglie una lista di permessi, a sua volta dipende dalla classe `Permission`.

Figura 7.19 — *Il diagramma mostra l'organizzazione dei permessi definiti nel package della sicurezza di Java* (`java.security`).

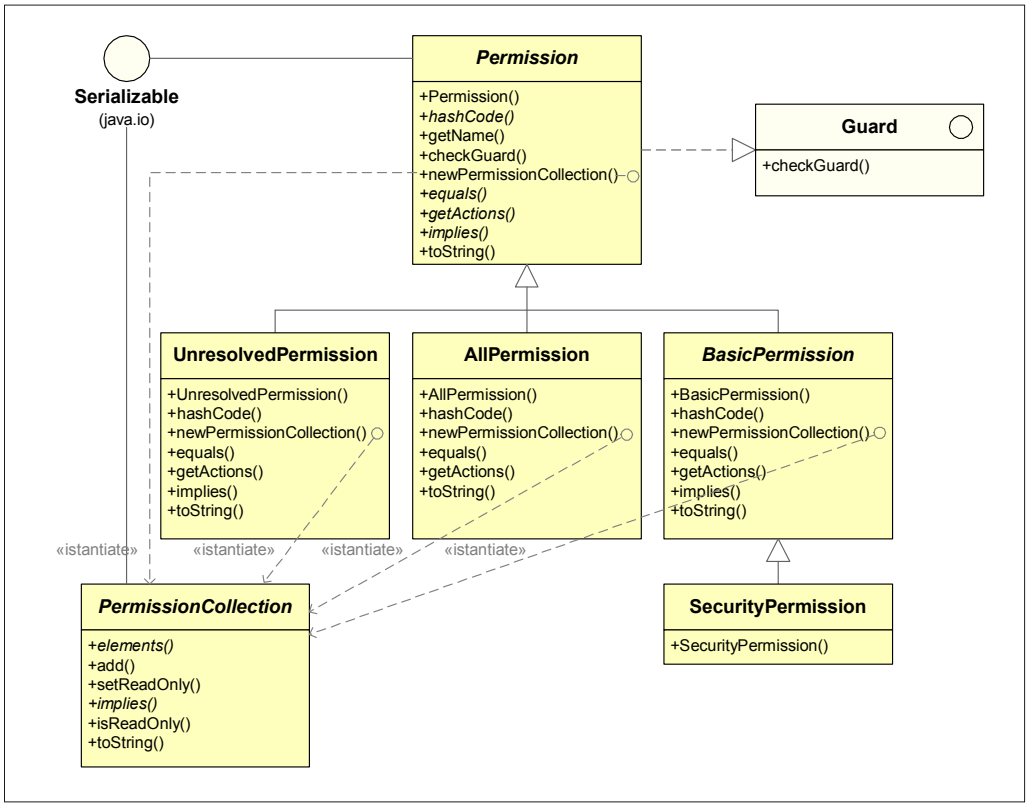
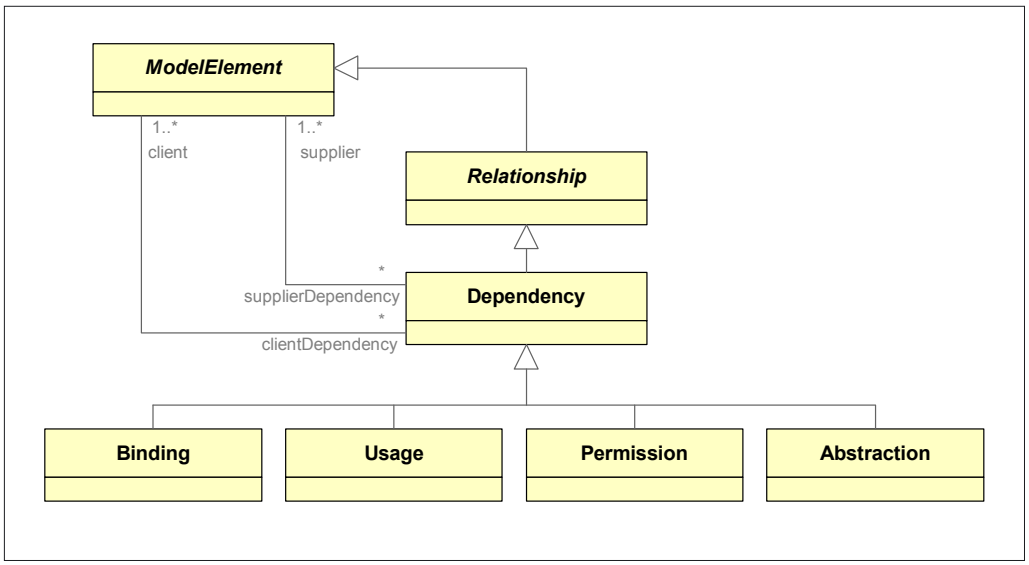


Figura 7.20 — *Esempio di relazione di Generalizzazione. Definizione formale della relazione di dipendenza (package* `Core` *del metamodello UML).*



Un esempio interessante di utilizzo della relazione di generalizzazione è fornito dalla definizione formale della relazione di dipendenza, illustrata nel precedente paragrafo, riportata nel package `Core` del metamodello dello UML (fig. 7.20).

Tutti gli elementi utilizzabili nei diagrammi dello UML discendono da uno stesso elemento: la metaclasse astratta `ModelElement` (elemento del modello). Si tratta di una tecnica utilizzata per disporre di un'elegante relazione gerarchica e cumulativa di tutti gli elementi dello UML. Una delle sottoclassi di `ModelElement` è la metaclasse `Relationship`, utilizzata per rappresentare legami tra elementi del modello. Dalla metaclasse `Relationship` discendono tutte le relazioni utilizzabili nei diagrammi UML. Una di queste relazioni è appunto la relazione di dipendenza (`Dependency`). In questo caso si tratta di una metaclasse concreta (non astratta) e quindi utilizzabile direttamente nel modello. Tale relazione è virtualmente utilizzabile per associare qualsiasi coppia di elementi dello UML, anche tra loro eterogenei. Teoricamente (bisogna sempre fare i conti con i tool) si potrebbe associare una classe con un use case per mezzo della relazione di dipendenza, per indicare magari che la classe deriva da requisiti specificati nello use case. Meglio ancora, si potrebbe associare un'interfaccia a uno use case per indicare da quale use case deriva la specifica di un componente. La metaclasse `Dependency` non prevede

restrizioni della semantica, in altre parole non sono presenti ulteriori vincoli. Comunque, per la costruzione dei vari modelli, tipicamente si preferisce utilizzare le specializzazioni di tale metaclasse, le quali prevedono una semantica più specifica e tutta una serie di restrizioni che ne caratterizzano l'utilizzo. Ciò è dovuto alla necessità di condizionarne l'utilizzo alle regole semantiche della specializzazione. Per esempio, nella relazione di Binding l'elemento client deve essere necessariamente conforme al tipo dell'elemento supplier, il numero degli argomenti deve coincidere con quello dei parametri, e così via.

Una stessa relazione di dipendenza può coinvolgere diversi supplier e ciascuno di essi può essere coinvolto in svariate relazioni di dipendenza. Lo stesso discorso vale per i client: ancora relazione *n* a *n*.

Per terminare, la relazione di dipendenza prevede quattro specializzazioni predefinite: Abstraction, Binding, Permission e Usage.

Associazione binaria

Un'associazione è una relazione strutturale tra due o più classificatori descrivente connessioni tra le relative istanze.

> Come per le altre relazioni, anche la sfera di applicazione della relazione di associazione non è circoscritta ai soli diagrammi delle classi, bensì è utilizzabile in diversi contesti. Per questa ragione nell'illustrazione si fa riferimento al concetto generale di classificatore e non a quello più specifico di classe. Volendo circoscrivere la spiegazione ai soli diagrammi delle classi è sufficiente operare una sostituzione dei nomi: ad ogni occorrenza del termine classificatore è necessario inserire il vocabolo classe. Così per esempio, si ha che una associazione binaria è una relazione strutturale che associata istanze di una classe a quelle di un'altra.

In base al numero degli elementi coinvolti nella relazione si hanno diverse "specializzazioni" della relazione: associazione unaria, binaria e *n*-aria. Il caso decisamente più frequente è costituito dall'associazione binaria che, come lecito attendersi, coinvolge due classificatori, in particolare specifica che oggetti di un tipo sono collegati a quelli di un altro. Pertanto, data un'associazione binaria, è possibile navigare dagli oggetti di un tipo a quelli dell'altro. L'associazione binaria è così importante, da essere stata definita da Rumbaugh come la "colla" che unisce il sistema. In effetti, senza la relazione di associazione le istanze delle classi sarebbero destinate a una vita isolata. Una "degenerazione" dell'associazione binaria è costituita dalla relazione unaria detta anche autoassociazione (*self association*) in cui un classificatore è associato con sé stesso.

Graficamente un'associazione binaria è rappresentata attraverso un segmento che unisce i due classificatori connessi. Tipicamente alle associazioni è assegnato un nome per specificare la natura della relazione stessa e, al fine di eliminare ogni possibile fonte di

Figura 7.21 — *Esempio di relazione binaria.*

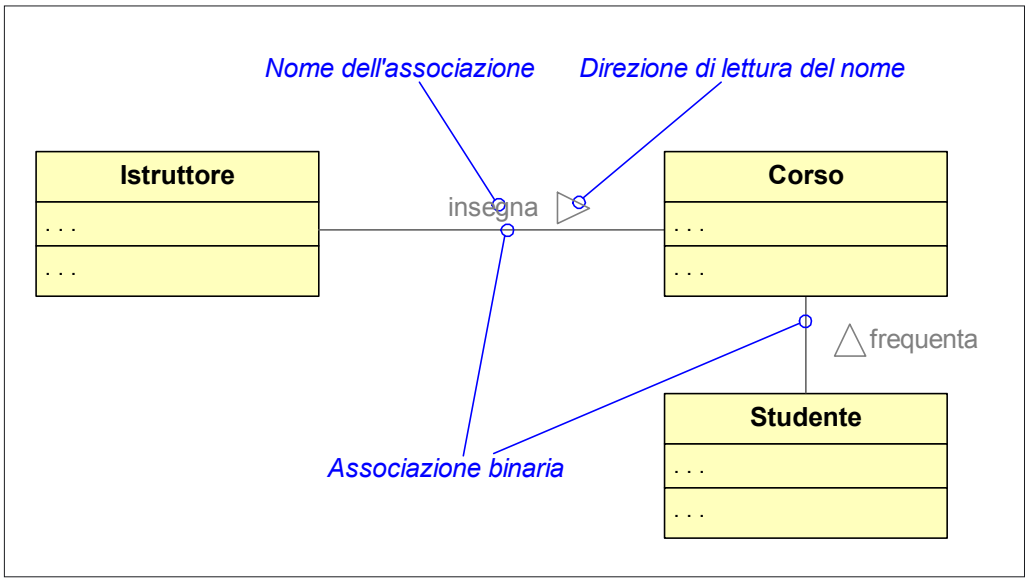
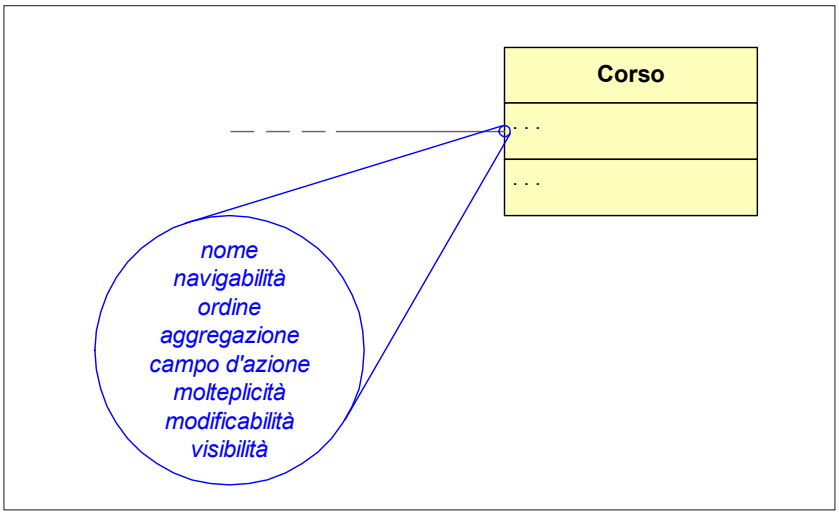


Figura 7.22 — *Illustrazione informale degli attributi della metaclasse* `AssociationEnd`. *Da notare che alcuni attributi sono di tipo enumerato, come* `aggregazione`, *i cui valori sono* `none`, `aggregate` *e* `composite`.



ambiguità, spesso l'etichetta del nome viene corredata da una freccia che ne indica l'ordine di lettura (*cfr* fig. 7.21). Nel caso del modello a oggetti del dominio, i nomi delle associazioni concorrono a illustrare le business rule. Nome e direzione devono essere visualizzati nei pressi dell'associazione a cui appartengono, ma non vicini ai classificatori, per non essere confusi con i ruoli.

Fin qui l'associazione binaria è stata descritta nella sua forma essenziale: in realtà prevede tutta una serie di ornamenti (*adornments*) che permettono di illustrarne proprietà specifiche. In particolare, consentono di specificare importanti caratteristiche con cui i classificatori partecipano nelle associazioni. Idealmente, l'intorno del punto dello spazio in cui una terminazione della relazione di associazione incontra il classificatore è sede di tutta una serie di proprietà. Nel metamodello tali proprietà sono rappresentate dagli attributi di un'apposita metaclasse: `AssociationEnd` (*cfr* fig. 7.22).

Navigabilità

Quando la relazione di associazione tra due classi è mostrata senza alcuna direzione ne segue che, nell'ambito della relazione, entrambe le classi sono navigabili, quindi, data un'istanza di una classe, è possibile transitare nelle istanze dell'altra a cui è associata e viceversa. L'associazione così definita in fase di disegno, non rappresenta però una situazione auspicabile poiché genera un forte accoppiamento tra le classi partecipanti alla relazione: entrambe le classi che partecipano alla relazione devono prevedere un riferimento all'altra classe. Ciò permette di navigare dalle istanze di un tipo a quelle dell'altro. Se inoltre il tipo di relazione è *n* a *n*, allora la situazione diventa ancora meno piacevole: entrambe le classi devono prevedere una lista di riferimenti alle istanze dell'altra.

In fase di disegno, quando si hanno maggiori informazioni sul sistema, è opportuno ponderare bene l'assenza di navigabilità tra le classi di un'associazione, sebbene spesso non sia possibile farne a meno... Diversamente, invece, nei modelli a oggetti generati nelle fasi precedenti del processo di sviluppo — modelli a oggetti del dominio, del business e di analisi — non sempre è opportuno prendere decisioni che possano condizionare prematuramente la realizzazione del sistema. Pertanto in tali modelli, a meno di casi evidentissimi, non è il caso di investire tempo nello stabilire i vincoli di navigabilità delle varie relazioni di associazione.

Qualora in una relazione di associazione non si voglia permettere alle istanze di una classe di "vedere" quelle dell'altra (essenzialmente, di invocarne i metodi), nella rappresentazione della relazione è necessario inserire una freccia indicante il verso di percorrenza. Pertanto gli oggetti istanza della classe posta nella coda dell'associazione (`navigabilità = false`)

possono invocare i metodi e accedere agli attributi pubblici delle istanze della classe puntata dalla freccia (`navigabilità = true`), mentre non è possibile il contrario (*cfr* fig. 7.23). In termini pratici, ciò implica la necessità di memorizzare nella classe che non permette la propria navigazione il riferimento (o i riferimenti) alle istanze dell'altra classe. La presenza del vincolo di navigabilità è preferibile poiché diminuisce l'accoppiamento tra classi.

Un errore comunemente commesso, specie nella realizzazione di modelli a oggetti del dominio o del business, consiste nell'utilizzare la navigabilità di una relazione di associazione per indicare il verso di lettura del nome dell'associazione. Ciò è dovuto al fatto che molti tool non prevedono la possibilità di visualizzare la freccia indicante il verso di lettura del nome delle associazioni. Chiaramente si tratta di un errore in quanto le informazioni connesse con la direzione dell'associazione hanno un notevole impatto semantico sull'implementazione del modello.

In una relazione di associazione è un nonsenso avere entrambe le classi non navigabili, mentre la situazione opposta è generalmente evidenziata attraverso un segmento senza direzione.

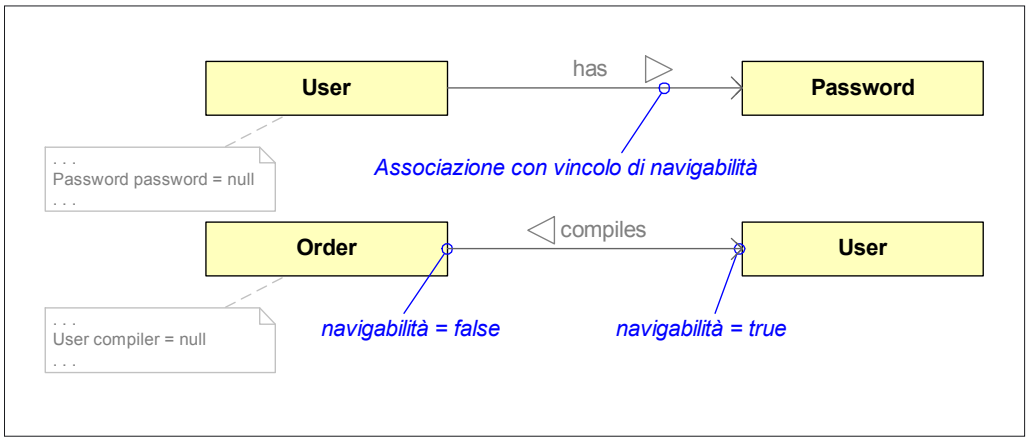
Nella fig. 7.23 sono mostrate due diverse situazioni. Nella prima, dato uno specifico `User` è possibile risalire alla relativa `Password` (nell'implementazione della classe `User` deve essere presente un membro di tipo Password), mentre data specifica istanza di `Password` non è possibile scoprirne il proprietario. La seconda relazione mostra il caso in cui dato un ordine è possibile navigare all'utente che lo ha compilato, mentre non è vero il contrario. In questo caso si è voluto mostrare come la direzione del nome dell'associazione non debba necessariamente concordare con il vincolo di navigabilità, sebbene possa avere senso riportare un nome coerente con la navigabilità.

La freccia della relazione di associazione ha dunque diverse ripercussioni: definisce il senso della navigabilità, il che tende a ripercuotersi sia sull'organizzazione dell'interfaccia utente, sia sulla realizzazione in termini di componenti del sistema, ecc..

La presenza del vincolo di navigabilità in una relazione di associazione sancisce che da una istanza di una classe (per esempio `Utente`, fig. 7.23) non sia possibile navigare direttamente in quelle dell'altra classe associata (gli ordini compilati dall'utente). Però ciò non significa che non sia possibile in assoluto, bensì è possibile risalire comunque all'istanza associata, effettuando però un percorso più complesso e coinvolgendo istanze di diverse altre classi. Quindi il vincolo di navigabilità può avere notevoli ripercussioni sulle performance di alcuni servizi forniti dal sistema.

Considerate le varie problematiche, si potrebbe addivenire alla frettolosa conclusione che, sebbene sia una pratica assolutamente non auspicabile, è meglio non specificare vin-

Figura 7.23 — *Esempi di navigabilità.*



coli di navigabilità. Purtroppo, questa regola semplicistica e non priva di effetti collaterali (come l'elevato accoppiamento tra le classi) non sempre è applicabile: qualora si utilizzino sistemi component based è necessario decidere accuratamente le navigabilità al fine di poter ripartire le classi in opportuni componenti.

Molteplicità

Poiché una associazione è relazione strutturale tra gli oggetti istanze delle classi relazionate, è importante specificare, per ogni istanza di una classe, a quanti oggetti dell'altra può essere connessa e viceversa. Queste informazioni sono definite "molteplicità di un ruolo di associazione" o più semplicemente "molteplicità" (*multiplicity*).

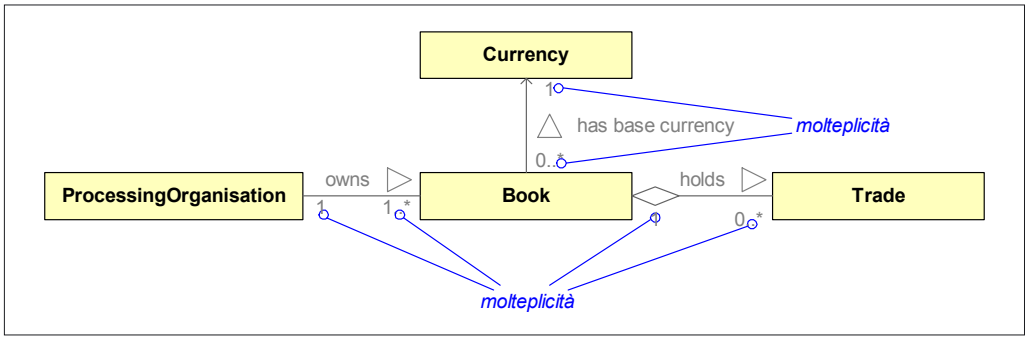
La definizione formale afferma che una molteplicità specifica l'**elenco** dei valori delle possibili cardinalità che un insieme può assumere.

Per quanto concerne la notazione non c'è nulla da aggiungere rispetto a quanto visto in precedenza per gli attributi e i metodi: stringa di intervalli di interi separati da virgola. Anche le convenzioni rimangono inalterate: un carattere asterisco (`*`) posto come limite superiore denota un intervallo infinito (`* = 0..*`), qualora sia riportato un unico valore, ciò indica che l'intervallo è degenerato in un solo valore e quindi i limiti inferiori e superiore coincidono, ecc.

Esempi di cardinalità sono: "`0..1`", "`1`", "`0..*`", "`*`", "`2, 4..6, 8..*`". L'ultimo esempio evidenzia che la molteplicità, nel caso più generale, è data da una lista di intervalli e non da solo uno.

Nei modelli richiesti nelle prime fasi del ciclo di vita del software (modello a oggetti del dominio e del business), le molteplicità sono molto importanti poiché concorrono a illustrare le regole del business del sistema reale oggetto di studio, mentre nel modello di

Figura 7.24 — *Esempi di molteplicità. Una* `ProcessingOrganisation` *gestisce diversi* `Book` *e ciascuno di essi appartiene a una sola* `ProcessingOrganisation`*. I* `Book` *sono costituiti da diversi* `Trade` *e ciascuno di questi appartiene a un preciso* `Book` *in funzione della valuta base.*



disegno forniscono importanti informazioni di carattere implementativo. Per esempio, nel modello a oggetti del dominio di una banca (fig. 7.24), l'affermazione che un `Book` è basato su una sola valuta fornisce un'informazione molto importante: ogni `ProcessingOrganisation` (dipartimenti in cui una banca è suddivisa) deve possedere diversi `Book` in cui organizzare i `Trade` (almeno uno per ogni valuta considerata).

In un modello di disegno l'affermazione precedente indica che nell'implementazione della classe `Book` è necessario un unico attributo per memorizzare il riferimento alla relativa istanza della classe `Valuta`. Se invece fosse stato possibile associare un `Book` a diversi oggetti `Currency`, allora sarebbe stato necessario conservare una lista di riferimenti.

Ruoli associazione

Ogni qualvolta un'entità prende parte a un'organizzazione, recita un determinato ruolo; similmente, anche le classi, partecipando a una relazione, svolgono uno specifico ruolo. Questo è espresso per mezzo di una stringa il cui scopo è renderne esplicita la semantica. Qualora si decida di visualizzare il ruolo, lo si deve collocare nei pressi dello spazio in cui il segmento dell'associazione incontra la classe a cui si riferisce. La selezione di tale spazio non è casuale (si consideri la fig. 7.22): rappresenta la sede ideale di una particolare classe (`AssociationEnd`, terminale di associazione) del metamodello. Tale classe è utilizzata per contenere gli attributi atti a descrivere le proprietà che le classi possono manifestare partecipando a una determinata associazione. In altre parole, questi attributi non appartengono alla classe né tanto meno all'associazione, bensì a entrambe: l'associazione e la classe che vi partecipa. Quindi nel contesto di un'associazione si hanno tanti oggetti "terminali di associazione" quante sono le classi coinvolte. Per esempio in un'autoassociazione si hanno due terminali di associazione, così come in un'associazione binaria.

`AssociationEnd` è una classe del metamodello, pertanto, quando si realizzano i vari diagrammi delle classi, il suo utilizzo è del tutto trasparente. È interessante però sapere che quando si inserisce il nome di un ruolo, una molteplicità, ecc. non si fa altro che impostare gli attributi di un oggetto di questo tipo opportunamente istanziato dal tool di modellazione che si sta utilizzando. L'attributo ruolo è ereditato dalla classe genitore di tutti gli elementi (`ModelElement`), nella quale viene denominato, genericamente *name* e, tipicamente, possiede anche funzioni di identificatore: individua univocamente un terminale di associazione.

Tipicamente l'utilizzo del ruolo è alternativo al nome dell'associazione, nulla però vieta di specificarli entrambi. Nella pratica si utilizzano una serie di semplici regole pratiche atte a semplificare la selezione di quale alternativa utilizzare. In un'autoassociazione è preferibile specificare i ruoli poiché chiariscono il legame con cui una classe è associata con sé stessa (*cfr.* fig. 7.25).

Nei diagrammi prodotti come risultato delle prime fasi di analisi dei requisiti è preferibile specificare i nomi delle associazioni poiché concorrono a definire le regole appartenenti al dominio che il sistema dovrà automatizzare. Un'eccezione si ha qualora sia necessario utilizzare l'OCL (Object Constraint Language) per specificare vincoli, invarianti, pre- e postcondizioni ecc. I nomi dei ruoli sono utilizzati per navigare le varie associazioni. Per esempio, nel caso in cui un'istanza della Classe A possa essere associata, al più, a una della Classe B, l'espressione `A.rolename` individua la particolare istanza della Classe B a cui

Figura 7.25 — *Ruoli in una autoassociazione. Un dipendente è soggetto a un solo responsabile, mentre quest'ultimo gestisce diversi subordinati.*

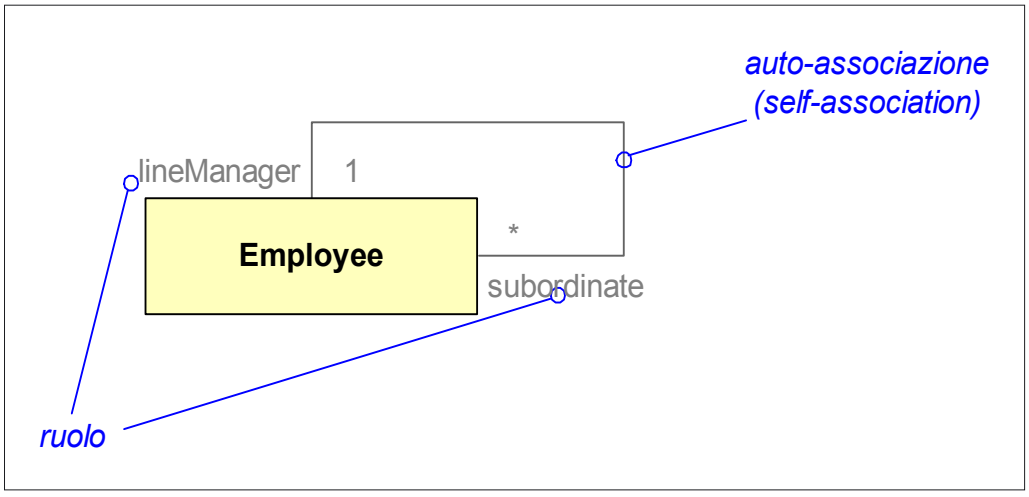
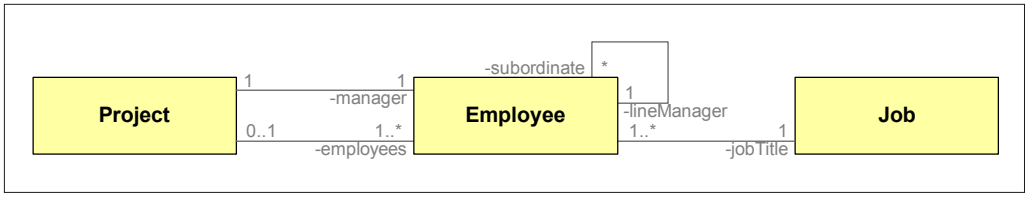


Figura 7.26 — *Utilizzo dei ruoli per rappresentare la navigabilità in OCL. La porzione di modello mostrata è stata volutamente semplificata al fine di non introdurre elementi (classi associazione) non ancora illustrati. In particolare si può notare come un progetto impieghi diversi dipendenti e un unico manager. Qualora un dipendente sia allocato a un progetto non può lavorare contemporaneamente ad altri. Chiaramente ci sono dipendenti le cui funzioni non sono relative ai progetti e quindi non sono allocati a nessuno.*



l'oggetto A è legato. Nella fig. 7.26, data un'istanza di progetto, per specificare il relativo manager è necessario specificare la navigabilità `project.manager`. Nel caso in cui la molteplicità sia "a molti", allora la navigabilità è ancora valida, però è necessario utilizzare funzioni dell'OCL più complesse.

Dato il diagramma di fig. 7.26 è possibile asserire il seguente vincolo:

```
Project.manager.jobTitle.description = #Manager
```

Un manager di un progetto è un dipendente il cui titolo è appunto quello di manager (si assume che la classe `Job` abbia un attributo `description`). Sempre con riferimento al diagramma di fig. 7.26, è possibile definire la funzione che permette di calcolare il costo annuale di un progetto relativo all'impiego delle risorse umane.

```
Project.getAnnualEmployeesCost()
    = Project.manager->getAnnualIncome()
    + Project.employees->getAnnualIncome()->sum()
```

I ruoli dell'associazione sono poi preferiti nel modello di disegno in quanto specificano l'attributo utilizzato per realizzare la relazione. Per esempio, nel caso mostrato nel frammento di fig. 7.26, l'implementazione della classe `Employee` disporrebbe di un attributo membro `jobTitle:Job`. Ancora, nel caso in cui fosse necessario memorizzare tutti i dipendenti assegnati a uno specifico progetto, la relativa classe (`Project`) disporrebbe di un attributo membro privato (di tipo vettore o qualsiasi collezione equivalente) denominato `employees`, i cui elementi sono istanze della classe `Employee`.

Ordinamento

Nel caso in cui due, o più, classi siano associate per mezzo di una relazione con molteplicità diversa da *1* a *1*, ossia almeno un'istanza di una classe possa essere connessa con un

opportuno insieme di oggetti istanze dell'altra, può verificarsi il caso in cui sia necessario ordinare, secondo un preciso criterio, tali relazioni. Qualora ciò avvenga, la condizione è evidenziata riportando la parola chiave `ordered`, racchiusa da parentesi graffe, (`{ordered}`) nell'opportuno terminale di associazione. Un esempio è l'associazione binaria tra `User` e `Password` mostrata in fig. 7.23: potrebbe essere necessario dover archiviare diverse parole chiavi utilizzate da un utente, magari perché la politica di sicurezza prevede che uno stesso utente non possa riutilizzare le precedenti *n* password esercitate in passato. In tal caso, tra l'altro, sarebbe opportuno inserire nella classe `Password` due attributi atti a tenere traccia del periodo di utilizzo della relativa parola chiave e quindi utilizzare il limite inferiore come criterio di ordinamento: la più recente nella prima posizione, la seconda più recente in seconda e così via. Un altro esempio è dato dalla definizione formale del tipo enumerato (`Enumeration`), in tal caso l'ordine dei relativi valori di cui è composto assume un'importanza fondamentale in quanto è utilizzato come criterio di mapping (*cfr.* fig. 7.27). Il segmento della relazione dotato del diamante pieno rappresenta una particolare associazione illustrata successivamente. La definizione nel metamodello UML è simile: le classi però non dispongono di nessun attributo poiché lo ereditano dalla classe `ModelElement`, dalla quale discendono tutte le altre. Per default l'insieme delle relazioni non è ordinato e quindi non è necessario specificare alcuna parola chiave. Eventualmente è possibile utilizzare altri valori definiti dal modellatore per indicare determinate condizioni, come per esempio che l'insieme delle relazioni sono riordinate `{sorted}`.

Da notare che sia l'informazione relativa all'ordinamento, sia quella attinente alla modificabilità sono utilizzate molto raramente. Tipicamente trovano applicazione nei modelli a maggiore grado di formalità, come per esempio il metamodello UML. Il problema è sempre lo stesso: pochissimi tecnici ne conoscono la semantica e sanno interpretare correttamente questi ornamenti. Pertanto è sempre cosa buona illustrarne le implicazioni attraverso ulteriore formalismo (linguaggio naturale) e quindi l'importanza di specificarle direttamente nel modello passa in secondo piano.

Figura 7.27 — *Definizione del tipo enumerato.*

Modificabilità

Un altro insieme di vincoli che è possibile specificare in un'associazione è relativo alla modificabilità (*changeability*) dei legami che le istanze delle classi coinvolte nell'associazione instaurano. Tali legami rappresentano la realizzazione dell'associazione stessa. Il particolare vincolo è associabile a ogni lato dell'associazione e quindi, nel metamodello UML, è rappresentato dall'apposito attributo (`changeability`) presente nella solita classe `AssociationEnd` (*cfr.* fig. 7.22). I valori ammessi sono tre:

- `changeable` (modificabile): si tratta del valore di default; semplicemente rappresenta la condizione in cui non sussista alcun vincolo e quindi le relazioni possono essere modificate liberamente.

- `frozen` (congelato): questo vincolo specifica che, dopo la creazione dell'oggetto, nessuna relazione può essere aggiunta per mezzo dell'esecuzione di un'operazione nella classe sorgente della relazione. A meno di diverse indicazioni, questo vincolo non ha alcun effetto sulla classe destinataria che quindi può aggiungere liberamente altre relazioni;

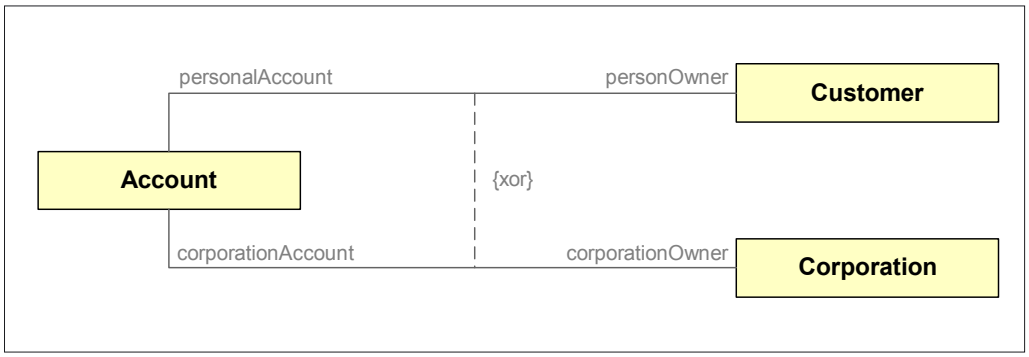
- `addOnly` (sola aggiunta): la presenza di questo vincolo sancisce che possono essere aggiunte relazioni ogniqualvolta se ne abbia la necessità, ma, una volta create, queste non possono essere più rimosse eseguendo un'operazione nella classe sorgente. Come nel caso precedente, il vicolo di per sé non ha alcun effetto sulla classe destinazione.

Campo d'azione destinatario

L'illustrazione dell'elenco delle caratteristiche specificabili in un'associazione si esaurisce con il campo d'azione delle associazioni. In modo del tutto analogo a quanto sancito per gli attributi e i metodi, anche alle associazioni (che in ultima analisi non sono altro che particolari attributi) è possibile specificare il campo d'azione (*scope*). Le alternative previste sono due: istanza (*instance*) e classificatore (*classifier*). Nel primo caso, quello di default, si ha che ogni istanza della classe sorgente è associata, in funzione della molteplicità, con una o più istanze della classe destinazione. Il valore *classifier* (associazione statica) implica una relazione tra un'istanza della classe destinazione e il classificatore sorgente (non un oggetto). Ciò è comprensibile se si considera che la relazione non deve essere memorizzata in ogni istanza della classe, quindi appartiene al classificatore stesso ed esiste prima della creazione di ogni suo oggetto. In sostanza si tratta di un'associazione che viene stabilita a tempo di "disegno" e non può più variare nel tempo.

Chiaramente il secondo caso non rappresenta una situazione molto ricorrente, sebbene in particolari circostanze (come per esempio il pattern Singleton) sia molto utile. In linea

Figura 7.28 — *Esempio di relazione* xor.



di massima, non si tratterebbe di una buona pratica OO in quanto prevede la conoscenza di informazioni a carattere globale.

Associazione *xor*

Una associazione *xor* (*xor-association*) rappresenta una situazione in cui, in un determinato istante di tempo per una specifica istanza della classe, può verificarsi solo una delle potenziali associazioni che una stessa classe può instaurare con altre.

L'associazione *xor* viene rappresentata graficamente tramite un segmento che unisce le associazioni (due o più) soggette al vincolo, con evidenziata la stringa `{xor}` (*cfr.* fig. 7.28). Chiaramente, le diverse associazioni devono condividere la medesima classe sorgente.

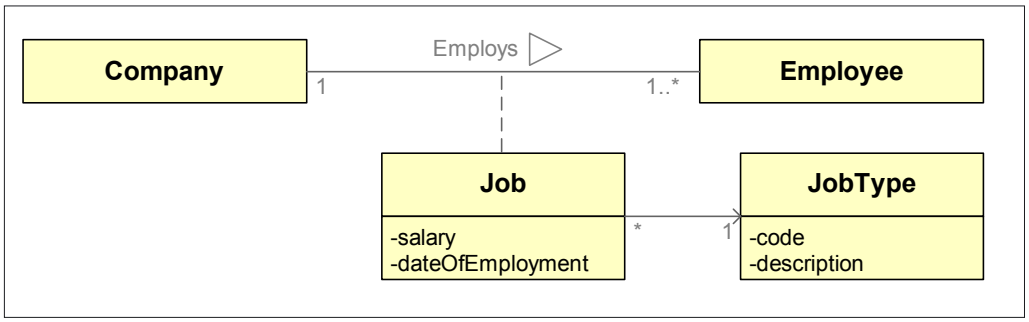
Qualora si utilizzi la relazione *xor*, bisogna porre attenzione ad alcune logiche conseguenze. In primo luogo la molteplicità minima delle classi destinatarie delle relazioni soggette al vincolo *xor* deve essere necessariamente zero. In altre parole deve essere prevista la possibilità che le relative istanze possano non essere coinvolte nella relazione. Poi, i nomi dei ruoli delle classi di cui sopra devono essere distinti per poter, eventualmente, esprimere la navigabilità.

Si tratta di una relazione di importanza molto relativa (è sempre possibile specificare il vincolo per mezzo del linguaggio naturale o di opportune clausole OCL) che però risulta particolarmente efficace ed elegante grazie al grande valore di sintesi offerto dalla parola *xor*. Lo stesso metamodello UML ne fa uso in diversi diagrammi.

Associazione o dipendenza?

Spesso alcuni tecnici tendono a confondere quando utilizzare le relazioni di dipendenza e quando quelle di associazione. Questa confusione purtroppo è abbastanza frequente ed è riscontrabile anche dall'analisi di diagrammi riportati su siti molto importanti.

Figura 7.29 — *Esempio dell'utilizzo della association class tra l'azienda e i dipendenti.*



Per esempio è possibile osservare classi client collegate a classi Singleton attraverso una relazione di associazione. Ciò è particolarmente sbagliato specie se si pensa che uno dei vantaggi nel ricorrere al pattern Singleton consiste nel permettere alle varie classi di ottenere un riferimento a quella Singleton ogniqualvolta se ne abbia bisogno, senza aver bisogno di mantenere il riferimento stesso.

Molto spesso la regola semplice e grossolana utilizzata è che se una relazione è comune a tutta la classe (il relativo attributo è specificato come privato alla classe), allora rappresenta un'associazione; altrimenti siamo in presenza di una dipendenza. Questa regola molto semplicistica trascura il fatto che spesso alcune variabili, per questioni esclusivamente di carattere implementativo, vengono dichiarate una sola volta e riutilizzate in tutta la classe.

Qualora si abbiano dei dubbi circa quale relazione utilizzare, è necessario ricordare che la differenza sostanziale consiste nel fatto che mentre una relazione di associazione è una relazione strutturale (quindi "persistente"), che evidenzia classi semanticamente correlate, la relazione di dipendenza ha un carattere transitorio (al netto di ottimizzazioni), un legame che si instaura (o almeno così dovrebbe essere) temporaneamente, per il lasso di tempo necessario per fruire di un servizio, per creare un oggetto, ecc., per poi perdere di significato.

Classe associazione

In un'associazione tra classi spesso si verifica la situazione in cui la relazione stessa possieda proprietà strutturali e comportamentali (attributi, operazioni e riferimenti ad altre classi). In tali circostanze è possibile ricorrere all'utilizzo della classe associazione

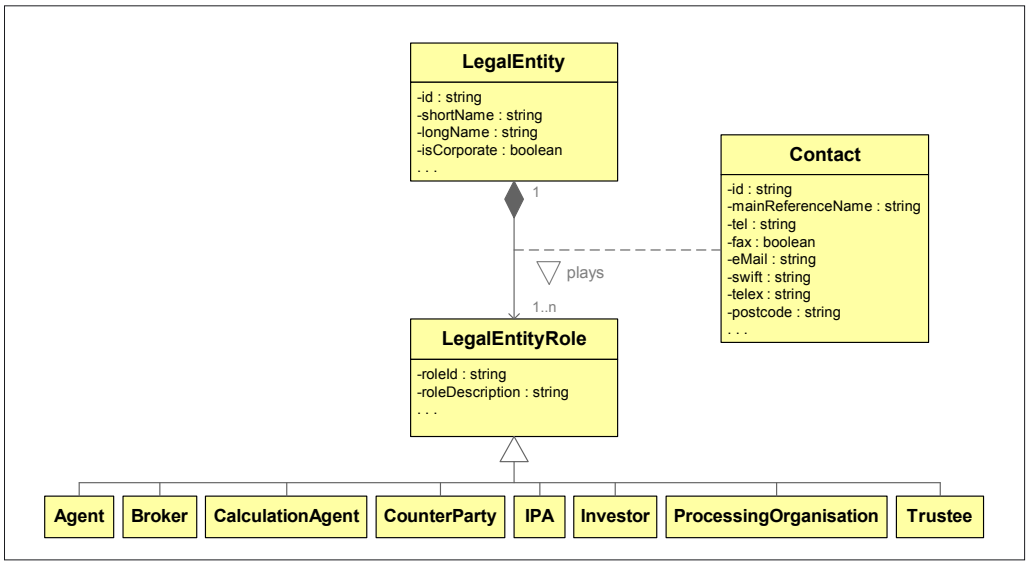
(*association class*) che, come suggerito dal nome, possiede contemporaneamente proprietà di classe e di associazione.

L'esempio che generalmente è presentato è relativo alle mansioni che un dipendente espleta in un'azienda. Un'organizzazione che impiega diverse persone viene modellata attraverso le classi `Company` e `Person`, legate dalla relazione `employees` (fig. 7.29). Tipicamente però non è sufficiente sapere quante e quali persone lavorino per una data azienda, ma è anche importante disporre di ulteriori informazioni, quali per esempio, il ruolo svolto dal dipendente, l'anno di assunzione, ecc. Queste proprietà non appartengono né alla classe `Company`, né a quella `Person`, bensì, al legame tra le due. Ecco quindi che la relazione necessita di ulteriori proprietà.

Una association class è rappresentata graficamente attraverso il normale formalismo previsto per le classi con, in aggiunta, un segmento tratteggiato di connessione con la relazione che la origina. Il punto di congiunzione (per non far dispiacere a nessuna delle due classi relazionate), dovrebbe appartenere a un intorno del centro del segmento rappresentante l'associazione.

Nel diagramma in fig. 7.30 è mostrato un frammento del modello utilizzato per rappresentare le "entità legali" coinvolte nel business bancario. Una stessa `LegalEntity` può "recitare" diversi ruoli e, per ciascuno di essi, è necessario disporre dei dati relativi ai riferimenti interni all'organizzazione. In altre parole, la situazione generale è che, ogni `LegalEntity` dispone di un riferimento diverso per ciascun ruolo esercitato (per esempio

Figura 7.30 — *Esempio di utilizzo della relazione di associazione.*



il sig. Tizio è il riferimento della *Legal Entity X* nel ruolo di *Agent*, il sig. Caio è il riferimento della stessa *Legal Entity X* ma questa volta per il ruolo di *Processing Organisation*, ecc.). Quindi si ha il caso di una relazione con attributi e dunque la relazione di associazione.

Nel ricorso alla classe associazione è necessario tenere bene a mente un vincolo implicito, spesso sottovalutato, di una certa importanza: per ogni singola associazione esiste **una e una sola** istanza della classe associazione. Si riconsideri l'esempio dell'azienda e dei relativi dipendenti: come si può notare il modello presenta un'anomalia. La condizione in cui uno stesso dipendente interrompa il rapporto di dipendenza con un'azienda per poi riprenderlo in un secondo momento non è gestibile con tale versione del modello. Eventualmente, si potrebbe inserire una nuova istanza della classe `Employee`, con esattamente gli stessi dati, ma ciò determinerebbe la perdita della storia della collaborazione: le diverse istanze `Employee` relative allo stesso soggetto non sarebbero in alcun modo relazionate. Si consideri i problemi che ciò potrebbe generare, dovendo per esempio fornire dettagli relativi ai contributi pensionistici e similari. Il modello, per poter annoverare anche il caso predetto, dovrebbe quindi essere modificato come riportato nel frammento di diagramma di fig. 7.31.

Quindi, prima di utilizzare la classe associazione si tenga ben in mente il vincolo implicitamente inserito: una sola istanza dell'association class per ciascuna associazione.

Analizzando attentamente il modello di fig. 7.32 si può notare come l'utilizzo del meccanismo della eredità multipla tenda facilmente a generare il "problema del diamante" di cui si è parlato nel Capitolo 6.

Figura 7.31 — *Esempio del modello Company, Employee senza l'utilizzo della classe associazione.*

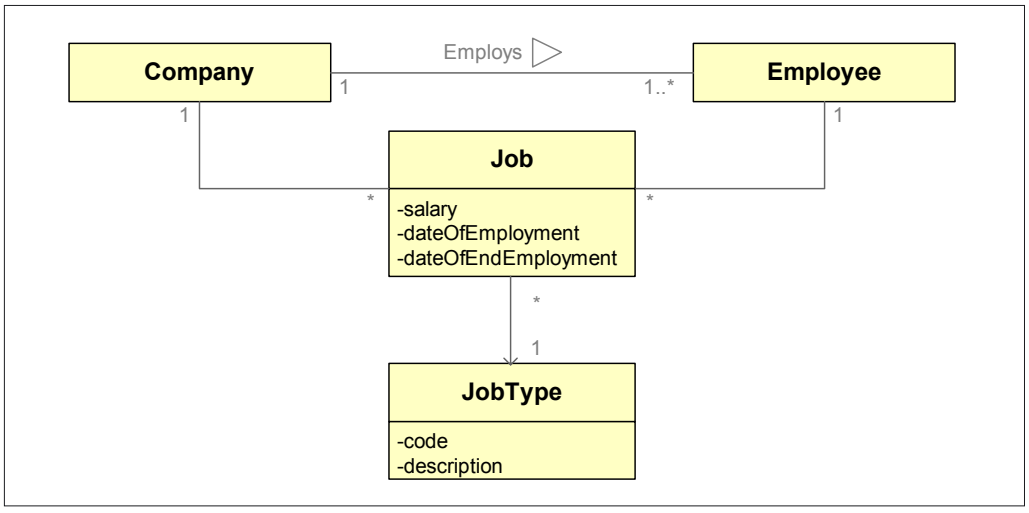


Figura 7.32 — *Porzione del metamodello UML relativo all'elemento AssociationClass del package Core – Relationship. Esso mostra come l'elemento AssociationClass sia definito attraverso una eredità multipla. Ciò gli consente di avere caratteristiche sia di classe (attributi, operazioni, relazioni, ecc.) sia di relazione.*

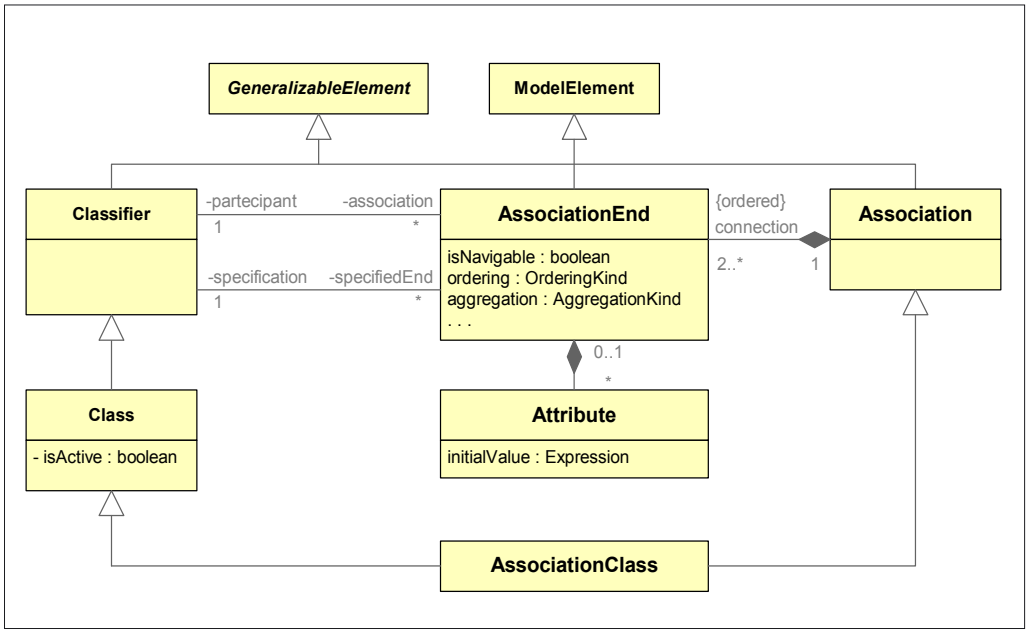
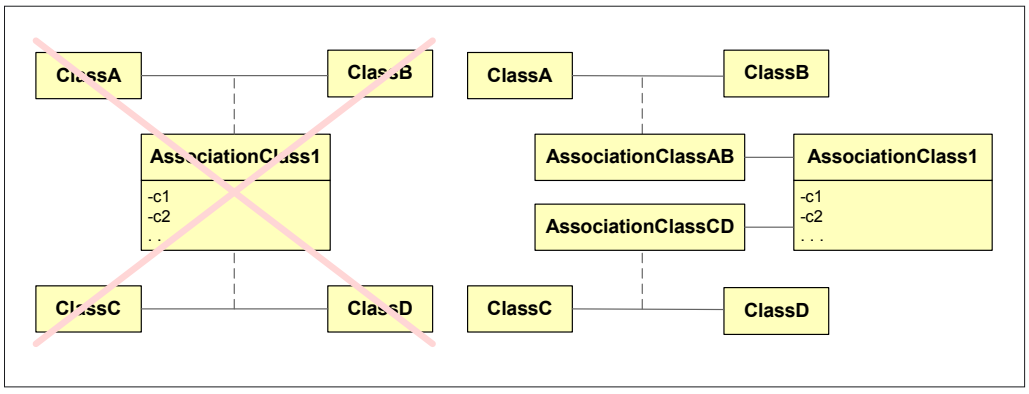


Figura 7.33 — *Classe associazione per due associazioni.*



Poiché la classe associazione è a tutti gli effetti una relazione, ne seguono interessanti constatazioni. Per esempio, anche se talune volte si potrebbe avere la necessità di collegare una classe associazione a più relazioni, ciò non è lecito. Sarebbe come voler collegare più associazioni tra loro. In queste circostanze, la soluzione è abbastanza semplice: si definisce un'ulteriore classe che contenga il comportamento comune e quindi si legano le varie association class a tale classe, come mostrato nella fig. 7.33.

Anche sull'utilizzo della relazione di associazione esistono diverse correnti di pensiero. C'è chi non la vorrebbe utilizzare giacché, dal punto di vista dell'implementazione, si tratta di un concetto non direttamente codificabile, c'è chi invece la ritiene assolutamente indispensabile. Con approccio assolutamente pragmatico, l'autore crede che sia opportuno avvalersi dell'association class per tutte le tipologie di modelli che precedono quello di disegno (dominio, business, analisi, …), qualora sussistano le condizioni. Ciò semplicemente perché permette di realizzare modelli più rispondenti alla realtà. Nei modelli di disegno, però, probabilmente, è preferibile utilizzare meccanismi alternativi, in quanto altrimenti si genererebbe un gap di astrazione rispetto alla codifica non sempre desiderabile.

Aggregazione e composizione

Un'associazione tra classi mostra una relazione strutturale paritetica (la famosa *peer to peer*), per cui tra le classi coinvolte non è possibile distinguerne una concettualmente più importante delle altre: sono tutte allo stesso livello. Spesso però è necessario modellare situazioni opposte in cui una classe, in una determinata associazione, esprime una nozione concettualmente più grande delle altre che la costituiscono. In altre parole è necessario utilizzare relazioni del tipo "tutto–parte" (*whole–part*), in cui esiste una classe che rappresenta il concetto "più grande" (il tutto) costituito dalle restanti che rappresentano i concetti più piccoli (le parti). Questi tipi di relazioni sono detti di aggregazione e, in particolari circostanze, diventano composizioni.

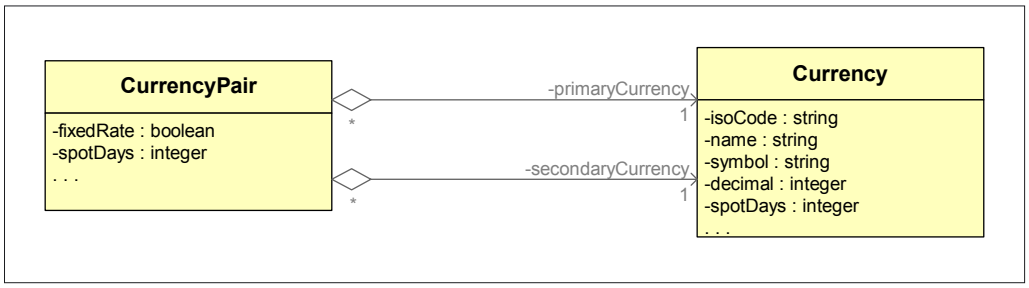
Chiaramente una stessa classe può essere la classe aggregata (il "tutto") di una specifica relazione e contemporaneamente essere parte componente di un'altra, così come una stessa classe può rappresentare la classe aggregata in diverse relazioni whole–part. Discorso analogo vale anche per le classi rappresentanti le parti.

Un esempio molto inflazionato di relazione di aggregazione è quello dell'autovettura. In questo caso, la classe Automobile è la classe aggregata costituita dagli elementi Motore, Ruote, Volante, e così via. Questo è uno dei tipici casi in cui l'autore comprende perché gli informatici siano tacciati a volte di essere un po' tediosi.

Graficamente un'aggregazione è visualizzata con un rombo vuoto dalla parte della classe aggregata (whole). Nel caso in cui l'aggregazione sia una composizione, allora il rombo è disegnato colorato al proprio interno.

Un esempio un po' meno noioso di relazione di aggregazione è costituito dalla `CurrencyPair` (coppia di valute) ossia il prodotto utilizzato per effettuare lo scambio di valute (*Foreign eXchange*). Si tratta del prodotto finanziario che, quotidianamente, muove

Figura 7.34 — *Esempio di relazione di aggregazione:* `CurrencyPair`.



un giro di affari dell'ordine dei 900 miliardi di dollari americani, di cui circa il 95% è generato da fini puramente speculativi, mentre il restante è dovuto a scambi commerciali tra nazioni utilizzanti valute diverse (da cui il nome del trade: `FX`, Foreign eXchange) illustrato in fig. 7.34.

Un altro esempio è quello relativo a un biglietto aereo. In particolare, un biglietto (`Ticket`) tipicamente prevede diverse tratte (`Journey`) ognuna delle quali è costituita da un determinato aeroporto di partenza e da uno di arrivo (*cfr.* fig. 7.35).

La relazione di aggregazione (e quindi anche quella di composizione) è effettivamente una particolare versione di quella di associazione visualizzata "ornando" l'associazione stessa per mezzo di un rombo nella parte della classe aggregata. La differenza tra l'associazione e l'aggregazione è puramente concettuale, tanto che spesso la decisione su quale utilizzare è veramente una questione di gusti. In ultima analisi un'aggregazione è un'associazione con semantica più forte: la classe aggregata è data delle parti componenti. Un vincolo del tutto logico implicito nella relazione di aggregazione è relativo alla totale mancanza di validità di associazioni di aggregazione circolari. In altre parole non ha senso dire che una Classe A sia composta da una Classe B, composta da una Classe C a sua volta composta dalla Classe A.

Anche dal punto di vista implementativo, non esiste grossa differenza tra una relazione di associazione e una di aggregazione; quest'ultima viene utilizzata per enfatizzare relazioni a semantica più forte.

Per quanto concerne l'associazione di composizione, è necessario effettuare un discorso diverso. In questo caso si ha una forma di aggregazione con una forte connotazione di possesso e una (quasi) coincidenza del ciclo di vita tra istanze delle classi "parte" e quella istanza della classe "tutto" (classe composta). Più precisamente, le parti possono essere generate anche in un tempo successivo alla creazione dell'istanza della classe composta, ma, una volta generate, queste vivono e sono distrutte con l'istanza della classe composta di appartenenza. Tipicamente, avviene che la classe composta si occupa di eliminare le istanze proprie parti in un momento antecedente alla propria distruzione.

Mentre la relazione di aggregazione rappresenta una relazione di associazione con un significato più forte, ma senza alcuna implicazione sulla navigabilità o sull'esistenza delle parti componenti, con la relazione di composizione le cose cambiano. Per esempio, in ogni istante di tempo un oggetto può essere "parte" esclusivamente di una composizione. Ciò non preclude alle istanze di una classe di partecipare a diverse relazioni di composizione in differenti istanti di tempo. Questo vincolo spiega il motivo per cui le relazioni che legano le `Currency` alle `CurrencyPair` sono legittimamente aggregazioni e non composizioni (una stessa valuta può partecipare a diversi prodotti di scambio di valute: `USDEUR`, `USDGBP`, `GBPEUR` ecc.), così come avviene per gli aeroporti nell'esempio di fig. 7.35. Pertanto, in una relazione di composizione, molteplicità con cui le classi componenti sono associate alla relativa classe composta diverse da `1` o `0..1` potrebbero non avere molto senso.

Nell'implementazione di una classe composta, è necessario ricordare che essa ha la responsabilità di creare e distruggere le proprie parti, e quindi deve essere disegnata considerando appositi metodi. Questo concetto è più evidente in linguaggi (per esempio il C++) in cui è necessario allocare e disallocare esplicitamente la memoria dei relativi oggetti; in tal caso la classe composta deve occuparsi di eseguire anche queste funzioni. In tali contesti è più evidente la necessità di visualizzare esplicitamente una relazione di composizione: rende più difficile commettere errori con la gestione della memoria. In linguaggi come Java, in cui è sufficiente dereferenziare un oggetto per consegnarlo alle amorevoli cure del Garbage Collector, la responsabilità di distruggere un oggetto ha un significato meno rilevante.

Si consideri il diagramma di fig. 7.36. Esso fa riferimento a un politica tipica dei sistemi di sicurezza atta a impedire che un utente possa riutilizzare la stessa password prima che ne abbia variate un certo numero (per esempio 6). In altre parole un utente è autorizzato a "riciclare" una stessa password solo dopo averne utilizzate almeno altre 6 differenti.

Quando un oggetto composto viene generato, tipicamente, si deve incaricare di creare le istanze delle sue parti e di associarle correttamente a sé stesso. Quando poi viene distrutto, esso ha la responsabilità di distruggere tutte le parti: ciò però non significa che necessariamente debba farlo direttamente.

Il frammento del modello di fig. 7.37 mostra la struttura gerarchica di un `DataSet` SQL, relativo allo standard SQL99. Il concetto di `Cluster` coincide (abbastanza) con quello di un'installazione di server RDBMS (Relational DataBase Management System, sistema di gestione di database relazionali), ed è costituito da un insieme di cataloghi. Questi, nei RDBMS più comuni (Oracle, SQLServer, ecc.), sono comunemente denominati istanze. Sebbene le direttive ANSI standard prevedano che i servizi di sicurezza (utenti abilitati all'accesso dei dati, elenco dei dati visibili, ecc.) siano applicati al livello di `Cluster`, la quasi totalità di RDBMS commerciali li rendono disponibili a livello di catalogo. Il passo successivo consiste nel considerare il livello di database denominato, sempre secondo direttive standard, `Schema`. Questo raggruppa un insieme di oggetti come tabelle,

Figura 7.35 — *Porzione di modello relativo a un biglietto aereo. Le istanze della classe* `Airport`*, che recita il ruolo di parte in due aggregazioni, in entrambi i casi, possono essere associate a diverse istanze della classe aggregata* `Journey`*.*

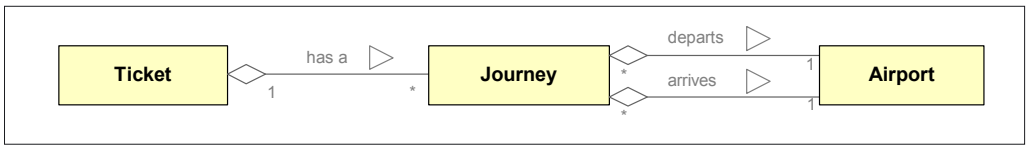
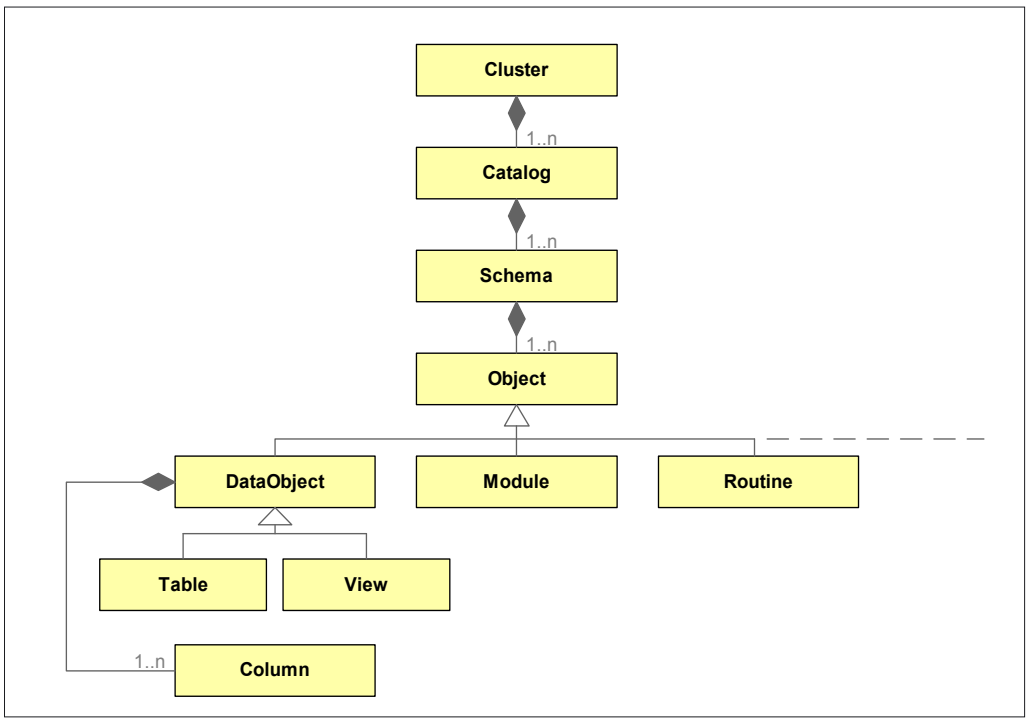


Figura 7.36 — *Frammento di modello relativo alla relazione tra gli utenti del sistema e le relative password.*

Figura 7.37 — *Frammento della rappresentazione gerarchica di un DataSet SQL.*

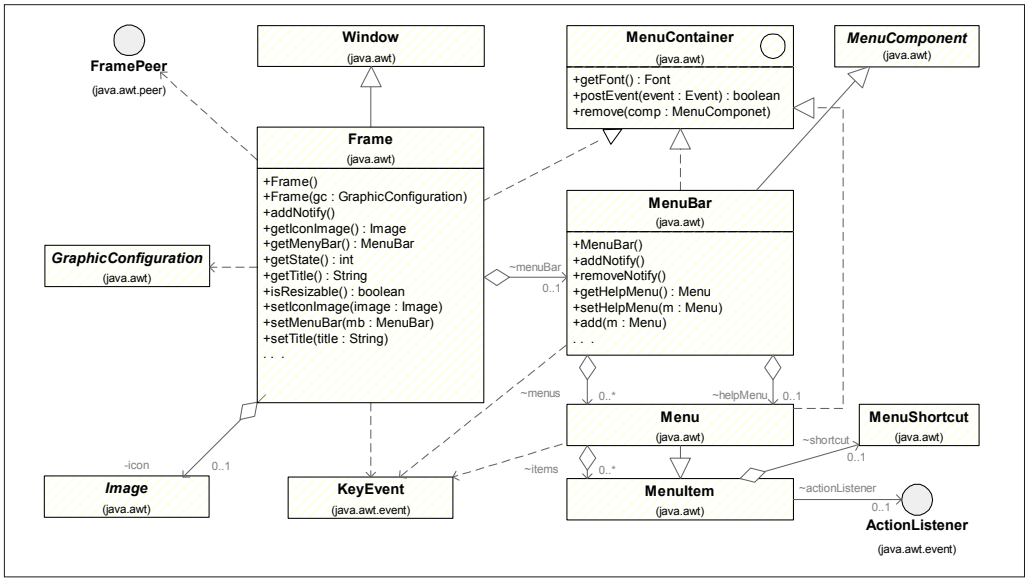


viste, moduli, routine, stored procedure, ecc. Gli oggetti di tipo data (`DataObject`), sono costituiti da colonne (attributi delle tabelle).

Il diagramma in questione si presta a essere ulteriormente dettagliato. Per esempio si potrebbero considerare i domini e le regole delle colonne, così come lo schema informativo (`INFORMATION_SCHEMA`), ossia metadata relativi a tutti gli oggetti memorizzati in un catalogo, ma l'obiettivo dell'esempio è mostrare l'utilizzo dello UML e non spiegare il funzionamento dei database relazionali.

Il diagramma di fig. 7.38 mostra una versione semplificata di alcuni componenti utilizzabili per la realizzazione della GUI (Graphic User Interface, interfaccia utente grafica) di applicativi realizzati in linguaggio Java. In particolare si fa riferimento ai componenti `Frame` e `MenuBar` del package `java.awt`. La classe `Frame` rappresenta una finestra generale utilizzabile per costruire apposite interfacce utente. A tal fine specializza la classe `Window`, aggiungendo caratteristiche supplementari, come la possibilità di avere una barra di menu, un'icona, un bordo, ecc. Un oggetto di tipo `Frame` può eventualmente essere disegnato secondo specifiche direttive fornite da un opportuno oggetto istanza di una classe specializzante: quella astratta `GraphicsConfigurator`. Il disegno "ridefinito" di oggetti "grafici" è utile qualora il dispositivo di destinazione non sia il classico monitor. Un oggetto `Frame` può eventualmente disporre di un'icona grafica che lo rappresenti (caratteristica non prevista dalla classe `Window`). Questa deve essere specificata attraverso un'istanza di una classe specializzante: quella base `Image`. Gli oggetti di tipo `Frame` sono poi in grado di processare eventi generati dall'utente e, a patto che dispongano della

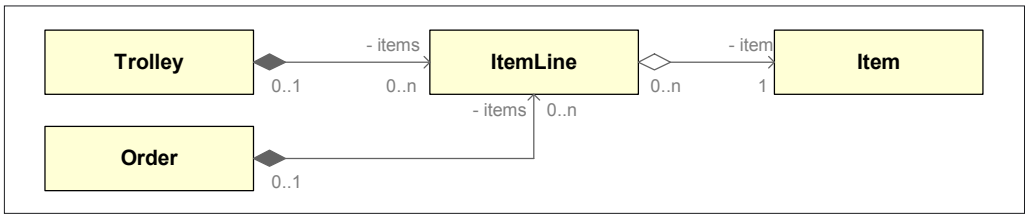
Figura 7.38 — `MenuBar` *nella libreria* `java.awt`.



barra dei menu, anche quelli provenienti dalla tastiera. In tal caso, a ogni pressione di un tasto è necessario verificare se questo corrisponda o meno a uno di scelta rapida associato al menu. Pertanto, la gestione dell'evento è demandata alla barra del menu. Ogni `Frame` può disporre al massimo di una barra del menu, e a tal fine implementa l'interfaccia `MenuContainer`. La barra del menu è poi costituita da due diverse tipologie di menu: help e menu classici. Del primo tipo ne può esistere una sola istanza, mentre dei secondi è possibile specificarne una lista (l'elenco delle voci visualizzate direttamente nella barra). La classe `Menu` estende `MenuItem` (ossia gli elementi che lo costituiscono) in quanto si tratta di un elemento particolare di un item in grado di ospitare altri menu: una voce che richiama un menu. Da un'attenta analisi si può notare che le classi `Menu` e `MenuItem`, rappresentano una forma collassata del pattern Composite, in cui la classe base è collassata in quella genitore (`MenuItem`). Ciò permette di ottenere menu annidati *n* volte (composti da altri menu, composti da altri menu e così via). Ogni menu può prevedere un `MenuShortcut`, ossia un oggetto che dichiara il relativo tasto di scelta rapida. Pertanto, quando un utente preme un tasto in un oggetto `Frame`, si verifica l'esistenza di un oggetto `MenuShortcut` corrispondente. In caso affermativo viene generato un evento di notifica (`ActionEvent`). Analizzando il modello, potrebbero scaturire diversi interrogativi. In primo luogo, ci si potrebbe chiedere se gli oggetti `Frame` utilizzino la barra dei menu — nel qual caso sarebbe stato opportuno utilizzare un'associazione semplice — oppure ne siano costituiti. La differenza è molto sottile ed entrambe le scelte sarebbero valide. L'autore ha scelto la seconda alternativa. Un'altra domanda sulla quale ci si potrebbe interrogare è la tipologia delle relazioni che legano la classe `MenuBar` a quella `Menu`. Si tratta di un'aggregazione o di una composizione? In effetti per molti versi potrebbe essere una composizione (almeno l'aggregazione *menus*), d'altronde che senso avrebbe una `MenuBar` senza `Menu`? Però, da come è stato codificato, sembrerebbe che la classe `MenuBar` non abbia alcuna responsabilità sulla costruzione e distruzione dei propri menu i quali, eventualmente, potrebbero essere utilizzati contemporaneamente da altri oggetti. Per questo motivo si è ritenuto più opportuno ricorrere a una relazione di aggregazione.

L'immagine di fig. 7.39 rappresenta un frammento di un modello a oggetti di un sito per il commercio elettronico. In particolare è rappresentata una situazione in cui il carrello della spesa (`Trolley`) è composto da una serie di "righe" (`ItemLine`) ognuna delle quali è costituita da un apposito prodotto (`Item`). In particolare le righe specificano il prodotto selezionato, la quantità, il prezzo, ecc. Ora, un utente, in un qualsiasi momento all'interno del periodo di validità del carrello, può decidere di acquistare alcuni prodotti precedentemente accantonati nel carrello della spesa. Ecco che quindi tale contenuto (o una sua parte) è incorporato in un apposito ordine (`Order`). Ora, mentre è lecito attendersi una relazione di aggregazione tra `Item` e `ItemLine`, uno stessa istanza della classe `Item` potrebbe essere contemporaneamente parte di molti oggetti `ItemLine` la presenza della relazione di composizione tra le classi `Trolley` e `ItemLine`, e `Order` e `ItemLine` potrebbe destare qualche perplessità insita nel fatto che una stessa classe (`ItemLine`)

Figura 7.39 — *Carrello della spesa e ordine.*

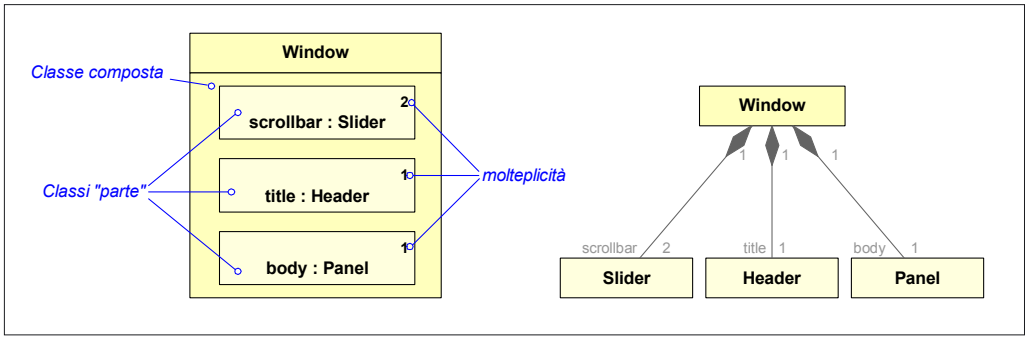


reciterebbe il ruolo di parte in più relazioni di composizione. Da un'analisi più attenta si scopre che proprio la presenza della relazione di composizione va a specificare un funzionamento molto preciso: poiché, in una relazione di composizione, una stessa istanza della classe parte non può appartenere contemporaneamente a più classi composte; ne consegue che, per rispettare tale vincolo, quando una riga del carrello della spesa viene selezionata per essere acquistata, il relativo oggetto (`ItemLine`) deve essere "staccato" dall'oggetto carrello ed essere attaccato a quello ordine.

Per molti autori, ogniqualvolta si utilizza una relazione di composizione, in qualche modo è possibile pensare alle classi rappresentanti le parti della relazione come private della classe composta. Poiché il vincolo implicito nella relazione di composizione è così forte, le classi componenti sono paragonate agli attributi della stessa classe composta. Per questi motivi è possibile visualizzare relazioni di composizione attraverso una rappresentazione grafica alternativa, come quella mostrata in fig. 7.40.

L'autore del libro non è completamente d'accordo con il paragone, o meglio, la maggior parte delle volte, una classe parte di una relazione di composizione è veramente simile a un attributo privato della classe composta, ma, in altri casi tale similitudine si rivela molto

Figura 7.40 — *Rappresentazioni alternative della relazione di composizione.*



sfumata. Tale considerazione trova giustificazione nel fatto che il vincolo della composizione asserisce che un'istanza di una classe composta, in ogni istante di tempo, può essere parte di una sola composizione. Logica conseguenza è che il vincolo non si applica alla classe stessa bensì alle relative istanze, quindi una classe parte di una composizione potrebbe partecipare a diverse relazioni eventualmente anche di composizione.

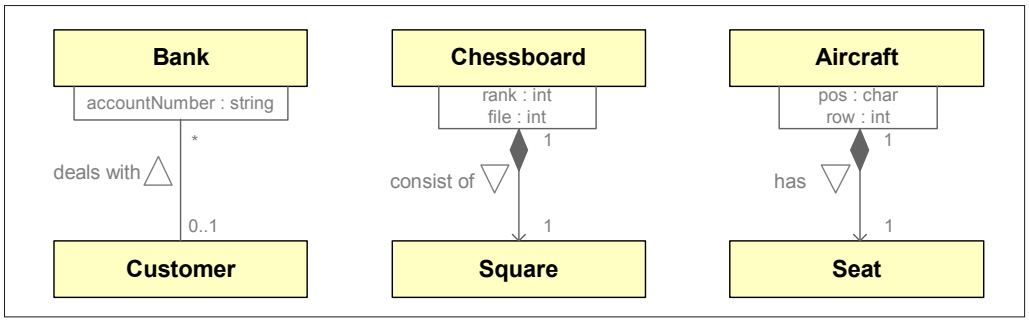
Qualificazione o associazione qualificata

Nella realizzazione di un modello, spesso capita di dover dar luogo a relazioni di associazione particolari, in cui esiste il problema della selezione (*lookup*). In altre parole, in una relazione *1* a *n* (o *n* a *n*) tra una Classe A e una B, fissato un determinato oggetto istanza della classe A, è necessario specificare un criterio in grado di individuare un preciso oggetto o un sottoinsieme di quelli associati, istanze dell'altra classe (B). Tali circostanze si prestano a essere modellate attraverso l'associazione qualificata (*qualification*), il cui nome è dovuto all'attributo, o alla lista di attributi, detti qualificatori i cui valori permettono di partizionare l'insieme delle istanze associate a quella di partenza. La Classe A viene comunemente definita classe qualificata, mentre quella B è detta classe obiettivo. In un contesto implementativo, ai qualificatori si fa comunemente riferimento con il nome di indici. Si tratta di attributi che, tipicamente, appartengono alla relazione stessa e, meno frequentemente, alla classe oggetto della selezione. Nel primo caso, i valori sono specificati dalla classe che gestisce la generazione di nuovi link.

Un esempio molto semplice è costituito da un vettore. In questo caso l'indice rappresenta il qualificatore, gli elementi del vettore sono gli oggetti obiettivo (*target object*), mentre la classe in cui il vettore è definito rappresenta la classe qualificata.

Dal punto di vista grafico, l'associazione qualificata viene mostrata per mezzo del classico segmento congiungente le due classi, con aggiunto un rettangolo dedicato alla specificazione dei qualificatori. Da tener presente che tale rettangolo appartiene alla relazione e va posto nelle prossimità della classe qualificata. Pertanto è in grado di individuare un

Figura 7.41 — *Esempi di utilizzo dell'associazione qualificata.*



sottoinsieme (eventualmente costituito da un solo elemento) degli elementi della classe destinazione associata, quest'ultima attraverso i valori dei qualificatori.

Il qualificatore è un attributo opzionale di ogni fine associazione binaria. Nelle relazioni *n-arie* (relazioni che coinvolgono più di due classi) non viene utilizzato: la comprensione del significato sarebbe una vera e propria impresa. Mentre in ogni associazione binaria, teoricamente, se ne potrebbero specificare due (si tratterebbe di un caso assolutamente raro), uno a ogni capo della relazione, a patto che la stessa relazione sia di tipo molti a molti.

Si considerino gli esempi classici riportati in fig. 7.41, relativi a un account bancario, una scacchiera e i posti a sedere su un volo aereo.

Per comprendere gli esempi in fig. 7.41 è necessario fornire qualche dettaglio relativo al significato della molteplicità che, in questo contesto assume un'accezione leggermente diversa da quella classica. In particolare, si possono avere i seguenti casi:

`0..1` indica che l'utilizzo del qualificatore (qualsiasi ne siano i valori), tipicamente, permette di identificare una sola istanza nella classe di destinazione. Lo zero invece evidenzia la possibilità che nessun oggetto possa scaturire dalla selezione operata per mezzo del criterio specificato dal qualificatore. Chiaramente la molteplicità `1` rappresenta la casistica più frequente.

`1` questo è un sottocaso del precedente, nel quale non esiste l'eventualità che nessun elemento possa venir selezionato. Ciò implica che il dominio dei qualificatori sia un insieme ben definito e chiuso;

`*` indica che i valori dei qualificatori permettono, tipicamente, di individuare un insieme di istanze del destinatario. Questa casistica non sempre aggiunge molto significato in quanto la molteplicità resta sempre "molti", indipendentemente dalla presenza del qualificatore. L'eventuale utilizzo potrebbe essere evidenziare una chiave (eventualmente composta) in grado di navigare nelle relazioni con gli oggetti obiettivo, oppure sottolineare il fatto che una particolare navigazione sia sottoposta a vincoli di performance per cui una semplice ricerca lineare non sarebbe idonea.

L'utilizzo dell'associazione qualificata, sebbene possa creare qualche perplessità iniziale, ha un campo di azione piuttosto ben definito: è utilizzata in tutti quei casi in cui vi è una struttura di *lookup* da una parte della relazione. Ciò equivale anche a dire che, dal punto di vista dell'implementazione, la relazione dovrebbe essere realizzata per mezzo di opportuni oggetti che agevolino la ricerca, come `Hashtable`, `Map`, ecc.

Caratteristica peculiare di tali oggetti è la realizzazione di un mapping tra un identificatore e un oggetto. Per esempio, la classe `Hashtable` in Java possiede i seguenti metodi per memorizzare e reperire oggetti:

```
put(key:Object, value:Object)
get(key:Object )
```

Pertanto le istanze di questa classe permettono di realizzare un mapping tra un oggetto che rappresenta la chiave (il qualificatore) e un altro che invece rappresenta ciò che si vuole individuare.

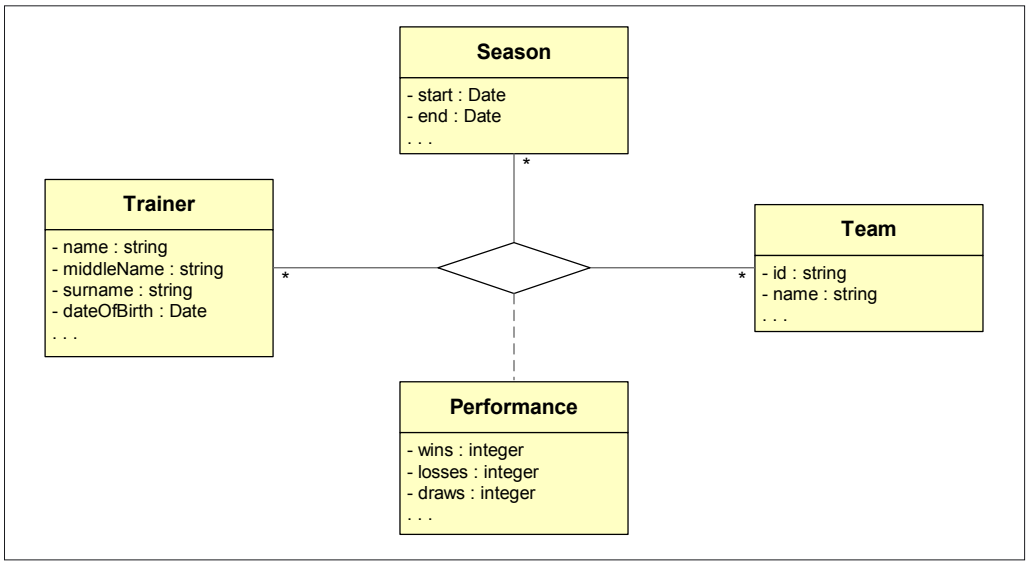
Associazione *n*-aria

Come è lecito attendersi, una associazione *n*-aria è una relazione che coinvolge più di due classi, tenendo presente che (caso rarissimo) una stessa classe può apparire più volte. Così come l'autoassociazione è un particolare caso dell'associazione binaria, allo stesso modo la relazione binaria può essere considerata un caso particolare di quella *n*-aria.

Una sfida interessante nel tracciare questo tipo di relazioni, consiste nell'attribuire i valori delle molteplicità: in questo caso il tutto è molto meno intuitivo. Queste devono specificare, per ogni classe, il potenziale numero di istanze della classe che possono partecipare nella relazione, fissando i valori delle altre *n–1* classi.

La notazione grafica utilizzata prevede un rombo in cui terminano le relazioni con le classi che prendono parte alla relazione. Eventualmente è possibile specificare il nome della relazione *n*-aria per mezzo di un'opportuna etichetta sistemata nei pressi del rombo.

Figura 7.42 — *Esempio di relazione n-aria con classe d'associazione.*



Utilizzare composizioni e/o aggregazioni non ha molto significato in relazioni *n*-aria.

Nella realtà si tende a utilizzare con molta parsimonia relazioni di questo tipo. Si tenga presente che un criterio da perseguire nel disegno del sistema è la semplicità. Nella pratica, si cerca sempre di ridurre relazioni n-arie in una serie di associazioni binarie attraverso l'introduzione di apposite classi di legame.

Volendo complicare ulteriormente le cose, è possibile dichiarare attributi/metodi appartenenti a una relazione *n*-aria. In questo caso la classe associazione viene collegata al rombo rappresentante la relazione per mezzo della solita linea tratteggiata.

Per i pochissimi lettori per i quali il calcio è esclusivamente un elemento della tavola periodica, l'esempio di fig. 7.42 mostra un modello relativo alla storia professionale degli allenatori, limitata al ruolo di preparatori tecnici. In particolare, il modello mostra che ciascuno di essi, nell'arco della medesima stagione agonistica, può allenare diversi team (non contemporaneamente si intende). Per complicare le cose si è aggiunta anche la classe associazione `Performance`, al fine di tener traccia dei risultati ottenuti dall'allenatore nel preparare una precisa squadra durante una determinata stagione.

Ogniqualvolta si ha un'associazione di questo tipo, si può immaginare che le istanze della classi coinvolte formino una particolare tupla; in questo caso gli elementi sono (Allenatore, Squadra, Stagione). Ora, per assegnare le molteplicità è necessario fissare la tupla ottenuta eliminando l'elemento del quale si vuole quotare la molteplicità, e quindi deter-

Figura 7.43 — *Scioglimento della relazione n-aria con classe associativa.*

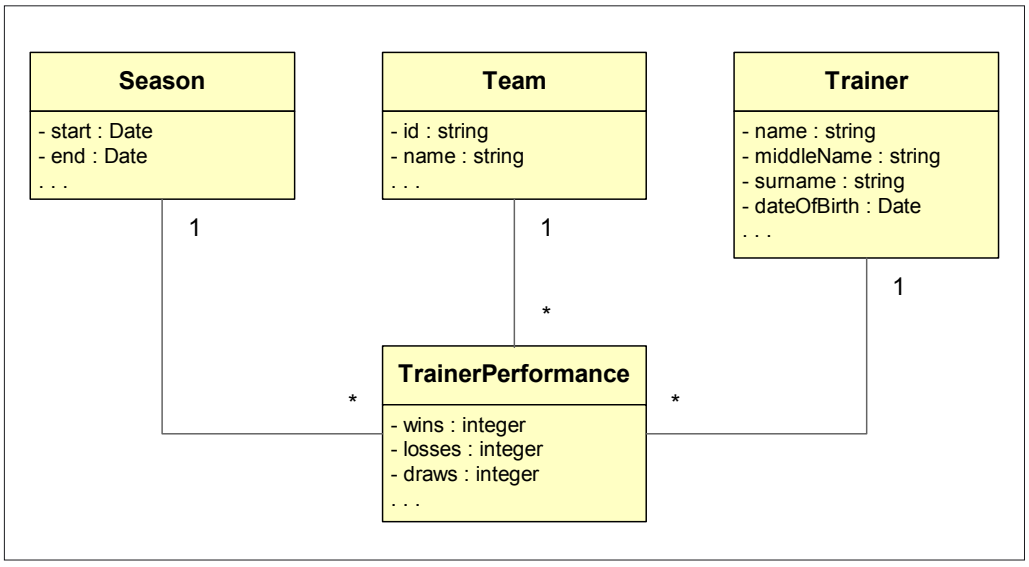
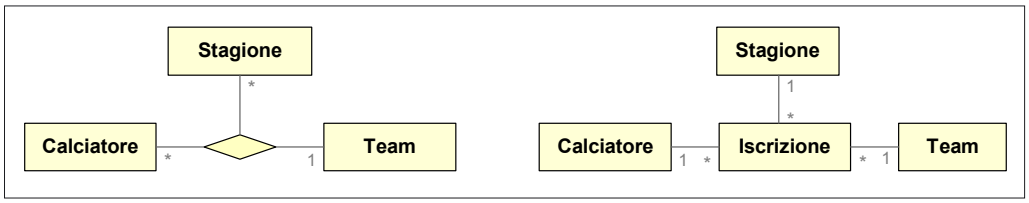


Figura 7.44 — *Proprietà della relazione n-aria.*



minare il valore. Per esempio volendo stabilire la cardinalità dell'allenatore, è necessario fissare la coppia (Stagione, Squadra) e porsi l'interrogativo "una squadra, durante una determinata stagione, da quanti allenatori diversi può essere allenata?": la risposta è, ovviamente, *n*. Analogamente, questa volta fissando l'attenzione sulla coppia (Allenatore, Squadra), la domanda da porsi è "per quante stagioni, un determinato allenatore può allenare una medesima squadra?". Anche in questo caso la risposta è "molte". Infine, per determinare la molteplicità mancante, è necessario rispondere all'interrogativo "durante la stessa stagione, quante diverse squadre un allenatore può allenare?". La risposta è ancora una volta *n*.

Il modello precedente può essere facilmente riportato a quello mostrato in fig. 7.43. In questo caso particolare entrambi i modelli rappresentano adeguatamente la stessa realtà, con un livello di intuitività diverso, sebbene il secondo modello non vieti la presenza ripetuta di una classe `TrainerPerformance`, relativa allo stesso allenatore, allenante la medesima squadra nel contesto della stessa stagione. Allora la domanda da porsi è se la relazione *n*-aria sia veramente utile… La risposta è, ovviamente, affermativa. Esistono casi in cui la relativa decomposizione in relazioni più semplici con una classe intermedia richiede l'aggiunta di ulteriori vincoli al fine di ottenere la stessa rappresentazione.

Si consideri la situazione in cui una business rule vigente in una competizione calcistica (magari internazionale) escluda la possibilità, nell'arco della medesima competizione, di iscrivere uno stesso giocatore in diversi team (ciò per evitare che un team proiettato verso le fasi finali della competizione possa acquistare giocatori da squadre già eliminate). In altre parole, si consideri il caso in cui per la competizione di ciascuna stagione non sia permesso ai giocatori di cambiare team e continuare a disputare lo stesso torneo. Questa situazione sarebbe facilmente rappresentabile con la relazione *n*-aria. In sostanza si otterrebbe un modello simile a quello di fig. 7.42, solo con la molteplicità posta `= 1` al lato `Team` (fig. 7.44). Il problema è che la scomposizione porterebbe a un modello il quale, senza ulteriori vincoli, non sarebbe in grado di evidenziare la famosa business rule. Da nessuna parte emerge l'impossibilità di iscrivere uno stesso giocatore, nell'arco della medesima competizione, con squadre diverse. In questo caso le molteplicità non sono di grosso aiuto; infatti, tipicamente, un giocatore effettua diverse iscrizioni per una competizione (chiaramente in anni diversi) e non è infrequente il caso in cui, giochi con diversi `Team`.

Attributi e relazioni

Al termine della sezione dedicata all'illustrazione delle relazioni definite dallo UML si è deciso di riportare una brevissima dissertazione relativa agli attributi e alle relazioni.

In termini programmativi (o, se si desidera, a tempo di esecuzione) non esiste alcuna differenza: entrambi rappresentano riferimenti in memoria "virtuale". Nel caso di relazioni, l'elemento indirizzato è lo stato di uno specifico oggetto (più qualche altra informazione), mentre nel caso degli attributi si tratta del valore impostato in una varabile di un determinato tipo. Se si considera poi che nei linguaggi puramente OO (come per esempio SmalTalk) non esistono tipi di dato, e tutti gli elementi sono oggetti, anche gli interi, i reali, i caratteri ecc., si comprende come, in effetti, la differenza tra queste due entità sia inesistente.

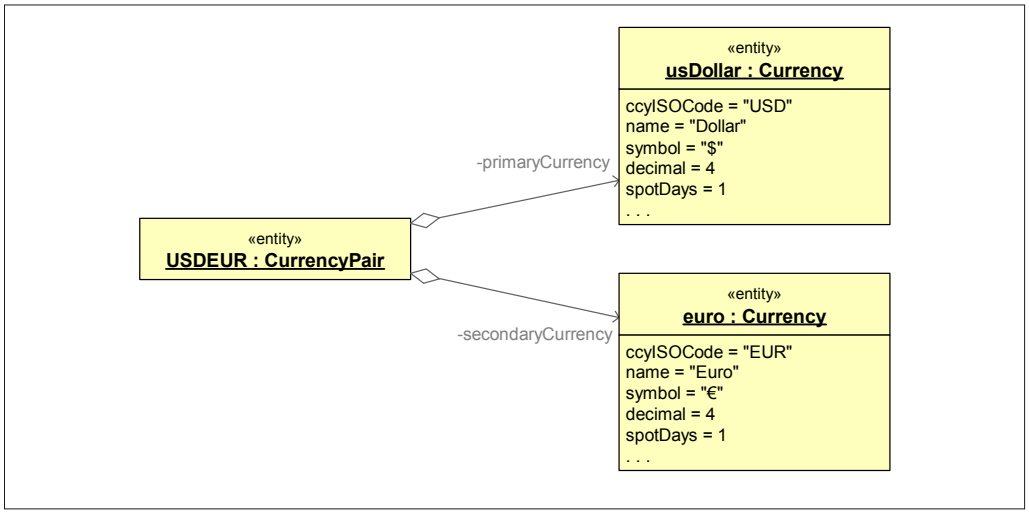
Nonostante ciò, al fine di agevolare la realizzazione di diagrammi leggibili ed eleganti, nella notazione dello UML si è preferito rappresentare gli attributi in un'opportuna sezione della classe e le relazioni attraverso le notazioni grafiche esaminate. Si pensi a quale caos si genererebbe se ogni attributo di una classe dovesse essere rappresentato per mezzo di un'apposita associazione o, viceversa, alla perdita della vista di insieme che si avrebbe qualora le relazioni venissero improvvisamente inglobate nella sezione di una classe.

Premesso ciò, molto spesso ci si può imbattere in modelli in cui alcune relazioni siano inglobate nella sezione dedicata agli attributi. Si tratta sempre di un errore? Concettualmente sì. In alcuni casi però si può placidamente chiudere un occhio e rassegnarsi all'idea che non si vincerà alcun premio nella competizione del diagramma a maggiore grado di formalità.

Si consideri un primo caso in cui si abbia una relazione (*1* a *n*) tra due classi (per esempio un `Cliente` può disporre di diversi `Indirizzi`). Se in questo caso si specificasse, nella classe `Cliente`, un attributo del tipo `Vector` dedicato a ospitarne gli indirizzi, verosimilmente non si tratterebbe esattamente di una buona pratica (utilizzando un eufemismo). Diversi modellatori trovano gratificante realizzare una soluzione mista che consiste nel rappresentare sia la relazione, sia l'attributo. La giustificazione addotta deriva dalla necessità di dover specificare l'oggetto che permette di realizzare fisicamente la relazione come per esempio un `Vector`. Chiaramente si tratta di una motivazione non accettabile: dettagli di questo tipo appartengono al modello implementativo e non a quello di disegno. In questi modelli è possibile specificare il nome dell'attributo attraverso il nome del ruolo dell'associazione. Volendo si possono fornire informazioni supplementari attraverso vincoli quali `ordered`, che forza a selezionare un tipo di struttura anziché un altro, eventualmente relazioni qualificate, ecc. Tutto il resto non è corretto.

Si consideri invece il caso di un diagramma concettuale relativo a un sito per il commercio elettronico. In questo caso, dovendo rappresentare diversi importi soggetti alla valuta utilizzata, se si utilizzasse un oggetto `Money` a mo' di tipo base (quindi rappresentato direttamente nella sezione degli attributi anziché attraverso opportune relazioni), non si tratterebbe probabilmente della peggiore bestialità del mondo. Un discorso del tutto analogo

Figura 7.45 — *Istanza del modello relativo alla* `CurrencyPair` *Dollaro USA/Euro. In questo diagramma si è deciso di mostrare gli stereotipi per mezzo dell'apposita etichetta. Nulla vieta di utilizzare la relativa icona (circonferenza con un segmento orizzontale alla base).*



vale per gli attributi di tipo *data*. Verosimilmente non è il caso di trattarli come relazioni con la medesima classe, bensì è sufficiente riportarli nel compartimento degli attributi.

I veri diagrammi a oggetti

I diagrammi degli oggetti (*object diagram*) propriamente detti rappresentano una variante dei diagrammi delle classi, o meglio ne sono istanze, e ne condividono gran parte della notazione. Rappresentano la famosa istantanea del sistema eseguita in preciso istante di tempo di un'ipotetica esecuzione. Quindi, a differenza dei diagrammi delle classi, popolati di elementi astratti come classi e interface, i diagrammi degli oggetti sono colonizzati da oggetti sospesi in un particolare stato. 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, posseduti da precisi attributi).

> Una delle caratteristiche molto apprezzate dello UML è la cosiddetta *dicotomia tipo-istanza*. Si tratta di un meccanismo che permette di rappresentare sia aspetti generici, sia esempi di elementi concreti derivanti dai primi. I casi classici di dicotomia tipo-istanza sono dati dalle coppie classi–

oggetti, parametri–valori, ecc. In questo contesto, un esempio importante è la dicotomia associazioni-collegamenti (`Link`). Questi ultimi sono istanze (quindi presenti nei diagrammi degli oggetti) delle associazioni che invece vivono nei diagrammi delle classi. La presenza della metaclasse `Link` impone la presenza della metaclasse fine collegamento (`LinkEnd`) corrispondente all'elemento di fine associazione (`AssociationEnd`). L'esistenza di quest'ultima dicotomia, permette di visualizzare nei collegamenti riportati nei diagrammi degli oggetti diverse informazioni. Per esempio è possibile riportare il nome dei ruoli (nel diagramma di fig. 7.45 sono presenti i ruoli `primaryCurrency` e `secondaryCurrency`), l'eventuale nome dell'associazione (coerentemente visualizzato sottolineato: si tratta del nome di un'istanza), altri adornamenti quali composizioni, aggregazioni, navigabilità ecc. Per terminare è possibile evidenziare la modalità con cui una relazione è ottenuta (ossia gli stereotipi della fine collegamento). In particolare sono previste le alternative mostrate nella tabella 7.4.

Chiaramente non ha alcun senso riportare le molteplicità giacché le varie relazioni sono riportate in maniera esplicita. Quindi ogni collegamento è, ovviamente del tipo *uno-a-uno*.

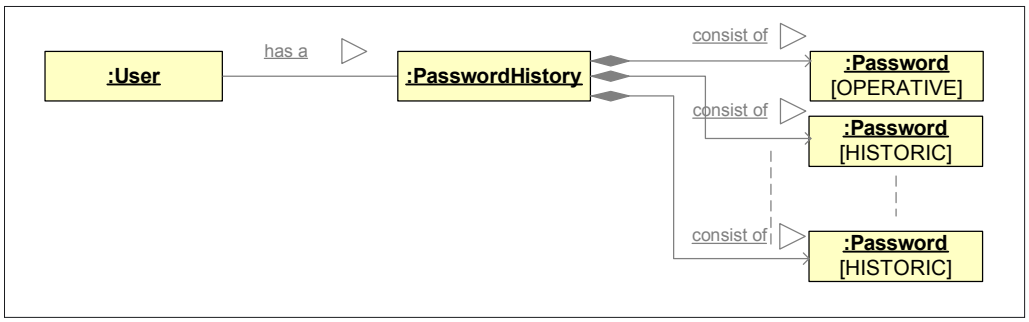
Quantunque lo UML preveda la possibilità di specificare molte informazioni nei collegamenti presenti nei diagrammi degli oggetti, si tratta di una pratica abbastanza infrequente: si tratta di informazioni già riportate nei corrispettivi diagrammi delle classi.

Ogni oggetto possiede una sua identità che, nel caso limite, è data dai valori assunti da tutti i suoi attributi e dalle relazioni stabilite. Graficamente è possibile rappresentare tale

Tabella 7.4— *Stereotipi della metaclasse* `LinkEnd` *utilizzati per rappresentare la natura di una relazione. La rappresentazione grafica riportata (lettera bianca nel quadratino nero) è utilizzata da diversi tool, ma non appartiene alle direttive standard.*

Stereotipo	Descrizione	Spiegazioni
«association» A	associazione	Indica che l'istanza collegata è visibile per mezzo di una relazione di associazione. Si tratta della situazione di default.
«parameter» P	parametro di un metodo.	Il riferimento all'oggetto è realizzato attraverso un parametro.
«local» L	variabile locale di un metodo.	Il riferimento all'oggetto è ottenuto per mezzo di una variabile locale.
«global» G	variabile globale	L'oggetto di destinazione è visibile in quanto il relativo riferimento è memorizzato in una variabile globale.
«self» S	auto collegamento	Indica che l'istanza è visibile in quanto è la stessa che effettua la richiesta.

Figura 7.46 — *Diagramma degli oggetti relativo al diagramma delle classi di fig. 7.36.*



identità per mezzo di un apposito nome, e/o mostrare lo stato dell'oggetto (va riportato sotto il nome dell'oggetto racchiuso tra parentesi quadre). Volendo si può anche associare a tale stato un nome simbolico e limitarsi alla rappresentazione di quest'ultimo, evitando magari di dover visualizzare il valore associato a ogni attributo (*cfr* fig. 7.46).

Dall'analisi del diagramma di fig. 7.46 è possibile evidenziare come i diagrammi degli oggetti mostrano un insieme di oggetti istanza di opportune classi, con i legami evidenziati in modo esplicito. Per esempio, nel diagramma delle classi di fig. 7.36, una classe `PasswordHistory` è associata con, al più, 6 istanze della classe `Password` per mezzo della relazione `consist of`, mentre nel relativo diagramma degli oggetti tale relazione è resa esplicita attraverso la visualizzazione di (al più) 6 legami espliciti con altrettanti oggetti di tipo `Password`. Questi sono rappresentati nel diagramma mostrando unicamente il tipo (oggetti anonimi) e il relativo stato: solo una password è operativa, mentre le restanti sono storicizzate.

I diagrammi degli oggetti, come quelli delle classi, sono utilizzati per illustrare la vista statica di disegno del sistema, però, a differenza di quest'ultimi, utilizzano una prospettiva diversa, focalizzata su un esempio, reale o ipotetico, dello stato di evoluzione del sistema, piuttosto che su una sua astrazione.

Gli elementi tipicamente presenti in un diagramma degli oggetti sono oggetti e collegamenti espliciti tra di essi. Come tutti gli altri diagrammi, poi, è possibile inserire annotazioni, vincoli, ecc. Spesso si introducono anche elementi come package e sottosistemi per mostrare l'organizzazione strutturale del sistema.

La realizzazione del diagramma degli oggetti può essere paragonata a un'ideale "congelamento" del sistema, operato in un preciso istante di tempo al fine di analizzare lo stato degli oggetti presenti. L'idea è paragonabile a un processo di debugging del sistema: si tenta di congelare lo stato (se possibile) e si ispezionano gli oggetti per analizzarne il valore degli attributi, le relazioni, ecc. In effetti, una utility molto interessante, che i tool di sviluppo (IDE) potrebbero prevedere, potrebbe consistere in una funzione avanzata di

reverse engineering inserita nel debug, in grado di tracciare, su richiesta, il diagramma degli oggetti del sistema in un particolare istante di tempo per un insieme specifico di oggetti e quindi permetterne la navigazione.

Chiaramente, i diagrammi degli oggetti possono essere utilizzati anche per mostrare esempi di modelli delle classi che non necessariamente verranno implementati. Per esempio frammenti dei modelli degli oggetti business e di dominio.

I diagrammi degli oggetti servono principalmente per esemplificare:

- strutture e relazioni particolarmente complesse o poco chiare, attraverso esempi pseudoreali;
- livelli di astrazione che potrebbero generare problemi di comprensione delle caratteristiche relative a singoli meccanismi nel loro dettaglio.

Nel realizzare diagrammi degli oggetti, tipicamente, si cerca di conferire rilievo a particolari stati in cui specifici insiemi di oggetti potrebbero trovarsi in un determinato istante di tempo, trascurandone altri di minore interesse. Questi oggetti potrebbero consistere in opportuni frammenti di una struttura di dati, di un particolare meccanismo, e così via. Quindi è molto importante saper discernere gli oggetti che potenzialmente possono concorrere a fornire valore aggiunto alla spiegazione da quelli che invece è preferibile trascurare (in sostanza si definisce un particolare scenario). Ciò evita di realizzare diagrammi di esempio con un numero eccessivo di elementi, che correrebbero il rischio di non raggiungere il risultato sperato. Pertanto, non solo è necessario stabilire quali oggetti visualizzare, ma anche lo stato in cui presentare ciascuno di essi in funzione del meccanismo generale che si vuole descrivere. Ciò comporta la dichiarazione dei valori degli attributi ritenuti più significativi (volendo è possibile dichiarare direttamente lo stato) e l'illustrazione esplicita delle relazioni tra gli oggetti.

Come indicato in precedenza gli oggetti sono visualizzati con una notazione simile a quella delle classi, con la differenza che:

- non ha senso mostrare il compartimento (compartimento relativo ai metodi);
- il nome dell'oggetto e la relativa classe di appartenenza sono visualizzati sottolineati (`objectName:classname`). Da notare che entrambi gli elementi possono essere omessi ma almeno uno dei due deve essere presente (nel caso in cui il nome sia omesso, si dice che ha un'istanza anonima, mentre nel caso in cui sia l'indicazione della classe a mancare, si dice che l'istanza è orfana);
- tutte le istanze delle relazioni vengono mostrate;

- gli attributi vengono evidenziati con il relativo valore (`attributeName:type = value`). Gli attributi i cui valori non risultano utili possono essere omessi, così come l'intero compartimento.

I diagrammi degli oggetti non sono di fondamentale importanza per la progettazione del sistema, però risultano decisamente utili per esemplificare sezioni particolarmente complesse o poco chiare dei corrispondenti diagrammi delle classi.

Da notare che i diagrammi degli oggetti sono gli unici che utilizzano propriamente il concetto di oggetto. Negli altri diagrammi (quali sequenza e collaborazione, per esempio) gli oggetti compaiono con il significato più di ruoli di oggetti che di oggetti veri e propri.

Si consideri il diagramma delle classi di fig. 7.47. Esso mostra un frammento di un modello relativo ai possessori di carta di credito. Questi ultimi possono avere diversi o nessuno account primari, e nessuno o molteplici account ordinari (chiaramente almeno uno ne devono possedere…). La differenza tra i due ruoli di account consiste nel fatto che gli ordinari sono utilizzati per gestire carte di credito e quindi tutti i possessori ne debbono possedere almeno uno, mentre i primari sono utilizzati per pagare i conti, pertanto tutti gli account ordinari devono essere associati a uno primario: ciò è mostrato attraverso l'autoassociazione presente nella classe `Account`.

La relazione tra la classe `CreditCardHolder` e `Account`, il cui ruolo è `ordinaryAccounts`, potrebbe sembrare ridondante (dato un oggetto di tipo `Account` che recita il ruolo di account primario è possibile risalire a tutti i suoi ordinari). Un'attenta analisi, magari eseguita con l'ausilio di un diagramma degli oggetti, mostra che invece si

Figura 7.47 — *Frammento di modello relativo al legame tra i possessori di carta di credito e i relativi account.*

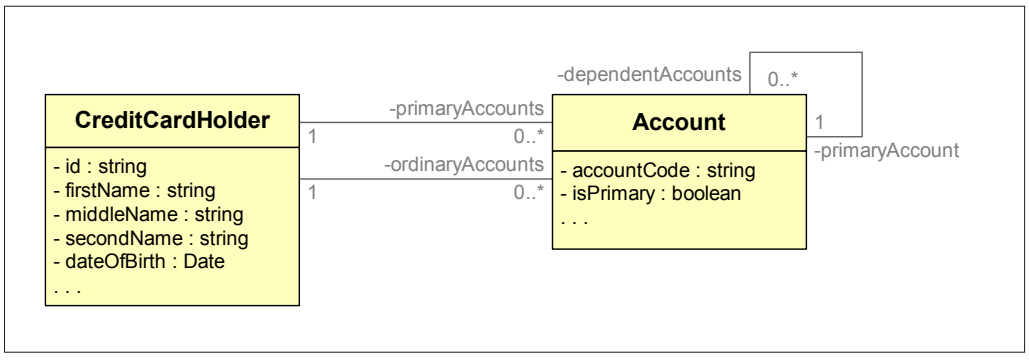
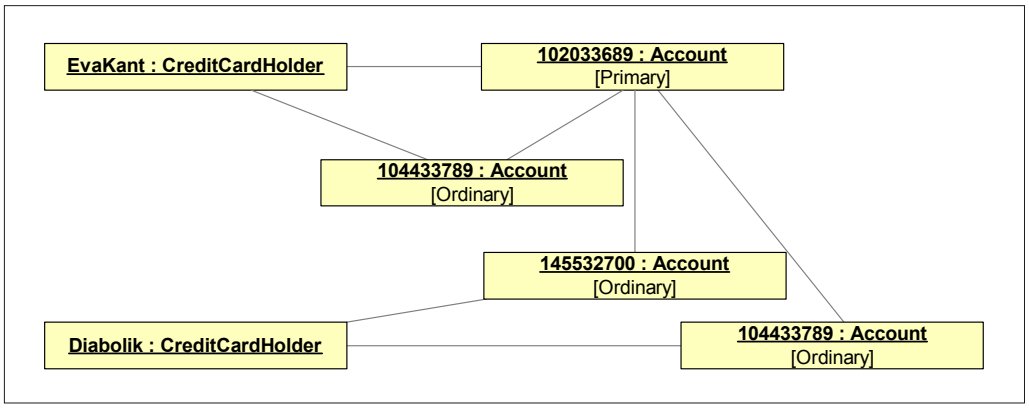


Figura 7.48 — *Diagramma degli oggetti relativo al precedente diagramma delle classi. Come si può notare, sia Eva Kant, sia Diabolik possiedono carte di credito e* quindi `Account` *ordinari per la loro gestione. I relativi conti vengono però addebitati tutti sull'account primario di Eva. Pertanto, il diagramma degli oggetti mostra chiaramente perché l'associazione tra* `CreditCardHolder` *e* `Account` *il cui ruolo è* `ordinaryAccounts` *non è ridondante.*



tratta di una relazione di associazione fondamentale. Volendo chiarire il frammento di modello si consideri diagramma degli oggetti di fig. 7.48.

Purtroppo, la quasi totalità dei tool non prevede esplicitamente funzionalità per la realizzazione di diagrammi degli oggetti. Tale lacuna può essere in parte raggirata grazie all'utilizzo degli strumenti previsti per la realizzazione dei diagrammi di collaborazione e, raramente, dei diagrammi delle classi stessi.

Istanze nel metamodello

Il dualismo comportamentale di alcuni elementi, in gergo denominato "dicotomia tipo–istanza" (*type–instance dichotomy*), come per esempio classi e oggetti, è presente nel metamodello dello UML. Le istanze, tipicamente, non restano isolate, ma vengono associate a un'opportuna astrazione. La notazione grafica utilizzata prevede l'utilizzo dello stesso simbolo geometrico per la coppia di elementi relazionati. La distinzione degli elementi è ottenuta sottolineando il nome dell'istanza. Ciò è molto utile perché da un lato concorre a mostrare che i due concetti condividano molte similitudini, e dall'altro permette di evidenziarne le differenze in maniera molto semplice e diretta.

Il diagramma mostrato in fig. 7.49 rappresenta una versione semplificata delle varie istanze previste dallo UML. Chiaramente si tratta di istanze delle specializzazioni dell'elemento astratto `Classifier`. Nella figura non sono mostrati gli "stimoli" che tipicamente sono composti da istanze.

L'interpretazione del diagramma dovrebbe essere ormai chiara. Si tenga presente che si tratta di istanze e pertanto ha senso dire che una particolare istanza di un componente può risiedere al più in un solo nodo (un'istanza di un nodo è, tipicamente, un determinato computer), così come una particolare istanza può risiedere in un solo componente. La metaclasse `AttributeLink` serve, essenzialmente, per visualizzare i valori degli attributi che caratterizzano una particolare istanza. Questi possono essere legami con altri oggetti o, più semplicemente, classici valori di attributi.

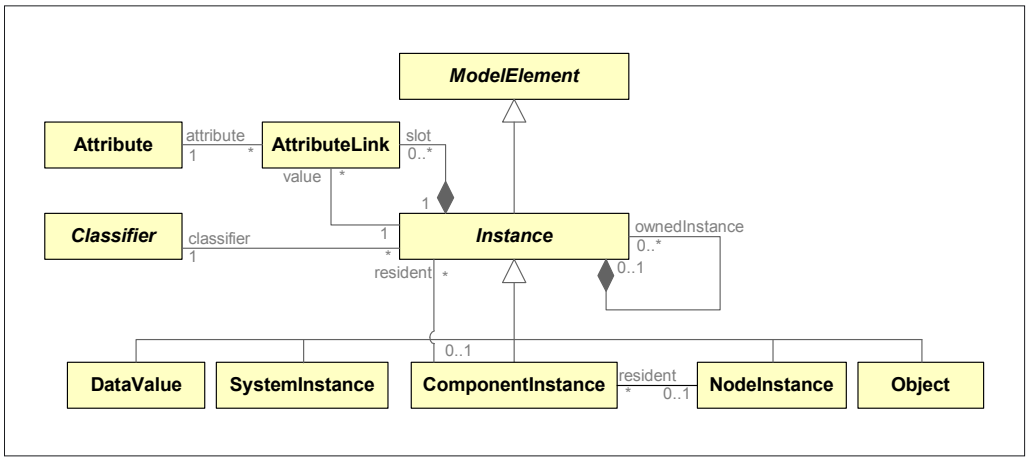
Esempio: definizione formale del comportamento dei casi d'uso

Nei prossimi paragrafi si elaborerà un modello delle entità che, opportunamente relazionate, permettono di descrivere formalmente il modello su cui si basa la descrizione del comportamento dinamico dei casi d'uso. In altre parole, utilizzando il formalismo dei diagrammi delle classi, si vuole realizzare il modello di dominio della versione dei flussi degli eventi illustrata nel capitolo relativo ai casi d'uso.

L'esempio oggetto di studio è stato selezionato poiché:

- si riferisce ad un dominio (quello dei casi d'uso) che ormai dovrebbe essere ben noto ai lettori;
- fornisce ottimi spunti per ricapitolare l'utilizzo degli elementi introdotti nel corso di questo capitolo;
- permette di riepilogare e consolidare i princìpi base della descrizione del comportamento dinamico dei casi d'uso, illustrate nel relativo capitolo;

Figura 7.49 — *Frammento del metamodello package Common Behavior – Instances.*



- offre l'opportunità di evidenziare la versatilità dello UML (e dell'OO più in generale) come strumento in grado di rappresentare formalmente modelli reali, non necessariamente vincolati alla realizzazione di sistemi software. Nulla, ovviamente, vieta di utilizzare l'esempio come modello a oggetti del dominio di un sistema atto ad agevolare l'analisi dei requisiti e la produzione di casi d'uso consistenti.

Il modello che si tratterà di seguito rappresenta un'istanza di un modello a oggetti del dominio (Domain Object Model), pertanto l'interesse è focalizzato sugli oggetti coinvolti (o meglio, sulle relative classi), le reciproche relazioni e gli attributi, mentre l'attenzione

Tabella 7.5 — *Struttura del template utilizzato per la descrizione del comportamento dinamico dei casi d'uso.*

CASO D'USO: ***Codice***	***Nome***	**Data:** ***dd/mm/yyyy***
		Versione: *n.nn.nnn*
Descrizione:	*Breve descrizione.*	
Durata:	*Durata.*	
Priorità:	*Elevata/Media/Bassa.*	
Attore primario:	*Nome attore*	
	Coinvolgimento nel caso d'uso	
Attore secondario:	*Nome attore*	
	Coinvolgimento nel caso d'uso	
Precondizioni:	*Elenco precondizioni.*	
Garanzie di fallimento:	*Elenco garanzie di fallimento.*	
Garanzie di successo:	*Elenco garanzie di successo.*	
Avvio:	*Descrizione dell'evento che avvia il caso d'uso.*	

Scenario principale.		
	ATTORE X	***SISTEMA***
1.	*Azione 1*	
2.		*Azione 2*
	...	...
n.		*Azione n*

Scenario alternativo: *Descrizione.*	
Step	*Azione.*

Scenario di errore: *Descrizione.*	
Step.	*Azione.*

Annotazioni.	
Step.	*Descrizione nota.*

per i metodi è piuttosto relativa. Volendo, anziché rappresentare gli attributi, si sarebbero potuti definire appositi metodi `get/set`. Chiaramente il contenuto informativo sarebbe stato lo stesso, mentre il livello di chiarezza, probabilmente, ne avrebbe risentito.

Per comodità di esposizione, il modello preso in esame è mostrato nella tabella seguente, la cui descrizione dettagliata è riportata nel paragrafo *Template* del Capitolo 4.

Individuazione delle classi

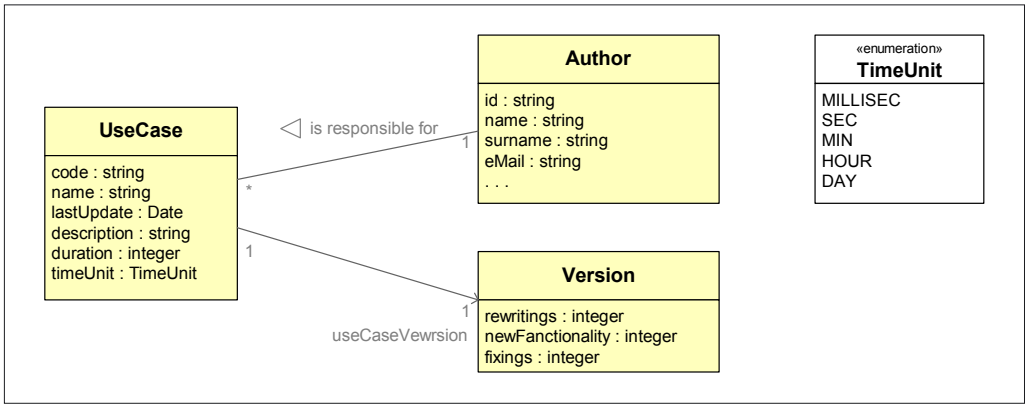
Il processo di scoperta delle classi appartenenti al modello a oggetti del dominio è, tipicamente, un'attività complessa nella quale giocano un ruolo fondamentale caratteristiche oggettive del modellatore quali esperienza, capacità analitiche ecc. In questo capitolo, poiché l'attenzione è focalizzata sul formalismo dei diagrammi delle classi e non sul processo di scoperta delle classi, viene presentato un approccio abbastanza informale, mentre nel capitolo successivo è presentato un approccio più organico e rigoroso.

Qualora nell'area business di cui si vuole realizzare un modello siano disponibili dei moduli, sia cartacei, sia digitali (schermate di sistemi esistenti), questi forniscono un ottimo punto di partenza per realizzare prime versioni del modello. Però, a fronte del vantaggio di avere un riscontro diretto e oggettivo delle entità coinvolte, esiste una serie di rischi cui è opportuno prestare attenzione, come per esempio realizzare un'analisi del dominio limitata alla proiezione mostrata nei documenti, produrre modelli poco flessibili, e così via. Ciò nonostante rappresentano un buon punto di partenza per realizzare un embrione del modello, il cui passo successivo consisterebbe nel realizzarne una versione generalizzata.

Si cominci a costruire il modello partendo da una classe che, in qualche modo, rappresenta il punto di ingresso dell'intero modello il cui nome non può che essere `UseCase` (fig. 7.50). Questa classe è associata con quella denominata `Author`, in quanto è opportuno che ciascuno use case sia gestito da una sola persona. Il "numero" della più recente versione del caso d'uso disponibile è mostrato attraverso un'apposita relazione con la classe `Version`.

Durante il processo di analisi dei requisiti utente, è molto importante assegnare la responsabilità di ciascuno use case a una, e una sola, persona, al fine di evitare perdite di tempo per mancanza di "memoria storica" derivante dagli avvicendamenti, di avere continuità e coerenza nelle specifiche, di disporre di personale specializzato su determinati argomenti e con un rapporto consolidato con gli utenti di riferimento, ecc.

Figura 7.50 — *Particella iniziale del modello rappresentante la classe* `UseCase` *e le relative relazioni con* `Author` *e* `Version`.



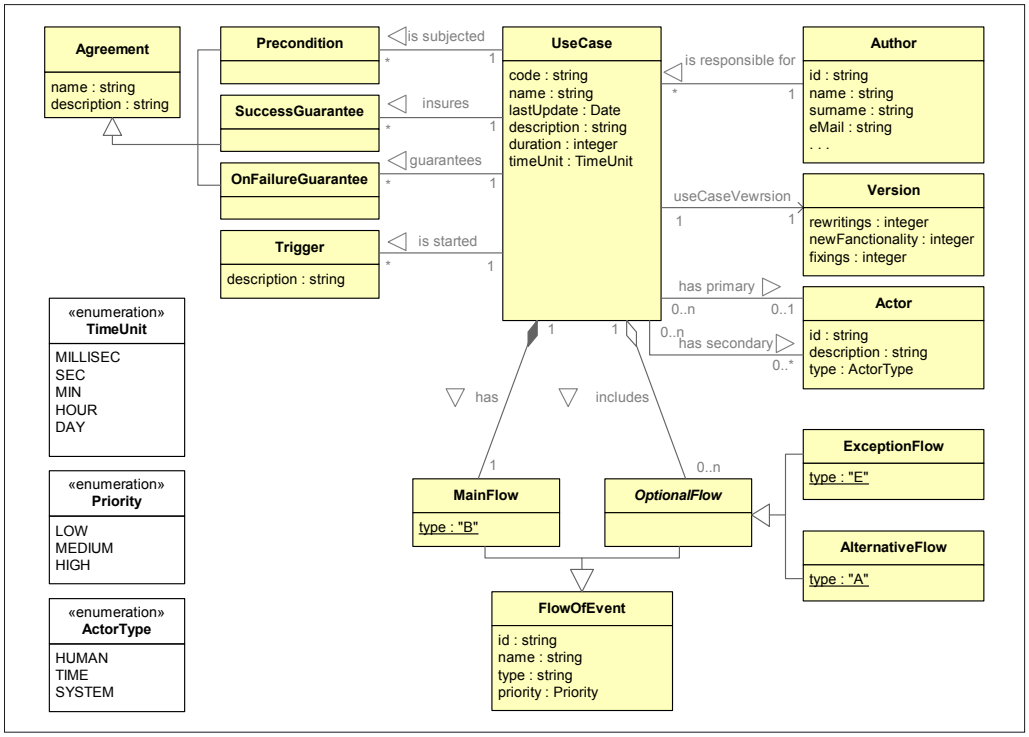
La classe `Version` può essere considerata a tutti gli effetti un'estensione dei tipi di dato forniti dal linguaggio (a dire il vero, tutte le classi lo sono, ma alcune più delle altre). Le relative istanze permettono di memorizzare informazioni nel formato "`n.n.n`" in cui i tre numeri sono legati da un legame gerarchico. In particolare, l'incremento di uno di essi, determina l'azzeramento di quelli che si trovano alla propria destra. Per esempio se si decidesse di riscrivere interamente la versione "`0.03.05`" perché esasperati dalla scarsa qualità della descrizione dinamica del caso d'uso, la versione successiva sarebbe "`1.00.00`". Questo concetto potrebbe essere rappresentato in maniera più sofisticata e flessibile con qualche accorgimento (magari ricorrendo a un pattern Composite). L'aumento di flessibilità prodotto, in questo caso, non solo sarebbe di relativo interesse ma avrebbe addirittura degli effetti negativi: il diagramma risulterebbe decisamente meno chiaro. Il consiglio è di rendere i modelli più flessibili possibile laddove vi sia una concreta necessità. Come al solito, la flessibilità ha il suo costo e nei modelli di dominio è sempre consigliabile tendere alla semplicità.

Il passo successivo consiste nel considerare le precondizioni (`Precondition`), le garanzie in caso di successo (`SuccessGuarantee`), quelle di fallimento (`OnFailureGuarantee`) e gli eventi in grado di innescare l'esecuzione del caso d'uso (`Trigger`).

Le prime tre classi estendono (magari in maniera un po' artificiosa, ma senza effetti collaterali), la classe base `Agreement`, mentre per quanto concerne i `Trigger`, si è interessati esclusivamente alla loro descrizione.

Procedendo con l'esame del modello, si giunge agli attori e ai primi embrioni dei flussi degli eventi. Per quanto concerne i primi, si può notare che, per ogni caso d'uso, può esisterne al più uno primario e diversi secondari.

Figura 7.51 — *Introduzione dei concetti di* `Precondition`, `SucessGuarantee`, `OnFailureGuarantee` *e* `Trigger`.



Teoricamente ogni use case dovrebbe essere avviato da un solo attore e, pertanto, ogni caso d'uso dovrebbe disporre almeno dell'attore primario. Ciò però non è vero nel caso di use case solo "inclusi" che eseguono specifiche attività senza mai interagire con gli attori o semplicemente fornendo alle stesse il risultato di opportune elaborazioni. Si tratta di casi d'uso introdotti per condensare funzionalità utilizzate da diversi altri casi d'uso, al fine di rendere il modello più elegante evitando di doverne ripetere ogni volta la descrizione del medesimo servizio.

Dal punto di vista degli attori, ciascuno di essi potrebbe non essere mai primario o mai secondario (ciò è evidenziato dalla molteplicità `0..n` della classe `Actor` nelle relazioni `has primary` e `has secondary`). Chiaramente entrambe le possibilità sono piuttosto inusuali.

Si noti che spesso è utile fornire informazioni supplementari relative all'interazione tra il caso d'uso e l'attore: per esempio quale sia l'interesse di un attore nell'interagire con uno specifico caso d'uso. Tale situazione è esprimibile come illustrato nel modello di fig. 7.53.

Figura 7.52 — *Inclusione degli attori e dei primi embrioni di flussi.*

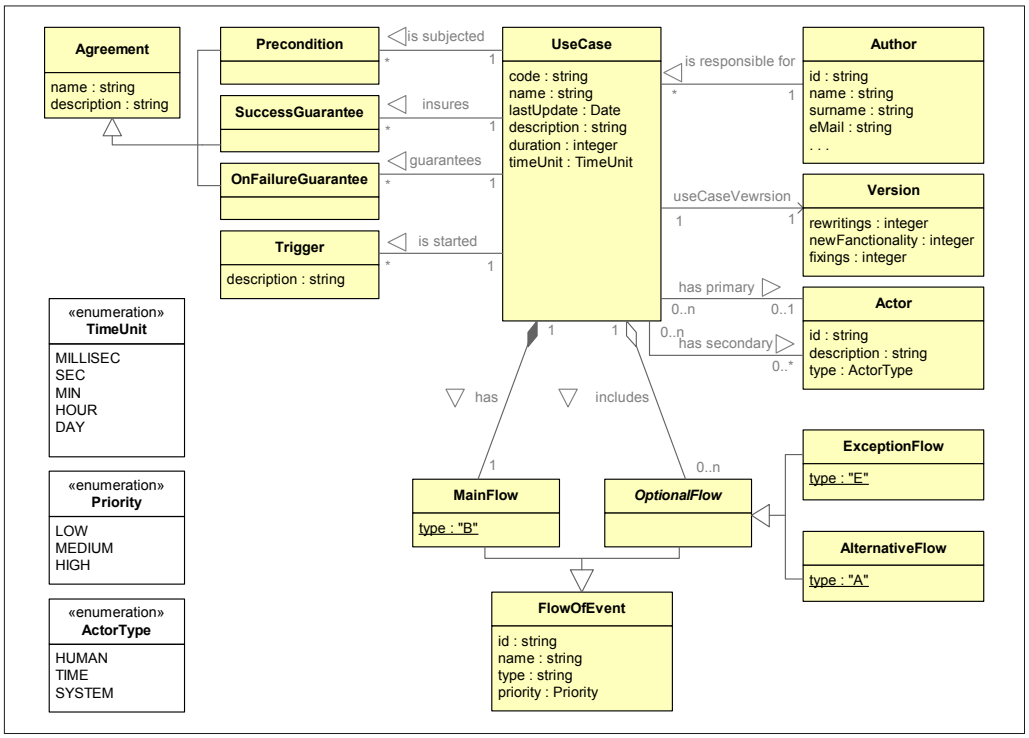
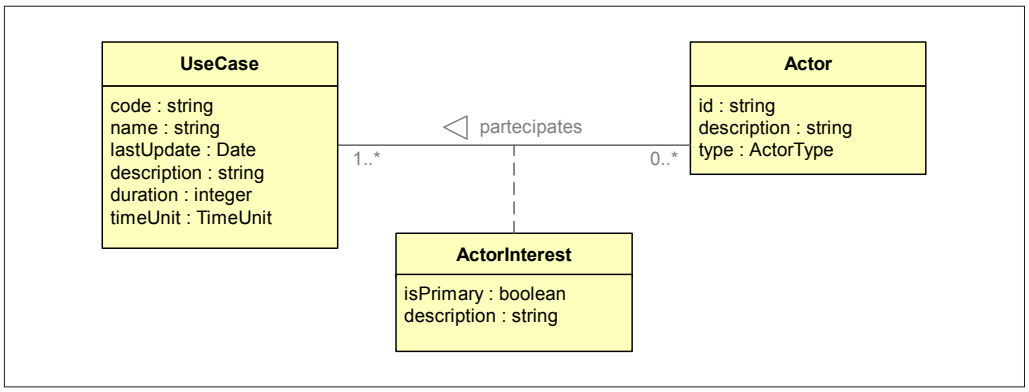


Figura 7.53 — *Frammento di modello che permette di evidenziare quale sia lo scopo di un attore nell'interagire con uno specifico caso d'uso.*



Il flusso primario (`MainFlow`) è uno, e uno solo, mentre quelli opzionali (`OptionalFlow`) sono zero o più. Lo scenario primario viene anche detto *best scenario* poiché, nella relativa stesura, si assume che tutto funzioni correttamente e non insorga alcuna condizione anomala. I flussi degli eventi opzionali possono essere di due tipi (si noti che `OptionalFlow` è una classe astratta e quindi deve essere necessariamente specializzata):

1. alternativi (`AlternativeFlow`): si tratta di percorsi che possono ancora portare al successo del caso d'uso, ma attraverso un percorso diverso da quello canonico sancito nel flusso principale;

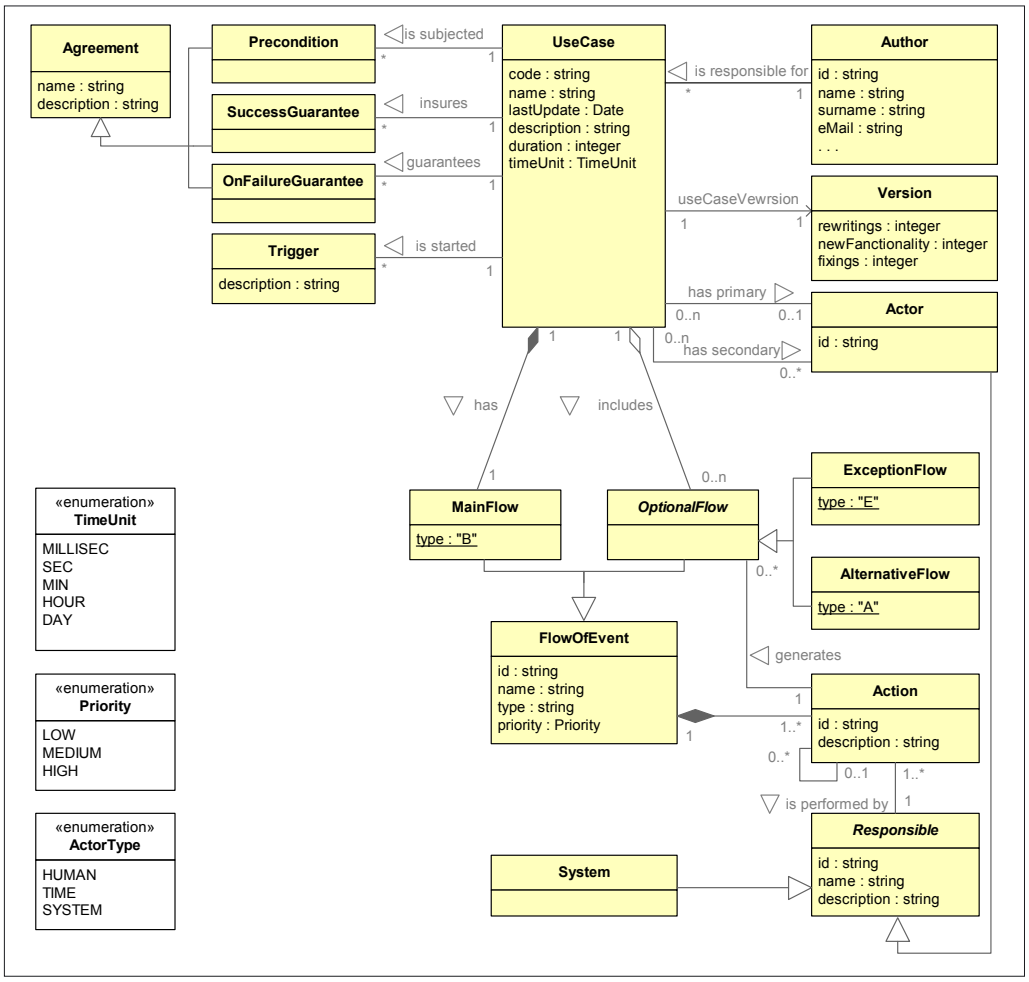
2. di errore (`ExceptionFlow`): sono i flussi generati da una condizione di errore e quindi forniscono dettagli su come gestire la relativa anomalia.

Ai vari flussi è associata una lettera identificativa: ciò si rende necessario per il modello dei test case mostrato successivamente, ove è necessario sapere con precisione a quale flusso appartenga ogni passo.

Nella classe astratta `FlowOfEvents` è presente un attributo `priority` di tipo enumerato. Nel caso in questione sono stati previsti unicamente i valori `LOW`, `MEDIUM` e `HIGH`. Chiaramente nulla vieta di utilizzare più elementi in modo da ottenere indicazioni di maggior dettaglio.

> L'attributo di priorità è di interesse per il processo (*workflow*) di *Project Management* (gestione del progetto) tipicamente presenti nei processi formali. In particolare le priorità sono utilizzate per controllare i fattori di rischio presenti nel progetto attraverso pianificazioni accurate delle iterazioni. Il buon senso (e anche i processi di sviluppo formali) suggeriscono di affrontare prima i rischi maggiori (ma senza esagerare), sia per poter gestire eventuali crisi del progetto il prima possibile quando ancora si ha tempo e non si è sottoposti allo stress della consegna, sia per minimizzare l'eventuale spreco di investimenti qualora si rendano necessari ripensamenti legati allo spazio delle soluzioni.
>
> I fattori di rischio, in prima analisi, possono essere suddivisi in due categorie: tecnici (per esempio l'architettura progettata non è in grado di garantire i requisiti non funzionali) e d'uso (il sistema non fornisce i servizi per i quali era stato inizialmente ideato). L'attributo in questione fa riferimento a rischi del secondo tipo; nulla vieta però di inserire ulteriori priorità: una relativa ai rischi tecnici e una finale, risultato di una media pesata delle due precedenti.
>
> I fattori di rischio possono variare all'interno di uno stesso use case. Per esempio, potrebbe aver senso assegnare una priorità elevata a uno scenario principale, e prevederne invece di più modeste per quelli opzionali o viceversa. Ciò ha senso considerando che le iterazioni, tipicamente, non prendono in considerazione solo interi casi d'uso, bensì raggruppano insiemi di scenari.

Figura 7.54 — *Dettaglio dei flussi degli eventi.*

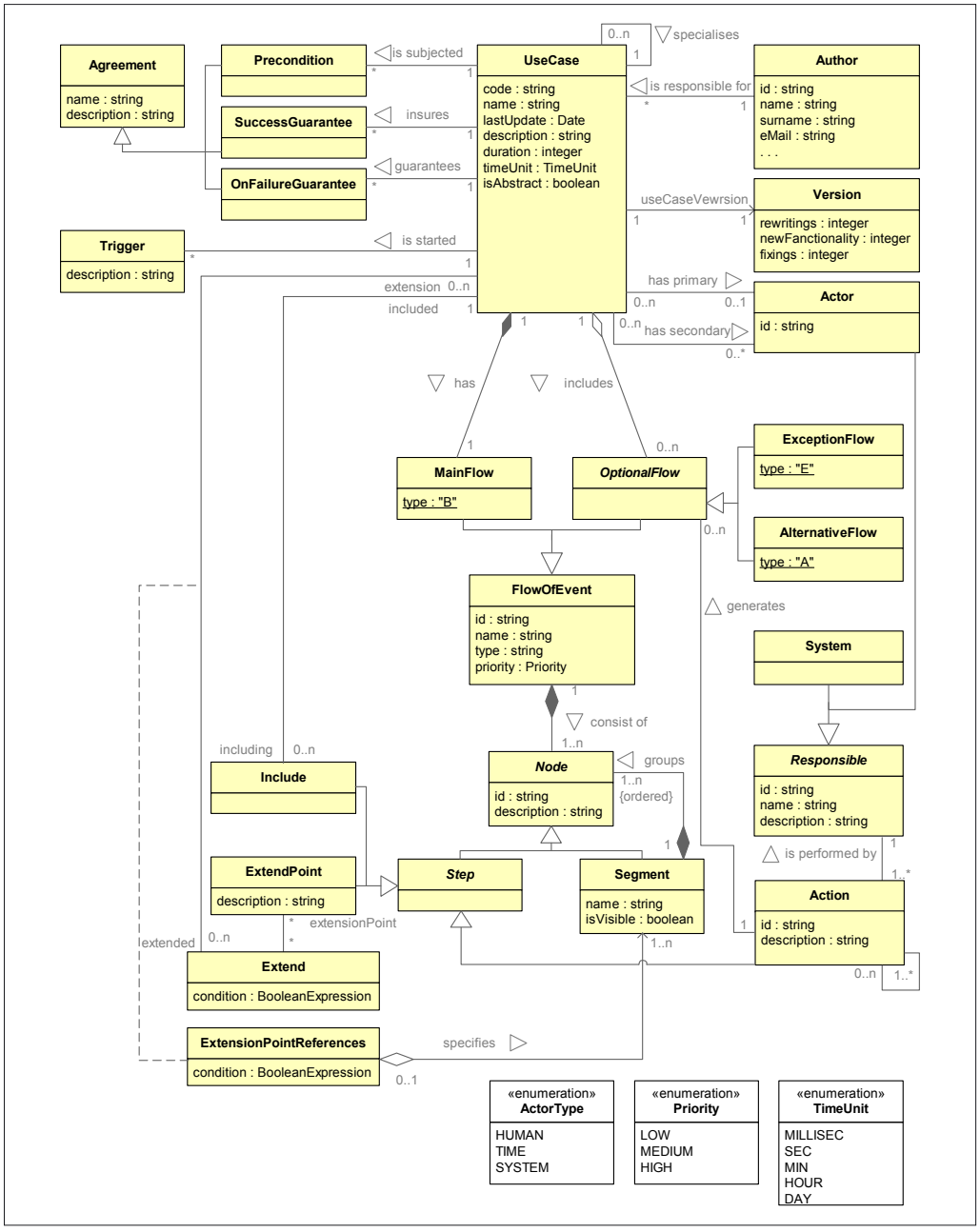


Obiettivo del passo successivo è definire chiaramente la struttura dei flussi degli eventi. In prima analisi, è possibile notare come ciascun flusso sia costituito da una serie di passi — azioni (`Action`) — mentre ogni azione può appartenere, al più, a un solo flusso.

L'autoassociazione della classe `Action` mostra che una particolare azione può dipendere da un'altra. Per esempio l'azione "`1.2.2`" appartiene al sottoflusso dipendente dall'azione "`1.2`". In generale, un'azione può averne diverse dipendenti, mentre una può dipendere esclusivamente da un'altra.

Il fatto che i flussi opzionali siano diramazioni del flusso principale, originate da particolari passi, è mostrato dalla relazione `generates` che lega la classe `Action` a quella

Figura 7.55 — *Modellazioni delle relazioni attraverso le quali è possibile interconnettere i casi d'uso.*



`OptionalFlow`. Tipicamente, la presenza nel flusso degli eventi di azioni con descrizione del tipo "verifica", "controlla", "tenta", "assicura", ecc. è il segnale lampante che tale azione genera flussi alternativi o di errore. Quindi, una particolare azione può generare nessuno o diversi flussi opzionali, mentre ciascuno di essi dipende da un particolare passo nel flusso degli eventi.

Nei casi d'uso è poi molto importante mostrare esplicitamente chi esegue una particolare azione. In questo caso gli esecutori possono essere esclusivamente di due tipi: il sistema o uno degli attori. Questa regola è espressa formalmente attraverso la relazione `is performed by` che lega la classe `Action` con quella astratta `Responsible`. Si tratta di una classe astratta che prevede due specializzazioni: `System` e `Actor`.

A questo punto è giunto il momento di considerare le relazioni che permettono di collegare tra loro i vari casi d'uso (fig. 7.55)… Argh! Qui la situazione si complica.

Si proceda per gradi. In primo luogo è stato aggiunto l'attributo `isAbstract` di tipo booleano nella classe `UseCase`, al fine di indicare se uno specifico caso d'uso sia astratto o meno. Sempre nella stessa classe è stata aggiunta l'auto-relazione `specialises`. Questa indica che uno stesso caso d'uso può essere specializzato da diversi altri, mentre ciascuno di essi ne può specializzare al più uno solo.

Si ricordi che i casi d'uso costituiscono il veicolo preferenziale per catturare i requisiti utente e quindi per interagire con personale che, per definizione, ha poca o nessuna esperienza informatica. Pertanto, complicare inutilmente i casi d'uso con **sofisticate** relazioni di generalizzazione di certo non aiuta a conseguire l'obiettivo primario: comunicare.

Per descrivere i flussi, è stato aggiunto il pattern Composite (probabilmente si tratta più di un Decorator). Uno dei pregi è relativo alla facoltà di rappresentare facilmente strutture intrinsecamente ricorsive come alberi. In questo contesto permette di avere o passi semplici, oppure composti e quindi rappresentati attraverso appositi segmenti, eventualmente costituiti, a loro volta, da passi semplici e/o altri segmenti, e così via. L'elemento base che termina la ricorsione è un oggetto `Step` che prevede tre specializzazioni: `Include`, `Extend` e `Action`. È stato necessario definire il concetto di segmento (`Segment`) per via della relazione di estensione. In tutti i casi in cui uno specifico use case non ne estende un altro, può non avere importanza raggruppare i passi in segmenti. In tali casi, i flussi potrebbero essere composti da un unico segmento di default la cui presenza non è di grande interesse (`isVisible = false`) in quanto trattasi essenzialmente di un artificio.

Nella classe `Step` sono state migrate diverse caratteristiche della classe `Action`. Si è deciso di dar luogo a tale ulteriore strutturazione gerarchica in modo da distinguere i passi (istanze delle classi che specializzano `Step`) relativi a relazioni, da quelli ordinari. Esistono due specializzazioni dei passi relativi a relazioni: `Include` ed `ExtensionPoint`. Le inclusioni possono essere considerati "passi attivi" e prevedono l'indicazione esplicita del caso d'uso incluso, per questo motivo è prevista la associazione con la classe *UseCase*. Un esempio classico è l'inclusione dello use case di reperimento dati articoli. In un flusso degli eventi, tale passo assumerebbe una forma del tipo: `Include(Reperimento dati`

`articolo)`. Da notare che uno stesso caso d'uso può essere incluso in molti altri e viceversa. Chiaramente in uno specifico passo del flusso degli eventi è possibile includere un solo caso d'uso.

Riguardo la relazione di estensione, nello use case base (quello che deve essere esteso), essa assume una forma per così dire passiva: si limita a dichiarare esplicitamente le locazioni in cui il comportamento dinamico deve essere esteso. Per questo motivo, un passo del flusso degli eventi può dichiarare al più un solo punto di estensione, il cui nome (`definition`) è specificato nella classe `ExtensionPoint`. Per un singolo punto di estensione possono essere previste diverse condizioni (per esempio lo use case base viene esteso da diversi altri) e una stessa condizione può essere applicata a diverse locazioni del caso d'uso base. Un use case può estendere ed essere esteso da diversi altri. Qualora un use case ne estenda un altro, è necessario che il primo fornisca i segmenti di comportamento da sostituire ai punti di estensioni dichiarati nel caso d'uso base. Ciò è modellato attraverso la classe di associazione `ExtensionPointReferences`.

A questo punto il modello possiede una forma abbastanza matura e non resta altro da fare che aggiungere informazioni supplementari (ciò che tecnicamente rientra nella categoria dei "fiocchetti") relative, perlopiù, all'utilizzo dei casi d'uso nel contesto di processi formali e alla gestione di eventuali feedback (*cfr* fig. 7.56).

Per esempio è possibile associare a uno use case una lista di assunzioni, lo stato di elaborazione in cui si trova (base, elaborato, definito sinteticamente e definito completamente), eventuali problemi ecc.

Modello dei test case

Obiettivo della presente sezione è di presentare una versione del modello del dominio dei test case, realizzando un diagramma delle classi che descriva formalmente le entità, corredate dalle relative relazioni, che permettono di descrivere rigorosamente i test case.

L'argomento dei test case è stato trattato nel Capitolo 5.

Prima di procedere nello studio del modello dei test case, si vuole ricordare ancora una volta che questi dipendono fortemente dagli use case. Pertanto se si è realizzato un modello dei requisiti utente (attraverso il formalismo dei casi d'uso), e questi sono chiari, accurati e completi, allora la pianificazione dei test case si presenta come un'attività piuttosto immediata. Diversamente, l'esercizio diventa decisamente più complesso e non sempre di sicuro effetto o di qualche utilità. Per esempio, si può correre il rischio di non verificare alcuni percorsi di estrema importanza, di non riuscire a definire correttamente i "prodotti" attesi dal sistema come risultato di uno specifico percorso, ecc. Da quanto riportato poc'anzi, è evidente che anche la pianificazione dei test case rappresenta uno strumento per verificare la qualità del modello dei casi d'uso.

Figura 7.56 — *Inclusione delle informazioni relative al processo.*

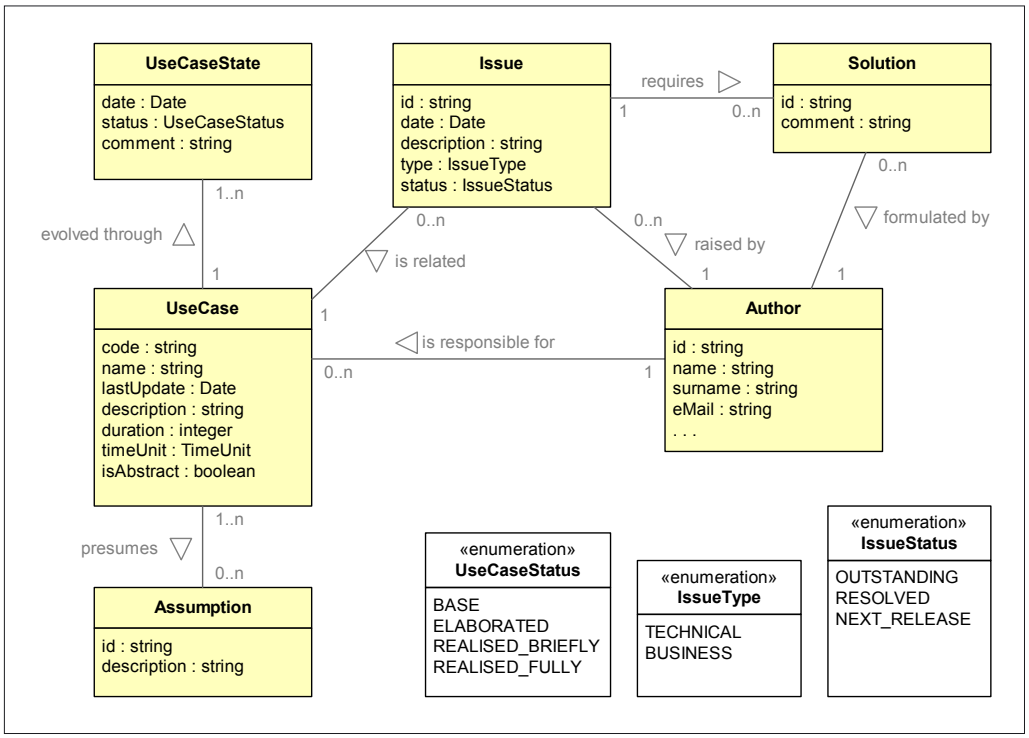
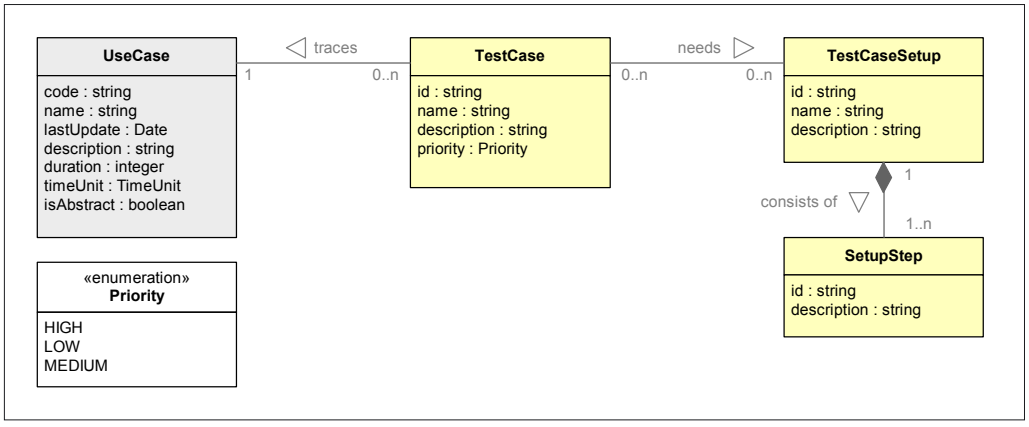


Figura 7.57 — *Embrione del modello ad oggetti dei test case.*



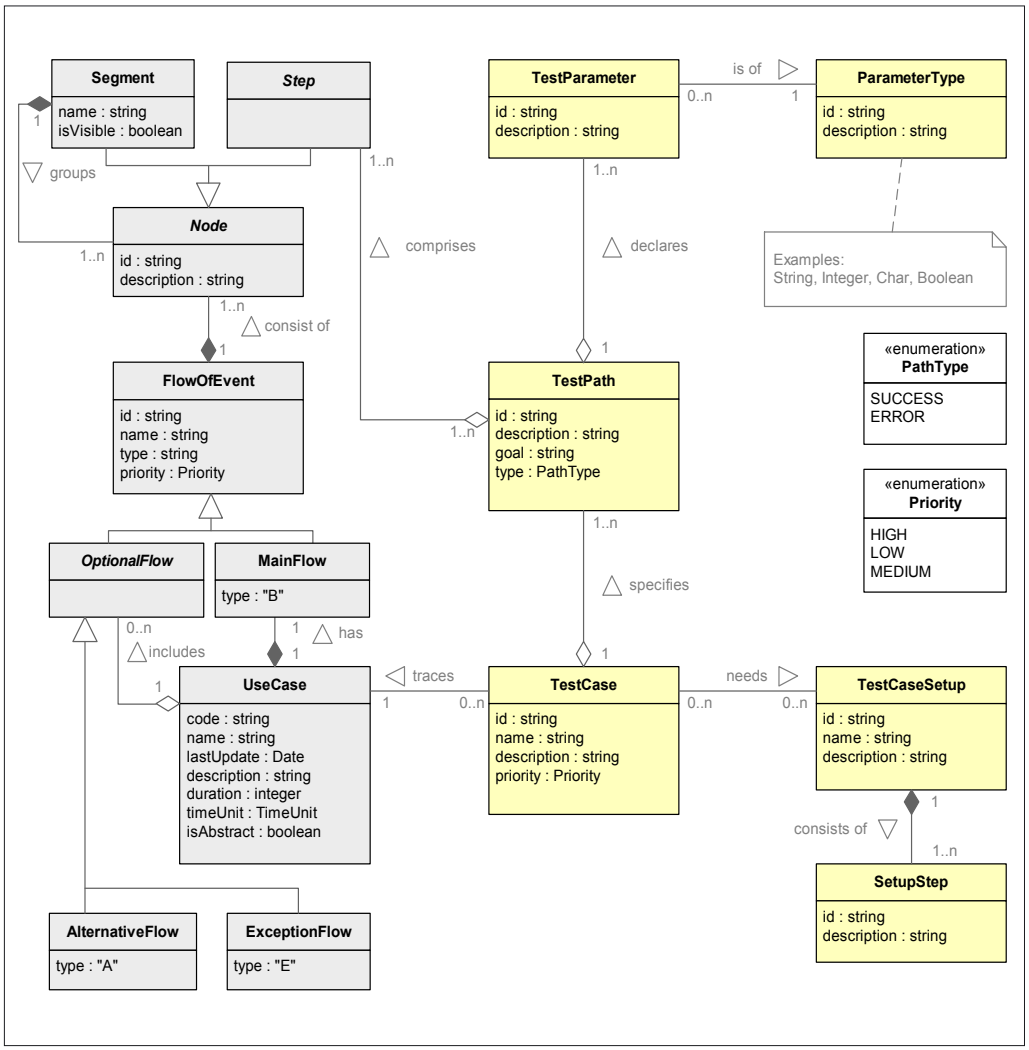
In maniera del tutto analoga al caso precedente, si cominci a considerare la classe principale denominata `TestCase` che, in qualche misura, si presta ad essere la radice del modello (*cfr* fig. 7.57). Gli attributi di interesse sono, al solito, `id`, `name`, `description` e `priority`.

Ogni caso di test è utilizzato per verificare il funzionamento di uno e un solo caso d'uso, mentre ciascuno di questi, generalmente, richiede diversi test (relazione `traces`). Le procedure di verifica, prima di poter essere eseguite richiedono tipicamente una serie di inizializzazione o passi preliminari. Per esempio, se si volesse verificare un sistema per la gestione dei conto correnti dei clienti di una banca, bisognerebbe eseguire un'attività propedeutica di caricamento della base di dati (inserimento di un nuovo cliente, di un conto, versamento di un predefinito importo nel conto, ecc.), al fine di dotare il sistema di una base conoscitiva su cui poi condurre le varie verifiche. Le procedure di inizializzazione si prestano a essere modellate in svariati modi con diversi livelli di formalità e flessibilità. In questo contesto si è deciso di utilizzare quella più semplice in quanto più flessibile: riportare attraverso descrizione testuale l'elenco ordinato dei passi da eseguire. Nel modello di fig. 7.57, le procedure di inizializzazione sono rappresentate mediante la classe `TestCaseSetup`. Questa può essere utilizzata da diversi test case, così come uno stesso test case può utilizzare diverse procedure di inizializzazione (relazione `needs`). Ogni istanza della classe `TestCaseSetup` è composta da un insieme di passi (`SetupStep`) propri della procedura di inizializzazione e non condivisibili con altre procedure.

Il passo successivo consiste nel dettagliare i percorsi di test previsti dai test case e nel mostrare le relative dipendenze dallo use case di riferimento. Le informazioni a cui si è particolarmente interessati nei test case sono i percorsi da seguire durante ciascun test. Nel diagramma di fig. 7.58, questi percorsi sono rappresentati dalla classe `TestPath`. Ciascuno di essi è identificato da un apposito `id`, prevede una descrizione, specifica gli obiettivi che si devono conseguire con la relativa esecuzione, enumera esattamente quali passi dello use case eseguire e dichiara gli eventuali parametri necessari. L'enumerazione dei passi dello use case da eseguire avviene attraverso la relazione `comprises`. Chiaramente un medesimo passo di uno specifico use case può essere incluso in svariati percorsi test (oggetti di tipo `TestPath`), così come ciascun percorso prevede l'esecuzione di diversi passi del caso d'uso sotto verifica.

Per esempio un percorso di test potrebbe selezionare tutti i passi dello scenario principale al fine di verificare che, partendo da una situazione valida e sottoponendo stimoli corretti, il sistema risponda con gli effetti attesi. Si potrebbe poi impostare un percorso che generi una condizione di errore (flusso di errore) al fine si verificare la capacità del sistema di riconoscere la particolare situazione anomala e quindi reagire come da specifiche. Un percorso di test in genere necessita di diversi parametri (relazione `declares`). Ciò si rende necessario poiché i casi d'uso specificano un comportamento generale valido per tutti i possibili dati di input, mentre i test devono avvenire su dati ben definiti.

Figura 7.58 — *Sviluppo del modello: definizione del concetto di percorso di test.*



Per esempio, dopo aver popolato la base dati del sistema di cui sopra, si vuole verificare che esso effettui correttamente l'addebitamento di un importo. In questo caso, il percorso di test da eseguire dovrebbe utilizzare diversi parametri: l'identificatore del possessore del conto corrente, il relativo codice, l'importo da addebitare, ecc.

A questo punto l'ossatura del modello è definita e non resta altro da fare che modellare l'esecuzione del test case. In particolare è necessario disporre di informazioni quali: data

dell'esecuzione, versione del sistema, nome del tester, esito della verifica di ogni test path, eventuali errori individuati, ecc.

In primo luogo uno stesso percorso di test può essere eseguito molteplici volte (relazione `runs` tra la classe `TestPath` e `TestCaseIteration`). Ciò è abbastanza naturale considerando che a ogni nuova versione del sistema bisognerebbe eseguire nuovamente

Figura 7.59 — *Modello finale dei test case.*

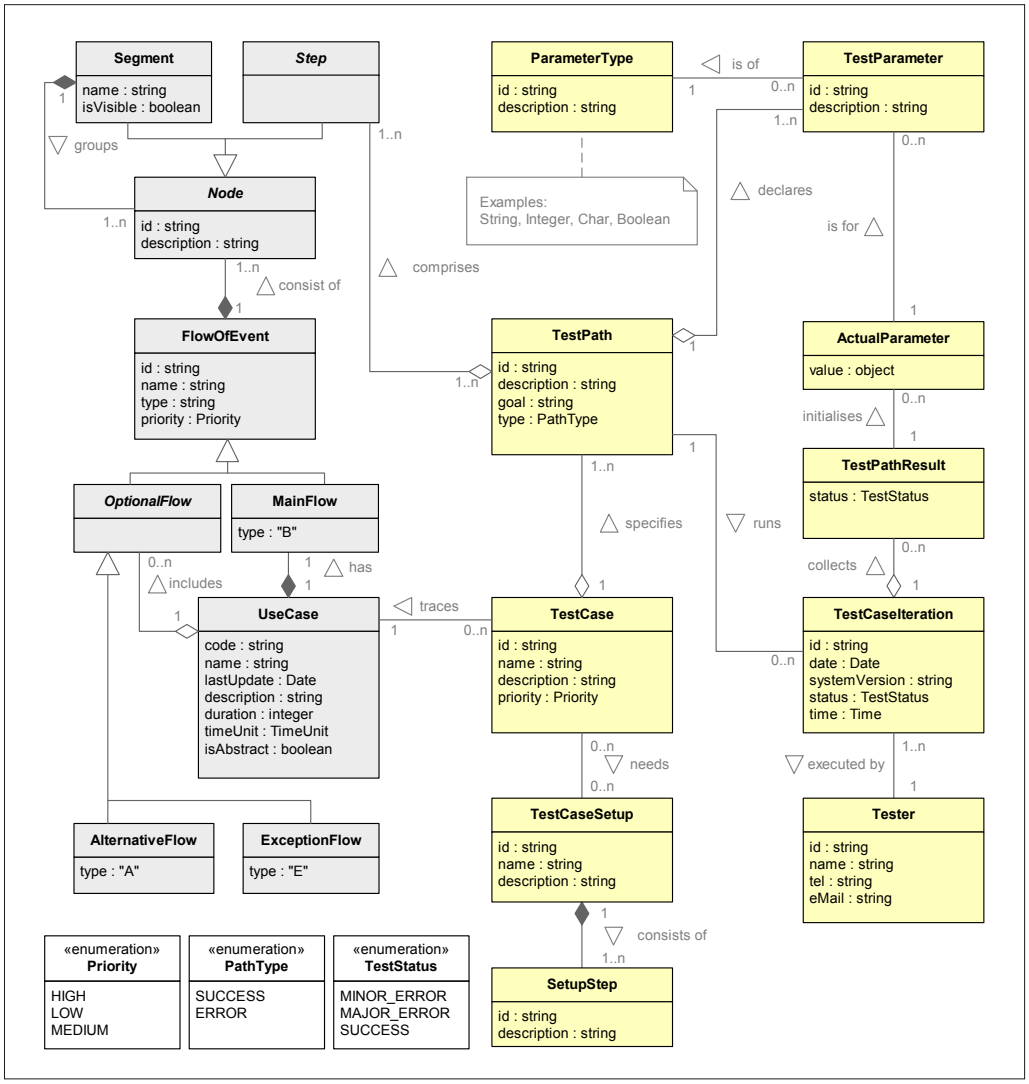
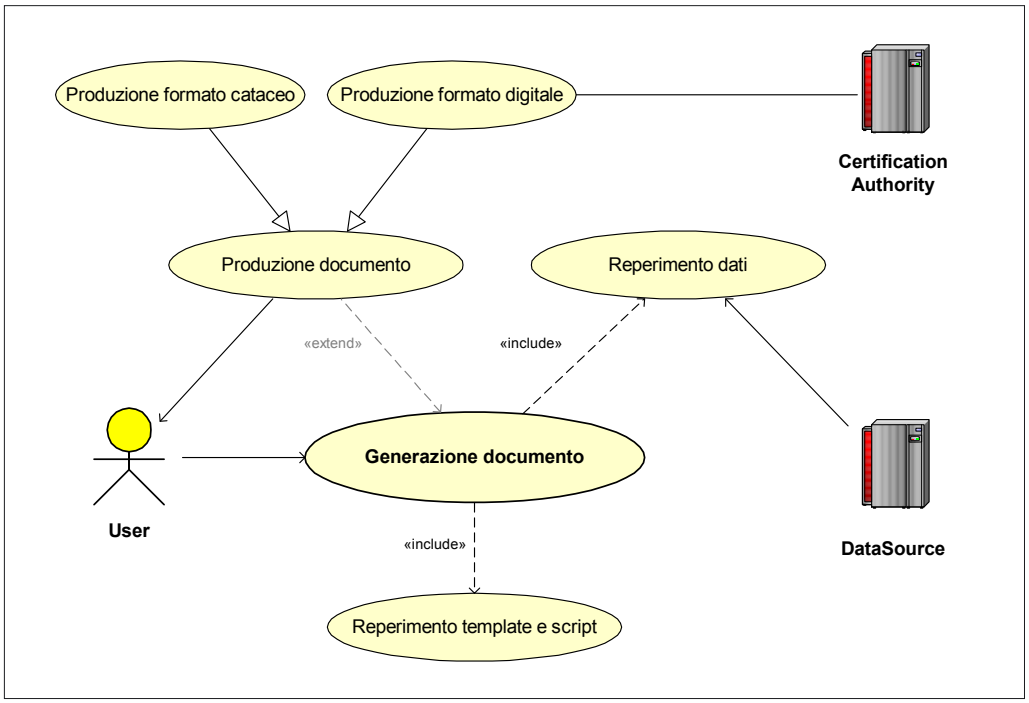


Figura 7.60 — *Use case relativo alla generazione e produzione del documento. La specializzazione dello use case* `Produzione documento` *è dovuto al fatto che, in funzione del supporto, debbano essere eseguite diverse procedure. Per esempio la stampa su carta potrebbe richiedere un'operazione di impaginazione, mentre il formato destinato a una smartcard dovrebbe essere strutturato in funzione alla specifica tipologia.*



tutte le verifiche necessarie, eventualmente eliminandone alcune divenute non più applicabili, aggiornandone altre e inserendone delle nuove. Ogni test è eseguito o controllato da un `Tester`, il quale, ahimè, è tipicamente gravato dal compito di effettuare svariati test. L'esecuzione di un percorso di test produce dei risultati che spesso è opportuno registrare. A tal fine è presente la classe `TestPathResult`.

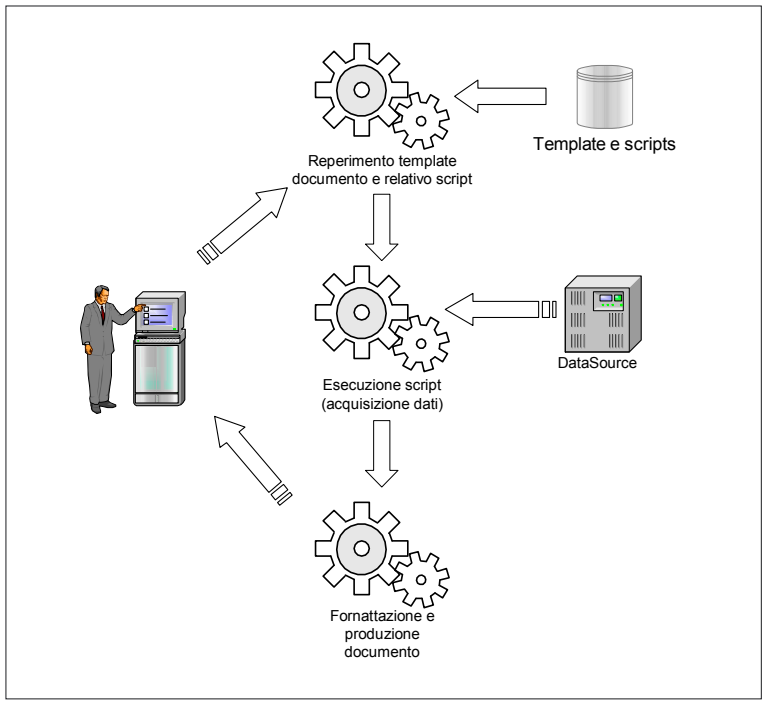
Un esempio più tecnico: generatore automatico di certificati

Presentazione

Si presenterà ora un sistema che ha fatto parte dei desideri reconditi dell'autore… Lavorando nel settore IT si tende a contrarre una strana patologia denominata *informatizzazione o ottimizzazione indiscriminata*, il cui sintomo principale è dato dall'in-

sofferenza all'inutile uso della risorsa tempo. La manifestazione classica del sintomo avviene qualora, per qualsivoglia motivo, si incappi in lungaggini burocratiche e/o in strane procedure manuali. In tali evenienze, nel cervello automaticamente si accende la famosa lampadina "si potrebbe fare tutto più velocemente e meglio con un opportuno sistema informatico"… E qui la vocina interna dell'autore è letteralmente presa da una crisi di delirio. L'esempio presentato in questo paragrafo rientra in questa categoria. A chi non è mai capitato di dover perdere intere mattinate in fila per ottenere un banale certificato? Ebbene l'idea è quella di realizzare un sistema a servizio del cittadino in grado di fornire una serie predefinita di documenti. Il sistema potrebbe essere fruito, in un primo momento, attraverso apposite postazioni (*kiosk*) e in successive evoluzioni direttamente dalla classica poltrona — ormai logora a forza di esempi — della propria abitazione… Chiaramente i "chioschi" potrebbero erogare documenti stampati su appositi moduli filigranati, mentre da una normale postazione Internet si potrebbero ottenere file firmati elettronicamente. Con un lettore/scrittore di smart card si potrebbe poi caricare nella propria carta il documento firmato digitalmente e quindi consegnarlo al destinatario o, più semplicemente, inoltrarlo via posta elettronica...

Figura 7.61 — *Descrizione delle componenti principali del processo per la produzione dei documenti richiesti.*



> Da più di venti anni si sente parlare di uffici "senza carta". L'autore ha sempre pensato che, con tale espressione, ci si riferisse all'utilizzo di documenti digitali al fine di ridurre il consumo mondiale di carta; forse invece si intendeva parlare della mancanza di carta dovuta all'intenso utilizzo.

In ogni modo, l'obiettivo è realizzare un sistema in grado di fornire tutta una serie di documenti standard utilizzando appositi template. I dati da inserire in questi template sono richiesti direttamente alle sorgenti (*datasource*): sistemi anagrafici, casellario giudiziario, università, scuole superiori, banche, ecc. Chiaramente, per rendere tutto ciò possibile, ognuno di questi fornitori di dati dovrebbe offrire apposite interfacce (i famosi *wrapper*) dei loro sistemi, a cui richiedere informazioni utilizzando interfacce predefinite. Il traffico dei dati deve poi avvenire sotto il vigile controllo di un robusto sistema di sicurezza.

In contesti come questi, la flessibilità assume un ruolo di primaria importanza: dovrebbe essere sempre possibile aggiungere nuovi template di documenti, nuove sorgenti di dati, ecc. senza dover ogni volta modificare il sistema. Pertanto, più che realizzare un singolo sistema, l'obiettivo è realizzare un apposito framework da personalizzare per le varie esigenze.

Costruzione del modello

Si focalizza ora l'attenzione su una serie di componenti del sistema. Particolare attenzione è conferita al motore in grado di raccogliere i dati (*Data Collector Engine*) sparsi per le varie sorgenti, necessari alla produzione dei documenti. Sono illustrati alcuni componenti ritenuti particolarmente significativi, data l'impossibilità di presentarli tutti in maniera diffusa.

Le parti proposte supportano la realizzazione di un semplice interprete che, a seguito della richiesta di produzione di uno specifico documento, sia in grado di inizializzare e quindi eseguire un apposito grafo di oggetti rappresentanti il "programma" da eseguire per il reperimento dati. Il relativo diagramma dei casi d'uso è mostrato in fig. 7.60, mentre nella successiva fig. 7.61 viene illustrato in maniera semplificata il workflow.

L'associazione tra tipologia di documento e dettaglio del relativo programma da eseguire, per questioni di flessibilità, dovrebbe essere dichiarativa. In altre parole, è opportuno specificare gli script attraverso appositi file di configurazione (verosimilmente XML) che, per questioni di efficienza, potrebbero essere caricati in memoria all'avvio del sistema. Sempre per questioni di flessibilità, il sistema dovrebbe tentare di acquisirli ogniqualvolta non siano disponibili in memoria. Questo meccanismo permetterebbe di aggiungere dinamicamente nuove tipologie di documentazione, o modificarne altre, senza dover ricompilare il sistema, e tanto meno, senza dover interrompere la fruizione dello stesso.

Per cominciare, si analizzi una semplice istruzione. La relativa struttura, dal punto di vista logico, potrebbe avere la seguente forma:

```
<parametro di output> {, <parametro di output>}
    = <sorgente dati>.<funzione da invocare> ( {<parametro di input>})
```

Un esempio di un semplice script potrebbe essere:

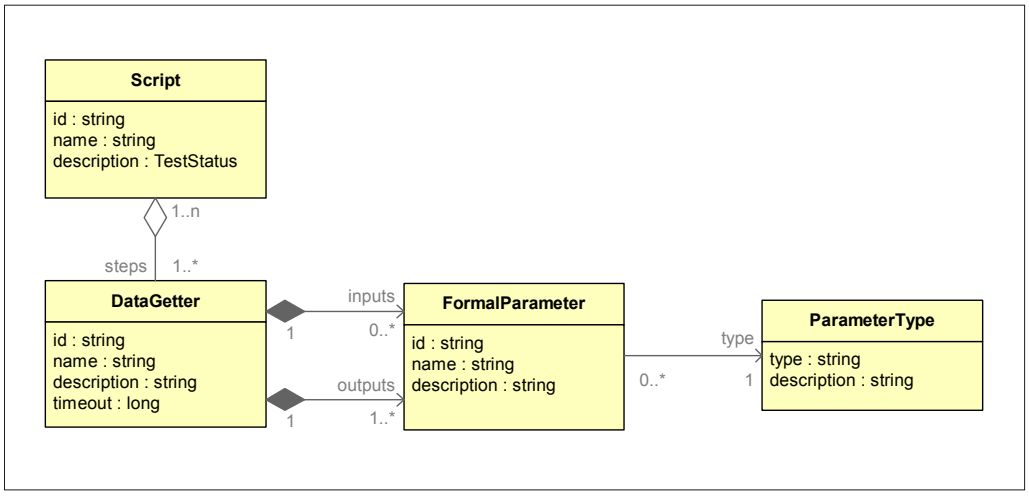
```
userFiscalCode    = smartcard.getFiscalCode();
name              = smartcard.getName();
surname           = smartcard.getSurname();
dateOfBirth       = utilityAnag.getDateFromFC(userFiscalCode);
offencesCommitted = casellarioGiudiziario.getOffences(userFiscalCode, name,
                                                      surname,     dateOfBirth);
```

L'esempio — che per ovvi motivi risulta un po' artificioso — prevede tre sorgenti di dati, di cui due esterne e una interna. Le `DataSource` esterne sono:

- la smartcard, da cui prelevare le informazioni relative all'utente;
- il casellario giudiziario, in grado di fornire eventuali imputazioni a carico dell'utente;

La sorgente interna è costituita da un componente denominato `utilityAnag` in grado di eseguire elaborazioni sui dati anagrafici forniti in input.

Figura 7.62 — *Modello dello script per il reperimento dei dati.*



Una volta reperito lo script, si tratta di eseguirne tutte le istruzioni. Da tener presente che l'esecuzione di alcune funzioni potrebbe essere bloccata da quella di altre. Per esempio l'ottenimento dello stato penale o, nel caso peggiore, della lista delle condanne inflitte all'utente, è subordinata al reperimento del codice fiscale, del nome, del cognome e della data di nascita di un individuo.

Un buon punto di inizio consiste nel modellare le funzioni da eseguire, denominate `DataGetter` (i singoli passi di cui è composto un programma).

> Nelle classi del modello successivo non è stato visualizzato il compartimento dedicato ai metodi. Ciò è dovuto al fatto che si tratta di classi entità appartenenti al modello del dominio, e quindi i relativi metodi sarebbero i classici `get/set`.

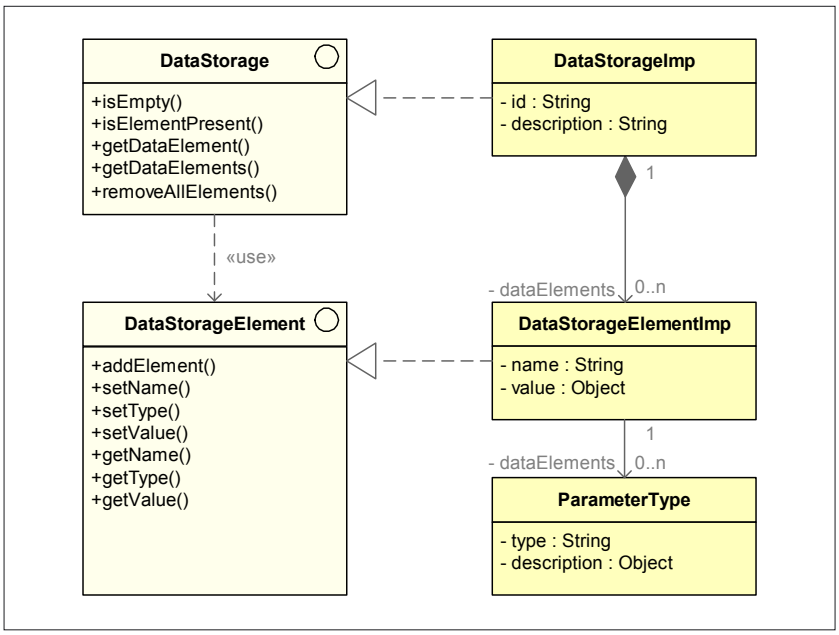
Uno script è costituito da diverse funzioni e ciascuna di esse può essere utilizzata in diversi script (relazione di aggregazione tra le classi `Script` e `DataGetter`). Analizzando la struttura della classe `DataGetter` si nota che si tratta di funzioni che, nel caso generale, consentono di produrre come risultato diversi valori. Tra i vari attributi, ne compare uno denominato `timeout`. Ciò è necessario per evitare che l'esecuzione di uno script possa rimanere in uno stato di attesa indefinita, qualora una funzione non possa reperire i dati dalla relativa sorgente a causa di inconvenienti (per esempio congestione del traffico di rete). Pertanto l'attributo `timeout` è utilizzato per interrompere l'esecuzione di una specifica richiesta di dati qualora questi non pervengano entro uno specifico lasso di tempo.

Diverse classi possiedono campi descrittivi assolutamente inutili per il sistema. Questi però assumono un'importanza notevole per la realizzazione di meccanismi di logging, per il debug e soprattutto per la realizzazione di interfacce utente atte a realizzare le installazioni del framework (i famosi script). Per esempio, dovendo dichiarare la struttura di uno script, potrebbe essere di grande aiuto riuscire a individuare le varie istanze della classe `DataGetter` da inserire, attraverso i valori degli attributi nome e descrizione.

L'esecuzione di ciascuna funzione necessita di ricevere i valori di specifici parametri formali e produrre quelli di output. I parametri formali sono rappresentati dalla classe `FormalParameter`. Pertanto, un oggetto di tipo `DataGetter` può disporre di nessuno o di una serie di parametri formali di input e di almeno uno di output.

I risultati generati dall'esecuzione di ciascuna istanza `DataGetter` devono poter essere gestiti dal sistema. Pertanto è necessario realizzare un apposito meccanismo per la memorizzazione e il ripristino dei dati (`DataStorage`, fig. 7.63). In contesti come questi, l'utilizzo di interfacce, e più in generale del polimorfismo, rappresenta un meccanismo obbligatorio. In particolare si vogliono utilizzare degli oggetti che implementano un'opportuna interfaccia, senza però curarsi dei dettagli della relativa implementazione. Il vantaggio è che quest'ultima può variare senza generare ripercussioni sulle classi clienti: meccanismi come quello per la memorizzazione dei risultati si prestano a essere utilizzati am-

Figura 7.63 — *Modello per la realizzazione di un meccanismo di memorizzazione dei dati (*data storage*). Le classi* `DataStorageImp` *e* `DataStorageElementImp` *sono visualizzate senza la sezione dei metodi, giacché quelli più interessanti sono indicati nelle relative interfacce.*

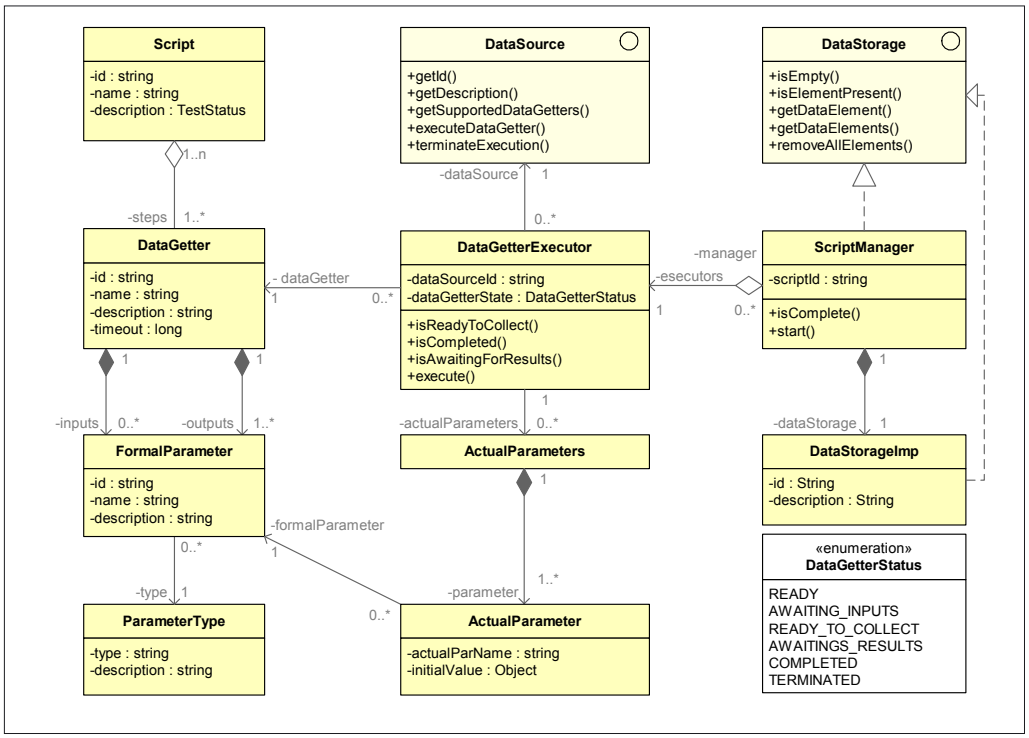


piamente in diverse parti del sistema, pertanto la variazione dell'implementazione non genera ripercussioni in quanto "protetta" da apposite interfacce.

Il modello di fig. 7.63 mostra come un'istanza di `DataStorage` sia costituita da un insieme di elementi, `DataStorageElement`. Ciascuno di questi elementi appartiene esclusivamente a un `DataStorage` (ciò è evidenziato dalla presenza della relazione di composizione). Entrambe le classi prevedono un'apposita interfaccia, ciò fa si che l'implementazione del `DataStorage` e dei relativi elementi possa variare senza avere ripercussioni sulle restanti parti del sistema, e che lo stesso possa utilizzare diverse versioni (per esempio una persistente e una no) per diversi fini.

A questo punto è necessario inserire delle classi di gestione in grado di coordinare e controllare l'esecuzione dei vari script. In altre parole è necessario inserire la classe le cui istanze hanno la responsabilità di gestire l'esecuzione degli oggetti `DataGetter` che costituiscono i vari script. I modelli presentati finora erano essenzialmente costituiti da elementi statici: rappresentazioni formali di oggetti appartenenti al dominio considerato di carattere passivo. Ora l'attenzione viene spostata sul motore in grado di far funzionare in maniera coordinata i vari ingranaggi (*cfr* fig. 7.64).

Figura 7.64 — *Meccanismo di esecuzione degli script.*



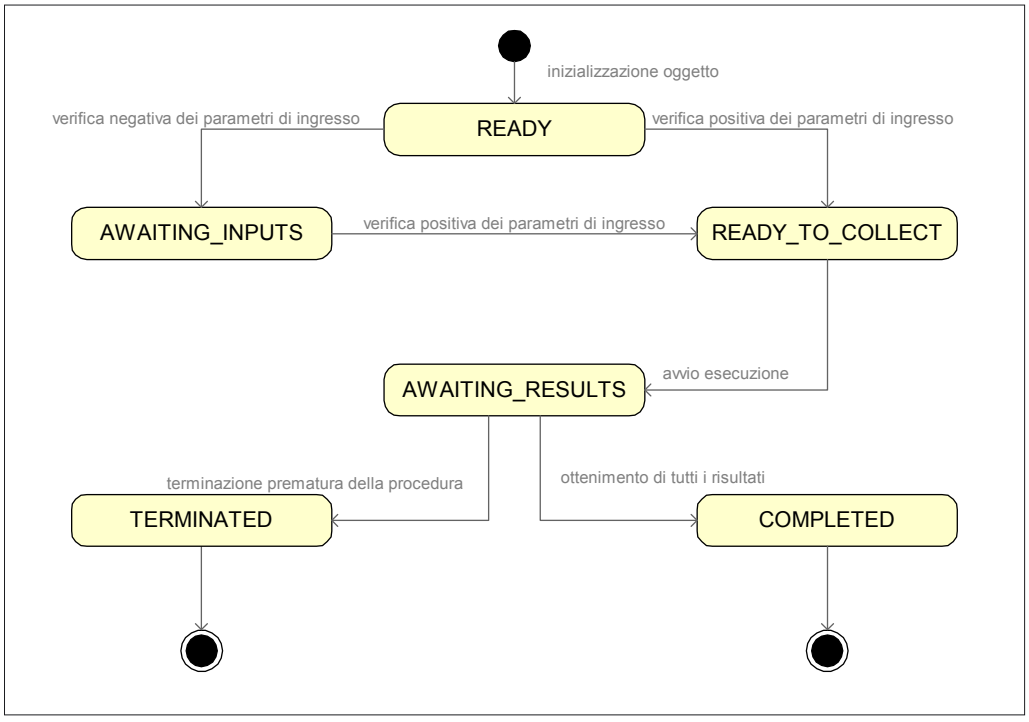
In primo luogo, per ogni passo dello script, ossia per ciascun oggetto `DataGetter`, è necessario selezionare il `DataSource` in grado di fornire i parametri richiesti. Anche il concetto di `DataSource` è un formidabile esempio di polimorfismo. Esistono diverse tipologie di sorgenti di dati: remote, locali, realizzate su diversi sistemi, con differenti tecnologie e funzionamenti ecc., tutte pilotate per mezzo di una medesima interfaccia (`DataSource` appunto); l'invocazione di uno stesso metodo, `executeDataGetter()` per esempio, genera un comportamento assolutamente diverso in funzione della particolare `DataSource` che lo implementa.

Ogni `DataGetter` necessita di un apposito oggetto (`DataGetterExecutor`) che si assume la responsabilità di coordinarne l'esecuzione (*cfr* fig. 7.64), di reperire i valori dei parametri di input, di memorizzare quelli prodotti in output e di effettuare l'associazione con la sorgente dati in grado di eseguirne la funzione. Chiaramente ogni `DataGetter` può essere inserito in più script, e quindi, in ciascuno di essi, può eventualmente essere associato con un diverso `DataGetterExecutor` (per esempio in un opportuno script la stessa funzione viene richiesta a un diverso `DataSource`). Le istanze della classe `DataGetterExecutor` agiscono a livello di singola funzione e pertanto devono essere a

loro volta controllate da un altro oggetto che opera a livello di script. Tale oggetto è lo `ScriptManager` che espleta due macroservizi: controllo dell'esecuzione dell'intero script e memorizzazione dei dati prodotti nel corso dell'esecuzione dei vari `DataGetter`. Quest'ultima capacità è enfatizzata dal fatto che `ScriptManager` implementa l'interfaccia `DataStorage` e quindi possiede anche un comportamento di immagazzinamento dati. È interessante notare come questa classe possa essere considerata a tutti gli effetti una specializzazione della classe `DataStorageImp`. Tale specializzazione non è ottenuta per mezzo della canonica relazione di generalizzazione, bensì attraverso una composizione; in altre parole, `ScriptManager` deroga a `DataStorageImp` la fornitura dei servizi di memorizzazione dei dati.

Ogni oggetto `DataGetterExecutor`, non appena avviato, inizializza il proprio stato interno: in altre parole, verifica che tutti i parametri di input di cui necessita il relativo oggetto (`DataGetter`) siano disponibili (*cfr* fig. 7.65). In caso di esito positivo, transita nello stato di pronto all'esecuzione (`Ready to collect`), in caso contrario, invece, transita nello stato di attesa parametri (`Awaiting inputs`). Questa verifica avviene mol-

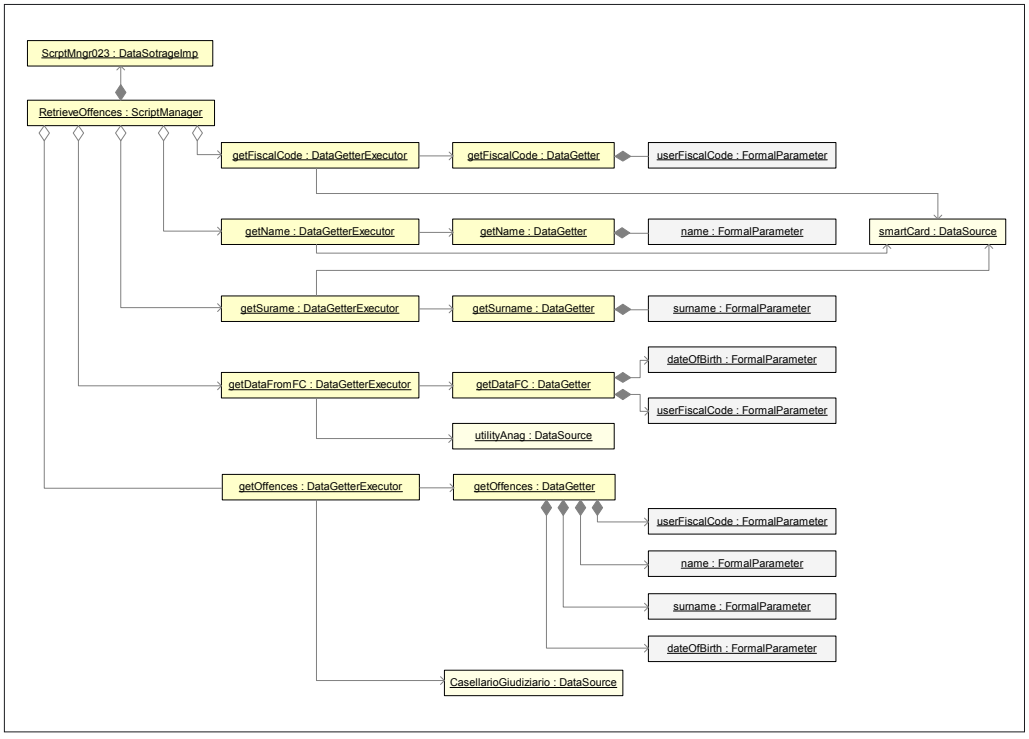
Figura 7.65 — *Ciclo di vita degli oggetti* `DataGetterExecutor`.



to semplicemente: ottiene dall'oggetto `DataGetter` la lista dei parametri di input, eventualmente ne cambia il nome come suggerito dagli oggetti `ActualParameter`, e quindi chiede conferma dell'esistenza all'oggetto `ScriptManager` (invoca il metodo `isElementPresent`) che, in questo contesto, si comporta da `DataStorage`. Come si può notare, la relazione tra `ScriptManager` e `DataGetterExecutor` non è dotata di navigabilità in quanto, come appena evidenziato, è necessario poter navigare gli oggetti in entrambi i sensi.

Qualora un oggetto `DataGetterExecutor` sia pronto a essere eseguito e ottenga il permesso dall'istanza `ScriptManager` (invocazione del metodo `execute`), acquisisce l'appropriata istanza dell'oggetto `DataSource` (si tratta dell'istanza di una classe in grado di gestire la comunicazione con il reale fornitore, un proxy), invoca il metodo `executeDataGetter` e si pone in uno stato di attesa dei risultati (`Awaiting results`). Ottenuti anche questi, li memorizza nel `DataStorage` gestito dall'oggetto `ScriptManager` e quindi si pone nello stato di completamento esecuzione (`Completed`).

Figura 7.66 — *Diagramma degli oggetti relativo a un ipotetico stato di esecuzione dello script mostrato in precedenza.*



L'evento di memorizzazione di un parametro nell'oggetto `ScriptManager` avvia una serie di processi. In primo luogo si verifica se l'intero script abbia terminato la propria esecuzione (tutti gli oggetti `DataGetterExecutor` si trovano nello stato `Completed`); in caso negativo si verifica se i nuovi parametri siano in grado di sbloccare qualche esecutore in attesa dei parametri di input.

Il ciclo di vita degli oggetti `DataGetterExecutor` è mostrato nel seguente diagramma degli stati. Sebbene tale formalismo non sia stato ancora presentato, la sua comprensione è piuttosto immediata.

Nel diagramma degli oggetti di fig. 7.66 è mostrato come lo script presentato in precedenza possa essere memorizzato negli oggetti istanza del modello disegnato fino a questo punto.

Nel diagramma, la rappresentazione di oggetti di tipo `DataSource` non è correttissima. Chiaramente si tratta di istanze di classi che implementano tale interfaccia e che quindi, per questioni di chiarezza, possono essere considerate come oggetti di tipo `DataSource`.

Per terminare la descrizione di questo meccanismo è necessario riportare alcune riflessioni. In primo luogo si può notare che la relazione tra gli oggetti `ScriptManager` e i relativi `Script` è di tipo relazionale e non OO. In altre parole, esso avviene per mezzo dell'esportazione dell'identificatore dello script, `scriptId` — come si usa nei database relazionali per relazionare le tabelle — anziché per mezzo di apposito attributo che contenga il riferimento in memoria dell'apposito oggetto script. Questo meccanismo è utilizzato per "disaccoppiare" le classi. Ciò permette di raggruppare gli oggetti in sottoinsiemi ad alto livello di coesione e minimo accoppiamento con gli oggetti esterni. Ciò è particolarmente utile per la progettazione component based del sistema (per esempio un componente si potrebbe occupare della gestire degli script, un altro di coordinarne l'esecuzione, ecc). Lo stesso meccanismo dovrebbe essere utilizzato tra gli oggetti `ActualParameter` e `FormalParameter`.

Un altra constatazione è relativa all'utilizzo dei parametri attuali. Ciò si è reso necessario essenzialmente per due motivi: disporre della capacità di impostare un valore iniziale per i parametri di input (attributo `initialValue`) ed evitare problemi di omonimia.

Si consideri il seguente esempio di un ipotetico script per la generazione di un documento relativo ai beni immobili posseduti da una coppia di coniugi:

```
userFiscalCode              =  smartcard.getFiscalCode();
userName                    =  smartcard.getName();
userSurname                 =  smartcard.getSurname();
partnerFiscalCode           =  registryOffice.getPatnerFiscalCode(userFiscalCode);
partnerName, partnerSurname =  registryOffice.getNominative(partnerFiscalCode);
listOfProperties            =  landRegistryOffice.getProperties(userFiscalCode,
                                                                partnerFiscalCode);
```

> Chiaramente si tratta di un esempio e come tale va considerato. È evidente che, per poter ottenere documenti più articolati, sarebbe necessario incorporare opportuni costrutti nella grammatica utilizzata dagli script, come flussi alternativi e cicli ripetitivi.

Da una prima analisi dello script, probabilmente, non emergerebbe nulla di strano, ma analizzandolo più in dettaglio sarebbe possibile evidenziare alcuni problemi. Per esempio, la funzione dell'ufficio anagrafico (*registry office*) in grado di fornire il nome e il cognome dell'utente, verosimilmente, dovrebbe assumere una sintassi come quella riportata di seguito (si ricordi che si tratterebbe di una funzione utilizzata per generare diversi script):

```
userName, userSurname = registryOffice.getNominative(userFiscalCode);
```

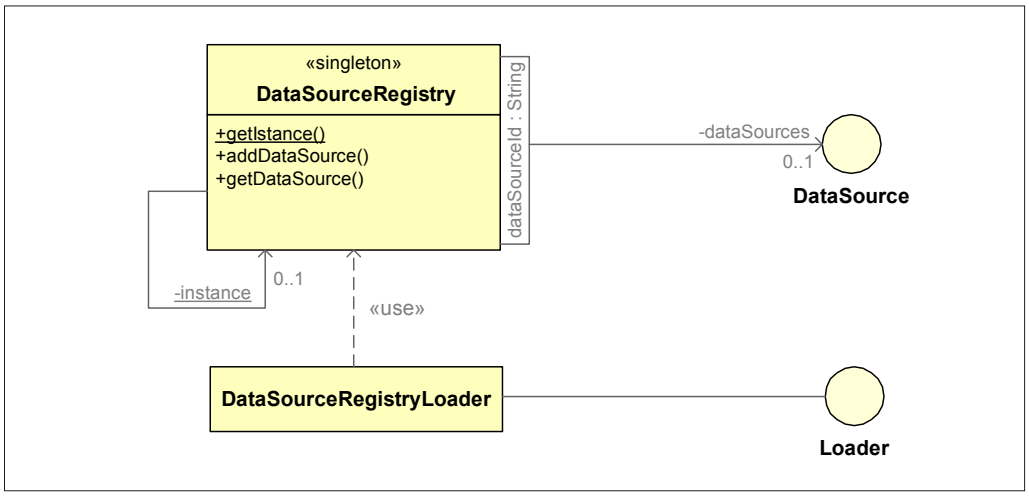
Ciò significa che l'oggetto `DataGetterExecutor` atto a eseguire la funzione si dovrebbe occupare di reperire dallo `ScriptManager` — che a sua volta delega al relativo `DataStorage` — il valore di un elemento di nome `userFiscalCode` e quindi memorizzare i risultati prodotti negli elementi di nome `userName` e `userSurname`. Il problema è che il metodo dovrebbe essere invocato per due persone, e quindi il risultato prodotto durante la seconda chiamata andrebbe a sovrascrivere diverse informazioni individuate dallo stesso identificatore. L'anomalia è risolta grazie alla presenza dei parametri attuali. Pertanto nell'esempio, si avrebbero le seguenti associazioni:

`userFiscalCode`	viene rinominato in	`partnerFiscalCode`
`userName`	viene rinominato in	`partnerName`
`userSurname`	viene rinominato in	`partnerSurname`

Questo meccanismo permette all'oggetto `DataGetterExecutor`, durante il reperimento del parametro di input `userFiscalCode`, di verificare l'esistenza dei parametri attuali e quindi di reperirne i nomi (`partnerFiscalCode`). Questi sono quindi utilizzati come chiave di ricerca nel `DataStorage`. Lo stesso meccanismo è utilizzato per memorizzare i parametri generati dall'esecuzione della funzione. Pertanto la presenza dei parametri attuali consente di definire liberamente e genericamente la sintassi delle funzioni `DataGetter` senza incidere sulla funzionalità degli script.

Nella classe `DataGetterExecutor` non è presente un legame con la sorgente di dati che dovrà eseguire la funzione, bensì è presente il relativo identificatore. In sostanza avviene un bind dinamico. Affinché ciò possa realizzarsi, è necessario fornire un meccanismo in grado di memorizzare e gestire le varie sorgenti di dati. In sostanza occorre realizzare un registry in cui inserire gli identificatori delle datasource disponibili corredati dal percorso delle relative classi. Questo compito è abbastanza semplice in linguaggi come Java in cui esistono meccanismi della riflessione e tutti gli oggetti discendono da un medesimo genitore. Pertanto, attraverso il metodo `classForName` è possibile generare dinamicamente

Figura 7.67 — *DataSourceRegistry.*



istanze di classi, il cui percorso è specificato come parametro del metodo. In altri linguaggi sarebbe stato necessario memorizzare istanze — e non il percorso della propria classe — di oggetti implementanti una comune interfaccia. In ogni modo, le classi inserite non sono, ovviamente, le sorgenti di dati, bensì classi (*proxy*) in grado di gestirne il collegamento: veri e propri strati di interfacciamento.

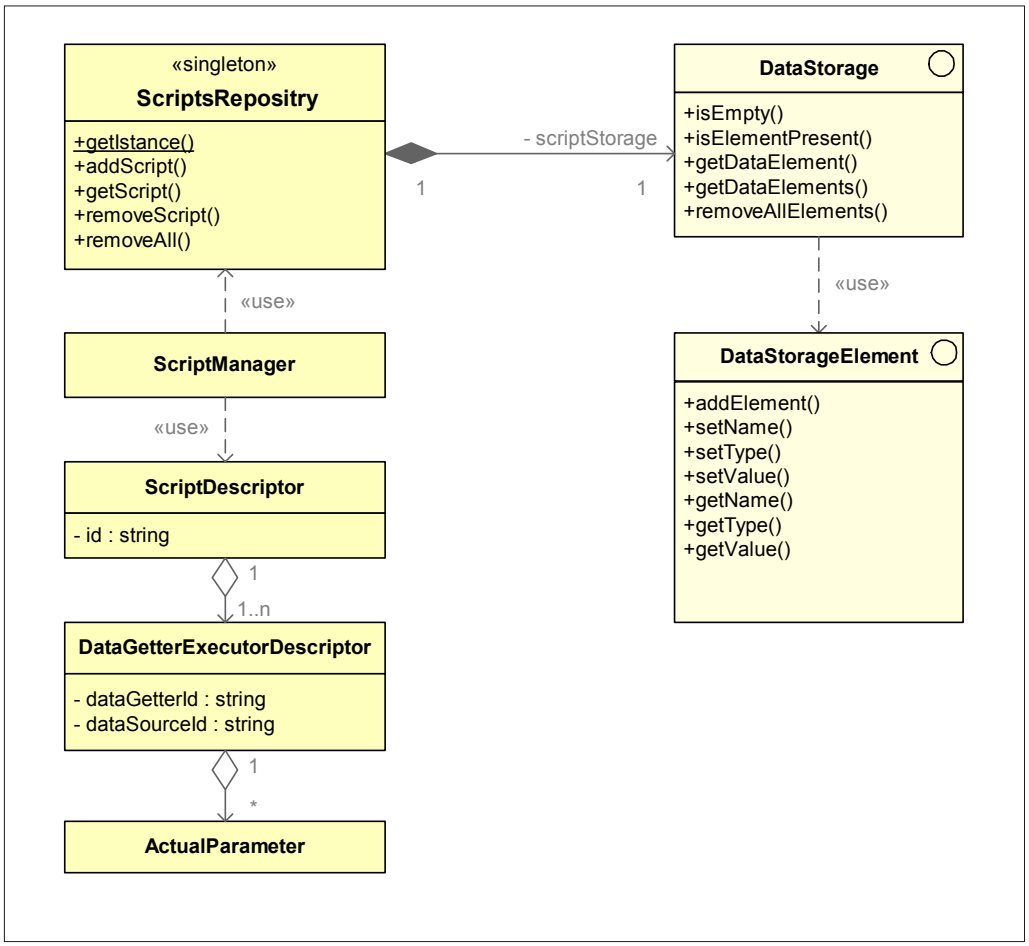
> Sebbene, forse per un eccessivo accademismo, diversi tecnici e docenti universitari siano convinti dell'opposto, l'autore ritiene che nel modello di disegno sia assolutamente legittimo utilizzare i meccanismi propri del linguaggio e della tecnologia con cui si è deciso di implementare il sistema. Chiaramente, nel modello di analisi lo sarebbe molto meno.

Il `DataSourceRegistry` implementa il famoso pattern Singleton, pertanto tutti gli oggetti che ne hanno bisogno possono ottenere dinamicamente il riferimento all'unica istanza (*cfr* fig. 7.67). L'istanza `DataSourceRegistry` permette di reperire la descrizione delle `DataSource` desiderate specificandone unicamente l'id.

La presenza dell'associazione di qualificazione implica che si tratta di una relazione tra un'istanza — l'unica — della classe `DataSourceRegistry` e un insieme di oggetti `DataSource`, in cui il reperimento delle singole istanze (`DataSource`) avviene per mezzo del relativo id.

Al fine di conferire un aspetto compiuto ai vari frammenti del sistema presentati fino a questo punto, occorre disegnare un meccanismo che si occupi di creare gli oggetti necessari all'esecuzione dei vari script (*cfr* fig. 7.68). In altre parole, a seguito della richiesta

Figura 7.68 — `ScriptsRepository`.



dell'utente di generare un particolare documento, un apposito componente deve acquisire le informazioni del relativo script e quindi realizzarne l'immagine in grado di essere eseguita. In particolare, gli oggetti necessari sono lo `ScriptManager` corredato dal relativo `DataStorage`, i vari `DataGetterExecutor` con gli eventuali `ActualParameter`. Il problema da risolvere è quello di creare una struttura in grado di memorizzare un'immagine del programma, uno `ScriptsRepository`, da cui generare le istanze runtime. Gli oggetti da inserirvi devono permettere di descrivere completamente e senza ambiguità gli stati in cui porre i vari oggetti `DataGetterExecutor` che compongono i vari script. Lo

stato di ciascuno di essi è dato dai valori assunti dai relativi attributi e dalle relazioni instaurate con gli altri oggetti. Per esempio, per ogni `DataGetterExecutor` è necessario inizializzare le istanze specificando il `DataGetter` di riferimento, il `DataSource` in grado di eseguire la funzione, la lista degli attributi attuali e le relative associazioni con quelli formali, ecc. Quindi, dato l'identificatore di uno script, è necessario reperire la lista dei vari `DataGetterExecutor` corredati dalla descrizione del relativo stato. Ciò è ottenibile associando allo `ScriptsRepository` un `DataStorage` in grado di memorizzare le associazioni tra gli identificatori degli script e opportuni grafi di oggetti. Da tener presente che l'attributo `value` è di tipo `Object` e quindi è in grado di memorizzare un qualsiasi grafo di oggetti serializzati.

Nel diagramma di fig. 7.68 è mostrato sia lo `ScriptsRepository`, sia il descrittore di script. Come si può notare, tale struttura risulta completamente idonea a essere memorizzata in un apposito file XML di configurazione.

L'unica istanza `ScriptsRepository` potrebbe venir inizializzata con l'associazione dei vari script sia all'atto dell'avvio del server, sia ogniqualvolta il tentativo di reperimento di uno script fallisca, magari tentando di localizzare un apposito file XML. Chiaramente, qualora fallisse anche questo secondo tentativo, il processo relativo alla richiesta del particolare documento terminerebbe in una condizione di errore.

Come si può notare, né l'oggetto `DataStorage`, né quello `ScriptRepository` sono al corrente del tipo di struttura che viene memorizzata (`ScriptDescriptor`).

Classi "di classe"

Quali linee guida è possibile seguire per aumentare il livello di eleganza dei diagrammi delle classi? In questa sede si concentrerà l'attenzione sullo stile limitando, per quanto possibile, digressioni relative alla semantica, sebbene poi le due caratteristiche non possano essere trattate in maniera completamente disgiunta.

Il formalismo dei diagrammi delle classi si presta a essere impiegato per la realizzazione di molteplici tipologie di modelli statici: da quelli di business e dominio realizzati per aumentare la propria comprensione dell'aera oggetto di studio e dei requisiti utenti, a quelli di analisi e disegno, volti a trasformare i requisiti in soluzioni tecniche sempre di più basso livello, dai modelli della base di dati a quelli dei messaggi, e così via. Logica conseguenza è che lo stile deve essere adeguato ai fini che, di volta in volta, si intende perseguire con la realizzazione dello stesso modello, nonché alla platea dei fruitori. L'analisi dettagliata dei vari modelli realizzabili attraverso il formalismo dei diagrammi delle classi è oggetto di studio del Capitolo 8 dove sono presenti innumerevoli suggerimenti specifici per ogni tipologia di modello presentata.

Contrariamente a quanto si potrebbe pensare, curare lo stile dei diagrammi per renderli armoniosi ed eleganti non è un esercizio fine a sé stesso o un vezzo estetico. Diagrammi ben realizzati anche dal punto di vista grafico risultano più facilmente fruibili, elevano il livello di comprensione e memorizzazione, permettono di aumentare il contenuto informativo — per esempio utilizzando in maniera intelligente i colori —, concorrono a diminuire il grado di rifiuto psicologico che si può instaurare nei fruitori privi di dimestichezza con il formalismo o estranei al modello stesso, e così via.

Organizzazione del modello

La realizzazione di modelli eleganti richiede una serie di prerequisiti indispensabili. Uno dei primi consiste nell'aver selezionato nomi significativi per gli elementi utilizzati (package, diagrammi, classi, relazioni, metodi e attributi). Ciò permette di "leggere" i diagrammi minimizzando lo sforzo necessario per la comprensione: un diagramma ben realizzato già da solo è in grado di illustrare l'organizzazione dei relativi elementi, semplicemente collegando i nomi delle classi con le associazioni e i ruoli recitati. Gli stessi diagrammi devono avere nomi chiari che consentano di stabilire immediatamente quale sia l'aspetto o l'organizzazione che intendono illustrare.

Spesso, per "risparmiare" qualche carattere alcuni tecnici danno luogo a nomi che solo un potente algoritmo di offuscazione potrebbe rendere meno leggibili. L'esperienza insegna che, senza esagerare, conviene sempre dettagliare quanto più possibile i nomi delle varie entità: tutto diviene più lineare, semplice, si riduce la necessità di ricorrere a ulteriori spiegazioni, e così via. Inoltre, convenzioni di fatto suggeriscono di riportare i nomi delle classi al singolare, anche se rappresentano molteplici oggetti, e di utilizzare per i metodi e le relazioni nomi che enfatizzino il verbo. Un altro importantissimo requisito consiste nel dar luogo a una corretta organizzazione in package (questo argomento è stato trattato nel capitolo precedente e verrà ripreso in quello successivo).

Da tener presente che esistono due organizzazioni parallele in termini di package:

- sistema da produrre (quella in cui organizzare i vari moduli di codice e quindi il sistema "eseguibile");
- modello (o più in generale dell'intero progetto).

L'oggetto di studio del presente paragrafo è la seconda, la quale, evidentemente, include l'organizzazione del codice, nel senso che possiede sicuramente tutte le classi che dovranno essere implementate, con in più i diagrammi, che ovviamente forniscono molte informazioni su come implementare il sistema ma non hanno, per così dire, un corrispettivo diretto in termini implementativi.

Per quanto riguarda la struttura del modello, in ogni package è opportuno disporre di almeno un diagramma illustrante le relazioni tra le classi incluse (che dovrebbero possedere un livello di coesione non trascurabile, criterio principale di raggruppamento). Qualora poi gli elementi o gli aspetti da presentare risultino molteplici, oppure qualora si voglia conferire particolare rilievo all'organizzazione di alcuni di essi, è possibile dar luogo a ulteriori specifici diagrammi delle classi realizzati all'uopo. Non è sempre necessario riportare in ogni diagramma tutte le classi con tutte le relazioni: spesso è sufficiente riportare solo gli elementi di interesse per il contesto preso in esame. Qualora un package o un diagramma sia eccessivamente popolato, andrebbe anche considerata l'eventualità di procedere a un'ulteriore strutturazione.

Altra buona norma consiste nel realizzare un diagramma che mostri, per ciascun package del modello, la relativa organizzazione. In particolare, potrebbe risultare vantaggioso dar luogo a un diagramma con il nome del package in cui riportare: *1*. una breve descrizione del package corredata dal relativo criterio di aggregazione e delle finalità; *2*. la struttura del package stesso, sia in termini di eventuali sottopackage, sia dei diagrammi delle classi contenute, il tutto corredato da una breve spiegazione.

Tutto ciò è particolarmente utile alla navigabilità dei modelli: la quasi totalità dei tool permette di fruire i modelli eseguendo il famoso doppio click sugli elementi mostrati nei vari diagrammi. Pertanto, disponendo, per ogni package, di un diagramma che ne mostri la struttura, permette di visitare le varie parti (diagrammi, sottopackage, classi, ecc.) effettuando un doppio click sugli elementi che le rappresentano.

Criteri generali

Anche nell'ambito dei diagrammi delle classi mantengono la loro validità i criteri generali validi per qualsiasi tipologia di diagramma, per esempio, è cosa buona limitare il numero degli elementi presenti in ciascun diagramma a sette (più o meno due): sembrerebbe che questo sia il valore più confacente alla mente umana [BIB28]. È opportuno evitare il più possibile la sovrapposizione di linee, ricercare — anche in modo artificiale — simmetrie, mantenere un elevato aspetto di pulizia e chiarezza, cercare di rappresentare relazioni di ereditarietà con sviluppo verticale, cercare di rappresentare le altre relazioni attraverso linee rette, ecc.

La coerenza o *consistenza* recita sempre un ruolo di primo piano, il cui livello è direttamente proporzionale al grado di comprensione e apprendimento. I fruitori dei diagrammi tendono a prendere familiarità con le convenzioni utilizzate e quindi diventano pratici nel riuscire a intuire rapidamente i concetti che stanno alla base di tali convenzioni.

Ordine e posizione degli elementi non dovrebbero, in teoria, avere particolare significato ma è evidente che i lettori tendono, più o meno consciamente, a conferire maggiore importanza agli elementi più evidenti e a rispettare un ordine di lettura che va da sinistra verso destra — faranno eccezione lettori arabi o asiatici — e dall'alto verso il basso. Pertanto è utile cercare di sistemare in una posizione centrale l'elemento a cui si intende conferire maggiore attenzione — eventualmente cercando di evidenziarlo artificialmente utilizzando un tratto leggermente più spesso, una tinta più marcata, ecc. — ed è bene riportare gli elementi interconnessi considerando il tipico ordine di lettura.

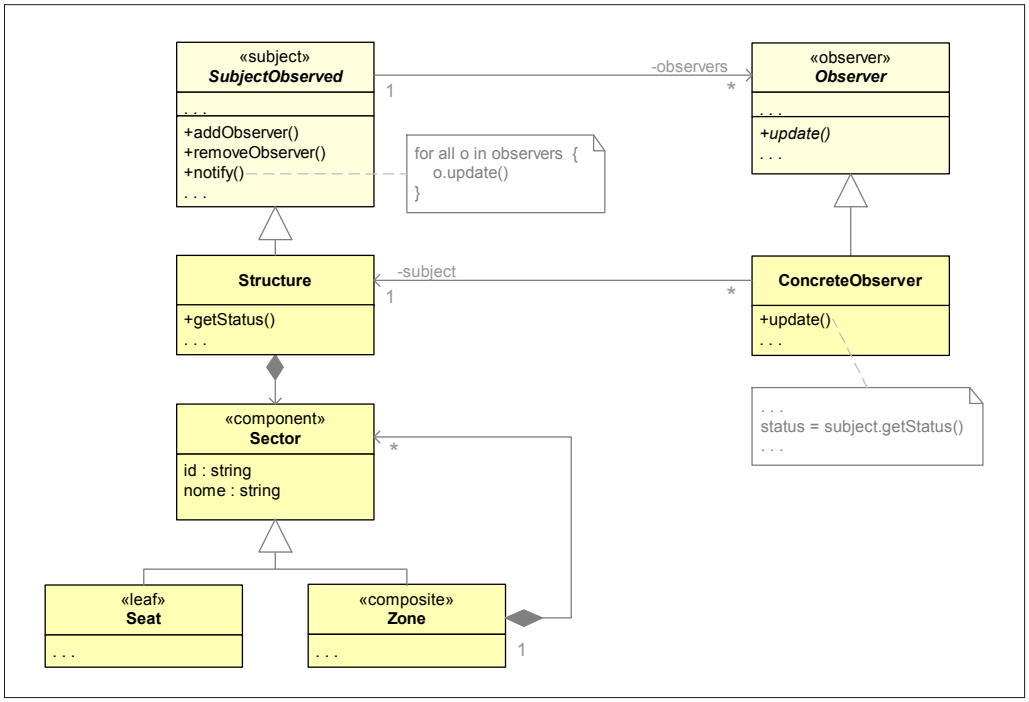
Così come nella codifica, anche nella modellazione è importante riportare commenti al fine di chiarire particolari linee guida utilizzate per la costruzione del modello. Si potrà quindi spiegare il perché di specifiche decisioni, illustrare porzioni particolarmente complesse, attributi molto importanti, ecc. Le note però non devono "appesantire" il diagramma. Quindi è consigliabile ricorrere all'elemento standard UML — la porzione di foglio di carta con un angolo piegato — impostandone un colore di sfondo trasparente con linee e testo in grigio.

Qualora si utilizzi un pattern, è molto utile sia evidenziarlo, magari variando sensibilmente i colori delle relative classi, sia specificare chiaramente il ruolo recitato dalle varie classi che lo compongono. Ciò è ottenibile attraverso l'utilizzo degli stereotipi eventualmente arricchiti da opportune note. Per esempio (*cfr* fig. 7.69) qualora si utilizzi il pattern Observer, è opportuno evidenziare le classi osservate (`Subject`) e quelle osservatrici (`Observer`). Nel pattern Composite le classi da evidenziare sono quella componente (`Component`), base (`Leaf`) e iterativa (`Composite`).

Un altro elemento di stile consiste nel rappresentare interfacce e classi astratte con una tinta più chiara al fine di evidenziarne la funzione principale ovvero definire un tipo e non poter essere istanziate direttamente. Lo stesso vale per elementi secondari, come tipi enumerati, ad esempio, i quali non dovrebbero assolutamente catturare l'attenzione del lettore. Si consiglia pertanto di ricorrere agli stessi colori dei campi note.

Per quanto concerne la rappresentazione grafica delle interfacce, è consigliabile ricorrere il più possibile alla notazione basata sul simbolo della circonferenza. Se da un lato questa notazione nasconde importanti dettagli (i metodi di cui è composta la classe), dall'altro permette di evidenziarle a colpo d'occhio e di realizzare diagrammi più eleganti e lineari.

Figura 7.69 — *Diagramma di "stile" che mostra l'utilizzo dei pattern Composite e Observer. In particolare si immagina una situazione — quale per esempio quella di un cinema — in cui una particolare struttura, come una sala (classe* `Structure`*) possa essere impostata secondo una qualsivoglia organizzazione gerarchica (pattern* `Composite`*) e di cui sia importante sapere, in tempo reale, la situazione dell'allocazione dei posti, per aggiornare sia un display, sia le varie postazioni client per la vendita di biglietti. A tal fine è stato utilizzato il pattern Observer e le classi derogate all'aggiornamento dei vari dispositivi devono estendere la classe* `ConcreteObserver`*.*



Livello di dettaglio

Un altro elemento molto importante — ricorrente in ogni attività di modellazione e quindi particolarmente presente nella costruzione di modelli UML — è il livello di dettaglio a cui spingersi. Nel contesto dei diagrammi delle classi assume una forma diversa a seconda del particolare modello preso in considerazione. Queste problematiche sono riprese e illustrate abbondantemente nel corso del capitolo successivo.

Per esempio, in un modello a oggetti del dominio è necessario mostrare tutte e sole le classi appartenenti all'area oggetto di studio che il sistema dovrà automatizzare (quindi l'analisi non va estesa all'intera area di business); per ciascuna di esse ha senso evidenziare

i relativi attributi, mentre le operazioni assumono un'importanza secondaria, è utile attribuire un nome esplicativo a ogni associazione e via discorrendo.

Per quanto riguarda il modello di disegno esistono due macroversioni: quello di specifica e quello di implementazione. Il secondo dovrebbe essere allineato con il codice, e quindi spesso è ottenuto attraverso procedure di reverse engineering — o sincronizzazione in linea dell'attività di codifica — mentre il primo è il modello da consegnare ai programmatori affinché possano procedere con l'implementazione del sistema. In questo caso il livello di dettaglio cui giungere è cruciale e dipende fortemente dalle capacità del team degli sviluppatori. Se non si dispone di un team preparato e con buona visione del disegno globale, è assolutamente necessario investire molto tempo nel realizzare modelli di dettaglio, relegando, ahimè, i vari sviluppatori al mero ruolo di "riempitori di classi". Se invece il team di programmatori è ben preparato, è possibile dar luogo a un più razionale utilizzo della risorsa tempo, concentrandosi sul disegno di insieme, limitandosi a disegnare le classi e i metodi più importanti da corredare eventualmente con opportune linee guida. Si demanda quindi alle capacità dei singoli programmatori la definizione del cosiddetto codice di "impalcatura" (*scaffolding*, attributi e metodi privati, classi *helper*, metodi di *get* e *set*, ecc.). Se poi dovesse verificarsi lo scenario peggiore (architetto con capacità di disegno argomentabili…), allora, per la salvaguardia della salute del team di sviluppo, è opportuno che il modello resti più possibile sospeso nel mondo dell'astratto… tenendo bene a mente che nell'informatica "capolavoro di arte moderna" non sempre assume una connotazione positiva.

Diagrammi delle classi a colori

L'utilizzo dei colori nei diagrammi delle classi è un argomento che negli ultimi anni ha assunto un certo rilievo — non fosse altro per il dibattito che vi ruota intorno — tanto da meritare di essere trattato in un apposito paragrafo.

Tutti i modellatori, prima o poi, si rendono conto che utilizzare opportuni colori nei diagrammi tende ad aumentarne notevolmente la comprensione: nelle parole di Peter Coad, "i colori forniscono lo strumento per codificare strati aggiuntivi di informazione. Un saggio utilizzo dei colori aumenta il contenuto informativo esprimibile".

Pertanto, invece di utilizzare lo storico giallino per tutti gli elementi, ricorrere a tinte diverse per specifici elementi può aumentare il contenuto informativo dei diagrammi stessi, senza per altro dover introdurre elementi aggiuntivi e quindi senza appesantire l'organizzazione. Per esempio già nell'ambito dei diagrammi dei casi d'uso si è proposto di utilizzare il grigio per evidenziare relazioni di estensione e il nero per quelle di inclusione. Ciò permette, a colpo d'occhio, di distinguere nettamente relazioni a carattere opzionale da quelle obbligatorie.

Non solo i colori rendono i diagrammi più accattivanti e comprensibili, ma tendono anche a renderli fruibili più rapidamente. I diversi strati informativi dovuti all'utilizzo dei

colori "sono visibili a distanza, cosicché il quadro d'insieme del modello (*the big picture*) emerge chiaramente prima di iniziare a leggerne i contenuti" (Peter Coad). Chiaramente tutto ciò è possibile a patto che i colori siano utilizzati in maniera intelligente, il che implica che sia utilizzata una ristretta palette di colori ben definita con un chiaro significato studiato accuratamente, che la convenzione sia conosciuta dai fruitori, che i vari colori siano applicati coerentemente, ecc.

Sebbene tutti i modellatori finiscano per utilizzare un proprio set di colori con significati più o meno specifici, il primo tentativo di stabilire un approccio organico si deve a Peter Coad [BIB23]. In particolare la convenzione iniziale prevede quattro colori base: giallo, rosa, verde e blu. Da notare che si tratta di una convenzione di recente ideazione, frutto di una continua revisione che spesso ha generato un processo di "attenuazione delle tinte". Ciò non dovrebbe stupire più di tanto: il tempo da sempre attenua le tinte dei ricordi…

Tabella 7.6 — *Archetipi, e relativi colori, proposti da Peter Coad.*

Archetipo	Colore	Descrizione
Moment-Interval Momento-Intervallo	Rosa	È utilizzato per modellare elementi relativi a momenti nel tempo e intervalli temporali. Rappresenta un'entità di cui bisogna tenere traccia per questioni di business o legali che avviene, appunto, in un momento specifico o un intervallo di tempo (eventi, transazioni, ecc). Per esempio il noleggio di una macchina avviene in un determinato istante e prosegue fino alla riconsegna.
Description Descrizione	Blu	È un archetipo atto a modellare classi utilizzate essenzialmente per descrizioni, decodifica di valori, ecc. Si tratta di una collezione di valori utilizzata diverse volte.
Party, Place or Thing Parte, Posto o Cosa	Verde	Evidenzia persone o cose che recitano più ruoli nel modello globale.
Role Ruolo	Giallo	È utilizzato per i ruoli delle classi, ossia il comportamento specifico assunto da un'entità (persona, luogo o cosa) che prende parte in un determinato contesto. Inizialmente si era ipotizzato un solo colore direttamente per le classi-attore; successivamente Coad ha proposto di utilizzarlo per i relativi ruoli. Per esempio, in un sistema di gestione di un campionato di calcio, una generica entità persona dovrebbe essere rappresentata con il colore verde, mentre ruoli di Calciatore, Allenatore, Arbitro, ecc. con il colore giallo.

Nella sua metodologia, Coad individua quattro elementi base denominati *archetypes* (archetipi) evidenziati dai colori di cui sopra. L'idea è molto simile a quella degli stereotipi standard dello UML, ma la definizione è meno formale e più flessibile tanto da richiedere l'invenzione di un elemento non standard. In particolare, "un archetipo è una forma dalla quale seguono classi dello stesso tipo, *più o meno*, considerando attributi, legami, metodi, punti di plug-in e interazioni" (Peter Coad). Il "più o meno" dovrebbe essere l'elemento cruciale che fa la differenza tra stereotipi, che appunto sono più formali e non ammettono nulla di simile, e archetipi. In ogni modo, la convenzione prevede quanto illustrato nella tab. 7.6. Da notare che, sempre secondo la convenzione proposta da Coad (novello impressionista), la palette di colori da utilizzare dovrebbe avere delle gradazioni molto tenui.

Da quanto emerso, la convenzione proposta si presta a essere utilizzata nei modelli di struttura statica a carattere non fortemente tecnico, come il modello a oggetti del dominio e quello di business (illustrati in dettaglio nel capitolo seguente). In questi casi può avere molto senso evidenziare con colore diverso elementi (comunque appartenenti alla realtà oggetto di studio) con semantica diversa. L'uso degli archetipi comincerebbe a perdere di utilità già dal modello di analisi (propriamente detto) e ancora di più in quello di disegno, in cui questi oggetti tendono a lasciare la scena ad altri a maggior contenuto tecnico. In questi modelli, volendo continuare ad applicare la tecnica di Coad, probabilmente bisognerebbe riadattare il significato dei vari colori e, verosimilmente, sarebbe necessario ricorrere all'utilizzo di altri (classi infrastruttura, librerie, ecc.). Il rischio che si correrebbe è quello di produrre modelli non chiari, che sfruttano un insieme di colori non sempre ben combinati (veri esempi di "inquinamento ottico"), diagrammi di un livello di professionalità discutibile, ecc. "Due o tre colori tipicamente non sono sufficienti; cinque sono troppi. La combinazione di quattro colori deve essere selezionata con cura: nulla appare peggiore di troppi colori, specie quando mancano di elementi comuni" [Hideaki Chijiwa, *Color Harmony*]. Come se non bastasse, va tenuto a mente che non sempre è possibile disporre di stampanti a colori, e che comunque il loro utilizzo, tipicamente, prevede costi superiori a quelle in bianco e nero. Da notare che la convenzione di Coad non si limita a evidenziare elementi con specifici colori, ma sulla base di questi viene realizzato tutto un procedimento di sviluppo dei vari modelli.

A questo punto è giunto il momento di tirare un po' le somme circa l'utilità del metodo dei colori di Coad. Per quanto riguarda i vantaggi, potrebbe effettivamente concorrere ad aumentare la chiarezza e il contenuto informativo di modelli del dominio e di business, mentre, verosimilmente, già dal modello di analisi i benefici potrebbero risultare ampiamente ridotti.

Gli svantaggi degli archetipi sono relativi al fatto che:

- non sempre si dispone di una stampante a colori e la stampa monocromatica di immagini a colori spesso sortisce effetti poco gradevoli;

- realizzando propriamente i diagrammi (vedi "regola del 7" più o meno 2 elementi), probabilmente le etichette degli stereotipi potrebbero già essere sufficienti;

- gli archetipi stessi non sono elementi standard UML e quindi non sono supportati da molti tool commerciali;

- non sempre il loro utilizzo apporta vantaggi proporzionali al tempo speso per evidenziarli, ecc.

> Le convenzioni utilizzate dall'autore sono decisamente più semplici e con una semantica piuttosto limitata il cui obiettivo consiste esclusivamente nel conferire maggiore o minore enfasi ai vari elementi. In particolare:
>
> - le relazioni e i vari adornamenti (nome della relazione, ruoli delle classi coinvolte, molteplicità, ecc.) sono evidenziate con una tinta grigio scura, comunque ben riproducibile con stampanti monocromatiche;
> - le classi del diagramma non appartenenti al package o al contesto preso in esame, sono rappresentate con uno sfondo grigio chiaro, eventualmente visualizzando solo il compartimento relativo al nome senza ulteriori dettagli;
> - le classi appartenenti al package e/o al contesto preso in esame, sono visualizzate con una tinta giallina, resa ancora più leggera per le interfacce e le classi astratte;
> - gli elementi secondari, per esempio i tipi enumerati, sono rappresentati con sfondo trasparente corredati da testo e linee grigie, utilizzando la stessa convenzione stabilita per le note;
> - infine, qualora si voglia conferire maggiore enfasi a pattern applicati, è possibile rappresentarli con una tinta leggermente diversa, ferma restando l'indicazione dei ruoli che compongono il pattern.
>
> Cosa sarebbe la vita senza colore?

Adornamenti

Si prenderanno ora in considerazione delle linee guida da utilizzare per rappresentare concetti come visibilità, firma dei metodi, navigabilità, molteplicità, ecc.

Un primo elemento da considerare è la **visibilità** di attributi e metodi delle classi. Si tratta di un evidentissimo caso di dipendenza dalla tipologia del modello oggetto di studio. Qualora si stia realizzando un modello di disegno è evidente che la visibilità degli elementi assume un ruolo cruciale (specifica le regole secondo le quali il relativo elemento è accessibile da parte di altri oggetti), mentre in tutti gli altri modelli si tratta di dati completamente inutili. Per esempio, l'autore consiglia di costruire modelli ad oggetti del dominio utilizzando un approccio basato sull'analisi dei dati (*data-driven*), il cui obiettivo è rappresentare un modello logico dei dati coinvolti nell'area oggetto di studio, disinteres-

sandosi, momentaneamente, dei servizi. In questo contesto ha interesse mostrare i dati, la relativa struttura e la rete delle relazioni tra di essi: è quindi evidente che il concetto di visibilità non ha alcun rilievo.

Qualora invece ci si trovi nel contesto di modelli di disegno, non solo è fondamentale mettere in evidenza la visibilità degli elementi, ma è anche molto importante organizzare i due elenchi (attributi e metodi) in modo tale che elementi con stessa visibilità siano raggruppati e che gli elementi con visibilità più estese siano mostrati prima (quindi tutti gli elementi `public`, poi quelli `package`, di seguito i `protected` e in fine i `private`). Ciò perché gli elementi con visibilità più estesa tendono a fornire maggiori informazioni sul disegno d'insieme del modello e quindi tendono a ricevere maggiore attenzione da parte dei fruitori.

La visualizzazione di elementi privati tipicamente ha un'importanza relativa. Probabilmente ha più senso in diagrammi di carattere documentativo (magari in appositi diagrammi di dettaglio delle singole classi). Generalmente la presentazione di tutti gli elementi tende a rendere i modelli decisamente pesanti e confusi, generando inevitabili problemi alla comprensione del disegno globale (*the big picture*). Quando si realizzano modelli di disegno di specifica, non sempre ha molto senso cercare di disegnare tutti i dettagli, tipicamente si preferisce derogare al team di sviluppo la responsabilità di progettare metodi e attributi privati.

Nei casi — frequentissimi — in cui si decida di non visualizzare l'elenco completo dei metodi e degli attributi, è molto importante evidenziare alla fine delle liste il simbolo di elisione costituito dai punti di sospensione (. . .). La mancanza di questo simbolo tende a generare nei lettori errate conclusioni o comunque confusione: non si è in grado di sapere per certo se siano state mostrate tutte le caratteristiche della classe o solo quelle ritenute più importanti per il particolare contesto.

Un problema particolarmente ricorrente nel disegnare i modelli delle classi è relativo alla **firma dei metodi**. Tipicamente, nella sua versione completa, la firma tende a occupare molto spazio. Ciò si ripercuote sulle relative classi costrette a ingigantirsi orizzontalmente, e quindi sull'intero modello che tende a divenire confuso. Qualora ci si trovi in situazioni simili, un buon consiglio potrebbe essere quello di agire sui parametri. In particolare è possibile rendere la descrizione delle classi più lineare, limitandosi a riportare il nome dei parametri e trascurandone il tipo.

Per esempio, invece di riportare la firma:

```
+ insertElement(key:Object, value:Object):Object
```

si potrebbe visualizzare la forma più concisa:

```
+ insertElement(key, value):Object
```

Un accorgimento virtualmente molto interessante consiste nell'indicare, a fianco alla firma dei metodi, l'elenco delle eventuali eccezioni scatenate. Per esempio:

```
+ insertElement(key:Object, value:Object):Object {throws NullPointerException,
                                                    DuplicateKeyException}
```

Sebbene questa tecnica permetta di specificare informazioni molto utili, aggrava il problema dell'espansione orizzontale delle classi menzionato poc'anzi. È anche assolutamente sconsigliato utilizzare un criterio impreciso: nei metodi in cui la firma lo consente, si riporta l'elenco delle eccezioni, mentre dove non c'è posto lo si omette. La consistenza è una qualità importantissima della modellazione soprattutto nei diagrammi a maggiore connotazione tecnica. Un "approccio misto" tenderebbe a confondere il lettore: non trovando dettagliato l'elenco delle eccezioni scatenabili da un metodo, tenderebbe, legittimamente, a dedurre che il metodo non preveda la possibilità di generare eccezioni, mentre potrebbe trattarsi di un caso di omissione per questioni di spazio.

Per quanto concerne gli **stereotipi**, da un lato è vero che, se utilizzati intelligentemente, concorrono a rendere i diagrammi più chiari, a facilitare una rapida comprensione del disegno di insieme, e così via; ma, dall'altro, è altrettanto vero che, se utilizzati in modo "ridondante", tendono unicamente ad aumentare il livello di rumore. Per esempio, alcuni tecnici usano premettere stereotipi del tipo `get`, `set` a metodi quali `getName()`, `setName()`. Si tratta di dati assolutamente ridondanti da evitare poiché non aggiungono alcuna informazione e invece rendono i diagrammi meno chiari.

Molto importante è poi riportare le **molteplicità** con cui le classi partecipano alle relazioni. Non è mai opportuno affidarsi al "default". Per esempio, spesso la molteplicità di una classe aggregata o composta è omessa in quanto presunta uguale a uno. Sebbene si tratti del caso più ricorrente, non è valido in assoluto. Se si considera per esempio una classe aggregata `Team` composta dagli elementi `Employee`, si può notare che un elemento di un team può appartenere a diversi altri ed ecco quindi che la molteplicità posta uguale a uno, dal lato della classe aggregata non è accettabile. Pertanto è sempre opportuno riportare la molteplicità. Molto importante è anche cercare di essere precisi. Per esempio, invece di riportare molteplicità generiche specificate dal carattere asterisco (`*`) è consigliabile riportare forme meno sintetiche del tipo `1..n` oppure `0..n`.

L'interesse relativo alla **navigabilità** delle associazioni dipende molto dal modello oggetto di studio: è marginale nei modelli relativi alle prime fasi del processo di sviluppo del software nei quali si può fissare *bidirezionale* convenzionalmente e senza troppi problemi, mentre assume un'importanza notevole in quello di disegno, nel quale, per questioni essenzialmente relative all'accoppiamento tra le classi, si vuole *unidirezionale*. Ciò equivale a dire che, in una determinata associazione, oggetti di un tipo posso navigare in quelli di un altro ma non viceversa. In termini tecnici, ciò comporta che i primi devono possedere un riferimento (o più riferimenti nel caso che la relazione sia a molti) agli oggetti associati e quindi navigabili. Si tratta evidentemente di decisioni a forte connotazione tecnica che quindi è opportuno prendere nelle relative fasi quando si dispongono sufficienti informazioni sull'ambiente ed il sistema.

Qualora si decida di evidenziare la navigabilità dell'associazione è consigliato riportare il nome della stessa in modo che l'interpretazione rispetti la direzione della navigabilità. Per esempio, dovendo rappresentare la relazione di associazione tra `Persona` e `Indirizzo`, nella stragrande maggioranza dei casi si tratta di un'associazione unidirezionale: istanze di tipo `Persona` possono navigare in oggetti di tipo `Indirizzo` associati, ma non viceversa. Quindi è lecito attendersi nomi di associazioni del tipo `domicilia`, *ha* `residenza`, ecc.

Sempre nei modelli di disegno è molto importante riportare il **ruolo** che la classe recita partecipando nelle relazioni: si tratta del nome da assegnare agli attributi che, secondo anche la molteplicità e navigabilità, eventualmente implementano le associazioni stesse. Nei modelli di business e dei requisiti il ruolo tende a divenire un dato ridondante per via del nome dell'associazione, eccezion fatta qualora sia necessario specificare dei vincoli, nel qual caso divengono molto importanti perché permettono di definire gli stessi.

Ricapitolando…

Il presente capitolo è stato dedicato all'illustrazione del formalismo dei diagrammi delle classi, probabilmente uno dei più noti e affascinanti tra quelli definiti dallo UML.

Nei processi di sviluppo del software, questo formalismo si presta a rappresentare formalmente molteplici *artifact* (manufatti) presenti in diversi stadi del processo. Nelle primissime fasi (*Inception* ed *Elaboration*) lo si adotta per rappresentare i modelli del dominio e di business, nello stadio di analisi è utilizzato per dar luogo all'omonimo modello (rappresentazione formale dei requisiti funzionali sanciti nel modello dei casi d'uso); nella fase di disegno è impiegato per progettare e documentare l'organizzazione in codice del sistema e così via.

La modellazione è un ottimo strumento di investigazione e di comunicazione sia tra i membri del team, sia tra diversi team (requisiti, architettura, sviluppo, ecc.). Chiaramente è un'attività difficile, ad alto grado di creatività, legata all'esperienza dei singoli che concede poco spazio all'improvvisazione: tutto vive nel regno delle regole formali dell'OO. Le difficoltà che si incontrano, quindi, non sono tanto dovute all'UML, quanto alla complessità intrinseca del processo di disegno.

Tecnicamente un diagramma delle classi è definito come un grafo di classificatori connessi attraverso opportune relazioni. Pertanto, in questa tipologia di diagrammi vengono collocati elementi come classi, interfacce, package, tipi di dati, e così via (elementi che nel metamodello UML sono specializzazioni della classe astratta `Classifier`). Anche per questo motivo probabilmente un nome più appropriato per i diagrammi "delle classi" dovrebbe essere "diagramma della struttura statica".

Una classe è una descrizione di un insieme di oggetti che condividono la stessa struttura (attributi), il medesimo comportamento (operazioni) e le stesse relazioni (particolari attributi). La rappresentazione grafica efferente prevede un rettangolo, tipicamente suddiviso in tre sezioni dedicate rispettivamente al nome, agli attributi e alle operazioni. Di queste, solo la prima è obbligatoria. In funzione delle esigenze del modellatore, è possibile eventualmente inserirne di ulteriori, dedicate, per esempio, all'elenco delle eccezioni scatenabili, alle dichiarazione delle responsabilità della classe e così via.

Il nome di una classe può essere riportato nella forma semplice, ossia solo il nome, oppure corredato dalla lista dei package: il percorso (*path name*). Da notare che gli elementi della lista vengono separati da una coppia del carattere due punti (`::`) anziché dal singolo punto come avviene in Java. Qualora una classe rappresenti un'istanza di un determinato stereotipo, è possibile specificare il nome di quest'ultimo, opportunamente racchiuso dalle famose parentesi angolari (`<` `>`), riportato al di sopra di quello della classe. In alternativa è possibile ricorrere all'utilizzo della relativa notazione grafica prevista dallo stereotipo (icona). Subito sotto il nome della classe è possibile specificare una lista di stringhe riportanti o appositi valori etichettati (*tagged value*) o attributi del metamodello.

Un attributo è una proprietà della classe, identificata da un nome, che descrive un intervallo di valori che le relative istanze possono assumere. Il concetto è del tutto analogo a quello di variabile di un qualsiasi linguaggio di programmazione. Una classe può non avere attributi o averne un numero qualsiasi. Di un attributo è possibile indicare il solo nome, oppure il nome e il tipo, oppure la precedente coppia corredata da un valore iniziale. Eventualmente è possibile specificarne lo stereotipo.

Un'operazione può essere considerata come la trasposizione, in termini informatici, di un servizio che può essere richiesto a ogni istanza di una classe al fine di modificare lo stato del sistema e/o di fruire di un servizio. Pertanto, un'operazione (metodo) è un'astrazione di qualcosa che può essere eseguito su tutti gli oggetti di una stessa classe. Come nel caso degli attributi, una classe può non disporre di alcun metodo (anche se ciò sarebbe molto discutibile), oppure averne un numero qualsiasi.

In UML è possibile mostrare l'annidamento di una classe all'interno di un'altra attraverso un apposito formalismo che consiste nel connettere le classi attraverso un segmento con riprodotto, nell'estremità della classe "esterna", un apposito simbolo detto *anchor* (ancora): una croce circoscritta da una circonferenza.

In UML un'interfaccia è definita come un insieme, identificato da un nome, di operazioni che caratterizzano il comportamento di un elemento. Si tratta di un meccanismo che rende possibile dichiarare esplicitamente operazioni visibili dall'esterno di classi, componenti, sottosistemi, ecc. senza specificarne la struttura interna. L'attenzione viene quindi focalizzata sulla struttura del servizio esposto e non sull'effettiva realizzazione. Nel metamodello UML le interfacce sono specializzazioni della metaclasse `Classifier`, quindi ne possiedono tutte le caratteristiche. Si prestano a essere rappresentate graficamente attraverso il classico formalismo del rettangolo diviso in compartimenti, anche se, tipicamente, viene preferita la rappresentazione che fa uso dell'apposita icona (circonferenza).

Un *template* è un descrittore di una classe parametrica in funzione di uno o più parametri. Pertanto un template non definisce una singola classe bensì una famiglia, in cui ciascuna istanza è caratterizzata dal valore assunto dai parametri. Questi valori diversificano le varie classi condizionando il comportamento di specifici metodi. La notazione grafica di un template prevede il tipico formalismo utilizzato per le classi, con l'aggiunta di un rettangolo tratteggiato, nell'angolo in alto a destra, destinato a ospitare la dichiarazione della lista dei parametri.

Il tipo enumerato rappresenta un tipo di dato definito dall'utente costituito da una lista di nomi, a loro volta definiti dall'utente. Esso è utilizzato per vincolare i valori impostabili nelle relative istanze a uno degli elementi appartenenti alla lista che costituisce il tipo. Questa lista è caratterizzata dal possedere un rigido ordine (quello della dichiarazione degli elementi della lista) ma nessuna algebra.

La modellazione di un sistema non solo richiede l'identificazione delle classi di cui è composto, ma anche dei legami che le interconnettono. Tali legami, nel mondo OO, sono definiti relazioni e sono tutte riconducibili ai tipi fondamentali: dipendenza, generalizzazione e associazione (la relazione di realizzazione è una variante della generalizzazione).

La dipendenza è una relazione semantica tra due elementi (o insiemi di elementi) utilizzata per evidenziare condizioni in cui una variazione dell'elemento indipendente (*supplier*) comporta modifiche dell'elemento dipendente (*client*). Tipicamente si ricorre a una relazione di dipendenza qualora non ne esistano altre più appropriate per associare gli elementi; in altre parole, la relazione di dipendenza è la più debole di quelle previste. Poiché la definizione del metamodello UML prevede che la relazione di dipendenza possa prevedere nel ruolo di *supplier* e *client* qualsiasi `ModelElement` (metaclasse astratta del metamodello UML, da cui derivano tutti gli altri elementi), ne segue che tutti gli elementi dello UML possono essere relazionati tra loro per mezzo di una relazione di dipendenza. Graficamente questa relazione è rappresentata attraverso una freccia tratteggiata che collega due elementi: quello dipendente dal lato della coda e quello indipendente dal lato della freccia. La relazione di dipendenza prevede diverse specializzazioni atte ad evidenziare la semantica dell'utilizzo che ne viene fatto. In particolare, nel metamodello sono predefiniti una serie di specializzazioni standard (le metaclassi `Binding`, `Abstraction`, `Usage` e `Permission`). Anche a queste, tipicamente, si preferiscono stereotipi — diversi dei quali sono predefiniti nel metamodello — dotati di semantica specializzata.

La generalizzazione è una relazione tassonomica tra un elemento più generale (detto padre) e uno più specifico (detto figlio). Quest'ultimo è completamente consistente con quello più generale e, tipicamente, definisce ulteriori informazioni (in termini di struttura e comportamento). Pertanto un'istanza di un elemento figlio può essere utilizzata in ogni parte in cui è prevista oggetto dell'elemento padre. Nel caso di relazione di generalizzazione tra classi, alla nomenclatura "ecclesiale" (padre e figlio) si preferisce quella più specifica di superclasse e sottoclasse (*superclass* e *subclass*). La relazione di generalizzazione viene mostrata attraverso un segmento congiungente la sottoclasse alla superclasse, culminante con un triangolo vuoto posto nell'estremità di quest'ultima.

Un'associazione è una relazione strutturale tra due o più classificatori che descrive connessioni tra le relative istanze. In base al numero degli elementi coinvolti nella relazione, si hanno diverse "specializzazioni", quali l'associazione unaria, binaria ed *n*-aria. Il caso decisamente più frequente è costituito dall'associazione binaria che coinvolge due classificatori e specifica che da oggetti di un tipo è possibile (a meno di vincoli particolari) navigare a quelli dell'altro e viceversa. Tipicamente alle associazioni viene assegnato un nome per specificare la natura della relazione stessa e, al fine di eliminare ogni possibile fonte di ambiguità di lettura, spesso viene visualizzata a fianco del nome stesso una freccia indicante il verso di lettura.

Una relazione di associazione tra due classi mostrata senza alcuna direzione implica che, nell'ambito della relazione, entrambe le classi sono navigabili, quindi, data un'istanza di una classe, è possibile transitare nelle istanze dell'altra a cui è associata e viceversa. Qualora invece, in una relazione di associazione, non si voglia permettere alle istanze di una classe di "vedere" quelle dell'altra (essenzialmente, di invocarne i metodi), nella rappresentazione della relazione è necessario inserire una freccia indicante il verso di percorrenza. Pertanto gli oggetti istanza della classe posta nella coda dell'associazione (navigabilità =

`false`) possono invocare i metodi ed accedere agli attributi pubblici delle istanze della classe puntata dalla freccia (navigabilità = `true`), mentre non è possibile il contrario.

Poiché una associazione è relazione strutturale tra gli oggetti istanze delle classi coinvolte, è importante specificare, per ogni istanza di una classe, a quanti oggetti dell'altra può essere connessa e viceversa. Queste informazioni vengono definite molteplicità di un ruolo di associazione o più semplicemente "molteplicità" (*multiplicity*).

Ogniqualvolta un'entità prende parte a un'organizzazione, recita un determinato ruolo, quindi, similmente, anche le classi, partecipando ad una relazione, svolgono uno specifico ruolo. Questo viene espresso per mezzo di una stringa il cui scopo è renderne esplicita la semantica. Qualora si decida di visualizzare il ruolo, questo è collocato nei pressi dello spazio in cui il segmento dell'associazione incontra la classe a cui si riferisce.

Nel caso in cui due o più classi siano associate per mezzo di una relazione con molteplicità diversa da *1* a *1*, ossia almeno un'istanza di una classe possa essere connessa con un opportuno insieme di oggetti istanze dell'altra, può verificarsi il caso in cui sia necessario ordinare, secondo un preciso criterio, tali relazioni. Qualora ciò avvenga, la condizione è evidenziata riportando la parola chiave *ordered*, racchiusa da parentesi graffe, (`{ordered}`) nell'opportuno terminale di associazione.

Un altro insieme di vincoli che è possibile specificare in un'associazione è relativo alla modificabilità (*changeability*) dei legami che le istanze delle classi coinvolte nell'associazione instaurano. I valori ammessi sono tre: `changeable` (modificabile: default), `frozen` (congelato: dopo la creazione dell'oggetto nessuna relazione può essere aggiunta per mezzo dell'esecuzione di un'operazione nella classe sorgente della relazione), `addOnly` (sola aggiunta: relazioni possono essere aggiunte ogni qualvolta se ne abbia la necessità, ma, una volta create, queste non possono essere più rimosse).

In modo del tutto analogo a quanto sancito per gli attributi e i metodi, anche alle associazioni (che in ultima analisi non sono altro che particolari attributi) è possibile specificare il campo d'azione (*scope*). Le alternative previste sono due: istanza (*instance*) e classificatore (*classifier*), che equivale a dichiarare un'associazione statica.

Una associazione xor (*xor-association*) rappresenta una situazione in cui solo una delle potenziali associazioni, che una stessa classe può instaurare con altre, può verificarsi in un determinato istante di tempo per una specifica istanza della classe. Graficamente è rappresentata per mezzo di un segmento che unisce le associazioni (due o più) soggette al vincolo, con evidenziata la stringa `{xor}`.

In un'associazione tra classi, spesso si verifica la situazione in cui la relazione stessa possegga proprietà strutturali e comportamentali (attributi, operazioni e riferimenti ad altre classi). In tali circostanze è possibile ricorrere all'utilizzo della classe associazione (*association class*) che, come suggerisce il nome, possiede contemporaneamente proprietà di classe e di associazione. Graficamente è rappresentata attraverso il normale formalismo previsto per le classi con, in aggiunta, un segmento tratteggiato di connessione con la relazione che la origina.

Un'associazione tra classi mostra una relazione strutturale paritetica (la famosa *peer to peer*), per cui tra le classi coinvolte non è possibile distinguerne una concettualmente più importante delle altre: sono tutte allo stesso livello. Spesso però è necessario utilizzare relazioni del tipo "tutto–parte" (*whole–part*), in cui esiste una classe che rappresenta un concetto più "grande" (il tutto) costituito dalle restanti che rappre-

sentano concetti più piccoli (le parti). Questi tipi di relazioni sono dette di aggregazione e, in particolari circostanze, diventano di composizione. Graficamente un'aggregazione è visualizzata con un rombo vuoto dalla parte della classe aggregata (*whole*). Nel caso in cui l'aggregazione sia una composizione, allora il rombo viene colorato al proprio interno. La composizione è una forma di aggregazione con una forte connotazione di possesso e coincidenza della vita tra le classi parti e quella "tutto". Le parti possono essere generate anche in un tempo successivo alla creazione della classe composta (tutto), ma, una volta generate, queste vivono e sono distrutte con l'istanza della classe composta di appartenenza. Inoltre uno stesso oggetto, in ogni istante di tempo, può essere parte esclusivamente di una composizione. Quando un oggetto composto viene generato, esso si deve incaricare di creare le istanze delle sue parti e associarle correttamente a sé stesso. Quando poi un oggetto composto viene distrutto, esso ha la responsabilità di distruggere tutte le parti (ciò non significa che debba farlo direttamente). Ogni qualvolta si utilizza una relazione di composizione, in qualche modo è possibile pensare le classi rappresentanti le parti della relazione come private della classe composta.

Nella realizzazione di un modello, spesso è necessario dar luogo a relazioni di associazione particolari, in cui esiste il problema del lookup. In altre parole, in una relazione *1* a *n* (o *n* a *n*) tra due classi, fissato uno determinato oggetto istanza di una classe, è necessario specificare un criterio per individuare un preciso oggetto o un sottoinsieme di quelli associati istanze dell'altra classe. Tali circostanze si prestano a essere modellate attraverso l'associazione qualificata (*qualification*), il cui nome è dovuto all'attributo, o alla lista di attributi, detti qualificatori, i cui valori permettono di partizionare l'insieme delle istanze associate a quella di partenza.

Come è lecito attendersi, una associazione *n*-aria è una relazione che coinvolge più di due classi, tenendo presente che una stessa classe può apparire più volte. Così come l'autoassociazione è un particolare caso dell'associazione binaria, allo stesso modo, quest'ultima può essere considerata un caso particolare della relazione *n*-aria.

I diagrammi degli oggetti (*object diagram*) rappresentano una variante dei diagrammi delle classi, o meglio ne sono istanze e ne condividono gran parte della notazione. Rappresentano la famosa istantanea del sistema eseguita in un preciso istante di tempo di un'ipotetica esecuzione. Quindi, a differenza dei diagrammi delle classi, popolati di elementi astratti come classi e interfacce, i diagrammi degli oggetti sono colonizzati da oggetti sospesi in un particolare stato. 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.

I diagrammi degli oggetti, come quelli delle classi, sono utilizzati per illustrare la vista statica di disegno del sistema, però, a differenza di questi ultimi, utilizzano una prospettiva diversa, focalizzata su un esempio, reale o ipotetico, dello stato di evoluzione del sistema.