

Jim Arlow
Ila Neustadt

UML e Unified Process

*Analisi e progettazione
Object-Oriented*

McGraw-Hill

**J. Arlow
I. Neustadt**

www.mcgraw-hill.it
www.ateneonline.it
www.hyperbook.it

UML e Unified Process

*Analisi e progettazione
Object-Oriented*

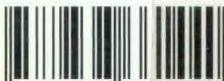
**Una guida per i "practitioners",
ricca di suggerimenti
metodologici sull'utilizzo
di UML nello sviluppo
dei sistemi informatici**

McGraw-Hill

A Division of The McGraw-Hill Companies



ISBN 88-386-6144-8



9 788838 661440

€ 29,50 (i.i.)

**Jim Arlow
Ila Neustadt**

UML e Unified Process

*Analisi e progettazione
Object-Oriented*

McGraw-Hill

Milano • New York • San Francisco • Washington D.C. • Auckland
Bogotá • Lisboa • London • Madrid • Mexico City • Montreal
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto

Introduzione	xv
Parte 1 Introduzione all'UML e all'UP	1
Capitolo 1 UML	3
1.1 Contenuto del capitolo	3
1.2 Cos'è l'UML?	3
1.3 La nascita dell'UML	5
1.4 Perché "unificato"?	6
1.5 L'UML e gli Oggetti	7
1.6 Struttura dell'UML	7
1.7 Costituenti fondamentali dell'UML	8
1.7.1 Entità	8
1.7.2 Relazioni	9
1.7.3 Diagrammi	10
1.8 Meccanismi comuni dell'UML	11
1.8.1 Specifiche	11
1.8.2 Ornamenti	12
1.8.3 Distinzioni comuni	13
1.8.4 Meccanismi di estendibilità	15
1.9 Architettura	17
1.10 Riepilogo	19
Capitolo 2 UP	21
2.1 Contenuto del capitolo	21
2.2 L'UP	21

2.3	La nascita dell'UP	23
2.4	L'UP e il RUP	25
2.5	Come istanziare l'UP su un progetto	27
2.6	Assiomi dell'UP	27
2.7	L'UP è un processo iterativo e incrementale	28
2.7.1	Iterazioni e flussi di lavoro	29
2.7.2	Baseline e incrementi	30
2.8	Struttura dell'UP	30
2.9	Le fasi dell'UP	32
2.9.1	Avvio: obiettivi	32
2.9.2	Avvio: flussi di lavoro	32
2.9.3	Milestone Avvio: Obiettivo del Ciclo di Vita	32
2.9.4	Elaborazione: obiettivi	33
2.9.5	Elaborazione: flussi di lavoro	34
2.9.6	Milestone Elaborazione: Architettura del Ciclo di Vita	34
2.9.7	Costruzione: obiettivi	35
2.9.8	Costruzione: flussi di lavoro	35
2.9.9	Milestone Costruzione: Capacità Operativa Iniziale	35
2.9.10	Transizione: obiettivi	36
2.9.11	Transizione: flussi di lavoro	36
2.9.12	Milestone Transizione: Rilascio del Prodotto	36
2.10	Riepilogo	37
Parte 2	Requisiti	39
Capitolo 3	Il flusso di lavoro dei requisiti	41
3.1	Contenuto del capitolo	41
3.2	Il flusso di lavoro dei requisiti	41
3.3	Requisiti del software: meta-modello	43
3.4	Flusso di lavoro dei requisiti in dettaglio	44
3.5	L'importanza dei requisiti	46
3.6	Definire i requisiti	46
3.6.1	Le Specifiche dei Requisiti di Sistema	47
3.6.2	Formulazione dei requisiti	47
3.6.3	Requisiti funzionali e non-funzionali	48
3.6.4	Raccolta dei requisiti: la mappa non è il territorio	48
3.7	Riepilogo	50
Capitolo 4	Modellazione dei casi d'uso	51
4.1	Contenuto del capitolo	51

4.2	Modellazione dei casi d'uso	52
4.3	Attività UP: individuare attori e casi d'uso	53
4.3.1	Il confine del sistema	54
4.3.2	Cosa sono gli attori?	54
4.3.3	Cosa sono i casi d'uso?	56
4.3.4	Il Glossario di Progetto	58
4.4	Attività UP: descrivere un caso d'uso	59
4.4.1	Specifiche del caso d'uso	59
4.4.2	Ramificazione di una sequenza	62
4.4.3	Ripetizione all'interno di una sequenza: Per (For)	64
4.4.4	Ripetizione all'interno di una sequenza: Fintantoché (While)	64
4.4.5	Mappatura dei requisiti	66
4.5	Casi d'uso complessi	66
4.5.1	Scenari	67
4.5.2	Specificare lo scenario principale	67
4.5.3	Specificare gli scenari secondari	67
4.5.4	Individuare gli scenari secondari	69
4.5.5	Quanti scenari?	69
4.6	Quando applicare la modellazione dei casi d'uso	70
4.7	Riepilogo	70
Capitolo 5 Modellazione dei casi d'uso: tecniche avanzate		73
5.1	Contenuto del capitolo	73
5.2	Generalizzazione tra attori	73
5.3	Generalizzazione tra casi d'uso	76
5.4	«include»	79
5.5	«estende»	81
5.5.1	Il caso d'uso di estensione	83
5.5.2	Segmenti inseribili multipli	83
5.5.3	Estensioni condizionali	85
5.6	Quando usare le tecniche avanzate	85
5.7	Riepilogo	86
Parte 3 Analisi		89
Capitolo 6 Il flusso di lavoro dell'analisi		91
6.1	Contenuto del capitolo	91
6.2	Il flusso di lavoro dell'analisi	91
6.2.1	Artefatti dell'analisi: meta-modello	93
6.2.2	Flusso di lavoro dell'analisi in dettaglio	93

6.3	Modello dell'analisi: regole pratiche	94
6.4	Riepilogo	95
Capitolo 7 Classi e oggetti		97
7.1	Contenuto del capitolo	97
7.2	Cosa sono gli oggetti?	97
7.2.1	Incapsulazione	99
7.2.2	Messaggi	101
7.3	Notazione UML per gli oggetti	102
7.3.1	Valori degli attributi degli oggetti	103
7.4	Cosa sono le classi?	103
7.4.1	Classi e oggetti	105
7.4.2	Istanziazione di un oggetto	106
7.5	Notazione UML per le classi	106
7.5.1	Sottosezione nome	107
7.5.2	Sottosezione attributo	108
7.5.3	Sottosezione operazione	111
7.5.4	Sintassi per le classi con stereotipo	112
7.6	Ambito	112
7.6.1	Ambito di istanza e ambito di classe	113
7.6.2	L'ambito determina l'accessibilità	114
7.7	Creazione e distruzione degli oggetti	114
7.7.1	Costruttori: classe ContoBancario	115
7.7.2	Distruttori: classe ContoBancario	115
7.8	Riepilogo	116
Capitolo 8 Individuare le classi di analisi		119
8.1	Contenuto del capitolo	119
8.2	Attività UP: analizzare un caso d'uso	119
8.3	Cosa sono le classi di analisi?	119
8.3.1	Anatomia di una classe di analisi	122
8.3.2	Cosa contraddistingue una buona classe di analisi?	123
8.3.3	Regole pratiche per le classi di analisi	124
8.4	Individuare le classi	125
8.4.1	Individuare le classi con l'analisi nome/verbo	126
8.4.2	Individuare le classi con l'analisi CRC	127
8.4.3	Cercare altre fonti di classi	128
8.5	Creazione di una prima bozza di modello dell'analisi	129
8.6	Riepilogo	130

Capitolo 9 Relazioni	133
9.1 Contenuto del capitolo	133
9.2 Cos'è una relazione?	133
9.3 Cos'è un collegamento?	134
9.3.1 Diagrammi degli oggetti	135
9.3.2 Percorsi	136
9.4 Cos'è un'associazione?	137
9.4.1 Sintassi per le associazioni	138
9.4.2 Molteplicità	139
9.4.3 Navigabilità	143
9.4.4 Associazioni e attributi	144
9.4.5 Classi associazione	146
9.4.6 Associazioni qualificate	148
9.5 Cos'è una dipendenza?	149
9.5.1 Dipendenze di uso	150
9.5.2 Dipendenze di astrazione	152
9.5.3 Dipendenze di accesso	152
9.6 Riepilogo	154
Capitolo 10 Ereditarietà e polimorfismo	159
10.1 Contenuto del capitolo	159
10.2 Generalizzazione	159
10.2.1 Classi e generalizzazione	159
10.3 Ereditarietà di classe	161
10.3.1 Ridefinizione	162
10.3.2 Classi e operazioni astratte	163
10.3.3 Livelli di astrazione	164
10.4 Polimorfismo	165
10.4.1 Esempio di polimorfismo	166
10.5 Riepilogo	168
Capitolo 11 Package di analisi	171
11.1 Contenuto del capitolo	171
11.2 Che cosa è un package?	171
11.3 Dipendenze tra package	173
11.4 Transitività	175
11.5 Package annidati	176
11.6 Generalizzazione tra package	178
11.7 Stereotipi di package	179

11.8	Analisi dell'architettura	179
11.8.1	Individuare i package di analisi	180
11.8.2	Dipendenze circolari tra package	181
11.9	Riepilogo	182
Capitolo 12 Realizzazione di caso d'uso		187
12.1	Contenuto del capitolo	187
12.2	Attività UP: analizzare un caso d'uso	187
12.3	Cosa sono le realizzazioni di caso d'uso?	187
12.4	Realizzazioni di caso d'uso: elementi	189
12.5	Diagrammi di interazione	190
12.6	Collaborazioni e interazioni	191
12.7	Diagrammi di collaborazione	192
12.7.1	Diagrammi di collaborazione in forma descrittore	192
12.7.2	Diagrammi di collaborazione in forma istanza	194
12.7.3	Interazioni tra oggetti	194
12.7.4	Multioggetti	198
12.7.5	Iterazione	200
12.7.6	Ramificazione e auto-delegazione	201
12.7.7	Concorrenza: oggetti attivi	203
12.7.8	Stato di un oggetto	206
12.8	Diagrammi di sequenza	208
12.8.1	Iterazione	211
12.8.2	Ramificazione e auto-delegazione	212
12.8.3	Concorrenza: oggetti attivi	212
12.8.4	Stato di un oggetto e vincoli	213
12.9	Riepilogo	215
Capitolo 13 Diagrammi di attività		219
13.1	Contenuto del capitolo	219
13.2	Cosa sono i diagrammi di attività?	219
13.3	Stati di azione	220
13.4	Stati di sottoattività	222
13.5	Transizioni	222
13.6	Decisioni	223
13.7	Biforazioni e ricongiunzioni	224
13.8	Corsie	225
13.9	Flussi degli oggetti	226
13.10	Segnali	228
13.11	Riepilogo	229

Parte 4 Progettazione	233
Capitolo 14 Il flusso di lavoro della progettazione	235
14.1 Contenuto del capitolo	235
14.2 Il flusso di lavoro della progettazione	235
14.3 Artefatti della progettazione: meta-modello	237
14.3.1 È necessario mantenere due modelli?	239
14.4 Flusso di lavoro della progettazione in dettaglio	240
14.5 Artefatti	241
14.5.1 Relazioni di origine tra artefatti	241
14.6 Riepilogo	242
Capitolo 15 Classi di progettazione	245
15.1 Contenuto del capitolo	245
15.2 Cosa sono le classi di progettazione?	245
15.3 Anatomia di una classe di progettazione	247
15.4 Classi di progettazione ben formate	248
15.4.1 Completezza e sufficienza	248
15.4.2 Essenzialità	249
15.4.3 Massima coesione	250
15.4.4 Minima interdipendenza	250
15.5 Ereditarietà	251
15.5.1 Aggregazione o ereditarietà?	251
15.5.2 Ereditarietà multipla	253
15.5.3 Ereditarietà o realizzazione di interfaccia?	254
15.6 Template di classe	255
15.7 Classi annidate	257
15.8 Riepilogo	258
Capitolo 16 Rifinitura delle relazioni di analisi	263
16.1 Contenuto del capitolo	263
16.2 Relazioni di progettazione	263
16.3 Aggregazione e composizione	265
16.4 Semantica dell'aggregazione	266
16.5 Semantica della composizione	268
16.5.1 Composizione e attributi	270
16.6 Come rifinire le relazioni di analisi	270
16.7 Associazioni uno-a-uno	271

16.8	Associazioni multi-a-uno	271
16.9	Associazioni uno-a-molti	272
16.10	Classi contenitore	272
16.10.1	Classi contenitore semplici dell'OCL	274
16.10.2	La mappa	275
16.11	Relazioni reificate	276
16.11.1	Associazioni multi-a-molti	276
16.11.2	Associazioni bidirezionali	277
16.11.3	Classi associazione	278
16.12	Riepilogo	279
Capitolo 17 Interfacce e sottosistemi		285
17.1	Contenuto del capitolo	285
17.2	Cos'è un'interfaccia?	285
17.3	Interfacce e sviluppo basato sui componenti	288
17.4	Individuare le interfacce	291
17.5	Progettare usando le interfacce	291
17.6	Cosa sono i sottosistemi?	292
17.6.1	Sottosistemi e interfacce	294
17.6.2	Il pattern Façade	295
17.6.3	Architettura fisica e pattern di layering	295
17.7	Vantaggi e svantaggi delle interfacce	297
17.8	Riepilogo	298
Capitolo 18 Realizzazione di caso d'uso della progettazione		301
18.1	Contenuto del capitolo	301
18.2	Realizzazione di caso d'uso della progettazione	302
18.3	Diagrammi di interazione della progettazione	302
18.4	Interazioni a livello di sottosistema	304
18.5	Riepilogo	306
Capitolo 19 Diagrammi di stato		307
19.1	Contenuto del capitolo	307
19.2	Diagrammi di stato	307
19.3	Macchine a stati e classi	309
19.4	Sintassi di base per i diagrammi di stato	310
19.5	Stati	310
19.5.1	Sintassi per gli stati	311

19.6	Transizioni	312
19.7	Eventi	313
19.7.1	Eventi di chiamata	313
19.7.2	Eventi di segnale	314
19.7.3	Eventi di variazione	315
19.7.4	Eventi del tempo	316
19.8	Riepilogo	316
Capitolo 20 Diagrammi di stato: tecniche avanzate		319
20.1	Contenuto del capitolo	319
20.2	Stati compositi	319
20.3	Stati compositi sequenziali	321
20.4	Stati compositi concorrenti	323
20.5	Comunicazione tra sotto-macchine	325
20.5.1	Comunicazione tramite attributi	326
20.5.2	Comunicazione tramite stati di sync	327
20.6	Stati con memoria	328
20.6.1	Stati con memoria semplice	328
20.6.2	Stati con memoria multilivello	329
20.7	Stato della sotto-macchina	330
20.8	Riepilogo	332
Parte 5 Implementazione		333
Capitolo 21 Il flusso di lavoro dell'implementazione		335
21.1	Contenuto del capitolo	335
21.2	Il flusso di lavoro dell'implementazione	335
21.3	Relazioni di origine del modello	337
21.4	Flusso di lavoro dell'implementazione in dettaglio	338
21.5	Artefatti	339
21.6	Riepilogo	339
Capitolo 22 Componenti		341
22.1	Contenuto del capitolo	341
22.2	Cos'è un componente?	341
22.3	Semplice esempio con Java	343
22.4	Esempio con Enterprise JavaBean	346
22.5	Riepilogo	348

Capitolo 23 Deployment	351
23.1 Contenuto del capitolo	351
23.2 Il diagramma di deployment	352
23.3 Sintassi per i diagrammi di deployment	352
23.4 Esempio con Enterprise JavaBean	354
23.5 Riepilogo	357
Appendice 1 Esempio di modello dei casi d'uso	359
A1.1 Introduzione	359
A1.2 Modello dei casi d'uso	359
A1.3 Esempi di casi d'uso	359
Appendice 2 L'XML e i casi d'uso	365
A2.1 Usare l'XML per i template di casi d'uso	365
Bibliografia	367
Indice analitico	369

Scopo del libro

Lo scopo di questo libro è di studiare il processo di analisi e progettazione object-oriented (OO - orientata agli oggetti) utilizzando il Linguaggio di Modellazione Unificato (UML) ed il Processo Unificato (UP).

L'UML ci fornisce una sintassi di modellazione visuale per la modellazione OO, mentre l'UP ci fornisce il framework di processo di ingegneria del software che ci dice come dobbiamo effettuare l'analisi e la progettazione OO.

Abbiamo tentato di rendere la nostra presentazione dell'UML e dell'UP il più semplice ed accessibile possibile.

Convenzioni

Per facilitare la consultazione del libro, ogni capitolo contiene uno schema della propria struttura, sotto forma di diagramma di attività UML. Questi diagrammi indicano attività di lettura e l'ordine in cui si consiglia di leggere le diverse sezioni che compongono i capitoli. La Figura I.1 ne riporta un esempio.

La maggior parte dei diagrammi di questo libro sono diagrammi UML. Le annotazioni che vi sono riportate in grigio non fanno parte della sintassi UML.

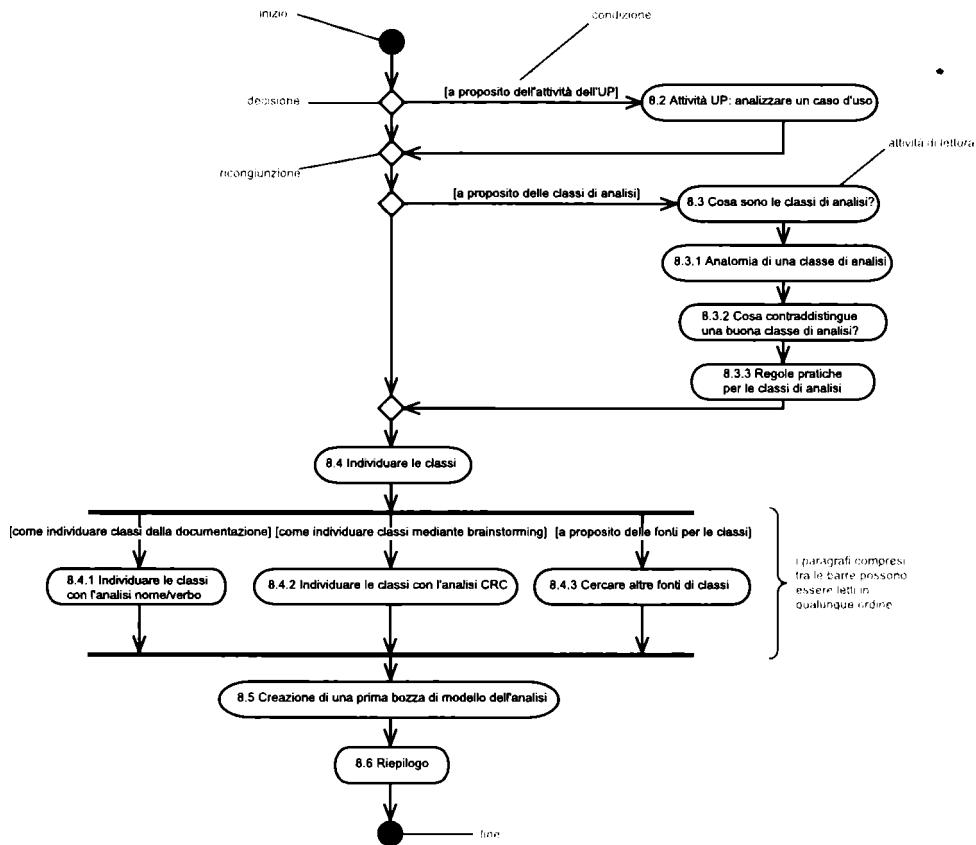
Nel libro abbiamo utilizzato diversi stili di carattere:

Questo stile è per gli elementi di modellazione UML.

Questo stile è per il codice.

Come leggere questo libro

Così tanti libri e così poco tempo per leggerli tutti! Tenendo presente questo, abbiamo progettato questo libro in modo che possa essere letto con diverse tecniche (oltre che sequenzialmente, dalla prima all'ultima pagina) a secondo delle esigenze dei diversi tipi di lettore.

**Figura I.1**

Lettura veloce

La lettura veloce è più adatta a chi vuole avere una panoramica dell'intero libro, o anche solo di qualche capitolo. È anche adatta per chi vuole avere una “sinossi per il management”.

- Scegliere un capitolo.
- Leggere la sezione “Contenuto del capitolo” e consultare il relativo diagramma della struttura del capitolo, giusto per orientarsi.
- Sfogliare il capitolo, esaminando le figure e le tabelle.
- Leggere la sezione “Cosa abbiamo imparato”.
- Se una qualche sezione ha suscitato interesse, tornare indietro a leggerla.

La lettura veloce è una tecnica rapida ed efficace per leggere questo libro. Anche se non sembra, la quantità di concetti che si può riuscire ad assimilare con questa tecnica è sorprendente. Si osservi comunque che questa tecnica funziona meglio se si parte avendo una buona idea delle informazioni che si desidera ottenere. Ad esempio “Vorrei capire come usare la modellazione dei casi d’uso.”

Consultazione

Per chi avesse bisogno di trovare informazioni su una parte specifica dell’UML o su una qualche tecnica di modellazione particolare, abbiamo fornito un indice analitico ed un sommario molto dettagliati, che dovrebbero consentire una consultazione rapida ed efficace.

Ripasso

Chi avesse bisogno di ripassare le proprie conoscenze di UML nel modo più veloce ed efficiente, deve leggere i riepiloghi schematici contenuti nella sezione finale di ciascun capitolo (“Cosa abbiamo imparato”). Per risolvere eventuali dubbi si può sempre tornare indietro e rileggere la relativa sezione di approfondimento.

Immersione rapida

Chi ha qualche minuto di tempo libero può prendere il libro ed aprirlo ad una pagina a caso. Abbiamo cercato di garantire che ci sia qualcosa di interessante in ogni pagina. Con questa tecnica, anche chi conosce l’UML molto bene potrebbe scoprire qualche nuovo dettaglio da assimilare.

Parte 1

Introduzione all'UML e all'UP

1.1 Contenuto del capitolo

Questo capitolo fornisce una breve panoramica della storia e della struttura di alto livello dell'UML. Introduce molti argomenti che verranno approfonditi nei capitoli successivi.

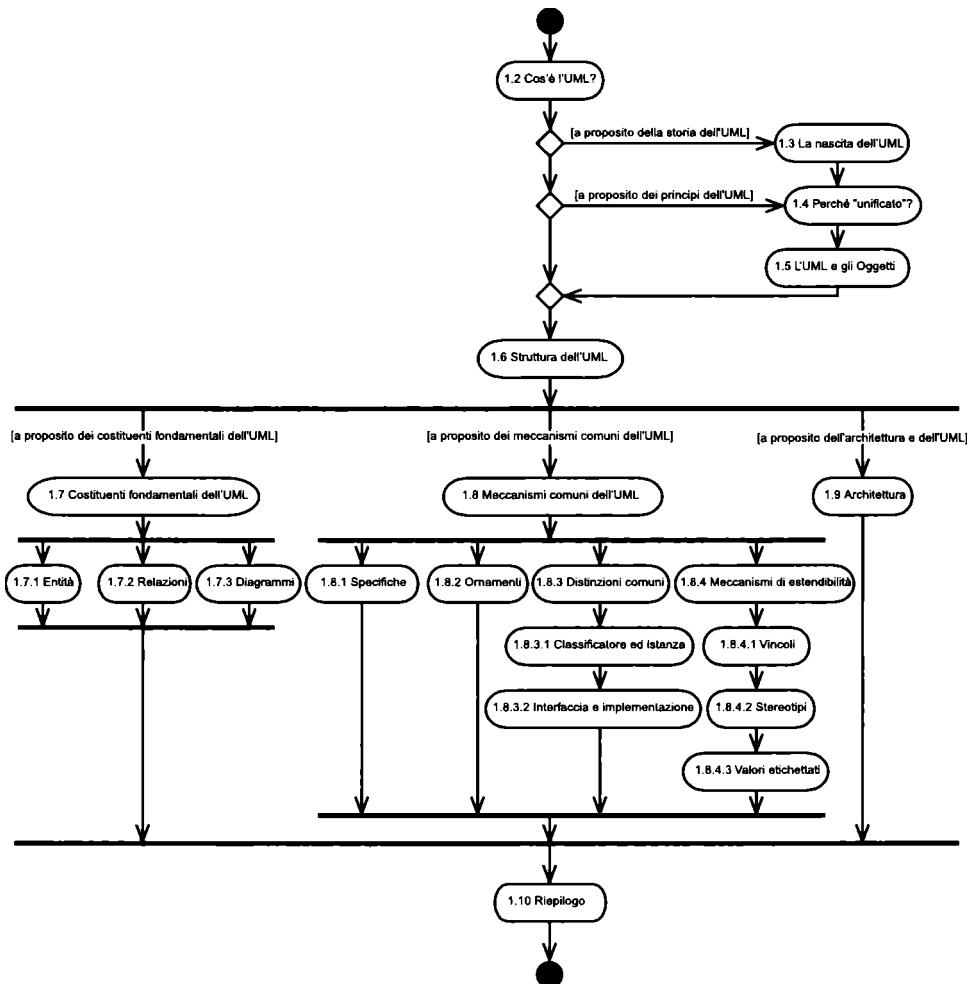
Chi ha poca dimestichezza con l'UML, dovrebbe cominciare leggendo della sua storia e dei suoi principi. Chi già conosce l'UML, o non ha interesse ad approfondire la sua storia, può saltare direttamente al Paragrafo 1.6, dove si discute della struttura dell'UML. Quest'ultima è suddivisa in tre argomenti principali, che possono essere letti in un qualunque ordine: costituenti fondamentali dell'UML (1.7), meccanismi comuni dell'UML (1.8), architettura e UML (1.9).

1.2 Cos'è l'UML?

Il Linguaggio Unificato per la Modellazione (Unified Modeling Language, UML) è un linguaggio visuale di modellazione dei sistemi di uso generico. Anche se l'UML viene tipicamente associato alla modellazione dei sistemi software Object Oriented (Orientati agli Oggetti, OO), in realtà ha un campo d'impiego molto più ampio, grazie ai meccanismi di estendibilità di cui è provvisto.

L'UML è stato studiato per incorporare tutte le migliori pratiche di modellazione e di ingegneria del software correntemente utilizzate. Per questo motivo è stato progettato specificatamente per essere implementato negli strumenti di ingegneria del software assistita da computer (Computer-Assisted Software Engineering, CASE). Questo anche in riconoscimento del fatto che la realizzazione dei grandi e moderni sistemi software tipicamente richiede il supporto di strumenti CASE. I diagrammi UML sono, per noi umani, leggibili, se non immediati, eppure possono essere facilmente generati da strumenti CASE.

È importante comprendere che UML *non* ci fornisce alcuna metodologia di modellazione. Ovviamente, alcuni aspetti metodologici sono impliciti negli elementi che costituiscono un modello UML, tuttavia l'UML ci fornisce esclusivamente la sintassi visuale da utilizzare per la costruzione dei modelli.

**Figura 1.1**

Lo Unified Process (UP, Processo Unificato) invece è una metodologia: ci dice quali sono le risorse, le attività e gli artefatti che dobbiamo utilizzare, effettuare e creare per poter modellare un sistema software.

L'UML *non* ci vincola all'uso di alcuna specifica metodologia e, anzi, può essere utilizzato con tutte le metodologie esistenti. L'UP utilizza l'UML come sintassi visuale di modellazione e si potrebbe dunque pensare che l'UP sia la metodologia *preferita* da utilizzare assieme all'UML. Anche se sicuramente l'UP è la metodologia che meglio si adatta all'UML, l'UML stesso può fornire (e fornisce) supporto alla modellazione visuale anche ad altre metodologie.¹

¹ Un esempio specifico di metodologia matura che utilizza l'UML come sintassi visuale, è il metodo OPEN (Object-oriented Process, Environment and Notation)... vedere il sito www.open.org.au

Lo scopo dell'UML e dell'UP è sempre stato quello di supportare e incorporare le migliori pratiche utilizzate dall'ingegneria del software, basandosi sull'esperienza maturata nell'ultimo decennio. Per assolvere questo scopo l'UML e l'UP *unificano* i precedenti tentativi di definire un linguaggio di modellazione visuale e un processo di ingegneria del software in quella che rappresenta la migliore soluzione evolutiva.

1.3 La nascita dell'UML

Prima del 1994 c'era molta confusione nel mondo dei metodi OO. Esistevano diversi linguaggi di modellazione visuale e metodologie, ciascuno con i propri punti di forza e le proprie debolezze, ciascuno con i propri sostenitori e i propri detrattori. Per quanto riguarda i linguaggi di modellazione visuale (sintetizzati nella Figura 1.2), i più diffusi erano Booch (il metodo Booch) e Rumbaugh (Object Modeling Technique, Tecnica di Modellazione degli Oggetti, OMT) che si spartivano oltre la metà del mercato. Per quanto concerne le metodologie, Jacobson era di gran lunga avanti a tutti dato che, anche se molti autori sostenevano di avere un "metodo", in realtà in molti casi si trattava solo di una sintassi di modellazione visuale affiancata da un insieme di aforismi e indicazioni più o meno utili.

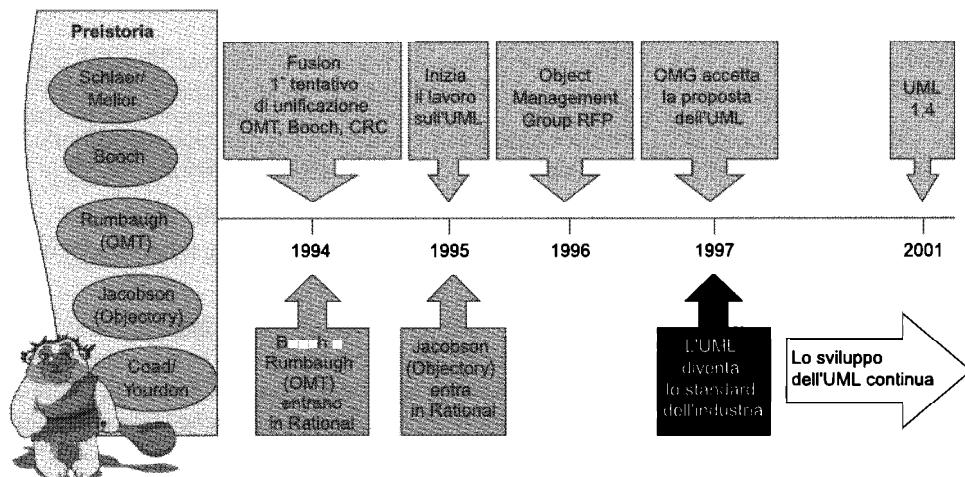


Figura 1.2

Nel 1994 ci fu un primo tentativo di unificazione, il metodo Fusion di Coleman. Tuttavia, per quanto lodevole, questo tentativo non coinvolse affatto gli autori originali dei metodi costituenti (Booch, Jacobson e Rumbaugh) e, inoltre, ci volle troppo tempo per giungere alla pubblicazione di un primo libro che spiegasse il metodo.

Fusion venne rapidamente sopraffatto dal corso degli eventi quando, sempre nel 1994, Booch e Rumbaugh si unirono alla Rational Corporation per lavorare sull'UML. In quei giorni, questa collaborazione preoccupò molti di noi, in quanto conferiva di fatto alla Rational oltre la metà del mercato delle metodologie. Tuttavia queste paure si rilevarono successivamente infondate, in quanto l'UML è divenuto uno standard non proprietario dell'industria del settore.

Nel 1996, l'Object Management Group (Gruppo di Lavoro sugli Oggetti, OMG) produsse una RFP (Request For Proposal, richiesta di proposta) per un linguaggio di modellazione visuale OO, a cui venne sottoposto l'UML. Nel 1997 l'OMG approvò l'UML e nacque così il primo standard non proprietario dell'industria per un linguaggio di modellazione visuale. Da allora tutti gli altri metodi che facevano concorrenza all'UML sono caduti in disuso, e l'UML resta l'unico linguaggio di modellazione OO standard dell'industria. Proprio mentre scriviamo questo libro, è stato appena rilasciato l'UML 1.4, e anche la versione 2.0 non sembra essere molto lontana.

In uno dei suoi libri, Grady Booch scrive “Se hai avuto una buona idea, allora è mia!”. In un certo modo questa frase sintetizza bene la filosofia di fondo dell'UML: prende il meglio di ciò che già esisteva, lo integra e lo arricchisce. Con questa forma di riuso l'UML incorpora molte delle migliori idee contenute nei metodi “preistorici”, pur rifiutandone alcune delle conclusioni e peculiarità più estreme.

1.4 Perché “unificato”?

L'unificazione operata dall'UML non deve essere vista esclusivamente in senso storico. L'UML vuole essere un linguaggio unificante anche sotto altri aspetti.

- **Ciclo di sviluppo:** l'UML fornisce una sintassi visuale per la modellazione di tutte le fasi del ciclo di sviluppo, dall'analisi dei requisiti all'implementazione.
- **Domini applicativi:** l'UML è stato utilizzato per modellare di tutto, dai sistemi *embedded*, che lavorano in tempo reale, ai sistemi direzionali per il supporto decisionale.
- **Linguaggi e piattaforme di sviluppo:** l'UML è indipendente dal linguaggio di sviluppo e dalla piattaforma. Ovviamente offre un supporto eccellente per i linguaggi OO puri (Smalltalk, Java, C#), ma è anche utile per linguaggi OO ibridi, come il C++, o linguaggi basati sugli oggetti, come il Visual Basic. È anche stato utilizzato per la modellazione per linguaggi non OO, come il C.
- **Processi di sviluppo:** anche se l'UP e le sue varianti sono i processi di sviluppo preferiti per i sistemi OO, l'UML può supportare (e supporta) molti altri processi di ingegneria del software.
- **Rappresentazione di concetti interni:** l'UML, tenta, con coraggio, di essere consistente e uniforme anche nell'applicazione della propria sintassi visuale anche a un piccolo insieme dei propri concetti interni. Non sempre (almeno per ora) ci riesce, ma si tratta di un significativo miglioramento rispetto ai precedenti tentativi.

1.5 L'UML e gli Oggetti

La premessa fondamentale dell'UML è che noi possiamo modellare i sistemi software (ma non solo) come *insiemi di oggetti che collaborano* tra loro. Questo ovviamente è molto adatto ai sistemi software e ai linguaggi OO, ma si presta anche molto bene per modellare i processi aziendali e in altri campi applicativi.

Un modello UML ha due aspetti:

- struttura statica: tipi di oggetto necessari per modellare il sistema e come sono tra loro correlati;
- comportamento dinamico: ciclo di vita di questi oggetti e come collaborano per fornire le funzionalità richieste al sistema.

Questi due aspetti del modello UML sono complementari: nessuno dei due è completo senza l'altro.

Discuteremo del concetto di oggetti (e di classi) in modo approfondito nel Capitolo 7. Fino ad allora, basti pensare a un oggetto come a un insieme aggregato di dati e comportamenti. In altre parole, gli oggetti possono contenere informazioni e possono eseguire funzioni.

1.6 Struttura dell'UML

Per capire come l'UML possa funzionare come linguaggio visuale, possiamo iniziare a esaminarne la struttura. Questa è illustrata nella Figura 1.3 (come si vedrà più avanti, si tratta di un diagramma UML valido). Questa struttura è composta da:

- costituenti fondamentali: gli elementi, le relazioni e i diagrammi di base dell'UML;
- meccanismi comuni: tecniche comuni per raggiungere specifici obiettivi con l'UML;
- architettura: il modo in cui l'UML esprime l'architettura di un sistema.

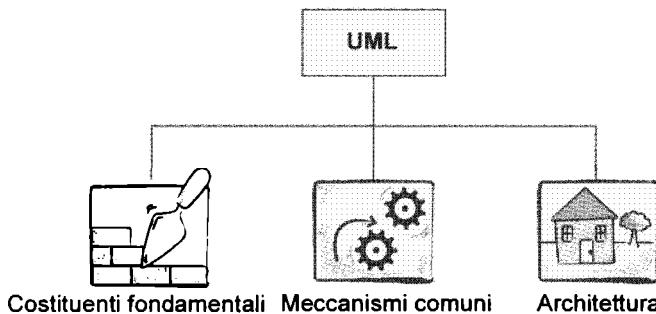
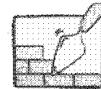


Figura 1.3

Comprendendo come sia strutturato l'UML, si può anche capire meglio come sono organizzate le informazioni presentate nel resto di questo libro. Questo diagramma, inoltre, evidenzia che l'UML stesso può essere visto come un sistema. In effetti, l'UML è stato modellato e progettato utilizzando proprio l'UML! Il diagramma rappresenta il meta-modello del sistema UML.

1.7 Costituenti fondamentali dell'UML



Secondo *The UML User Guide* [Booch 2], l'UML è composto di soli tre costituenti fondamentali:

- entità: gli elementi di modellazione;
- relazioni: legano tra loro le entità; le associazioni definiscono come due o più entità sono semanticamente correlate;
- diagrammi: le *viste* dei modelli UML mostrano insiemi di elementi che “raccontano una storia” che riguarda un sistema software, e sono lo strumento a disposizione per visualizzare *cosa* farà un sistema (diagrammi dell'analisi), e *come* lo farà (diagrammi della progettazione).

Nelle prossime tre sezioni saranno approfonditi meglio i concetti di entità, relazioni e diagrammi.

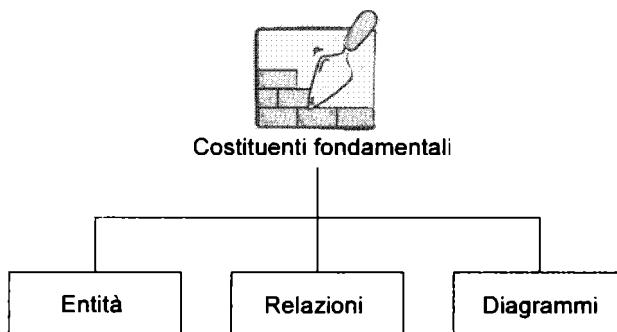


Figura 1.4

1.7.1 Entità

Le entità in UML possono essere suddivise in quattro categorie:

- entità strutturali: i sostantivi di un modello UML, quali classe, interfaccia, collaborazione, caso d'uso, classe attiva, componente, nodo;
- entità comportamentali: i verbi di un modello UML, quali interazioni e macchine a stati;

- entità di raggruppamento: il *package*, che viene utilizzato per raggruppare elementi semanticamente correlati in unità coesive;
- entità informative: l'annotazione, che può essere aggiunta al modello per fissare informazioni particolari, in modo simile a un comune giallino adesivo.

Saranno esaminati questi elementi più da vicino, e come possano essere utilizzati con profitto nella modellazione UML, iniziando dalla Parte 2.

1.7.2 Relazioni

In un modello, le relazioni mostrano come due o più entità siano tra loro correlate. Per farsi un'idea del ruolo che le relazioni rivestono nei modelli UML, basta pensare a una famiglia e a tutte le relazioni tra i vari familiari. Le relazioni consentono di fissare i legami significativi (semanticci) tra gli elementi. In un modello, le relazioni si possono applicare alle entità strutturali e alle entità di raggruppamento, così come illustrato nella Figura 1.5.

La modellazione UML richiede una buona conoscenza dei diversi tipi di relazione, dei loro diversi significati e utilizzi. Tuttavia si tratterà in modo approfondito questa parte semantica più avanti nel libro. Nella Tabella 1.1 si riporta un breve elenco di relazioni.

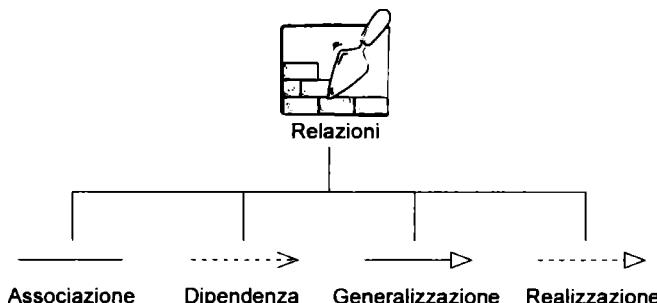


Figura 1.5

Tabella 1.1

Tipo di relazione	Descrizione	Sezione
Associazione	Describe l'insieme di collegamenti tra entità	9.4
Dipendenza	La variazione di un'entità impatta sulla semantica di una diversa entità	9.5
Generalizzazione	Un elemento è una specializzazione di un altro e può quindi sostituire quello più generico	10.2
Realizzazione	Una relazione tra classificatori per cui un classificatore definisce un contratto che l'altro classificatore si impegna a soddisfare	12.03

1.7.3 Diagrammi

In tutti gli strumenti CASE che gestiscono l'UML, le nuove entità e relazioni che vengono create sono aggiunte al modello. Il modello è l'archivio di tutte le entità e le relazioni che sono state create per descrivere il comportamento richiesto al sistema software che si vuole progettare.

I diagrammi sono *viste* o *finestre* che consentono di vedere il contenuto del modello. Il diagramma *non* è il modello! Questa distinzione è fondamentale. Un'entità o una relazione può essere eliminata da un diagramma o anche da tutti i diagrammi, ma può continuare a esistere nel modello. In effetti, resta nel modello fin quando non ne viene esplicitamente eliminata. I principianti spesso commettono l'errore di eliminare le entità dai diagrammi, ma non dal modello.

In totale esistono nove diversi tipi di diagrammi UML, elenchiati nella Figura 1.6. Di solito questi diagrammi vengono suddivisi tra quelli che modellano la struttura statica del sistema (il cosiddetto modello statico) e quelli che modellano la struttura dinamica del sistema (il modello dinamico). Il modello statico fissa le entità e le relazioni strutturali tra le entità; il modello dinamico fissa il modo in cui le entità interagiscono per generare il comportamento richiesto al sistema software. Il modello statico e quello dinamico saranno approfonditi a partire dalla Parte 2.

Non è necessario seguire un determinato ordine nella creazione dei diagrammi UML, anche se tipicamente si comincia con un diagramma dei casi d'uso, per definire lo scopo del sistema. In effetti, spesso si lavora parallelamente su diversi diagrammi, ridefinendo meglio ciascuno di essi, man mano che si scoprono e si mettono a fuoco informazioni e dettagli aggiuntivi sul sistema software che si sta progettando. I diagrammi costituiscono, dunque, non solo una vista sul modello, ma anche il principale strumento per immettere le informazioni nel modello.

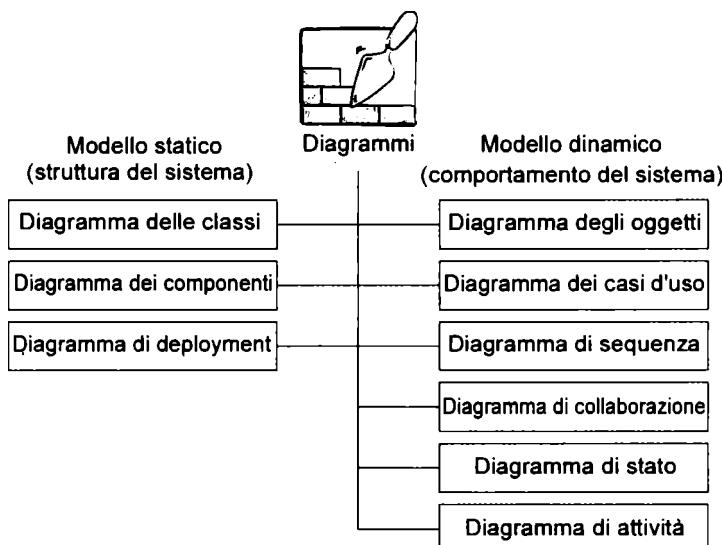


Figura 1.6

1.8 Meccanismi comuni dell'UML



L'UML prevede quattro meccanismi comuni che vengono applicati in modo consistente in tutto il linguaggio. Essi descrivono quattro diverse strategie alla modellazione OO che possono essere applicate e riapplicate in diversi contesti UML. Si osservi, ancora una volta, quanto sia semplice ed elegante la struttura dell'UML (Figura 1.7).

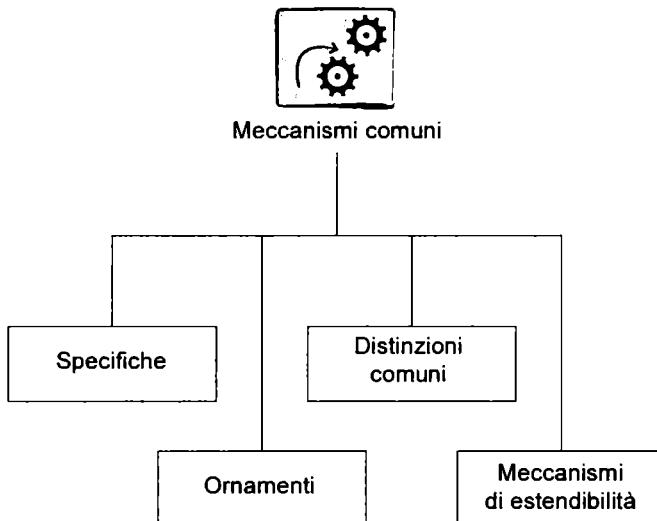


Figura 1.7

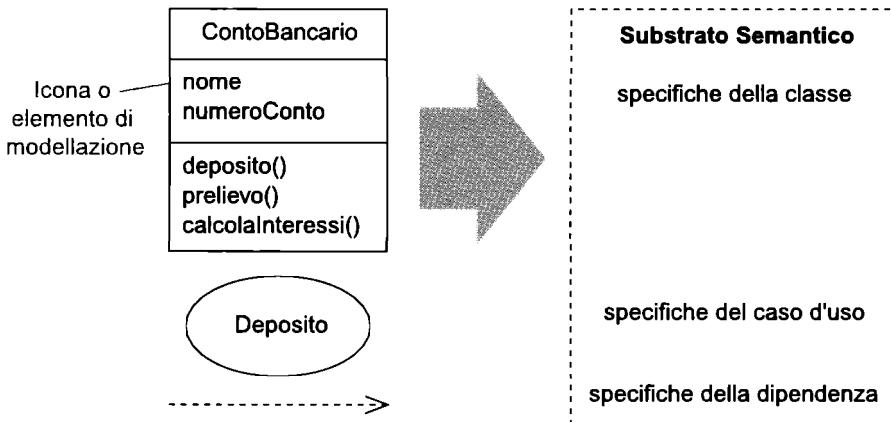
1.8.1 Specifiche

I modelli UML possiedono almeno due dimensioni: una grafica, che consente di visualizzare il modello utilizzando diagrammi e icone, e una dimensione testuale, che è costituita dalle specifiche dei vari elementi di modellazione. Le specifiche sono la descrizione testuale della semantica di un elemento.

Si può, per esempio, rappresentare graficamente una classe, in questo caso la classe ContoBancario, come un rettangolo suddiviso in rettangoli più piccoli (Figura 1.8), ma questa semplice rappresentazione non dice nulla sul significato funzionale di quella classe. La semantica sottostante gli elementi di modellazione è fissata nelle loro specifiche, e senza queste specifiche non si può far altro che tentare di indovinare ciò che gli elementi di modellazione rappresentano effettivamente.

L'insieme delle specifiche costituisce il vero “contenuto” del modello, quel *substrato semantico* che tiene insieme il modello e lo riempie di significato. I diversi diagrammi sono solo viste o proiezioni visuali di quel contenuto.

Questo substrato semantico viene tipicamente aggiornato utilizzando uno strumento CASE, il quale consente l'immissione, la visualizzazione e la modifica delle specifiche di ciascun elemento di modellazione.

**Figura 1.8**

L'UML offre una grande flessibilità durante la costruzione di un modello. In particolare un modello può essere:

- parzialmente nascosto: alcuni elementi possono esistere nel modello, ma essere nascosti in alcuni diagrammi, in modo da semplificare la visualizzazione;
- incompleto: alcuni elementi del modello possono essere del tutto assenti;
- inconsistente: il modello può contenere incongruenze.

È molto importante che i vincoli di completezza e consistenza non siano rigidi. Si vedrà, infatti, che i modelli si evolvono nel tempo e vengono sottoposti a molte modifiche. Tuttavia è pur sempre necessario aspirare a un modello sufficientemente *consistente* e *completo*, in modo da consentire la costruzione di un sistema software.

Una pratica UML diffusa è quella di cominciare con un modello per lo più grafico, che consente di iniziare a visualizzare il sistema, e successivamente aggiungere significato al substrato semantico man mano che il modello si evolve. Tuttavia, affinché un modello possa essere considerato utile o completo, è *necessario* che le specifiche del modello siano presenti. Se non lo fossero, non si ha veramente un modello, ma solo un insieme senza significato di rettangoli e figure connessi da linee! In effetti, un errore di modellazione comunemente commesso dai principianti è proprio la “morte da diagrammi”: il modello ha troppi diagrammi e troppe poche specifiche.

1.8.2 Ornamenti

Una caratteristica molto utile dell'UML è che ogni elemento di modellazione viene reso con un simbolo molto semplice, al quale è possibile aggiungere un certo numero di ornamenti, nel caso si renda necessario mostrare maggiori informazioni sul diagramma.

Questo vuol dire che si può iniziare a costruire un modello molto astratto usando i simboli base, aggiungendo magari uno o due ornamenti, per poi rifinire progressivamente il modello aggiungendovi nuovi ornamenti, fin quando il modello non contiene tutti i dettagli che servono.

È importante ricordarsi che ogni diagramma UML è solo una vista del modello. Gli ornamenti devono essere visualizzati solo se aumentano la leggibilità e la chiarezza generale del diagramma, o se evidenziano una qualche caratteristica importante del modello. Non è necessario far vedere tutto su ciascun diagramma: è molto più importante che il diagramma sia comprensibile, che illustri esattamente ciò che deve illustrare e che sia di facile lettura.

La Figura 1.9 mostra che il simbolo minimo per rappresentare una classe è un rettangolo contenente il nome della classe. Tuttavia è possibile estendere questa vista minimalista, esponendo ulteriori caratteristiche del modello sottostante utilizzando degli ornamenti. Il testo grigio della figura indica dei possibili ornamenti facoltativi.

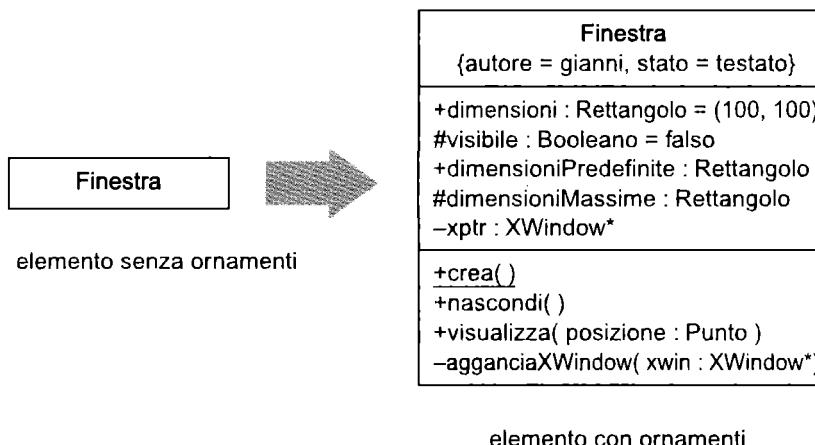


Figura 1.9

1.8.3 Distinzioni comuni

Le distinzioni comuni descrivono i diversi modi di ragionare sul mondo che ci circonda. L'UML prevede due distinzioni comuni: classificatore/istanza, e interfaccia/implementazione.

1.8.3.1 Classificatore e istanza

L'UML consente di definire una nozione astratta di un certo tipo di entità (per esempio, un conto bancario), ma anche di gestire specifiche e concrete istanze di quella stessa astrazione (il “mio conto bancario” o il “tuo conto bancario”).

La nozione astratta di un tipo di entità è un “classificatore”, mentre le entità specifiche e concrete sono “istanze”. Questo è un concetto importante e abbastanza facile da comprendere. Si possono trovare intorno a noi esempi di classificatori e istanze. Basti pensare a questo libro di UML: si potrebbe dire che il concetto astratto del libro è “UML e Unified Process”, ma che esistono molte istanze di questo libro, tra cui quella che si trova sulla mia scrivania in questo momento. Si vedrà come la distinzione classificatore/istanza sia un concetto fondamentale che permea l’UML.

In UML un’istanza di solito usa la stessa icona del corrispondente classificatore, ma il nome dell’icona, nel caso di un’istanza, è sottolineato. Questa subdola distinzione visuale può essere difficile da padroneggiare per un principiante.

L’UML definisce i classificatori elencati nella Tabella 1.2, approfonditi più avanti nel libro.

Tabella 1.2

Classificatore	Significato	Paragrafo
Attore	Un ruolo svolto da un utente esterno del sistema a cui il sistema fornisce del valore aggiunto	4.3.2
Classe	La descrizione di un insieme di oggetti che condividono le stesse caratteristiche	7.4
Ruolo classificatore	Un classificatore il cui ruolo in una collaborazione è ben determinato	12.5
Componente	Una parte fisica e sostituibile di un sistema che soddisfa o realizza una o più interfacce: buoni esempi di componenti possono essere i controlli ActiveX e i JavaBean	22.2
Tipo di dato	Un tipo il cui valore non ha identità, quali i tipi primitivi, int, float e char messi a disposizione in linguaggi quali C++ e Java: per esempio, tutte le istanze di int che hanno valore 4 sono identiche e non possono essere fra loro distinte. I linguaggi OO puri, quali Smalltalk, non hanno tipi di dati	
Interfaccia	Un insieme di operazioni che vengono usate per definire il servizio offerto da una classe o da un componente	17.2
Nodo	Un elemento fisico presente a <i>run-time</i> che rappresenta una risorsa computazionale, per esempio, un PC	23.2
Segnale	Un messaggio asincrono che viene passato da un oggetto a un altro	13.10
Sottosistema	Un raggruppamento di elementi; alcuni di questi possono definire il comportamento offerto dagli elementi in essi contenuti	17.6
Caso d’uso	La descrizione di una sequenza di azioni che il sistema effettua per fornire a un utente un valore aggiunto	4.3.3

1.8.3.2 Interfaccia e implementazione

Questa distinzione serve per separare che cosa fa un oggetto (la sua interfaccia) da come lo fa (la sua implementazione). Per esempio, guidando un'automobile, si interagisce con un'interfaccia molto semplice e ben definita. Questa interfaccia può essere implementata in modo molto differente da ciascuna automobile in particolare.

Un'interfaccia definisce un contratto (che ha molto in comune con un contratto legale) che le specifiche implementazioni garantiscono di rispettare. Questa separazione tra quanto un oggetto promette di fare e la reale implementazione di quella promessa è un concetto UML molto importante. Ne discuteremo in dettaglio nel Capitolo 17.

È facile trovare, intorno a noi, esempi concreti di interfacce e implementazioni. Per esempio, i pulsanti sul pannello di un videoregistratore forniscono un'interfaccia (relativamente) semplice per quello che in realtà è un meccanismo piuttosto complesso. L'interfaccia protegge da questa complessità, nascondendola.

1.8.4 Meccanismi di estendibilità

Gli ideatori dell'UML si resero conto che era semplicemente impossibile progettare un linguaggio di modellazione completamente universale, che potesse soddisfare le esigenze di tutti, oggi e domani. Per questo motivo, l'UML incorpora tre semplici meccanismi di estendibilità sintetizzati nella Tabella 1.3.

Nelle prossime tre sezioni si approfondiscono questi tre meccanismi di estendibilità.

Tabella 1.3

Meccanismo di estendibilità	Descrizione
Vincoli	Estendono la semantica di un elemento consentendo di aggiungere nuove regole.
Stereotipi	Definiscono un nuovo elemento di modellazione UML basandoci su uno esistente: è possibile definire la semantica degli stereotipi, che aggiungono nuovi elementi al meta-modello UML.
Valori etichettati	Permettono di estendere la definizione di un elemento tramite l'aggiunta di nuove informazioni specifiche.

1.8.4.1 Vincoli

Un vincolo è una frase di testo racchiusa tra parentesi graffe ({}), che definisce una certa condizione o regola che riguarda l'elemento di modellazione e che *deve* risultare sempre vera. In altre parole, il vincolo limita in qualche modo qualche caratteristica dell'elemento in questione. Si possono trovare esempi di vincoli in molti punti del libro.

1.8.4.2 Stereotipi

The UML Reference Manual [Rumbaugh 1] dice: “Uno stereotipo rappresenta una variazione di un elemento di modellazione esistente, che ha la stessa forma (quali attributi e relazioni), ma ha un diverso scopo”.

Gli stereotipi consentono di introdurre nuovi elementi di modellazione basandosi sugli elementi *esistenti*. Si usano facendo seguire al nuovo elemento il nome dello stereotipo tra virgolette angolari («...»). Ogni elemento di modellazione può avere al massimo uno stereotipo.

Ogni stereotipo può definire un insieme di valori e vincoli che verranno applicati all'elemento stereotipato. Si può anche associare allo stereotipo una nuova icona, un colore o un motivo di sfondo. In realtà, si cerca di evitare l'uso significativo del colore e dei motivi nei modelli UML, in quanto alcuni lettori (come per esempio i daltonici) potrebbero avere difficoltà a interpretare i diagrammi, e comunque spesso i diagrammi devono poi essere stampati in bianco e nero. Tuttavia è una pratica abbastanza comune associare allo stereotipo una nuova icona. Questo consente di estendere, in modo controllato, la notazione grafica dell'UML.

Dato che gli stereotipi introducono *nuovi* elementi di modellazione con un diverso scopo, è necessario definire la semantica di questi nuovi elementi da qualche parte. Come si può fare? Ebbene, qualora lo strumento CASE non fornisca supporto per la documentazione degli stereotipi, la maggior parte dei modellatori aggiunge un'annotazione al modello o inserisce un collegamento a un documento esterno in cui viene definito lo stereotipo. A dire il vero, a oggi, il supporto per gli stereotipi offerto dagli strumenti CASE è incompleto: molti strumenti offrono un supporto limitato per l'uso degli stereotipi, ma non tutti consentono la definizione della parte semantica.

Gli stereotipi possono essere modellati utilizzando l'elemento classe (Capitolo 7) seguito dallo speciale stereotipo predefinito UML "stereotipo". In questo modo si crea un meta-modello del sistema di stereotipi. Si tratta di un meta-modello, in quanto è un modello di un elemento di modellazione e, quindi, si trova su di un livello di astrazione completamente diverso dai normali modelli UML di sistemi o processi.

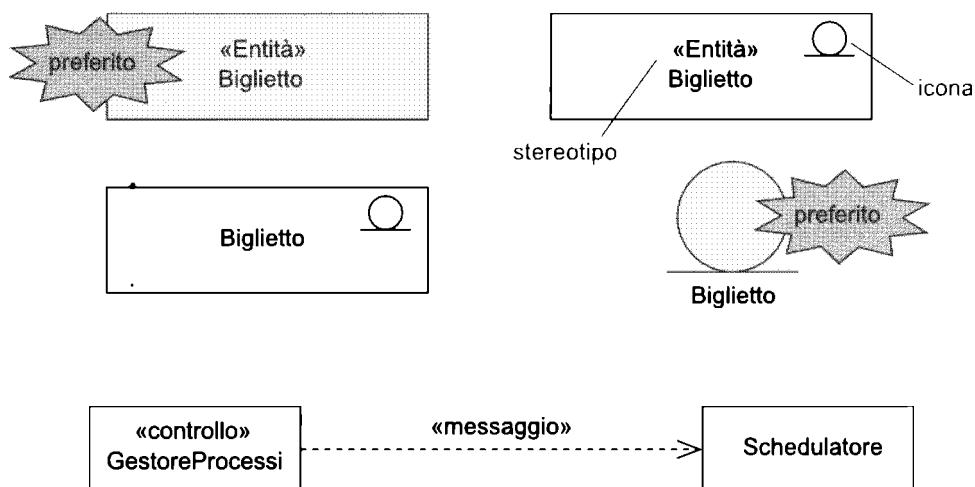


Figura 1.10

Dato che è un meta-modello, non dovrebbe mai trovarsi insieme ai modelli normali: deve essere *sempre* gestito come un modello separato. Potrebbe non valere la pena di creare un nuovo modello solo per gli stereotipi, a meno che ci sia bisogno di definirne molti. Dato che questo capita di rado, molti modellatori tendono a documentare gli stereotipi con una semplice annotazione o con un documento esterno.

Ci sono diversi modi di indicare gli stereotipi sui diagrammi. Tuttavia quelli più diffusi sono il nome dello stereotipo compreso tra “ ” o l’uso di un’icona apposita. Gli altri modi tendono a essere utilizzati molto meno e gli strumenti CASE spesso non li mettono tutti a disposizione. La Figura 1.10 mostra alcuni altri modi (N.B. Le stelle *non* fanno parte della sintassi UML; evidenziano semplicemente i due modi preferiti).

È possibile stereotipare non solo classi, ma anche relazioni. Nel libro si troveranno molti esempi del genere.

1.8.4.3 Valori etichettati

In UML si definisce proprietà un qualunque valore associato a un elemento di modellazione. Molti elementi hanno un gran numero di proprietà predefinite. Alcune di queste sono visualizzabili nei diagrammi, mentre altre fanno solo parte del substrato semantico del modello.

L’UML consente di utilizzare i valori etichettati per aggiungere nuove proprietà agli elementi di modellazione. Il valore etichettato è un concetto molto semplice: è un nome o identificativo a cui è possibile associare un valore. Ecco la sintassi per i valori etichettati: { etichetta1 = valore1 , etichetta2 = valore2 , ... , etichettaN = valoreN }. Si tratta di un elenco separato da virgolette di coppie etichetta/valore, separate dall’uguale. L’elenco dei valori etichettati deve essere racchiuso tra parentesi graffe.

Alcune etichette rappresentano semplicemente delle informazioni aggiuntive da applicare agli elementi di modellazione, come, per esempio, {autore = Jim Arlow}. Tuttavia altre etichette possono invece definire proprietà di nuovi elementi di modellazione definiti da uno stereotipo. In questo caso, non si dovrebbe associare i valori etichettati agli elementi modellati, ma direttamente allo stereotipo stesso. In questo modo, quando lo stereotipo verrà applicato a un elemento di modellazione, verrà automaticamente arricchito con tutte le etichette dello stereotipo.

1.9 Architettura



The UML Reference Manual [Rumbaugh 1] definisce l’architettura di sistema come: “la struttura organizzativa di un sistema, inclusa la sua scomposizione in parti, la loro connettività, l’interazione, i meccanismi e i principi guida che insieme formano il progetto del sistema stesso”. La IEEE definisce l’architettura di sistema come: “il concetto, il più astratto possibile, che descrive un sistema nel suo ambiente”.

L’architettura serve per fissare gli aspetti strategici della struttura ad altro livello di un sistema. A tal fine, l’UML definisce quattro diverse viste del sistema: la vista logica, la vista dei processi, la vista di implementazione e la vista di *deployment*. Queste vengono, inoltre, integrate con una quinta vista, la vista dei casi d’uso, così come mostrato nella Figura 1.11.

Ecco, in sintesi, ciascuna di queste viste.

- Vista logica: fissa la terminologia del dominio del problema sotto forma di insieme di classi e oggetti. L'enfasi deve essere posta su come gli oggetti e le classi che compongono il sistema, implementano il comportamento richiesto al sistema.
- Vista dei processi: modella i *thread* e i processi del sistema sotto forma di classi attive. In realtà, si tratta di una variante orientata ai processi della vista logica e contiene gli stessi tipi di elementi.

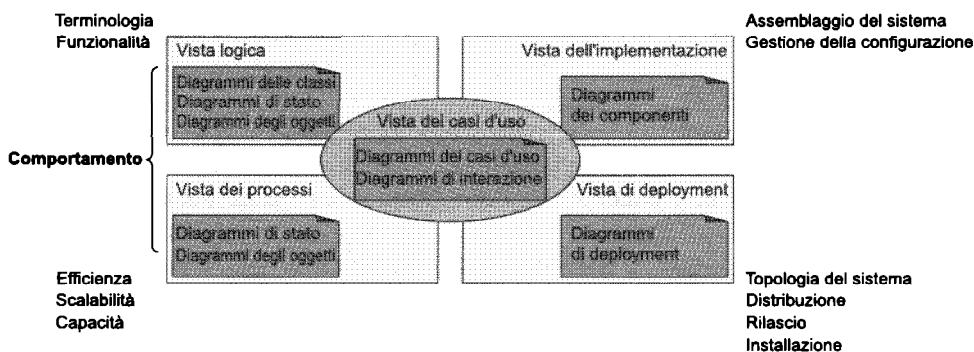


Figura 1.11 Adattata dalla Figura 5.1 [Kruchten 1], con il consenso della Addison-Wesley.

- **Vista di implementazione:** modella i file e i componenti che costituiscono la base di codice fisica del sistema. Deve, inoltre, illustrare le dipendenze tra i componenti e la gestione della configurazione dei sottoinsiemi di componenti, in modo da definire il concetto di versione del sistema.
- **Vista di deployment:** modella lo sviluppo fisico del sistema come componenti da installare su un insieme di nodi computazionali fisici, quali computer e periferiche. Consente di modellare la dislocazione e la distribuzione dei componenti su tutti i nodi di un sistema distribuito.
- **Vista dei casi d'uso:** tutte le altre viste derivano dalla vista dei casi d'uso. Questa vista fissa i requisiti di base di un sistema come un insieme di casi d'uso (vedere Capitolo 4) e, dunque, gli elementi di base per la costruzione delle altre viste.

Una volta creata questa architettura 4+1, saranno esplorati tutti gli aspetti fondamentali del sistema con i modelli UML. Seguendo il ciclo di vita dell'UP, questa architettura 4+1 non verrà creata tutta in un unico momento, ma piuttosto si evolverà nel tempo.

La modellazione UML vista all'interno di un contesto UP è dunque un processo ad approssimazioni successive il cui scopo ultimo è quello di produrre un architettura 4+1 in grado di fissare le informazioni *necessarie e sufficienti* per consentire la costruzione del sistema.

1.10 Riepilogo

Il capitolo ha presentato un'introduzione alla storia, alla struttura, ai concetti e alle caratteristiche principali dell'UML. Sono stati trattati i seguenti concetti.

1. L'UML è un linguaggio di modellazione visuale standard per l'industria, non proprietario, estendibile e approvato dall'OMG.
2. L'UML non è una metodologia.
3. L'UP e le sue varianti sono le metodologie che meglio complementano l'UML.
4. La modellazione a oggetti vede il mondo come un sistema di oggetti che interagiscono tra di loro. Gli oggetti contengono informazioni e possono eseguire funzioni. I modelli UML hanno:
 - struttura statica: quali tipi di oggetti sono importanti e come sono in relazione tra loro;
 - comportamento dinamico: come gli oggetti collaborano tra loro per eseguire le funzioni richieste al sistema.
5. L'UML è composto da costituenti fondamentali:
 - entità:
 - entità strutturali: i sostantivi di un modello UML;
 - entità comportamentali: i verbi di un modello UML;
 - una sola entità di raggruppamento, il *package*: usato per raggruppare entità semanticamente correlate;
 - una sola entità informativa, l'annotazione, del tutto simile al comune giallo adesivo;
 - le relazioni collegano le entità tra loro;
 - i diagrammi mostrano le viste di interesse di un modello.
6. L'UML ha quattro meccanismi comuni:
 - le specifiche sono frasi testuali che descrivono le caratteristiche e la semantica degli elementi modellati: il vero contenuto del modello;
 - gli ornamenti sono informazioni aggiunte a un elemento modellato in un diagramma per illustrare un qualche concetto supplementare;
 - le distinzioni comuni:
 - classificatore e istanza: classificatore, il concetto astratto di tipo di entità (per esempio un conto bancario); istanza, una specifica istanza di un tipo di entità (per esempio il mio conto bancario);
 - interfaccia e implementazione: interfaccia è un contratto che specifica il comportamento di un'entità; implementazione specifica invece i dettagli di come tale entità esegue il comportamento;
 - meccanismi di estendibilità:
 - vincoli: consentono di aggiungere nuove regole agli elementi modellati;
 - stereotipi: introducono nuovi elementi di modellazione basandosi su quelli esistenti;
 - valori etichettati: permettono di aggiungere proprietà agli elementi modellati. Un valore etichettato è composto da un etichetta identificativa associata a un valore.

7. L'UML definisce l'architettura di sistema basandosi su 4+1 viste:

- vista logica: terminologia e funzionalità del sistema;
- vista dei processi: efficienza, scalabilità e capacità del sistema;
- vista di implementazione: assemblaggio del sistema e gestione della configurazione;
- vista di *deployment*: topologia, distribuzione, rilascio e installazione del sistema;
- vista dei casi d'uso: integra le prime quattro viste, descrivendo i requisiti delle parti interessate.

2.1 Contenuto del capitolo

Lo scopo di questo capitolo è di fornire una panoramica molto sintetica dell'UP (Unified Process). Chi ha poca domestichezza con l'UP dovrebbe cominciare leggendo della sua storia. Chi già la conosce, può saltare direttamente al Paragrafo 2.4, dove si discute dell'UP e del Rational Unified Process (RUP) oppure al Paragrafo 2.5, dove si spiega come applicare l'UP a progetti reali.

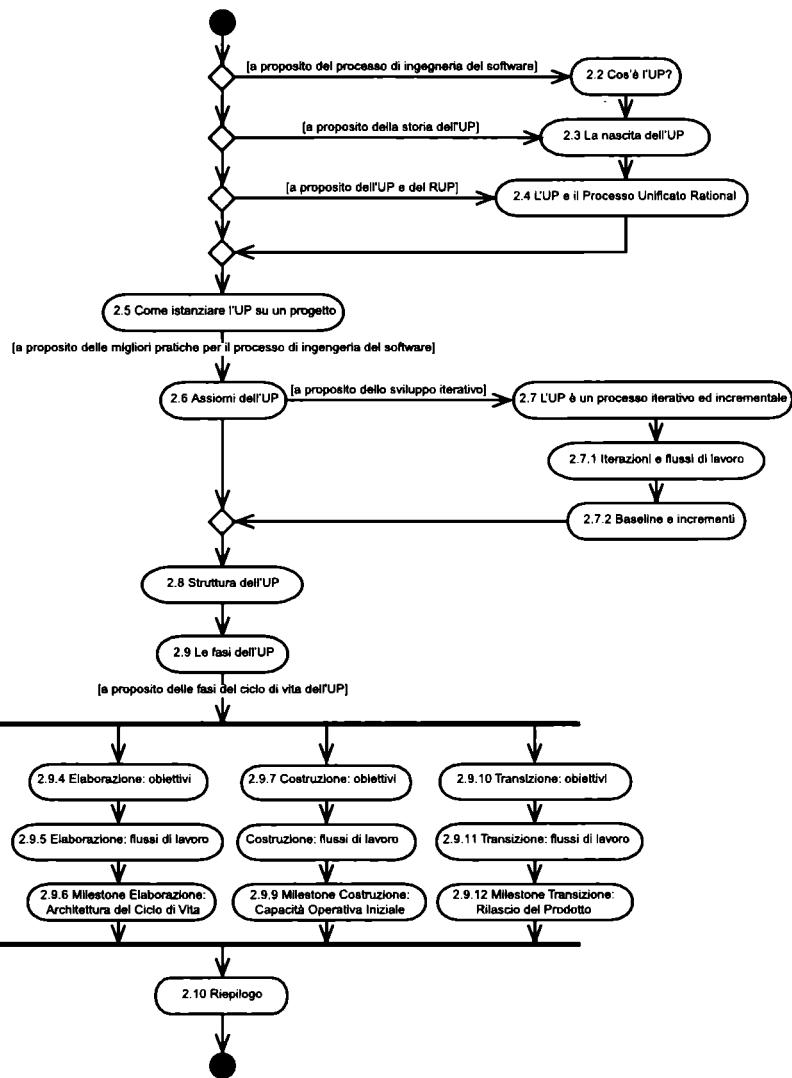
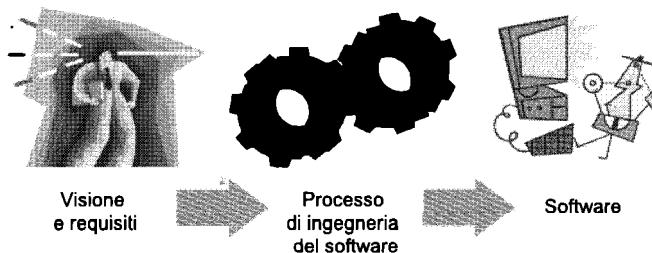
Per quanto riguarda questo libro, per l'UP è limitato alla sua capacità di fornire un *framework* di processo adatto alla presentazione delle tecniche di analisi e progettazione OO. Chi volesse approfondire, può trovare una guida completa all'UP in [Jacobson 1] e una presentazione eccellente del suo analogo, il RUP, in [Kruchten 1] e [Ambler 1], [Ambler 2] e [Ambler 3].

2.2 L'UP

Un processo di sviluppo del software (Software Development Process, SDP), detto anche processo di ingegneria del software (Software Engineering Process, SEP) definisce il *chi*, il *cosa*, il *quando* e il *come* dello sviluppo del software. Nella Figura 2.2 viene illustrato come un SEP sia il processo con cui si trasformano le richieste dell'utente in software.

USDP (Unified Software Development Process, Processo Unificato per lo Sviluppo del Software) è il SEP standard dell'industria del software ideato dagli stessi autori dell'UML e viene comunemente chiamato Unified Process [Jacobson 1]. Nel resto di questo libro useremo l'acronimo UP.

In origine il progetto UML doveva fornire sia un linguaggio visuale, sia un processo di ingegneria del software. Ciò che oggi conosciamo come UML rappresenta solo la parte relativa al linguaggio visuale; l'UP è, invece, la parte che concerne il processo.

**Figura 2.1****Figura 2.2**

L'UP si basa sul lavoro relativo ai processi svolto dalla Ericsson (Ericsson approach, approccio della Ericsson) nel 1967, dalla Rational (Ration Objectory Process, Proceso Rational Objectory) tra il 1996 e il 1997, e su altre esperienze. Per questo motivo l'UP è un metodo per lo sviluppo del software, pragmatico e maturo, che incorpora le migliori pratiche messe a punto dai suoi predecessori.

2.3 La nascita dell'UP

Studiando la storia dell'UP, illustrata nella Figura 2.3, sembra giusto sostenere che il suo sviluppo è principalmente legato alla carriera di un uomo, Ivar Jacobson. In effetti molti considerano Jacobson il padre dell'UP. Con questa affermazione non si vuole minimizzare il lavoro di tutti gli altri individui (specialmente di Booch) che hanno partecipato allo sviluppo dell'UP. Piuttosto si vuole evidenziare il contributo insostituibile di Jacobson.

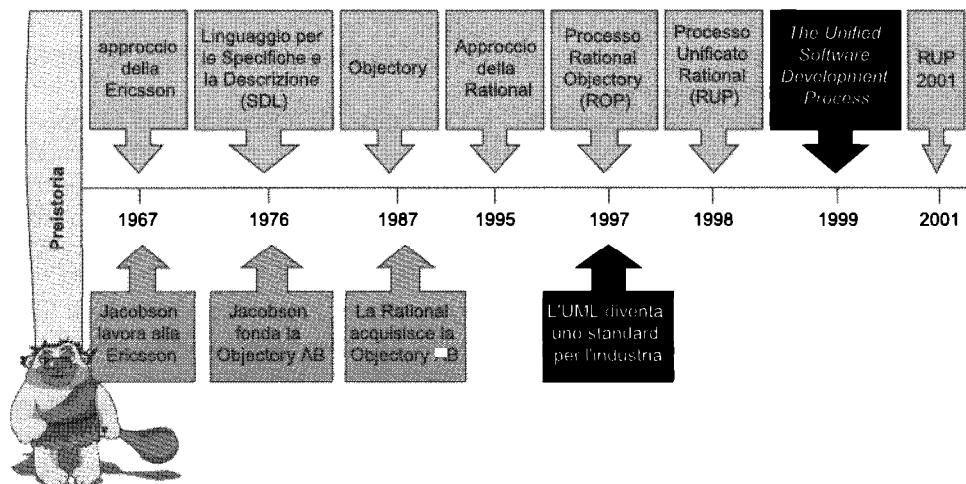


Figura 2.3

L'UP ha le sue origini nel lontano 1967, con l'approccio della Ericsson, che intraprese la strada radicale di modellare un sistema complesso con un insieme di blocchi interconnessi. Blocchi più piccoli venivano collegati tra loro a formare blocchi sempre più grandi i quali, infine, costituivano e descrivevano un sistema completo. Questa tecnica, oggi conosciuta come sviluppo basato sui componenti (Component-Based Development, CBD), ha le sue radici nel concetto di “dividi e conquista”.

Un sistema completo può risultare incomprensibile e inafferrabile per un individuo che lo analizza come se fosse un unico oggetto monolitico. Quando, però, viene diviso e suddiviso in componenti più piccoli è possibile assimilarlo, comprendendendo i servizi che ciascun componente offre (l'interfaccia del componente) e come questi componenti siano tra loro interconnessi. Usando la terminologia dell'UML, i blocchi più grandi sono dei “sottosistemi”, ciascuno dei quali è implementato tramite blocchi più piccoli chiamati “componenti”.

Un'altra innovazione introdotta dalla Ericsson era il modo in cui identificava questi blocchi, creando dei “casi di traffico” che descrivevano come il sistema doveva essere usato. Questi “casi di traffico” si sono evoluti in ciò che oggi, nell’UML, vengono chiamati “casi d’uso”. Il risultato di questo processo era un’architettura che descriveva tutti i blocchi e come l’insieme di questi formasse il sistema. Questo strumento è stato il precursore del modello statico dell’UML.

Oltre alla vista dei requisiti (i “casi di traffico”) e la vista statica (la descrizione dell’architettura), la Ericsson utilizzava anche una vista dinamica che descriveva come tutti i blocchi comunicassero tra loro nel tempo. Questa vista era costituita da diagrammi di sequenza, di collaborazione e di stato, tuttora presenti nell’UML, anche se in forma molto più evoluta.

La successiva più importante innovazione dell’ingegneria del software OO fu quando il CCITT, ente internazionale per gli standard, rilasciò il Linguaggio per le Specifiche e la Descrizione (Specification and Description Language, SDL). Questo linguaggio era stato progettato per fissare il comportamento dei sistemi di telecomunicazione. Tali sistemi venivano modellati come un insieme di componenti che comunicavano scambiandosi segnali. In effetti, l’SDL fu il primo standard per la modellazione a oggetti. Viene utilizzato ancora oggi.

Nel 1987, Jacobson fondò a Stoccolma la Objectory AB. Questa società sviluppò e mise sul mercato un processo per l’ingegneria del software, basato sull’approccio della Ericsson, che si chiamava Objectory (da *Object Factory*: Fabbrica degli Oggetti). Il SEP Objectory era costituito da un insieme di documentazione standard, uno strumento CASE di non immediato utilizzo e un servizio di consulenza della Objectory AB, di cui probabilmente era difficile fare a meno.

Durante questo periodo, l’innovazione più importante fu probabilmente che il SEP Objectory era considerato esso stesso un sistema a tutti gli effetti. Le fasi e i flussi che costituiscono il processo (requisiti, analisi, progettazione, implementazione, test) venivano espressi tramite un insieme di diagrammi. In altre parole, il processo Objectory veniva modellato e sviluppato allo stesso modo di un sistema software. Questo rese possibile i futuri sviluppi del processo. Objectory, come l’UP, era un *framework* di processo e richiedeva una personalizzazione pesante prima di poter essere applicato a un qualunque specifico progetto. Il processo Objectory veniva distribuito con alcuni modelli standard da applicare ai diversi tipi di progetti di sviluppo software, ma era quasi sempre necessario personalizzare ulteriormente il processo. Jacobson si rese conto che tutti i progetti di sviluppo software sono diversi e che non era né ipotizzabile, né auspicabile riuscire a mettere a punto un SEP “per tutti gli usi”.

Quando la Rational acquisi la Objectory AB nel 1995, Jacobson iniziò a dedicarsi all’unificazione del processo Objectory con tutto il lavoro riguardante i processi che era stato svolto in Rational, che non era poco. Sviluppò così una vista “4+1” dell’architettura, la quale si basava, appunto, su quattro viste distinte (logica, di processo, fisica e di sviluppo) più una vista dei casi d’uso che legava le altre insieme. Questa idea si trova ancora oggi alla base dell’approccio che l’UP e l’UML hanno nei confronti dell’architettura del sistema. Venne, inoltre, formalizzato il concetto di sviluppo iterativo, basato su una sequenza di fasi (avvio, elaborazione, costruzione e transizione), che combina la disciplina del ciclo di vita a “cascata” con la dinamicità e la capacità di risposta dello sviluppo iterativo incrementale. I personaggi che hanno maggiormente

contribuito a questo lavoro sono stati: Walker Royce, Rich Reitmann, Grady Booch (ideatore del metodo Booch) e Philippe Kruchten. In particolare, nell'Approccio Rational fu incorporata gran parte dell'esperienza di Booch e delle sue idee innovative concernenti l'architettura (un ottimo testo per eventualmente approfondire queste idee è [Booch 1]).

Il ROP (Rational Objectory Process, Processo Rational Objectory) fu il risultato dell'unificazione di Objectory e del lavoro sui processi svolto dalla Rational. Il ROP in particolare apportò dei miglioramenti laddove Objectory era più debole: requisiti diversi dai casi d'uso, implementazione, test, gestione del progetto, *deployment*, gestione della configurazione e ambiente di sviluppo. Fu introdotto il concetto di rischio come elemento che pilota il processo, fu definita l'architettura e si formalizzò la sua qualità di artefatto consegnabile chiamato "descrizione dell'architettura". Durante questo periodo, in Rational, Booch, Jacobson, e Rumbaugh stavano sviluppando l'UML. Questo divenne il linguaggio utilizzato per esprimere sia i modelli ROP, sia il ROP stesso.

A partire dal 1997 la Rational acquisì molte altre società e con esse anche ulteriori esperti di analisi dei requisiti, gestione della configurazione, test e altro. Fu così che si giunse, nel 1998, al rilascio del Rational Unified Process (RUP)¹. Da allora si sono susseguite diverse nuove versioni del RUP, ciascuna delle quali ha apportato notevoli migliorie.

Nel 1999 è stato pubblicato un testo importante: *The Unified Software Development Process* [Jacobson 1], che descrive l'UP. Mentre il RUP è un prodotto per processi della Rational, l'UP è un SEP non proprietario creato dagli autori dell'UML. Non sorprende certo che l'UP e il RUP siano parenti molto stretti. Per questo libro, abbiamo deciso di occuparci dell'UP e non del RUP, in quanto il primo è un SEP non proprietario, accessibile a tutti, non associato in particolare ad alcun prodotto o società.

2.4 L'UP e il RUP

Esistono sul mercato diverse varianti commerciali dell'UP. Se si ragiona in termini UML, l'UP definisce una classe di processi di sviluppo del software, e queste sue varianti commerciali sono assimilabili a delle sottoclassi dell'UP. In altre parole queste varianti commerciali ereditano tutte le caratteristiche dell'UP, ne ridefiniscono alcune, aggiungendone altre.

La variante commerciale più diffusa è il RUP. Questo prodotto mette a disposizione tutti gli standard, gli strumenti e altre cose che non sono direttamente inclusi nell'UP, ma che sarebbe, comunque, necessario procurarsi per poter applicare il processo. Il RUP mette anche a disposizione via web un ricco ambiente di supporto che include molta documentazione del processo e dei *tool mentor* (esperti di strumento) per ciascuno degli strumenti Rational.

Anche il RUP deve essere istanziato per ciascun progetto, ma il lavoro necessario per istanziarlo è minore di quello richiesto partendo con il solo UP. In effetti, nessuna delle sottoclassi di UP esistenti può considerarsi completo e applicabile così com'è, da subito. Per adattare questi processi alle esigenze di una società o di un progetto, è necessario prevedere un certo investimento di tempo e di soldi per pagare le consulenze dirette da parte del *vendor* del SEP.

¹ Vedere www.rational.com e [Kruchten 1].

Nel non lontano 1999 il RUP veniva generalmente considerato come una semplice implementazione commerciale dell'UP. Tuttavia, da allora il RUP ha fatto molta strada e oggi estende l'UP significativamente in molti modi. Oggi l'UP deve essere considerato come una classe base generica non proprietaria, di cui il RUP è una sottoclasse commerciale specifica che estende e ridefinisce alcune delle caratteristiche dell'UP. Il RUP e l'UP restano, comunque, fondamentalmente più simili che diversi. La differenza principale riguarda la completezza e il livello di dettaglio e non le questioni semantiche o ideologiche. I flussi di lavoro di base per l'analisi e la progettazione OO sono sufficientemente simili e una discussione impostata dal punto di vista dell'UP è del tutto valida anche per chi già conosce e utilizza il RUP. Abbiamo scelto di usare l'UP in modo che questo testo potesse essere utile alla maggioranza degli analisti e dei progettisti OO, i quali non utilizzano il RUP, ma sapendo anche che può, comunque, essere molto utile per la minoranza significativa che invece utilizza il RUP.

L'UP e il RUP modellano entrambi il *chi*, il *quando* e il *cosa* del processo di sviluppo del software, ma lo fanno in modo leggermente diverso. L'ultima versione del RUP (RUP 2001) presenta alcune differenze di termini e di sintassi rispetto all'UP, anche se mantiene essenzialmente la stessa semantica degli elementi del processo. La Figura 2.4 illustra come le icone di processo di RUP 2001 corrispondano alle icone UP che si utilizzano in questo libro. Si osservi l'uso della relazione «origine» tra l'icona RUP 2001 e l'icona originale dell'UP. In UML la relazione «origine» è una dipendenza particolare tra elementi di modellazione che indica che l'elemento iniziale della relazione è uno sviluppo storico dell'elemento che si trova alla fine della relazione. Questo si adatta perfettamente agli elementi di modellazione del RUP e dell'UP.

Per modellare il *chi* del SEP, l'UP introduce il concetto di risorsa. Questo concetto descrive il ruolo che un individuo o un gruppo ha all'interno di un progetto. Nel RUP le risorse vengono chiamate ruoli, ma la semantica resta invariata.

UP	RUP	Semantica
		<i>Chi</i> – Il ruolo che un individuo o un gruppo riveste nel progetto
		<i>Cosa</i> – l'unità di lavoro eseguita da una risorsa (ruolo)
		<i>Quando</i> – una sequenza di attività correlate che apporta valore al progetto

Figura 2.4

L'UP modella il *quando* usando le attività. Queste verranno eseguite dagli individui o dai gruppi che partecipano al progetto. Questi ultimi *assumeranno un ruolo specifico* per eseguire ciascuna attività. L'UP (e il RUP) indica, dunque, le risorse (ruoli) che partecipano a ciascuna singola attività. Quando è necessario ottenere un maggiore livello di dettaglio, è possibile scomporre le attività in sottoattività più piccole. Sia l'UP, sia il RUP definiscono il flusso di lavoro come una sequenza di attività correlate.

Il RUP e l'UP descrivono il *cosa* di un SEP come artefatti. Gli artefatti sono materiali, di qualunque natura, richiesto o prodotto dal progetto: codice sorgente, eseguibili, standard, documentazione ecc. Si possono utilizzare icone diverse per rappresentare gli artefatti.

2.5 Come istanziare l'UP su un progetto

L'UP è un processo di sviluppo del software generico e deve essere istanziato per ciascuna organizzazione e anche per ciascun specifico progetto. Questo è necessario perché tutti i progetti software sono tra loro differenti. Semplicemente non esiste un SEP “per tutte le misure”. Il processo di istanziazione consiste nella definizione e integrazione di:

- standard per il gruppo di lavoro;
- modelli di documento standard;
- strumenti: compilatori, gestione della configurazione ecc.;
- archiviazione: gestione dei bug, gestione del progetto ecc.;
- modifiche del ciclo di vita: per esempio, strumenti più sofisticati per il controllo del livello di qualità per i sistemi in cui la sicurezza è critica.

Non rientra tra gli scopi di questo libro descrivere in dettaglio il processo di personalizzazione e adattamento dell'UP (o del RUP). Questo aspetto può essere approfondito in [Rumbaugh 1].

2.6 Assiomi dell'UP

L'UP è basato su tre assiomi fondamentali:

- è pilotato dai casi d'uso e dal fattore di rischio;
- è incentrato sull'architettura;
- è iterativo e incrementale.

I casi d'uso saranno trattati in modo approfondito nel Capitolo 4. Per ora ci si limita a dire che sono un modo per fissare i requisiti di un sistema. È, quindi, corretto dire che l'UP è pilotato dai requisiti.

L'altro fattore che pilota l'UP è il rischio. L'idea di fondo è che se non vengono affrontati attivamente tutti i rischi individuati in un progetto, questi possono "agredire" il progetto e consumarlo da dentro! Chiunque abbia lavorato in un progetto di sviluppo software sarà sicuramente d'accordo con la precedente affermazione. L'UP affronta questo punto predicando una costruzione del software basata sull'analisi del rischio. In realtà, la gestione del rischio resta comunque un compito da capoprogetto e da architetto e non verrà quindi trattato in questo libro.

L'UP prevede che lo sviluppo di un sistema software richieda la messa a punto di una robusta architettura. L'architettura descrive gli aspetti strategici di un sistema, come questo sia scomposto in componenti, e come tali componenti interagiscano tra loro e debbano essere dislocati sui diversi nodi hardware. È, dunque, ovvio che un'architettura di sistema di buona qualità potrà consentire lo sviluppo di un sistema di buona qualità e non, invece, di un insieme poco coeso di codice sorgente messo insieme con poca premeditazione.

Infine l'UP è iterativo e incrementale. L'UP prevede, infatti, che il progetto venga scomposto in sottoprogetti (o iterazioni) ciascuno dei quali porterà al rilascio di qualche nuova funzionalità, giungendo, dunque, al completamento del sistema in modo incrementale. In altre parole il software viene costruito con un processo a passi successivi, ciascuno dei quali ci avvicina di più all'obiettivo finale del progetto. Questo approccio è profondamente diverso dal vecchio ciclo di vita a cascata, il quale prevede che le fasi di analisi, progettazione e costruzione si susseguano in modo piuttosto rigido. Sviluppando un progetto con l'UP si ritorna invece diverse volte, in tempi diversi, sui flussi di lavoro principali, quali, per esempio, quello dell'analisi.

2.7 L'UP è un processo iterativo e incrementale

Per comprendere l'UP, è necessario comprendere come funzionino le iterazioni. L'idea è fondamentalmente molto semplice: la storia dimostra che generalmente gli uomini riescono ad affrontare e risolvere i problemi più piccoli meglio di quelli più complessi. Sembra, quindi, naturale scomporre un progetto di sviluppo software in un certo numero di "miniprogetti" più piccoli, ma più facili da gestire e completare con successo. Ciascun "miniprogetto" diventa un'iterazione. Il concetto chiave è che *ciascuna* iterazione contiene tutti gli elementi di un tipico progetto di sviluppo software:

- pianificazione;
- analisi e progettazione;
- costruzione;
- integrazione e test;
- un rilascio (esterno o interno).

Ogni iterazione porta al raggiungimento di una *baseline*, la quale comprende una versione *parzialmente completa* del sistema finale, oltre alla relativa documentazione di progetto.

Le *baseline* si accumulano una sull'altra man mano che le iterazioni si susseguono, fino a ottenere il sistema finale completo.

La differenza tra due *baseline* consecutive viene chiamata incremento: questo è il motivo per cui l'UP è conosciuto come un ciclo di vita iterativo e incrementale.

Nel Paragrafo 2.8 si vedrà che le iterazioni sono raggruppate in fasi. Le fasi forniscono la macrostruttura dell'UP.

2.7.1 Iterazioni e flussi di lavoro

Esistono cinque fondamentali flussi di lavoro che specificano quali attività debbano essere svolte in ciascuna iterazione e le *skill* richieste. Oltre a questi cinque, esistono anche altri flussi di lavoro, quali quello della pianificazione e quello della valutazione degli assetti, che possono però non essere applicabili a tutte le iterazioni. Tuttavia l'UP tratta esclusivamente dei cinque flussi di lavoro fondamentali:

- requisiti: fissa ciò che il sistema dovrebbe fare;
- analisi: mette a punto i requisiti e aggiunge loro struttura;
- progettazione: concretizza i requisiti in un'architettura del sistema;
- implementazione: costruisce il software;
- test: verifica che l'implementazione funzioni come da requisiti.

La Figura 2.5 illustra alcuni possibili flussi di lavoro per un'iterazione. Più avanti nel libro si approfondiranno i flussi di lavoro dei requisiti, dell'analisi, della progettazione e dell'implementazione (il flusso di lavoro dei test non rientra tra gli obiettivi di questo libro).

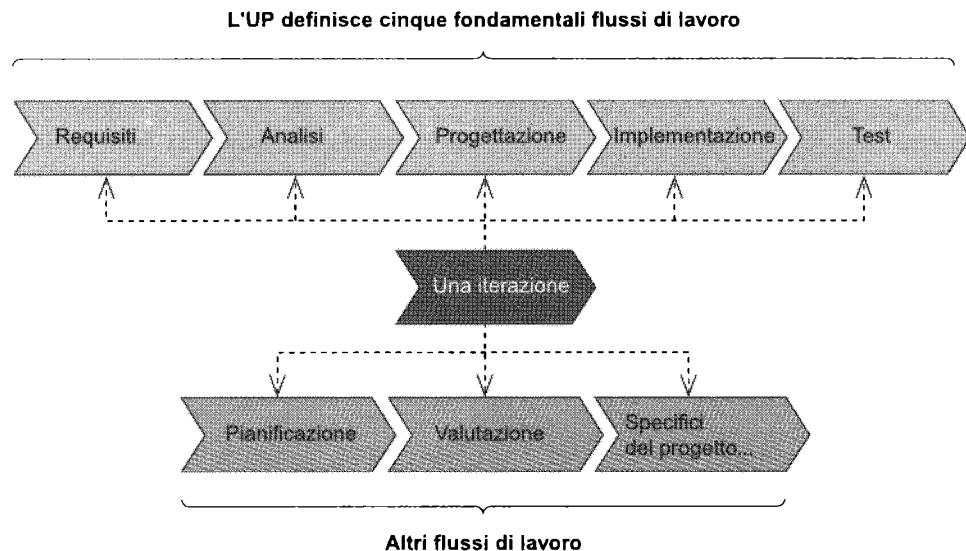


Figura 2.5

Anche se ciascuna iterazione può prevedere tutti e cinque i flussi di lavoro, la collocazione dell’iterazione all’interno del ciclo di vita del progetto determina una maggiore enfasi su uno dei flussi di lavoro.

La scomposizione del progetto in una serie di iterazioni consente di gestire la pianificazione di progetto in modo più flessibile. L’approccio più semplice è quello di sistematizzare le iterazioni in sequenza sulla scala temporale, in modo che ciascuna iterazione abbia inizio quando quella precedente è conclusa. Tuttavia è spesso possibile schedulare le iterazioni in parallelo. Questo richiede una comprensione delle dipendenze esistenti tra gli artefatti che interessano ciascuna iterazione, ovvero implica che lo sviluppo software venga basato sullo studio e la modellazione dell’architettura. Le iterazioni parallele possono offrire diversi benefici, tra cui un minore tempo di rilascio e un migliore impiego delle risorse, tuttavia richiedono anche una migliore pianificazione.

2.7.2 **Baseline** e incrementi

Ogni iterazione UP genera una *baseline*. Questo è il rilascio (interno o esterno) dell’insieme di artefatti, previsti e approvati, che sono stati generati da quell’iterazione. Ogni *baseline*:

- fornisce una base approvata per successive attività di analisi e sviluppo;
- può essere modificata *esclusivamente* tramite procedure formali di gestione della configurazione e delle modifiche.

Gli incrementi, tuttavia, sono solo la *differenza* tra una *baseline* e quella successiva. Costituiscono un passo avanti verso il rilascio ultimo del sistema completo.

2.8 Struttura dell’UP

La Figura 2.6 illustra la struttura dell’UP. Il ciclo di vita del progetto è suddiviso in quattro fasi: Avvio, Elaborazione, Costruzione e Transizione; ciascuna delle quali si conclude con un’importante *milestone*. Ogni fase può essere composta da una o più iterazioni. In ogni iterazione vengono eseguiti i cinque flussi di lavoro fondamentali, oltre a eventuali flussi di lavoro aggiuntivi. Il numero esatto di iterazioni per fase dipende dalle dimensioni del progetto; le singole iterazioni non dovrebbero, comunque, durare più di due o tre mesi. L’esempio può essere considerato tipico per un progetto di medie dimensioni della durata complessiva di circa 18 mesi.

Come si può ben vedere in Figura 2.6, l’UP è costituito da una sequenza di quattro fasi, ciascuna delle quali si conclude con un’importante *milestone*:

- Avvio: Obiettivi del Ciclo di Vita;
- Elaborazione: Architettura del Ciclo di Vita;
- Costruzione: Capacità Operativa Iniziale;
- Transizione: Rilascio del Prodotto.

La quantità di lavoro richiesta per ciascuno dei cinque fondamentali flussi di lavoro varia al passaggio del progetto attraverso le diverse fasi dell'UP.

La Figura 2.7 è la chiave per comprendere come funziona l'UP. In alto sono indicate le fasi del processo; sul lato sinistro ci sono i cinque flussi di lavoro fondamentali; in basso sono elencate alcune iterazioni. Le curve rappresentano la quantità di lavoro dedicata a ciascuno dei cinque flussi di lavoro fondamentali man mano che il progetto procede, attraversando le diverse fasi.

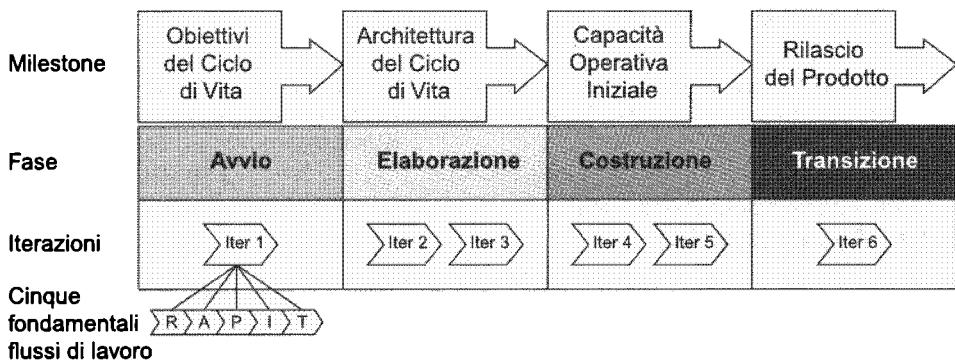


Figura 2.6

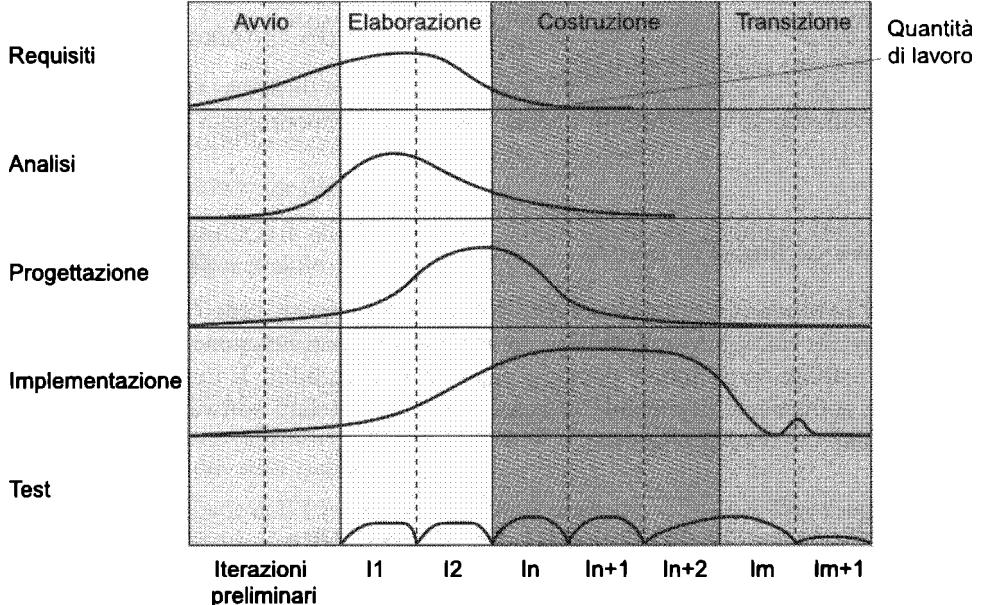


Figura 2.7 Adattata dalla Figura 1.5 [Jacobson 1], con il consenso della Addison-Wesley.

La Figura 2.7 mostra che nella fase Avvio la maggior parte del lavoro viene svolto sul flusso dei requisiti e dell'analisi. Durante l'Elaborazione l'enfasi si sposta sui requisiti e, in parte, sulla progettazione. Durante la Costruzione l'enfasi è chiaramente sulla progettazione e sull'implementazione. Infine, durante la fase di Transizione l'enfasi è decisamente sull'implementazione e sui test.

Il resto del capitolo è dedicato a una panoramica delle quattro fasi dell'UP.

2.9 Le fasi dell'UP

Ciascuna fase ha un obiettivo, si concentra su una o più attività e flussi di lavoro e si conclude con una *milestone*. Questo è lo schema utilizzato per trattare le quattro fasi.

2.9.1 Avvio: obiettivi

L'Avvio ha come obiettivo quello di “far decollare” il progetto. L'Avvio comporta:

- stabilire la fattibilità; questo potrebbe richiedere la messa a punto di un prototipo tecnico o di una prova di fattibilità per validare i requisiti funzionali;
- creare un caso di *business* per dimostrare che il progetto apporterà un beneficio quantificabile dal *business*;
- fissare i requisiti essenziali per definire l'ambito del sistema;
- identificare i rischi più critici.

Le principali risorse coinvolte in questa fase sono il capoprogetto e l'architetto di sistema.

2.9.2 Avvio: flussi di lavoro

L'avvio enfatizza principalmente i flussi di lavoro dei requisiti e dell'analisi. Tuttavia potrebbe anche richiedere attività di progettazione e implementazione nel caso si decidesse di costruire un prototipo tecnico o una prova di fattibilità. Il flusso di lavoro dei test non è solitamente applicabile in quanto gli unici artefatti software prodotti sono prototipi che verranno poi scartati.

2.9.3 Milestone Avvio: Obiettivo del Ciclo di Vita

Mentre altri SEP si concentrano sulla creazione di artefatti, l'UP adotta una diversa strategia basata su obiettivi. Ogni *milestone* definisce degli obiettivi; il raggiungimento di *tutti* tali obiettivi coinciderà con il raggiungimento della *milestone*.

Non tutti gli obiettivi coincidono con la produzione di un artefatto.

La *milestone* dell'Avvio è proprio costituita dalla definizione degli Obiettivi del Ciclo di Vita. La Tabella 2.1 elenca le condizioni che devono essere soddisfatte per raggiungere questa *milestone*.

Tabella 2.1

Condizione da soddisfare	Artefatto
Concordare con le parti interessate gli obiettivi del progetto	Documento di alto livello con i principali requisiti, caratteristiche e vincoli del progetto
Definire l'ambito del progetto concordandolo con le parti interessate	Modello dei casi d'uso iniziale (completo al 10-20%)
Fissare i principali requisiti concordandoli con le parti interessate	Glossario di progetto
Concordare con le parti interessate le stime dei costi e dei tempi	Prima pianificazione del progetto
Il capoprogetto ha creato un caso di <i>business</i>	Caso di <i>business</i>
Il capoprogetto ha effettuato una valutazione dei rischi	Documento o archivio di valutazione dei rischi
La fattibilità è stata confermata da uno o più studi tecnici e/o prototipi	Uno o più prototipi (da scartare)
È stato individuato uno schema di architettura	Documento iniziale di architettura

2.9.4 Elaborazione: obiettivi

L'Elaborazione ha i seguenti obiettivi:

- creare una *baseline* eseguibile;
- rifinire la Valutazione dei Rischi;
- definire gli attributi di qualità (velocità di individuazione dei difetti, densità massima di difetti accettabile ecc);
- fissare i casi d'uso relativi a circa l'80% dei requisiti funzionali (si vedrà esattamente cosa comporta questo obiettivo nei Capitoli 3 e 4);
- creare un piano dettagliato della fase di costruzione;
- formulare una valutazione iniziale delle risorse, del tempo, degli strumenti, del personale e dei costi richiesti.

L'obiettivo principale dell'Elaborazione è quello di produrre una *baseline* eseguibile. Si tratta di un vero e proprio sistema eseguibile costruito seguendo l'architettura specificata. La *baseline* eseguibile non è un prototipo (e non verrà quindi scartato), ma piuttosto una prima approssimazione del sistema desiderato. Questa *baseline* eseguibile costituisce le fondamenta su cui verranno progressivamente aggiunte le altre funzioni e che,

durante le fasi di Costruzione e di Transizione, si evolverà nel sistema finale completo che verrà rilasciato. Dato che le successive fasi si basano sui risultati dell'Elaborazione, quest'ultima è senza dubbio la fase più critica di tutto il processo. In effetti gran parte di questo libro è dedicata proprio alle attività di Elaborazione.

2.9.5 Elaborazione: flussi di lavoro

La fase di Elaborazione prevede i seguenti scopi per ciascuno dei cinque fondamentali flussi di lavoro:

- requisiti: fissare il grosso dei requisiti e definire meglio l'ambito del sistema;
- analisi: definire ciò che deve essere costruito;
- progettazione: creare un'architettura stabile;
- implementazione: costruire una *baseline eseguibile*;
- test: testare la *baseline eseguibile*.

L'Elaborazione si concentra, ovviamente, sui flussi di lavoro dei requisiti, dell'analisi e della progettazione, anche se l'implementazione comincia a assumere un maggior peso verso la fine della fase, quando si produce la *baseline eseguibile*.

2.9.6 Milestone Elaborazione: Architettura del Ciclo di Vita

La *milestone* è l'Architettura del Ciclo di Vita. La Tabella 2.2 elenca le condizioni che devono essere soddisfatte per raggiungere questa milestone.

Tabella 2.2

Condizione da soddisfare	Artefatto
Creare una <i>baseline</i> robusta ed eseguibile	La <i>baseline</i> eseguibile
La <i>baseline</i> eseguibile dimostra che i rischi principali sono stati identificati e risolti	Modello statico UML; modello dinamico UML; modello dei casi d'uso UML
La visione d'insieme del sistema si è stabilizzata	Un documento descrittivo del sistema
Rivedere la valutazione dei rischi	Valutazione dei Rischi aggiornata
Rivedere e riconcordare il caso di <i>business</i> con le parti interessate	Caso di <i>business</i> aggiornato
La pianificazione di progetto ha raggiunto un dettaglio tale da consentire la formulazione di una stima realistica dei tempi, dei costi e delle risorse richieste dalle fasi successive	Pianificazione di progetto aggiornata
Le parti interessate hanno approvato la pianificazione di progetto	Caso di <i>business</i>
Il caso di <i>business</i> è stato verificato e confrontato con la pianificazione di progetto	Pianificazione di progetto
È stato raggiunto un accordo con le parti interessate per continuare il progetto	Accordo firmato

2.9.7 Costruzione: obiettivi

L'obiettivo della Costruzione è quello di completare tutti i requisiti, l'analisi e la progettazione e di far evolvere la *baseline* eseguibile generata dall'Elaborazione nel sistema completo finale. È essenziale che durante la Costruzione si *mantenga l'integrità dell'architettura del sistema*. Quando sul progetto inizia a sentirsi la pressione del futuro rilascio e si giunge al culmine del lavoro di implementazione del codice sorgente, capita spesso che si cominci anche a prendere qualche scorciatoia. In questo modo si può presto perdere di vista l'architettura prevista per il sistema, col rischio di produrre un sistema finale di qualità scadente, con elevati costi di manutenzione. È ovviamente consigliabile evitare che un progetto finisca in questo modo.

2.9.8 Costruzione: flussi di lavoro

L'enfasi in questa fase si sposta decisamente sul flusso di lavoro dell'implementazione. Gli altri flussi di lavoro prevedono il minimo di attività necessarie per finire di fissare i requisiti e completare l'analisi e la progettazione. Anche i test diventano prioritari: dato che ogni nuovo incremento costruisce su quello precedente, diventa necessario effettuare sia gli unit test che i test di integrazione. Possiamo sintetizzare il tipo di attività previsto per ciascun flusso di lavoro nel modo seguente:

- requisiti: portare alla luce tutti i requisiti mancanti;
- analisi: ultimare il modello dell'analisi;
- progettazione: ultimare il modello della progettazione;
- implementazione: costruire la Capacità Operativa Iniziale;
- test: testare la Capacità Operativa Iniziale.

2.9.9 Milestone Costruzione: Capacità Operativa Iniziale

In pratica questa *milestone* è molto semplice: il sistema software deve essere finito e predisposto in modo che gli utenti possano eseguire il beta test. La Tabella 2.3 elenca le condizioni che devono essere soddisfatte per raggiungere questa *milestone*.

Tabella 2.3

Condizione da soddisfare	Artefatto
Il software ha una stabilità e qualità giudicata sufficiente per essere rilasciato agli utenti	Prodotto software; modello UML; suite di test
Le parti interessate hanno accettato che il software venga rilasciato agli utenti e sono pronti per la transizione	Manuale utente; descrizione del rilascio
I costi reali confrontati con i costi previsti risultano accettabili	Pianificazione del progetto

2.9.10 Transizione: obiettivi

La fase di Transizione inizia quando il beta test è ultimato e il sistema deve essere finalmente rilasciato agli utenti. Comporta, quindi, la sistemazione dei difetti scoperti durante il beta test, e la preparazione del rilascio a tutti gli utenti. Gli obiettivi di questa fase possono essere sintetizzati in:

- correggere i difetti;
- preparare i siti utente per il nuovo software;
- adattare il software in modo da operare correttamente presso i siti utente;
- modificare il software qualora insorgessero problemi non previsti;
- creare i manuali utente ed eventuale altra documentazione;
- fornire consulenza agli utenti;
- eseguire un riepilogo di fine progetto.

2.9.11 Transizione: flussi di lavoro

L'enfasi in questa fase resta sui flussi di lavoro dell'implementazione e dei test. Si esegue un minimo di progettazione per correggere eventuali errori di progettazione scoperti durante il beta test. A questo punto del ciclo di vita del progetto è auspicabile che non ci sia quasi nulla da fare sul flussi di lavoro dei requisiti e dell'analisi. Se non è così, vuol dire che il progetto è a rischio.

- Requisiti: non applicabile.
- Analisi: non applicabile.
- Progettazione: rivedere il modello se sono insorti problemi durante il beta test.
- Implementazione: adattare il software al rilascio presso i siti utente e correggere i difetti scoperti durante il beta test.
- Test: beta test e test di accettazione eseguito presso i siti utente.

2.9.12 Milestone Transizione: Rilascio del Prodotto

Questa è la *milestone* finale: si conclude il beta test e il test di accettazione, si correggono i difetti e il prodotto finito viene rilasciato e accettato dagli utenti. La Tabella 2.4 elenca le condizioni che devono essere soddisfatte per raggiungere questa *milestone*.

Tabella 2.4

Condizione da soddisfare	Artefatto
Il beta test è ultimato, le necessarie modifiche al software sono state effettuate, gli utenti concordano nell'affermare che il sistema è stato rilasciato con successo	Il prodotto software
Gli utenti stanno attivamente utilizzando il prodotto	
Sono state concordate con gli utenti e implementate le strategie di supporto per il prodotto	Pianificazione del supporto Manuali utente

2.10 Riepilogo

1. Un processo di ingegneria del software (software engineering process, SEP) trasforma i requisiti dell'utente in un software, specificando *chi fa cosa e quando*.
2. L'UP ha le sue origini nel lontano 1967. È un SEP maturo, non proprietario, prodotto dagli autori dell'UML.
3. Il RUP è un'estensione commerciale dell'UP. È del tutto compatibile con l'UP, ma è più completo e ricco di dettagli.
4. L'UP (e il RUP) deve essere istanziato per ciascun specifico progetto tramite l'aggiunta di standard interni e di altri strumenti.
5. L'UP è un SEP moderno che è:
 - pilotato dai casi d'uso (requisiti) e dal fattore di rischio;
 - incentrato sull'architettura;
 - iterativo e incrementale.
6. L'UP prevede la costruzione del software a iterazioni:
 - ogni iterazione è simile a un “miniprogetto” che produce una parte del sistema;
 - ciascuna iterazione aggiunge funzioni e valore ai risultati delle precedenti iterazioni, in modo da arrivare a costruire il sistema completo finale.
7. Ogni iterazione prevede cinque flussi di lavoro fondamentali:
 - requisiti: fissare ciò che il sistema dovrebbe fare;
 - analisi: mettere a punto i requisiti e dar loro struttura;
 - progettazione: concretizzare i requisiti in un'architettura di sistema (definire come il sistema farà quello che deve fare);
 - implementazione: costruire il software;
 - test: verificare che l'implementazione funzioni come da requisiti.

8. L'UP prevede quattro fasi, ciascuna delle quali si conclude con un'importante *milestone*:

- avvio: far decollare il progetto: Obiettivi del Ciclo di Vita;
- elaborazione: far evolvere l'architettura del sistema: Architettura del Ciclo di Vita;
- costruzione: costruire il software: Capacità Operativa Iniziale;
- transizione: rilasciare il software agli utenti: Rilascio del Prodotto.

Parte 2

Requisiti

Il flusso di lavoro dei requisiti

3.1 Contenuto del capitolo

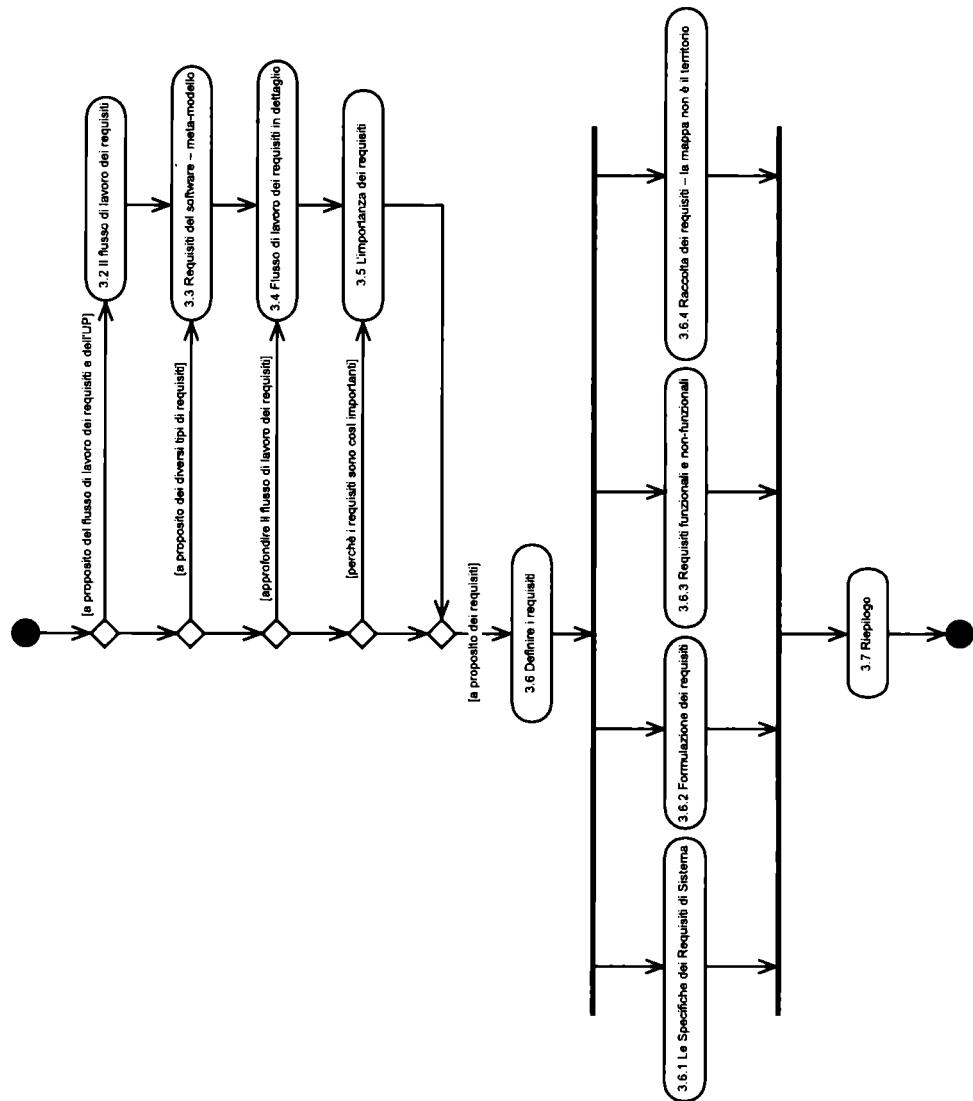
Questo capitolo spiega come comprendere i requisiti di un sistema. Introduce il concetto di requisiti e analizza il flusso di lavoro dei requisiti, così come previsto dall'UP. Presenta, inoltre, un'estensione all'UP che consente di fissare i requisiti *senza utilizzare i casi d'uso UML*.

3.2 Il flusso di lavoro dei requisiti

La Figura 3.2 illustra come la maggior parte delle attività relative al flusso di lavoro dei requisiti siano svolte durante le fasi di Avvio ed Elaborazione, proprio all'inizio del ciclo di vita del progetto. Questo non deve sorprendere, dato che non ha molto senso procedere oltre la fase di Elaborazione se non si sa cosa si vuole costruire!

Prima ancora di iniziare le attività di analisi e progettazione OO, è necessario avere qualche idea su cosa si vuole ottenere, ed è proprio questo lo scopo ultimo del flusso di lavoro dei requisiti: scoprire e accordarsi su ciò che il sistema dovrebbe fare, esprimendolo nel linguaggio degli utenti del sistema. Creare una specifica ad alto livello di ciò che il sistema dovrebbe fare: questa pratica è anche conosciuta come *ingegneria dei requisiti*.

In ogni sistema specifico possono esistere molte parti interessate: diversi tipi di utente, ingegneri della manutenzione, personale per il supporto e l'assistenza, commerciali, dirigenti ecc. L'*ingegneria dei requisiti* è la disciplina che si occupa della raccolta dei requisiti che queste parti interessate hanno per il sistema e dell'assegnazione delle priorità a questi stessi requisiti. Si tratta di un processo di negoziazione, dato che spesso è necessario trovare il giusto compromesso tra requisiti tra loro contrastanti. Per esempio, un gruppo potrebbe avere l'esigenza di rendere il sistema accessibile a molti utenti, il che potrebbe generare un traffico non gestibile dalle infrastrutture di comunicazione o di *database* esistenti. Questo tipo specifico di contrasto tra requisiti è molto comune in questi giorni in cui molte società hanno necessità di rendere parte dei loro sistemi accessibili via internet a una base molto estesa di utenti.

**Figura 3.1**

Molti testi di UML (e anche molti corsi) affermano che i casi d'uso previsti dall'UML sono l'unico strumento valido per fissare i requisiti, tuttavia non è difficile confutare questa affermazione. I casi d'uso in realtà sono in grado di fissare esclusivamente i requisiti funzionali, ovvero quelli che descrivono *cosa debba fare il sistema*. Esiste tuttavia un altro insieme di requisiti, non-funzionali, che descrive *vincoli che il sistema deve rispettare* (prestazioni, affidabilità ecc). Questi requisiti non possono essere espressi correttamente tramite i casi d'uso. In questo libro si presenta, invece, un approccio robusto da "ingegneria dei requisiti", che comprende tecniche potenti e complementari per fissare requisiti di *entrambe* le categorie.

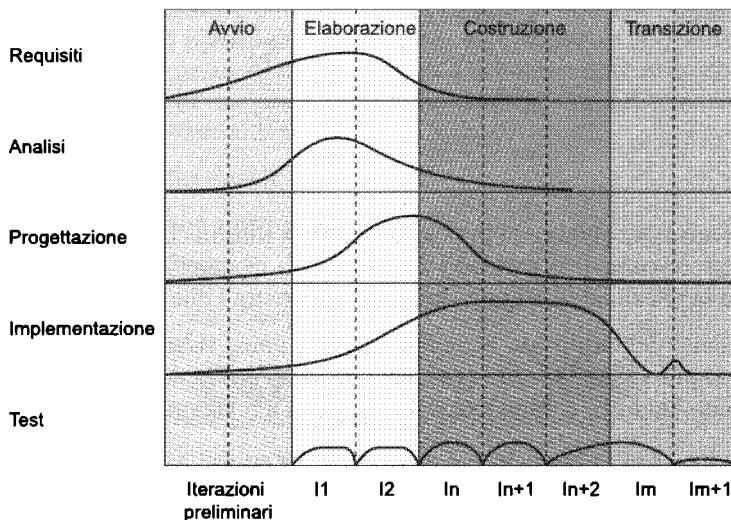
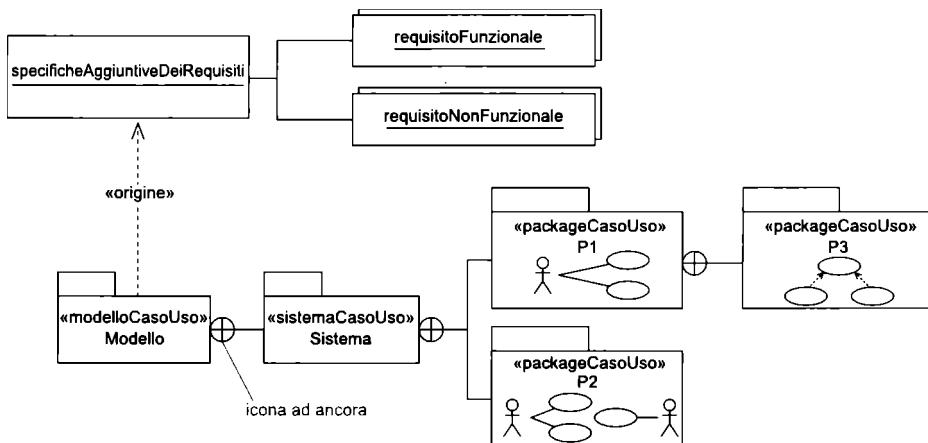


Figura 3.2 Adattata dalla Figura 1.5 [Jacobson 1], con il consenso della Addison-Wesley.

3.3 Requisiti del software: meta-modello

La Figura 3.3 mostra il meta-modello dell'approccio da ingegneria dei requisiti illustrato in questo libro. Contiene diversi elementi di sintassi UML non ancora spiegati. Non c'è bisogno di preoccuparsi! Questi elementi saranno trattati più avanti. Anticipiamo quanto segue.

- Le icone che assomigliano a cartelline sono *package* UML. Sono il meccanismo che l'UML utilizza per raggruppare elementi di modellazione. In effetti, hanno un ruolo simile a quello delle cartelline di un sistema di archiviazione, ovvero consentono di organizzare e raggruppare informazioni o elementi correlati.
- L'icona ad ancora indica che l'elemento che si trova di fianco al cerchio crociato contiene l'elemento che si trova all'altra estremità della linea.
- Il meta-modello mostra che il documento specificheAggiuntiveDeiRequisiti (modellato come un oggetto) contiene molti requisitiFunzionali e molti requisitiNonFunzionali. Come si vedrà più avanti, i nomi degli oggetti sono sempre sottolineati.

**Figura 3.3**

- La freccia tratteggiata con l'etichetta «origine» illustra che il modello dei casi d'uso può essere composto partendo dal contenuto di specificheAggiuntiveDeiRequisiti e indica la relazione di origine storica che esiste tra i due elementi.
- Il modello dei casi d'uso è composto da un unico sistema dei casi d'uso. Questo sistema può essere, invece, composto da molti *package* (ne abbiamo mostrati tre) ciascuno dei quali contiene casi d'uso, attori e relazioni.

Ma allora cosa sono i requisiti, i casi d'uso e gli attori? Nel resto di questo capitolo saranno trattati i requisiti, mentre nel prossimo capitolo i casi d'uso e gli attori.

3.4 Flusso di lavoro dei requisiti in dettaglio

La Figura 3.4 illustra le specifiche attività previste dall'UP per il flusso di lavoro dei requisiti. Questo tipo di diagramma si chiama dettaglio del flusso di lavoro, in quanto elenca tutte le attività che compongono un flusso di lavoro specifico.

Nell'UP i dettagli di flusso di lavoro vengono modellati utilizzando risorse (le icone sul lato sinistro) e attività (le icone a forma di ingranaggio). Le varianti dell'UP, quali il RUP, possono usare icone differenti, ma hanno comunque il medesimo significato (il Paragrafo 2.4 illustra sinteticamente il rapporto che esiste tra l'UP e il RUP). Le frecce sono relazioni che illustrano il flusso di lavoro normale tra un'attività e quella successiva. È utile ricordare che questo tipo di diagramma vuole solo illustrare il flusso di lavoro, così come dovrebbe svolgersi “tipicamente”; potrebbe non essere sempre una rappresentazione esatta di quello che succede veramente. In un progetto reale può succedere che alcune attività vengano effettuate in un ordine diverso da quello illustrato, o magari in parallelo, a seconda delle circostanze.

Dato che questo testo vuole approfondire soprattutto gli aspetti dell'analisi e della progettazione OO, ci si concentrerà sulle attività più importanti per gli analisti e i progettisti OO.

In questo caso le attività che interessano sono le seguenti:

- individuare attori e casi d'uso;
- descrivere un caso d'uso;
- strutturare il modello dei casi d'uso.

Le altre attività del flusso di lavoro dei requisiti non riguardano molto gli analisti/progettisti. “Assegnare priorità ai casi d'uso” è un’attività che concerne più che altro l’architettura e la pianificazione di progetto, mentre “Prototipare l’interfaccia utente” è un’attività di programmazione. Se necessario, è possibile approfondire questi argomenti in [Jacobson 1].

La Figura 3.4 evidenzia come il flusso di lavoro standard dell’UP si concentri sui casi d'uso, escludendo qualunque altra tecnica di raccolta dei requisiti. Questo va bene solo fino a un certo punto perché, come già detto, non consente di gestire in modo ottimale l’aspetto non-funzionale di alcuni requisiti. Per poter gestire in modo completo tutti i requisiti, si estende in modo semplice il flusso di lavoro dei requisiti dell’UP, introducendo le seguenti nuove attività:

- individuare i requisiti funzionali;
- individuare i requisiti non-funzionali;
- assegnare priorità ai requisiti;
- estrarre dai requisiti i casi d'uso.

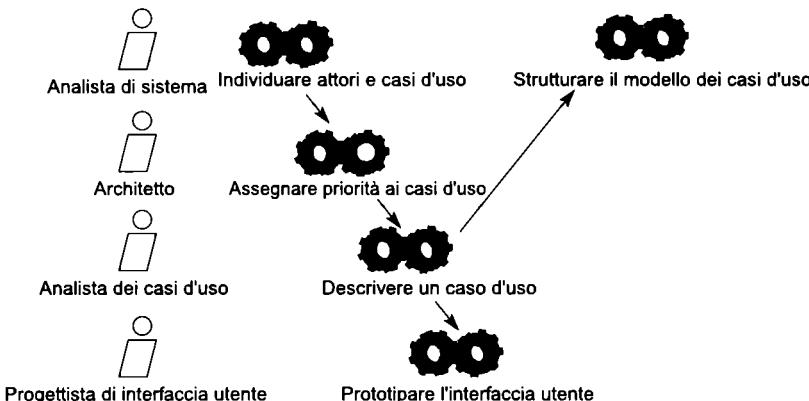


Figura 3.4 Riprodotta da Figura 7.10 [Jacobson 1], con il consenso della Addison-Wesley

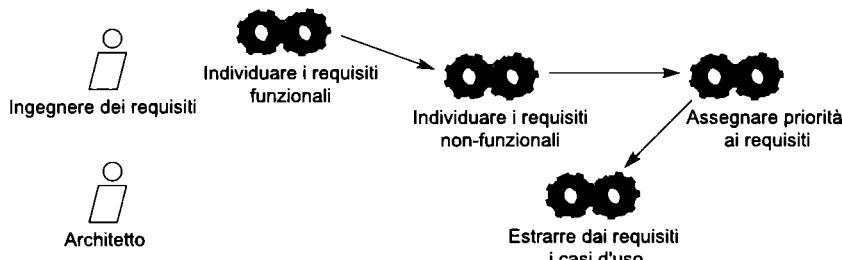


Figura 3.5

È stata, inoltre, introdotta una nuova risorsa: l'ingegnere dei requisiti. La Figura 3.5 illustra le nuove attività e risorse coinvolte.

3.5 L'importanza dei requisiti

“Ingegneria dei requisiti” è il termine utilizzato per descrivere le attività necessarie per raccogliere, documentare e tenere aggiornato l’insieme dei requisiti di un sistema software. Lo scopo è quello di scoprire e capire ciò che il sistema deve fare per soddisfare le diverse parti interessate.

Diversi studi hanno dimostrato che la prima causa di fallimento dei progetti software è il mancato successo delle attività di ingegneria dei requisiti.

La seconda causa di fallimento dei progetti software è la mancanza di coinvolgimento degli utenti. Dato che gli utenti rappresentano la fonte primaria dei requisiti, in realtà si tratta sempre di una carenza che riguarda l’ingegneria dei requisiti!

Visto che il sistema software finale viene costruito sulla base dei requisiti, un’ingegneria dei requisiti efficiente è essenziale e critica per il successo di un progetto di sviluppo software.

3.6 Definire i requisiti

Un requisito può essere definito come “una specifica che dovrebbe essere implementata”. Esistono due tipi base di requisiti:

- requisiti funzionali: il comportamento che il sistema dovrebbe avere;
- requisiti non-funzionali: proprietà o vincoli specifici del sistema.

I requisiti sono (o, almeno, dovrebbero essere) ciò su cui si basano tutti i sistemi. Sono essenzialmente un’affermazione di ciò che il sistema dovrebbe fare. In teoria, i requisiti dovrebbero essere un’affermazione esclusivamente di *cosa* un sistema deve fare, e non di *come* lo deve fare. Si tratta di una precisazione molto importante. È possibile specificare *cosa* debba fare un sistema e quale comportamento debba mostrare senza precisare nulla di *come* debba essere realizzata questa funzionalità.

Anche se in teoria è desiderabile separare il *cosa* dal *come*, in pratica un qualunque insieme di requisiti (noto come Specifiche dei Requisiti di Sistema o, più semplicemente, Specifiche dei Requisiti) ha la tendenza a mischiare i due elementi. In parte questo è dovuto al fatto che la definizione astratta di un problema (il *cosa*) può spesso essere più difficile da scrivere e comprendere della descrizione dell’implementazione (il *come*). In secondo luogo possono effettivamente esistere dei vincoli sull’implementazione del sistema che predeterminano, almeno in parte, il *come*.

Nonostante il fatto che il comportamento di un sistema e, quindi, in ultima analisi, anche la soddisfazione dell’utente finale, dipenda moltissimo dall’ingegneria dei requisiti, molte società, ancora oggi, non la considerano una disciplina importante. Come si è visto, la principale causa di fallimento dei progetti software è dovuta ai requisiti.

3.6.1 Le Specifiche dei Requisiti di Sistema

Molte società producono Specifiche dei Requisiti di Sistema (System Requirements Specifications, SRS) che definiscono ciò che fanno i sistemi software. Queste vengono tipicamente scritte in linguaggio naturale, ma è anche possibile utilizzare con profitto appositi strumenti di ingegneria dei requisiti quali Requisite Pro (Rational Corporation) o DOORS¹.

Le SRS rappresentano il punto di partenza del processo di costruzione del software. Costituiscono spesso l'input iniziale per l'analisi e la progettazione OO. Nel corso degli anni si sono visti molti tipi di SRS; variano da società a società, ma anche da progetto a progetto. Ne esistono di ogni forma e dimensione, ma soprattutto ne esistono di diversi gradi di utilità. In effetti, le domande fondamentali che bisogna chiedersi di una *qualunque* SRS sono: "quanto è utile?" e: "aiuta a capire cosa dovrebbe fare il sistema o no?". Non esistono altri modi di valutare una SRS.

3.6.2 Formulazione dei requisiti

L'UML non fornisce alcuna indicazione su come scrivere una SRS. In effetti, l'UML si occupa dei requisiti esclusivamente tramite il meccanismo dei casi d'uso, che vedremo più avanti. Tuttavia molti modellatori (inclusi noi) credono che i casi d'uso non siano sufficienti, e che sono ancora necessarie le SRS e gli strumenti di gestione dei requisiti.

Per la formulazione dei requisiti, si consiglia l'uso di un formato molto semplice (vedere la Figura 3.6). Ogni requisito ha un identificatore univoco (tipicamente un numero), una parola chiave ("dovrà") e l'affermazione di una funzione. Formulando i requisiti con un formato così rigido e uniforme, si ha il vantaggio che alcuni strumenti di gestione dei requisiti, quali DOORS, sono in grado di interpretare facilmente l'SRS.



Figura 3.6

¹ È possibile reperire informazioni dettagliate sullo strumento di gestione dei requisiti DOORS direttamente sul sito www.telelogic.com.

3.6.3 Requisiti funzionali e non-funzionali

È utile separare i requisiti funzionali da quelli non-funzionali. Esistono molti altri modi di classificare i requisiti, ma, almeno inizialmente per semplificare il più possibile, ci si limiterà a queste due categorie.

Un requisito funzionale specifica cosa farà il sistema: è l'affermazione di una funzione del sistema. Per esempio, raccogliendo i requisiti per uno sportello automatico tipo bancomat, alcuni requisiti funzionali potrebbero essere i seguenti:

- il sistema dovrà controllare la validità della tessera inserita;
- il sistema dovrà validare il codice PIN inserito dal cliente;
- il sistema dovrà erogare non più di € 250 a fronte della stessa carta, in un unico periodo di 24 ore.

Un requisito non-funzionale è un vincolo imposto al sistema. Requisiti non-funzionali per lo stesso sportello automatico potrebbero essere così descritti:

- il sistema dovrà essere scritto in C++;
- il sistema dovrà comunicare con la banca utilizzando un algoritmo di cifratura a 256-bit;
- il sistema dovrà validare la carta entro tre secondi;
- il sistema dovrà validare il PIN entro tre secondi.

Dall'esempio risulta evidente che i requisiti non-funzionali specificano, o limitano, come il sistema debba essere implementato.

3.6.4 Raccolta dei requisiti: la mappa non è il territorio

Ogni volta che si lavora con qualcuno per raccogliere e fissare i requisiti di un sistema software, si cerca di ottenere un'immagine precisa, o mappa, del modello che hanno del mondo. Noam Chomsky, nel suo libro del 1975: *Syntactic Structures* [Chomsky 1] che tratta della grammatica della trasformazione, sostiene che questa mappa viene creata tramite tre processi: rimozione, distorsione e generalizzazione. Questi processi sono necessari, in quanto non abbiamo le capacità cognitive per fissare ogni singolo dettaglio del mondo reale in una mappa mentale altrettanto dettagliata: dobbiamo dunque essere selettivi.

Sopraffatti da una quantità eccessiva di informazioni, si selezionano quelle giudicate importanti applicando tre filtri:

- rimozione: l'informazione viene scartata;
- distorsione: l'informazione viene modificata tramite i correlati meccanismi della creazione e dell'allucinazione;
- generalizzazione: l'informazione viene incorporata in una regola, assioma o principio che riguarda i concetti di verità e falsità.

Questi filtri danno forma al linguaggio naturale. È importante rendersene conto quando si effettua la raccolta o l'analisi di requisiti molto dettagliati, perché potrebbe essere necessario individuarli e metterli in discussione, per riuscire a risalire a informazioni precise.

Seguono alcuni esempi tratti da un sistema di gestione di una biblioteca. Per ciascun esempio si riporta anche come è stato messo in discussione un possibile filtro e una possibile risposta.

- Esempio: “Utilizzano il sistema per prendere in prestito i libri”; rimozione.
 - Messa in discussione: Chi è che utilizza il sistema per prendere in prestito i libri? Tutti gli utenti, o soltanto alcuni?
 - Risposta: Alcuni utenti utilizzano il sistema solo per controllare la disponibilità.
- Esempio: “Non è possibile prendere in prestito un altro libro se non sono stati restituiti tutti i libri il cui periodo di prestito sia già scaduto”; distorsione.
 - Messa in discussione: Non esiste alcuna situazione in cui qualcuno può prendere in prestito un nuovo libro prima di aver restituito tutti i libri in prestito il cui periodo di prestito sia già scaduto?
 - Risposta: Effettivamente esistono due situazioni in cui viene ripristinato il diritto di un utente a prendere nuovi libri in prestito. Quando restituisce tutti i libri il cui periodo di prestito sia scaduto, oppure quando paga per quelli che non ha restituito.
- Esempio: “Tutti devono avere una tessera per prendere i libri in prestito”; generalizzazione.
 - Messa in discussione: Non esiste nessun utente del sistema che possa operare senza essere fornito di tessera?
 - Risposta: Alcuni utenti del sistema, quali per esempio le altre biblioteche, possono non aver bisogno di una tessera, oppure possono aver bisogno di una tessera di tipo speciale soggetta a termini e condizioni diverse.

Gli ultimi due casi sono particolarmente interessanti, in quanto esemplificano uno schema linguistico molto comune: il quantificatore universale. I quantificatori universali sono parole come:

- tutto;
- ogni;
- sempre;
- mai;
- nessuno;
- niente.

Ogni volta che si incontra un quantificatore universale ci si può trovare dinanzi a una rimozione, una distorsione o una generalizzazione. La presenza di queste parole potrebbe indicare che è stato raggiunto il limite, i confini della mappa mentale dell'interlocutore. Per questo motivo, quando si effettua analisi, è spesso una buona idea mettere in discussione i

quantificatori universali. Si stava per scrivere: “è *sempre* una buona idea mettere in discussione i quantificatori universali”, ma poi si sono messe in discussione queste stesse parole!

3.7 Riepilogo

Questo capitolo ha presentato il flusso di lavoro dei requisiti dell’UP e una discussione generica sui requisiti di un sistema software. Si sono imparati i seguenti concetti.

1. L’UP prevede che la maggior parte delle attività che riguardano il flusso di lavoro dei requisiti siano effettuate durante le fasi di Avvio e di Elaborazione del ciclo di vita di un progetto.
2. Il meta-modello dei requisiti (Figura 3.3) mostra che esistono due modi per fissare i requisiti: come requisiti funzionali e non-funzionali oppure come casi d’uso e attori.
3. Nell’UP il dettaglio del flusso di lavoro dei requisiti prevede le seguenti attività che interessano in qualità di analisti e progettisti OO:
 - individuare attori e casi d’uso;
 - descrivere un caso d’uso;
 - strutturare il modello dei casi d’uso.
4. Il flusso di lavoro dei requisiti standard dell’UP si estende con:
 - attore: Ingegnere dei requisiti;
 - attività: Individuare requisiti funzionali;
 - attività: Individuare requisiti non-funzionali;
 - attività: Assegnare priorità ai requisiti;
 - attività: Estrarre dai requisiti i casi d’uso.
5. Circa il 25% dei progetti falliscono per problemi di ingegneria dei requisiti.
6. Esistono due tipi di requisiti:
 - requisiti funzionali: il comportamento che il sistema dovrebbe avere;
 - requisiti non-funzionali: proprietà o vincoli specifici del sistema.
7. I requisiti ben formulati dovrebbero essere espressi in italiano semplice e strutturato, utilizzando affermazioni standard, in modo che possano essere facilmente interpretati dagli strumenti di ingegneria dei requisiti.
8. Le Specifiche dei Requisiti di Sistema contengono i requisiti funzionali e non-funzionali di un sistema. Può trattarsi di:
 - un documento;
 - un *database* gestito da uno strumento di gestione dei requisiti.
9. La mappa non è il territorio. Il linguaggio naturale contiene:
 - rimozioni: informazioni scartate;
 - distorsioni: informazioni modificate;
 - generalizzazioni: informazioni incorporate in regole, assiomi o principi che riguardano i concetti di verità e falsità.

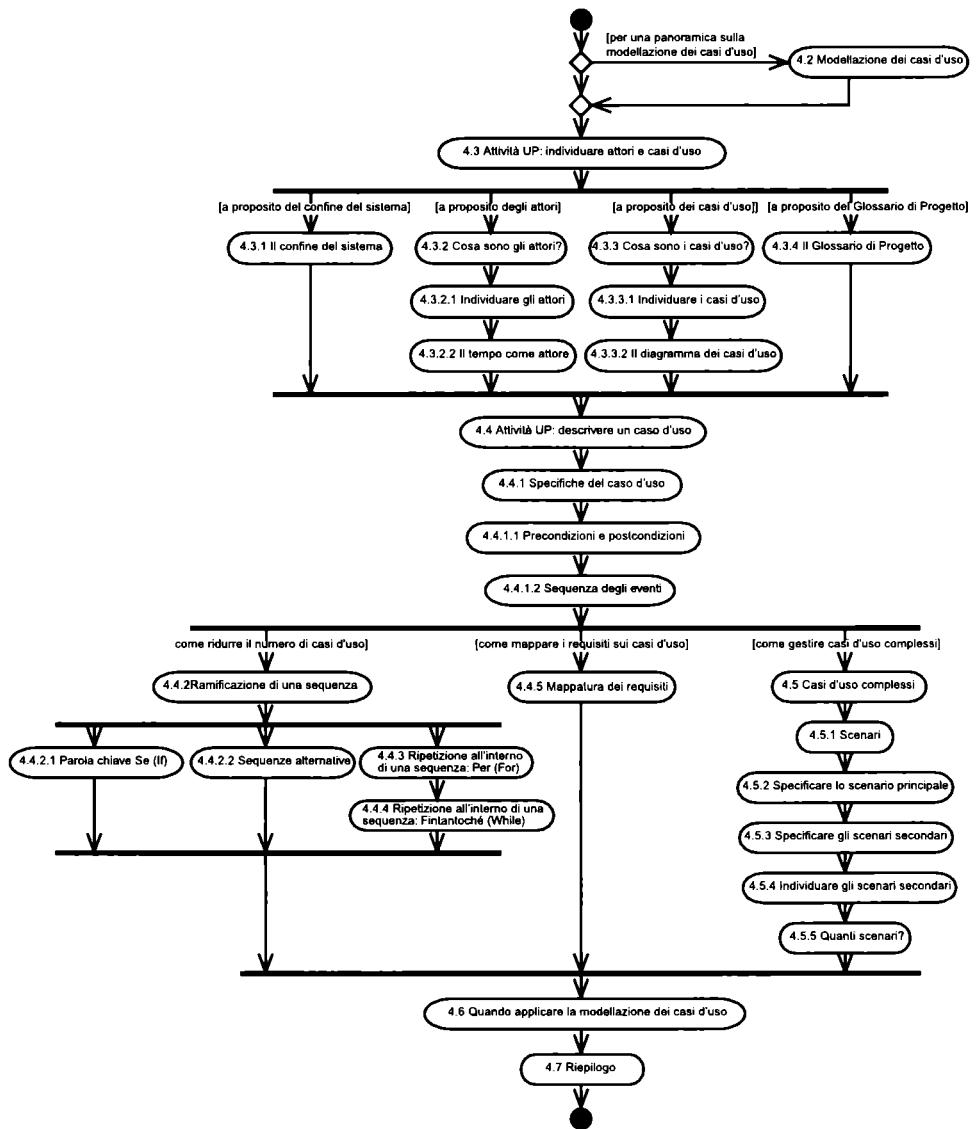
Modellazione dei casi d'uso

4.1 Contenuto del capitolo

In questo capitolo si trattano le nozioni di base della modellazione dei casi d'uso, che è un'altra forma di ingegneria dei requisiti. Si illustra il processo della modellazione dei casi d'uso, così come è definito dall'UP. Si concentra su alcune tecniche e strategie specifiche che possono essere utilizzate dall'analista/progettista OO per modellare i casi d'uso in modo efficace. I casi d'uso utilizzati in questa sezione sono volutamente semplici, in modo da potersi concentrare sulle tecniche. Un esempio più completo (e più complesso) è disponibile sul sito: www.umlandtheunifiedprocess.com.

L'UML non definisce una struttura formale per le specifiche dei casi d'uso. Questo è un problema, perché fa sì che modellatori diversi utilizzino standard diversi. Per evitare parzialmente questo ostacolo, in questo capitolo e nell'esempio completo, è stato adottato uno standard semplice ed efficace. Per facilitare l'applicazione di questo nostro standard, sono a disposizione sul sito web i sorgenti degli Schema XML (eXtensible Markup Language) per i casi d'uso e per gli attori. Questo materiale è liberamente utilizzabile in qualunque progetto. Si tratta di schemi basati sulle migliori pratiche dell'industria del settore che forniscono uno standard semplice, quanto efficace, per fissare le specifiche dei casi d'uso.

Il sito web include anche un foglio di stile XSL (eXtensible Stylesheet Language) molto semplice che trasforma i documenti di caso d'uso XML in formato HTML, in modo da essere visualizzabili e stampabili con un normale *browser*. Questo foglio di stile è un esempio utile e può essere facilmente personalizzato per incorporare i logo o gli standard grafici delle diverse aziende. Dato che in questo libro non si tratta in dettaglio dell'XML, per qualunque difficoltà con questi documenti o per qualunque ulteriore approfondimento, si può far riferimento a testi specifici di XML, quali [Pitts 1] e [Kay 1].

**Figura 4.1**

4.2 Modellazione dei casi d'uso

La modellazione dei casi d'uso è una forma di ingegneria dei requisiti. Nel Paragrafo 3.6 abbiamo illustrato come creare una SRS in quello che si può chiamare il modo “tradizionale”. La modellazione dei casi d'uso è una tecnica diversa e complementare per raccogliere e documentare i requisiti.

La modellazione dei casi d'uso prevede tipicamente i seguenti passi:

- individuare il confine del sistema;
- individuare gli attori;
- individuare i casi d'uso:
 - a. specificare il caso d'uso;
 - b. creare gli scenari.

Queste attività producono il modello dei casi d'uso. Il modello è composto da quattro componenti:

- attori: i ruoli assunti dalle persone e dalle cose che usano il sistema;
- casi d'uso: quello che gli attori possono fare con il sistema;
- relazioni: relazioni significative tra gli attori e i casi d'uso;
- confine del sistema: un rettangolo disegnato intorno i casi d'uso per indicare il confine del sistema oggetto del modello.

Inoltre, il modello dei casi d'uso costituisce una fonte primaria di oggetti e classi. Costituisce l'input più importante per la modellazione delle classi.

4.3 Attività UP: individuare attori e casi d'uso

In questa sezione ci si concentra sull'attività “Individuare attori e casi d'uso” del flusso di lavoro dei requisiti (vedere il Paragrafo 3.4), illustrata nella Figura 4.2. Nel Paragrafo 4.4 si prosegue con l'attività “Descrivere un caso d'uso”.

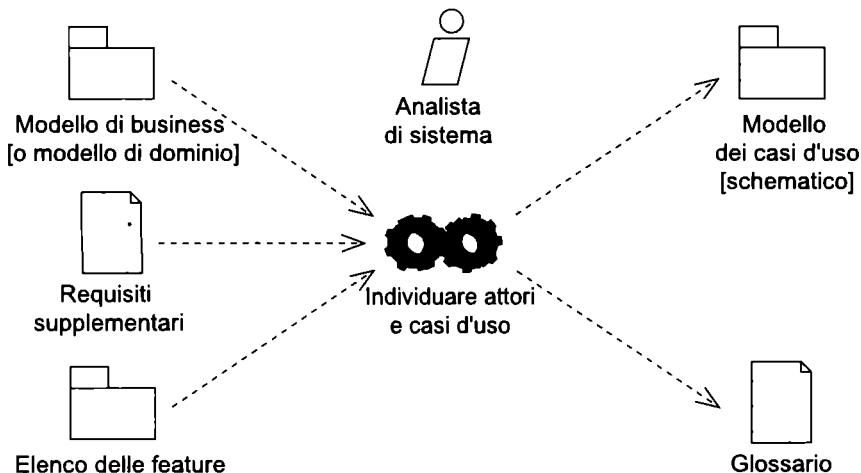


Figura 4.2 Riprodotta da Figura 7.11 [Jacobson 1], con il consenso della Addison-Wesley

4.3.1 Il confine del sistema

Quando si comincia a pensare di costruire un sistema, la prima cosa da fare è stabilire quali siano i suoi confini. In altre parole, è necessario decidere cosa *fa parte* del sistema (dentro i suoi confini) e cosa *non ne fa parte* (fuori dai suoi confini). Questa affermazione può sembrare banale e ovvia, ma su molti progetti sono sorti problemi proprio a causa di un confine di sistema non ben definito. Il posizionamento del confine del sistema può avere un enorme impatto sui requisiti funzionali (e, a volte, anche su quelli non-funzionali), e abbiamo già visto quanto i requisiti incompleti o mal definiti possano contribuire al fallimento di un progetto.

Il confine del sistema è determinato da chi o cosa usa il sistema (ovvero gli attori) e da quali specifici benefici o funzioni il sistema offre ai suoi attori (ovvero i casi d'uso).

Il confine del sistema viene rappresentato da un rettangolo, etichettato con il nome del sistema. Gli attori vengono posizionati all'*esterno* del rettangolo, mentre i casi d'uso finiscono all'*interno*. Quando si inizia a effettuare la modellazione dei casi d'uso si ha un'idea piuttosto vaga di dove si trovi effettivamente il confine del sistema. Man mano che vengono individuati gli attori e i casi d'uso, il confine del sistema diventa sempre più definito.

4.3.2 Cosa sono gli attori?

Un attore identifica il ruolo che una entità esterna assume quando interagisce direttamente con il sistema. Può rappresentare il ruolo di un utente o di un altro sistema, che in qualche modo interessa il confine del sistema.

La Figura 4.3 illustra come vengono rappresentati gli attori nell'UML. L'attore è semplicemente uno stereotipo di classe, con una propria icona. Entrambe le forme utilizzate per l'elemento attore sono valide, ma molti modellatori preferiscono la forma a "uomo stecchino" per rappresentare i ruoli che probabilmente verranno interpretati da persone umane, e la forma a "rettangolo" per i ruoli che probabilmente verranno interpretati da altri sistemi.

È importante rendersi conto che gli attori devono sempre essere *esterni* al sistema. Quando, per esempio, un utente utilizza un sistema di commercio elettronico (quale una libreria *online* per l'acquisto di un libro) si ritrova a essere all'esterno di quel sistema. È tuttavia interessante osservare che anche se gli attori veri e propri restano sempre esterni a un sistema, spesso i sistemi mantengono una qualche rappresentazione interna di uno o più attori. Nell'esempio appena citato, il sistema di commercio elettronico della libreria *online* tiene traccia di alcune informazioni degli utenti clienti: nome, indirizzo ecc.



Figura 4.3

Questa è una rappresentazione interna dell'attore Cliente. È importante che sia ben chiara la differenza: l'attore Cliente è *esterno* al sistema, ma il sistema può utilizzare una classe DettaglioCliente, che è una rappresentazione *interna* degli individui che interpretano il ruolo dell'attore Cliente.

4.3.2.1 Individuare gli attori

Per individuare gli attori, è necessario capire chi o cosa utilizzi il sistema e quali ruoli rivestano mentre interagiscono con il sistema. Per fissare i ruoli che le persone e le cose rivestono nei confronti di un sistema, si può iniziare a prendere in considerazione persone o cose specifiche e quindi generalizzare. Le risposte alle seguenti domande possono aiutare a individuare gli attori.

- Chi o cosa usa il sistema?
- Quale ruolo rivestono durante l'interazione con il sistema?
- Chi installa il sistema?
- Chi avvia e ferma il sistema?
- Chi effettua manutenzione sul sistema?
- Quali altri sistemi interagiscono con il sistema?
- Chi ottiene informazioni dal sistema e chi ne fornisce?
- Esistono funzioni che vengono eseguite a intervalli, orari o date prestabiliti?

Per modellare correttamente gli attori occorre rammentare i sottoelencati punti.

- Gli attori sono sempre esterni al sistema: non possono quindi essere controllati.
- Gli attori interagiscono direttamente con il sistema: è così che aiutano a fissare il confine del sistema.
- Gli attori rappresentano i ruoli generici che persone o cose possono rivestire nei confronti del sistema: non rappresentano persone o cose specifiche.
- Una stessa persona può rivestire, contemporaneamente o in tempi diversi, più ruoli nei confronti del sistema. Per esempio, dal punto di vista di un sistema di pianificazione dei corsi, un'unica persona che scrive corsi, ma li tiene anche, ha due ruoli distinti: "Responsabile di corso" e "Autore del corso".
- Ogni attore deve essere identificato con un nome breve, che abbia senso dal punto di vista del *business*.
- Ogni attore deve essere caratterizzato da una breve descrizione (un paio di righe) che lo definisca dal punto di vista del *business*.
- L'icona di un attore, come quella di una classe, può essere caratterizzata da una sottosezione che elenca i suoi attributi, e da una che elenca gli eventi a cui possono rispondere. Queste sottosezioni sono utilizzate molto di rado nei diagrammi dei casi d'uso, e non saranno trattate oltre.

4.3.2.2 Il tempo come attore

Per modellare delle funzioni che avvengono nel sistema in un certo momento noto, e che *non sembrano* essere provocate da alcun attore specifico, si può ricorrere a un attore speciale, chiamato Tempo, come illustrato nella Figura 4.4. Questa tecnica è, per esempio, indicata per modellare una procedura di *backup* del sistema che viene automaticamente eseguita tutte le sere.



Figura 4.4

4.3.3 Cosa sono i casi d'uso?

The UML Reference Manual [Rumbaugh 1] definisce il caso d'uso come: “La specifica di una sequenza di azioni, incluse eventuali sequenze alternative e sequenze di errore, che un sistema, un sottosistema o un classe può eseguire interagendo con attori esterni.”

Il caso d'uso è qualcosa che un attore vuole che il sistema faccia. È un “caso d'uso” del sistema da parte di uno specifico attore:

- i casi d'uso vengono *sempre* avviati dall'intervento di un attore;
- i casi d'uso vengono *sempre* descritti dal punto di vista di un attore.

Si è abituati a ragionare sui casi d'uso a livello di sistema ma in realtà, come afferma la definizione riportata, ci si può anche occupare di casi d'uso a livello di sottosistema o, addirittura, di singola classe. I casi d'uso possono anche essere molto efficaci per modellare i processi di *business*, anche se quest'aspetto non verrà trattato in questo libro.

La Figura 4.5 illustra l'icona che l'UML utilizza per rappresentare i casi d'uso. Il nome del caso d'uso può essere riportato indifferentemente dentro o sotto l'ovale.

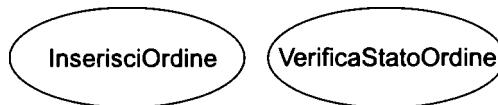


Figura 4.5

4.3.3.1 Individuare i casi d'uso

Il miglior modo per individuare i casi d'uso è di partire dall'elenco degli attori e di ragionare su ciò che ciascun attore fa per utilizzare il sistema. Questa strategia consente di ottenere un elenco di potenziali casi d'uso. È, quindi, necessario assegnare a ogni caso d'uso una frase corta e descrittiva composta essenzialmente da un verbo: dopotutto, il caso d'uso *fa* qualcosa!

Mettendo insieme i casi d'uso potrebbe spuntare qualche nuovo attore: è normale. A volte è necessario approfondire le funzionalità del sistema prima di riuscire a individuare tutti gli attori o tutti gli attori *giusti*.

La modellazione dei casi d'uso è un'operazione iterativa che viene effettuata per approssimazioni successive. Per ciascun caso d'uso inizialmente si fissa solo il nome. I dettagli vengono completati successivamente: una prima descrizione sintetica, che viene in seguito rifinata fino a diventare una specifica completa. Le risposte alle seguenti domande possono aiutare a individuare i casi d'uso.

- Quali funzioni si aspetta dal sistema ciascun attore?
- Il sistema archivia e recupera informazioni? Se sì, quali attori provocano questo comportamento?
- Esistono attori che vengono notificati quando il sistema cambia stato?
- Esistono eventi esterni che producono effetti sul sistema? Chi o cosa notifica al sistema questi eventi?

4.3.3.2 Il diagramma dei casi d'uso

Nel diagramma dei casi d'uso il confine del sistema viene rappresentato da un rettangolo identificato dal nome del sistema. Gli attori sono collocati fuori dal confine del sistema (esterni al sistema) e i casi d'uso, che costituiscono il comportamento del sistema, sono collocati dentro il confine del sistema (interni al sistema). Questo diagramma viene illustrato nella Figura 4.6.

La relazione tra un attore e un caso d'uso viene rappresentata da una linea solida, che è il simbolo UML dell'associazione. Le associazioni saranno trattate in modo approfondito nel Capitolo 9. Questa associazione è sempre caratterizzata dallo stereotipo implicito «comunicazione», il quale indica che l'attore e il caso d'uso comunicano in qualche modo. Questo stereotipo non viene mai mostrato esplicitamente sul diagramma dei casi d'uso.

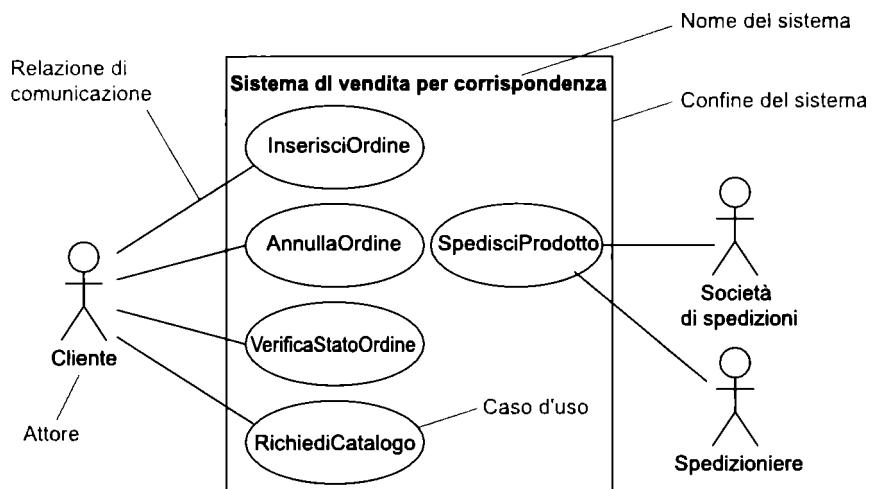


Figura 4.6

4.3.4 Il Glossario di Progetto

Il Glossario del Progetto è sicuramente tra gli artefatti più importanti del progetto. Ogni dominio di problema ha un suo linguaggio specifico; lo scopo primario dell'ingegneria dei requisiti e dell'analisi è proprio quello di comprendere e fissare quel linguaggio. Il Glossario del Progetto è il dizionario che contiene i principali termini del dominio e le loro definizioni. Dovrebbe risultare comprensibile a tutti i partecipanti e le parti interessate al progetto.

Oltre a definire i principali termini, il Glossario di Progetto deve anche risolvere i casi di sinonimia e omonimia.

- I sinonimi sono vocaboli diversi che hanno lo stesso significato. L'analista OO del progetto dovrà scegliere uno di questi vocaboli (quello che sembra essere più utilizzato dalle parti interessate) e usare solo quello. Gli altri vocaboli non dovranno assolutamente comparire in alcuno dei modelli. L'uso di sinonimi può, infatti, facilmente portare a implementare classi con funzionalità molto simili, ma con nomi diversi. Laddove si consente l'uso di diversi sinonimi all'interno dello stesso modello, si può star certi che, col tempo, il significato di questi vocaboli comincerà a divergere.
- L'omonimia si presenta quando lo stesso vocabolo assume un diverso significato per diverse persone. Questo fenomeno produce sempre seri problemi di comunicazione, dato che le diverse parti interessate si ritrovano in effetti a parlare lingue diverse, pur *convinte*, invece, di parlare tutte la stessa. Anche in questo caso, sarà compito dell'analista scegliere uno dei significati per il vocabolo, magari introducendo nuovi vocaboli per gestire gli altri significati.

Le scelte di vocaboli e di significato effettuate devono essere riportate nel Glossario di Progetto, assieme all'eventuale elenco dei sinonimi individuati. Questa pratica potrebbe richiedere che si insista con alcune parti interessate affinché si abituino a utilizzare una diversa terminologia. Non è facile convincere le parti interessate a cambiare il loro uso del linguaggio, eppure, con un minimo di perseveranza, è possibile.

L'UML non impone alcuno standard per il Glossario di Progetto. È buona norma utilizzare una soluzione semplice e concisa. Si consiglia un formato simile a quello di un dizionario, con un elenco di vocaboli, ordinato alfabeticamente, accompagnati dalle relative definizioni. Può essere sufficiente un documento di solo testo, ma nei progetti grandi potrebbe essere richiesto che il glossario sia disponibile *online*, in formato HTML o XML, o magari addirittura in un semplice *database*. Si ricorda che l'impatto che il glossario avrà sul progetto sarà tanto più positivo, quanto più sarà accessibile e facile da utilizzare il glossario stesso.

Un punto di attenzione che riguarda il Glossario di Progetto è che gli stessi vocaboli e definizioni vengano anche utilizzati in tutto il modello UML. È necessario assicurarsi che i due documenti risultino sempre allineati e sincronizzati. Sfortunatamente la maggior parte degli strumenti CASE non fornisce alcun supporto per questo tipo di documento ed è quindi solitamente necessario aggiornarlo manualmente.

4.4 Attività UP: descrivere un caso d'uso

Dopo aver creato un diagramma dei casi d'uso e aver individuato gli attori e i casi d'uso più importanti, è necessario iniziare a definire le specifiche di ciascun caso d'uso. Questa attività dell'UP, nota come “Descrivere un caso d'uso”, viene illustrata nella Figura 4.7.

A questo punto è importante osservare che tipicamente queste attività non vengono svolte secondo una sequenza rigida. È possibile descrivere alcuni, o tutti, i casi d'uso man mano che vengono individuati. È sempre molto difficile spiegare un processo a attività parallelizzabili in un libro che, per sua natura, è lineare!

Questa attività produce un caso d'uso corredato di maggiori dettagli. È necessario perlomeno includere il nome del caso d'uso e le sue specifiche. Opzionalmente si può anche aggiungere una breve descrizione.

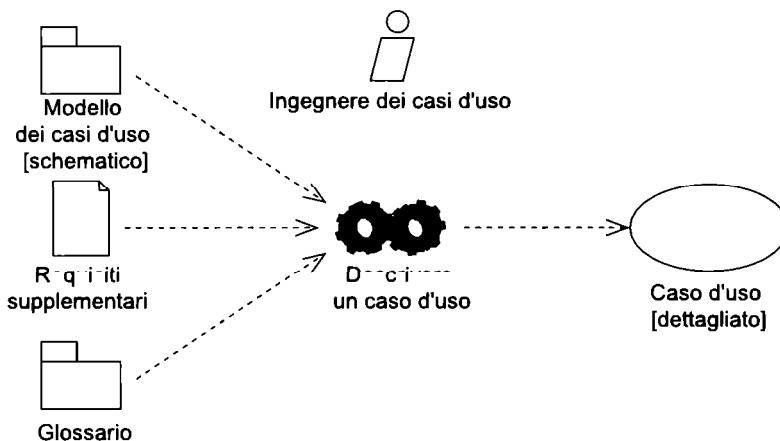


Figura 4.7 Riprodotta da Figura 7.14 [Jacobson 1], con il consenso della Addison-Wesley.

4.4.1 Specifiche del caso d'uso

L'UML non fornisce alcuno standard per la formulazione delle specifiche del caso d'uso. Tuttavia il modulo illustrato nella Figura 4.8 è di uso piuttosto comune. Qualcuno utilizza moduli più dettagliati e complessi, ma in linea generale è consigliabile che la modellazione dei casi d'uso resti il più semplice possibile. Il caso d'uso illustrato nella Figura 4.8 riguarda il pagamento dell'IVA.

Ogni caso d'uso ha un nome e una specifica. La specifica è composta da:

- precondizioni: condizioni che devono essere vere prima che il caso d'uso possa essere eseguito; sono vincoli sullo stato iniziale del sistema;
- sequenza degli eventi: i passi che compongono il caso d'uso;
- postcondizioni: condizioni che devono essere vere quando il caso d'uso termina l'esecuzione.

Nome del caso d'uso Identificatore univoco Gli attori interessati dal caso d'uso Lo stato del sistema prima che il caso d'uso possa iniziare I passi del caso d'uso Lo stato del sistema quando l'esecuzione del caso d'uso è terminata	<p>Caso d'uso: PagamentoIVA</p> <p>ID: UC1</p> <p>Attori: Tempo Fisco</p> <p>Precondizioni: 1. Si è concluso un trimestre fiscale.</p> <p>Sequenza degli eventi:</p> <ol style="list-style-type: none"> Il caso d'uso inizia quando si conclude un trimestre fiscale. Il sistema calcola l'ammontare dell'IVA dovuta al Fisco. Il sistema trasmette un pagamento elettronico al Fisco. <p>Postcondizioni: 1. Il Fisco riceve l'importo IVA dovuto.</p>
--	---

Figura 4.8

Nell'esempio riportato, il fisco ottiene sempre le tasse che gli spettano, in un modo o nell'altro, e quindi questa è stata specificata come postcondizione del caso d'uso.

4.4.1.1 Precondizioni e postcondizioni

Le precondizioni e le postcondizioni sono vincoli.

- Le precondizioni vincolano lo stato del sistema prima che il caso d'uso possa iniziare l'esecuzione. Possono essere raffigurate come dei guardiani che impediscono all'attore di far scattare il caso d'uso a meno che siano state soddisfatte tutte le condizioni previste.
- Le postcondizioni vincolano lo stato del sistema quando il caso d'uso ha terminato la propria esecuzione.

Un altro modo di vederla è che le precondizioni specificano cosa debba essere vero *prima* che il caso d'uso possa essere avviato, mentre le postcondizioni specificano cosa debba essere vero *dopo* che il caso d'uso è stato eseguito.

Le precondizioni e le postcondizioni possono aiutare a progettare sistemi che funzionano correttamente.

Le precondizioni e le postcondizioni dovrebbero sempre essere espresse come semplici affermazioni sullo stato del sistema, che possono essere vere o false; questo tipo di affermazioni sono anche dette condizioni Booleane.

4.4.1.2 Sequenza degli eventi

La sequenza degli eventi elenca i passi che compongono il caso d'uso. Comincia sempre con un attore che fa qualcosa per dare inizio al caso d'uso. Un buon modo per iniziare la sequenza degli eventi è la seguente.

1. Il caso d'uso inizia quando un <attore> <funzione>.

Come abbiamo detto precedentemente, l'attore potrebbe anche essere il tempo. In questo caso il caso d'uso può iniziare con un'espressione temporale al posto di <attore> <funzione>, come nell'esempio della Figura 4.8.

La sequenza degli eventi è costituita da un elenco di passi che devono essere concisi, dichiarativi, numerati e ordinati temporalmente. Ogni passo del caso d'uso dovrebbe avere la seguente struttura:

<numero> || <qualcosa> <qualche azione>.

È possibile esprimere la sequenza degli eventi del caso d'uso in prosa, ma è sconsigliabile in quanto solitamente risulta veramente troppo impreciso.

È possibile illustrare delle alternative nella sequenza degli eventi di un caso d'uso utilizzando le ramificazioni o elencando dei frammenti di comportamento nella sezione Sequenza Alternativa del caso d'uso. Queste due tecniche vengono approfondite nel Paragrafo 4.4.2.

Ecco un esempio di un paio di passi della sequenza degli eventi del caso d'uso NuovoOrdine:

1. Il caso d'uso inizia quando il cliente seleziona la funzione "ordina libro".
2. Il cliente inserisce nel form il suo nome e indirizzo.

Questi passi sono ben strutturati. Entrambi sono semplici affermazioni che dichiarano che un qualcosa eseguirà una qualche azione. Ecco invece un esempio di passo di caso d'uso mal strutturato:

2. Vengono inseriti i dati del cliente.

Questa frase ha subito tre rimozioni importanti.

- Chi inserisce i dati del cliente? Chi dà inizio al caso d'uso?
- Dove vengono inseriti i dati?
- Quali sono esattamente i "dati del cliente"?

Quando si descrive una sequenza degli eventi di un caso d'uso è molto importante riconoscere una rimozione ed evitarla. Non importa se l'informazione mancante può sembrare desumibile dal contesto, o facilmente intuibile. Il problema è che ogni caso d'uso deve essere una dichiarazione precisa di una delle funzionalità del sistema!

Quando durante le attività di analisi capita di riscontrare vaghezza, imprecisione, rimozione o generalizzazione, può essere utile insistere, formulando alcune domande.

Tipicamente queste sono:

- Esattamente chi...?
- Esattamente cosa...?
- Esattamente quando...?
- Esattamente dove...?

4.4.2 Ramificazione di una sequenza

In un caso d'uso è spesso necessario indicare che esistono diverse sequenze o passi alternativi. Un'ottima tecnica è quella di utilizzare linguaggio strutturato. Si introduce un insieme ridotto di parole chiave che esprimono ramificazione, ripetizione o sequenze alternative. (Gli autori consigliano l'uso di parole chiave nella propria lingua, quindi saranno utilizzate parole italiane; si lasceranno comunque tra parentesi le corrispondenti parole inglesi, dato che sono molto utilizzate anche in Italia - N.d.T.)

È forse il caso di notare che alcuni modellatori non accettano l'uso della ramificazione all'interno di un caso d'uso. Sostengono che ogni ramificazione dovrebbe essere espressa introducendo un nuovo caso d'uso. In teoria questa tesi ha dei meriti; tuttavia in questo libro si utilizza un approccio più pragmatico. Una quantità non eccessiva di ramificazioni semplici è desiderabile in quanto riduce il numero complessivo di casi d'uso, consentendo una rappresentazione più compatta dei requisiti.

Nel Paragrafo 4.5 saranno affrontate le tecniche specifiche per gestire casi d'uso complessi, dove l'uso di ramificazione interna della sequenza risulta del tutto inappropriato.

4.4.2.1 Parola chiave Se (*If*)

La parola chiave Se indica una ramificazione della sequenza degli eventi. La Figura 4.9 illustra una sequenza degli eventi ben strutturata che contiene tre rami. Ogni ramo inizia con la parola chiave Se seguita da una semplice espressione Booleana che potrà risultare vera o falsa, come per esempio: "Se l'utente digita una quantità diversa". Il testo indentato che si trova sotto questa dichiarazione dice cosa succederà quando l'espressione Booleana risulta vera. L'indentazione del testo e la numerazione esprimono chiaramente la struttura del ramo Se, e non è necessario ricorrere a esplicativi costrutti sintattici di chiusura del ramo quali, per esempio, il FineSe.

4.4.2.2 Sequenze alternative

Taluni tipi di ramificazione non possono essere facilmente espressi utilizzando il Se. In particolare, questo è vero per quelle condizioni che si possono verificare un qualunque momento. Dove potrebbe essere posizionata la dichiarazione Se per gestire questo tipo di evento? La Figura 4.10 illustra un caso tipico.

Caso d'uso: AggiornaCarrello
ID: UC10
Attori: Cliente
Precondizioni: 1. Il contenuto del carrello della spesa è visibile
Sequenza degli eventi: 1. Il caso d'uso inizia quando il Cliente seleziona un articolo nel carrello. 2. Se il Cliente seleziona "rimuovi articolo" 2.1 Il sistema elimina l'articolo dal carrello. 3. Se il Cliente digita una nuova quantità 3.1 Il sistema aggiorna la quantità dell'articolo presente nel carrello.
Postcondizioni: 1. Il contenuto del carrello è stato aggiornato.

Figura 4.9

Caso d'uso: MostraCarrello
ID: UC11
Attori: Cliente
Precondizioni: 1. Il Cliente è stato autenticato dal sistema.
Sequenza degli eventi: 1. Il caso d'uso inizia quando il Cliente seleziona "mostra carrello". 2. Se non ci sono articoli nel carrello della spesa 2.1 Il sistema informa il Cliente che non ci sono articoli nel carrello. 2.2 Il caso d'uso termina. 3. Il sistema mostra l'elenco di tutti gli articoli presenti nel carrello della spesa del Cliente, riportando l'ID, il nome, la quantità ed il prezzo.
Postcondizioni:
Sequenza alternativa 1: 1. In qualunque momento il Cliente può abbandonare la pagina del carrello della spesa.
Postcondizioni:
Sequenza alternativa 2: 1. In qualunque momento il Cliente può abbandonare il sistema.
Postcondizioni:

Figura 4.10

Il modo migliore per esprimere ramificazioni che possono avvenire in un qualunque punto della sequenza degli eventi è quello di introdurre una o più sequenze alternative. Ecco come si fa.

- Specificare le precondizioni del caso d'uso: queste devono essere vere per tutte le possibili sequenze degli eventi del caso d'uso.
- Specificare la sequenza degli eventi normale.
- Specificare le postcondizioni valide per la sequenza degli eventi normale.

A questo punto si aggiunge al caso d'uso una nuova sezione per ogni sequenza alternativa degli eventi. Questa sezione deve contenere:

- la sequenza degli eventi alternativa. Questa dovrebbe essere una sequenza semplice (composta da pochi passi) e deve *sempre* cominciare con la condizione Booleana che dà inizio all'esecuzione della sequenza.
- Le postcondizioni della sequenza degli eventi alternativa.

Non è possibile aggiungere le postcondizioni delle sequenze alternative a quelle della sequenza principale, in quanto la sequenza alternativa potrebbe anche non essere eseguita.

4.4.3 Ripetizione all'interno di una sequenza: Per (For)

A volte è necessario che un'azione venga ripetuta in una stessa sequenza di eventi. È piuttosto raro ricorrere a questo tipo di specifica nella modellazione dei casi d'uso, ma quando serve è meglio disporre di una tecnica adatta. Per modellare la ripetizione si usa la parola chiave Per. Il formato è:

n. Per (espressione di iterazione)

- n.1. Fai qualcosa
- n.2. Fai qualcos'altro
- n.3. ...

n+1.

L'espressione di iterazione è una qualche espressione che indica il numero, intero e positivo, di iterazioni che devono essere eseguite. Le azioni riportate sulle righe indennate che seguono la dichiarazione Per verranno ripetute il numero di volte indicato dall'espressione di iterazione. La Figura 4.11 illustra un esempio.

4.4.4 Ripetizione all'interno di una sequenza: Fintantoché (While)

La parola chiave Fintantoché serve per modellare una serie di azioni che deve essere ripetuta all'interno della sequenza degli eventi fintantoché è vera una certa condizione Booleana. Il formato è:

n. Fintantoché (condizione Booleana)

- n.1. Fai qualcosa
- n.2. Fai qualcos'altro
- n.3. ...

n+1.

Caso d'uso: TrovaProdotto
ID: UC12
Attori: Cliente
Precondizioni:
<p>Sequenza degli eventi:</p> <ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Cliente seleziona "trova prodotto". 2. Il sistema chiede al Cliente i criteri di ricerca. 3. Il Cliente inserisce i criteri di ricerca richiesti. 4. Il sistema ricerca i prodotti che soddisfano i criteri specificati dal Cliente. 5. Se il sistema trova uno o più prodotti <ol style="list-style-type: none"> 5.1. Per ogni prodotto trovato <ol style="list-style-type: none"> 5.1.1. Il sistema mostra l'immagine ridotta del prodotto. 5.1.2. Il sistema mostra l'elenco delle caratteristiche del prodotto. 5.1.3. Il sistema mostra il prezzo del prodotto 6. Altrimenti <ol style="list-style-type: none"> 6.1. Il sistema comunica al cliente che non sono stati trovati prodotti che soddisfano i criteri specificati.
Postcondizioni:
<p>Sequenza alternativa:</p> <ol style="list-style-type: none"> 1. In qualunque momento il Cliente può spostarsi su una pagina diversa.
Postconditions:

Figura 4.11

Questa parola chiave non è molto utilizzata, proprio come il **Per**.

La Figura 4.12 illustra un esempio. Le azioni riportate sulle righe indentate che seguono la dichiarazione **Fintantoché** verranno ripetute fin quando l'espressione Booleana non diventa falsa.

Caso d'uso: MostraInformazioniSocietà
ID: UC13
Attori: Cliente
Precondizioni:
<p>Sequenza degli eventi:</p> <ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Cliente seleziona "mostra informazioni società". 2. Il sistema mostra una pagina web contenente tutte le informazioni della società. 3. Fintantoché il Cliente visualizza le informazioni della società <ol style="list-style-type: none"> 3.1. Il sistema fa ascoltare della musica di sottofondo. 3.2. Il sistema visualizza offerte speciali in un banner.
Postcondizioni:

Figura 4.12

4.4.5 Mappatura dei requisiti

Usando sia le SRS, sia un insieme di casi d'uso, di fatto ci si ritrova ad avere due distinti "archivi" di requisiti funzionali. È molto importante incrociare il contenuto dei due archivi per scoprire se esistono dei requisiti nelle SRS che non sono coperti dai casi d'uso e viceversa. Questa problematica è una di quelle affrontate dalla mappatura dei requisiti.

La mappatura dei requisiti funzionali sui casi d'uso è ulteriormente complicata dal fatto che la relazione è del tipo molti-a-molti. Un singolo caso d'uso può coprire diversi requisiti funzionali, e un singolo requisito funzionale può manifestarsi in casi d'uso distinti.

È consigliabile utilizzare uno strumento CASE che supporti la mappatura dei requisiti. Effettivamente diversi strumenti di ingegneria dei requisiti, quali Rational Requisite Pro e DOORS, consentono di associare i singoli requisiti funzionali inseriti nel loro *database* dei requisiti a diversi casi d'uso, e viceversa. Anche l'UML offre un buon supporto per la mappatura dei requisiti. È possibile associare a un caso d'uso un insieme di requisiti funzionali inserendo i loro identificativi in una lista di valori etichettati. Nello strumento di gestione dei requisiti è, invece, possibile elencare il nome di uno o più casi d'uso nelle specifiche di ciascun requisito.

Se non si dispone di supporto adeguato da parte dello strumento CASE, allora è necessario fare questo lavoro manualmente. Una buona soluzione può essere quella di creare una Matrice di Mappatura dei Requisiti. Si tratta di una semplice griglia che riporta gli identificativi numerici dei singoli requisiti su un'asse, e i nomi dei casi d'uso (e/o i loro identificativi numerici) sull'altro. Si colloca quindi una crocetta in ogni cella corrispondente a un requisito e a un caso d'uso tra loro associati. Le Matrici di Mappatura dei Requisiti possono essere facilmente create con un qualunque strumento che gestisca fogli elettronici. La Tabella 4.1 ne fornisce un esempio.

La Matrice di Mappatura dei Requisiti è uno strumento molto utile per controllare la consistenza. Se esiste un requisito che non corrisponde ad alcun caso d'uso, allora manca almeno un caso d'uso. D'altra parte, se esistono dei casi d'uso che non corrispondono ad alcun requisito, allora l'elenco dei requisiti è incompleto.

Tabella 4.1

		Caso d'uso			
		UC1	UC2	UC3	UC4
Requisito	R1	X			
	R2		X	X	
	R3			X	
	R4				
	R5	X			

4.5 Casi d'uso complessi

Come regola pratica, si consiglia di definire dei casi d'uso che siano relativamente semplici. Occasionalmente può tuttavia capitare di non riuscire a ridurre la complessità, e di dover quindi formulare un caso d'uso complesso. Per evitare di fissare questa complessità tramite ramificazioni e sequenze alternative, si può ricorrere all'uso di scenari separati.

Questa tecnica permette di isolare la sequenza normale degli eventi dalle eccezioni, riducendo la possibilità di errore.

4.5.1 Scenari

Gli scenari sono solo un modo diverso di vedere i casi d'uso. Ogni scenario corrisponde a uno specifico percorso di attraversamento di un caso d'uso.

Quando si documenta un caso d'uso, è possibile separare i diversi percorsi che possono essere seguiti nella sequenza degli eventi; ciascuno di questi percorsi è uno scenario. La caratteristica principale degli scenari è che *non hanno ramificazioni*. Dunque ogni possibile percorso o ramo della sequenza degli eventi di un caso d'uso genera uno scenario *distinto*.

Ogni caso d'uso ha uno, e uno solo, scenario principale. Questo è il percorso più semplice che attraversa la sequenza complessa; è il percorso ottimista, da “mondo perfetto”. Nello scenario principale succede tutto quello che ci si aspetta e si desidera dal sistema, senza errori, interruzioni o deviazioni.

Ogni caso d'uso ha anche molti scenari secondari: sono i percorsi alternativi per attraversare la sequenza degli eventi.

Può essere utile immaginare che un caso d'uso complesso sia simile alla foce di un fiume, dove si diramano molti canali secondari. Ogni caso d'uso ha uno scenario principale, che è rappresentato dal fiume vero e proprio. Gli altri canali, più piccoli e tortuosi sono gli scenari secondari. Questi scenari secondari descrivono gli errori (scenari di eccezione), le ramificazioni o le interruzioni della sequenza principale.

4.5.2 Specificare lo scenario principale

Quando si utilizzano gli scenari per documentare i casi d'uso, la specifica del caso d'uso deve contenere lo scenario principale e l'elenco degli scenari secondari, riportato in un'apposita sezione. Gli scenari secondari sono quindi documentati a parte, in modo molto simile a come si documentano i casi d'uso. Si riporta un esempio nella Figura 4.13.

4.5.3 Specificare gli scenari secondari

Gli scenari secondari vengono definiti in modo molto simile a come si definiscono i casi d'uso semplici. È necessario dichiarare chiaramente come ha inizio lo scenario e assicurarsi che corrisponda a un unico percorso della sequenza di eventi del caso d'uso; non deve quindi contenere alcuna ramificazione.

Ogni scenario secondario deve essere riconducibile al proprio caso d'uso. La Figura 4.14 illustra una semplice convenzione che risolve bene questo aspetto. Si osservi che gli scenari secondari all'interno della loro sequenza di eventi possono anche fare riferimento allo scenario principale.

Caso d'uso: OperazioniDiCassa	
ID: UC14	
Attori: Cliente	
Precondizioni:	
Scenario principale:	<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Cliente seleziona "procedi alla cassa". 2. Il sistema visualizza l'ordine del Cliente. 3. Il sistema richiede l'identificatore del cliente. 4. Il Cliente inserisce un identificatore di cliente valido. 5. Il sistema recupera e visualizza i dati del Cliente. 6. Il sistema richiede le informazioni della carta di credito – intestatario, numero e scadenza della carta. 7. Il Cliente inserisce le informazioni della carta di credito. 8. Il sistema richiede la conferma dell'ordine. 9. Il Cliente conferma l'ordine. 10. Il sistema addebita la carta di credito. 11. Il sistema visualizza la ricevuta.
Scenari secondari: IdentificatoreClienteNonValido InformazioniCartaCreditoNonValida SuperatoLimiteUtilizzoCartaCredito CartaCreditoScaduta	
Postcondizioni:	

Figura 4.13

Caso d'uso: OperazioniDiCassa	
Scenario secondario: IdentificatoreClienteNonValido	
ID: UC15	
Attori: Cliente	
Precondizioni:	
Scenario secondario:	<ol style="list-style-type: none"> 1. Il caso d'uso inizia al passo 3 del caso d'uso OperazioniDiCassa quando il Cliente inserisce un identificatore cliente non valido 2. Per tre tentativi di inserimento <ol style="list-style-type: none"> 2.1. Il sistema richiede al Cliente di inserire nuovamente il proprio identificatore. 3. Il sistema informa il Cliente che l'identificatore cliente inserito non è stato riconosciuto.
Postcondizioni:	

Figura 4.14

4.5.4 Individuare gli scenari secondari

Gli scenari secondari possono essere individuati analizzando lo scenario principale. A ogni passo della sequenza potrebbero presentarsi:

- ramificazioni del percorso;
- errori o eccezioni;
- interruzioni della sequenza: eventi che possono capitare in qualunque momento.

Ciascuno di questi può dare origine a un nuovo scenario secondario.

4.5.5 Quanti scenari?

Come già detto, ogni caso d'uso può avere un solo scenario principale. Tuttavia gli scenari secondari possono essere *molti*. Il dubbio è: quanti? È indicato cercare di limitare il numero degli scenari secondari al minimo necessario. Esistono due strategie per limitare il numero degli scenari.

- Scegliere gli scenari secondari più importanti e documentare solo quelli.
- Se esiste un insieme di scenari secondari molto simili, documentarne uno solo come esempio e, se necessario, aggiungere delle annotazioni per spiegare come gli altri scenari differiscano dall'esempio.

Tornando alla metafora della foce del fiume: oltre al fiume stesso, possono esistere moltissimi canali alternativi ramificati e contorti. Non ci si può veramente permettere di mapparli tutti, quindi ci si limita a quelli più importanti. Inoltre molti di questi canali scorrono tra loro in parallelo, con lievi differenze di percorso. È possibile mapparne uno solo in modo dettagliato, fornendo delle indicazioni su come gli altri canali se ne discostino. Questa è una tecnica molto efficace per modellare un caso d'uso complesso.

Il principio di base della modellazione dei casi d'uso è di fissare sempre la quantità *minima indispensabile* di informazioni. Questo significa che molti scenari secondari possono anche non essere specificati: sul caso d'uso compare comunque una loro descrizione da una riga che potrebbe essere un dettaglio sufficiente per comprendere il funzionamento del sistema. Questo è un punto importante: è molto facile ritrovarsi sommersi di scenari secondari, ed è capitato più di una volta che l'attività di modellazione dei casi d'uso sia fallita proprio per questo motivo. Si ricordi che la descrizione dei casi d'uso e degli scenari deve servire per *capire il comportamento desiderato* del sistema e non, invece, per produrre un modello completo dei casi d'uso. L'attività di modellazione deve, quindi, essere interrotta non appena si abbia la sensazione di aver compreso questo comportamento. Inoltre, dato che l'UP prevede un ciclo di vita iterativo, nel caso in cui ci si accorga che qualche aspetto del sistema non è molto chiaro, si può sempre ritornare sui relativi casi d'uso per approfondirli.

4.6 Quando applicare la modellazione dei casi d'uso

Dato che i casi d'uso fissano le funzionalità del sistema dal punto di vista degli attori, sono ovviamente poco efficaci per un sistema che abbia un attore solo, o magari anche nessuno. I casi d'uso fissano i requisiti funzionali e sono, quindi, anche poco indicati per i sistemi dominati da requisiti non-funzionali. I casi d'uso sono il miglior modo di fissare i requisiti se:

- il sistema è dominato da requisiti funzionali;
- il sistema fornisce diverse funzionalità a molti tipi di utente (esistono molti attori);
- il sistema ha molte interfacce con altri sistemi (esistono molti attori).

I casi d'uso non sono molto indicati se:

- il sistema è dominato da requisiti non-funzionali;
- il sistema ha pochi tipi di utente;
- il sistema ha poche interfacce con altri sistemi.

Esempi di sistemi poco adatti alla modellazione dei casi d'uso sono i sistemi *embedded* e i sistemi ricchi di algoritmi complessi, ma poveri di interfacce. Per questi sistemi conviene di gran lunga ripiegare su tecniche di ingegneria dei requisiti più convenzionali. È proprio solo una questione di scegliere sempre lo strumento più adatto per il lavoro che si vuole fare.

4.7 Riepilogo

Il capitolo è stato tutto dedicato all'uso della modellazione dei casi d'uso per fissare i requisiti di un sistema. Sono stati spiegati i seguenti concetti.

1. Le attività di modellazione dei casi d'uso fanno parte del flusso di lavoro dei requisiti.
2. Nel ciclo di vita di un progetto UP, la maggior parte delle attività del flusso di lavoro dei requisiti vengono effettuate durante le fasi di Avvio ed Elaborazione.
3. La modellazione dei casi d'uso riguarda soprattutto le attività che l'UP chiama “Individuare attori e casi d'uso” e “Descrivere un caso d'uso”.
4. La modellazione dei casi d'uso è una tecnica di ingegneria dei requisiti che segue la seguente procedura:
 - individuare il confine del sistema;
 - individuare gli attori;
 - individuare i casi d'uso.
5. Gli attori rappresentano i ruoli delle entità esterne al sistema che interagiscono direttamente con esso.

- Si possono individuare gli attori ragionando su chi o cosa utilizza o si interfaccia con il sistema.
6. I casi d'uso rappresentano le funzioni che il sistema esegue per conto di, e a beneficio di, un qualche attore. Si possono individuare i casi d'uso ragionando su come ciascun attore interagisce con il sistema.
7. Il diagramma dei casi d'uso mostra:
- il confine del sistema;
 - gli attori;
 - i casi d'uso;
 - le interazioni.
8. Il Glossario di Progetto fornisce un elenco e una definizione dei termini usati nel dominio del problema: risolve i casi di sinonimia e omonimia.
9. La specifica di un caso d'uso contiene:
- il nome del caso d'uso;
 - un identificatore univoco;
 - le precondizioni: vincoli iniziali sul sistema che impattano sull'esecuzione del caso d'uso;
 - la sequenza degli eventi: la sequenza dichiarativa e ordinata temporalmente delle azioni che compongono il caso d'uso;
 - postcondizioni: vincoli finali sul sistema, dovuti all'esecuzione del caso d'uso.
10. È possibile ridurre il numero di casi d'uso utilizzando una quantità non eccessiva di ramificazione all'interno della sequenza degli eventi.
- La parola chiave **Se** indica una possibile ramificazione che avviene in un punto specifico della sequenza.
 - La sezione **Sequenza Alternativa** indica una possibile ramificazione che può avvenire in un qualunque punto della sequenza degli eventi.
11. Per documentare una ripetizione di azioni all'interno della sequenza si usano le parole chiave:
- **Per** (espressione di iterazione);
 - **Fintantoché** (condizione Booleana).
12. La Matrice di Mappatura dei Requisiti consente di documentare la corrispondenza esistente tra i requisiti delle Specifiche dei Requisiti di Sistema (SRS) e i casi d'uso individuati.
13. I casi d'uso complessi possono essere scomposti in diversi scenari; ogni scenario rappresenta uno specifico percorso di attraversamento del caso d'uso.

14. Ogni caso d'uso complesso ha un unico scenario principale: lo scenario dove tutto avviene come previsto, nei migliori dei modi: lo scenario da “mondo perfetto”.
15. Ogni caso d'uso complesso può avere uno o più scenari secondari: sono i percorsi alternativi che corrispondono a eccezioni, ramificazioni o interruzioni.
16. È possibile individuare gli scenari secondari cercando nella sequenza degli eventi dello scenario principale:
 - percorsi alternativi;
 - possibili errori o eccezioni;
 - interruzioni.
17. Un caso d'uso non deve essere scomposto in scenari a meno che questi aggiungano valore al modello.
18. La modellazione dei casi d'uso è particolarmente indicata per sistemi che:
 - sono dominati da requisiti funzionali;
 - hanno molti tipi di utente;
 - hanno molte interfacce con altri sistemi.
19. La modellazione dei casi d'uso è poco indicata per sistemi che:
 - sono dominati da requisiti non-funzionali;
 - hanno pochi utenti;
 - hanno poche interfacce con altri sistemi.

Modellazione dei casi d'uso: tecniche avanzate

5.1 Contenuto del capitolo

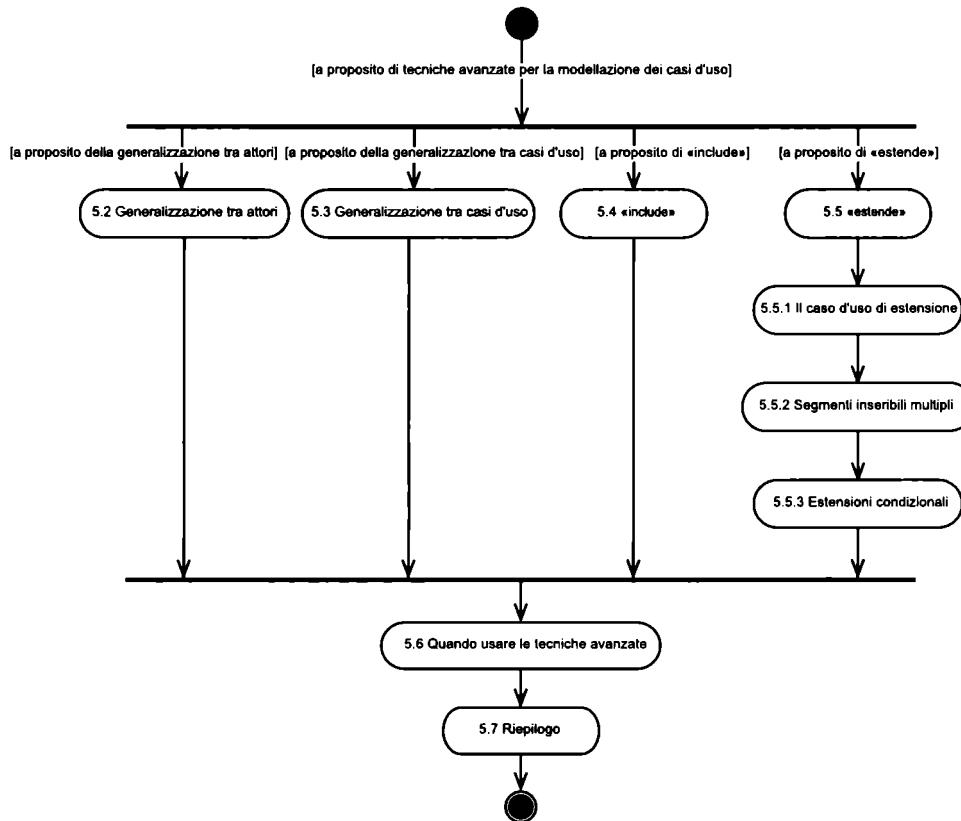
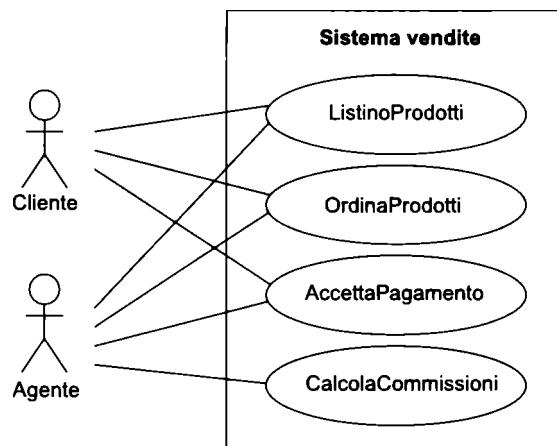
Nel capitolo vengono trattate alcune tecniche avanzate di modellazione dei casi d'uso. Si discute, in particolare, delle relazioni che possono intercorrere tra attori distinti o tra casi d'uso distinti.

- Generalizzazione tra attori: una relazione di generalizzazione tra un attore generico e un attore più specifico.
- Generalizzazione tra casi d'uso: una relazione di generalizzazione tra un caso d'uso generico e un caso d'uso più specifico.
- «include»: una relazione tra casi d'uso che consente a un caso d'uso di includere il comportamento e le funzionalità di un altro caso d'uso.
- «estende»: una relazione tra casi d'uso che consente a un caso d'uso di estendere il proprio comportamento utilizzando uno o più frammenti di comportamento di un altro caso d'uso.

È molto importante che i modelli restino i più semplici possibile, quindi queste tecniche dovrebbero essere utilizzate con molta cura e solo quando contribuiscono ad aumentare la comprensibilità del modello dei casi d'uso. In particolare, è piuttosto facile esagerare con l'uso di «include» e «estende», pratica che deve essere assolutamente evitata.

5.2 Generalizzazione tra attori

Nell'esempio riportato in Figura 5.2 si può vedere come esista una certa somiglianza tra i due attori, Cliente e Agente, nel modo in cui interagiscono con il sistema vendite (in questo sistema è previsto che l'Agente possa gestire una vendita per conto di un Cliente).

**Figura 5.1****Figura 5.2**

Entrambi gli attori danno inizio ai casi d'uso ListinoProdotti, OrdinaProdotti, e AccettaPagamento. In effetti, l'unica differenza tra i due attori è che l'Agente dà anche inizio al caso d'uso CalcolaCommissioni. A parte il fatto che questa sovrapposizione genera una serie di linee incrociate sul diagramma, sembra anche indicare la presenza di un comportamento comune ai due attori che può essere scorporato e gestito definendo un attore più *generico*.

La Figura 5.3 illustra come si possa utilizzare la generalizzazione tra attori per scorporare questo comportamento comune. Si può creare un attore astratto Acquirente che interagisce con i casi d'uso ListinoProdotti, OrdinaProdotti e AccettaPagamento. Cliente e Agente vengono detti attori concreti, perché rappresentano ruoli che potrebbero essere coperti da persone (o sistemi esterni) reali. Acquirente, invece, è un attore astratto in quanto rappresenta esclusivamente un'astrazione introdotta per fissare meglio il comportamento comune dei due attori concreti. Cliente e Agente ereditano tutti i ruoli e le relazioni che l'attore generalizzato (o genitore) astratto ha nei confronti dei casi d'uso. Quindi, interpretando la Figura 5.3: Cliente e Agente hanno entrambi una relazione con i casi d'uso ListinoProdotti, OrdinaProdotti e AccettaPagamento che ereditano dal loro attore genitore comune, Acquirente. Agente, inoltre, ha un'altra interazione con il caso d'uso CalcolaCommissioni che non è ereditata, ma specifica dell'attore specializzato Agente. Si osserva, quindi, come l'uso attento di attori generalizzati possa semplificare i diagrammi dei casi d'uso.

Vale la pena notare che, in una generalizzazione di attori, l'attore genitore non deve necessariamente essere astratto; potrebbe benissimo essere un ruolo concreto da far interpretare a una persona o sistema esterno reale. Tuttavia, per una questione di stile e di leggibilità, si consiglia di utilizzare per lo più attori generalizzati astratti.

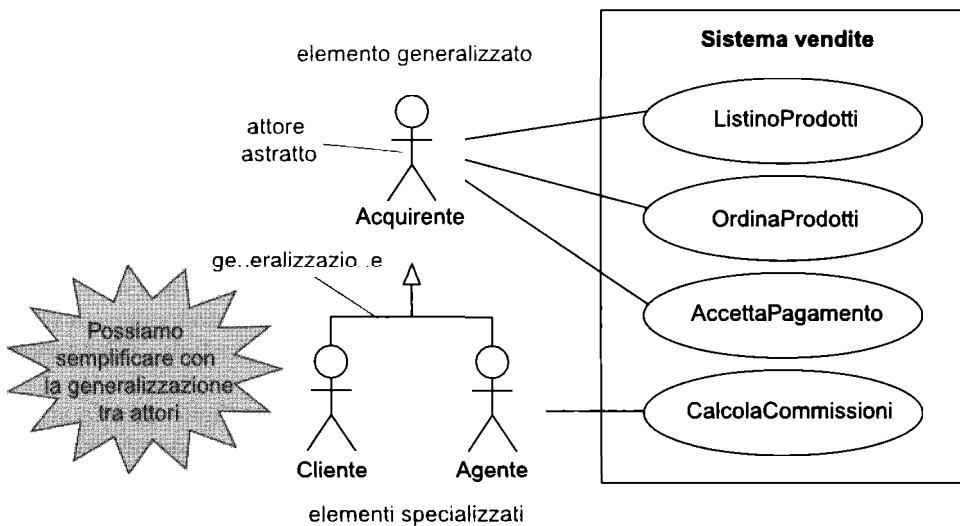


Figura 5.3

Abbiamo visto che il fatto che due attori interagisca con uno stesso insieme di casi d'uso può essere espresso tramite una generalizzazione tra attori introducendo un altro attore genitore (possibilmente astratto). Gli attori specializzati (o figli) ereditano i ruoli e le relazioni dell'attore generalizzato. È possibile utilizzare un attore specializzato in qualunque punto ci si aspetti l'utilizzo di un suo attore generalizzato. Questo viene detto principio di sostituibilità ed è un ottimo modo per verificare l'uso corretto della generalizzazione, con qualunque tipo di classificatore.

In questo esempio è ragionevole aspettarsi che sia possibile sostituire un **Agente** o un **Cliente** ovunque sia richiesto l'uso di un **Acquirente** (ovvero nell'interazione con i casi d'uso **ListinoProdotti**, **OrdinaProdotti** e **AccettaPagamento**), quindi la generalizzazione è una strategia applicabile.

5.3 Generalizzazione tra casi d'uso

La generalizzazione tra casi d'uso viene utilizzata quando esistono uno o più casi d'uso che sono una specializzazione di un caso d'uso più generalizzato. Come la generalizzazione tra attori, andrebbe utilizzata esclusivamente nel caso in cui semplifichi o renda più comprensibili i diagrammi dei casi d'uso.

Nella generalizzazione tra casi d'uso, i casi d'uso specializzati (figli) rappresentano delle varianti più specifiche del caso d'uso generalizzato (genitore) da cui ereditano. I casi d'uso figli possono:

- ereditare caratteristiche dal caso d'uso genitore;
- aggiungere nuove caratteristiche;
- ridefinire (modificare) caratteristiche ereditate.

Il caso d'uso figlio eredita automaticamente tutte le caratteristiche del caso d'uso genitore. Tuttavia non tutti i tipi di elemento di caso d'uso sono ridefinibili. La Tabella 5.1 illustra le restrizioni esistenti sulla ridefinizione degli elementi.

Come si documenta la generalizzazione tra casi d'uso nelle specifiche dei casi d'uso? Le specifiche UML non rispondono in alcun modo a questo quesito, ma esistono diverse tecniche abbastanza diffuse. Meglio una semplice convenzione tipografica che evidenzi le tre casistiche che si possono presentare nel caso d'uso specializzato. La Tabella 5.2 illustra questa convenzione.

Oltre a evidenziare le caratteristiche ereditate, ridefinite o aggiunte, è anche necessario prestare attenzione alla corrispondenza tra gli elementi del caso d'uso specializzato e quelli del caso d'uso generalizzato, ed esprimere in qualche modo. Questo serve per poter risalire dall'elemento ereditato o ridefinito nel caso d'uso figlio all'elemento originale del caso d'uso genitore.

A volte un elemento ereditato o ridefinito in un caso d'uso figlio avrà la stessa numerazione del corrispondente elemento del caso d'uso genitore. In questo caso non è necessario fare niente. Laddove un elemento del caso d'uso figlio necessiti di una numerazione diversa da quella presente nel caso d'uso genitore, allora è necessario riportare il numero dell'elemento originale, tra parentesi, subito dopo il numero dell'elemento ereditato o ridefinito.

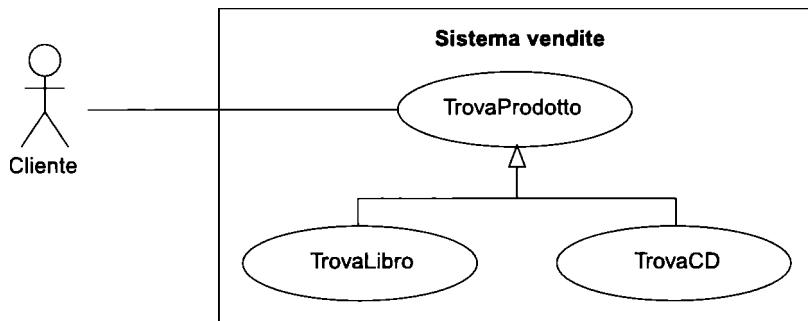
Tabella 5.1

Elemento del caso d'uso	Ereditare	Aggiungere	Ridefinire
Relazione	Sì	Sì	No
Precondizione	Sì	Sì	Sì
Postcondizione	Sì	Sì	Sì
Passo della sequenza principale	Sì	Sì	Sì
Sequenza alternativa	Sì	Sì	Sì
Attributo	Sì	Sì	No
Operazione	Sì	Sì	Sì

Tabella 5.2

La caratteristica è...	Convenzione tipografica
Ereditata e non ridefinita	Testo normale
Ereditata e ridefinita	Testo corsivo
Aggiunta	Testo grassetto

La Figura 5.4 illustra un particolare del diagramma dei casi d'uso di un sistema di vendite. C'è un caso d'uso generalizzato TrovaProdotto da cui discendono due casi d'uso specializzati TrovaLibro e TrovaCD.

**Figura 5.4**

La Figura 5.5 riporta le specifiche del caso d'uso genitore TrovaProdotto; si osservi che è descritto in modo molto astratto.

La Figura 5.6 riporta le specifiche dei due casi d'uso figli. L'esempio illustra l'applicazione di uno standard tipografico per evidenziare le caratteristiche ridefinite e quelle aggiunte.

Caso d'uso: TrovaProdotto	
ID: UC12	
Attori:	Cliente
Precondizioni:	
Sequenza degli eventi:	<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Cliente seleziona "trova prodotto". 2. Il sistema chiede al Cliente i criteri di ricerca. 3. Il Cliente inserisce i criteri di ricerca richiesti. 4. Il sistema ricerca i prodotti che soddisfano i criteri specificati dal Cliente. 5. Se il sistema trova uno o più prodotti <ol style="list-style-type: none"> 5.1. Il sistema mostra un elenco dei prodotti che soddisfano i criteri di ricerca. 6. Altrimenti <ol style="list-style-type: none"> 6.1. Il sistema comunica al cliente che non sono stati trovati prodotti che soddisfano i criteri specificati.
Postcondizioni:	
Sequenza alternativa:	<ol style="list-style-type: none"> 1. In qualunque momento il Cliente può spostarsi su una pagina diversa.
Postcondizioni:	

Figura 5.5

Questi due casi d'uso sono molto più concreti: sono delle versioni specializzate del caso d'uso generalizzato, ciascuna adatta a gestire un particolare tipo di prodotto.

Se un caso d'uso generalizzato non ha una sequenza degli eventi, oppure ne ha una incompleta, allora si dice che è un caso d'uso astratto. I casi d'uso generalizzati astratti sono molto comuni, in quanto consentono di fissare un comportamento a un altissimo livello di astrazione. Dato che i casi d'uso astratti non hanno una sequenza degli eventi completa, non potranno mai essere eseguiti dal sistema. I casi d'uso astratti potrebbero avere, al posto della sequenza degli eventi, anche solo una semplice descrizione testuale del comportamento tipico che ci si aspetta che venga implementato nei casi d'uso specializzati che ne derivano.

Nell'UML i casi d'uso (come anche le classi) sono classificatori e hanno, quindi, attributi e operazioni (vedere il Paragrafo 1.8.3.1 per maggiori informazioni sui diversi tipi di classificatori). Ma perché l'UML modella i casi d'uso come dei classificatori? Ebbene, ciascun caso d'uso può essere visto come la descrizione di un insieme di istanze di caso d'uso, ciascuna delle quali corrisponde a un'esecuzione del caso d'uso in questione. A essere sinceri, si sta sfiorando un argomento dell'UML piuttosto dogmatico, che peraltro non ha alcuna rilevanza pratica. In effetti, si potrebbe facilmente argomentare che i casi d'uso non dovrebbero essere affatto modellati in UML come classificatori!

Caso d'uso specializzato: TrovaLibro	Caso d'uso specializzato: TrovaCD
ID: UC16	ID: UC17
ID caso d'uso generalizzato: UC12	ID caso d'uso generalizzato: UC12
Attori: Cliente	Attori: Cliente
Precondizioni:	Precondizioni:
Sequenza degli eventi:	Sequenza degli eventi:
<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Cliente seleziona "trova libro". 2. Il sistema chiede al Cliente i criteri di ricerca, ovvero nome dell'autore, titolo, codice ISBN o argomento. 3. Il Cliente inserisce i criteri di ricerca richiesti. 4. Il sistema ricerca i libri che soddisfano i criteri specificati dal Cliente. 5. Se il sistema trova uno o più libri <ol style="list-style-type: none"> 5.1. Il sistema mostra una pagina con i dati di non più di cinque libri. 5.2. Per ogni libro sulla pagina il sistema mostra il titolo, l'autore, il prezzo e il codice ISBN. 5.3. Fintantoché ci sono altri libri <ol style="list-style-type: none"> 5.3.1. Il sistema consente al Cliente di visualizzare la successiva pagina di libri. 6. Altrimenti <ol style="list-style-type: none"> 6.1. Il sistema visualizza nuovamente la pagina di ricerca "trova libro". 6.2. (6.1.) Il sistema comunica al cliente che non sono stati trovati prodotti che soddisfano i criteri specificati. 	<ol style="list-style-type: none"> 1. Il caso d'uso inizia quando il Cliente seleziona "trova CD". 2. Il sistema chiede al Cliente i criteri di ricerca, ovvero artista, titolo o genere musicale. 3. Il Cliente inserisce i criteri di ricerca richiesti. 4. Il sistema ricerca i CD che soddisfano i criteri specificati dal Cliente. 5. Se il sistema trova uno o più CD <ol style="list-style-type: none"> 5.1. Il sistema mostra una pagina con i dati di non più di dieci CD. 5.2. Per ogni CD sulla pagina il sistema mostra il titolo, l'artista, il prezzo e il genere musicale. 5.3. Fintantoché ci sono altri CD <ol style="list-style-type: none"> 5.3.1. Il sistema consente al Cliente di visualizzare la successiva pagina di CD. 6. Altrimenti <ol style="list-style-type: none"> 6.1. Il sistema visualizza nuovamente la pagina di ricerca "trova CD". 6.2. (6.1.) Il sistema comunica al cliente che non sono stati trovati prodotti che soddisfano i criteri specificati.
Postcondizioni:	Postcondizioni:
Sequenza alternativa:	Sequenza alternativa:
<ol style="list-style-type: none"> 1. In qualunque momento il Cliente può spostarsi su una pagina diversa. 	<ol style="list-style-type: none"> 1. In qualunque momento il Cliente può spostarsi su una pagina diversa.
Postcondizioni:	Postcondizioni:

Figura 5.6

I valori degli attributi di un caso d'uso rappresentano lo stato di un'istanza di caso d'uso, durante la sua esecuzione. Le operazioni rappresentano qualche frammento di lavoro che il caso d'uso può effettuare. Si ricordi: gli attributi e le operazioni dei casi d'uso non sembrano aggiungere alcun valore e, inoltre, non sono neanche supportati dalla maggior parte degli strumenti CASE. Secondo le specifiche dell'UML, le operazioni di un caso d'uso non sono neanche richiamabili dall'esterno, risulta quindi difficile immaginare perché mai siano state definite. Si consiglia semplicemente di ignorare questa caratteristica dei casi d'uso, fin quando non si chiarirà bene come debba essere utilizzata.

5.4 «include»

Scrivere i casi d'uso può diventare un'attività molto ripetitiva. Si supponga, per esempio, di modellare un sistema per il Personale (vedere Figura 5.7). A fronte di quasi qualunque operazione che chiederemo al sistema, sarà prima necessario individuare uno specifico impiegato sui cui dati operare. Se fosse necessario scrivere questa sequenza di eventi (autenticazione utente, inserimento degli estremi identificativi dell'im-

piegato ecc), ogni volta che serve reperire le informazioni di un impiegato, i diversi casi d'uso risulterebbero piuttosto ripetitivi. La relazione «include» tra casi d'uso consente di includere il comportamento di un caso d'uso in un punto della sequenza di eventi di un altro caso d'uso.

Il caso d'uso che include il comportamento è detto caso d'uso *cliente*. Il caso d'uso il cui comportamento è *incluso* viene detto caso d'uso *fornitore*. Questo perché il caso d'uso incluso fornisce comportamento al caso d'uso cliente.

Deve essere indicato il punto preciso del caso d'uso cliente dove è richiesta l'inclusione del comportamento del caso d'uso fornitore. La sintassi «include» è simile alla chiamata di una funzione, e in effetti anche il suo significato è molto simile.

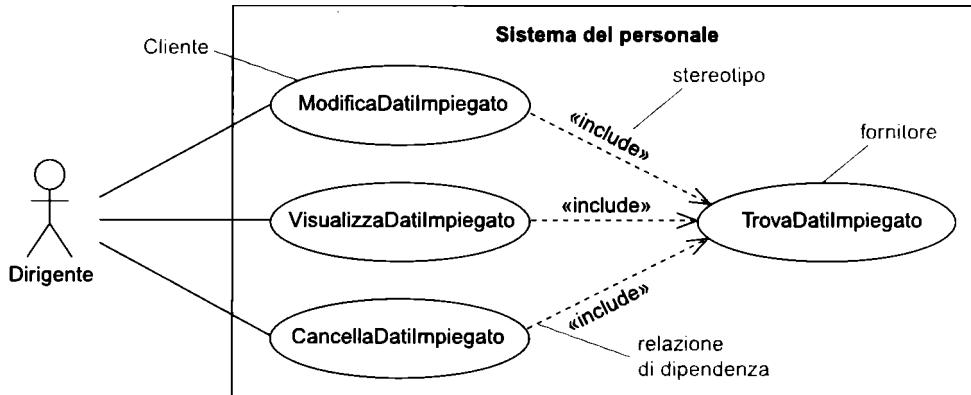


Figura 5.7

La semantica di una relazione «include» è piuttosto semplice (vedere Figura 5.8). Il caso d'uso cliente esegue la propria sequenza di eventi fino al punto di inclusione, quindi l'esecuzione passa al caso d'uso fornitore. Quando il caso d'uso fornitore ha terminato la propria esecuzione, il controllo torna nuovamente al caso d'uso cliente.

Il caso d'uso cliente non è completo senza tutti i suoi casi d'uso fornitori. Il caso d'uso fornitore è parte integrante del caso d'uso cliente. Tuttavia il caso d'uso fornitore può anche non essere completo. Questo succede se il caso d'uso fornitore contiene una sequenza di eventi parziale, e non ha un proprio significato se non viene incluso in un caso d'uso cliente predisposto. Ci si riferisce spesso a questo tipo di caso d'uso come a un frammento di comportamento. Si può anche dire che in questa situazione il

ModificaDatilImpiegato	VisualizzaDatilImpiegato	CancellaDatilImpiegato
ID: UC1	ID: UC2	ID: UC3
Attori: Dirigente	Attori: Dirigente	Attori: Dirigente
Precondizioni: 1. Il Dirigente è stato autenticato dal sistema.	Precondizioni: 1. Il Dirigente è stato autenticato dal sistema.	Precondizioni: 1. Il Dirigente è stato autenticato dal sistema.
Flow of events: 1. Il Manager inserisce l'ID dell'impiegato. 2. include('TrovadatilImpiegato'). 3. Il Dirigente seleziona la parte di dati dell'impiegato da modificare. 4. ...	Flow of events: 1. Il Manager inserisce l'ID dell'impiegato. 2. include('TrovadatilImpiegato'). 3. Il sistema mostra i dati dell'impiegato 4. ...	Flow of events: 1. Il Manager inserisce l'ID dell'impiegato. 2. include('TrovadatilImpiegato'). 3. Il sistema mostra i dati dell'impiegato 4. Il Dirigente cancella i dati dell'impiegato. 5. ...
Postcondizioni:	Postcondizioni:	Postcondizioni:

Figura 5.8

caso d'uso non è istanziabile: non può essere avviato direttamente da un attore, ma può essere esclusivamente eseguito quando viene incluso in un caso d'uso cliente. Se, invece, il caso d'uso fornitore è *di per sé completo*, allora si tratta di un normale caso d'uso ed è istanziabile. È, infatti, ragionevole che possa essere avviato da un attore.

5.5 «estende»

La relazione «estende» fornisce un modo di aggiungere comportamento a un caso d'uso esistente (vedere Figura 5.9). Il caso d'uso base mette a disposizione un insieme di punti di estensione, a ciascuno dei quali è possibile agganciare un nuovo comportamento. Il caso d'uso di estensione fornisce, invece, un insieme di segmenti inseribili che possono essere agganciati ai punti di estensione del caso d'uso base. La stessa relazione «estende» può essere utilizzata per specificare *esattamente* quali punti di estensione del caso d'uso base vengono estesi dal caso d'uso di estensione.

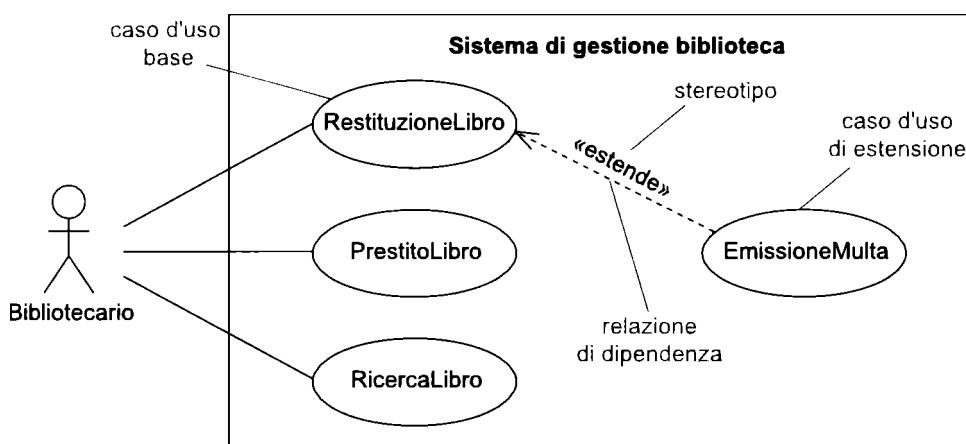


Figura 5.9

Ciò che rende interessante la relazione «estende» è che il caso d'uso base non sa assolutamente niente del caso d'uso di estensione: fornisce solo dei punti di inserzione di comportamento. In effetti, il caso d'uso base è del tutto completo anche senza le sue estensioni. Questo è *molto* diverso dalla relazione «include» in cui il caso d'uso cliente è incompleto senza i casi d'uso inclusi. L'altro elemento interessante è che i punti di estensione non fanno propriamente parte della sequenza di eventi del caso d'uso base, come un qualunque passo, ma vengono inseriti tra due passi successivi della sequenza.

I punti di estensione sono indicati nella sequenza di eventi del caso d'uso base come mostrato nella Figura 5.10. È, inoltre, possibile indicare i punti di estensione anche sul diagramma dei casi d'uso, elencandoli in una nuova sottosezione dell'icona del caso d'uso base.

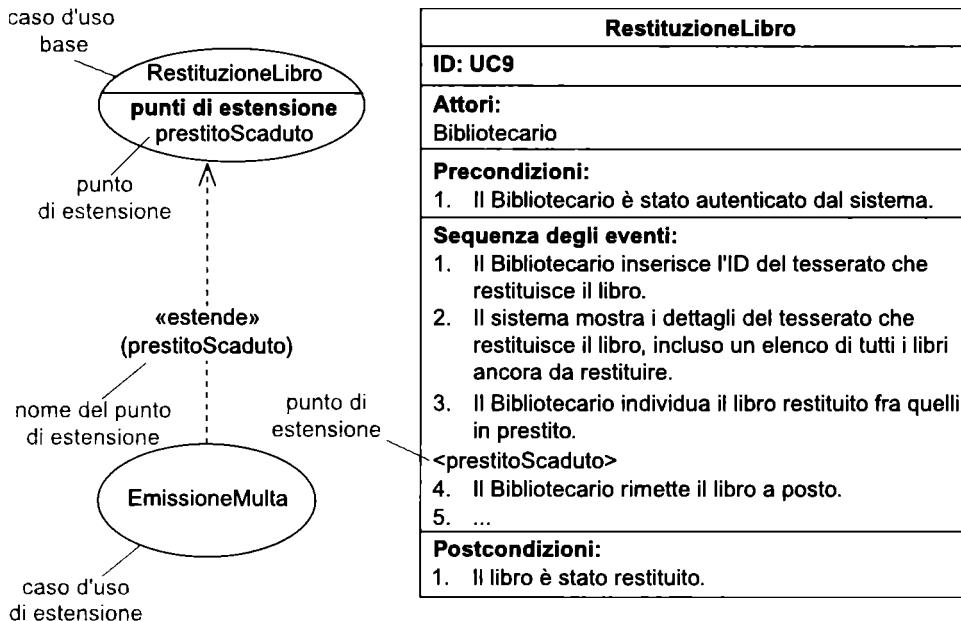


Figura 5.10

Si osservi che i punti di estensione presenti nella sequenza degli eventi non sono numerati. Vengono inseriti *tra* due passi successivi della sequenza.

In effetti l'UML afferma esplicitamente che i punti di estensione in realtà esistono su un piano distinto da quello della sequenza degli eventi, ma a esso sovrapponibile. Quindi non fanno affatto parte della sequenza degli eventi principale. È come se questi punti di estensione fossero disegnati su una foglio di acetato trasparente da sovrapporre al caso d'uso.

Lo scopo di questa precisazione è di rendere la sequenza del caso d'uso base del tutto indipendente dai punti di estensione. In altre parole, il caso d'uso base non sa, e non gli interessa sapere, se e dove altri casi d'uso lo estendono. Questo consente di utilizzare la relazione «estende» per creare estensioni arbitrarie e mirate alla sequenza di un caso d'uso base.

Con la relazione «estende», il caso d'uso base si comporta come un *framework* modulare in cui è possibile inserire estensioni in punti di estensione predefiniti.

Nell'esempio riportato nella Figura 5.10 si può vedere che il caso d'uso base **RestituzioneLibro** ha un punto di estensione chiamato **<prestitoScaduto>** tra il passo 3 e il passo 4 della sequenza di eventi.

Si può, quindi, capire come la relazione «estende» sia un buon modo di risolvere casi eccezionali o casi in cui non sia possibile conoscere a priori tutte le possibili estensioni.

5.5.1 Il caso d'uso di estensione

I casi d'uso di estensione *soltanamente* sono incompleti e non possono quindi essere istanziati. Sono tipicamente costituiti da uno o più frammenti di comportamento, chiamati segmenti inseribili. La relazione «estende» specifica il punto di estensione del caso d'uso base in cui verrà inserito il segmento inseribile. La relazione è soggetta alle seguenti regole.

- La relazione «estende» deve specificare uno o più punti di estensione del caso d'uso base; in caso contrario resta sottinteso che la relazione «estende» si riferisce a tutti i punti di estensione.
- Il caso d'uso di estensione deve avere tanti segmenti inseribili quanti punti di estensione sono stati specificati nella relazione «estende».
- È consentito avere due casi d'uso di estensione che "estendano" lo stesso punto di estensione dello stesso caso d'uso base, ma se succede, l'ordine in cui verranno eseguiti i due segmenti inseribili è indeterminato.

La Figura 5.11 mostra il caso d'uso di estensione **EmissioneMultà** che contiene un segmento inseribile.

È previsto che il caso d'uso di estensione possa avere precondizioni e postcondizioni. Le precondizioni devono essere soddisfatte, altrimenti il segmento inseribile non viene eseguito. Le postcondizioni rappresentano un vincolo sullo stato del sistema che deve essere vero quando termina l'esecuzione del segmento inseribile.

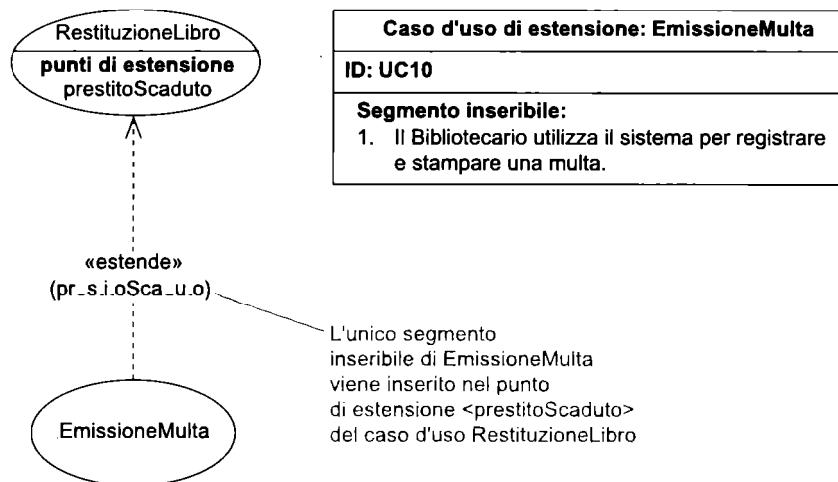


Figura 5.11

5.5.2 Segmenti inseribili multipli

Un caso d'uso d'estensione può contenere più di un segmento inseribile. Questo è utile quando non è possibile effettuare l'estensione in modo pulito, in un unico segmento

inseribile, perché è necessario intercalare alcuni passi del caso d'uso base tra quelli del segmento inseribile. Nell'esempio riportato nella Figura 5.12, si vede che dopo aver registrato e stampato una multa, il controllo torna alla sequenza di eventi del caso d'uso base per processare la riconsegna di eventuali altri libri in prestito scaduto, quindi, infine, al punto di estensione <pagamentoMultta>, viene gestito l'eventuale incasso complessivo di tutte le multe. Senza segmenti inseribili multipli sarebbe stato necessario combinare le operazioni di emissione, stampa e incasso della multa, con la conseguenza che ogni multa sarebbe stata gestita singolarmente... un processo che risulta sicuramente meno efficiente.

Nell'esempio è anche interessante notare che il secondo segmento inseribile inizia con una dichiarazione Se. Si tratta, quindi, di una sequenza condizionale e, come tale, risulta essere un buon candidato per diventare un caso d'uso base per una seconda estensione. Questo è accettabile: i casi d'uso di estensione possono a loro volta essere estesi o includere altri casi d'uso. In questo specifico caso abbiamo scelto di usare un condizionale, in quanto la logica non sembrava essere complessa al punto di richiedere la creazione di un ulteriore caso d'uso. Tuttavia, se avesse senso riusare il segmento inseribile del pagamento in altri punti del sistema, allora la situazione sarebbe diversa e forse sarebbe una buona strategia scorporare questo comportamento in un ulteriore caso d'uso di estensione. Bisogna stare però attenti: non è assolutamente una buona pratica avere troppe relazioni «include» ed «estende» in un modello dei casi d'uso.

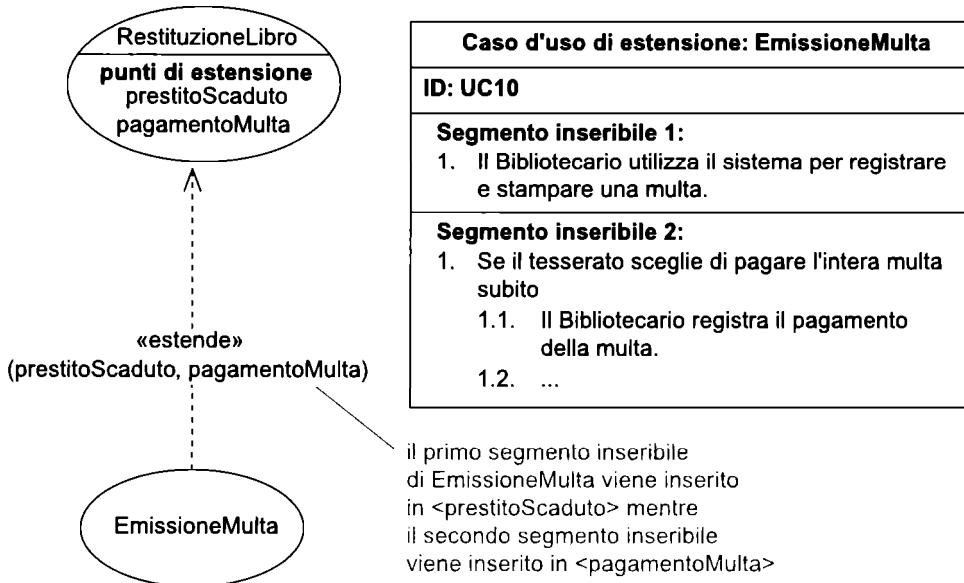


Figura 5.12

5.5.3 Estensioni condizionali

La Figura 5.13 riporta l'esempio di un sistema di gestione di una biblioteca leggermente meno aggressivo, in cui la prima volta che si restituisce un libro in ritardo si riceve solo un avviso, mentre dalla seconda volta in poi si viene multati. È possibile modellare questo sistema aggiungendo un nuovo caso d'uso di estensione EmissioneAvviso e assoggettando le relazioni «estende» a delle nuove condizioni. Le condizioni sono espressioni Booleane e ciascun segmento eseguibile verrà eseguito soltanto se la relativa espressione risulta vera.

Si osservi che il nuovo caso d'uso di estensione EmissioneAvviso estende soltanto il punto di estensione <prestitoScaduto>. D'altra parte (come nell'esempio precedente), il caso d'uso di estensione EmissioneMultta estende sia il punto <prestitoScaduto>, sia il punto <pagamentoMultta>. In questo modo è immediato capire che il caso d'uso EmissioneAvviso deve contenere esattamente un segmento inseribile, mentre il caso d'uso EmissioneMultta deve contenerne due (come già visto).

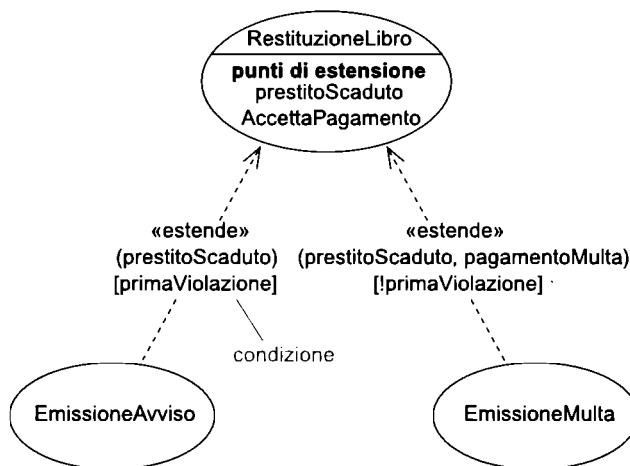


Figura 5.13

5.6 Quando usare le tecniche avanzate

Le tecniche avanzate descritte in questo capitolo devono essere utilizzate soltanto se semplificano il modello dei casi d'uso.

L'esperienza dimostra che i migliori modelli di casi d'uso sono semplici e immediati. È importante ricordarsi che il modello dei casi d'uso è una dichiarazione di requisiti e, in quanto tale, deve essere accessibile non solo ai modellatori, ma anche a tutte le parti interessate.

Un modello dei casi d'uso semplice che utilizza queste tecniche avanzate con parsimonia è, sotto ogni aspetto, preferibile a uno che ne utilizza troppe, anche se quest'ultimo, almeno per un modellatore, può risultare in qualche modo più elegante.

L'esperienza di modellazione di casi d'uso, maturata su molti e diversi progetti, insegna che:

- solitamente le parti interessate riescono a capire facilmente attori e casi d'uso, anche se è necessario un minimo di formazione;
- le parti interessate hanno maggiori difficoltà a capire la generalizzazione tra attori;
- un uso eccessivo di relazioni «include» può rendere i modelli dei casi d'uso difficili da comprendere: le parti interessate e i modellatori devono studiare più casi d'uso per avere un buon quadro d'insieme;
- le parti interessate difficilmente riescono a afferrare il concetto di relazione «estende»: questo anche a fronte di approfondite e chiare spiegazioni;
- anche un numero sorprendentemente elevato di modellatori non capisce bene la semantica della relazione «estende»;
- la generalizzazione tra casi d'uso andrebbe evitata, a meno che vengano utilizzati casi d'uso generalizzati astratti (e non concreti): in caso contrario è necessario aggiungere troppa complessità ai casi d'uso specializzati.

5.7 Riepilogo

Sono state spiegate tecniche avanzate di modellazione dei casi d'uso.

1. La generalizzazione tra attori consente di scorporare i comportamenti comuni a due o più attori, attribuendo tali comportamenti a un attore generalizzato genitore.
 - L'attore genitore è più generalizzato dei suoi figli e i figli sono più specializzati del loro genitore.
 - È possibile utilizzare un attore specializzato figlio ovunque sia richiesta la presenza del relativo attore generalizzato genitore: questo è il principio di sostituitività.
 - L'attore generalizzato è spesso astratto: definisce un ruolo astratto.
 - Gli attori specializzati sono concreti: definiscono ruoli concreti.
 - La generalizzazione tra attori può semplificare un diagramma dei casi d'uso.
2. La generalizzazione tra casi d'uso consente di scomporre le caratteristiche comuni a due o più casi d'uso, attribuendole a un caso d'uso generalizzato genitore.
 - I casi d'uso figli ereditano tutte le caratteristiche del proprio caso d'uso genitore (attori, relazioni, precondizioni, postcondizioni, sequenza degli eventi, sequenze alternative).
 - I casi d'uso figli possono aggiungere nuove caratteristiche.

- I casi d'uso figli possono ridefinire alcune caratteristiche ereditate:
 - le relazioni con attori o con altri casi d'uso possono essere ereditate o aggiunte;
 - le precondizioni e le postcondizioni possono essere ereditate, ridefinite o aggiunte;
 - i passi della sequenza degli eventi principale o delle sequenze alternative possono essere ereditati, ridefiniti o aggiunti;
 - gli attributi possono essere ereditati o aggiunti;
 - le operazioni possono essere ereditate, ridefinite o aggiunte.
 - Nei casi d'uso specializzati si usa una semplice convenzione tipografica:
 - caratteristiche ereditate e non ridefinite: testo normale;
 - caratteristiche ereditate e ridefinite: *testo corsivo*;
 - caratteristiche aggiunte: **testo grassetto**;
 - Il modello risulta più semplice e immediato se ci si limita all'uso di casi d'uso generalizzati astratti.
3. La relazione «include» consente di incapsulare passi ripetuti nelle sequenze di diversi casi d'uso in un caso d'uso separato che può essere quindi incluso dove serve.
- Nel caso d'uso si utilizza la sintassi **include** (nome del caso d'uso) per includere il comportamento di un altro caso d'uso in un passo specifico della sequenza degli eventi.
 - Il caso d'uso che include viene detto caso d'uso cliente.
 - Il caso d'uso incluso viene detto caso d'uso fornitore.
 - Il caso d'uso cliente è incompleto senza tutti i suoi casi d'uso fornitori.
 - I casi d'uso fornitori possono essere:
 - completi: sono casi d'uso normali e possono essere istanziati;
 - incompleti: contengono solo un frammento di comportamento e non possono essere istanziati.
4. La relazione «estende» aggiunge nuovo comportamento a un caso d'uso esistente (detto caso d'uso base).
- Il caso d'uso base contiene, su un piano sovrapponibile alla propria sequenza degli eventi, dei punti di estensione: i punti di estensione sono inseriti tra due passi successivi della sequenza degli eventi.
 - I casi d'uso di estensione forniscono dei segmenti inseribili: frammenti di compor-

tamento che possono essere agganciati ai punti di estensione del caso d'uso base.

- La relazione «estende» specifica a quali punti di estensione del caso d'uso base vengono agganciati i segmenti inseribili del caso d'uso di estensione.
- Il caso d'uso base è completo anche senza i segmenti inseribili forniti dai casi d'uso di estensione: questo caso d'uso non sa nulla dei potenziali segmenti inseribili, ma fornisce solo dei punti in cui questi possono essere inseriti.
- Il caso d'uso di estensione tipicamente è incompleto: di solito è costituito esclusivamente da uno o più segmenti inseribili; può anche trattarsi di un caso d'uso completo, ma è raro.
- Se il caso d'uso di estensione ha delle precondizioni, queste devono risultare vere altrimenti il caso d'uso d'estensione non verrà eseguito.
- Se il caso d'uso di estensione ha delle postcondizioni, queste vincolano lo stato del sistema al termine dell'esecuzione del caso d'uso di estensione.
- Un caso d'uso di estensione può contenere diversi segmenti inseribili.
- Due o più casi d'uso di estensione possono estendere lo stesso caso d'uso base, nello stesso punto di estensione; in questo caso resta tuttavia indeterminato l'ordine di esecuzione dei singoli segmenti inseribili.
- Estensioni condizionali: condizioni di guardia Booleana specificate sulla relazione «estende» possono consentire l'esecuzione del segmento inseribile quando risultano vere o impedirla quando risultano false.

5. Linee guida per l'uso delle tecniche avanzate di modellazione:

- generalizzazione tra attori: da usare solo se semplifica il modello;
- generalizzazione tra casi d'uso: meglio *non* usarla, oppure usarla solo con casi d'uso generalizzati astratti;
- relazione «include»: da usare solo se semplifica il modello; attenzione all'uso eccessivo, che può trasformare il modello in una scomposizione funzionale del sistema;
- relazione «estende»: meglio *non* usarla; prima di usarla è meglio accertarsi che tutti i modellatori e le parti interessate esposte al modello ne comprendano correttamente l'uso e il significato.

L'obiettivo deve sempre essere quello di produrre un modello dei casi d'uso semplice e immediato che fissa le informazioni necessarie nel modo più chiaro e conciso possibile. È spesso preferibile un modello dei casi d'uso che non sfrutta nessuna delle tecniche avanzate esposte in questo capitolo, piuttosto di uno così ricco di generalizzazioni, relazioni «include» ed «estende» da non riuscire a capire che cosa dovrebbe fare il sistema. Una buona regola pratica in questo caso è: “nel dubbio, non metterlo nel modello”.

Parte 3

Analisi

Il flusso di lavoro dell'analisi

6.1 Contenuto del capitolo

In questo capitolo si inizia a esplorare il processo dell'analisi OO. Vengono brevemente introdotti il flusso di lavoro dell'analisi e alcune "regole pratiche" dei modelli dell'analisi, per porre le basi della discussione più dettagliata che verrà svolta nei successivi capitoli di questa parte del libro.

6.2 Il flusso di lavoro dell'analisi

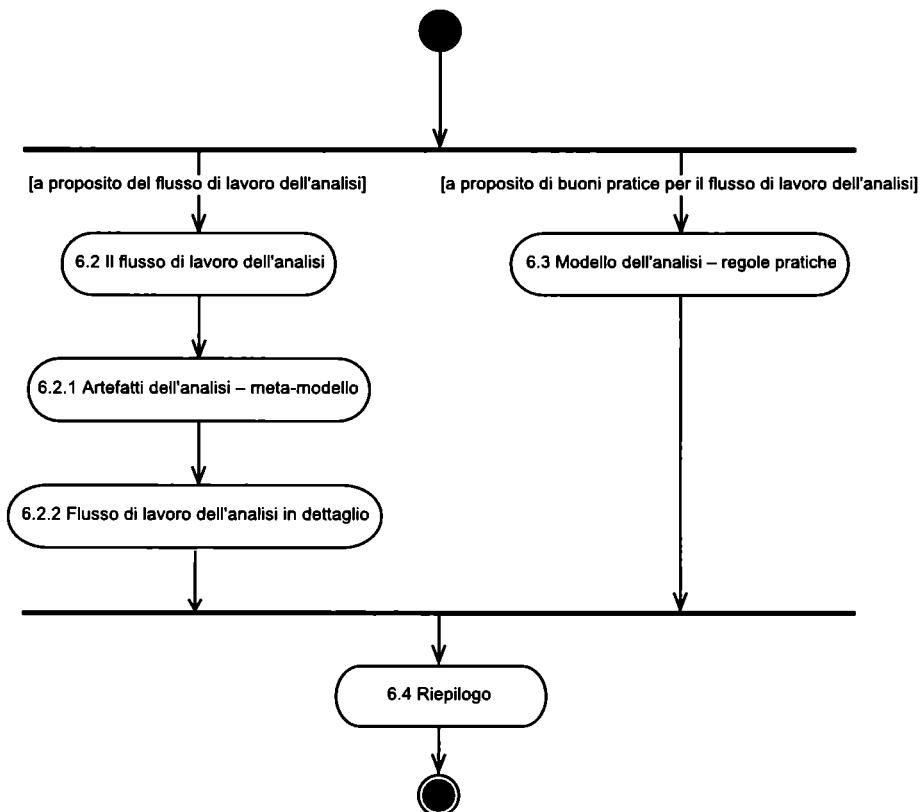
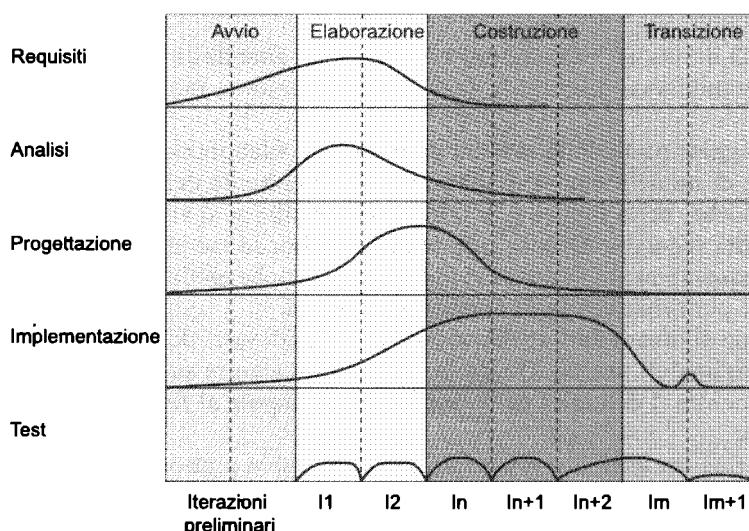
Le attività di analisi entrano nel vivo verso la fine della fase di Avvio, e diventano preponderanti durante la fase di Elaborazione, insieme con le attività sui requisiti.

La maggior parte delle attività della fase di Elaborazione sono finalizzate alla creazione di modelli che descrivono il comportamento desiderato del sistema.

Si noti che, nella Figura 6.2, il lavoro di analisi si sovrappone estesamente alla raccolta dei requisiti. Difatti, queste attività procedono spesso mano nella mano: capita frequentemente di dover svolgere una qualche analisi dei propri requisiti per chiarirli e per scoprire quelli mancanti o distorti.

Il flusso di lavoro dell'analisi mira, dal punto di vista dell'analista OO, a produrre un modello dell'analisi. Questo modello si concentra su *che cosa* il sistema debba fare, mentre il dettaglio del *come* lo farà, compete al flusso di lavoro della progettazione.

Il confine tra analisi e progettazione può essere piuttosto sfumato e, fino a un certo punto, tocca all'analista porre la linea di confine dove lo ritiene opportuno. Il Paragrafo 6.3 illustra alcune regole pratiche che possono servire per produrre modelli dell'analisi efficaci.

**Figura 6.1****Figura 6.2** Adattata dalla Figura 1.5 [Jacobson 1], con il consenso della Addison-Wesley.

6.2.1 Artefatti dell'analisi: meta-modello

Nel flusso di lavoro dell'analisi vengono prodotti due artefatti cardine:

- classi di analisi: queste modellano i concetti chiave del dominio del problema;
- realizzazioni di caso d'uso: mostrano come le istanze delle classi di analisi possono interagire per realizzare il comportamento del sistema specificato in un caso d'uso.

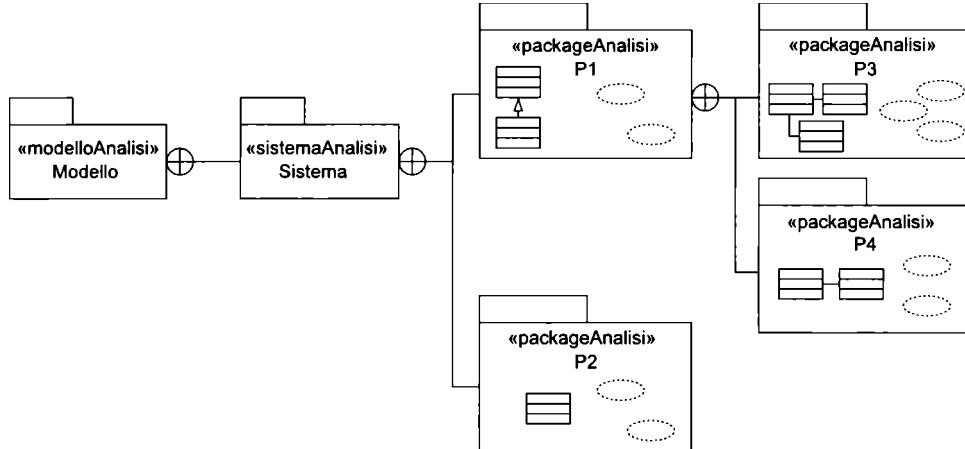


Figura 6.3

È possibile creare un modello UML del modello dell'analisi stesso. La Figura 6.3 mostra questo meta-modello.

Si è già considerata la sintassi per le classi, ma la sintassi per i *package* (le entità raffigurate con le cartelline) è una novità. Per ora, si pensi al *package* come a un contenitore di diagrammi e di elementi di modellazione UML. Ogni elemento (o diagramma) appartiene a un unico *package*.

Il modello dell'analisi è una collezione di diagrammi e di elementi di modellazione, si può, quindi, modellarlo come un *package* con stereotipo "modelloAnalisi". Questo *package* contiene esattamente un solo sistema dell'analisi (che viene modellato come un *package* con stereotipo "sistemaAnalisi") e questo sistema è composto da uno o più *package* di analisi.

La Figura 6.3 riporta quattro *package* di analisi. In generale, il sistema dell'analisi può contenere molti *package* di analisi che possono, a loro volta, contenere altri *package* di analisi annidati.

6.2.2 Flusso di lavoro dell'analisi in dettaglio

La Figura 6.4 rappresenta il flusso di lavoro dell'analisi previsto dall'UP. Le attività rilevanti verranno approfondite più avanti nel libro, ma prima è necessario comprendere le classi e gli oggetti. Questo argomento viene trattato nel Capitolo 7.

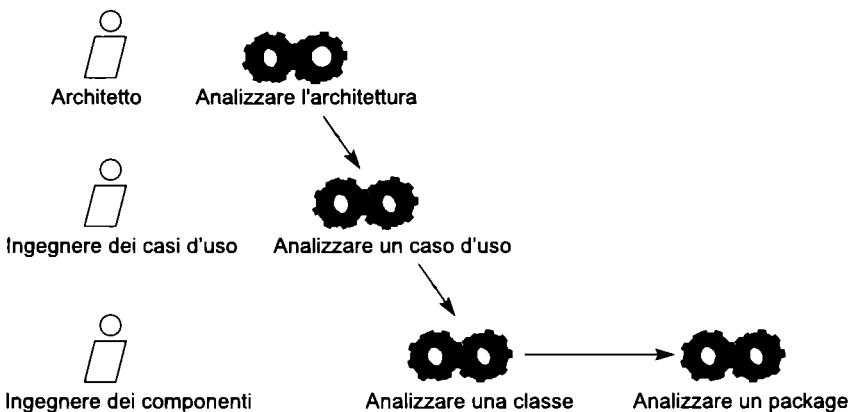


Figura 6.4 Riprodotta da Figura 8.18 [Jacobson 1], con il consenso della Addison-Wesley.

6.3 Modello dell'analisi: regole pratiche

Ogni sistema è diverso ed è quindi difficile generalizzare sui modelli dell'analisi. Comunque, per un sistema di medie dimensioni e complessità, il modello dell'analisi può contenere tra le 50 e le 100 classi.

Si rammenti che, quando si costruisce il modello dell'analisi, è d'importanza vitale limitarsi *esclusivamente* a quelle classi che fanno parte del vocabolario del dominio del problema. Si deve sempre evitare la tentazione di aggiungere classi di progettazione (come quelle per le telecomunicazioni o per l'accesso ai *databases*) nel modello dell'analisi (a meno che il problema non riguardi proprio un *database* o le telecomunicazioni!). Ci si autolimita in questa maniera per fare del modello dell'analisi una stipulazione semplice e concisa della struttura e del comportamento del sistema. Tutte le decisioni implementative devono essere lasciate ai flussi di lavoro della progettazione e dell'implementazione.

Di seguito sono elencate alcune regole pratiche per l'efficace modellazione dell'analisi.

- Esprimere *sempre* il modello dell'analisi nel linguaggio del problema. Le astrazioni trovate nel modello dell'analisi devono sempre far parte del vocabolario del dominio del problema.
- Creare modelli che “raccontano una storia”. Ogni diagramma dovrebbe esporre una qualche parte importante del comportamento richiesto al sistema. Se non lo facesse, a che servirebbe? Vedremo alcune buone tecniche per la creazione di questi diagrammi studiando le realizzazioni di caso d'uso.
- Concentrarsi sulla descrizione della vista d'insieme. Non perdersi nei dettagli di come il sistema svolgerà il suo lavoro: ci sarà tempo per occuparsene nella fase di progettazione.
- Mantenere una netta distinzione tra il dominio del problema (i requisiti del *business*) e il dominio delle soluzioni (considerazioni sui dettagli di progettazione).

Concentrarsi sempre sulle astrazioni che esistono nel dominio applicativo. Per esempio, nella modellazione di un sistema per il commercio elettronico, ci si aspetta che il modello contenga classi quali: Cliente, Ordine e CarrelloSpesa. Ci si sorprenderebbe a trovarvi classi per l'accesso ai *databases*, o per telecomunicazioni, le quali sono artefatti che ovviamente nascono dal dominio delle soluzioni.

- Impegnarsi sempre per minimizzare le interdipendenze. Ogni associazione tra due classi crea delle interdipendenze tra loro. Il Capitolo 9 illustra come usare le molteplicità e la navigabilità per ridurre queste interdipendenze.
- Esplorare l'uso dell'ereditarietà, se sembra che esista un'ovvia e naturale gerarchia tra le astrazioni. Nell'analisi non si applica mai l'ereditarietà solo per riutilizzare il codice. Come si vedrà nel Paragrafo 15.5, l'ereditarietà è la forma più forte di interdipendenza tra classi.
- Chiedersi sempre: “questo modello sarà utile a tutte le parti interessate?”. Non c'è nulla di peggio del produrre un modello dell'analisi che venga ignorato dai committenti, oppure dai progettisti o dagli sviluppatori. Eppure questo si verifica anche troppo spesso, solitamente a analisti poco esperti. Le strategie chiave con cui cautelarsi sono: conferire la massima visibilità al modello e alle attività dell'analisi, coinvolgere le parti interessate nel processo ogni qualvolta possibile, e indire frequenti riunioni aperte a tutti.

Infine, mantenere semplice il modello! Naturalmente, questo è più facile a dirsi che a farsi, ma l'esperienza insegna che dentro a un modello dell'analisi complesso si nasconde sempre un modello dell'analisi più semplice che fatica a venire a galla. Un modo per semplificare consiste nel guardare al caso generale piuttosto che a quelli particolari. A questo proposito, un progetto recentemente valutato prevedeva modelli completamente distinti per la vendita di biglietti, la prenotazione di alberghi e l'autonoleggio. Chiaramente, quel modello conteneva implicitamente un “sistema di vendita” che avrebbe potuto gestire tutti i casi con una combinazione abbastanza semplice di ereditarietà e polimorfismo.

6.4 Riepilogo

1. L'analisi consiste nel creare modelli che fissano i requisiti e le caratteristiche essenziali del sistema richiesto: la modellazione dell'analisi è strategica.
2. La maggior parte delle attività del flusso di lavoro dell'analisi vengono eseguite verso la fine della fase di Avvio e durante tutta la fase di Elaborazione.
3. I flussi di lavoro dell'analisi e dei requisiti si sovrappongono, soprattutto nella fase di Elaborazione; spesso conviene analizzare i requisiti mentre vengono individuati per scoprire se vi siano distorsioni oppure omissioni.
4. Il modello dell'analisi:
 - è sempre espresso nel linguaggio del dominio del problema;
 - descrive la vista d'insieme;

- contiene artefatti che modellano il dominio del problema;
 - racconta una storia concernente il sistema richiesto;
 - è utile a quante più parti interessate possibile.
5. Gli artefatti dell'analisi sono:
- classi di analisi: modellano i concetti chiave del dominio del problema;
 - realizzazioni di caso d'uso: illustrano come le istanze delle classi di analisi possono interagire per realizzare il comportamento del sistema specificato da un caso d'uso.
6. Il flusso di lavoro dell'analisi dell'UP comprende queste attività:
- analizzare l'architettura;
 - analizzare un caso d'uso;
 - analizzare una classe;
 - analizzare un *package*.
7. Modello dell'analisi: regole pratiche:
- il modello dell'analisi di un sistema medio dovrebbe contenere tra le 50 e le 100 classi di analisi;
 - includere solo le classi che modellano il vocabolario del dominio del problema;
 - non prendere decisioni implementative;
 - concentrarsi su classi e associazioni: minimizzare le interdipendenze;
 - usare l'ereditarietà dove vi sia un'ovvia gerarchia di astrazioni;
 - semplicità!

Classi e oggetti

7.1 Contenuto del capitolo

Il capitolo è dedicato agli oggetti e alle classi. Questi sono i gli elementi costitutivi fondamentali dei sistemi OO. Chi ha già esperienza con i concetti di oggetto e di classe, può magari saltare le Sezioni 7.2 e 7.4. Potrebbe comunque risultare interessante la notazione UML per gli oggetti (Sezione 7.3) e per le classi (Sezione 7.5).

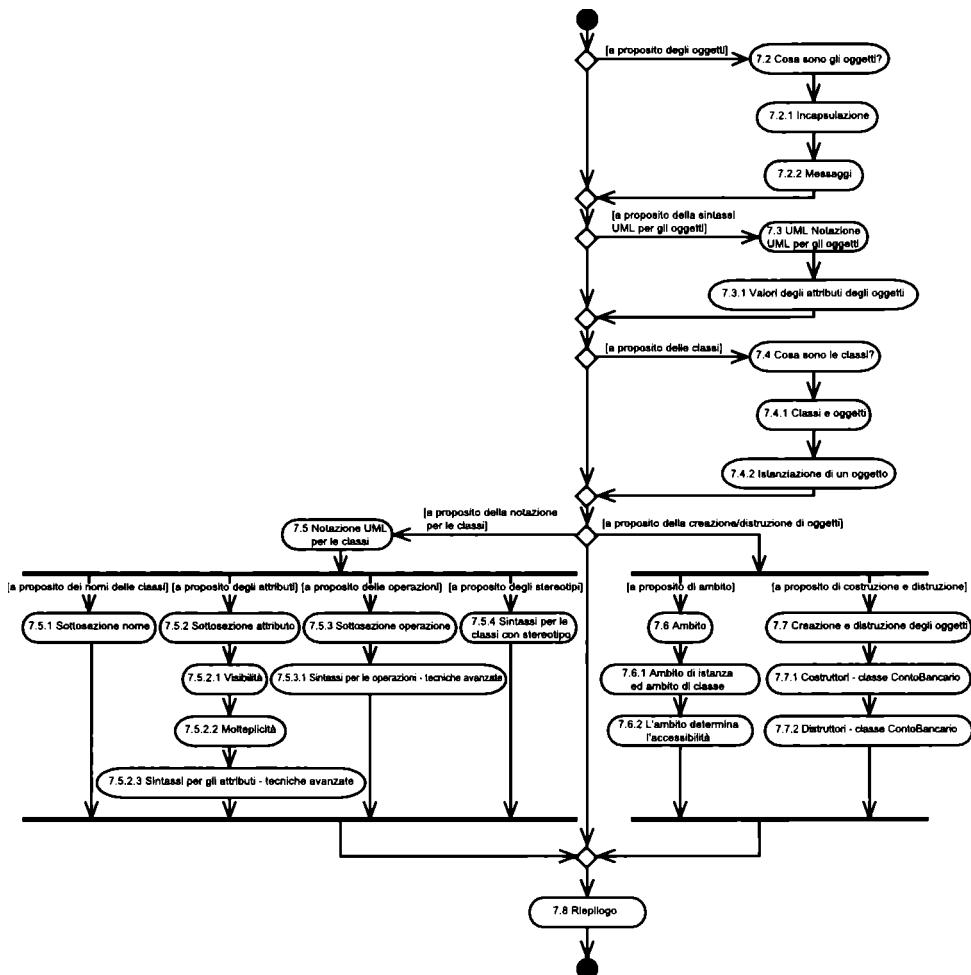
Il capitolo conclude con la discussione di due argomenti di interesse: la visibilità di metodi e attributi (Sezione 7.6) e la costruzione e distruzione degli oggetti (Sezione 7.7.).

7.2 Cosa sono gli oggetti?

The UML Reference Manual [Rumbaugh 1] definisce un oggetto come “un’entità discreta, con confini ben definiti che incapsula stato e comportamento; un’istanza di una classe”.

Si pensi a un oggetto come a un pacchetto coeso di dati e funzioni. In generale, l’unico modo di ottenere accesso alla parte dati di un oggetto consiste nel chiamare una delle funzioni rese disponibili dall’oggetto stesso. Nell’analisi ci si riferisce a queste funzioni come *operazioni*, mentre nella progettazione le si chiama *metodi*. Questa distinzione evidenzia che nell’analisi si descrive la specifica astratta di una funzione (un’operazione), mentre nella progettazione si definisce la effettiva, materiale implementazione di una funzione (un metodo). Il mascherare la parte dati di un oggetto sotto uno strato di funzioni si chiama *incapsulazione* ovvero *opacità* dei dati. L’incapsulazione non è imposta da UML, perché alcuni linguaggi di programmazione non la richiedono. Tuttavia, nascondere la parte dati dell’oggetto dietro le sue operazioni o metodi, è una buona pratica di programmazione OO.

Ogni oggetto è di per sé un’istanza di una classe che definisce l’insieme di caratteristiche (attributi e operazioni/metodi) comuni a tutte le istanze di quella classe.

**Figura 7.1**

Le idee di classe e classificazione sono in fondo semplici. Si pensi a una stampante di modello “Epson Photo 1200”. Indicando il modello si descrivono le proprietà di tutte le istanze specifiche della classe, come la particolare “Epson Photo 1200 S/N 34120098” che si trova sulla nostra scrivania. Un’istanza specifica di una classe si chiama oggetto.

Pensando ancora all’esempio della stampante Epson, si notano certe proprietà comuni a tutti gli oggetti.

- **Identità:** questa è la peculiare esistenza dell’oggetto nel tempo e nello spazio, ciò che lo distingue da tutti gli altri oggetti. Nell’esempio, il numero di serie della stampante sulla nostra scrivania può servire da identificatore per questa particolare stampante e rappresentarne l’identità. Un numero di serie specifica perfettamente l’unicità di un oggetto materiale.

Anche nell'analisi e progettazione OO si utilizza un principio simile: l'idea di un *riferimento* a un oggetto, per specificare l'identità di ogni oggetto software che si descrive. Sebbene nel mondo reale non tutti gli oggetti abbiano un numero di serie, essi hanno sempre delle identità distinte grazie alle loro particolari coordinate temporali e spaziali. Analogamente, nei sistemi software OO ogni oggetto ha un qualche tipo di riferimento.

- Stato: questo viene stabilito dai valori degli attributi di un oggetto in un dato istante. La Tabella 7.1 dà una lista parziale degli stati in cui si può trovare la stampante. Si osservi come lo stato di un oggetto dipenda dai valori dei suoi attributi.
- Comportamento: ci sono cose che la stampante può fare:

```
accendi()
spegni()
stampaDocumento()
avanzamentoPagina()
puliziaTestina()
cambiaCartucciaInchiostro()
```

Questi comportamenti sono descritti come operazioni durante l'analisi e come metodi durante la progettazione. L'esecuzione di un'operazione o di un metodo di un oggetto provoca spesso una variazione di valore di uno o più dei suoi attributi, cosa che potrebbe risultare in una transizione di stato. Quest'ultima è uno spostamento significativo dell'oggetto da uno stato a un altro. Riferendosi ancora alla Tabella 7.1, risulta evidente che lo stato di un oggetto può anche influenzarne il comportamento.

Per esempio, se la stampante finisce l'inchiostro (stato dell'oggetto = FineInchiostroNero), allora la chiamata alla funzione stampaDocumento() provocherà una segnalazione d'errore. Di conseguenza, il comportamento di stampaDocumento() *dipende dallo stato*.

Tabella 7.1

Stato dell'oggetto	Attributo della classe	Valore dell'attributo dell'oggetto
On	alimentazione	On
Off	alimentazione	Off
FineInchiostroNero	cartucciaNero	vuoto
FineInchistroColore	cartucciaColore	vuoto

7.2.1 Incapsulazione

Come già detto, l'identità di un oggetto è una forma di identificatore univoco stabilito dal linguaggio d'implementazione, normalmente un indirizzo di memoria. D'ora in avanti, nell'analisi OO questi identificatori verranno sempre detti riferimento all'oggetto, senza preoccuparsi di come essi siano realizzati: è sufficiente presupporre che ogni ogget-

to abbia una propria identità gestita dalla tecnologia utilizzata per l'implementazione. È possibile che durante la progettazione si debba tener conto della implementazione dei riferimenti agli oggetti, qualora si intenda utilizzare un linguaggio OO, come il C++, che consente di gestire direttamente tipi particolari di riferimento, detti puntatori.

La Figura 7.2 mostra la rappresentazione concettuale di un oggetto enfatizzandone l'incapsulazione. Si osservi che la Figura 7.2 *non è* un diagramma UML. Vedremo, tra poco, quale sia la sintassi UML per gli oggetti.

Lo stato dell'oggetto è l'insieme dei valori degli attributi (in questo caso 1234567801, "Jim Arlow", 300.00) posseduti dall'oggetto in un dato istante. Tipicamente, alcuni di questi valori cambieranno nel tempo, mentre altri rimarranno invariati. Per esempio, è molto improbabile che il nome e il numero di conto cambino nel tempo, ma si spera che il saldo aumenti costantemente!

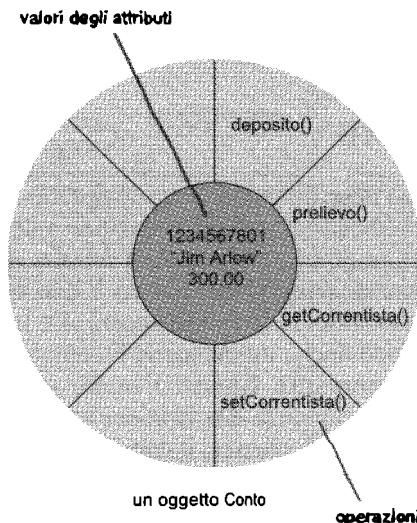


Figura 7.2

Si vede come, al variare del saldo nel tempo, possa cambiare anche lo stato dell'oggetto. Per esempio, se il saldo è negativo, allora si può dire che l'oggetto è nello stato scoperto. Al cambiare del saldo da negativo a zero, l'oggetto compie un mutamento significativo della sua natura: passa da scoperto a esaurito. Inoltre, quando il saldo dell'oggetto Conto diventa positivo, esso compie un'altra transizione dallo stato esaurito allo stato solvibile. L'insieme delle transizioni possibili può essere molto più ampio di quello descritto. Infatti, ogni chiamata di un metodo che generi una modifica sostanziale della natura dell'oggetto crea una transizione di stato. L'UML fornisce un potente insieme di tecniche di modellazione per descrivere i passaggi di stato, chiamate diagrammi di stato, che saranno trattati nel Capitolo 19.

Fondamentalmente, il comportamento di un oggetto rappresenta "ciò che può fare per noi". L'oggetto della Figura 7.2 mette a disposizione le operazioni elencate nella Tabella 7.2.

Questo insieme di operazioni specifica il comportamento dell'oggetto. Si osservi che l'esecuzione di alcune di queste operazioni (`deposito()`, `prelievo()`, `setCorrentista()`) modifica i valori di qualche attributo, e può generare delle transizioni da uno stato a un altro. L'altra operazione (`getCorrentista()`) non cambia il valore di alcun attributo e quindi non scatena alcuna transizione di stato.

Tabella 7.2

Operazione	Semantica
<code>deposito()</code>	Deposita del denaro nell'oggetto Conto Incrementa il valore dell'attributo Saldo
<code>prelievo()</code>	Preleva del denaro dall'oggetto Conto Decrementa il valore dell'attributo Saldo
<code>getCorrentista()</code>	Restituisce il correntista dell'oggetto Conto
<code>setCorrentista()</code>	Cambia il correntista dell'oggetto Conto

L'incapsulazione oppure opacità dei dati, è uno dei vantaggi più importanti della programmazione OO e può consentire la creazione di software più robusto e affidabile. In questo semplice esempio, un utilizzatore dell'oggetto Conto non deve preoccuparsi della struttura dei dati nascosti all'interno dell'oggetto, ma solo di ciò che l'oggetto può fare: ovvero, dei *servizi* che l'oggetto offre ad altri oggetti.

7.2.2 Messaggi

Gli oggetti hanno attributi che contengono valori e un comportamento, ma come si combinano gli oggetti per creare dei sistemi? Gli oggetti collaborano per eseguire le funzioni del sistema, stabilendo dei collegamenti tra loro e scambiandosi messaggi lungo questi collegamenti. Quando un oggetto riceve un messaggio, esso esamina le sue operazioni (o metodi, se si parla di progettazione) per trovarne una la cui segnatura corrisponda alla segnatura del messaggio. Se tale operazione esiste, allora l'oggetto la esegue (vedere la Figura 7.3). Queste segnature sono composte dal nome del messaggio (o dell'operazione), dai tipi dei parametri e dal valore restituito.

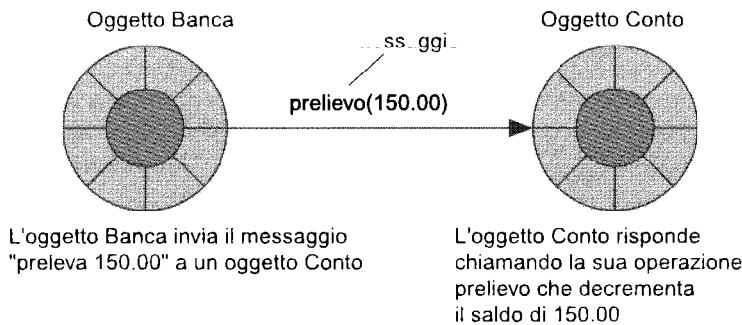


Figura 7.3

La struttura di *run-time* di un sistema OO è costituita da numerosi oggetti che vengono creati, esistono per un certo tempo e poi, forse, vengono distrutti. Questi oggetti si scambiano messaggi per utilizzare i rispettivi servizi. Questa struttura è profondamente diversa dal quella dei sistemi di software procedurale che evolvono nel tempo applicando sequenzialmente funzioni ai dati.

7.3 Notazione UML per gli oggetti

L'icona di un oggetto dell'UML è un rettangolo con due sottosezioni, di cui si vede un esempio nella Figura 7.4. La sottosezione superiore contiene l'identificatore dell'oggetto che è *sempre sottolineato*. Questo è molto importante, perché la notazione UML per le classi assomiglia molto a quella per gli oggetti. Utilizzando rigorosamente la sottolineatura si evita ogni confusione relativa al fatto che un elemento del modello sia una classe oppure un oggetto.

L'UML consente di rappresentare un oggetto, in un diagramma degli oggetti, in modo molto flessibile. L'identificatore di un oggetto può essere uno qualunque dei seguenti:

- Il solo nome della classe, per esempio: Conto. Questo indica un oggetto anonimo, ovvero una qualunque istanza di quella classe (vale a dire, si sta esaminando un'istanza di un Conto, ma non si vuole o non interessa, specificare di quale istanza si tratti). Gli oggetti anonimi si usano spesso quando in un diagramma è presente un solo oggetto di una data classe. Se occorre mostrare due oggetti della stessa classe, è meglio distinguerli assegnando loro un nome.
- Il solo nome dell'oggetto, per esempio contoJim. Questo identifica un oggetto specifico, ma non dice a quale classe esso appartenga. Questa notazione è utile per un'analisi molto preliminare, quando non siano ancora state individuate tutte le classi.
- Il nome dell'oggetto, concatenato al nome della classe, separati dai due punti. Si possono leggere i due punti come: "che è un'istanza della classe". Così, il diagramma della Figura 7.4 si leggerebbe come segue: "esiste un oggetto di nome contoJim che è un'istanza della classe Conto".

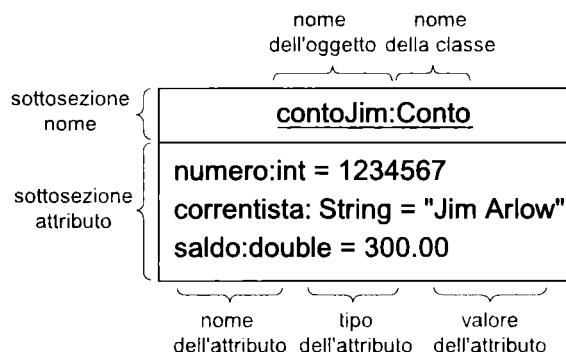


Figura 7.4

Normalmente i nomi degli oggetti sono scritti con lettere maiuscole e minuscole, ma iniziando sempre con una lettera minuscola. Si devono evitare caratteri speciali, come lo spazio e il segno di sottolineatura. Questo modo di scrivere viene detto “CamelCase”, perché produce parole con le gobbe, proprio come un “carmel”: cammello.

Dalla Sezione 7.2 sappiamo che una classe definisce gli attributi e le operazioni di un insieme di oggetti. Poiché tutti gli oggetti hanno le stesse operazioni, queste sono elencate nell’icona della sola classe, ma non nell’icona dell’oggetto.

Non è obbligatorio mostrare gli attributi nella sottosezione inferiore dell’icona di un oggetto, ma quelli mostrati devono avere un nome e possono avere un tipo e un valore opzionali. Anche i nomi degli attributi vengono scritti in “CamelCase”, ma iniziano con una lettera minuscola.

7.3.1 Valori degli attributi degli oggetti

Ogni valore di un attributo di un oggetto ha questo formato:

nome : tipo = valore

Si può scegliere di mostrare tutti, solo alcuni, o nessuno dei valori degli attributi di un oggetto, a seconda dello scopo del diagramma.

Per mantenere i diagrammi semplici e chiari, si può voler omettere i tipi degli attributi, visto che sono già definiti nella classe dell’oggetto. Il Capitolo 12 illustra come si usano i diagrammi degli oggetti nell’analisi e chiarisce perché si possa voler mostrare soltanto parte delle informazioni dell’oggetto nella sua icona.

7.4 Cosa sono le classi?

The UML Reference Manual [Rumbaugh 1] definisce una classe come: “Il descrittore di un insieme di oggetti che condividono gli stessi attributi, operazioni, metodi, relazioni e comportamento”. Lo si può riassumere dicendo che una classe descrive un insieme di oggetti che hanno le stesse caratteristiche.

Ogni oggetto è un’istanza di esattamente una sola classe. Ecco alcuni modi utili di pensare alle classi.

- Si pensi a una classe come a un formato per gli oggetti: una classe definisce la struttura (l’insieme delle caratteristiche) di tutti gli oggetti di quella classe. Tutti gli oggetti della stessa classe devono avere *lo stesso* insieme di operazioni, *lo stesso* insieme di attributi e *lo stesso* insieme di relazioni, ma possono avere valori *diversi* degli attributi.
- Si pensi a una classe come a un timbro a inchiostro, gli oggetti sono allora le singole timbrature su di un foglio di carta. O si immagini una classe come una formella per fare i biscotti: gli oggetti sono i biscotti.

Il classificatore e l’istanza sono una delle distinzioni comuni dell’UML (vedere il Capitolo 1) e gli esempi più comuni di questa distinzione sono la classe e l’oggetto.

Una classe è una specifica o formato a cui devono corrispondere tutti gli oggetti (le istanze) della classe. Ogni oggetto della classe ha specifici valori per gli attributi definiti dalla classe e risponderà ai messaggi attivando le operazioni definite dalla classe.

A seconda del loro stato, oggetti diversi possono rispondere diversamente allo stesso messaggio. Per esempio, se si tenta di prelevare €100 da un oggetto ContoBancario che è già in scoperto, questo darà un risultato diverso dal richiedere un prelievo di €100 da un conto con una disponibilità di diverse centinaia di euro.

La classificazione potrebbe essere il singolo mezzo più importante di cui dispongono gli esseri umani per organizzare la conoscenza del mondo. Come tale, essa è anche uno dei concetti OO più importanti. Utilizzando il concetto di classe si può parlare di un particolare tipo di automobile o di un tipo di albero, senza mai menzionarne un'istanza specifica. Lo stesso avviene con il software. Le classi consentono di descrivere l'insieme di caratteristiche che tutti questi oggetti *devono* possedere, senza che si debba descrivere ogni singolo oggetto.

Si osservi la Figura 7.5, ragionando sulle classi per un minuto o due. Quante classi ci sono nella figura?

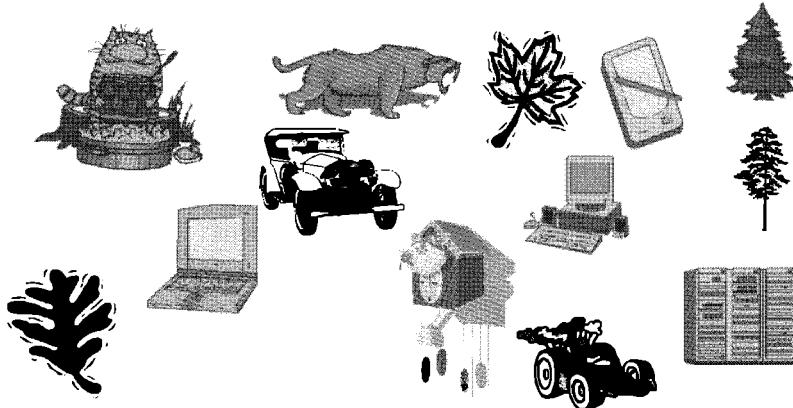


Figura 7.5

In realtà, a questa domanda non c'è risposta! Ci sono un numero quasi infinito di modi per classificare gli oggetti del mondo reale. Alcune delle classi individuabili sono:

- la classe dei gatti;
- la classe dei gatti golosi (esiste un gatto che è un'istanza proprio di questa classe!);
- la classe degli alberi;
- la classe delle foglie;
- ecc.

Avendo così tante possibilità, uno degli aspetti più importanti dell'analisi e progettazione OO, è la scelta dello schema di classificazione più appropriato, come si vedrà nel Capitolo 8.

Esaminando molto attentamente la Figura 7.5, si iniziano a vedere altri tipi di relazioni, oltre a quelle di classificatore e istanza. Per esempio, si delineano diversi livelli di classificazione. Avendo la classe dei gatti, li si può raggruppare ulteriormente nelle sottoclassi "gatto domestico" e "gatto selvatico", oppure anche nelle sottoclassi "gatto moderno" e "gatto preistorico". Questa è una relazione tra classi: l'una è la sottoclasse dell'altra. Di converso, la classe "gatto" è la superclasse di "gatto domestico" e di "gatto selvatico". Si potranno approfondire questi concetti studiando l'ereditarietà, nel Capitolo 10.

Inoltre, considerando gli alberi e le foglie della Figura 7.5, si vede che gli oggetti albero possiedono ognuno un gruppo di oggetti foglia. La relazione esistente tra alberi e foglie è di un tipo molto forte. Ogni oggetto foglia appartiene a un oggetto albero specifico e non può essere scambiato o condiviso con altri alberi; il ciclo di vita di una foglia è strettamente vincolato al proprio albero e controllato da esso. Nell'UML, questa relazione tra oggetti è nota come composizione.

D'altra parte, se prendiamo in considerazione la relazione tra computer e periferiche, si tratta di una relazione molto diversa. Una periferica, come una coppia di casse, può essere spostata da un computer a un altro, e computer separati possono addirittura condividere una stessa periferica. Inoltre, se un computer viene dismesso, le sue periferiche possono essere riutilizzate con una nuova macchina. Il ciclo di vita delle periferiche è, di norma, indipendente da quello del computer. Nell'UML, questo tipo di relazione è nota come aggregazione. Si discute più a fondo delle relazioni tra oggetti, e in particolare di composizione e aggregazione, nel Capitolo 16.

7.4.1 Classi e oggetti

La relazione esistente tra una classe e gli oggetti di quella classe è una relazione «istanzia». Questo è il primo esempio di relazione che si incontra. In *The UML Reference Manual* [Rumbaugh 1] si definisce una relazione come: "una connessione tra elementi del modello". Nell'UML esistono relazioni di molti tipi, e si arriverà a esaminarli tutti.

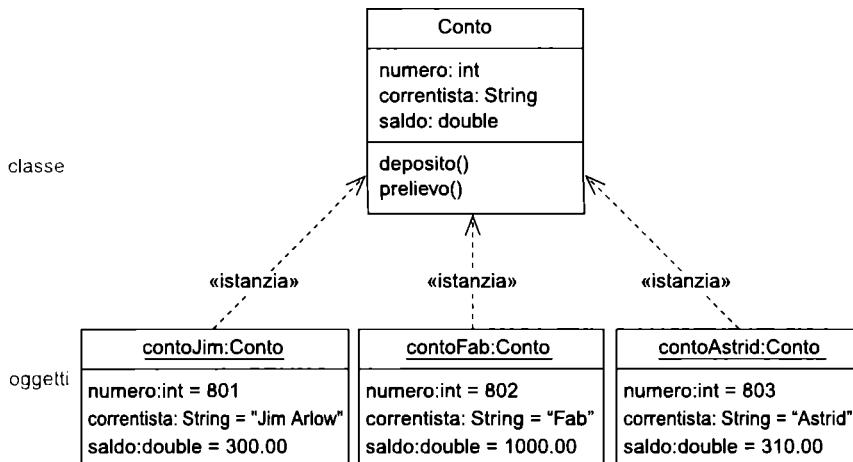


Figura 7.6

La relazione “*«instanzia»*” tra oggetti e classi è illustrata nella Figura 7.6. La freccia tratteggiata è in realtà una relazione di dipendenza a cui viene assegnato un significato speciale mediante lo stereotipo “*«instanzia»*”. Come visto nel Capitolo 1, i nomi tra virgolette rappresentano uno stereotipo, che è uno dei tre meccanismi di estendibilità dell’UML. Gli stereotipi sono un modo per personalizzare gli elementi di un modello; un modo per creare varianti con una nuova semantica. In questo caso, lo stereotipo “*«instanzia»*” trasforma una normale dipendenza in una relazione d’istanziazione tra la classe e gli oggetti di quella classe.

The UML Reference Manual [Rumbaugh 1] definisce una dipendenza come: “una relazione tra due elementi in cui un cambiamento in un elemento (il fornitore) può influenzare o fornire informazione di cui abbisogna l’altro elemento (il cliente)”. Nella Figura 7.6 la classe Conto deve chiaramente essere il fornitore, poiché essa determina la struttura di tutti gli oggetti della classe, e gli oggetti sono i clienti.

7.4.2 Istanziazione di un oggetto

L’istanziazione è la creazione di nuove istanze di elementi del modello. In questo caso, si istanziano oggetti da classi, ovvero si creano *nuove istanze* di classi.

L’UML si sforza di essere molto generico, quindi l’istanziazione si applica anche ad altri elementi del modello così come alle classi e agli oggetti. Infatti, istanziazione significa la generica creazione di un’istanza particolare di qualcosa a partire da un formato predisposto.

La maggior parte dei linguaggi di programmazione OO contiene operazioni speciali, dette costruttori, che apparterrebbero più propriamente alla classe che non agli oggetti di quella classe. Di queste operazioni speciali si dice che hanno uno *ambito* di classe. Si dirà qualcosa di più sull’ambito nel Paragrafo 7.6. Un’operazione costruttrice serve a creare nuove istanze della propria classe. A tal fine, il costruttore alloca della memoria per il nuovo oggetto, gli assegna un’identità unica e stabilisce i valori iniziali degli attributi dell’oggetto. Il costruttore crea anche gli eventuali collegamenti ad altri oggetti.

7.5 Notazione UML per le classi

La ricca sintassi grafica dell’UML per le classi diviene più gestibile applicando l’importante concetto di ornamenti opzionali dell’UML. Il nome è l’unica sottosezione obbligatoria nella sintassi grafica UML per le classi, mentre tutte le altre sottosezioni e gli altri ornamenti sono facoltativi. Comunque, quale riferimento, nella Figura 7.7 mostriamo l’intera classe.

Lo scopo per cui si produce il diagramma delle classi determina quali sottosezioni e ornamenti vengono indicati in una classe. Se si è interessati a mostrare le sole relazioni tra le classi, allora può bastare la sola sottosezione nome. Se il diagramma deve illustrare il comportamento delle classi, probabilmente occorre far vedere anche la sottosezione delle operazioni con le operazioni principali di ogni classe. Se, invece, il diagramma è un modello più “orientato ai dati”, conviene tentare di mostrare la corrispondenza tra classi e tabelle relazionali: aggiungendo alla sottosezione nome anche le sottosezioni operazioni e attributi, ed eventualmente anche i tipi degli attributi.

Ci si deve sforzare di utilizzare questa flessibilità dell'UML per comunicare la giusta quantità d'informazione che metta in evidenza il punto di vista desiderato, in modo chiaro e conciso.

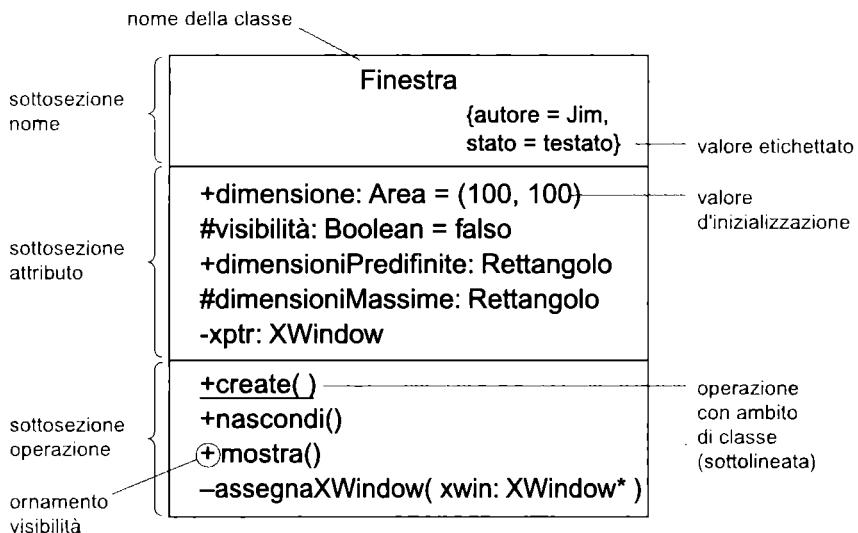


Figura 7.7

7.5.1 Sottosezione nome

Sebbene l'UML non imponga alcuna regola per dare i nomi alle classi, esiste una convenzione usata quasi universalmente.

- Il nome della classe viene scritto in “CamelCase”: iniziando con una lettera maiuscola e proseguendo sia con maiuscole che minuscole; i caratteri speciali, quali i segni d'interpunkzione e di sottolineatura, i vari simboli, le lineette, gli apici e le sbarrette, non sono ammessi. L'ottimo motivo è che questi caratteri vengono spesso usati in linguaggi quali HTML e XML, e in quelli dei sistemi operativi. Usarli nei nomi delle classi condurrebbe a conseguenze imprevedibili, qualora si dovesse generare della documentazione o del codice HTML/XML a partire dal modello UML. (Si osservi che, tra i “vari simboli” vietati dall'UML, si devono includere anche le lettere accentate, che a noi italiani può venir voglia di utilizzare, soprattutto sul modello dell'analisi, che viene letto anche dagli utenti comuni - N.d.T.)
- Si evitano le abbreviazioni *a tutti i costi*. I nomi delle classi dovrebbero sempre riflettere i nomi di entità del mondo reale, *senza* alcuna abbreviazione. Per esempio, **TappaViaggio** è meglio di **TpVg**, come **ContoDeposito** è preferibile a **CntDep**. Il motivo è, anche in questo caso, semplice: le abbreviazioni rendono il modello (e il codice che ne risulta) difficile da leggere. Tutto il tempo risparmiato a digitare si perderebbe poi, moltiplicato, qualora si dovesse effettuare manutenzione sul modello o sul codice.

7.5.2 Sottosezione attributo

Il nome dell'attributo è l'unica parte obbligatoria della sintassi UML per gli attributi (Figura 7.8):

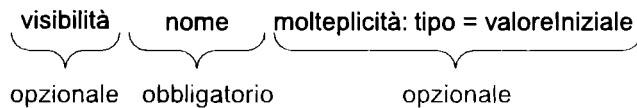


Figura 7.8

È consentito definire il valore che un attributo assumerà al momento della istanziazione dell'oggetto. Lo si chiama valore iniziale dell'attributo, perché è il valore che l'attributo prende al momento della creazione dell'oggetto. Definire ogniqualvolta possibile i valori iniziali è considerato un ottimo approccio nella progettazione, perché aiuta a garantire che gli oggetti di una classe siano sempre creati con uno stato utilizzabile e consistente. Nell'analisi si usano i valori iniziali solo quando possono esprimere od evidenziare un importante vincolo del problema; cosa che accade di rado.

7.5.2.1 Visibilità

L'ornamento di visibilità (Tabella 7.3) si applica agli attributi e alle operazioni della classe, ma può anche applicarsi ai nomi dei ruoli delle relazioni (Capitolo 9). Durante l'analisi, normalmente non si affollano i diagrammi con indicatori di visibilità, perché questi sono piuttosto un'affermazione del *come*, invece che del *cosa*.

Tabella 7.3

Ornamento	Tipo di visibilità	Semantica
+	Visibilità pubblica	Ogni elemento che può accedere alla classe può anche accedere a ogni suo membro con visibilità pubblica.
-	Visibilità privata	Solo le operazioni della classe possono accedere ai membri con visibilità privata
#	Visibilità protetta	Solo operazioni appartenenti alla classe o ai suoi discendenti possono accedere ai membri con visibilità protetta
~	Visibilità package	Ogni elemento nello stesso <i>package</i> della classe (o suo sotto- <i>package</i> annidato) può accedere ai membri della classe con visibilità package

Tutti i tipi di visibilità, a eccezione di *pubblica*, sono indipendenti dal linguaggio d'implementazione. Questa precisazione è importante, perché i diversi linguaggi possono definire tipi differenti di visibilità non tutti direttamente supportati dall'UML.

Il *The UML Reference Manual* [Rumbaugh 1], lascia aperta la questione di come questi specifici tipi di visibilità debbano essere espressi dagli strumenti di CASE.

“L’uso di valori addizionali (per la visibilità) deve avvenire per accordo tra l’utente e gli strumenti di modellazione e/o i generatori di codice.” Quindi, l’UML può, in linea di principio, supportare la visibilità come definita da un linguaggio qualunque, ma non definisce un metodo standard per farlo! Normalmente, questo non è un problema. I linguaggi OO più diffusi, C++ e Java, e perfino linguaggi parzialmente OO d’uso comune, quali Visual Basic, se la cavano in modo accettabile con le visibilità *pubblica*, *privata*, *protetta* e *package* usandole, mal che vada, come una prima approssimazione.

La Tabella 7.4 prende in esame due linguaggi OO: confronta la semantica della visibilità UML con quella di Java e C.

Come si vede, la definizione della visibilità dipende totalmente da linguaggio, e può essere alquanto complessa. L’esatto tipo di visibilità da utilizzare è una decisione implementativa di basso livello e dovrebbe, di solito, essere effettuata dal programmatore piuttosto che dall’alista/progettista. Dal punto di vista della modellazione in generale, le definizioni standard dell’UML (pubblica, privata, protetta e *package*) sono adeguate e sufficienti; pertanto ci si dovrebbe limitare a usare queste.

Tabella 7.4

Visibilità	Semantica UML	Semantica Java	Semantica C#
pubblica	Ogni elemento che può accedere alla classe, può accedere ad ogni suo membro con visibilità pubblica.	come in UML	come in UML
privata	Solo le operazioni della classe possono accedere a membri con visibilità privata	come in UML	come in UML
protetta	Solo operazioni appartenenti alla classe, o ai suoi discendenti, possono accedere ai membri con visibilità protetta	Come in UML, ma accessibile solo alle classi dello stesso package Java della classe in cui è definito	come in UML
package	Ogni elemento che sia nello stesso package della classe, o in un suo sotto-package annidato, può accedere ai membri della classe con visibilità package	La visibilità di default in Java -classi annidate in sotto-package annidati non hanno automaticamente accesso agli elementi del loro package genitore	—
privata	—	Lo stesso che protetta in UML	—
protetta	—	—	Accessibile da ogni elemento dello stesso programma
interna	—	—	Assomma le semantiche di interna e di protetta; si applica solo agli attributi
protetta interna	—	—	—

7.5.2.2 La molteplicità

La molteplicità è ampiamente usata nella progettazione, ma può anche essere utilizzata per i modelli dell’analisi, in quanto consente di esprimere concisamente certi vincoli del problema relativi al “numero di cose” che partecipano in una relazione. In particolare, la molteplicità consente di modellare due cose prettamente distinte con una sola specifica di molteplicità (si veda la Figura 7.9).

- Vettori: se la specifica di molteplicità è un intero più grande di 1, allora si sta definendo un vettore sul tipo. Per esempio, colori [7]: Colore è la specifica di un vettore di sette oggetti Colore, utilizzabile per descrivere i colori dell’arcobaleno.
- Valori nulli (“null”): molti linguaggi distinguono l’attributo che contiene un valore vuoto o non-inizializzato (come la stringa vuota, “”), dall’attributo che non si riferisce a nulla, ovvero il riferimento nullo a oggetto. Quando un attributo contiene un riferimento nullo, vuol dire che l’oggetto a cui si riferisce non è ancora stato creato, o che ha già cessato di esistere. Nella progettazione dettagliata capita a volte che sia importante evidenziare che null è un possibile valore di un certo attributo. Si esprime questa informazione con la speciale espressione di molteplicità [0..1]. Considerando l’esempio con indirizzoEmail della Figura 7.9, se l’attributo ha valore “” (stringa vuota), si può assumere che sia stato chiesto a qualcuno quale sia il suo indirizzo di email, ma che questi abbia detto di non averne una. D’altra parte, se l’attributo indirizzoEmail è “null”, si può assumere che non sia ancora stato richiesto ad alcuno di comunicare il proprio indirizzo di email, e che quindi il valore di questo attributo è sconosciuto. Si tratta di considerazioni di progettazione piuttosto dettagliate, ma che possono essere utili e importanti.

La Figura 7.9 mostra alcuni esempi di sintassi per la molteplicità.

espressione di molteplicità	
indirizzo[3] : String	un indirizzo si compone di un array di tre stringhe
nome[2..*] : String	un nome si compone di due o più stringhe
casellaEmail[0..1] : String	una casellaEmail si compone di una stringa o ha valore nullo

Figura 7.9

7.5.2.3 Sintassi per gli attributi: tecniche avanzate

Come per ogni altro elemento dei modelli dell’UML, la sintassi per gli attributi si può estendere facendoli precedere da stereotipi per indicare significati speciali.

Si può anche estendere la specifica di un attributo posponendogli dei *valori etichettati*. Per esempio:

"stereotipo" attributo { etichetta1 = valore1 , etichetta2 = valore2 , ... }

In realtà, sembra che nessuna di queste due convenzioni sia molto in uso, e il supporto per esse offerto dagli strumenti CASE è piuttosto limitato.

Nei valori etichettati è possibile registrare ciò che si vuole. Vengono spesso utilizzati per contenere informazioni come le seguenti:

indirizzo { aggiuntoDa = Jim Arlow , data=20MAR2001 }

Nell'esempio, abbiamo registrato che Jim Arlow ha aggiunto l'attributo indirizzo alla classe il giorno 20 marzo 2001.

7.5.3 Sottosezione operazione

Le operazioni sono funzioni vincolate a una certa classe. Come tali, hanno tutte le caratteristiche delle funzioni:

- nome;
- lista dei parametri;
- tipo del valore restituito.

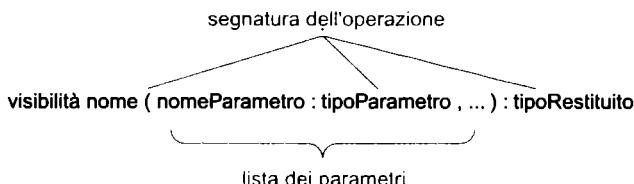


Figura 7.10

L'insieme costituito dal nome dell'operazione, dai tipi di tutti i parametri e dal tipo restituito, costituisce la *segnatura* dell'operazione (Figura 7.10). Ogni operazione di una classe deve avere una sua unica segnatura, in quanto essa determina l'identità dell'operazione. Quando s'invia un messaggio a un oggetto, la segnatura del messaggio viene confrontata con quella delle operazioni definite nella classe dell'oggetto e, se si trova una corrispondenza, l'operazione appropriata dell'oggetto viene chiamata.

I vari linguaggi d'implementazione interpretano il concetto di segnatura di un'operazione in modo leggermente diverso. Per esempio, C++ e Java non tengono conto del tipo restituito. Questo vuol dire che due operazioni di una stessa classe, che differiscono solo per il tipo restituito, saranno considerate stessa operazione, provocando dunque un errore di compilazione. In Smalltalk, che è un linguaggio a tipaggio piuttosto debole, i parametri e il valore restituito sono tutti di tipo Object, quindi la segnatura comprende il solo nome dell'operazione.

I nomi delle operazioni sono in *CamelCase*. A differenza dei nomi delle classi, essi iniziano sempre con una lettera minuscola, per continuare con lettere maiuscole e minuscole. Non si usano né i simboli speciali, né le abbreviazioni. I nomi delle operazioni sono normalmente dei verbi o delle frasi verbali.

7.5.3.1 Sintassi per le operazioni: tecniche avanzate

La sintassi per le operazioni può essere estesa sia preponendo uno stereotipo sia ponendo dei valori etichettati:

“stereotipo” operazione(...) { etichetta1 = valore1 , etichetta2 = valore2 , ...}

Tuttavia, queste tecniche non sembrano essere molto diffuse.

7.5.4 Sintassi per le classi con stereotipo

Sebbene siano previsti molti diversi modi per indicare uno stereotipo in un modello (Figura 7.11), molti modellatori preferiscono limitarsi a usare il nome tra apici (“nome-Stereotipo”) o l’icona. Le altre tecniche non sono molto diffuse, e gli strumenti CASE potrebbero non supportarle.

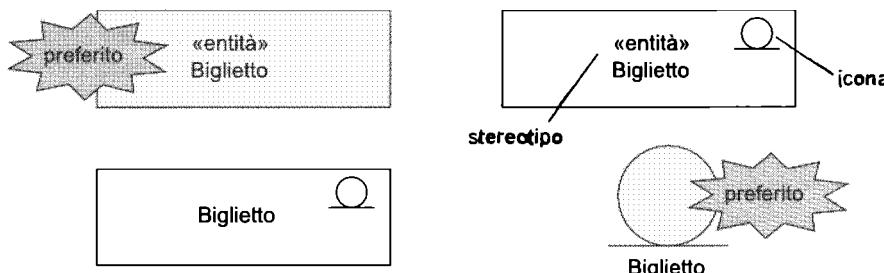


Figura 7.11

Uno stereotipo può anche essere associato a un colore o a un retino, ma questa è una pessima idea. Alcuni lettori, quali gli ipovedenti e i daltonici, potrebbero avere difficoltà a leggere questi diagrammi. Inoltre, i diagrammi devono spesso essere stampati in bianco e nero.

7.6 Ambito

Finora si sono visti oggetti che hanno una propria copia degli attributi definiti dalla loro classe, cosicché oggetti differenti possono avere differenti valori negli attributi. Analogamente, si sono finora considerate solo operazioni che agiscono su oggetti specifici. Questo è il caso normale, in cui si dice che attributi e operazioni hanno ambito di istanza.

Tuttavia, è qualche volta utile definire attributi che hanno un solo valore condiviso da tutti gli oggetti di una stessa classe, oppure definire operazioni (quali creazione e distruzione degli oggetti) che non operino solo su di una particolare istanza della classe. Si dice che questi attributi e operazioni hanno ambito di classe. Gli elementi che hanno ambito di classe sono condivisi da tutti gli oggetti di una stessa classe.

7.6.1 Ambito di istanza e ambito di classe

La Figura 7.12 illustra la notazione per l'ambito di istanza e di classe degli attributi e delle operazioni. La Tabella 7.5 riassume la semantica dell'ambito di istanza e di classe degli attributi e delle operazioni.

Tabella 7.5

	Ambito di istanza	Ambito di classe
Attributi	<p>Per <i>default</i>, gli attributi hanno ambito di istanza</p> <p>Ogni oggetto della classe ottiene una sua copia di un dato attributo con ambito di istanza</p> <p>Quindi, ciascun oggetto può avere valori diversi per un attributo con ambito d'istanza</p>	<p>Gli attributi possono essere definiti con ambito di classe</p> <p>Ogni oggetto della classe condivide un'unica copia di un attributo con ambito di classe</p> <p>Quindi, tutti gli oggetti avranno lo stesso valore per un attributo con ambito di classe</p>
Operazioni	<p>Senza non indicato altrimenti, le operazioni hanno ambito di istanza</p> <p>Ogni chiamata a un'operazione con ambito di istanza si applica a una specifica istanza della classe</p> <p>Non si può chiamare un'operazione con ambito di istanza se non si ha già un'istanza della classe.</p> <p>Chiaramente, questo implica che non si può usare questo tipo d'operazione per creare oggetti della classe, perché non si potrebbe mai crearne il primo</p>	<p>Le operazioni possono essere definite con ambito di classe</p> <p>La chiamata a un'operazione con ambito di classe non si applica a un'istanza specifica di una classe, ma invece si può pensare che l'operazione si applichi alla classe stessa</p> <p>Si può chiamare un'operazione con ambito di classe anche se non esiste un'istanza della classe. Questo consente di avere operazioni per la creazione degli oggetti</p>

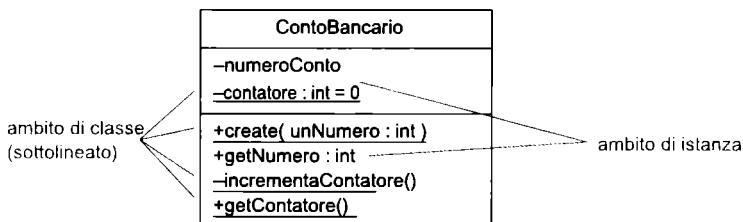


Figura 7.12

7.6.2 L'ambito determina l'accessibilità

Se un'operazione possa o meno accedere a un altro membro della classe viene stabilito dall'ambito dell'operazione e da quello del membro a cui essa cerca di accedere.

Operazioni con ambito di istanza possono accedere ad altre operazioni o attributi con ambito di istanza, come anche a tutti gli attributi e le operazioni con ambito di classe.

Le operazioni con ambito di classe possono accedere solamente ad altre operazioni e attributi con ambito di classe. Le operazioni con ambito di classe non possono accedere alle operazioni con ambito di istanza, perché:

- potrebbero non essere ancora state create delle istanze della classe;
- anche se esistessero delle istanze, non si saprebbe quale usare.

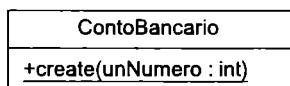
7.7 Creazione e distruzione degli oggetti

I costruttori sono operazioni speciali che creano nuove istanze delle classi: queste operazioni *devono* avere ambito di classe. Se avessero ambito di istanza, ovviamente non potrebbero essere chiamate per creare la prima istanza, perché non esisterebbe ancora alcuna istanza dal cui ambito chiamare il costruttore.

Ogni linguaggio ha uno standard diverso per i nomi dei costruttori. Un approccio completamente generico è quello di chiamare il costruttore semplicemente *create(...)*. Questo rende esplicito l'intento dell'operazione. I linguaggi Java, C# e C++ richiedono tuttavia che il nome del costruttore sia uguale a quello della classe.

Una classe può possedere molti costruttori, tutti dello stesso nome, ma ognuno reso distinguibile da una diversa lista di parametri. Il costruttore privo di parametri è noto come costruttore di *default* (predefinito). I parametri possono essere usati per inizializzare i valori degli attributi al momento della costruzione di un oggetto.

La Figura 7.13 mostra l'esempio di una semplice classe ContoBancario. Ogni volta che si vuole creare un nuovo oggetto ContoBancario, si deve passare al costruttore un numero come parametro. Questo numero stabilisce il valore dell'attributo numeroConto. Poiché il costruttore di ContoBancario necessita di un parametro, *non è possibile* creare un conto *senza* specificare questo parametro. Questo garantisce, in modo molto elegante, che ogni oggetto ContoBancario abbia il valore dell'attributo numeroConto stabilito all'atto della creazione. Nel modello dell'analisi non ci si preoccupa di specificare i costruttori, poiché normalmente non hanno alcun influenza o relazione con la parte di semantica del problema descritta dalla classe. Se proprio si vuole mostrare l'operazione costruttore, è possibile inserire come segnaposto un'operazione *create()* priva di parametri. In alternativa, si può specificare solo quei parametri che siano importanti dal punto di vista del dominio del problema.



Nome di costruttore generico



Nome di costruttore in java/C#/C++

Figura 7.13

Durante la progettazione si deve invece specificare il nome e tipo dei parametri e il tipo del valore restituito di *ogni* operazione; di conseguenza riprenderemo in esame la costruzione degli oggetti quando ci occuperemo del flusso di lavoro della progettazione, nella Parte 4.

La distruzione degli oggetti è un argomento più complesso della loro costruzione. I diversi linguaggi OO hanno semantiche differenti per la distruzione degli oggetti. Nelle prossime due sezioni esamineremo la costruzione e la distruzione degli oggetti mediante la classe di esempio ContoBancario.

7.7.1 Costruttori: classe ContoBancario

L'esempio ContoBancario nella Figura 7.14 illustra un tipico uso di attributi e di operazioni con ambito di classe. L'attributo contatore è un attributo con ambito di classe, privato, di tipo int. Questo attributo, pertanto, è condiviso da tutti gli oggetti della classe ContoBancario, e avrà lo stesso valore per ciascuno di questi oggetti.

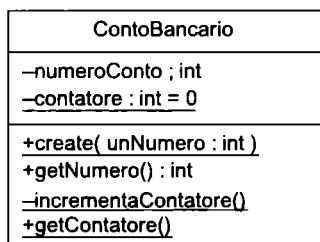


Figura 7.14

Al momento della creazione, l'attributo contatore viene inizializzato a zero. Se questo fosse stato un attributo con ambito di istanza, al momento della creazione ogni oggetto avrebbe ricevuto una sua propria copia dell'attributo. Invece, l'attributo ha ambito di classe: questo significa che ne esiste una sola copia, la quale viene inizializzata una sola volta. Il momento esatto in cui ciò avviene dipende dall'implementazione, ma per quanto ci interessa a questo punto, si può assumere che l'attributo venga inizializzato a zero all'avvio del programma.

Se nell'operazione `create()` si chiama l'operazione `incrementaContatore()`, che ha ambito di classe, essa incrementa il valore dell'attributo `contatore`, che ha anch'esso ambito di classe. L'attributo `contatore` si incrementa a ogni nuova creazione di un'istanza della classe e quest'ultima viene così ad avere un conteggio globale delle istanze. Si può ottenere il valore corrente del contatore mediante l'operazione `getContatore()`, che restituisce il numero di oggetti `ContoBancario` creati, e che ha anch'essa ambito di classe.

7.7.2 Distruttori: classe ContoBancario

Cosa accade se il programma oltre a creare oggetti `ContoBancario` li distrugge anche? Ovviamente, il valore di `contatore` diverrebbe presto inutile. Questo difetto si evita introducendo nella classe un'operazione che decrementa il conteggio delle istanze, e che viene chiamata a ogni distruzione di un'istanza di `ContoBancario`.

Alcuni linguaggi OO possiedono speciali metodi con ambito di classe, detti distruttori, che vengono automaticamente chiamati quando si elimina un oggetto. In C++, per esempio, il distruttore ha sempre la forma `~NomeDellaClasse(listaDiParametri)`, e il linguaggio *garantisce* che il metodo distruttore verrà chiamato al momento dell'eliminazione dell'oggetto.

Java possiede un meccanismo simile: ogni classe contiene un metodo, di nome `finalize()` chiamato alla terminazione dell'oggetto. Gli oggetti non vengono però eliminati subito, ma piuttosto sono passati a uno spazzino automatico detto *garbage collector*. Ne consegue che `finalize()` verrà chiamato successivamente, e in un istante impreciso, e questo non va bene per la semplice applicazione di conteggio di cui sopra. In questo caso, occorre decrementare esplicitamente il contatore da programma (per esempio, chiamando un metodo `decrementaConteggio()`, con ambito di classe o di istanza) ogni volta che si finisce di usare un oggetto, prima di affidarlo al *garbage collector*.

Il distruttore di C# ha la stessa semantica di Java, eccettuato il nome del metodo, che si chiama `Finalize()`.

7.8 Riepilogo

In questo capitolo sono stati presentati i concetti basilari concernenti le classi e gli oggetti, che vengono utilizzati nel resto del libro. Le classi e gli oggetti sono i mattoni con cui si costruiscono i sistemi OO, è quindi fondamentale comprenderli in modo completo e dettagliato. Sono stati trattati i seguenti concetti.

1. Gli oggetti sono unità coese di dati e funzioni.
2. Incapsulazione, così si dice quando i dati di un oggetto sono nascosti e si possono manipolare solo chiamando una delle funzioni dell'oggetto:
 - le operazioni sono specifiche astratte delle funzioni da usare nell'analisi;
 - i metodi sono specifiche concrete delle funzioni da usare nella progettazione.
3. Ogni oggetto è un'istanza di una classe; una classe definisce tutte le caratteristiche comuni condivise dagli oggetti di quella classe.
4. Ogni oggetto possiede queste caratteristiche:
 - Identità: l'oggetto è unico, ci si riferisce univocamente a esso mediante il suo riferimento all'oggetto.
 - Stato: un insieme significativo di valori degli attributi dell'oggetto in un dato istante.
 - Costituiscono uno stato solo quegli insiemi di valori degli attributi dell'oggetto, semanticamente distinti. Per esempio, oggetto ContoBancario con saldo < 0, stato = scoperto; saldo > 0, stato = solvibile.
 - Transizione: il passaggio di un oggetto da uno stato significativo a un altro.
 - Comportamento: i servizi che l'oggetto offre agli altri oggetti:
 - nell'analisi si rappresentano con le operazioni;
 - nella progettazione si rappresentano con i metodi;
 - eseguire un'operazione o un metodo *potrebbe* provocare una transizione di stato.

5. Gli oggetti interagiscono fra di loro per generare il comportamento del sistema. L'interazione comporta che gli oggetti si scambino messaggi: quando si riceve un messaggio viene eseguito il metodo corrispondente che, a sua volta, *potrebbe* produrre una transizione di stato.
6. Notazione UML per gli oggetti: le icone degli oggetti hanno due sottosezioni:
 - la sottosezione superiore contiene il nome dell'oggetto e/o quello della classe, tutto quanto sottolineato.
 - I nomi dell'oggetto e della classe sono in CamelCase. I nomi di oggetto iniziano con una lettera minuscola e quelli di classe con una maiuscola. La parte restante del nome può contenere lettere sia maiuscole che minuscole, ma senza simboli speciali.
 - Il segno di due punti separa il nome dell'oggetto da quello della classe.
 - La sottosezione inferiore contiene i nomi e i valori degli attributi separati dal segno d'egualanza.
 - I tipi degli attributi sono solitamente omessi dal diagramma.
 - I nomi e i valori sono in CamelCase, e iniziano con una lettera minuscola.
7. Una classe definisce le caratteristiche (attributi, operazioni, metodi, relazioni e comportamento) di un insieme di oggetti.
 - Ogni oggetto è un'istanza di una, e una sola, classe.
 - Oggetti diversi appartenenti alla stessa classe hanno lo stesso insieme di attributi, ma possono avere valori differenti per questi attributi. Per esempio, un tentativo di prelevare €100 da un oggetto ContoBancario che è già in scoperto produrrà un effetto diverso dal tentativo di prelievo di €100 da un conto con una disponibilità di €350.
 - Esistono molte classificazioni possibili per descrivere il mondo reale; trovare quella giusta è una delle chiavi per produrre una valida analisi OO.
 - Si mostra la relazione tra la classe e uno dei suoi oggetti utilizzando una dipendenza con stereotipo "«istanzia»":
 - le relazioni connettono cose tra loro;
 - una relazione di dipendenza indica che una variazione nel fornitore influisce sul cliente della relazione.
 - L'istanziazione crea un nuovo oggetto usandone la classe come schema costruttivo.
 - La maggior parte dei linguaggi OO possiede metodi speciali, detti costruttori, che sono attivati alla creazione dell'oggetto e lo inizializzano. I costruttori hanno ambito di classe (appartengono alla classe piuttosto che all'istanza).
 - Alcuni linguaggi OO hanno dei metodi speciali, detti distruttori, che sono chiamati all'eliminazione dell'oggetto. I distruttori fanno pulizia dopo la terminazione di un oggetto e hanno ambito di classe.

8. Notazione UML per le classi.

- La sottosezione nome contiene il nome della classe in CamelCase iniziante con maiuscola; non si devono usare abbreviazioni.
- La sottosezione attributi contiene per ogni attributo:
 - visibilità, controlla l'accesso ai membri della classe;
 - nome (obbligatorio), in CamelCase iniziante con minuscola;
 - molteplicità: vettore, per esempio [10]; valori null, per esempio [0..1];
 - tipo;
 - gli attributi possono avere uno stereotipo e uno o più valori etichettati.
- La sottosezione operazioni contiene, per ogni operazione:
 - visibilità;
 - nome (obbligatorio), in CamelCase iniziante con maiuscola;
 - lista dei parametri (nome e tipo di ciascuno);
 - tipo restituito;
 - stereotipo;
 - valori etichettati.
- La segnatura di un'operazione comprende, per ogni operazione:
 - nome.
 - lista dei parametri (tipi di tutti i parametri).
 - tipo restituito.
- Ogni operazione o metodo deve avere una segnatura univoca.

9. Ambito.

- Attributi e operazioni con ambito di istanza appartengono a, e operano su, oggetti specifici.
 - Le operazioni con ambito di istanza possono accedere ad altre operazioni o attributi con ambito di istanza.
 - Le operazioni con ambito di istanza possono accedere a tutti gli attributi e operazioni con ambito di classe.
- Attributi e operazioni con ambito di classe appartengono a, e operano su, tutti gli oggetti di una stessa classe.
 - Le operazioni con ambito di classe possono accedere solo ad altre operazioni e attributi con ambito di classe.
 - I costruttori sono operazioni speciali con ambito di classe, che creano nuovi oggetti.
 - I distruttori sono operazioni speciali, con ambito di classe, che fanno pulizia quando viene eliminato un oggetto.

Individuare le classi di analisi

8.1 Contenuto del capitolo

Il capitolo tratta dell'attività centrale dell'analisi OO: individuare le classi di analisi. Per comprendere l'attività UP in cui s'individuano le classi di analisi, si legga il Paragrafo 8.2. Per una definizione di classe di analisi, si veda il Paragrafo 8.3.

Il Paragrafo 8.4 spiega come individuare le classi di analisi. Vengono illustrate due tecniche specifiche: l'analisi nome/verbo (8.4.1) e l'analisi CRC (8.4.2). Si fanno anche considerazioni più generali su altre possibili fonti d'ispirazione per le classi.

8.2 Attività UP: analizzare un caso d'uso

I prodotti del flusso di lavoro dell'UP “Analizzare un caso d'uso” (vedere Figura 8.2) sono le classi di analisi e le realizzazioni di caso d'uso. Queste ultime sono collaborazioni tra oggetti che mostrano come sistemi di oggetti che interagiscono, realizzino il comportamento del sistema descritto nei casi d'uso. Questo capitolo si focalizza sulle classi di analisi, mentre le realizzazioni di caso d'uso verranno trattate nel Capitolo 12.

8.3 Cosa sono le classi di analisi?

Le classi di analisi sono classi che:

- rappresentano un'astrazione ben definita nel dominio del problema;
- dovrebbero corrispondere a concetti concreti del mondo del *business* (ed essere accuratamente denominate di conseguenza).

**Figura 8.1**

Il dominio del problema è quello in cui si è inizialmente manifestata l'esigenza di un sistema software (e, quindi, la necessità di un'attività di sviluppo). Normalmente si tratta di una specifica area di attività, quali le vendite telematiche o la gestione del contatto con i clienti. Tuttavia, è importante far notare che il dominio del problema potrebbe anche non essere un'attività gestionale, ma avere invece origine in un dispositivo che richiede del software per funzionare, quale, per esempio, un sistema *embedded*. Tutto lo sviluppo di software commerciale ha per fine ultimo una qualche necessità aziendale, che si tratti di automatizzare un processo gestionale o di sviluppare un nuovo prodotto con un importante componente di software.

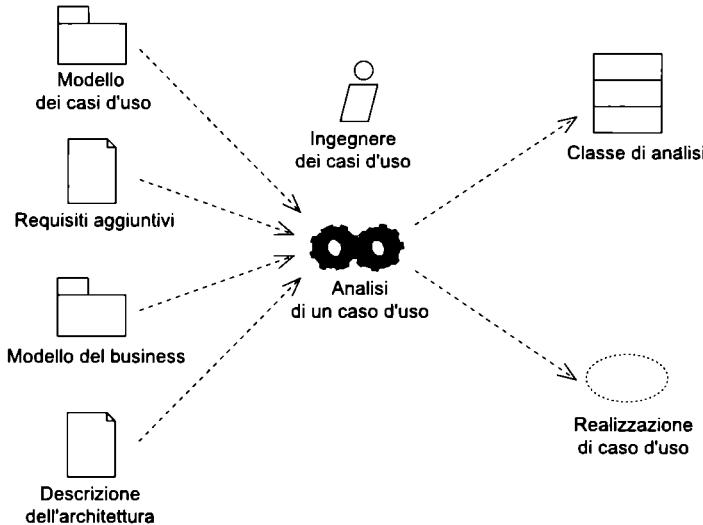


Figura 8.2 Adattata da Figura 8.25 [Jacobson 1], con il consenso della Addison-Wesley.

L’aspetto più importante di una classe di analisi è che deve corrispondere chiaramente e senza ambiguità a un concetto della realtà del *business*, come: cliente, prodotto o conto. Sebbene questa affermazione presuma che i concetti del *business* siano essi stessi chiari e privi d’ambiguità, questo succede di rado. L’analista OO deve, quindi, chiarificare i concetti di *business* che risultino confusi o inappropriate, per ottenerne qualcosa su cui si possa basare una classe di analisi. Per questa ragione l’analisi OO può risultare difficile.

Dunque, il primo passo nella costruzione di un sistema OO è la chiarificazione del dominio del problema. Se questo contiene dei concetti ben definiti e ha una struttura funzionale semplice, allora la soluzione è già pronta. La maggior parte di questo sforzo deve essere speso nel flusso di lavoro dei requisiti, durante le attività di individuazione dei requisiti, di creazione del modello dei casi d’uso e del Glossario di Progetto. Di contro, molti altri chiarimenti vengono prodotti durante la costruzione delle classi di analisi e delle realizzazioni di caso d’uso.

È importante che tutte le classi nel modello dell’analisi siano classi di analisi e non classi originate da considerazioni di progettazione (che concerne il dominio delle soluzioni). Quando poi si arriverà alla progettazione di dettaglio, si vedrà che ciascuna classe di analisi verrà alla fine rifinita in una o più classi di progettazione.

Sebbene nel precedente capitolo si fosse, di necessità, iniziato considerando oggetti specifici, si vedrà ora che il vero scopo dell’analisi OO consiste nell’individuare le classi di quegli oggetti. In effetti, individuare le giuste classi di analisi è la chiave dell’analisi e progettazione OO. Se le classi non sono corrette fin dal momento dell’analisi, si mette a rischio il resto del processo di sviluppo software, che dipende dai risultati dei flussi di lavoro dei requisiti e dell’analisi. Pertanto, nel flusso di lavoro dell’analisi è essenziale dedicare il tempo necessario a garantire che sia stato individuato il corretto insieme di classi di analisi. Questo tempo è ben speso, perché quasi certamente ne farà risparmiare più avanti.

In questo libro si focalizza soprattutto sullo sviluppo di sistemi gestionali, perché sono quelli di cui si occupa la maggior parte degli analisti e dei progettisti OO. Tuttavia, lo sviluppo di sistemi *embedded* non è altro che un caso specifico del normale processo di sviluppo, a cui si applicano gli stessi principi. I sistemi gestionali sono normalmente dominati dai requisiti funzionali, di conseguenza le attività più difficili sono di norma quelle relative ai requisiti e all'analisi. Nei sistemi *embedded*, dove sono preponderanti i requisiti non-funzionali, i requisiti sono spesso un dato di fatto e l'analisi tende a essere ovvia, ma la progettazione può risultare difficile.

8.3.1 Anatomia di una classe di analisi

Le classi di analisi dovrebbero contenere attributi molto ad “alto livello”. Esse *indicano* gli attributi che saranno *probabilmente* inclusi nelle risultanti classi di progettazione. Si può dire che le classi di analisi fissano gli attributi candidati per le classi di progettazione.

Le operazioni di una classe di analisi specificano, ad alto livello, i servizi principali offerti dalla classe che, al momento della progettazione, diverranno i metodi effettivamente implementabili. Può, comunque, avvenire che un'operazione di alto livello venga scomposta in più di un metodo.

Nell'analisi si usa solo un piccolo sottoinsieme della sintassi UML per le classi considerate nel Capitolo 7. Sebbene l'analista sia sempre libero di aggiungere qualunque ornamento ritenga utile per precisare il modello, la sintassi basilare per le classi di analisi esclude sempre i dettagli implementativi. Dopo tutto, nell'analisi si tenta di fissare la visione d'insieme.

Un formato minimale per una classe di analisi comprende quanto segue.

- Nome: questo è obbligatorio.
- Attributi: i nomi degli attributi sono obbligatori, anche se a questo punto si modella solo un sottoinsieme di candidati ad attributi. I tipi degli attributi sono opzionali.
- Operazioni: nell'analisi le operazioni possono anche essere solo delle affermazioni d'alto livello sulle responsabilità della classe. Si indicano i tipi restituiti e i parametri solo se sono necessari per rendere il modello più comprensibile.

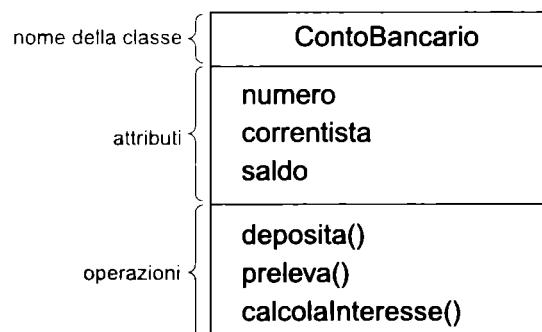


Figura 8.3

- Visibilità: generalmente non la si esplicita.
- Stereotipi: si possono indicare se migliorano il modello.
- Valori etichettati: si possono indicare se migliorano il modello.

La Figura 8.3 riporta un esempio di classe di analisi.

L'idea è quella di fissare l'essenza di un'astrazione nella classe di analisi e di lasciare i dettagli relativi all'implementazione per quando si arriverà alla progettazione.

8.3.2 Cosa contraddistingue una buona classe di analisi?

Le qualità di una buona classe di analisi si riassumono nei seguenti punti:

- il suo nome ne rispecchia l'intento;
- è un'astrazione ben definita, che modella uno specifico elemento nel dominio del problema;
- corrisponde a una caratteristica chiaramente identificabile del dominio del problema;
- ha un insieme ridotto e ben definito di responsabilità;
- ha la massima coesione interna;
- ha la minima interdipendenza con altre classi.

Nell'analisi si vuole modellare accuratamente e concisamente un aspetto del dominio del problema dal punto di vista del sistema che si intende costruire.

Per esempio, nel modellare un cliente per un sistema bancario, si vorranno includere il nome del cliente, il suo indirizzo ecc. Non avrebbe, invece, senso preoccuparsi delle sue preferenze di posto (finestrino o corridoio) per i viaggi in aereo. Occorre concentrarsi sugli aspetti del mondo reale che sono significativi per il sistema che si vuole costruire.

Molte volte, si può stabilire dal solo nome della classe se essa sia "buona" o meno. Considerando un sistema di commercio elettronico: **Cliente** si riferisce a un ente ben preciso del mondo reale, quindi sarebbe un buon candidato per una classe. Anche **CarrelloSpesa** sembrerebbe essere una buona astrazione: si comprende, quasi intuitivamente, quale semantica possa avere. Al contrario, **VisitatoreSitoWeb** sembra avere una semantica piuttosto indefinita; infatti sembra più un ruolo svolto dal **Cliente** in relazione al sistema di commercio elettronico. Si deve sempre ricercare una "astrazione ben definita": un qualcosa che possegga una semantica chiara ed evidente.

Una responsabilità è un contratto o un obbligo, che la classe ha verso i propri clienti. In sintesi, una responsabilità è un servizio che la classe offre ad altre classi. È cruciale che la classe abbia un insieme ben coeso di responsabilità in stretto accordo con l'intento della classe espresso dal nome, e con l'entità del mondo reale che viene modellata dalla classe.

Tornando all'esempio di CarrelloSpesa, ci si aspetta che questa classe abbia responsabilità del tipo:

- aggiungi un prodotto al carrello;
- rimuovi un prodotto dal carrello;
- mostra i prodotti nel carrello.

Questo è un insieme coeso di responsabilità, che hanno tutte a che fare con la gestione dell'insieme di prodotti scelti dal cliente. Lo si dice coeso, perché tutte le responsabilità lavorano per lo stesso fine: gestire il carrello della spesa del cliente. Tant'è che si potrebbero riassumere queste tre responsabilità con una di più alto livello chiamata: "gestire il carrello". Si potrebbero anche aggiungere a CarrelloSpesa queste tre responsabilità:

- verifica la carta di credito;
- accetta il pagamento;
- stampa una ricevuta.

Queste nuove responsabilità non sembrano, però, accordarsi con l'intento e la semantica intuitiva dei carrelli della spesa. Non sono coese e dovrebbero certamente essere assegnate altrove; possibilmente a una classe EmittenteCarteCredito, una classe Cassa-Negozio e una classe RicevutaAcquisti. Le responsabilità devono essere distribuite in modo da massimizzare la coesione interna di ciascuna classe.

Infine, le classi migliori hanno la minima interdipendenza con le altre classi. Si misura l'interdipendenza di una classe contando con quante altre classi ha relazioni. Una distribuzione uniforme di responsabilità tra le classi risulterà in una minore interdipendenza. Concentrare responsabilità di controllo o assegnare troppe responsabilità a un'unica classe, aumenta l'interdipendenza verso quella classe. Il Capitolo 15 tratta le tecniche per massimizzare la coesione e minimizzare l'interdipendenza tra classi.

8.3.3 Regole pratiche per le classi di analisi

Ecco alcune regole pratiche per creare delle classi di analisi ben formate.

- Tra le tre e le cinque responsabilità per classe. Tipicamente le classi dovrebbero essere mantenute per quanto possibile semplici, cosa che normalmente limita la quantità di responsabilità che possono sostenere a un numero compreso tra tre e cinque. Il precedente esempio di RicevutaAcquisti è un buon caso di classe ben delimitata, con poche responsabilità, facilmente gestibili.
- Nessuna classe può essere isolata. Per la corretta analisi e progettazione OO è fondamentale che le classi collaborino tra di loro a beneficio degli utilizzatori. Pertanto, ogni classe dovrebbe essere associata a un piccolo insieme di altre classi con cui collabora per produrre il risultato desiderato. Le classi possono delegare alcune delle loro responsabilità ad altre "classi ausiliarie" dedicate a quello scopo specifico.

- Evitare di avere tante classi troppo semplici. Può essere a volte difficile trovare il giusto equilibrio. Se il modello pare avere una miriade di classi molto semplici dotate solo di una o due responsabilità ciascuna, occorre rivederlo molto attentamente per consolidare alcune delle classi semplici in classi più complesse.
- Evitare di avere poche classi troppo complesse. Il reciproco del caso precedente è la situazione in cui si hanno poche classi, ma molte di esse hanno un gran numero (> 5) di responsabilità. In questo caso, si adotta la strategia di esaminare ciascuna classe a turno e vedere come decomporla in due o più classi meno complesse con il giusto numero di responsabilità.
- Evitare i “*functoid*”. Un *functoid* è una normale funzione procedurale travestita da classe. Grady Booch è solito raccontare un divertente aneddoto a proposito del modello di un sistema molto semplice che aveva migliaia di classi. A un esame più attento, ogni classe aveva un'unica operazione chiamata *esegui()*. I *functoid* sono un pericolo soprattutto quando un'analista abituato alla decomposizione funzionale *top-down* si accosta per la prima volta all'analisi OO.
- Evitare le classi onnipotenti. Si tratta di quelle classi che sembrano fare di tutto. Attenzione alle classi che contengono “sistema” o “controllore” nel loro nome! Una strategia utile per risolvere questo problema è quella di verificare se le responsabilità della classe onnipotente possano essere raggruppate in sottoinsiemi coesi. In questo caso, ciascuno di questi sottoinsiemi può forse essere fattorizzato in una classe separata. Queste classi semplificate collaborerebbero poi per implementare il comportamento già offerto dalla classe originale.
- Evitare gli alberi di ereditarietà profondi. La progettazione di una gerarchia di ereditarietà elegante richiede che ogni suo livello d'astrazione abbia uno scopo ben definito. È facile aggiungere molti livelli che non hanno alcuno scopo realmente utile. Un errore comune è quello di usare l'ereditarietà per creare una specie di decomposizione funzionale, dove ogni livello d'astrazione ha un'unica responsabilità. Questo è, da tutti i punti di vista, inutile e conduce solamente a un modello complesso e difficile da capire. L'ereditarietà viene usata per l'analisi solo quando un'ovvia ed evidente gerarchia di ereditarietà scaturisce direttamente dal dominio del problema.

8.4 Individuare le classi

Nel resto di questo capitolo si esamina la questione fondamentale dell'analisi e della progettazione OO: come individuare le classi di analisi.

Come osserva Meyer in *Object Oriented Software Construction* [Meyer 1], non esiste un semplice algoritmo d'individuazione delle classi. Se esistesse, equivarrebbe a un metodo infallibile per progettare il software OO e l'esistenza di un tale algoritmo è tanto improbabile, quanto quella di un metodo infallibile per provare la validità di teoremi matematici.

Ciononostante, esistono tecniche provate e sperimentate per trovare delle buone soluzioni, e adesso saranno illustrate. Queste tecniche si basano sull'analisi del testo e sulle interviste agli utenti e agli esperti del dominio. In definitiva, nonostante tutte le tecniche, individuare le classi "giuste" dipende comunque molto dalla percezione, abilità ed esperienza del singolo analista.

8.4.1 Individuare le classi con l'analisi nome/verbo

L'analisi nome/verbo è un modo molto semplice di analizzare un testo al fine d'individuare delle classi. In sintesi, nomi e frasi nominali indicano le classi o gli attributi delle classi, mentre i verbi e le frasi verbali indicano le responsabilità o le operazioni di una classe. L'analisi nome/verbo viene usata da molti anni e funziona bene in quanto analizza direttamente il linguaggio del dominio del problema. Bisogna, comunque, fare attenzione a problemi di sinonimia e omonimia che potrebbero dare origine a classi spurie.

Se il dominio del problema è poco noto e mal definito, occorre essere molto cauti. In questo caso, bisogna raccogliere la maggior quantità di informazioni del dominio possibile, dal maggior numero di persone possibile. Si consiglia anche di studiare domini di problemi simili, esistenti al di fuori della propria organizzazione.

L'aspetto forse più ingannevole dell'analisi nome/verbo è l'individuazione delle classi "nascoste". Si tratta di classi implicite del dominio del problema che possono anche non venir mai menzionate esplicitamente. Per esempio, i committenti di un sistema di prenotazioni per una compagnia di viaggi potrebbero parlare di prenotazioni, richieste ecc, ma potrebbero non menzionare mai l'astrazione più importante, Ordine, se essa non è già utilizzata nel sistema aziendale esistente. In genere, ci si rende conto di aver individuato una classe nascosta, perché introducendo questa nuova astrazione l'intero modello si assesta all'improvviso. Sorprendentemente, questo capita di frequente. In effetti, quando ci si trova nei guai con un modello dell'analisi che proprio non sembra aver senso, ci si mette in cerca di classi nascoste. Mal che vada, questo consente di fare alcune domande approfondite e di migliorare la conoscenza del dominio del problema.

8.4.1.1 Procedura per l'analisi nome/verbo

Come primo passo dell'analisi nome/verbo, si raccolgono quante più informazioni rilevanti possibile. Le potenziali fonti di informazioni sono:

- l'SRS, se esiste;
- i casi d'uso;
- il Glossario di Progetto;
- qualunque altro documento (architettura, visione aziendale ecc).

Dopo aver raccolto la documentazione, la si analizza semplicemente evidenziando (o registrando in qualche altro modo) i seguenti elementi:

- i nomi: per esempio, volo;
- i predicati nominali: per esempio, numero del volo;
- i verbi: per esempio, assegnare;
- i predicati verbali: per esempio, verificare la carta di credito.

I nomi e le frasi nominali possono indicare classi o attributi di classe. I verbi e le frasi verbali possono indicare responsabilità di classe.

Se, durante questo processo, si incontrano termini sconosciuti, è necessario farseli chiarire immediatamente da un esperto del dominio, e aggiungerli al Glossario di Progetto. A questo punto si prende la lista di nomi, frasi e verbi e si usa il Glossario di Progetto per risolvere tutti i casi di omonimia e di sinonimia. In questo modo, si ottiene una lista di candidati per classi, attributi e responsabilità.

Con la lista di candidati si predisponde una prima assegnazione di attributi e responsabilità alle classi. Si può utilizzare uno strumento CASE, inserendo le responsabilità nelle classi come operazioni. Anche i potenziali attributi individuati possono essere inseriti provvisoriamente nelle classi. Nel caso ci si sia fatta qualche idea sulle possibili relazioni tra le classi (i casi d'uso sono delle buone fonti), si possono anche inserire queste potenziali associazioni. In questo modo si ottiene una prima bozza del modello, da raffinare con ulteriori analisi.

8.4.2 Individuare le classi con l'analisi CRC

L'analisi CRC è un modo efficace (e divertente) di coinvolgere gli utenti nella ricerca delle classi. CRC significa classe, responsabilità e collaboratori (class, responsibilities, and collaborators). Questa tecnica usa il più potente strumento d'analisi del mondo: il giallino adesivo! Il metodo CRC è così popolare che si narra che una volta una ditta abbia messo in commercio dei giallini adesivi prestampati con le sottosezioni del nome della classe, delle responsabilità e dei collaboratori.

Si comincia preparando alcuni giallini adesivi come in Figura 8.4. Ognuno viene suddiviso in tre scomparti: in quella superiore si scrive il nome della classe candidata, a sinistra le sue responsabilità e a destra i suoi collaboratori. I collaboratori sono le classi che possono lavorare con questa per realizzare una parte della funzionalità del sistema. La sottosezione dei collaboratori consente di indicare le collaborazioni tra classi. Un altro modo per fissare queste relazioni (preferibile) è quello di appiccicare i giallini a una lavagna e tracciare delle linee tra le classi che collaborano.

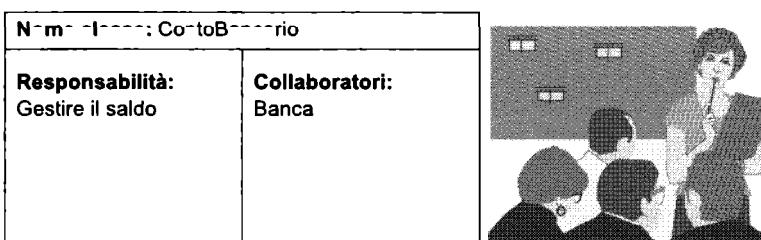


Figura 8.4

8.4.2.1 Procedura per l'analisi CRC

A meno che il sistema sia molto semplice, l'analisi CRC dovrebbe sempre essere usata congiuntamente all'analisi nome/verbo dei casi d'uso, requisiti, glossario e altre informazioni rilevanti. La procedura di analisi CRC è semplice e consente di separare la raccolta delle informazioni dall'analisi delle stesse. Conviene, quindi, eseguirla in due fasi.

8.4.2.2 Fase 1: Brainstorming: raccogliere informazioni

I partecipanti sono gli analisti OO, i committenti e le altre parti interessate e gli esperti del dominio. La procedura è la seguente.

1. Spiegare ai partecipanti che si tratta di una sessione di *brainstorming*.
 - 1.1 Tutte le idee saranno considerate buone idee.
 - 1.2 Le idee vengono registrate, ma non dibattute: non si discute nulla, ci si limita a scrivere tutto e a proseguire. Verrà tutto analizzato successivamente.
2. Chiedere ai partecipanti di fare il nome di “cose” che agiscono nel loro dominio del problema. Per esempio, cliente o prodotto.
 - 2.1 Scrivere ogni cosa su di un giallino: si tratta di una potenziale classe o di un potenziale attributo di una classe.
 - 2.2 Appiccicare il giallino a una parete o a una lavagna.
3. Chiedere ai partecipanti di indicare le responsabilità che quelle cose potrebbero avere e scriverle nella sottosezione delle responsabilità del giallino.
4. Lavorando assieme ai partecipanti, cercare di individuare le classi che potrebbero lavorare insieme. Risistemare i giallini sulla lavagna per illustrare questa organizzazione e tracciare linee per congiungerle, oppure scrivere i collaboratori nell'apposita sottosezione dei giallini.

8.4.2.3 Fase 2: Analizzare l'informazione

I partecipanti sono gli analisti OO e gli esperti del dominio. Come decidere quali giallini adesivi dovrebbero diventare classi e quali, invece, attributi? Si ritorni al Paragrafo 8.3.2: le classi di analisi *devono* essere delle astrazioni ben definite nel dominio del problema. Certi giallini rappresentano dei concetti chiave del *business* e dovranno chiaramente diventare delle classi. Per altri giallini potrebbe essere più difficile prendere una decisione. Se un giallino sembra essere logicamente una *parte* di un altro, questo è un buon indizio che forse si tratta di un attributo. Anche nel caso che un giallino non sembri essere particolarmente importante, oppure abbia un comportamento di scarso interesse, è necessario chiedersi se non sia possibile che si tratti di un attributo di un'altra classe.

I giallini su cui restano dei dubbi vengono trattati come classi. Lo scopo principale è di produrre una buona ipotesi iniziale e di concludere questo processo: il modello potrà sempre essere raffinato in seguito.

8.4.3 Cercare altre fonti di classi

Vale la pena di ricordare che, oltre all'analisi nome/verbo e CRC, esistono anche altre fonti di potenziali classi che dovrebbero essere prese in considerazione. Poiché si cercano delle astrazioni ben definite nella realtà del problema, ovviamente le si può cercare nel mondo reale.

Di seguito si elencano alcuni esempi.

- Oggetti fisici, quali aeroplani, persone e alberghi possono tutti denotare delle classi.
- La modulistica è un'altra ricca fonte di classi. Documenti quali fatture, ordini e libretti bancari possono tutti corrispondere a una classe. Bisogna però essere cauti esaminando la modulistica. In molte aziende le scartoffie si sono evolute negli anni proprio per supportare gli inefficienti processi gestionali che il nuovo sistema potrebbe voler tentare di rimpiazzare! L'ultima cosa che un analista/progettista OO vuol fare è automatizzare processi obsoleti e patologici basati sulla carta.
- Interfacce tra il sistema e il mondo esterno, come: schermi, tastiere, periferiche e altri sistemi possono essere dei candidati a classi; soprattutto nei sistemi *embedded*.
- Entità concettuali che siano essenziali per il funzionamento dell'organizzazione, ma che non si manifestano in cose concrete. Un esempio potrebbe essere una IniziativaFedeltà come per gli sconti fedeltà ai clienti. Chiaramente l'iniziativa stessa non è una cosa materiale (non la si può prendere a calci!), ma si tratta comunque di un'astrazione coesa e pertanto potrebbe prestarsi bene a essere modellata come classe.

8.5 Creazione di una prima bozza di modello dell'analisi

Per creare una prima bozza di modello dell'analisi, occorre utilizzare uno strumento CASE per consolidare i prodotti dell'analisi nome/verbo, CRC e le considerazioni ricavate da altre fonti di classi in un unico modello UML. Si può procedere nel seguente modo.

- Confrontare le tre fonti d'informazione: i risultati dell'analisi CRC e nome/verbo, e i risultati dell'esame delle altre fonti di classi.
- Consolidare le classi di analisi, gli attributi e responsabilità individuate dalle diverse fonti e inserirle in uno strumento CASE.
 - Eliminare sinonimi e omonimi utilizzando il Glossario di Progetto.
 - Cercare le differenze tra i risultati delle tre tecniche: tali discordanze indicano aree dove vi è incertezza o dove potrebbe essere necessario ulteriore lavoro. Risolvere subito queste differenze o contrassegnarle per una lavorazione successiva.
- I collaboratori (o le linee tra i giallini appiccicati alla lavagna) rappresentano relazioni tra le classi. Si vedrà come modellarle nel Capitolo 9.
- Normalizzare i nomi delle classi, degli attributi e delle responsabilità utilizzando gli standard della propria organizzazione o seguendo le semplici convenzioni descritte nel Capitolo 7.

Il prodotto di questa attività è un insieme di classi di analisi dove ognuna *potrebbe* avere alcuni attributi chiave e *dovrebbe* avere da tre a cinque responsabilità. Questo è la prima bozza del modello dell'analisi.

8.6 Riepilogo

Questo capitolo ha illustrato cosa siano le classi di analisi e come sia possibile individuarle mediante le tecniche di analisi nome/verbo, di *brainstorming* e di analisi CRC e esaminando altre fonti di potenziali classi.

Abbiamo imparato i seguenti concetti.

1. L'attività UP "Analizzare un caso d'uso" produce classi di analisi e realizzazioni di caso d'uso.
2. Le classi di analisi rappresentano un'astrazione ben definita nel dominio del problema.
 - Il dominio del problema è quello in cui si è inizialmente manifestata l'esigenza di un sistema software .
 - Le classi di analisi devono corrispondere chiaramente e senza ambiguità a un concetto della realtà del problema.
 - I concetti del dominio del problema devono spesso essere chiariti durante l'analisi.
3. Il modello dell'analisi contiene solo classi di analisi. Tutte le classi derivate da considerazioni di progettazione (il dominio delle soluzioni) devono essere escluse.
4. Le classi di analisi comprendono:
 - un insieme, di alto livello, di potenziali attributi;
 - un insieme, di alto livello, di operazioni.
5. Cosa contraddistingue una buona classe di analisi?
 - Il suo nome ne rispecchia l'intento.
 - È un'astrazione ben definita, che modella uno specifico elemento del dominio del problema.
 - Corrisponde a una caratteristica chiaramente identificabile del dominio del problema.
 - Ha un piccolo insieme di responsabilità ben definite:
 - una responsabilità è un contratto, o un obbligo, che la classe ha verso i propri clienti;
 - una responsabilità è un insieme semanticamente coeso di operazioni;
 - ogni classe dovrebbe avere tra le tre e le cinque responsabilità.
 - Ha la massima coesione: tutte le caratteristiche della classe devono collaborare per realizzare il suo scopo.
 - Ha la minima interdipendenza con altre classi: una classe dovrebbe collaborare solo con un numero limitato di altre classi per realizzare il suo scopo.

6. Cosa contraddistingue una cattiva classe di analisi?

- È un *functoid*: una classe con una sola operazione.
- È una classe onnipotente: una classe che fa di tutto; attenzione alle classi che contengono “sistema” o “controllore” nel loro nome!
- Appartiene a un albero di ereditarietà profondo: nel mondo reale gli alberi di ereditarietà tendono ad avere pochi livelli.
- Ha scarsa coesione.
- Ha troppe interdipendenze.

7. Analisi nome/verbo.

- Cercare i nomi e le frasi nominali: questi sono candidati per classi o attributi di classe.
- Cercare i verbi e le frasi verbali: questi sono candidati per responsabilità oppure operazioni.
- La procedura prevede la raccolta di tutte le informazioni rilevanti, seguita dall’analisi delle stesse.

8. L’analisi CRC è un’efficace e divertente tecnica di *brainstorming*.

- Scrivere le cose importanti del dominio del problema su giallini adesivi.
- Ogni giallino ha tre sottosezioni:
 - classe: contiene il nome della classe;
 - responsabilità: contiene la lista di responsabilità della classe;
 - collaboratori: contiene una lista di altre classi che collaborano con questa classe.
- Procedura – brainstorming e analisi:
 - chiedere ai partecipanti di nominare le “cose” che agiscono nel loro dominio del problema e scriverle sui giallini adesivi;
 - chiedere ai partecipanti di indicare le responsabilità che quelle cose potrebbero avere e scriverle nella sottosezione delle responsabilità dei corrispondenti giallini;
 - chiedere ai partecipanti di identificare le classi che potrebbero lavorare insieme e tracciare linee per congiungere i giallini, oppure scrivere i collaboratori nella apposita sottosezione dei corrispondenti giallini.

9. Considerare anche altre fonti di possibili classi quali oggetti fisici, modulistica, interfacce con il mondo esterno ed entità concettuali.

10. Creare una prima bozza del modello dell’analisi:

- confrontare i risultati dell’analisi CRC e dell’analisi nome/verbo, con i risultati dell’esame delle altre fonti di potenziali classi;
- eliminare sinonimi e omonimi;
- le differenze tra i risultati delle tre tecniche indicano aree d’incertezza;
- consolidare i risultati in una prima bozza di modello dell’analisi.

9.1 Contenuto del capitolo

Questo capitolo tratta delle relazioni tra oggetti e delle relazioni tra classi. Per sapere che cosa è una relazione, si veda il Paragrafo 9.2. Il capitolo prosegue lungo tre percorsi distinti. Si discute di collegamenti (relazioni tra oggetti) nel Paragrafo 9.3, di associazioni (relazioni tra classi) nel Paragrafo 9.4, e infine di dipendenze (relazioni generiche) nel Paragrafo 9.5.

9.2 Cos'è una relazione?

Le relazioni sono connessioni semanticamente significative tra elementi di modellazione: sono il metodo UML per collegare le cose tra di loro. Sono state già viste alcuni tipi di relazioni:

- tra attore e caso d'uso (associazione);
- tra caso d'uso e caso d'uso (generalizzazione, «include», «estende»);
- tra attore e attore (generalizzazione).

In questo capitolo si esamineranno le connessioni tra oggetti e le connessioni tra classi.. Si comincia con i collegamenti e le associazioni, tralasciando invece la generalizzazione e l'ereditarietà, che saranno trattate più avanti, nel Capitolo 10.

Per ottenere un sistema OO che funzioni, non si possono lasciare i singoli oggetti nel loro splendido isolamento. Occorre connetterli tra di loro, affinché svolgano dei compiti utili agli utilizzatori del sistema. Le connessioni tra gli oggetti si chiamano collegamenti e, quando gli oggetti lavorano insieme, si dice che essi collaborano.

Se tra due oggetti esiste un collegamento, allora deve anche esistere una connessione semantica tra le loro rispettive classi. Il buon senso dice che affinché due oggetti possano comunicare direttamente tra loro, le loro rispettive classi devono avere una qualche conoscenza l'una dell'altra. Le connessioni tra classi si chiamano associazioni.

I collegamenti tra oggetti sono proprio delle istanze delle associazioni tra le rispettive classi di tali oggetti.

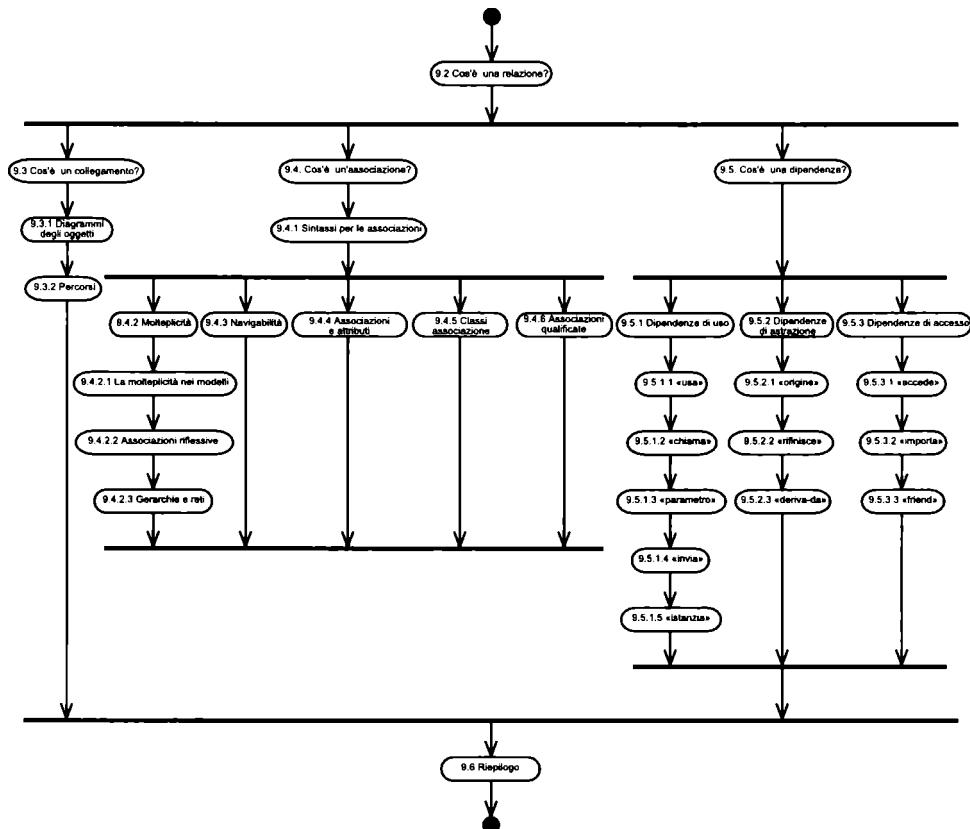


Figura 9.1

9.3 Cos'è un collegamento?

La creazione di un programma orientato agli oggetti richiede la presenza di oggetti che comunicano tra loro. In realtà, un programma OO in esecuzione non è altro che un'armoniosa cooperativa di oggetti.

Un collegamento è una connessione semantica tra due oggetti che consente loro di scambiarsi messaggi. Un sistema OO in esecuzione contiene molti oggetti che nascono e che muoiono, e molti collegamenti (che, anch'essi, nascono e muoiono) che connettono questi oggetti tra di loro. Gli oggetti si scambiano messaggi tramite questi collegamenti. Quando un oggetto riceve un messaggio, chiama il corrispondente metodo.

I vari linguaggi di programmazione OO realizzano i collegamenti in modi diversi. Java implementa i collegamenti come riferimenti a oggetto; C++ può realizzarli come puntatori, riferimenti o tramite incorporazione di un oggetto in un altro.

Qualunque sia l'approccio, il requisito minimo per stabilire un collegamento è che almeno uno dei due oggetti disponga di un riferimento all'altro. Questo costituisce un collegamento unidirezionale dall'oggetto proprietario del riferimento (l'origine) all'oggetto riferito (la destinazione). Si disegna una freccia a un'estremità del collegamento per indicare questa navigabilità unidirezionale. Se entrambi gli oggetti hanno riferimenti l'uno dell'altro, allora si parla di collegamento bidirezionale, il quale viene indicato con una semplice linea priva di frecce.

9.3.1 Diagrammi degli oggetti

Un diagramma degli oggetti mostra gli oggetti, con le loro relazioni, in un dato istante. È come un'istantanea di una parte del sistema OO in esecuzione, che mostra gli oggetti e i loro collegamenti in un certo momento specifico.

Gli oggetti collegati possono assumere dei ruoli l'uno rispetto all'altro. Nella Figura 9.2 si vede che l'oggetto **jim** assume il ruolo di **presidente** nel suo collegamento con l'oggetto **clubDiscesaLibera**. Lo si indica scrivendo il nome del ruolo sull'estremità appropriata del collegamento. Si possono scrivere nomi di ruolo su una sola, o su entrambe le estremità del collegamento. Nell'esempio l'oggetto **clubDiscesaLibera** svolge sempre il ruolo di "club" e sarebbe inutile indicarlo nel diagramma; non aggiungerebbe nulla alla comprensione della relazione tra gli oggetti.

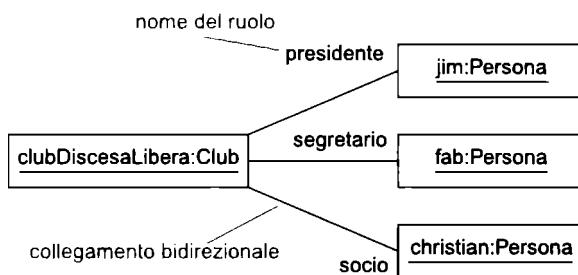


Figura 9.2

La Figura 9.2 mostra come, in un dato istante, l'oggetto **jim** svolga il ruolo di **presidente**. Tuttavia, è importante notare che i collegamenti sono connessioni dinamiche tra oggetti. In altre parole, non sono necessariamente stabili nel tempo. Nel nostro esempio, in un qualunque momento, il ruolo di **presidente** potrebbe passare a **fab** o **christian**, e si potrebbe facilmente disegnare un diverso diagramma degli oggetti che illustri la nuova situazione.

Normalmente un singolo collegamento connette esattamente due oggetti, come nella Figura 9.2. L'UML tuttavia consente anche di utilizzare un unico collegamento per connettere più di due oggetti. Questo tipo di collegamento viene detto collegamento n-ario e lo si rappresenta con un rombo e una linea connessa a ogni oggetto partecipante. Molti modellatori (tra cui gli autori) ritengono che sia una notazione superflua. Non se ne parlerà ulteriormente, perché viene usata di rado e non tutti gli strumenti CASE per UML la supportano.

Un attento esame della Figura 9.2 evidenzia che vi sono tre collegamenti tra quattro oggetti:

- un collegamento tra clubDiscesaLibera e jim;
 - un collegamento tra clubDiscesaLibera e fab;
 - un collegamento tra clubDiscesaLibera e Christian.

Nella Figura 9.2 i collegamenti sono bidirezionali, quindi si può correttamente affermare sia che il collegamento connette jim a clubDiscesaLibera, o che esso connette clubDiscesaLibera a jim.

È possibile specificare che il collegamento è unidirezionale aggiungendo una freccia all'estremità appropriata della linea. Per esempio, la Figura 9.3 mostra che il collegamento tra `:DettagliPersona` e `:Indirizzo` è unidirezionale. Ciò significa che l'oggetto `:DettagliPersona` ha un riferimento all'oggetto `:Indirizzo`, ma non vice versa. I messaggi possono essere inviati solo da `:DettagliPersona` a `:Indirizzo`.

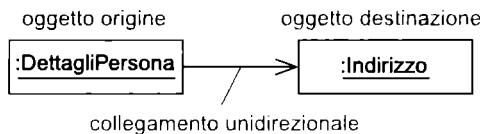


Figura 9.3

9.3.2 Percorsi

I simboli UML, come l'icona dell'oggetto, l'icona del caso d'uso e quella della classe sono connessi ad altri simboli mediante percorsi. Un percorso è “una sequenza connessa di segmenti” (in altre parole, è una linea!) che connette due o più simboli. Ci sono due stili per tracciare i percorsi:

- ortogonale: in cui il percorso è costituito da segmenti orizzontali e verticali;
 - obliqui: in cui il percorso è costituito da una o più linee inclinate.

I gusti personali dettano quale stile usare, e si può anche usarli entrambi per rendere più chiaro e leggibile il diagramma. È preferibile utilizzare lo stile ortogonale, come scelgono molti altri modellatori.

La Figura 9.4, è disegnata nello stile ortogonale e i percorsi sono aggregati in un albero. Si possono aggregare solo i percorsi dotati delle stesse proprietà. In questo caso, tutti i percorsi rappresentano collegamenti ed è quindi lecito aggregarli.

L'ordine, la leggibilità e l'aspetto generale dei diagrammi sono d'importanza cruciale. Si ricordi sempre che la maggior parte dei diagrammi viene creata per essere letta da qualcun'altro. Pertanto, qualunque sia lo stile adottato, sono vitali la chiarezza e la comprensibilità.

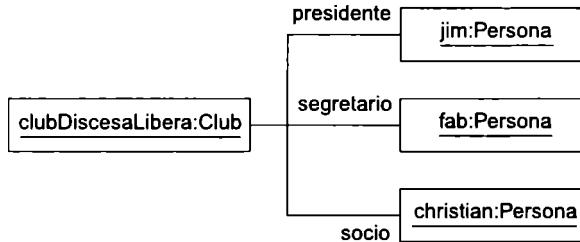


Figura 9.4

9.4 Cos'è un'associazione?

Le associazioni sono relazioni tra classi. Così come i collegamenti connettono gli oggetti, le associazioni connettono le classi. Il concetto chiave è che, se esiste un collegamento tra oggetti, allora deve anche esistere un'associazione tra le loro classi, in quanto un collegamento è l'istanziazione di un'associazione, così come un oggetto è l'istanziazione di una classe.

La Figura 9.5 illustra la relazione tra classi e oggetti, e quella tra collegamenti e associazioni.

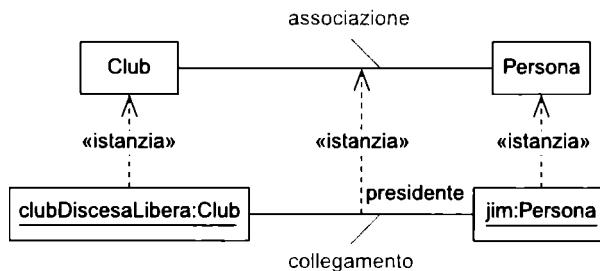


Figura 9.5

Se per avere un collegamento tra oggetti, occorre che esista un'associazione tra le loro classi, allora il collegamento *dipende* dall'associazione e lo si rappresenta con una relazione di dipendenza (freccia tratteggiata) di cui si parlerà in dettaglio nel Paragrafo 9.5.

Per rendere esplicito il significato della dipendenza tra collegamenti e associazioni, si utilizza una dipendenza con stereotipo «*istanzia*».

La semantica delle associazioni, nella loro forma base, non rifinita, è semplice: un'associazione tra classi indica che si possono avere dei collegamenti tra gli oggetti di quelle classi.

Esistono forme di associazione più complesse (aggregazione e composizione) che verranno esaminate nel Capitolo 16, dove viene trattato il flusso di lavoro della progettazione.

9.4.1 Sintassi per le associazioni

Le associazioni possono essere indicate con:

- un nome dell'associazione;
- nomi dei ruoli;
- molteplicità;
- navigabilità.

I nomi delle associazioni dovrebbero essere dei verbi o delle frasi verbali, poiché indicano un'azione che l'oggetto origine esegue sull'oggetto destinazione. Il nome può essere preceduto o seguito da una freccetta nera che indica in quale direzione si deve leggere il nome dell'associazione. I nomi sono scritti in CamelCase, iniziando con una lettera minuscola.

L'associazione dell'esempio in Figura 9.6 si legge: “un'Azienda impiega molte Persone”. Anche se la freccia indica la direzione di lettura dell'associazione, è anche sempre possibile leggerla in senso contrario. Così, dalla Figura 9.6 si può anche affermare: “ogni Persona è impiegata da una sola Azienda” in un dato istante.

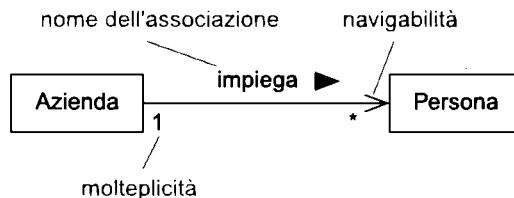


Figura 9.6

In alternativa, si possono assegnare nomi ai ruoli delle classi a una, o entrambe, le estremità dell'associazione. Questi nomi indicano i ruoli che gli oggetti di quelle classi rivestono quando sono collegati da istanze di quell'associazione. Nella Figura 9.7 si vede che, quando saranno collegati da istanze dell'associazione, l'oggetto Azienda avrà il ruolo datoreLavoro, mentre l'oggetto Persona avrà il ruolo dipendente. I ruoli dovrebbero essere nomi o frasi nominali, in quanto descrivono il ruolo che gli oggetti possono ricoprire.

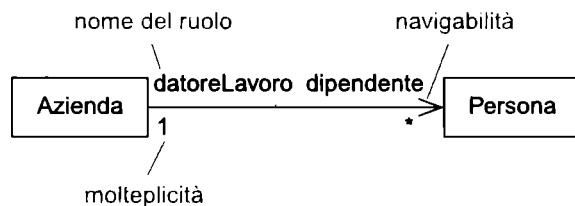


Figura 9.7

Le associazioni possono essere indicate o con il nome di associazione, oppure con i nomi dei ruoli. Anche se, in teoria, è lecito indicare sia l'uno che gli altri sulla stessa associazione, è pleonastico ed è decisamente poco elegante.

Il segreto per scegliere dei buoni nomi di associazioni e di ruoli è che essi devono suonare bene quando vengono letti. Dalla Figura 9.6 si legge che un'Azienda impiega molte Persone, e questo significa davvero qualcosa. Leggendo l'associazione all'inverso, si può dire che una Persona è impiegata da una sola Azienda in un dato istante, il che si legge altrettanto bene. Analogamente, i nomi dei ruoli nella Figura 9.7 indicano chiaramente la parte che avranno gli oggetti di queste classi quando saranno collegati in questo particolare modo.

9.4.2 Molteplicità

I vincoli sono uno dei tre meccanismi di estendibilità dell'UML. La molteplicità è il primo tipo di vincolo qui discusso ed è di gran lunga il più usato. La molteplicità limita il numero di oggetti di una classe che possono partecipare in una relazione in un dato istante. La frase “in un dato istante” è fondamentale per comprendere le molteplicità.

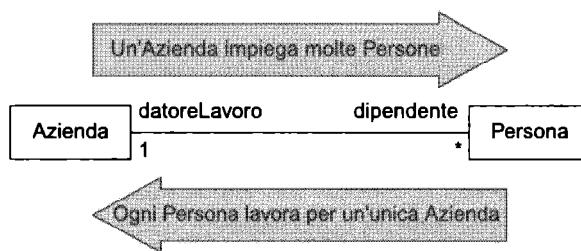


Figura 9.8

Esaminando la Figura 9.8, si vede che in ogni dato istante un oggetto *Persona* è dipendente di esattamente un oggetto *Azienda*. Tuttavia, nel corso del tempo, un stesso oggetto *Persona* potrebbe essere dipendente da una successione di diversi oggetti *Azienda*. Un altro aspetto interessante della Figura 9.8 è che un oggetto *Persona* non può mai essere disoccupato: è sempre dipendente di esattamente un oggetto *Azienda*. Di conseguenza, in questo modello, tale vincolo esprime due regole di *business*:

1. gli oggetti *Persona* possono essere dipendenti di un'Azienda per volta;
2. gli oggetti *Persona* devono sempre essere dei dipendenti.

Se questi siano dei vincoli ragionevoli o meno, dipende esclusivamente dai requisiti del sistema che viene modellato, ma le effettive asserzioni del modello sono queste due.

Si può capire che i vincoli di molteplicità sono importantissimi, perché codificano nel modello delle regole di *business* fondamentali. Però queste regole sono “sepolti” nei dettagli del modello. Alcuni modellatori chiamano “banalizzare” questo modo di na-

scondere regole e requisiti fondamentali del *business*. Per una discussione molto più approfondita di questo punto, si consiglia di visitare il sito *Literate Modeling* degli Autori presso www.literatemodeling.com.

La molteplicità viene specificata come una lista d'intervalli separati da virgole, ognuno nella forma:

minimo..massimo

dove minimo e massimo sono degli interi o delle espressioni che producono un valore intero.

Tabella 9.1

Ornamento	Significato
0..1	Zero o 1
1	Esattamente 1
0..*	Zero o più
*	Zero o più
1..*	1 o più
1..6	da 1 a 6
1..3,7..10,15,19..*	da 1 a 3, oppure da 7 a 10, oppure esattamente 15, oppure 19 o più

Se la molteplicità non viene indicata esplicitamente, allora è indefinita: nell'UML non esiste un valore predefinito per la molteplicità. In effetti, un errore molto comune nella modellazione UML è quello di presumere che una molteplicità indefinita sottintenda una molteplicità di 1. La Tabella 9.1 riporta alcuni esempi di sintassi per la molteplicità. L'esempio nella Figura 9.9 evidenzia che la molteplicità è un vincolo potente, e che di solito ha un impatto notevole sulla semantica di *business* del modello.

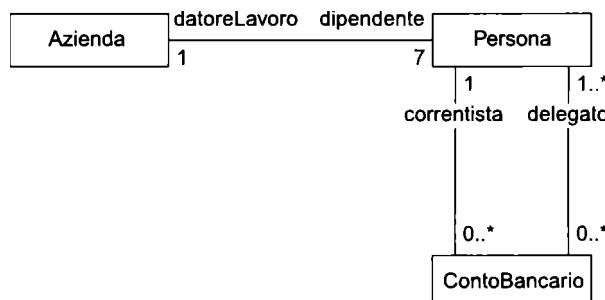


Figura 9.9

Leggendo attentamente l'esempio, si osserva che:

- un'Azienda può avere esattamente sette dipendenti;
- una Persona può essere dipendente di esattamente un'Azienda (ovvero: in questo modello una Persona non svolge più di un lavoro alla volta);
- un ContoBancario può avere un solo correntista;
- un ContoBancario può avere uno o più delegati;
- una Persona può essere correntista di zero o più di un ContoBancario;
- una Persona può essere delegato di zero o più di un ContoBancario.

Leggendo un modello è vitale ricavarne esattamente il significato letterale, piuttosto che fare assunzioni o inventarsi significati inesistenti. Si dice che “il modello deve essere letto così com'è stato scritto”.

Per esempio, la Figura 9.9 dichiara che un'Azienda deve avere esattamente sette dipendenti, né più, né meno. Quasi tutti lo considererebbero un vincolo perlomeno strano, se non addirittura errato (a meno che non si tratti di un'azienda davvero peculiare), però questo è quanto dice il modello. Non si deve mai ignorare questo aspetto.

9.4.2.1 La molteplicità nei modelli

Si discute abbastanza se la molteplicità debba o meno essere indicata nei modelli dell'analisi. È meglio che lo sia, perché la molteplicità esprime delle regole di *business*, dei requisiti e dei vincoli che potrebbero evidenziare delle assunzioni non previste concernenti il *business*. È ovviamente preferibile che queste vengano a galla, e messe in discussione, quanto prima possibile. Per fare un esempio, si veda la Figura 9.10.

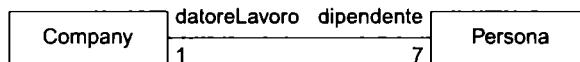
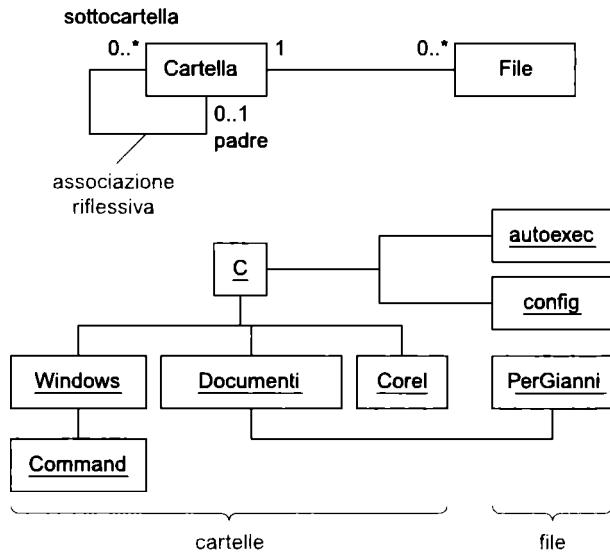


Figura 9.10

Il modello afferma che una Persona non può mai essere disoccupata, che l'Azienda deve avere esattamente sette dipendenti in ogni momento. Ma tutto questo è corretto? Forse sì, forse no: dipende dal sistema che si vuole modellare. Ma se non fossero state indicate le molteplicità, queste assunzioni non sarebbero mai venute a galla, e magari nessuno avrebbe pensato di verificarle.

9.4.2.2 Associazioni riflessive

È piuttosto comune che una classe abbia un'associazione con sé stessa. Questo significa che oggetti della classe possono avere collegamenti con altri oggetti della stessa classe, e la si chiama associazione riflessiva. La Figura 9.11 illustra un buon esempio di associazione riflessiva. Ogni oggetto Cartella può avere collegamenti a zero o più altri oggetti Cartella, i quali svolgono il ruolo di sottocartella, e a zero o uno oggetto Cartella, il quale svolge il ruolo di padre. Inoltre, ogni oggetto Cartella può essere collegato a zero o più oggetti File.

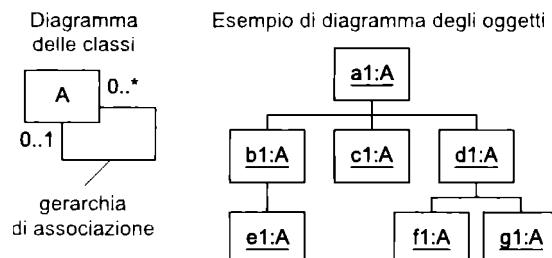
**Figura 9.11**

Questo modello rappresenta piuttosto bene la struttura di *directory*, tipica di molti file system.

Nella metà superiore della Figura 9.11 si vede il diagramma delle classi, mentre la metà inferiore illustra un esempio di un diagramma degli oggetti corrispondente a quello delle classi.

9.4.2.3 Gerarchie e reti

Creando i modelli si scopre di frequente che gli oggetti si organizzano in gerarchie o reti. Una gerarchia possiede un oggetto radice e ogni altro nodo della gerarchia ha un solo oggetto al di sopra di sé. Gli alberi di cartelle sono per loro natura delle gerarchie, così come la distinta base in ingegneria o gli elementi dei documenti XML e HTML. Le gerarchie organizzano gli oggetti in modo molto ordinato e strutturato, ma piuttosto rigido. La Figura 9.12 riporta un esempio di gerarchia.

**Figura 9.12**

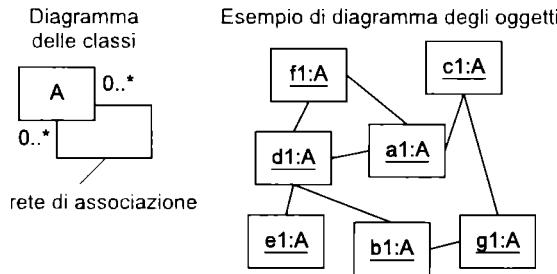


Figura 9.13

Con le reti, invece, spesso non si ha un oggetto radice, sebbene ciò non sia precluso. Nelle reti ogni oggetto può essere direttamente connesso a molti altri. La rete è una struttura molto più flessibile, in cui non esiste un concetto隐式的 di livello “superiore” o “inferiore”, e può accadere che nessun nodo abbia la supremazia sugli altri. Il *world wide web* costituisce una complessa rete di nodi, come nella rappresentazione semplificata riportata nella Figura 9.13.

Per illustrare gerarchie e reti, prendiamo in esempio i prodotti: distinguiamo due astrazioni di base:

- **TipoProdotto:** un tipo di prodotto quale “stampante a getto d’inkiostro”;
- **ArticoloProdotto:** una specifica stampante a getto d’inkiostro, numero di serie 0001123430.

I **TipoProdotto** solitamente sono strutturati in reti: un **TipoProdotto computer** potrebbe comprendere dei **TipoProdotto** quali CPU, schermo, tastiera, mouse, adattatore grafico e altri. Ognuno di questi denota un tipo di prodotto, ma non un prodotto specifico particolare, e questi tipi di prodotti possono partecipare anche in altri **TipoProdotto compositi**, in distinte configurazioni di computer.

Invece, se consideriamo gli **ArticoloProdotto**, che sono delle istanze specifiche di un **TipoProdotto**, ogni **ArticoloProdotto**, come una data CPU, è un entità singola e fisica, che può essere venduta e consegnata una sola volta. Le istanze di **ArticoloProdotto** sono quindi organizzate in una gerarchia.

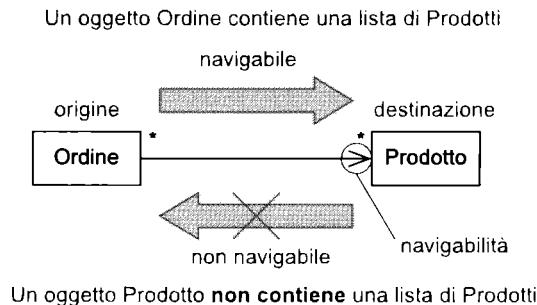
9.4.3 Navigabilità

La navigabilità indica che è possibile spostarsi da un qualsiasi oggetto della classe origine a uno o più oggetti della classe destinazione, a seconda della molteplicità. La navigabilità può essere letta come: “i messaggi possono essere inviati solo nella direzione della freccia”. Gli oggetti **Ordine** nella Figura 9.14 possono inviare messaggi agli oggetti **Prodotto**, ma non è vero il contrario.

La navigabilità è un buon mezzo per ridurre l’interdipendenza tra classi, che è uno degli scopi della buona analisi e progettazione OO. L’associazione unidirezionale tra **Ordine** e **Prodotto** stabilisce che si può passare facilmente da oggetti **Ordine** a oggetti

Prodotto, ma non esiste navigabilità nella direzione opposta dagli oggetti Prodotto agli oggetti Ordine. Di conseguenza, gli oggetti Prodotto non possono sapere se fanno parte di un dato Ordine, e quindi non hanno interdipendenza con la classe Ordine. È possibile che una relazione sia percorribile anche nella direzione lungo la quale non è navigabile, ma in modo probabilmente poco efficiente.

Nell'esempio della Figura 9.14, anche se non è possibile risalire *direttamente* da un Prodotto al suo Ordine, è comunque possibile individuare l'Ordine associato a un particolare oggetto Prodotto eseguendo una ricerca sequenziale su tutti gli oggetti Ordine. In questo modo si sarebbe indirettamente percorso la relazione in una direzione non-navigabile, ma con un costo di elaborazione piuttosto elevato.



Un oggetto Prodotto **non contiene** una lista di Prodotti

Figura 9.14

A questo punto, è utile rammentare che le relazioni prive di frecce sono sempre bidirezionali. In assenza di frecce, la logica suggerirebbe che la relazione non sia navigabile in nessun senso, ma, pensandoci bene, una relazione non-navigabile non sarebbe una relazione! Per questa ragione, e per semplificare la notazione, l'UML definisce una relazione priva di frecce come bidirezionale.

Se l'estrema destinazione di una relazione indica un ruolo, allora gli oggetti della classe origine possono riferirsi agli oggetti della classe destinazione mediante il nome di quel ruolo.

Nell'implementazione con linguaggi di programmazione OO, la navigabilità implica che l'oggetto origine conserva un riferimento all'oggetto destinazione e che può utilizzare questo riferimento per inviare messaggi all'oggetto destinazione. In un diagramma degli oggetti, questo può essere indicato mediante un collegamento unidirezionale associato a un messaggio.

9.4.4 Associazioni e attributi

Esiste una correlazione forte tra le associazioni delle classi e gli attributi delle classi.

Un'associazione tra una classe origine e una destinazione indica che oggetti della prima possono avere riferimenti agli oggetti della seconda. Un altro modo di vedere l'associazione è che la classe sorgente possiede uno pseudo-attributo il cui tipo è la classe destinazione. Tramite questo attributo, un oggetto della classe origine può far riferimento a un oggetto della classe destinazione, come si vede nella Figura 9.15.



Se una relazione navigabile ha un ruolo, allora è come se la classe origine avesse uno pseudo-attributo con lo stesso nome del ruolo e con lo stesso tipo della classe destinazione.

Figura 9.15

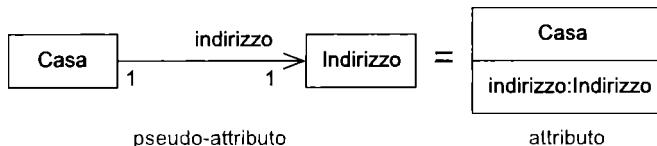


Figura 9.16

Non essendovi nei linguaggi di programmazione OO comunemente usati dei costrutti specifici per le associazioni, quando il codice viene generato automaticamente da un modello in UML, le associazioni uno-a-uno vengono trasformate in attributi della classe origine.

Nella Figura 9.16, il codice generato contiene una classe **Casa** contenente un attributo **indirizzo**, di tipo **Indirizzo**. Si noti che il nome del ruolo diviene il nome dell'attributo e che la classe sull'estremo destinazione dell'associazione diventa la classe dell'attributo. Il seguente codice Java è stato generato partendo dal modello di cui sopra:

```

public class Casa
{
    private Indirizzo indirizzo;
}
  
```

Dal codice si vede che esiste una classe **Casa**, che ha un attributo privato, chiamato **indirizzo**, che è di tipo **Indirizzo**.

Le molteplicità superiori a 1 si realizzano come:

- un attributo di tipo vettore (un costrutto supportato da molti linguaggi); oppure
- un attributo che abbia per tipo una qualche classe contenitore.

Le classi contenitore sono delle classi le cui istanze sono dedicate a contenere riferimenti ad altri oggetti. Un tipico esempio di classe contenitore in Java è **Vector**, ma ve ne sono molti altri.

Il concetto di pseudo-attributo funziona bene con le relazioni uno-a-uno e con quelle uno-a-molti, ma inizia a dare problemi quando si prendono in considerazione le relazioni molti-a-molti. Nel Capitolo 16 si vedrà come vengono implementate queste ultime.

Si usano le associazioni solo quando la classe destinazione è una parte importante del modello, altrimenti le relazioni si rappresentano mediante attributi. Le classi importanti sono quelle che descrivono il dominio del problema, mentre quelle non importanti sono componenti di libreria, come le classi per gestire Stringhe, Date e Ore.

Entro certi limiti la scelta tra associazioni esplicite e attributi è una questione di stile, per cui l'approccio migliore sarà quello che risulta in un modello i cui diagrammi esprimono il problema in modo più chiaro e preciso. Spesso risalta meglio un'associazione con un'altra classe che non la stessa relazione espressa mediante un attributo, che risulta più difficile da vedere. Una molteplicità della destinazione superiore a 1 è un buon indizio che la classe destinazione è importante nel modello e, di conseguenza, si preferisce usare associazioni per rappresentare questi tipi di relazione.

Se la molteplicità è esattamente 1, la destinazione potrebbe in realtà essere una mera parte dell'origine, e potrebbe non valere la pena di rappresentarla come associazione: potrebbe essere meglio rappresentarla come attributo. Questo è in particolar modo vero quando la molteplicità è esattamente 1 a entrambi gli estremi della relazione (come nella Figura 9.16), dove né l'origine, né la destinazione possono esistere da sole.

9.4.5 Classi associazione

Un problema frequente della modellazione OO è una relazione molti-a-molti tra due classi, ma con qualche attributo che non può essere facilmente assegnato a nessuna delle due. La Figura 9.17 illustra un semplice esempio di questa situazione.

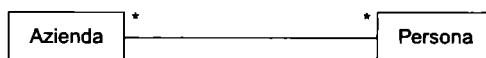


Figura 9.17

A prima vista sembrerebbe un modello del tutto innocuo:

- ogni oggetto Persona può essere dipendente di molti oggetti Azienda;
- ogni oggetto Azienda può impiegare molti oggetti Persona.

Però, cosa accade se si aggiunge una regola per cui ogni Persona percepisce uno stipendio da ogni Azienda in cui è impiegata? Dove sistemiamo l'attributo stipendio: nella classe Persona o nella classe Azienda?

Non si riesce a fare dello stipendio di una Persona un attributo della classe Persona, perché ogni istanza di Persona potrebbe lavorare per diverse Aziende percependo uno stipendio diverso da ciascuna di esse.

Analogamente, non si può fare dello stipendio di una Persona un attributo di Azienda, perché ogni istanza di Azienda impiega molte Persone tutte con stipendi potenzialmente diversi.

La risposta è che lo stipendio è *una proprietà dell'associazione stessa*. Per ogni associazione “impiega” che un oggetto Persona ha con un oggetto Azienda, c’è uno specifico stipendio per uno specifico rapporto di lavoro.

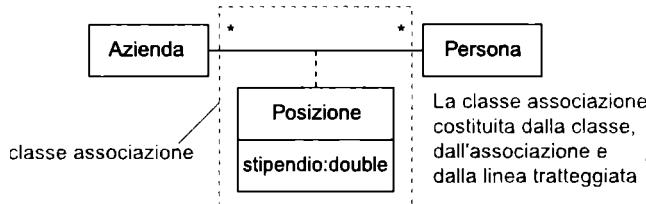


Figura 9.18

L'UML consente di modellare questa situazione con una classe associazione, come si vede nella Figura 9.18. È importante capire questa sintassi: molti pensano che la classe associazione sia semplicemente il rettangolo attaccato all'associazione, ma questo non è assolutamente vero. La classe associazione in realtà è costituita dalla linea dell'associazione (compresi tutti i nomi di ruolo e le molteplicità), dal rettangolo della classe e dalla linea tratteggiata verticale che li collega. In breve: tutto quanto si trova nell'area tratteggiata.

La classe associazione è, in effetti, un'associazione che è anche una classe. Essa non solo connette due classi, come un'associazione, ma definisce anche un insieme di caratteristiche proprie dell'associazione stessa. Le classi associazione possono avere attributi, operazioni e altre associazioni.

Le istanze dell'associazione sono in realtà *collegamenti*, dotati di attributi e operazioni, la cui identità individuale è stabilita *esclusivamente* dalle identità degli oggetti alle estremità del collegamento. Questo fatto limita la semantica della classe associazione, perché la si può usare solo se tra una data coppia di oggetti si ha, in ogni momento, un solo collegamento univocamente determinato. Questo semplicemente perché ogni collegamento, che è un'istanza della classe associazione, deve possedere una sua identità unica. Utilizzare una classe associazione nella Figura 9.18, indica che si vincola il modello in modo che, dato un certo oggetto **Persona** e dato un certo oggetto **Azienda**, possa esistere un solo oggetto **Posizione**. In altre parole, ogni **Persona** potrà avere una sola **Posizione** in una data **Azienda**.

Se, invece, la situazione richiede che un dato oggetto **Persona** debba poter avere più di una **Posizione** in una specifica **Azienda**, allora non si può usare una classe associazione, semplicemente perché le semantiche non corrispondono!

Poiché si ha comunque bisogno di mettere da qualche parte lo stipendio di ogni oggetto **Persona**, si reifica (rende reale) la relazione trasformandola in una vera classe. Nella Figura 9.19 si rappresenta **Posizione** come una classe normale e si vede che quindi una **Persona** può avere diverse **Posizioni**, ognuna presso esattamente una sola **Azienda**.

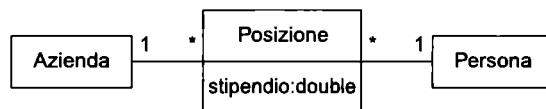


Figura 9.19

A essere sinceri, molti modellatori non comprendono bene la differente semantica tra le classi associazione e le relazioni reificate, e le usano spesso in modo intercambiabile. La differenza è peraltro semplice: si possono usare le classi associazione solo quando ogni collegamento ha un'identità univoca. Basta rammentarsi che l'identità dei collegamenti viene determinata dalle identità degli oggetti alle estremità di ogni collegamento.

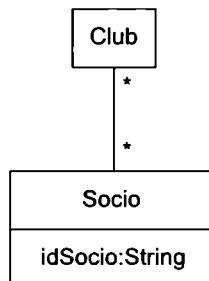
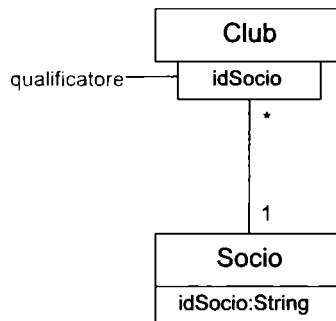


Figura 9.20

9.4.6 Associazioni qualificate

Le associazioni qualificate riducono un'associazione molti-a-molti in una del tipo molti-a-uno, specificando un unico oggetto (o gruppo) scelto da un insieme degli oggetti destinazione. Sono elementi di modellazione molto utili per illustrare come si possono reperire degli oggetti specifici di un insieme o navigare fino a essi.

Si esamini il modello della Figura 9.20. Un oggetto Club è collegato a un insieme di oggetti Socio, un oggetto Socio è, a sua volta, collegato a un insieme di oggetti Club. Ci si pone queste domande: dato un oggetto Club collegato a un insieme di oggetti Socio, come si può navigare fino a uno specifico oggetto Socio? Ovviamente, per reperire un oggetto Socio che fa parte dell'insieme, occorre avere una qualche chiave univoca.



La combinazione {Club, idSocio} specifica un unico destinatario

Figura 9.21

Ne vengono in mente diverse (nome, numero della carta di credito, codice fiscale), ma nell'esempio di cui sopra ogni oggetto Socio possiede già un valore univoco: quello dell'attributo idSocio. Si può, quindi, utilizzare questo attributo come chiave di ricerca per il modello.

Nel modello, si indica questa chiave di ricerca aggiungendo alla estremità Club della associazione un qualificatore che indica la chiave univoca e risolve la relazione molti-a-molti, come si vede in Figura 9.21

Le associazioni qualificate sono un ottimo modo per indicare la chiave univoca che si intende utilizzare per selezionare un oggetto specifico da un insieme completo. I qualificatori normalmente si riferiscono a un attributo della classe di destinazione, ma possono anche essere un'espressione generica, purché risulti chiara e leggibile.

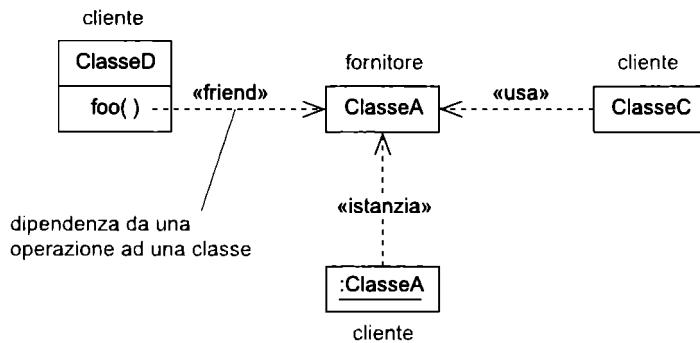
9.5 Cos'è una dipendenza?

The UML Reference Manual [Rumbaugh 1] afferma che: “Una dipendenza è una relazione tra due elementi, dove un cambiamento a uno di essi (il fornitore) può influenzare o fornire delle informazioni necessarie all'altro (il cliente)”. In altre parole, il cliente dipende in qualche modo dal fornitore. Si usano le dipendenze per descrivere relazioni tra classificatori, in cui un classificatore dipende in qualche modo da un altro. Questa relazione non è propriamente un'associazione.

Tabella 9.2

Tipo	Significato
Uso	Il cliente utilizza alcuni dei servizi resi disponibili dal fornitore per implementare il proprio comportamento.
Astrazione	Una relazione tra cliente e fornitore in cui il fornitore è più astratto del cliente
Accesso	Il fornitore assegna al cliente un qualche tipo di permesso di accesso al proprio contenuto: in questo modo il fornitore limita e controlla gli accessi al proprio contenuto
Binding	Una dipendenza di tipo avanzato che si vedrà nel paragrafo 15.6. È importante solo quando il linguaggio di sviluppo consente l'uso di tipi parametrici (i <i>template</i>); pertanto, è utile in C++, ma non in Java, Visual Basic o Smalltalk

Per esempio, si può passare un oggetto di una classe come parametro a un metodo di un'altra classe. Esiste certamente un qualche tipo di relazione tra le due classi coinvolte, ma non si tratta però di un'associazione. Per modellare questo tipo di relazioni, si possono utilizzare le dipendenze (specializzate con certi stereotipi predefiniti). Abbiamo già visto un tipo di dipendenza, la relazione “«istanzia»”, ma ve ne sono molti altri. Nelle prossime sezioni si esamineranno gli stereotipi comuni per le dipendenze.

**Figura 9.22**

L'UML 1.4 specifica quattro tipi base di dipendenze. Le dipendenze esistono non solo tra le classi, ma anche tra:

- *package e package;*
- *oggetti e classi.*

Possono anche esistere tra un'operazione e una classe, ma è piuttosto raro vederle esplicitate in un diagramma. La Figura 9.22 illustra alcuni esempi dei diversi tipi di dipendenza, a cui è peraltro dedicato il resto di questo capitolo.

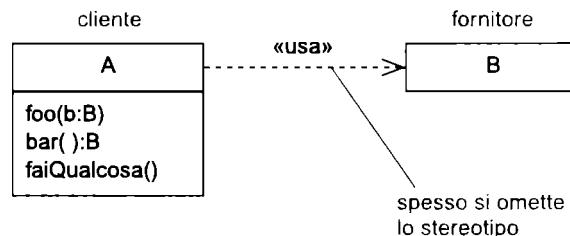
Il più delle volte, per descrivere una dipendenza si usa una semplice linea tratteggiata priva di ornamenti, senza indicarne il tipo. L'UML afferma che il tipo di dipendenza risulta di solito chiaro dal contesto, anche senza indicarne lo stereotipo, e questo è spesso vero. Tuttavia, nel caso si desideri, o sia necessario, essere più specifici sul tipo di dipendenza, l'UML definisce un insieme completo di stereotipi utilizzabili.

9.5.1 Dipendenze di uso

Tutte le dipendenze riferite all'uso sono di semplice comprensione. Lo stereotipo più usato è «usa», che significa che il cliente fa un qualche uso del fornitore.

9.5.1.1 «usa»

La dipendenza «usa» è la più comune tra classi. Se si vede una freccia tratteggiata che indica una dipendenza, ma priva di stereotipo, si può presumere che si tratti di una dipendenza «usa». La Figura 9.23 illustra due classi, A e B, collegate da una dipendenza «usa».

**Figura 9.23**

Questa dipendenza viene generata dai seguenti casi.

- Un'operazione della classe A ha bisogno di un parametro della classe B.
- Un'operazione della classe A restituisce un valore della classe B.
- Un'operazione della classe A utilizza un oggetto della classe B da qualche parte nella sua implementazione, ma non come un proprio attributo.

```
Class A
{
    ...
    void faiQualcosa()
    {
        B mioB = new B();
        // Usa mioB in qualche
        modo
        ...
    }
}
```

Figura 9.24

I primi due casi sono banali, mentre il terzo è più interessante. Questo caso si manifesta quando un metodo della classe A crea dinamicamente un oggetto temporaneo della classe B.

Nella Figura 9.24 si vede un frammento di codice Java che esemplifica il caso.

Sebbene sia possibile usare la sola dipendenza «usa» in tutti e tre i suddetti casi, esistono stereotipi più specializzati applicabili per queste dipendenze. Si possono rappresentare più accuratamente i casi 1 e 2 mediante una dipendenza «parametro», mentre il caso 3 è più adatto a una dipendenza «chiama». Tuttavia, capita di rado (se non mai) che sia richiesto questo livello di dettaglio in un modello UML. La maggior parte dei modellatori ritiene più facile, e più comprensibile, limitarsi all'utilizzo della sola dipendenza «usa» tra classi, come mostrato prima.

9.5.1.2 «chiama»

È una dipendenza tra operazioni: l'operazione cliente richiama l'operazione fornitore. Questo tipo di dipendenza viene utilizzata di rado nella modellazione UML. Compete a un livello di dettaglio superiore a quello oltre il quale la maggior parte dei modellatori non è disposta a spingersi. Inoltre, sono ben pochi gli strumenti CASE che al momento supportano dipendenze tra operazioni.

9.5.1.3 «parametro»

Questa dipendenza connette un'operazione a una classe. Il fornitore è un parametro, o il valore restituito, di un'operazione del cliente. Anche questo tipo di dipendenza non viene usato molto.

9.5.1.4 «invia»

Il cliente invia il fornitore (che deve essere un segnale) a un destinatario non specificato. Si esamineranno i segnali nel Paragrafo 13.10, per ora basti pensare a essi semplicemente come delle classi usate per trasferire dati dal cliente al destinatario.

9.5.1.5 «istanzia»

Il cliente è un'istanza del fornitore.

9.5.2 Dipendenze di astrazione

Le dipendenze di astrazione modellano dipendenze tra elementi con un diverso livello d'astrazione. Per esempio, una classe in un modello dell'analisi e la stessa classe nel modello della progettazione.

9.5.2.1 «origine»

Spesso si utilizza la dipendenza «origine» per mostrare la connessione generale tra elementi che si trovano in punti diversi del processo di sviluppo. Per esempio, il fornitore potrebbe essere la vista di analisi di una classe, mentre il cliente è una vista di progettazione, più dettagliata, della stessa classe. Un altro esempio di «origine» è quello esistente tra un requisito funzionale, del tipo: “Lo sportello bancomat consentirà il prelievo di contante fino al limite di utilizzo della carta”, e il caso d’uso che sviluppa questo requisito.

9.5.2.2 «rifinisce»

Laddove la dipendenza «origine» è puramente di significato storico e si manifesta di solito tra modelli separati, la dipendenza «rifinisce» può esistere tra elementi dello stesso modello. Per esempio, si possono avere due versioni di una classe in un modello, di cui una ottimizzata per le prestazioni. Siccome l’ottimizzazione delle prestazioni è un raffinamento, lo si può modellare come una dipendenza «rifinisce» tra due elementi dello stesso modello, accompagnandola con una nota che ne spiega la natura.

9.5.2.3 «deriva-da»

Si usa questo stereotipo quando si vuole esplicitare che un elemento può essere derivato in qualche modo da un altro. Per esempio, avendo una classe *ContoBancario* che contiene una lista di *Transazioni*, ognuna contenente una *Quantità* di denaro, si può sempre calcolare il saldo corrente al momento sommando la *Quantità* per tutte le *Transazioni*. Esistono tre modi, illustrati nella Tabella 9.3, per mostrare che il saldo del conto (una *Quantità*) può essere derivato.

Tutti questi modi di indicare che il saldo può essere derivato sono equivalenti, sebbene il primo modello della Tabella 9.3 sia il più esplicito.

9.5.3 Dipendenze di accesso

Le dipendenze di accesso concernono la capacità di un elemento di accedere a un altro elemento.

9.5.3.1 «accede»

Questa è una dipendenza tra *package*. I *package* sono usati in UML per raggruppare degli elementi. La dipendenza «accede» indica che un *package* ha l’accesso a tutto il contenuto pubblico di un altro *package*. Tuttavia, ogni *package* definisce uno proprio spazio dei nomi che rimane distinto anche quando si usa «accede». Quindi gli elementi nel *package* cliente della dipendenza devono, comunque, usare i *pathname* per riferirsi agli elementi nel *package* fornitore. Questo argomento viene approfondito nel Capitolo 11.

Tabella 9.3

Modello	Descrizione
<pre> classDiagram class ContoBancario class Transazione class Quantità ContoBancario "1" -- "0..*" Transazione : <<deriva-dai>> Transazione "1" -- "1" Quantità : saldo 1 </pre>	<p>La classe Quantità ha un'associazione derivata con ContoBancario, in cui Quantità svolge il ruolo di saldo di ContoBancario.</p> <p>Questo modello evidenzia che saldo deriva dalla collezione delle Transazione di ContoBancario.</p>
<pre> classDiagram class ContoBancario class Transazione class Quantità ContoBancario "1" -- "0..*" Transazione : /saldo Transazione "1" -- "1" Quantità : saldo 1 </pre>	<p>In questo caso si usa una barra inclinata (<i>slash</i>) nel nome del ruolo, per indicare che la relazione tra ContoBancario e Quantità è di tipo <i>deriva-dai</i>.</p> <p>È una forma meno esplicita, in quanto non mostra da cosa sia derivato saldo</p>
<pre> classDiagram class ContoBancario class Transazione class Quantità ContoBancario "1" -- "0..*" Transazione : /saldo:Quantità Transazione "1" -- "1" Quantità : saldo 1 </pre>	<p>Qui si mostra che saldo è un attributo derivato: lo si indica con una barra inclinata davanti al nome dell'attributo.</p> <p>Questa è la forma più concisa di questa dipendenza.</p>

9.5.3.2 «importa»

Questa dipendenza è concettualmente simile a «accede». La differenza è che «importa» indica che lo spazio dei nomi del fornitore viene fuso con quello del cliente. Questo consente agli elementi del cliente di accedere a quelli del fornitore senza qualificare il nome con il nome del loro *package*. D'altra parte, questo può a volte produrre delle collisioni nello spazio dei nomi, quando un elemento del cliente ha lo stesso nome di uno del fornitore. Ovviamente, in questo caso si devono usare i *pathname* esplicativi per risolvere il conflitto. Questo argomento viene approfondito nel Capitolo 11.

9.5.3.3 «friend»

Questo tipo di dipendenza consente una violazione controllata della incapsulazione e, in linea generale, è meglio evitarne l'utilizzo. L'elemento cliente ha accesso all'elemento fornitore, qualunque sia la visibilità dichiarata per quest'ultimo. Sovraccarico si ha una dipendenza «friend» tra due classi strettamente affini, dove risulta vantaggioso (spesso per motivi di efficienza) per la classe cliente accedere ai membri privati del fornitore. Non tutti i linguaggi di programmazione supportano le dipendenze «friend»: C++ supporta «friend» tra classi, ma questa funzionalità è stata (forse saggiamente) esclusa sia da Java che da C#.

9.6 Riepilogo

In questo capitolo si è iniziato a studiare le relazioni, che sono il tessuto connettivo dei modelli UML. Si sono imparati i seguenti concetti.

1. Le relazioni sono connessioni dotate di semantica tra elementi dei modelli.
2. Le connessioni tra oggetti sono dette collegamenti.
 - Un collegamento esiste quando un oggetto ha un riferimento a un altro oggetto.
 - Gli oggetti realizzano il comportamento del sistema collaborando tra di loro:
 - si ha collaborazione quando gli oggetti si scambiano messaggi tramite i collegamenti;
 - quando un oggetto riceve un messaggio, esegue il metodo corrispondente.
 - I diversi linguaggi di programmazione OO realizzano i collegamenti in modi differenti.
3. I diagrammi degli oggetti mostrano gli oggetti e i loro collegamenti in un dato istante.
 - Sono delle istantanee di una parte del sistema OO in esecuzione.
 - Gli oggetti collegati possono assumere dei ruoli l'uno rispetto all'altro: il ruolo svolto in un collegamento definisce la semantica dell'oggetto relativa alla sua parte della collaborazione.
 - I collegamenti n-ari possono connettere più di due oggetti: sono disegnati come rombi con percorsi che connettono tutti gli oggetti interessati; non sono molto usati.
4. I percorsi sono linee che connettono gli elementi del modello UML:
 - stile ortogonale: segmenti orizzontali e verticali, connessi a angolo retto;
 - stile obliqui: segmenti dritti, ma inclinati;
 - si deve essere consistenti attenendosi a un solo stile, a meno che mescolandoli non si migliori la leggibilità del diagramma (di solito non migliora).
5. Le associazioni sono connessioni semantiche tra classi.
 - Se esiste un collegamento tra due oggetti, allora deve esistere un'associazione tra le classi di quegli oggetti.
 - I collegamenti sono istanze delle associazioni, così come gli oggetti sono istanze delle classi.
 - Le associazioni possono, optionalmente, essere indicate con:
 - nome dell'associazione:
 - può essere preceduto o seguito da una freccetta nera che indica in quale direzione debba essere letto il nome dell'associazione;

- dovrebbe essere un verbo o una frase verbale;
 - in CamelCase, iniziando con una lettera minuscola;
 - indicare il nome dell’associazione, o i nomi dei ruoli, ma non entrambi.
- Nomi dei ruoli a uno o entrambi gli estremi:
 - dovrebbero essere nomi o frasi nominali che descrivono la semantica del ruolo;
 - in CamelCase, iniziando con una lettera minuscola.
- Navigabilità a un solo estremo:
 - indicata con una freccia a un estremo della relazione: se non c’è freccia allora la relazione è bidirezionale;
 - la navigabilità indica che si può percorrere la relazione nella direzione della freccia;
 - potrebbe essere possibile percorrere la relazione in senso opposto, ma potrebbe risultare computazionalmente costoso.
- Molteplicità a uno o entrambi gli estremi:
 - indica il numero di oggetti di una classe che possono partecipare in una relazione in un dato istante;
 - gli oggetti vanno e vengono, ma la molteplicità vincola il numero degli oggetti in ogni istante;
 - viene indicata con un elenco d’intervalli separati da virgolette, per esempio: 0..1, 3..5;
 - non esiste un valore predefinito per la molteplicità: se la molteplicità non è indicata, allora è indefinita.
- Un’associazione tra due classi equivale a una classe che ha uno pseudo-attributo che contiene un riferimento a un oggetto dell’altra classe:
 - attributi e associazioni possono spesso essere usati in modo intercambiabile;
 - si usa un’associazione quando si ha una classe significativa come destinazione dell’associazione;
 - si usano gli attributi quando la classe di destinazione non è importante (per esempio, classi di libreria come String o Date).
- Una classe associazione è un’associazione che è anche una classe:
 - può possedere attributi, operazioni e relazioni;
 - si può usare una classe associazione quando vi sia al massimo un unico collegamento tra ogni coppia distinta di oggetti in ogni istante;

- se una coppia di oggetti può avere molti collegamenti reciproci in un qualsiasi dato istante, allora si reifica la relazione rimpiazzandola con una classe normale.
- Le associazioni qualificate definiscono un qualificatore per selezionare univocamente un oggetto dall'insieme di oggetti destinazione:
 - il qualificatore deve essere una chiave univoca dell'insieme di destinazione;
 - le associazioni qualificate riducono la molteplicità da molti-a-molti a molti-a-uno;
 - sono un mezzo utile per evidenziare gli identificatori univoci;
- 6. Le dipendenze sono relazioni per cui un cambiamento a un elemento fornitore può influenzare o fornire delle informazioni a un elemento cliente.
 - Il cliente dipende dal fornitore in qualche modo.
 - Le dipendenze sono indicate con frecce tratteggiate dal cliente verso il fornitore.
 - Dipendenze di uso:
 - «usa»: il cliente usa il fornitore come parametro, valore restituito o nella sua implementazione – dipendenza jolly;
 - «chiama»: un'operazione del cliente richiama un'operazione del fornitore;
 - «parametro»: il fornitore è un parametro o un valore restituito di un'operazione del cliente;
 - «invia»: il cliente invia il fornitore (che deve essere un segnale) a un destinatario non specificato;
 - «instanzia»: il cliente è un'istanza del fornitore.
 - Dipendenze di astrazione:
 - «origine»: il cliente è un'evoluzione storica del fornitore;
 - «rifinisce»: il cliente è una versione rifinita del fornitore;
 - «deriva-da»: il cliente può essere derivato in qualche modo dal fornitore:
 - si può esplicitare la relazione derivata con una dipendenza «deriva-da»;
 - si può mostrare la relazione derivata premettendo una sbarra inclinata al nome della relazione o del ruolo;
 - si possono mostrare gli attributi derivati premettendo una sbarra inclinata al nome dell'attributo.
 - Dipendenze di accesso:
 - «accede»: una dipendenza tra *package*, per cui il *package* cliente può accedere a tutti i contenuti pubblici del *package* fornitore, ma gli spazi dei nomi dei *package* rimangono separati;

- «importa»: una dipendenza tra *package*, per cui il *package* cliente può accedere a tutti i contenuti pubblici del *package* fornitore, ma gli spazi dei nomi vengono fusi insieme;
- «friend»: una violazione controllata della encapsulazione per cui l'elemento cliente ha accesso all'elemento fornitore, qualunque sia la visibilità dichiarata per quest'ultimo; poco supportata e da evitare, se possibile.

Ereditarietà e polimorfismo

10.1 Contenuto del capitolo

Il capitolo approfondisce i concetti chiave di ereditarietà (Sezione 10.3) e polimorfismo (Sezione 10.4). Prima di affrontare questi due argomenti, è però importante comprendere il concetto di generalizzazione, che viene discusso nel Paragrafo 10.2.

10.2 Generalizzazione

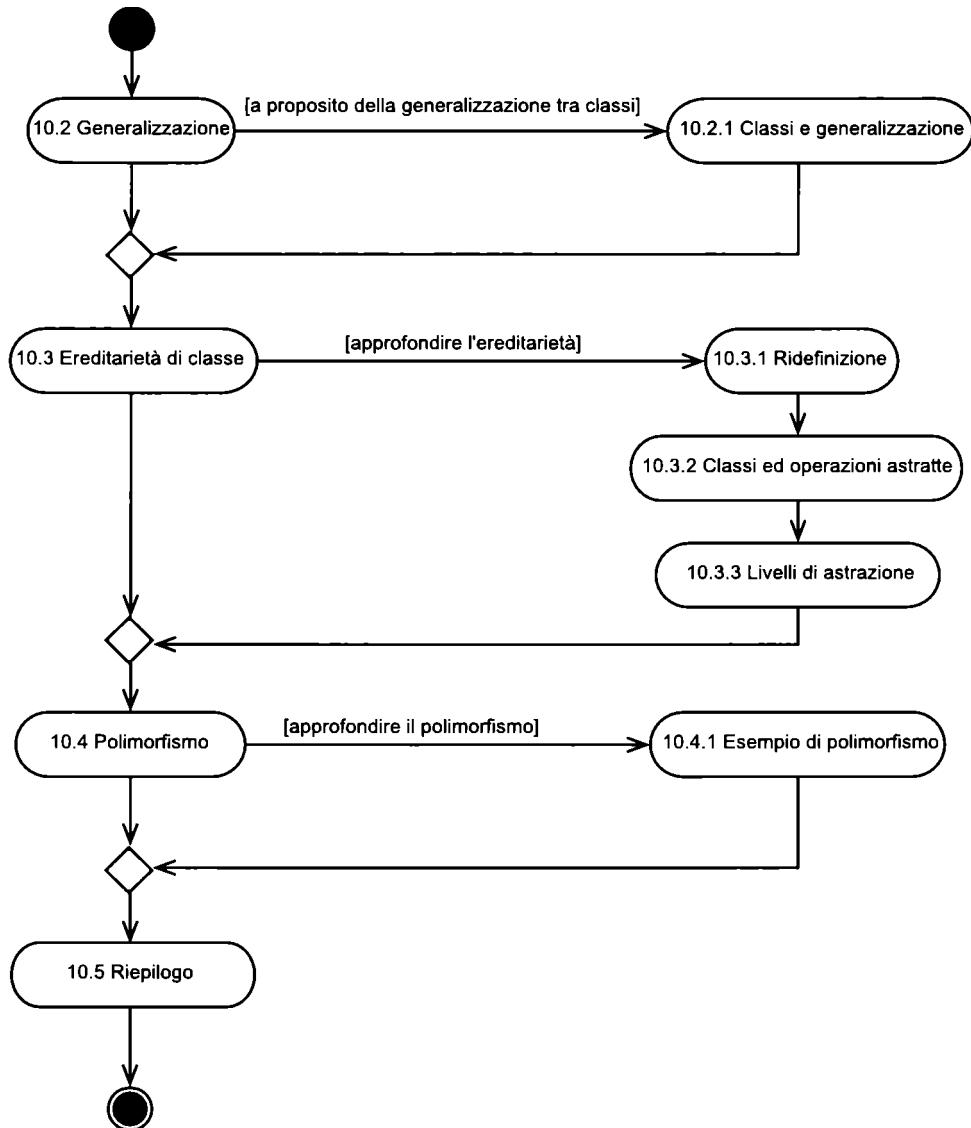
Prima di affrontare gli argomenti di ereditarietà e polimorfismo, occorre comprendere a fondo l'idea di generalizzazione. La generalizzazione è una relazione tra un elemento generico e uno più specializzato, in cui l'elemento più specializzato è completamente consistente con quello più generico, ma contiene più informazione.

I due elementi obbediscono al principio di sostituibilità: l'elemento specializzato può essere utilizzato ovunque sia richiesto o ammesso usare l'elemento più generico, senza rendere inconsistente il sistema. Questo tipo di relazione è ovviamente molto più forte di quanto non lo sia l'associazione. In effetti, la generalizzazione implica il massimo grado di interdipendenza tra due elementi.

10.2.1 Classi e generalizzazione

La generalizzazione è un'idea concettualmente semplice: i concetti di entità generica, come un albero, e di entità più specializzata, come una quercia, che è un tipo particolare d'albero, sono ben noti a tutti.

Si è già visto come applicare la generalizzazione ai casi d'uso e agli attori; ora si vedrà come possa essere applicata anche alle classi. In effetti, la generalizzazione può essere applicata a tutti i classificatori, e anche ad alcuni altri elementi di modellazione, quali le associazioni, gli stati, gli eventi e le collaborazioni.

**Figura 10.1**

Nella Figura 10.2, si osserva una classe di nome **Figura**: un concetto indubbiamente generico! Da questa si derivano classi figlie, sottoclassi, classi discendenti (tutti questi termini vengono usati comunemente), le quali sono varianti più specializzate dell'idea generica di **Figura**. Per il principio di sostituibilità, possiamo usare un'istanza di una di queste sottoclassi ovunque sia ammesso l'uso di un'istanza della superclasse **Figura**.

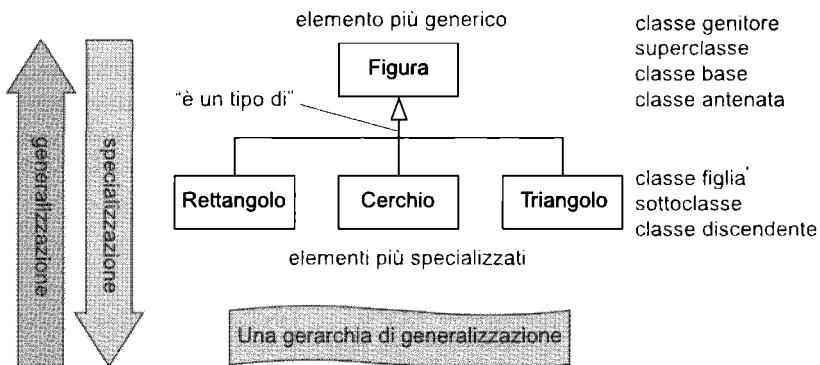


Figura 10.2

Come si potrà capire esaminando gli attributi e le operazioni specifiche di queste classi, la gerarchia illustrata può essere ottenuta con due distinte tecniche: un processo di specializzazione oppure uno di generalizzazione. Si ottiene tramite un processo di specializzazione, se durante l'analisi è stato dapprima individuato il concetto generico di Figura, da cui sono successivamente state i concetti specializzati relativi ai specifici tipi di figura. Si ottiene, invece, tramite un processo di generalizzazione, se durante l'analisi sono stati prima individuati i concetti specializzati (Rettangolo, Cerchio e Triangolo), e solo successivamente ci si è accorti che era possibile raggruppare alcune caratteristiche comuni in una superclasse più generica.

Gli analisti OO tendono a usare contestualmente sia la specializzazione, sia la generalizzazione. Tuttavia, l'esperienza suggerisce che è meglio imparare a individuare, durante le attività di analisi, i concetti generici quanto prima possibile.

10.3 Ereditarietà di classe

Quando si organizzano delle classi in una gerarchia di generalizzazione, come illustrato nella Figura 10.2, si implica l'esistenza di ereditarietà tra le classi; le sottoclassi ereditano tutte le caratteristiche delle proprie superclassi. Più precisamente, le sottoclassi ereditano:

- attributi;
- operazioni;
- relazioni;
- vincoli.

Le sottoclassi possono anche aggiungere nuove caratteristiche e ridefinire le operazioni delle superclassi. Le sezioni seguenti approfondiscono questi aspetti dell'ereditarietà.

10.3.1 Ridefinizione

Nell'esempio della Figura 10.3, le sottoclassi Figura, Rettangolo e Cerchio ereditano tutti gli attributi, operazioni e vincoli dalla superclasse Figura. In altre parole, queste caratteristiche sono implicitamente presenti nelle sottoclassi, nonostante non siano indicate esplicitamente sul modello. Si può correttamente dire che Rettangolo e Cerchio sono "un tipo di" Figura.

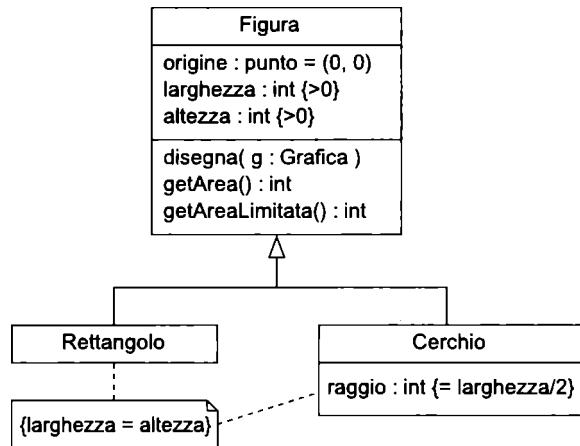


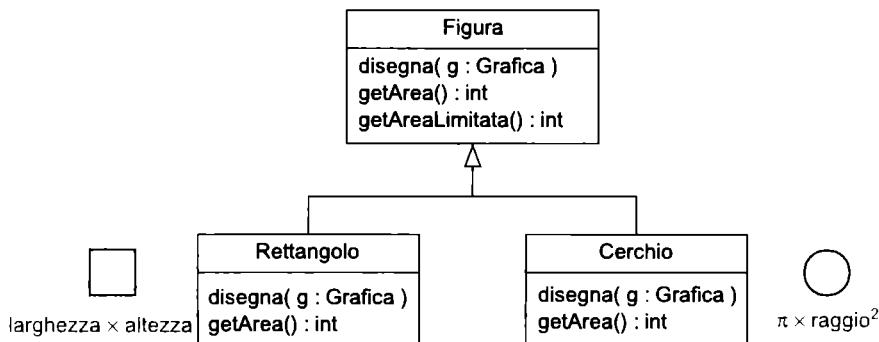
Figura 10.3

Si noti che le operazioni **disegna()** e **getArea()** definite in **Figura** non sono in alcun modo appropriate per le sottoclassi. Ci si aspetta che quando un oggetto **Rettangolo** riceve un messaggio **disegna()**, disegni un rettangolo, mentre se lo riceve un oggetto **Cerchio**, dovrebbe disegnare un cerchio. L'operazione **disegna()** predefinita, che entrambe le sottoclassi ereditano dalla loro classe genitore, non può essere adeguata. In effetti, questa operazione potrebbe anche non disegnare alcunché: dopotutto, che aspetto dovrebbe avere una **Figura** generica? Lo stesso ragionamento vale anche per **getArea()**: come si calcola l'area di una **Figura** generica?

Questi problemi indicano chiaramente che le sottoclassi devono poter modificare il comportamento della superclasse. **Rettangolo** e **Cerchio** necessitano di una loro implementazione delle operazioni **disegna()** e **getArea()** che ridefiniscono le operazioni predefinite ereditate dalla classe genitore, e che forniscano un comportamento più appropriato.

La Figura 10.4 mostra come si può risolvere questo problema: le sottoclassi **Rettangolo** e **Cerchio** hanno le loro proprie versioni delle operazioni **disegna()** e **getArea()**, le quali implementano comportamenti appropriati:

- **Rettangolo::disegna(g : Grafica)**: disegna un rettangolo;
- **Rettangolo::getArea() : int**: calcola e restituisce l'area del rettangolo;
- **Cerchio::disegna(g : Grafica)**: disegna un cerchio;
- **Cerchio::getArea() : int**: calcola e restituisce l'area del cerchio.

**Figura 10.4**

Per ridefinire un'operazione della superclasse, una sottoclasse deve avere una propria operazione con esattamente la stessa segnatura dell'operazione nella superclasse che si vuole ridefinire. L'UML definisce la segnatura di un'operazione come l'insieme di: nome dell'operazione, tipo restituito ed elenco ordinato dei tipi di tutti i parametri. I nomi dei parametri non contano, in quanto servono solo come riferimento di comodo a essi, da usarsi all'interno del corpo dell'operazione stessa, e quindi non fanno parte della segnatura.

Tutto ciò può sembrare semplice e ovvio, ma è meglio rammentare che i diversi linguaggi di programmazione possono definire “segnatura di un'operazione” in modo diverso. Per esempio, in C++ e Java il tipo restituito dall'operazione non fa parte della segnatura. Quindi, con questi linguaggi, se si ridefinisce un'operazione di una superclasse con un'operazione di una sottoclasse che sia del tutto identica, fatta eccezione per il tipo restituito, si ottiene un errore di compilazione o di interpretazione.

10.3.2 Classi e operazioni astratte

A volte si preferirebbe demandare l'implementazione di un'operazione alle sottoclassi specifiche. Nell'esempio precedente, questo è proprio il caso dell'operazione `Figura::disegna(g : Grafica)`. Non sembra plausibile implementare in modo sensato questa operazione nella classe **Figura** stessa, perché non si sa come debbano essere disegnate delle “figure” generiche: il concetto di “disegnare una figura” è troppo astratto per essere concretamente realizzata.

È possibile specificare che un'operazione è priva di implementazione; in questo caso si dice che l'operazione è astratta. In UML, le operazioni astratte sono semplicemente indicate scrivendone il nome in corsivo.

A pensarci bene, una classe con una o più operazioni astratte è incompleta, perché ci sono alcune operazioni prive d'implementazione. Pertanto, questa classe non può essere istanziata; in questo caso si dice che la classe è astratta. In UML, le classi astratte sono semplicemente indicate scrivendone il nome in corsivo.

Nell'esempio della Figura 10.5, esiste una classe astratta **Figura**, dotata di due operazioni astratte: `Figura::disegna(g : Grafica)` e `Figura::getArea() : int`. Entrambe le sottoclassi **Rettangolo** e **Cerchio** forniscono un'implementazione di queste due operazioni.

Sebbene Figura sia una classe incompleta e non possa essere istanziata, entrambe le sue sottoclassi forniscono l'implementazione mancante, sono complete e possono essere istanziate. Le classi che possono essere istanziate sono dette classi concrete.

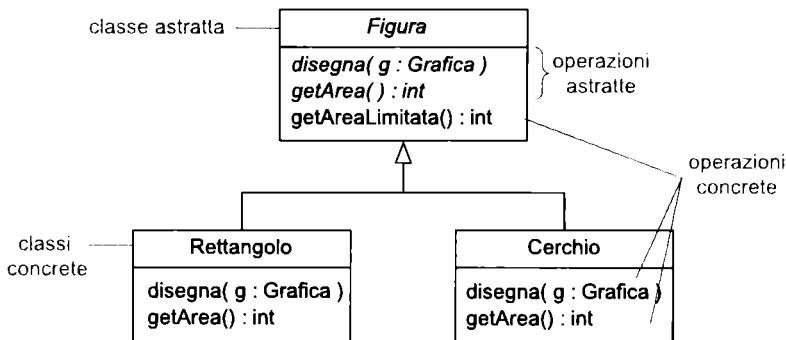


Figura 10.5

L'operazione `getAreaLimitata()` è un'operazione concreta di `Figura`, in quanto l'area limitata di qualunque `Figura` viene calcolata sempre allo stesso modo: è il prodotto della larghezza e dell'altezza della figura.

L'utilizzo di classi e operazioni astratte offre un paio di importanti vantaggi:

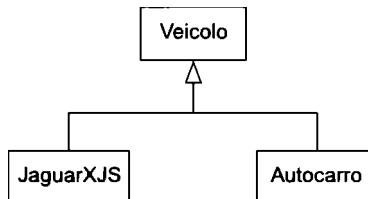
- Nella superclasse astratta `Figura` è possibile definire un insieme di operazioni astratte che tutte le sottoclassi sono tenute a implementare. Lo si pensi come un “contratto” che tutte le sottoclassi concrete di `Figura` dovranno rispettare.
- È possibile scrivere del codice che manipola oggetti di classe `Figura`, e poi sostituire opportunamente oggetti di classe `Cerchio`, `Rettangolo` o di altre sottoclassi di `Figura`. Secondo il principio di sostituibilità, il codice scritto per manipolare `Figura` deve funzionare anche con oggetti di una qualunque sottoclasse di `Figura`.

Questi vantaggi vengono approfonditi nel Paragrafo 10.4, dove si discute di polimorfismo.

10.3.3 Livelli di astrazione

Prima di affrontare il tema del polimorfismo, è meglio capire un concetto che riguarda i livelli di astrazione. Cosa c'è di sbagliato nel modello riportato nella Figura 10.6?

La risposta è: “i livelli di astrazione”. Una gerarchia di generalizzazione definisce un insieme di livelli di astrazione, dal più generale (alla sommità), al più specializzato (alla base). Si dovrebbe sempre mantenere un livello di astrazione uniforme all'interno di ciascun livello della gerarchia di generalizzazione. Non è così nell'esempio precedente: `JaguarXJS` è un tipo di automobile e si trova chiaramente a un livello di astrazione più basso rispetto ad `Autocarro`. Il modello può essere facilmente corretto introducendo una superclasse `Automobile`, tra `JaguarXJS` e `Veicolo`.

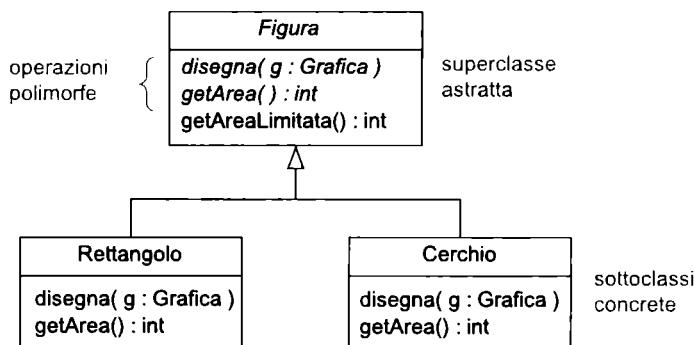
**Figura 10.6**

10.4 Polimorfismo

Polimorfismo significa “capacità di presentarsi sotto molte forme”. Un’operazione polimorfa ha molte implementazioni. Sono già state considerate due operazioni polimorfe nell’esempio della classe Figura. Le operazioni astratte `disegna()` e `getArea()` della classe Figura possono avere due implementazioni differenti: una nella classe Rettangolo, e un’altra nella classe Cerchio. Queste operazioni hanno “molte forme” e sono quindi polimorfe.

La Figura 10.7 illustra benissimo un caso di polimorfismo. In essa si vede una classe astratta Figura, che ha due operazioni astratte `disegna()` e `getArea()`. Le classi Rettangolo e Cerchio ereditano da Figura, fornendo un’implementazione delle operazioni polimorfe `Figura::disegna()` e `Figura::getArea()`. Tutte le sottoclassi concrete di Figura devono concretizzare le operazioni `disegna()` e `getArea()`, perché queste due sono astratte nella superclasse. Questo implica che, per quanto riguarda `disegna()` e `getArea()`, tutte le sottoclassi di Figura possono essere trattate allo stesso modo. Un insieme di operazioni astratte è pertanto un modo di definire un insieme di operazioni che tutte le sottoclassi devono implementare. In altre parole, è come definire un contratto che le sottoclassi concrete dovranno rispettare.

Naturalmente, Rettangolo e Cerchio forniscono un’implementazioni differente di `disegna()` e `getArea()`. L’operazione `disegna()` disegnerà un rettangolo per gli oggetti della classe Rettangolo, e disegnerà un cerchio per quelli della classe Cerchio. Ovviamente, anche l’operazione `getArea()` avrà implementazioni diverse. Restituirà larghezza*altezza per un rettangolo, e $\pi \cdot raggio^2$ per un cerchio. Questa è l’essenza del polimorfismo: oggetti di classi differenti hanno operazioni con la stessa segnatura, ma con diversa implementazione.

**Figura 10.7**

L'incapsulazione, l'ereditarietà e il polimorfismo sono i “tre pilastri” dell’OO. Il polimorfismo permette di progettare sistemi più semplici, che possano adattarsi meglio ai cambiamenti, perché consente di trattare oggetti differenti nello stesso modo.

Ciò che rende il polimorfismo un aspetto essenziale dell’OO è la possibilità di inviare lo stesso messaggio a oggetti di classi differenti, pur consentendo a tali oggetti di rispondere in modo appropriato. Se si invia il messaggio `disegna()` a un `Rettangolo`, viene disegnato un rettangolo, mentre inviando lo stesso messaggio a un `Cerchio`, viene disegnato un cerchio. Questi oggetti sembrano dunque dimostrare una forma di intelligenza.

10.4.1 Esempio di polimorfismo

Ecco ora un esempio pratico di come funziona il polimorfismo. Si supponga di avere una classe `Tela` che gestisce una collezione, o insieme, di oggetti `Figura`. Sebbene si tratti di una situazione semplificata, molti sistemi grafici lavorano effettivamente in modo molto simile. La Figura 10.8 illustra il modello di questo semplice sistema grafico.

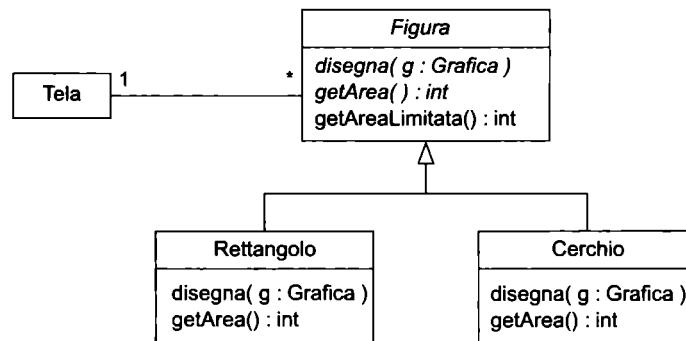


Figura 10.8

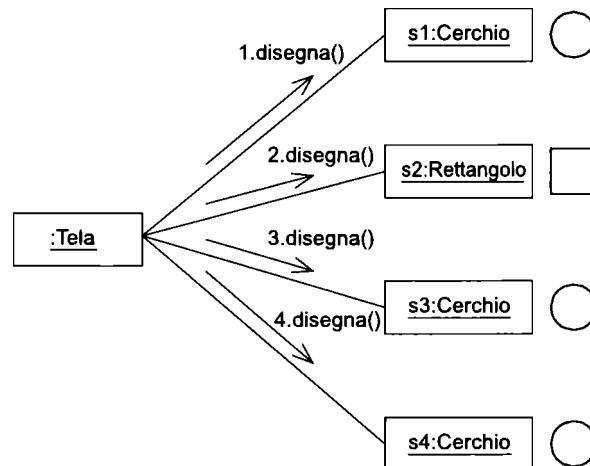


Figura 10.9

Si sa di non poter creare un'istanza di **Figura** (perché è una classe astratta), ma, secondo il principio di sostituibilità, si può creare istanze delle sue sottoclassi concrete e usarle ovunque si faccia riferimento a **Figura**.

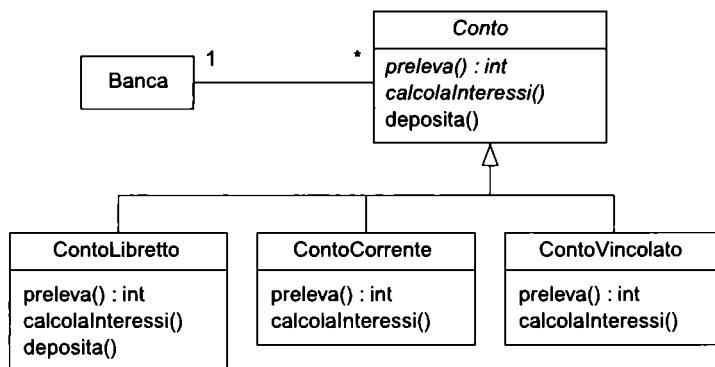
Così, anche se la Figura 10.8 mostra che un oggetto **Tela** contiene un insieme di molti oggetti **Figura**, gli unici oggetti che possono essere effettivamente inseriti in questo insieme sono le istanze delle sottoclassi concrete di **Figura**; questo perché **Figura** è una classe astratta, e non può essere istanziata. In questo esempio si dispone di due sottoclassi concrete **Cerchio** e **Rettangolo**, pertanto l'insieme può contenere oggetti di classe **Cerchio** e/o **Rettangolo**.

La Figura 10.9 illustra un modello a oggetti dal diagramma della Figura 10.8. Questo modello a oggetti mostra un oggetto **Tela** che contiene un insieme di quattro oggetti **Figura**: **s1**, **s2**, **s3** e **s4**. Dove **s1**, **s3** e **s4** sono oggetti di classe **Cerchio**, mentre **s2** è un oggetto di classe **Rettangolo**. Cosa accade quando l'oggetto **Tela** itera su questo insieme di oggetti mandando a ciascuno il messaggio **disegna()**? Non sorprendentemente, ogni oggetto fa la cosa giusta: gli oggetti **Rettangolo** disegnano rettangoli e gli oggetti **Cerchio** disegnano cerchi. La classe effettiva dell'oggetto stabilisce cosa viene disegnato dall'oggetto; ovvero, la classe effettiva dell'oggetto determina la semantica dell'insieme di operazioni offerte dall'oggetto.

Il punto chiave in questo caso è che ogni oggetto risponde al messaggio chiamando l'operazione corrispondente specificata dalla propria classe. Tutti gli oggetti della stessa classe risponderanno allo stesso messaggio chiamando la stessa operazione. Questo non significa necessariamente che tutti gli oggetti della stessa classe risponderanno allo stesso messaggio esattamente allo stesso modo. I risultati di una chiamata di un'operazione spesso dipendono dallo stato dell'oggetto: i valori dei suoi attributi e lo stato di tutte le sue relazioni. Per esempio, si possono avere tre oggetti di classe **Rettangolo** con diversi valori per gli attributi **larghezza** e **altezza**. Inviando a ciascuno di essi lo stesso messaggio **disegna()**, ciascuno disegnerà un rettangolo (in altre parole, la semantica delle operazioni effettivamente eseguite è sempre la stessa), ma ciascun rettangolo avrà dimensioni diverse, a seconda dei valori di larghezza e altezza di ciascun oggetto.

Si consideri ora un altro esempio. Le regole di gestione per i prelievi e il calcolo degli interessi sono diverse a seconda del tipo di conto bancario. Per esempio, i conti correnti di solito hanno dei limiti di scoperto e quindi possono avere saldo negativo. Invece, i depositi vincolati non consentono di operare in scoperto. In modo analogo, gli interessi vengono solitamente calcolati e accreditati sul conto con modalità differenti. La Figura 10.10 illustra un semplice modello di questo sistema. Vi è definita una classe astratta **Conto** e le classi concrete **ContoCorrente** e **ContoVincolato**. La classe astratta definisce le operazioni astratte per **prelievo()** e **calcolaInteressi()** che sono implementate diversamente dalle sottoclassi concrete.

Si osservi che viene anche definita l'operazione concreta **deposita()**, di cui però la classe concreta **ContoLibretto** ridefinisce l'implementazione. Si ricordi che per ridefinire un'operazione di una classe base, occorre semplicemente che la sottoclasse venga dotata di un'operazione con identica segnatura. Questa ridefinizione è utile, perché vi sono regole di *business* che rendono diverso il processo di deposito di denaro su un **ContoLibretto**, rispetto al deposito effettuato su altri tipi di **Conto**. Per esempio, potrebbero esserci regole che stabiliscono un ammontare minimo per richiesto per tale deposito.

**Figura 10.10**

Esistono due implementazioni di `deposita()`: una per `Conto` e un'altra per `ContoLibretto`. Quindi `deposita()` è, anch'essa, un'operazione polimorfa. Anche un'operazione concreta, come `deposita()` può essere polimorfa!

Non è, comunque, considerato buono stile ridefinire operazioni concrete. Questo perché invece di fornire un'implementazione mancante a una superclasse astratta, in questo caso si ignora un'implementazione esistente per sostituirne una nuova. Come si può essere sicuri che questo sia corretto, senza prima esaminare a fondo l'implementazione della superclasse? Contiene qualche peculiarità di cui non siamo a conoscenza? Le operazioni astratte possono sempre essere ridefinite con sicurezza: sono state concepite apposta. D'altro canto, ridefinire delle operazioni concrete può avere effetti collaterali imprevisti e potrebbe essere rischioso. Ciononostante la ridefinizione di operazioni concrete è una pratica ancora piuttosto diffusa. Bisogna dire che spesso l'operazione della sottoclasse fa qualcosa in più, ma poi richiama l'operazione della superclasse. In altre parole, aggiunge un proprio comportamento a quello già implementato nella superclasse. Questo particolare costrutto è effettivamente un modo legittimo di riutilizzare ed estendere il comportamento di una superclasse concreta, ed è generalmente sicuro.

Alcuni linguaggi consentono di impedire alle sottoclassi di ridefinire un metodo concreto della superclasse. In Java, posponendo la parola chiave `final` alla dichiarazione dell'operazione si può impedire esplicitamente la sua ridefinizione. In effetti, in Java è considerato buono stile definire `final` tutte le operazioni, eccettuate quelle che si vuole esplicitamente rendere polimorfe.

10.5 Riepilogo

In questo capitolo si è discusso di ereditarietà e di polimorfismo delle classi. Si sono imparati i seguenti concetti.

- l. La generalizzazione è una relazione tra un elemento più generico e uno più specificizzato:
 - l'elemento più specifico è consistente con quello più generico sotto tutti gli aspetti;

- il principio di sostituibilità afferma che l'elemento specializzato può essere utilizzato ovunque sia ammesso usare quello più generico;
 - la generalizzazione si applica a tutti i classificatori, e anche ad alcuni altri elementi di modellazione;
 - le gerarchie di generalizzazione possono essere create con un processo di generalizzazione di elementi specifici o di specializzazione di elementi generici;
 - tutti gli elementi appartenenti a uno stesso livello di una gerarchia di generalizzazione dovrebbero avere lo stesso livello di astrazione.
2. L'ereditarietà tra classi avviene quando esiste una relazione di generalizzazione tra le stesse classi.
- La sottoclasse eredita dai dalla classe genitore queste caratteristiche: attributi, operazioni, relazioni e vincoli.
 - Le sottoclassi possono:
 - aggiungere nuove caratteristiche;
 - ridefinire le operazioni che hanno ereditato:
 - la sottoclasse rende disponibile una nuova operazione con la stessa segnatura dell'operazione della superclasse che desidera ridefinire;
 - la segnatura dell'operazione è costituita da: nome dell'operazione, tipo restituito e elenco ordinato dei tipi di tutti i parametri.
 - Le operazioni astratte sono operazioni che non avranno alcuna implementazione:
 - servono come segnaposto;
 - tutte le sottoclassi concrete sono tenute a implementare tutte le operazioni astratte che hanno ereditato.
 - Una classe astratta contiene una o più operazioni astratte:
 - le classi astratte non possono essere istanziate;
 - le classi astratte definiscono un contratto, costituito dall'insieme delle loro operazioni astratte, che le sottoclassi concrete sono tenute a implementare.
 - Polimorfismo significa “capacità di presentarsi sotto molte forme”. Consente di progettare sistemi che utilizzano una classe astratta, a cui sostituiscono poi una classe concreta all'atto dell'esecuzione. Questi sistemi sono molto flessibili e facili da estendere, basta aggiungere nuove sottoclassi.
 - Le operazioni polimorfe hanno più di una implementazione:
 - classi differenti possono implementare la stessa operazione polimorfa in modi differenti;
 - il polimorfismo consente alle istanze di classi differenti di rispondere allo stesso messaggio in modi differenti.

Package di analisi

11.1 Contenuto del capitolo

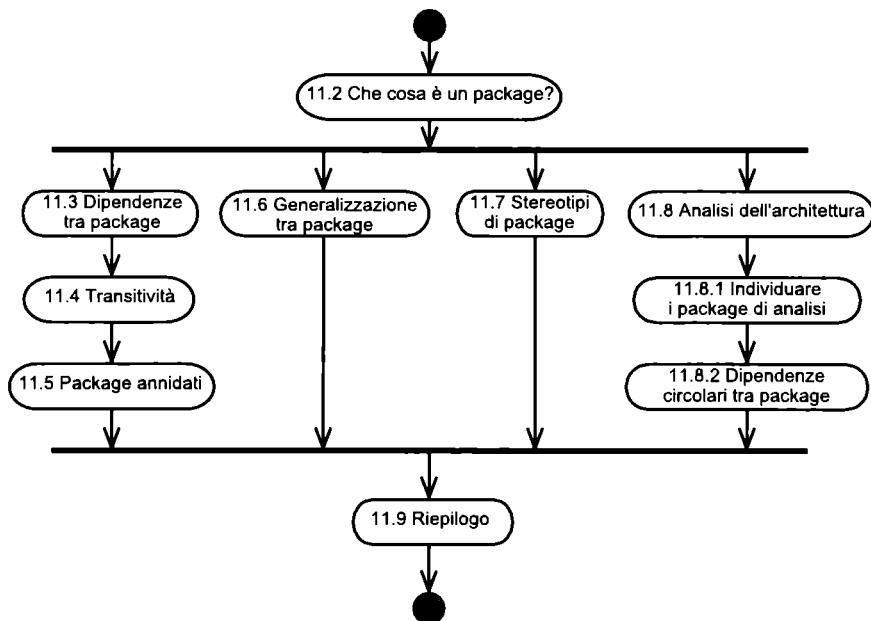


Figura 11.1

11.2 Che cosa è un package?

Tornando ai principi base dell'UML (Sezione 1.7), si sa che i costituenti fondamentali dell'UML sono entità, relazioni e diagrammi. Un *package* è l'entità di raggruppamento dell'UML: è il proprietario e il contenitore degli elementi del modello.

Ogni *package* possiede un suo spazio dei nomi, al cui interno tutti i nomi degli elementi devono essere univoci.

In pratica, un *package* è un meccanismo generalizzato per organizzare e raggruppare elementi e diagrammi. Lo si può usare per:

- raggruppare gli elementi semanticamente correlati;
- creare un “confine semantico” interno al modello;
- definire delle unità di gestione della configurazione;
- parallelizzare la progettazione su più unità;
- fornire uno spazio dei nomi encapsulato, al cui interno tutti i nomi devono essere univoci.

I *package* consentono la creazione di un modello navigabile e ben strutturato mediante il raggruppamento di elementi che hanno una forte correlazione semantica. Possono anche essere utili per creare dei confini semanticici interni al modello, con diversi *package* che supportano insiemi diversi delle funzionalità del sistema.

Ogni elemento del modello appartiene a un unico *package* e la gerarchia di appartenenza forma un albero la cui radice è il *package* di livello più alto. L’UML mette a disposizione l’apposito stereotipo «*topLevel*», che può essere utilizzato per identificare il *package* di livello più alto. Gli strumenti di CASE per l’UML solitamente presumono che ogni elemento del modello che non è assegnato esplicitamente a un *package*, appartenga al *package* «*topLevel*».

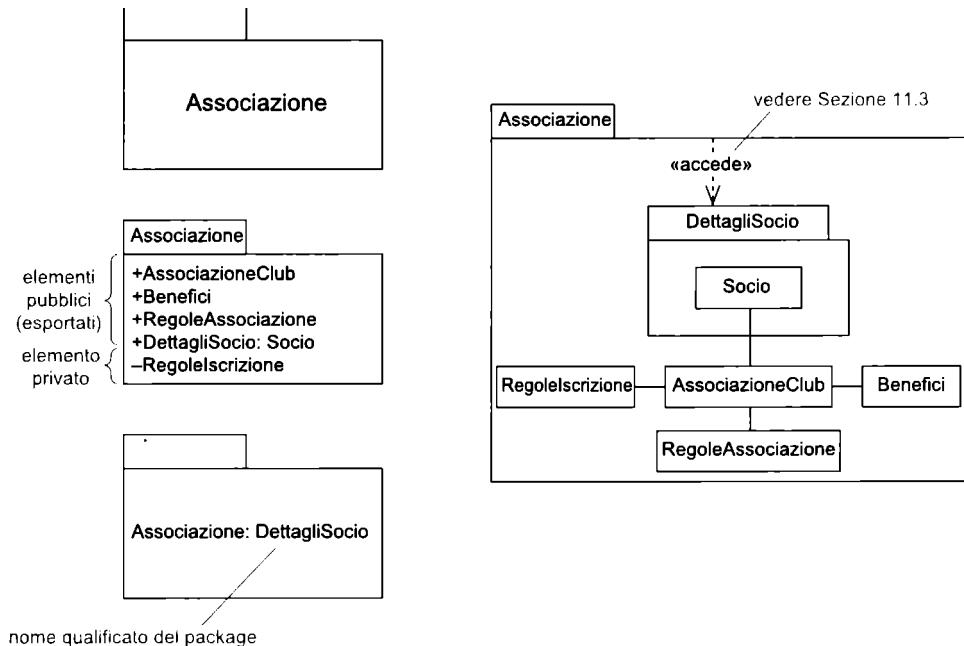


Figura 11.2

I *package* di analisi possono contenere:

- casi d'uso;
- classi di analisi;
- realizzazioni di caso d'uso.

La sintassi UML per i *package* è abbastanza semplice, ed è riportata nella Figura 11.2. L'icona del *package* è una cartella. Il nome del *package* può essere indicato nell'etichetta o nel corpo della cartella.

Gli elementi contenuti in un *package* possono avere una visibilità che indica se sono visibili o meno ai clienti del *package*. La Tabella 11.1 riporta le visibilità previste dall'UML per gli elementi di un *package*.

Tabella 11.1

Simbolo	Visibilità	Significato
+	pubblica	Gli elementi con visibilità pubblica sono visibili agli altri <i>package</i>
-	privata	Gli elementi con visibilità privata sono del tutto nascosti all'interno del <i>package</i> in cui risiedono.
#	protetta	Gli elementi con visibilità protetta sono visibili solo ai <i>package</i> discendenti del <i>package</i> in cui risiedono (vedere Sezione 11.5).

Si può sfruttare la visibilità degli elementi per limitare il grado d'interdipendenza tra *package*. Questo è possibile perché gli elementi esportati di un *package* agiscono come un'interfaccia, o una porta di accesso, sul resto del *package* stesso. Questa interfaccia deve essere sempre la più ridotta possibile.

Per dare a un *package* un'interfaccia piccola e semplice, occorre minimizzare il numero di elementi pubblici o protetti presenti all'interno del *package*, massimizzando contestualmente il numero di elementi privati. Questo potrebbe essere difficile da ottenere durante le attività di analisi, a meno che non si espliciti la navigabilità delle associazioni. In caso contrario, le associazioni tra classi sono tutte bidirezionali, il che significa che le classi coinvolte devono appartenere allo stesso *package*, oppure devono essere entrambe classi con visibilità pubblica. Nella progettazione le relazioni tra classi vengono spesso rifinite in relazioni unidirezionali e in questo caso non è più richiesto che la classe cliente sia pubblica.

11.3 Dipendenze tra *package*

Tra i diversi *package* di un sistema possono esistere delle relazioni di dipendenza. Per esempio, riprendendo in esame la Figura 11.2, ogni *package* che avesse una relazione

di dipendenza con il *package* *Associazione* sarà in grado di vedere gli elementi pubblici del *package* (*AssociazioneClub*, *Benefici*, ecc.), ma non l'elemento privato *RegoleIscrizione*. Si dice che il *package* esporta i propri elementi pubblici.

Esistono quattro diversi tipi di dipendenze tra *package*; ognuna ha un significato diverso e sono tutte riassunte nella Tabella 11.2.

Tabella 11.2

Dipendenza del package	Significato
<pre> graph LR F[Fornitore] -- "usa" --> C[Cliente] </pre>	<p>Un elemento è <i>package</i> cliente usa, in qualche modo, un elemento pubblico del <i>package</i> fornitore. Il cliente dipende dal fornitore. Quando una dipendenza tra <i>package</i> non ha struttura, si dice solitamente che si tratti di una dipendenza «usa».</p>
<pre> graph LR F[Fornitore] -- "importa" --> C[Cliente] </pre>	<p>Lo spazio dei nomi è <i>package</i> fornitore viene incorporato in quello del <i>package</i> cliente: gli elementi del <i>package</i> cliente possono accedere a tutti gli elementi pubblici del <i>package</i> fornitore.</p>
<pre> graph LR F[Fornitore] -- "accede" --> C[Cliente] </pre>	<p>Gli elementi del <i>package</i> cliente possono accedere a tutti gli elementi pubblici del <i>package</i> fornitore, ma lo spazio dei nomi non viene incorporato. Il cliente deve utilizzare i <i>pathnames</i>.</p>
<pre> graph LR MA[Modello dell'analisi] -- "origine" --> MP[Modello della progettazione] </pre>	<p>Solitamente «origine» rappresenta l'evoluzione di un elemento in un altro elemento più sviluppato. Di solito è una relazione tra modelli, e non tra elementi di uno stesso modello (relazione extra-modello).</p>

Un *package* definisce ciò che viene chiamato spazio dei nomi encapsulato. In pratica, questo vuol dire che il *package* crea un confine all'interno del quale tutti i nomi degli elementi devono essere univoci. Significa anche che un elemento di uno spazio dei nomi, per poter referenziare un elemento di un altro spazio dei nomi, deve indicare, oltre al nome del destinatario, anche un percorso per arrivarci, attraverso la gerarchia degli spazi dei nomi annidati. Tale indicazione è detta *pathname* di un elemento. I *pathnames* vengono trattati in dettagli nel Paragrafo 11.5, dedicata ai *package* annidati.

Il concetto di spazio dei nomi genera un'importante differenza di significato tra le dipendenze «accede» e «importa». Entrambe consentono agli elementi del *package* cliente di accedere agli elementi del *package* fornitore, ma hanno significato differente per quanto riguarda gli spazi dei nomi dei *package*.

Nel caso della dipendenza «importa», lo spazio dei nomi del *package* fornitore viene incorporato in quello del cliente. Questo significa che quando gli elementi del cliente fanno riferimento a elementi nel *package* fornitore, possono usare direttamente il nome dell'elemento e non hanno bisogno di qualificarlo con il nome del *package*. Questa è la dipendenza dei *package* di tipo più diffuso, anche se può produrre delle collisioni tra i

nomi degli elementi, quando un elemento del *package* fornitore ha lo stesso nome di un elemento del *package* cliente.

D'altro canto, la dipendenza «*accede*» indica che gli elementi del *package* cliente possono accedere agli elementi pubblici del *package* fornitore, ma lo spazio dei nomi del *package* fornitore non viene incorporato. Gli elementi nel *package* cliente devono quindi sempre usare i *pathname* degli elementi del *package* fornitore a ogni singolo accesso. Per esempio, se nel *package* fornitore esiste una classe pubblica A, allora gli elementi del *package* cliente possono accedere a questa classe, ma devono usare un *pathname* che includa il nome del *package*: NomePackageFornitore::A

La dipendenza «*usa*» è simile per vari aspetti alla dipendenza «*accede*». La differenza principale è che «*usa*» significa che ci sono dipendenze tra gli elementi dei due *package*, mentre «*accede*» non afferma nulla sugli elementi dei *package*: afferma solo che vi è una dipendenza generica tra i *package* in sé stessi. La differenza è sottile e non crea differenze sostanziali dal punto di vista della modellazione. La maggior parte dei modellatori preferisce utilizzare una dipendenza «*usa*» o una dipendenza non rifinita, priva di stereotipo.

«origine» è la mosca bianca. Laddove le altre dipendenze tra *package* collegano elementi dello stesso modello, «origine» solitamente rappresenta l'evoluzione, anche temporale, di un *package* in un altro. Di conseguenza, rappresenta spesso una relazione tra modelli diversi. Si può rappresentare un modello UML completo con un *package* con stereotipo «modello», come illustrato nella Tabella 11.2, dove abbiamo tracciato una dipendenza extra-modello «origine» tra il modello dell'analisi e il modello della progettazione. Ovviamente, quel diagramma è un meta-modello usato per descrivere le relazioni tra modelli e, in quanto tale, non viene usato molto spesso nella modellazione dei sistemi.

11.4 Transitività

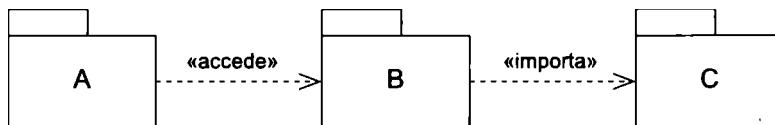
La transitività è un concetto che si applica alle relazioni. Significa che se esiste una prima relazione tra l'elemento A e l'elemento B, e una seconda relazione tra l'elemento B e l'elemento C, allora esiste anche una terza relazione implicita tra l'elemento A e l'elemento C.

È importante osservare che le dipendenze tra *package* «*accede*» e «*importa*» non sono transitive. Nell'esempio della Figura 11.3, il *package* A accede al *package* B, mentre il *package* B importa il *package* C.

L'assenza di transitività per «*accede*» e «*importa*» significa che:

- gli elementi del *package* A possono vedere gli elementi del *package* B;
- gli elementi del *package* B possono vedere gli elementi del *package* C;
- gli elementi del *package* A non possono vedere gli elementi del *package* C.

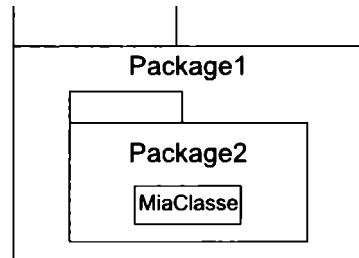
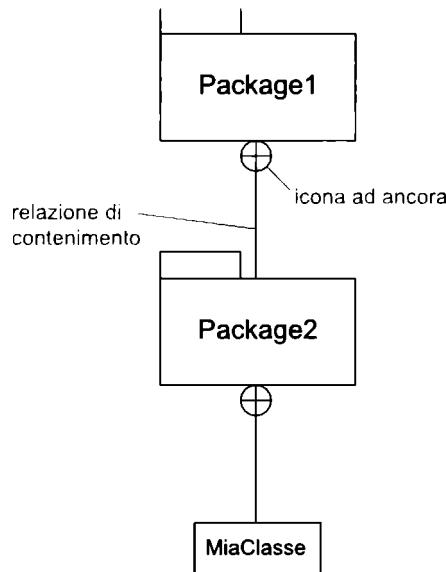
L'assenza di transitività per «*accede*» e «*importa*» consente di gestire e controllare l'interdipendenza e la coesione nel modello. Non si accede e non si importa nulla se non dichiarandolo esplicitamente; questo è esattamente ciò che si voleva ottenere.

**Figura 11.3**

11.5 Package annidati

È possibile annidare i *package* all'interno di altri *package*. È anche possibile raggiungere un qualsivoglia livello di annidamento, ma di solito sono sufficienti un livello o due. Usandone qualcuno in più, si rischia di rendere il modello incomprensibile o poco navigabile.

L'UML consente di rappresentare l'annidamento con due distinte notazioni. La prima è piuttosto grafica; il *package* annidato viene disegnato dentro al *package* ospite, come illustrato nella Figura 11.4.

**Figura 11.4****Figura 11.5**

La Figura 11.5 illustra la notazione alternativa, che è utile quando si abbiano diversi livelli di annidamento, o annidamenti complessi, che potrebbero risultare poco comprensibili qualora visualizzati mediante l'inclusione grafica.

Si può referenziare *package* o classi annidate usando i *pathname*. Questi indicano il nome della classe o del *package*, preceduto dal percorso di annidamento. Sono del tutto simili ai percorsi utilizzati per identificare cartelle e file in un file system gerarchico, quale quelli utilizzati dai più comuni sistemi operativi. Facendo riferimento alle Figure 11.4 e 11.5, il *pathname* di *MiaClasse* è *Package1::Package2::MiaClasse*.

Si guardi l'esempio della Figura 11.6: una classe A, di *Package1*, è associata a una classe B, di *Package2*.

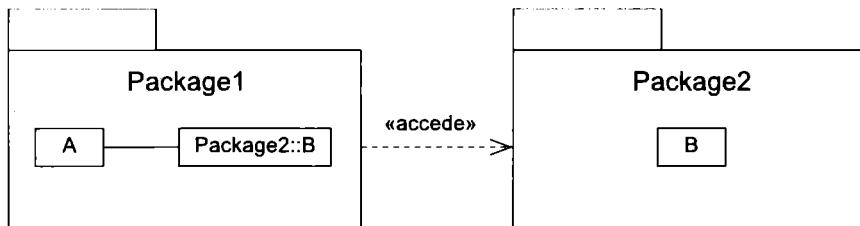


Figura 11.6

Nella Figura 11.6, *Package1* è collegato a *Package2* con una dipendenza «accede». Questa consente agli elementi in *Package1* di accedere agli elementi pubblici di *Package2*, ma non incorpora lo spazio dei nomi di *Package2*. Pertanto, dovendo referenziare la classe B, da un qualunque punto di *Package1*, è necessario utilizzare il *pathname* di B.

Se invece esistesse una dipendenza «importa» tra *Package1* e *Package2*, allora non sarebbe necessario utilizzare i *pathname*, perché lo spazio dei nomi di *Package2* sarebbe stato incorporato direttamente nello spazio dei nomi di *Package1*.

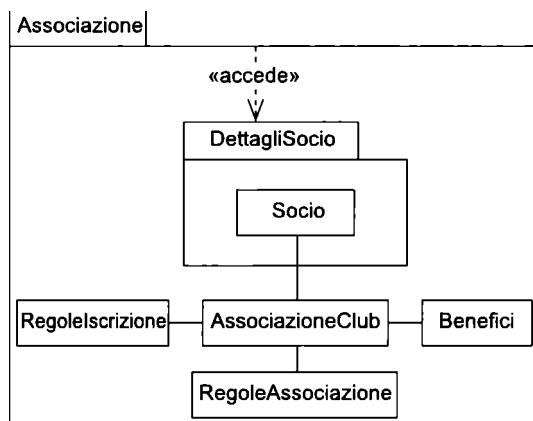


Figura 11.7

Un *package* annidato vede tutti gli elementi pubblici del *package* ospite. Un *package* ospite non vede nessuno degli elementi dei suoi *package* annidati, a meno che non abbia una dipendenza «*accede*» o «*importa*» con essi. Ciò è in pratica molto utile, perché consente di nascondere i dettagli implementativi nei *package* annidati.

Nell'esempio della Figura 11.7, nel *package* DettagliSocio potrebbero esserci molte classi, quali Nome, Indirizzo, IndirizzoEmail, ma possono comunque restare tutte inaccessibili per il *package* ospite, purché siano stati definiti come elementi privati del *package* annidato. Il *package* ospite Associazione ha una dipendenza «*accede*» verso il suo *package* annidato DettagliSocio, e gli è quindi concesso di accedere ai soli elementi pubblici di quest'ultimo. In questo caso, il *package* DettagliSocio ha una sola classe pubblica di nome Socio; gli altri suoi elementi privati rimangono nascosti e inaccessibili.

11.6 Generalizzazione tra package

La generalizzazione tra *package* è, sotto molti aspetti, simile a quella tra classi. Nella generalizzazione tra *package*, i *package* figli sono più specializzati ed ereditano gli elementi pubblici e protetti del loro *package* genitore. I *package* figli possono aggiungere nuovi elementi e possono ridefinire elementi del genitore fornendone un'implementazione alternativa con lo stesso nome.

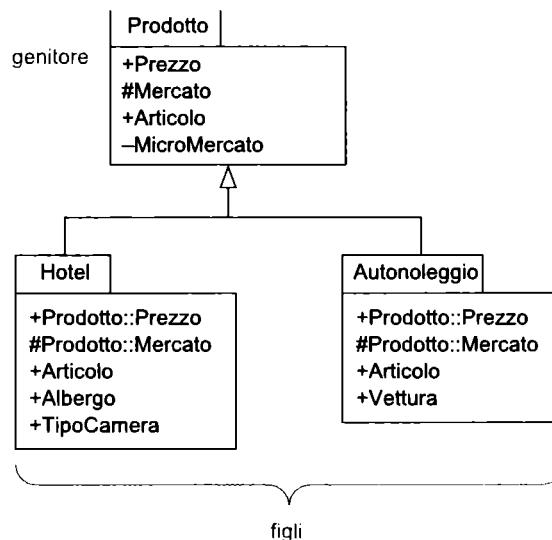


Figura 11.8

Nell'esempio della Figura 11.8, i *package* Hotel e Autonoleggio ereditano tutti i membri protetti e pubblici del loro *package* genitore Prodotto. Sia il *package* Hotel che Autonoleggio ridefiniscono la classe Articolo ereditata dal loro genitore fornendo una classe alternativa con lo stesso nome. I *package* figli possono anche aggiungere dei nuovi membri: il *package* Hotel aggiunge le classi Albergo e TipoCamera, mentre il *package* Autonoleggio aggiunge la classe Vettura.

Il principio di sostituibilità vale anche per i *package*, proprio come per l'ereditarietà tra classi: ovunque si può usare il *package* Prodotto, si deve poter anche usare il *package* Hotel o il *package* Autonoleggio.

11.7 Stereotipi di *package*

L'UML fornisce alcuni stereotipi standard per adattare la semantica dei *package* a situazioni particolari. Li elenchiamo nella Tabella 11.3.

Gli stereotipi di *package* usati più comunemente sono «sistema», «sottosistema» e «façade». Degli altri due, «framework» non è molto utilizzato, mentre «stub» trova qualche uso nella modellazione dei sistemi distribuiti dove, sul lato client, si ha uno stub, proxy o rimando a un analogo *package* che risiede sul lato server.

Tabella 11.3

Stereotipo	Significato
«sistema»	Un <i>package</i> che rappresenta l'intero sistema modellato
«sottosistema»	Un <i>package</i> che è una parte indipendente del sistema modellato
«façade»	Un <i>package</i> che è solo una vista di un altro <i>package</i>
«framework»	Un <i>package</i> che contiene dei <i>pattern</i>
«stub»	Un <i>package</i> che serve da proxy (rimando) per i contenuti pubblici di un altro <i>package</i>

11.8 Analisi dell'architettura

Nell'analisi dell'architettura, tutte le classi di analisi sono organizzate in un insieme coeso di *package* di analisi; questi sono ulteriormente organizzati in partizioni e strati, come si vede nella Figura 11.9. Ogni *package* di analisi in uno strato costituisce una partizione.

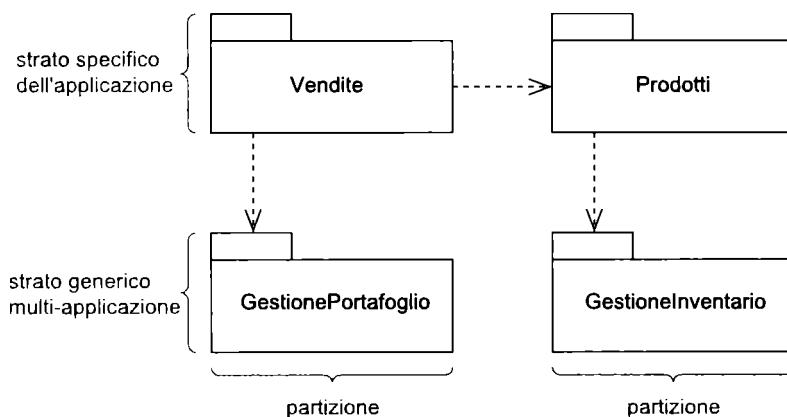


Figura 11.9

Uno degli obiettivi dell’analisi dell’architettura è tentare di minimizzare le interdipendenze presenti nel sistema. Esistono tre tecniche per ottenere questo risultato:

- minimizzare le dipendenze tra *package* di analisi;
- minimizzare il numero di elementi pubblici e protetti presenti in ciascun *package* di analisi;
- massimizzare il numero di elementi privati presenti in ciascun *package* di analisi.

Ridurre le interdipendenze è una delle preoccupazioni principali dell’analisi dell’architettura, perché i sistemi che mostrano un’interdipendenza notevole sono solitamente anche quelli più complessi e difficili da costruire e mantenere. Si deve sempre fare il possibile per ridurre l’interdipendenza a quella minima indispensabile.

Il numero di strati individuati nel modello tende a aumentare man mano che questo viene rifinito e arricchito di dettagli, avvicinandosi sempre più a quello che sarà il modello della progettazione. Tuttavia, durante l’analisi ci si può limitare a sistemare i *package* in due soli strati: quello specifico dell’applicazione e quello generico multi-applicazione. Lo strato specifico contiene funzionalità che sono peculiari dell’applicazione modellata, mentre lo strato generico contiene funzionalità di utilità generale. Nella Figura 11.9 GestioneAccount e GestioneInventario potrebbero essere riutilizzabili in diverse altre applicazioni, quindi questi *package* appartengono allo strato generico multi-applicazione.

11.8.1 Individuare i *package* di analisi

I *package* di analisi possono essere individuati identificando i gruppi di elementi del modello accomunati da forti correlazioni semantiche. I *package* vengono spesso individuati gradualmente, man mano che il modello matura e si evolve. È fondamentale che i *package* di analisi rispecchino dei raggruppamenti semanticci reali, piuttosto che una visione ideale (ma fittizia) dell’architettura logica.

Dove iniziare a cercare questi raggruppamenti? La fonte di potenziali *package* più utile è il modello statico. Occorre cercare:

- aggregati coesi di classi nei diagrammi delle classi;
- gerarchie di ereditarietà.

Anche il modello dei casi d’uso può essere preso in considerazione come fonte di potenziali *package*, perché è importante cercare di costruire *package* coesi, soprattutto, dal punto di vista del business. Tuttavia, capita spesso che i casi d’uso siano *trasversale* a diversi *package* di analisi: un unico caso d’uso può essere realizzato con classi appartenenti a diversi *package* distinti. Comunque, uno o più casi d’uso che supportino un dato processo aziendale o attore, o anche un insieme di casi d’uso collegati, potrebbe indicare un potenziale *package*.

Dopo l’individuazione di un insieme di potenziali *package*, bisogna tentare di minimizzare i membri pubblici e protetti dei *package* e le dipendenze tra i *package*.

Si può procedere come segue:

- spostando classi da un *package* a un altro;
- aggiungendo *package*;
- eliminando *package*.

I punti di forza di una buona struttura di *package* sono la massima coesione *interna* dei *package*, e la minima interdipendenza *tra package*. Un *package* dovrebbe contenere un gruppo di classi strettamente collegate. Le classi sono correlate nel modo più forte dall'ereditarietà (Capitolo 10), quindi dalla composizione (Capitolo 16), dall'aggregazione (Capitolo 16) e, infine, dalle dipendenze (Capitolo 9). Le classi appartenenti a gerarchie di ereditarietà o di composizione sono le migliori candidate per essere raggruppate in uno stesso *package*. Questo risulterà in una forte coesione interna al *package* e, probabilmente, produrrà anche una minore interdipendenza con gli altri *package*.

Come sempre, quando si crea il modello dei *package* di analisi si deve ricercare la semplicità. Ottenerne il giusto insieme di *package* è più importante che non fare ampio uso di costrutti come la generalizzazione tra *package* e le dipendenze con stereotipo. Li si può sempre aggiungere in un secondo tempo se, e solo se, rendono il modello più comprensibile. Un modo per mantenere il modello semplice consiste nell'evitare l'uso eccessivo di *package* annidati. Più un elemento si ritrova annidato nella gerarchia dei *package*, meno risulta accessibile. Abbiamo visto modelli con molti livelli di annidamento, in cui ogni *package* conteneva solo una o due classi. Questi modelli assomigliavano più a una tipica decomposizione funzionale *top-down*, che a non un modello a oggetti.

Come regola pratica, si può provare ad assegnare tra le cinque e le dieci classi di analisi per ogni *package*. Tuttavia, come per tutte le regole pratiche, ci sono eccezioni e non si deve esitare a violarla se, così facendo, il modello risulta più comprensibile. A volte, per spezzare una dipendenza circolare nel modello dei *package*, occorre introdurre un *package* contenente appena una o due classi. In questo caso, la violazione della regola pratica è del tutto giustificata.

11.8.2 Dipendenze circolari tra *package*

Si deve tentare di evitare le dipendenze circolari nel modello dei *package* di analisi. Pensandoci bene, se il *package* A dipende in qualche modo dal *package* B e viceversa, forse è il caso di fonderli in un unico *package*; è una tecnica semplice, ma perfettamente valida, per rimuovere una dipendenza circolare. Un approccio migliore, che funziona molto spesso, è quello di tentare estrarre gli elementi comuni e raggrupparli in un terzo *package* C. Si ricalcolano quindi le dipendenze per rimuovere il riferimento circolare. La Figura 11.10 illustra entrambe queste tecniche.

Molti strumenti di CASE verificano automaticamente le dipendenze tra *package*. Lo strumento crea una lista delle violazioni d'accesso: elementi in un *package* che accedono a elementi in altri *package*, senza che sia stata dichiarata una visibilità o una dipendenza appropriata tra i due *package*.

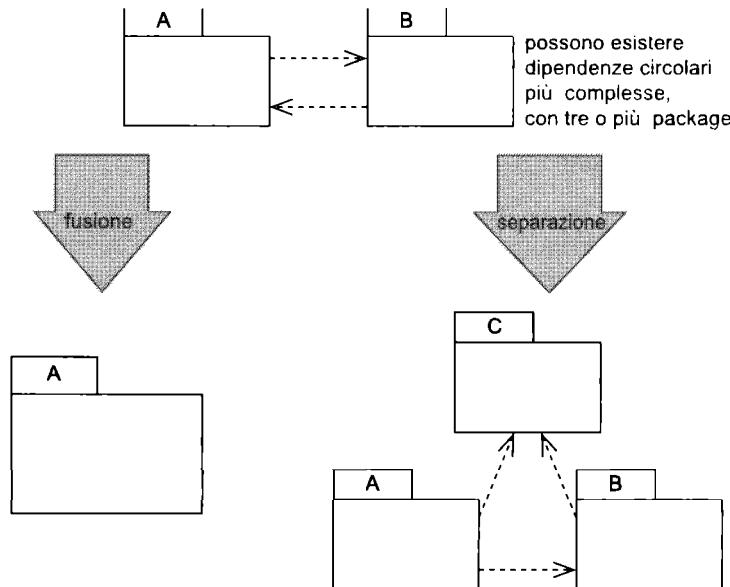


Figura 11.10

In un modello dell’analisi potrebbe risultare impossibile creare un diagramma dei *package* che sia privo di violazioni d’accesso; questo perché nell’analisi si usano relazioni tra classi non rifinite (ovvero bidirezionali).

Supponendo di avere un semplicissimo modello con una classe nel *package* A un’altra classe nel *package* B. Se la classe del *package* A possiede una relazione bidirezionale con quella del *package* B, allora il *package* A dipende dal *package* B, ma è anche vero il contrario, quindi è presente una dipendenza circolare tra i due *package*.

Gli unici modi per eliminare questa violazione, sono quelli di rifinire la relazione tra le due classi, rendendola unidirezionale, oppure di raggruppare le due classi in un unico *package*. Il proliferare di dipendenze tra *package* è un buon argomento a favore dell’uso della navigabilità già nei modelli dell’analisi!

D’altra parte, le classi che hanno delle dipendenze bidirezionali vere (ovvero, non dovute semplicemente all’incompletezza del modello) dovrebbero solitamente risiedere nello stesso *package*.

11.9 Riepilogo

In questo capitolo si sono affrontati i *package* di analisi. In particolare, si è visto come massimizzare la coesione interna di un *package* di analisi, e come minimizzarne l’interdipendenza con gli altri *package* di analisi. Queste tecniche aiutano a creare sistemi più robusti e mantenibili.

Si sono spiegati i seguenti concetti.

1. Il *package* è il meccanismo con cui l'UML consente di raggruppare elementi.
2. I *package* hanno diversi usi:
 - raggruppano gli elementi con semantica affine;
 - creano un “confine semantico” interno al modello;
 - costituiscono unità di gestione della configurazione;
 - consentono di parallelizzare la progettazione su più unità;
 - forniscono uno spazio dei nomi encapsulato, in cui tutti i nomi devono essere univoci; per accedere a un elemento si deve specificarne sia il nome che il percorso del *package*.
3. Ogni elemento del modello appartiene a un unico *package*:
 - i *package* sono organizzati in una gerarchia;
 - il *package* radice può avere lo stereotipo «topLevel»;
 - in assenza di una collocazione esplicita, gli elementi del modello appartengono al *package* «topLevel».
4. I *package* di analisi possono contenere:
 - casi d’uso;
 - classi di analisi;
 - realizzazioni di caso d’uso.
5. Gli elementi dei *package* possono avere una visibilità:
 - si usa la visibilità per limitare l’interdipendenza tra *package*;
 - esistono tre livelli di visibilità:
 - pubblica (+): gli elementi sono visibili agli altri *package*;
 - privata (-): gli elementi sono completamente nascosti;
 - protetta (#): gli elementi sono visibili solo ai *package* discendenti.
6. Una relazione di dipendenza tra *package* indica che il *package* cliente dipende in qualche modo dal *package* fornitore:
 - «usa»: un elemento del *package* cliente usa qualche elemento pubblico del *package* fornitore;
 - «importa»: gli elementi del *package* cliente possono accedere a tutti gli elementi pubblici del *package* fornitore e lo spazio dei nomi del *package* fornitore viene incorporato in quello del *package* cliente;
 - «accede»: gli elementi del *package* cliente possono accedere a tutti gli elementi pubblici del *package* fornitore, ma gli spazi dei nomi dei *package* rimangono separati;

- «origine»: il *package* cliente è l’evoluzione, anche temporale, del *package* fornitore; solitamente viene utilizzata come relazione tra modelli, e non tra elementi dello stesso modello.
7. **Transitività:** se A ha una relazione con B, e B ha una relazione con C, allora A ha una relazione con C.
- Le dipendenze «*accede*» e «*importa*» non sono transitive.
8. **Package annidati:**
- Il *package* annidato può accedere a tutti gli elementi dei suoi *package* esterni (ospiti);
 - il *package* ospite non può accedere a nessuno degli elementi dei suoi *package* annidati, a meno che non abbia una dipendenza esplicita con essi (di solito, «*accede*» o «*importa*»); questo consente di occultare i dettagli dell’implementazione all’interno dei *package* annidati.
9. **Generalizzazione tra *package*:**
- molto simile alla generalizzazione tra classi;
 - i *package* figli:
 - ereditano gli elementi del *package* genitore;
 - possono aggiungere nuovi elementi;
 - possono ridefinire gli elementi ereditati dal *package* genitore.
10. **Stereotipi di *package*:**
- «*sistema*»: un *package* che rappresenta l’intero sistema modellato;
 - «*sottosistema*»: un *package* che è una parte indipendente del sistema modellato;
 - «*façade*»: un *package* che è solo una vista di un altro *package*;
 - «*framework*»: un *package* che contiene dei *pattern*;
 - «*stub*»: un *package* che serve da rimando per i contenuti pubblici di un altro *package*.
11. **Analisi dell’architettura:**
- raggruppare gli insiemi coesi di classi di analisi in *package* di analisi;
 - organizzare in livelli i *package* di analisi secondo la loro semantica;
 - minimizzare l’interdipendenza tra *package*:
 - minimizzare le dipendenze tra *package*;
 - minimizzare il numero di elementi pubblici e protetti in tutti i *package*;
 - massimizzare il numero di elementi privati in tutti i *package*.

12. Individuare i *package* di analisi.

- Esaminare le classi di analisi cercando:
 - gruppi coesi di classi strettamente collegate;
 - gerarchie di ereditarietà;
 - le classi sono più strettamente correlate tramite (in ordine) ereditarietà, composizione, aggregazione e dipendenza.
- Esaminare i casi d'uso:
 - gruppi di casi d'uso che supportano un particolare processo gestionale o attore *potrebbero* aiutare a isolare classi di analisi che possono essere raggruppate;
 - casi d'uso affini *potrebbero* avere delle classi di analisi da raggruppare assieme;
 - cautela: spesso un *package* di analisi è trasversale a più casi d'uso!
- Rifinire il modello dei *package*, cercando di massimizzare la coesione tra *package* e di minimizzarne le interdipendenze:
 - spostare classi da un *package* a un altro;
 - aggiungere *package*;
 - eliminare *package*;
 - rimuovere le dipendenze circolari fondendo i *package* o suddividendoli per estrarne le classi interdipendenti.

Realizzazione di caso d'uso

12.1 Contenuto del capitolo

Il capitolo si suddivide in tre parti: concetti generali relativi all'attività UP “Analizzare un caso d'uso”, alle collaborazioni e alle interazioni (Sezioni dalla 12.2 alla 12.6), una descrizione dettagliata dei diagrammi di collaborazione (Sezione 12.7) e una descrizione dettagliata dei diagrammi di sequenza (Sezione 12.8).

I diagrammi di collaborazione e di sequenza sono due viste diverse di una stessa interazione tra oggetti. Per evitare il rischio di ripetere inutilmente gli stessi concetti, si è deciso di presentare la semantica dei diagrammi di interazione in modo dettagliato nel paragrafo relativo ai diagrammi di collaborazione, presentandola invece in modo più conciso nel paragrafo relativo ai diagrammi di sequenza. Chi scegliesse di leggere prima la sezione sui diagrammi di sequenza, voglia tener presente che una discussione più approfondita è disponibile nel paragrafo sui diagrammi di collaborazione.

12.2 Attività UP: analizzare un caso d'uso

Nei precedenti capitoli è stato considerato come l'attività UP “Analizzare un caso d'uso” produca l'artefatto classe di analisi. La stessa attività produce anche un secondo artefatto, la realizzazione di caso d'uso, come illustrato nella Figura 12.2.

Le classi di analisi modellano la struttura statica del sistema. Invece, le realizzazioni di caso d'uso mostrano come le istanze delle classi di analisi interagiscono tra loro per realizzare la funzionalità del sistema, e fanno quindi parte della vista dinamica del sistema.

12.3 Cosa sono le realizzazioni di caso d'uso?

Un aspetto cruciale dell'analisi, dopo l'individuazione delle classi, consiste nell'individuare le realizzazioni di caso d'uso. Queste ultime sono un insieme di classi che

realizzano il comportamento specificato da un caso d'uso. Per esempio, se si ha un caso d'uso *PrestitoLibro*, e sono state individuate le classi *Libro*, *Scheda*, *Debitore* e l'attore *Bibliotecario*, si deve allora creare una realizzazione di caso d'uso che dimostra come queste classi, e gli oggetti di queste classi, possono interagire per realizzare il comportamento descritto in *PrestitoLibro*. In questo modo, si trasforma un caso d'uso (che è una specifica di requisiti funzionali) in un insieme di diagrammi delle classi e di diagrammi di interazione (che sono una specifica ad alto livello del sistema).



Figura 12.1

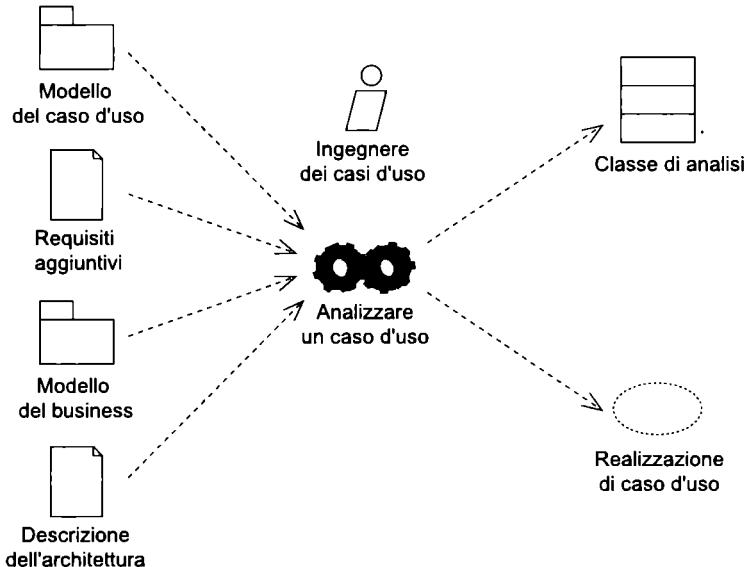


Figura 12.2 Adattata da figura 8.25 [Jacobson 1], con il consenso della Addison-Wesley.

Sebbene l'UML preveda un apposito simbolo (vedere la Figura 12.3) per rappresentare le realizzazioni di caso d'uso, questi ultimi sono modellati in modo esplicito molto raramente. Questo perché a ciascun caso d'uso corrisponde un'unica realizzazione e, quindi, un diagramma di realizzazione di caso d'uso non fisserebbe, in realtà, alcuna informazione aggiuntiva. Di solito, invece, si aggiungono gli elementi appropriati (vedere la Tabella 12.1) in uno strumento CASE e si lascia che le realizzazioni di caso d'uso siano un elemento隐式的 nell'archivio del modello.

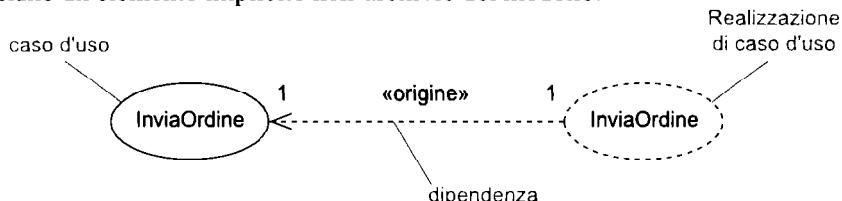


Figura 12.3

12.4 Realizzazioni di caso d'uso: elementi

Le realizzazioni di caso d'uso sono costituite dagli elementi elencati nella Tabella 12.1.

In pratica, la realizzazione di caso d'uso è un processo di rifinitura. Si parte da una specifica di un aspetto del comportamento del sistema, costituito da un caso d'uso e dai suoi eventuali requisiti aggiuntivi, e si crea un modello di come lo si potrebbe realizzare mediante collaborazioni e interazioni tra istanze delle classi di analisi già individuate.

Tabella 12.1

Elemento	Scopo
Diagrammi delle classi dell'analisi	Mostrare le classi di analisi che interagiscono per realizzare il caso d'uso
Diagrammi di interazione	Mostrare le collaborazioni e le interazioni tra istanze specifiche che realizzano il caso d'uso: sono delle "istantanee" del sistema in esecuzione.
Requisiti speciali	Formulare i requisiti specifici del caso d'uso eventualmente emersi durante il processo di realizzazione di caso d'uso.
Rifinitura del caso d'uso	Aggiornare il caso d'uso originario, qualora si ottengano nuove informazioni durante la sua realizzazione.

Partendo, dunque, da una specifica generica del comportamento richiesto, si perviene a una descrizione piuttosto dettagliata delle interazioni tra classi e oggetti che consentiranno di realizzare tale comportamento.

I diagrammi delle classi di analisi sono una parte fondamentale della realizzazione di caso d'uso. Dovrebbero "raccontare una storia" che riguarda il sistema: come un insieme di classi possano essere relazionate, in modo che le loro istanze collaborino per realizzare il comportamento specificato in uno, o più, casi d'uso.

Oltre ai diagrammi delle classi, si possono creare diagrammi che esplicitano come oggetti specifici di queste classi collaborano e interagiscono tra di loro per realizzare una parte, o tutto, del comportamento di un caso d'uso. Questi diagrammi sono noti come diagrammi di interazione e ne esistono di due tipi: diagrammi di collaborazione e diagrammi di sequenza, entrambi descritti in modo approfondito, in questo capitolo.

La modellazione OO è un processo iterativo, non ci si deve quindi stupire se, una volta che si inizia a modellare in modo più dettagliato, vengono alla luce nuovi requisiti, oppure si rende necessario revisionare i casi d'uso esistenti. Tutto questo fa parte delle attività di realizzazione di caso d'uso. È sempre necessario tenere aggiornata tutta la documentazione, ogni volta che emerge qualche nuova informazione relativa al sistema. Quindi, qualora si renda necessario, bisogna aggiornare i documenti dei requisiti ed eventualmente anche i casi d'uso stessi.

12.5 Diagrammi di interazione

Nell'UML i diagrammi di interazione modellano le collaborazioni e le interazioni tra oggetti che realizzano un caso d'uso o una parte di un caso d'uso. In effetti esiste un unico modello UML sottostante che contiene la definizione dell'interazione tra oggetti, ma di questo modello esistono due viste distinte.

- Diagrammi di collaborazione: enfatizzano le relazioni strutturali tra gli oggetti e sono utilissimi per l'analisi, soprattutto per creare rapidamente una bozza di una collaborazione tra oggetti.

- Diagrammi di sequenza: enfatizzano la sequenza temporale degli scambi di messaggi tra diversi oggetti. Gli utenti spesso riescono a comprendere i diagrammi di sequenza meglio che non quelli di collaborazione, perché sono molto più facili da leggere. I diagrammi di sequenza tendono a diventare molto affollati, e molto infretta.

Trattandosi di due rappresentazioni diverse dello stesso modello UML sottostante, i diagrammi di collaborazione e i diagrammi di sequenza sono isomorfi. Per questo motivo, molti strumenti di CASE riescono a effettuare una conversione automatica tra i due tipi di diagramma.

Entrambi i tipi di diagrammi possiedono due forme.

- Forma descrittore: descrive le collaborazioni e le interazioni tra i *ruoli* che le istanze dei classificatori possono svolgere nel sistema.
- Forma istanza: descrive le collaborazioni e le interazioni tra le effettive *istanze* dei classificatori.

Nel Capitolo 1 si era detto che l'UML prevede l'uso di molti tipi diversi di classificatore e di istanza. La maggior parte delle volte, quando si lavora con i diagrammi di interazione, si usano classi (come classificatori) e oggetti (come istanze). Tuttavia, è bene ricordarsi sempre che in questi diagrammi possono comparire anche altri tipi di classificatore e di istanza, secondo le necessità; ne vedremo un esempio specifico nel Capitolo 18.

12.6 Collaborazioni e interazioni

Prima di esaminare i diagrammi di interazione e il loro ruolo nelle realizzazioni di caso d'uso, occorre chiarire bene i concetti UML di collaborazione, interazione e ruolo.

- Una collaborazione descrive un insieme statico di relazioni tra istanze, e i ruoli che queste istanze svolgono in queste relazioni.
- Un'interazione descrive come le istanze interagiscano tra loro in modo dinamico. Descrive i messaggi che le istanze si scambiano. Dato che le istanze possono scambiarsi messaggi soltanto dopo aver stabilito una qualche relazione, un'interazione può esistere soltanto nel contesto di una collaborazione.
- Un ruolo è una modalità particolare di comportamento o di utilizzo di qualcosa. Oltre a collegamenti e istanze, i diagrammi di interazione possono indicare ruoli di classificatore, i quali definiscono come possano essere utilizzate le istanze dei classificatori e ruoli di associazione, i quali definiscono come possano essere utilizzate le istanze delle associazioni (i collegamenti).

Si pensi a una collaborazione come a un insieme di istanze con le loro relazioni, e alle interazioni come a un modo specifico in cui queste istanze interagiscono scambiandosi messaggi tramite le relazioni stabilite. I ruoli definiscono come qualcosa può comportarsi o essere utilizzata, all'interno di una certa collaborazione.

Forse un esempio del mondo reale può aiutare a chiarire. Uno degli autori ha da molto tempo una collaborazione con la Zühlke Engineering; questa collaborazione ha tuttavia comportato molte diverse interazioni: la fornitura di corsi di addestramento (ruolo: insegnante), consulenze (ruolo: consulente), la produzione di materiale per i corsi (ruolo: autore) ecc.

12.7 Diagrammi di collaborazione

I diagrammi di collaborazione si concentrano sugli aspetti strutturali dell'interazione tra oggetti. Si presentano in due differenti forme: descrittore e istanza. La forma descrittore offre una vista molto generica della collaborazione, specificando i ruoli svolti dalle istanze (detti: ruoli di classificatore) e le relazioni tra questi ruoli (dette: ruoli di associazione). Il diagramma di collaborazione in forma istanza è più concreto. Mostra le effettive istanze di classificatori (solitamente, oggetti) e i collegamenti tra queste istanze. Come riportato nella Tabella 12.2., le due forme hanno capacità di fissare informazioni del modello, leggermente diverse.

Tabella 12.2

Contenuto dei diagrammi di collaborazione	
Forma descrittore	Forma istanza
Ruoli di classificatore (solitamente, ruoli ricoperti da classi)	Istanze (solitamente, oggetti)
Ruoli di associazione (ruoli ricoperti dalle associazioni)	Collegamenti
Messaggi	Messaggi
—	Iterazioni
—	Ramificazioni

La differenza fondamentale è che non si possono mostrare ramificazioni e iterazioni sui diagrammi di collaborazione in forma descrittore. Questo perché le iterazioni e le ramificazioni richiedono la presenza di istanze, che la forma descrittore, rappresentando esclusivamente dei ruoli, non include.

Il diagramma di collaborazione in forma istanza è la più utile e anche quella usata più comunemente, in quanto consente di specificare le interazioni tra oggetti. Siccome tratta le istanze effettive, è più concreta della forma descrittore ed è anche, sotto certi aspetti, più facile da visualizzare.

12.7.1 Diagrammi di collaborazione in forma descrittore

Il diagramma di collaborazione in forma descrittore mostra i ruoli che verranno svolti da istanze non specificate di classificatori (solitamente, classi) e da istanze non specificate di associazioni (solitamente, collegamenti). Il diagramma non mostra *alcuna* particolare istanza o associazione. Questo tipo di diagramma è utile quando si vuole modellare una collaborazione in modo molto generico.

Il modo più facile per comprendere i diagrammi di collaborazione in forma descrittore è quello di esaminarne un semplice esempio. Si veda il diagramma delle classi riportato nella Figura 12.4.

In questo diagramma, la classe **Parte** rappresenta la nozione di una persona o di un'azienda dotata di nome e indirizzo; abbiamo nascosto l'attributo nome, in quanto non è importante in questo contesto. In un sistema reale, il concetto astratto di **Parte** verrebbe di solito specializzato in concetti più specifici, come **Persona** e **Azienda**, ma per questo esempio si tratterebbe di una complessità eccessiva e inutile.

La Figura 12.5 illustra il diagramma di collaborazione in forma descrittore di questo modello. Il diagramma mostra i ruoli di classificatore e i ruoli di associazione. Un ruolo di classificatore è un ruolo che può essere svolto da un'istanza del classificatore, mentre il ruolo di associazione indica un ruolo che può essere svolto da un'istanza dell'associazione (un collegamento).

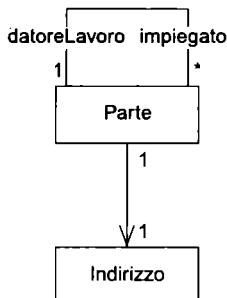


Figura 12.4

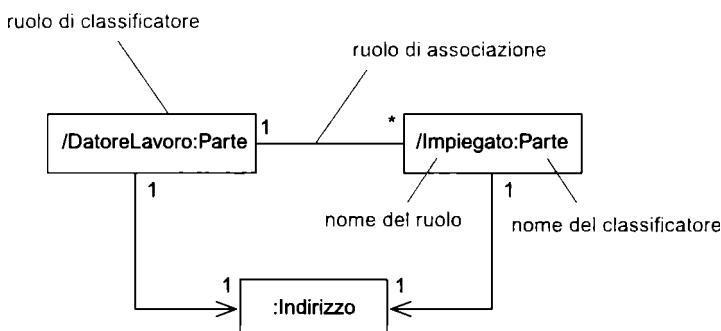


Figura 12.5

Sebbene questo diagramma non contenga molte informazioni oltre a quelle già presenti nel diagramma delle classi, con esso abbiamo potuto enfatizzare i diversi ruoli che vengono interpretati da istanze delle classi. Per questo motivo può, a volte, essere un diagramma utile. I nomi dei ruoli di classificatore hanno la forma:

/NomeRuolo:NomeClassificatore

Il nome del ruolo deve *sempre* iniziare con una barra inclinata (*slash*), mentre il nome del classificatore deve *sempre* essere preceduto dai due punti. Entrambi i nomi sono facoltativi, ma ne deve essere specificato *almeno uno*. Si può omettere il nome del classificatore, se si vuole solo indicare che un'istanza svolge un ruolo particolare, ma non se ne vuole indicare la classe. Similmente, si può omettere il nome del ruolo per indicare che il classificatore svolge un ruolo nella collaborazione, ma non si vuole assegnare un nome a quel ruolo. Nella Figura 12.5 è presente un esempio di quest'ultimo caso.

12.7.2 Diagrammi di collaborazione in forma istanza

Il diagramma di collaborazione in forma istanza mostra la collaborazione tra istanze di classificatori, connesse con collegamenti. La Figura 12.6 riporta il diagramma in forma istanza corrispondente a quello presente nella Figura 12.5.

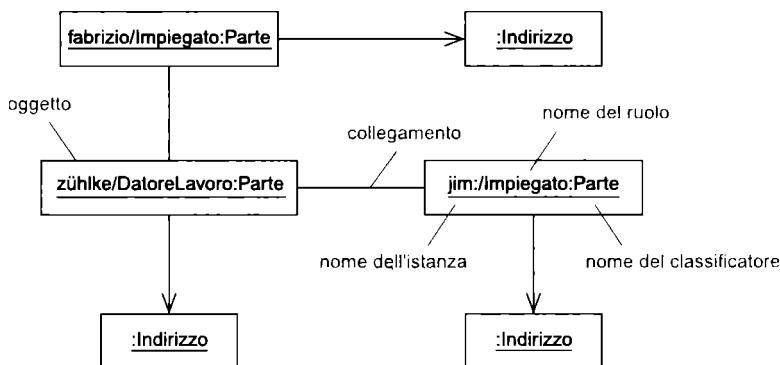


Figura 12.6

Il diagramma di collaborazione in forma istanza mostra le specifiche istanze che collaborano in un determinato istante. Si tratta di un'istantanea di un sistema OO in esecuzione. I nomi delle istanze vengono indicati come segue:

nomelstanza/NomeRuoloClassificatore:NomeClassificatore

Il nome del ruolo di classificatore inizia *sempre* con una barra inclinata (*slash*), mentre il nome del classificatore inizia *sempre* con i due punti. Tutte le parti del nome sono facoltative, ma deve esserne presente *almeno una*. Il nome del ruolo di classificatore può corrispondere a un nome di ruolo del diagramma di collaborazione in forma descrittore, seppure questo esista.

12.7.3 Interazioni tra oggetti

Né la forma descrittore, né la forma istanza dei diagrammi di collaborazione visti fin qui, mostrano interazioni tra ruoli di classificatore o tra istanze. Si sono limitate, invece, a mostrare il contesto in cui queste interazioni potrebbero avvenire. Per utilizzare questi diagrammi in una realizzazione di caso d'uso, occorre però illustrare come gli oggetti del sistema realizzino effettivamente il comportamento specificato nel caso

d'uso. Per fare questo, è necessario aggiungere le interazioni al diagramma. Si prenda, per esempio, un semplice sistema di registrazione di corsi. Il caso d'uso da realizzare si chiama AggiungiCorso e la Figura 12.7 ne riporta la specifica.

Caso d'uso: AggiungiCorso	
ID:	UC8
Attori:	Registrante
Precondizioni:	Il Registrante è stato autenticato dal sistema.
Sequenza degli eventi:	<ol style="list-style-type: none"> 1. Il Registrante seleziona "aggiungi corso". 2. Il sistema accetta il nome del nuovo corso. 3. Il sistema crea il nuovo corso.
Postcondizioni:	Un nuovo corso è stato aggiunto al sistema.

Figura 12.7

La specifica di questo caso d'uso è stata semplificata, in modo da potersi concentrare sulle interazioni tra oggetti, e non sulla modellazione del caso d'uso. Il Capitolo 4 tratta in modo dettagliato i casi d'uso.

È possibile applicare le tecniche usuali dell'analisi (descritte nel Capitolo 8) per estrarre da questo caso d'uso le classi elencate nella Tabella 12.3.

Tabella 12.3

Elemento	Tipo	Significato
Registrante	Attore	Responsabile per il mantenimento dell'informazione sui corsi nel sistema di registrazione dei corsi
Corso	Classe	Contiene i dettagli di un corso
GestoreRegistrazioni	Classe	Responsabile per il mantenimento di una collezione di corsi

A dire il vero, la classe GestoreRegistrazioni non era esplicitamente menzionata nel caso d'uso AggiungiCorso. Tuttavia, il caso d'uso AggiungiCorso sottintende la responsabilità implicita “mantenere un insieme di Corsi”, e, quindi, è stata introdotta la classe GestoreRegistrazioni per assegnarle tale responsabilità.

Il diagramma delle classi relativo a questa realizzazione di caso d'uso ha l'aspetto illustrato nella Figura 12.8.

La Figura 12.9 illustra il diagramma di collaborazione in forma istanza che realizza il comportamento specificato nel caso d'uso AggiungiCorso. In questo diagramma, vengono creati due nuovi oggetti Corso, quindi il caso d'uso viene di fatto eseguito *due volte*.



Figura 12.8

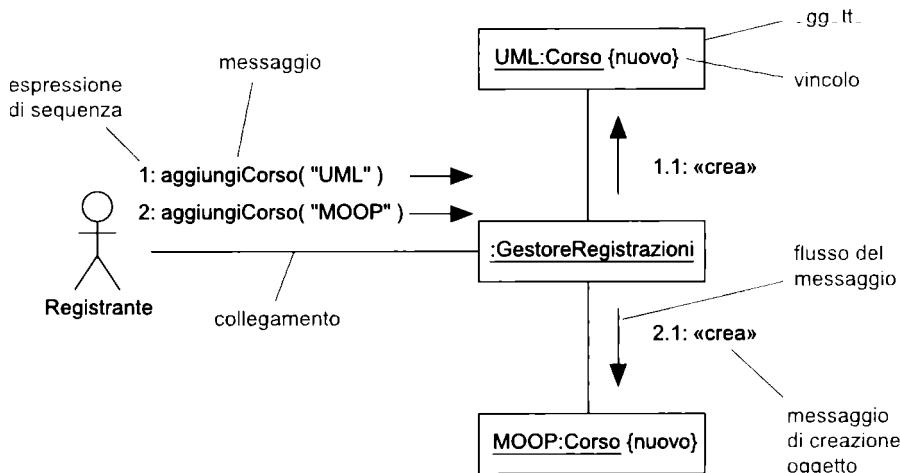


Figura 12.9

La Figura 12.9 presenta molti elementi della sintassi per i diagrammi di collaborazione. È già stato considerato come si rappresentano oggetti (rettangoli contenenti il nome dell'oggetto sottolineato) e i collegamenti tra oggetti (linee che collegano gli oggetti). Le frecce etichettate collocate nei pressi dei ruoli di associazione o dei collegamenti, rappresentano i messaggi. I messaggi devono essere indicati nel seguente modo:

EspressioneSequenza valoreRestituito := nomeMessaggio(argomento1 , argomento2 , ...)

L'espressione di sequenza indica l'ordine in cui verrà inviato il messaggio: di solito è un numero. Il valore restituito è il nome di una variabile a cui assegnare un eventuale valore restituito dal messaggio; di solito questa variabile è un attributo dell'oggetto che invia il messaggio.

Il nome del messaggio viene scritto in CamelCase, iniziando con una lettera minuscola. Gli argomenti sono l'elenco dei parametri del messaggio. Lo stile utilizzato per la punta della freccia indica il tipo di comunicazione rappresentato dal messaggio; la Tabella 12.4 riporta i possibili stili e significati.

Quando un oggetto riceve un messaggio, esegue un'operazione che abbia la stessa segnatura. Quindi, nella classe di un oggetto destinatario dovrebbero esistere operazioni corrispondenti alle segnature di tutti i messaggi ricevuti. In realtà, l'UML consente di avere dei inconsistenti e disallineamenti tra le operazioni delle classi e i messaggi presenti nei diagrammi di interazione; questa flessibilità permette di lavorare sui modelli in modo molto dinamico. Tuttavia, quando si giunge verso la fine delle attività di analisi o all'inizio di quelle di progettazione, operazioni e messaggi *devono* essere riallineati in modo consistente.

Tabella 12.4

Stile messaggio	Significato
→	Chiamata di procedura: il mittente attende fintantoché il ricevente non ha finito. Questa è la scelta più diffusa
→	Comunicazione asincrona: il mittente prosegue dopo l'invio del messaggio, senza attendere che il ricevente abbia finito. Usato spesso per i casi di concorrenza.
→	Ritorno da una chiamata di procedura: anche se il ritorno è sempre implicito in una chiamata di procedura, è possibile indicarlo esplicitamente con questa freccia.

Quando un oggetto sta eseguendo un'operazione, si dice che ha il focus di controllo. L'evolversi nel tempo della collaborazione provoca lo spostamento del focus di controllo tra i diversi oggetti modellati: questo spostamento viene detto flusso del controllo.

L'UML mette a disposizione alcuni vincoli standard da utilizzare, nei diagrammi di interazione, per indicare la creazione e la distruzione di istanze o di collegamenti; questi sono elencati nella Tabella 12.5.

Tabella 12.5

Vincolo	Significato
{nuovo}	L'istanza o il collegamento viene creato durante l'interazione
{distrutto}	L'istanza o il collegamento viene distrutto durante l'interazione
{transiente}	L'istanza o il collegamento viene creato e poi distrutto durante l'interazione: equivalente a usare {nuovo} e {distrutto} in sequenza, ma è più comprensibile

Per dimostrare come funzionano i diagrammi di collaborazione, e per illustrare le informazioni che sono in grado di fissare, la Tabella 12.6 descrive la sequenza di esecuzione dell'esempio riportato nella Figura 12.9.

Vista la quantità di elementi diversi presenti in un diagramma di collaborazione, questi possono risultare inizialmente difficili da leggere. I punti fondamentali da tenere presente sono che l'invio di un messaggio equivale alla chiamata di una funzione (ovvero un'operazione in analisi) di un oggetto e che i numeri di sequenza indicano la sequenza temporale e l'annidamento delle chiamate a funzione dentro altre chiamate a funzione. Le chiamate annidate sono anche dette "flussi di controllo annidati".

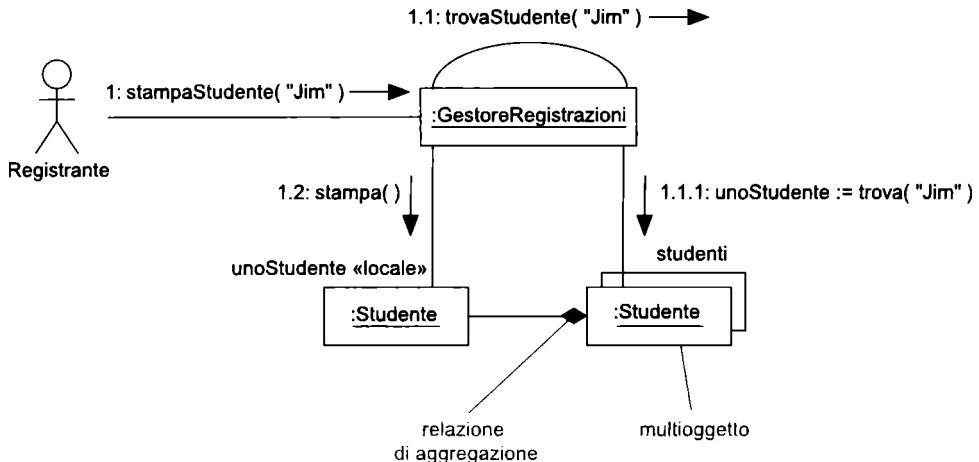
Lavorando sui diagrammi di collaborazione emergono sempre nuovi attributi e operazioni delle classi di analisi. Le attività di aggiornamento dei diagrammi delle classi di analisi, con queste nuove informazioni fanno parte integrante del processo di realizzazione di caso d'uso.

Tabella 12.6

Descrizione del flusso relativo alla Figura 12.9	
1: aggiungiCorso("UML")	<p>Il messaggio aggiungiCorso(...) viene inviato all'oggetto <u>:GestoreRegistrazioni</u> con il parametro "UML".</p> <p>L'oggetto <u>:GestoreRegistrazioni</u> chiama un'operazione di nome aggiungiCorso(...) con il parametro "UML" e il focus di controllo passa a questa operazione.</p>
1.1 «crea»	<p>Il numero di sequenza 1.1 indica che il focus di controllo è ancora nell'operazione aggiungiCorso(...).</p> <p>Il <u>:GestoreRegistrazioni</u> invia un messaggio anonimo con stereotipo «crea».</p> <p>Questo tipo di messaggi crea un nuovo oggetto, e questo particolare messaggio crea un oggetto: <u>UML:Corso</u> che ha un vincolo {nuovo} per indicare che viene creato in questa collaborazione, e prima non esisteva.</p> <p>In seguito, durante l'analisi o la progettazione, si darà un nome a questo messaggio anonimo, e forse anche dei parametri.</p> <p>Per adesso, è sufficiente che indichi che si sta creando un nuovo oggetto <u>UML:Corso</u>.</p> <p>Dopo la creazione dell'oggetto, non vi sono più messaggi nel focus di controllo per aggiungiCorso(...) e questo flusso restituisce il controllo al chiamante.</p>
2: aggiungiCorso("MOOP")	<p>Il messaggio aggiungiCorso(...) viene inviato all'oggetto <u>:GestoreRegistrazioni</u> con il parametro "MOOP".</p> <p>Il <u>:GestoreRegistrazioni</u> chiama un'operazione di nome aggiungiCorso(...) con parametro "MOOP".</p> <p>Il focus di controllo passa all'operazione aggiungiCorso(...).</p>
2.1: «crea»	<p>Il numero di sequenza 2.1 indica che si è ancora nel focus di controllo di aggiungiCorso(...). Il <u>:GestoreRegistrazioni</u> invia un messaggio anonimo, con stereotipo «crea», che produce un nuovo oggetto: <u>MOOP:Corso</u>.</p> <p>Anche in questo caso, l'oggetto appena creato ha il vincolo {nuovo}, per indicare che viene creato in questa collaborazione, e che prima non esisteva.</p> <p>Dopo la creazione dell'oggetto, il focus di controllo per aggiungiCorso(...) viene restituito al chiamante.</p>

12.7.4 Multioggetti

Un multioggetto rappresenta un insieme di oggetti: è un modo di rappresentare le collezioni di oggetti nei diagrammi di collaborazione. I messaggi inviati a un multioggetto vengono ricevuti e processati dall'insieme e *non* da un suo singolo oggetto. L'esempio nella Figura 12.10 è anch'esso riferito a un semplice sistema di registrazione dei corsi. Il caso d'uso realizzato in questo diagramma di collaborazione si chiama StampaDettagliStudente. In questo esempio, lo scopo è quello di stampare i dettagli di uno studente identificato dal nome "Jim". Si procede come descritto nella Tabella 12.7.

**Figura 12.10****Tabella 12.7****Descrizione del flusso relativo alla Figura 12.10**

1: stampaStudente("Jim")	L'attore Registrante invia il messaggio stampaStudente("Jim") all'oggetto <u>GestoreRegistrazioni</u> . Per prima cosa, il <u>GestoreRegistrazioni</u> deve trovare l'oggetto Studente richiesto nell'insieme di oggetti che gestisce.
1.1: trovaStudente("Jim")	Il <u>GestoreRegistrazioni</u> invia a se stesso il messaggio trovaStudente("Jim"). Questa si chiama auto-delegazione.
1.1.1: trova("Jim")	Il <u>GestoreRegistrazioni</u> invia il messaggio trova("Jim") al multioggetto di nome studenti. Questo messaggio <i>non è</i> destinato a un oggetto partolare, ma all'intero insieme. Viene restituito un riferimento all'oggetto studente cercato nella variabile locale unoStudente.
1.2: stampa()	Il <u>GestoreRegistrazioni</u> invia il messaggio stampa() all'oggetto referenziato dalla variabile locale unoStudente. Si osservi che si è dovuto estrarre l'oggetto richiesto dal multioggetto per potergli inviare un messaggio. Si usa lo stereotipo "locale" per indicare che questo riferimento a oggetto è locale all'operazione stampa() del <u>GestoreRegistrazioni</u> . Il numero di sequenza 1.2 indica che ci troviamo ancora nel focus di controllo dell'operazione stampaStudente("Jim") di <u>GestoreRegistrazioni</u> .

La relazione di aggregazione tra il multioggetto studenti e l'oggetto locale unoStudente, indica semplicemente che l'oggetto referenziato da unoStudente appartiene all'insieme multioggetto.

Sebbene l'UML affermi che un multioggetto è un insieme di oggetti, non dice quali siano i metodi supportati da un multioggetto. Questo perché, quando si arriva alla progettazione dettagliata, ogni multioggetto verrà sostituito con un'istanza specifica di una qualche classe contenitore.

Tuttavia, durante l'analisi si presume comunemente che i multioggetti possano perlomeno ricevere i seguenti messaggi:

- `trova(identificatoreUnivoco)`: restituisce un oggetto specifico, corrispondente all'identificatore univoco passato come parametro;
- `include(unOggetto)`: restituisce vero se il multioggetto contiene unOggetto;
- `conteggio()`: restituisce il numero di elementi contenuti nel multioggetto.

Se il messaggio inviato al multioggetto è preceduto da un indicatore di iterazione (*), allora il messaggio viene inoltrato a ogni oggetto elemento presente nel multioggetto.

Per inviare un messaggio a una specifica istanza presente nel multioggetto, occorre effettuare due distinte azioni:

- individuare l'istanza desiderata;
- inviare il messaggio a quell'istanza.

Per indicare che il messaggio viene inviato a una specifica istanza presente nell'insieme, occorre mostrarla *al di fuori* del multioggetto, ma ancora connessa a esso mediante una relazione di aggregazione, come illustrato nella Figura 12.10. La semantica delle relazioni di aggregazione viene approfondita nel Capitolo 16; per ora è sufficiente pensare che significa "una parte di".

12.7.5 Iterazione

Si indica un'iterazione premettendo al numero di sequenza un indicatore di iterazione (*) e, opzionalmente, un'espressione di iterazione. L'UML non prevede una notazione formale per le espressioni di iterazione; va, quindi, bene una qualunque espressione che sia leggibile, comprensibile e sensata. La Tabella 12.8 riporta alcune espressioni di iterazione di uso comune.

Quando l'indicatore di iterazione non è accompagnato anche da un'espressione, allora si itera sull'intero insieme.

Tabella 12.8

Espressione di iterazione	Significato
<code>[i:=1..n]</code>	Ripetere n volte
<code>[i:=1..7]</code>	Ripetere 7 volte
<code>[fintantoché(espressione booleana)]</code>	Ripetere fintantoché l'espressione booleana rimane vera
<code>[fino-a (espressione booleana)]</code>	Ripetere fino a che l'espressione booleana non diventi vera
<code>[per-ogni (espressione che risulta in un insieme d'oggetti)]</code>	Ripetere su ogni oggetto dell'insieme

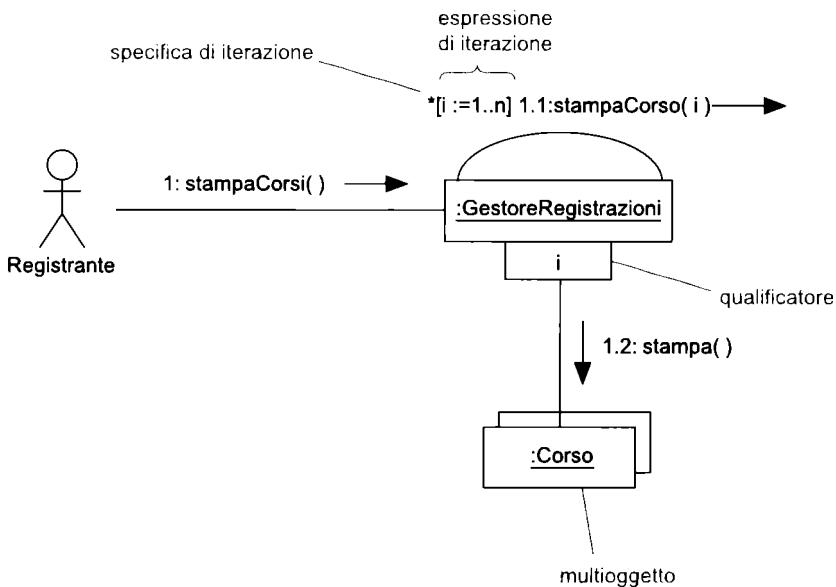


Figura 12.11

L'esempio nella Figura 12.11 mostra un'espressione con cui si itera su tutto l'insieme di corsi, stampandoli tutti, uno per volta. Si osservi l'uso del contatore (*i*) come qualificatore per indicare uno specifico elemento nel multioggetto, a cui poi si invia il messaggio stampa.

Si otterebbe lo stesso risultato inviando direttamente al multioggetto il messaggio stampa con l'indicatore di iterazione ed eliminando il qualificatore. Tuttavia, nella Figura 12.11 si è scelto di specificare esplicitamente come si svolge l'iterazione e come viene indicizzato il multioggetto. Visto il livello di dettaglio raggiunto, si tratta più propriamente di un modello della progettazione, che non di un modello dell'analisi.

L'indicatore di iterazione (*) indica sempre l'elaborazione sequenziale delle istanze del multioggetto. Tuttavia, a volte si vuole indicare che le istanze possono essere elaborate contemporaneamente in parallelo. In questo caso speciale, si usa fa seguire all'indicatore di iterazione da due barre inclinate (*slash*), per indicare l'elaborazione parallela (*//).

12.7.6 Ramificazione e auto-delegazione

Le ramificazioni possono essere modellate semplicemente aggiungendo delle condizioni davanti ai messaggi. Una condizione è un'espressione Booleana che può risultare vera o falsa. Il messaggio viene inviato solo se la condizione risulta vera.

Nella Figura 12.12 si vede un esempio di ramificazione del sistema di registrazione corsi. Questo diagramma di collaborazione realizza il caso d'uso *IscriviStudenteAlCorso*. In questo sistema l'iscrizione è un processo con tre passi:

- trovare la giusta scheda studente: non si possono iscrivere studenti che non siano già nel sistema;
- trovare il corso giusto;
- iscrivere lo studente al corso.

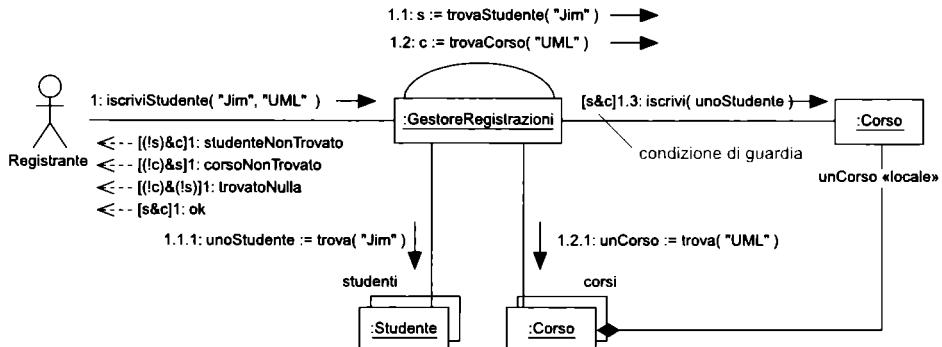


Figura 12.12

Tabella 12.9

Descrizione del flusso relativo alla Figura 12.11

1: iscriviStudente("Jim" "UML")	L'attore Registrante invia il messaggio iscriviStudente("Jim" "UML") all'oggetto :GestoreRegistrazioni.
1.1: trovaStudente("Jim")	Il :GestoreRegistrazioni invia a sé stesso il messaggio trovaStudente("Jim"). Il valore di questa operazione viene memorizzato nella variabile s. Avrà valore vero se la ricerca è riuscita, falso se la ricerca è fallita.
1.1.1: trova("Jim")	Il :GestoreRegistrazioni invia a sé stesso il messaggio trova("Jim") al multioggetto di nome studenti. Se la ricerca ha successo, viene assegnato alla variabile locale unoStudente un riferimento all'oggetto Studente trovato.
1.2: trovaCorso("UML")	Il :GestoreRegistrazioni invia a sé stesso il messaggio trovaCorso("UML"). Il valore di questa operazione viene memorizzato nella variabile c. Avrà valore vero se la ricerca è riuscita, falso se la ricerca è fallita.
1.2.1: trova("UML")	Il :GestoreRegistrazioni invia il messaggio trova("UML") al multioggetto di nome corsi. Se la ricerca ha successo, viene assegnato alla variabile locale unCorso un riferimento all'oggetto Corso trovato.
[s&c]1.3: iscrivi(unoStudente)	Il :GestoreRegistrazioni invia il messaggio iscrivi(unoStudente) all'oggetto di nome Corso. Questo messaggio è soggetto a una condizione e verrà inviato solo se sia c che s danno come risultato vero. In altre parole, si tenta di iscrivere lo Studente al Corso solo se entrambi gli oggetti sono stati trovati.
[s&c]1: ok	iscriviStudente(...) restituisce ok se lo Studente è stato iscritto al Corso (sia c che s valgono vero).
[!(s)&c]1: studenteNonTrovato	iscriviStudente(...) restituisce studenteNonTrovato se non è stato trovato lo Studente (s vale falso).
[!(c)&s]1: corsoNonTrovato	iscriviStudente(...) restituisce corsoNonTrovato se non è stato trovato il Corso (c vale falso).
[!(c)&!(s)]1: trovatoNulla	iscriviStudente(...) restituisce trovatoNulla se non è stato trovato né lo Studente, né il Corso (sia c che s valgono falso).

L'auto-delegazione si ha quando un oggetto chiama sé stesso. Se ne vede un esempio nei passi 1.1 e 1.2 nella Figura 12.12. L'auto-delegazione è molto comune nei sistemi OO. Gli oggetti espongono un insieme di servizi pubblici (le operazioni pubbliche) che possono essere chiamate da oggetti cliente, ma generalmente hanno anche un insieme di operazioni "ausiliarie" private, progettate appositamente per essere chiamate dall'oggetto stesso.

Nella Figura 12.12 si è fatto un ampio uso delle condizioni, per far vedere come le si adopera nei diagrammi di collaborazione. Non esiste una sintassi formale per le condizioni, ma queste sono spesso espressioni che coinvolgono variabili temporanee nell'ambito del focus di controllo corrente, o attributi delle classi coinvolte nella collaborazione. Nella Figura 12.12 il successo o la riuscita dei metodi `trovaStudente(...)` e `trovaCorso(...)` vengono memorizzati in due variabili temporanee: `s` e `c`. I valori di queste variabili vengono poi usati per la ramificazione al passo 1.3 e, infine, per decidere quale valore restituire al `Registrante`. La Tabella 12.9 riporta una descrizione della sequenza presente nella Figura 12.12.

È difficolto illustrare chiaramente le ramificazioni sui diagrammi di collaborazione; il diagramma si riempie presto di condizioni e diventa presto troppo complesso per essere utile. In generale, su questi diagrammi conviene usare solo ramificazioni molto semplici. Le ramificazioni complesse sono molto più facili da rappresentare nei diagrammi di sequenza, trattati tra breve.

12.7.7 Concorrenza: oggetti attivi

La concorrenza può essere facilmente modellata nei diagrammi di collaborazione. Il principio di base è che ogni *thread* o processo concorrente viene modellato come un oggetto attivo, che incapsula un proprio flusso di controllo.

Gli oggetti attivi eseguono in concorrenza e possono, contemporaneamente, avere ciascuno un proprio focus di controllo. Gli oggetti attivi vengono disegnati come quelli normali, ma con il bordo del rettangolo più spesso. Volendo, è anche possibile aggiungere la proprietà `{attivo}` al rettangolo.

La concorrenza tende a essere molto importante per i sistemi *embedded*, quali il software che aziona una macchina per lo sviluppo fotografico o quello di una sportello bancomat. Per esaminare la concorrenza conviene, dunque, prendere in considerazione un sistema *embedded* molto semplice: un sistema di sicurezza. Questo sistema di sicurezza può monitorare un insieme di sensori di intrusione e di incendio. Quando viene azionato un sensore, il sistema suona un allarme. La Figura 12.13 illustra il modello del caso d'uso di questo sistema.

La Figura 12.14 riporta alcune specifiche di caso d'uso del sistema. Non prendiamo in considerazione il caso d'uso `AttivaSoloIncendio`, in modo da poterci concentrare sui soli aspetti di concorrenza del sistema.

Oltre ai requisiti funzionali espressi dal caso d'uso, il sistema ha anche due requisiti non funzionali che ci interessano.

- Il sistema d'allarme sarà attivato e disattivato mediante una chiave.
- La chiave consente di commutare tra tre modalità: sistema disattivato, sensori di incendio e di intrusione attivati, solo sensori di incendio attivati.

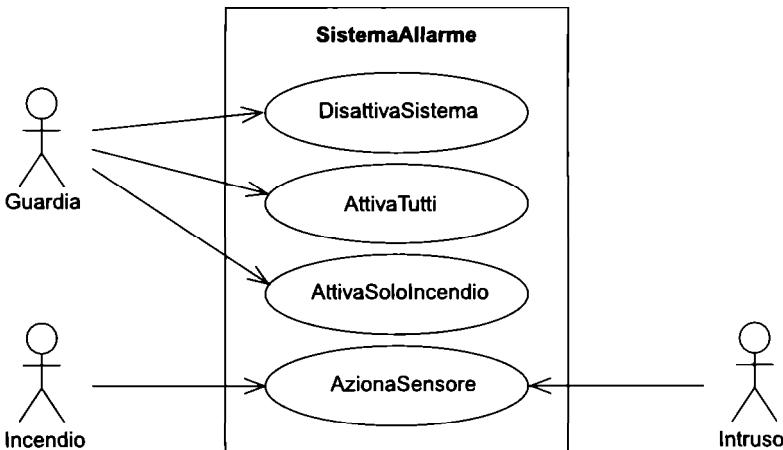


Figura 12.13

Caso d'uso: DistattivaSistema	Caso d'uso: AttivaTutti	Caso d'uso: AzionaSensore
ID: UC1	ID: UC2	ID: UC3
Attori: Guardia	Attori: Guardia	Attori: Incendio Intruso
Precondizioni: La Guardia dispone della chiave di attivazione.	Precondizioni: La Guardia dispone della chiave di attivazione.	Precondizioni: Il sistema è attivato.
Sequenza degli eventi: 1. La Guardia usa la chiave di attivazione per disattivare il sistema. 2. Il sistema interrompe il monitoraggio dei sensori antifurto ed antiincendio.	Sequenza degli eventi: 1. La Guardia usa la chiave di attivazione per attivare il sistema. 2. Il sistema inizia il monitoraggio dei sensori antifurto ed antiincendio. 3. Il sistema emette un segnale di conferma con la sirena per indicare che è attivo.	Sequenza degli eventi: 1. Se l'attore Incendio aziona un SensoreIncendio 1.1. La Sirena suona l'allarme antiincendio. 2. Se l'attore Intruso aziona un SensoreIntrusione 2.1. La Sirena suona l'allarme antifurto.
Postcondizioni: Il sistema è disattivato. Il sistema non effettua il monitoraggio dei sensori.	Postcondizioni: Il sistema è attivato. Il sistema effettua il monitoraggio dei sensori.	Postcondizioni: La Sirena suona.

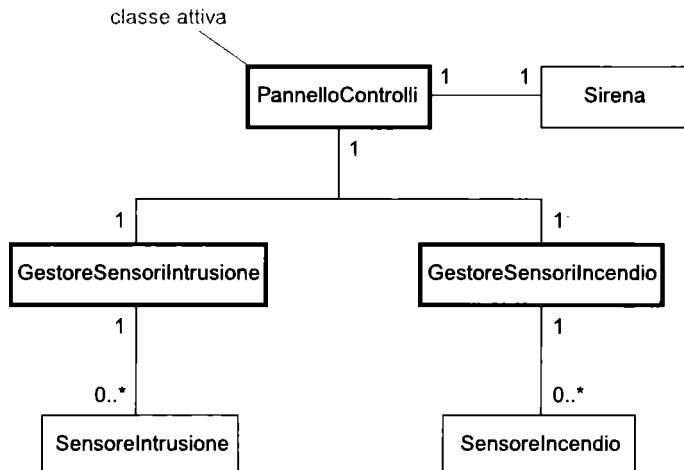
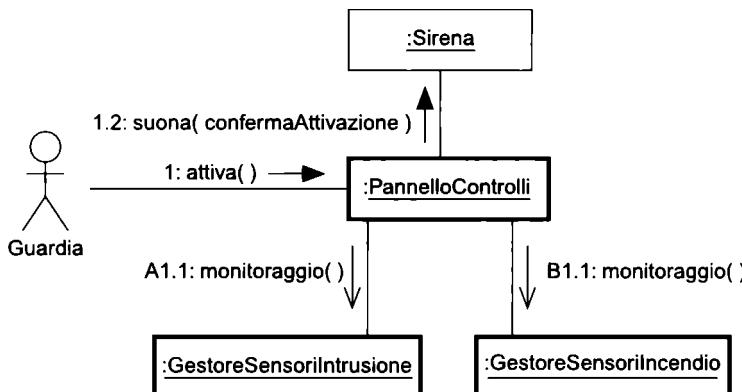
Figura 12.14

Si individuano le classi con le seguenti considerazioni. Nei sistemi *embedded*, l'hardware su cui viene eseguito il sistema è un'ottima fonte di potenziali classi. Infatti, capita spesso che l'architettura software migliore corrisponda strettamente all'architettura fisica del sistema. In questo caso, l'hardware del sistema di sicurezza è costituito da quattro componenti: il pannello dei controlli, la sirena, i sensori di incendio e i sensori di intrusione. Scoperchiando il pannello dei controlli, osserviamo che nasconde una scheda controllore per ciascun tipo di sensore.

Disponendo di queste informazioni relative ai casi d'uso e all'hardware, possiamo estrapolare il diagramma delle classi riportato nella Figura 12.15.

Si noti che le classi PannelloControlli, GestoreSensorIntrusione e GestoreSensorIncendio hanno i bordi spessi: questo indica che sono classi attive. Gli oggetti attivi sono istanze di classi attive. In questo caso occorre della concorrenza, in quanto il sistema di sicurezza deve continuamente monitorare i sensori di incendio e di intrusione.

Ora si hanno informazioni sufficienti per creare qualche diagramma. La Figura 12.16 illustra il diagramma di collaborazione in forma istanza relativo al caso d'uso AttivaTutti e la Tabella 12.10 descrive il flusso di questo diagramma.

**Figura 12.15****Figura 12.16****Tabella 12.10****Descrizione del flusso relativo alla della Figura 12.16**

1: attiva()	Quando la Guardia attiva il sistema , viene inviato all'oggetto :PannelloControlli il messaggio che rappresenta il thread di controllo principale nel programma. :PannelloControlli invia il messaggio <i>asincrono</i> monitoraggio() al :GestoreSensoriIntrusione e al :GestoreSensoriIncendio .
A1.1: monitoraggio()	Si assegna lo stesso numero di sequenza a entrambi per indicare che l'ordine non ha importanza: ogni messaggio causa l'avvio di un nuovo thread.
B1.1: monitoraggio()	Il thread A effettua il monitoraggio dei sensori di intrusione: lo si modella con l'oggetto attivo :GestoreSensoriIntrusione .
1.2: suona(confermaAttivazione)	Il sistema emette un segnale di conferma con la sirena per indicare che è attivo.

Passiamo ora al caso d'uso **AzionaSensore**. La Figura 12.17 ne illustra un diagramma di collaborazione in cui un intruso aziona il sensore collocato nell'ingresso dell'edificio e la Tabella 12.11 della pagina seguente descrive il flusso di questo diagramma.

L'esempio nella Figura 12.17 mostra uno schema abbastanza tipico per i sistemi concorrenti: osserviamo che, al centro, c'è un oggetto controllore (**:PannelloControlli**) in questo caso) che coordina i *thread* secondari, i quali sono stati modellati come oggetti attivi. Il motivo per cui questo schema è molto comune nei sistemi concorrenti è che occorre poter iniziare, fermare e sincronizzare i *thread* concorrenti in modo controllato. Una tecnica semplice è proprio quella di assegnare queste responsabilità a un oggetto controllore centrale.

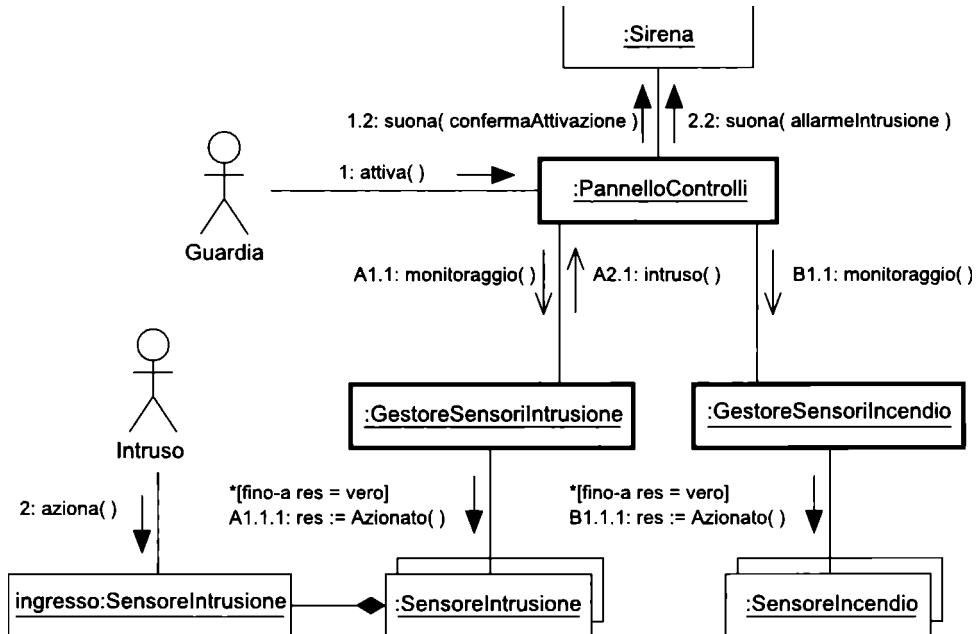


Figura 12.17

12.7.8 Stato di un oggetto

The *UML Reference Manual* [Rumbaugh 1] definisce uno stato come “una condizione o situazione durante la vita di un oggetto in cui esso soddisfa una qualche condizione, esegue una qualche attività o attende un qualche evento”. Uno stato è una condizione notevole di un oggetto che dipende dai valori dei suoi attributi, dai collegamenti ad altri oggetti, o da quello che sta facendo.

Nei sistemi OO gli oggetti, mentre interagiscono e collaborano tra loro, passano attraverso una successione di stati distinti. Un messaggio può spesso provocare una transizione di stato sull'oggetto che lo riceve. Per esempio, un oggetto *ContoBancario* può trovarsi nei diversi stati *solvibile*, *scoperto* ed *esaurito*, man mano che il denaro viene depositato e prelevato.

Tabella 12.11

Descrizione del flusso relativo alla Figura 12.17	
1: attiva()	Quando la Guardia attiva il sistema con la chiave, il messaggio viene inviato all'oggetto <u>PannelloControlli</u> che rappresenta il thread di controllo principale nel programma.
A1.1: monitoraggio() B1.1: monitoraggio()	:PannelloControlli invia il messaggio monitoraggio() al <u>GestoreSensoriIntrusione</u> e al <u>GestoreSensoriIncendio</u> .
1.2: suona(confermaAttivazione) *[fino-a res = vero] A1.1.1: res := Azionato()	Il sistema emette un segnale con la sirena per confermare che è attivo. Questa è un'iterazione: l'espressione di iterazione afferma che il messaggio Azionato() verrà inviato a turno a ogni oggetto nell'insieme multioggetto, fino a che Azionato() non restituirà il valore vero. In altre parole, il <u>GestoreSensoriIntrusione</u> esamina continuamente ogni <u>SensoreIntrusione</u> , fino all'azionamento di uno di essi.
*[fino-a res = vero] B1.1.1: res := Azionato()	Questa è un'iterazione: l'espressione di iterazione afferma che il messaggio Azionato() verrà inviato a turno a ogni oggetto nell'insieme multioggetto, fino a che Azionato() non restituirà il valore vero. In altre parole, il <u>GestoreSensoriIncendio</u> esamina continuamente ogni <u>SensoreIncendio</u> , fino all'azionamento di uno di essi.
2: aziona()	L'attore Intruso aziona l'oggetto <u>ingresso:SensoreIntrusione</u> . Questo fa in modo che venga assegnato falso al valore restituito res e l'iterazione del passo A1.1.1 si conclude.
A2.1: intruso()	Il <u>GestoreSensoriIntrusione</u> invia il messaggio asincrono intruso() a <u>PannelloControlli</u> .
2.2: suona(allarmeIntrusione)	Il <u>GestoreSensoriIntrusione</u> invia il messaggio suona(allarmeIntrusione) alla <u>Sirena</u> , che suona l'allarme antifurto.

Si possono utilizzare diagrammi di collaborazione in forma istanza per mostrare come gli oggetti effettuino transizioni di stato in risposta ai messaggi.

Lo stato dell'oggetto viene indicato tra parentesi quadre dopo il nome dell'oggetto. Non esiste una notazione per lo stato di un oggetto, e si può quindi utilizzare una qualunque espressione comprensibile e che abbia un senso. Se esiste un diagramma di stato (vedere il Capitolo 19) per la classe dell'oggetto, allora i nomi di stato utilizzati devono essere consistenti con quelli utilizzati in tale diagramma.

Nella Figura 12.18 si vede un oggetto unOrdine:Ordine effettua una transizione dallo stato non-pagato allo stato pagato se, e solo se, il suo attributo saldo diventa esattamente zero. Se il saldo divenisse strettamente positivo, questo significherebbe che è stato pagato del denaro in eccesso di quanto dovuto per l'ordine.

In questo caso l'oggetto unOrdine:Ordine effettua una transizione dallo stato non-pagato allo stato sovrappagato (che non è riportato nel diagramma). Questa transizione di stato potrebbe far scattare un'altra interazione tra oggetti finalizzata all'emissione di una nota di credito.

Nella Figura 12.18, il passo 1.2.1 è una dipendenza. Lo stereotipo «diviene» a essa associato indica che in questa interazione l'oggetto origine *diviene* l'oggetto destinazione al verificarsi del passo 1.2.1.

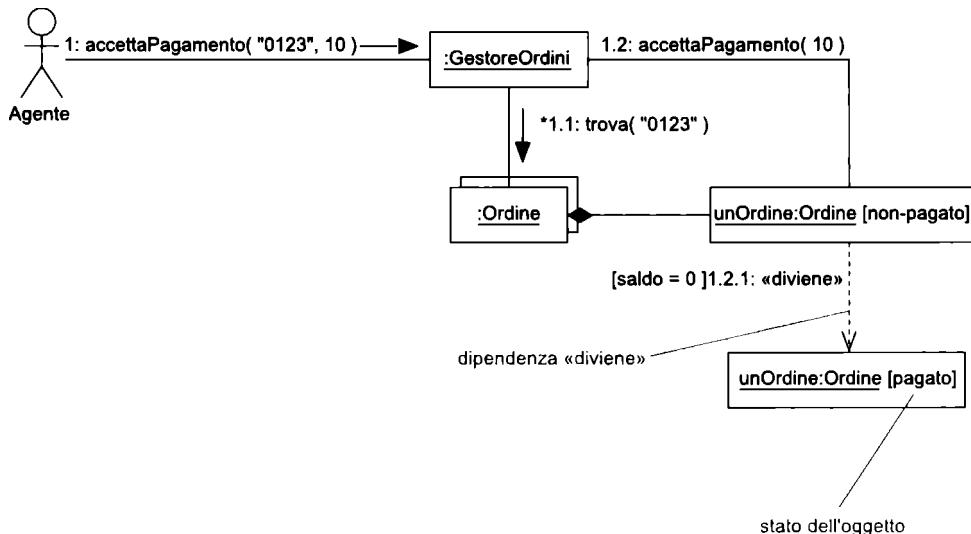


Figura 12.18

12.8 Diagrammi di sequenza

I diagrammi di sequenza mostrano le interazioni tra oggetti ordinate temporalmente. Sono isomorfi ai diagrammi di collaborazione e contengono gli stessi elementi di modellazione, più altri due: le linee di vita degli oggetti e i focus di controllo. Nei diagrammi di collaborazione il focus di controllo viene indicato mediante l’annidamento dei numeri di sequenza, ma nei diagrammi di sequenza si può far vedere il focus di controllo, noto anche come attivazione, in modo molto più chiaro ed esplicito.

Nell’analisi OO, i diagrammi di sequenza servono uno scopo leggermente differente rispetto a quelli di collaborazione. I diagrammi di collaborazione funzionano bene per mostrare gli oggetti e le loro relazioni strutturali (la collaborazione), ma sono più deboli nell’esplicitare le interazioni tra gli oggetti come sequenza di eventi ordinata temporalmente. Per quest’ultimo aspetto i diagrammi di sequenza sono decisamente più appropriati.

Anche i diagrammi di sequenza possono essere modellati in forma istanza o in forma descrittore. Ci si concentra, tuttavia, soltanto sulla forma istanza, che è la forma maggiormente utilizzata dai modellatori.

Nella modellazione, spesso si inizia abbozzando una realizzazione di caso d’uso mediante un diagramma di collaborazione; questo perché è molto facile piazzare degli oggetti sul diagramma e connetterli con dei collegamenti.

Tuttavia, quando occorre focalizzarsi sull’effettiva *sequenza temporale* degli eventi, è sempre molto più facile usare un diagramma di sequenza. Fortunatamente, siccome i due diagrammi sono solo due viste distinte di uno stesso modello

sottostante, gli strumenti CASE riescono spesso a convertire automaticamente un diagramma di collaborazione in un diagramma di sequenza, e viceversa. Questo consente un rapido passaggio tra le due viste, ottenendo così il massimo beneficio da entrambe.

Per studiare i diagrammi di sequenza, useremo alcuni casi d'uso tratti dal nostro semplice sistema di registrazione dei corsi. La Figura 12.19 riporta il caso d'uso AggiungiCorso, già discusso nel Paragrafo 12.7.3. La Figura 12.20 riporta, invece, un diagramma di sequenza per lo stesso caso d'uso.

Sui diagrammi di sequenza, l'asse verticale viene utilizzato per riportare il tempo che scorre sull'asse verticale, dall'alto verso il basso, mentre l'asse orizzontale serve per riportare le istanze (o i ruoli di classificatore), da sinistra verso destra. Le istanze vengono disposte orizzontalmente in modo da minimizzare gli incroci di linee presenti nel diagramma, e verticalmente in base a quando, all'interno dell'interazione, vengono create.

Caso d'uso: AggiungiCorso	
ID:	UC8
Attori:	Registrante
Precondizioni:	Il Registrante è stato autenticato dal sistema.
Sequenza degli eventi:	<ol style="list-style-type: none"> Il Registrante seleziona "aggiungi corso". Il sistema accetta il nome del nuovo corso. Il sistema crea il nuovo corso.
Postcondizioni:	Un nuovo corso è stato aggiunto al sistema.

Figura 12.19

Al di sotto di ogni istanza, verticalmente, si traccia la sua linea di vita tratteggiata; questa rappresenta l'esistenza dell'istanza nel tempo. Per indicare la distruzione di un'istanza, si termina la linea di vita con una grossa croce, come illustrato nella Figura 12.21. Se non si sa, o non interessa sapere, quando un'istanza viene distrutta, basta terminare la sua linea di vita normalmente, senza croce.

Un lungo e sottile rettangolo verticale, sovrapposto alla linea di vita, indica che l'oggetto ha il focus di controllo. Nella Figura 12.20, l'attore Responsabile inizia con il focus di controllo. Invia il messaggio AggiungiCorso("UML") a GestoreRegistrazioni, il quale esegue la propria operazione AggiungiCorso(...) con il parametro "UML". Durante l'esecuzione di questa operazione, GestoreRegistrazioni ha il focus di controllo.

Si osservi però, che questo focus di controllo è *annidato* all'interno di quello dell'attore Responsabile. Questo è corretto: un oggetto inizia con il focus di controllo e chiama un'operazione di un altro oggetto passandogli il focus di controllo annidato.

A sua volta, questi può chiamare un'altra operazione di un altro oggetto ancora, annidando ulteriormente il focus di controllo, e così via.

Durante l'esecuzione della operazione `AggiungiCorso(...)` il `:GestoreRegistrazioni` crea un nuovo oggetto `UML:Corso`. Si mostra la creazione inviando un messaggio con stereotipo (come nella Figura 12.20), oppure mediante un messaggio specifico, dotato di nome, anch'esso stereotipato con «crea». In C++, C# o Java, la creazione è un'operazione speciale, detta costruttore, che ha lo stesso nome della classe dell'oggetto, nessun valore restituito, e zero o più parametri. In questo esempio, se si usasse Java, si potrebbe inviare il messaggio: «crea» `Corso("UML")` che chiamerebbe il costruttore `Corso` della classe. Tuttavia, non tutti i linguaggi OO prevedono i costruttori: in Smalltalk, per esempio, si potrebbe mandare il messaggio «crea» `init: "UML"`.

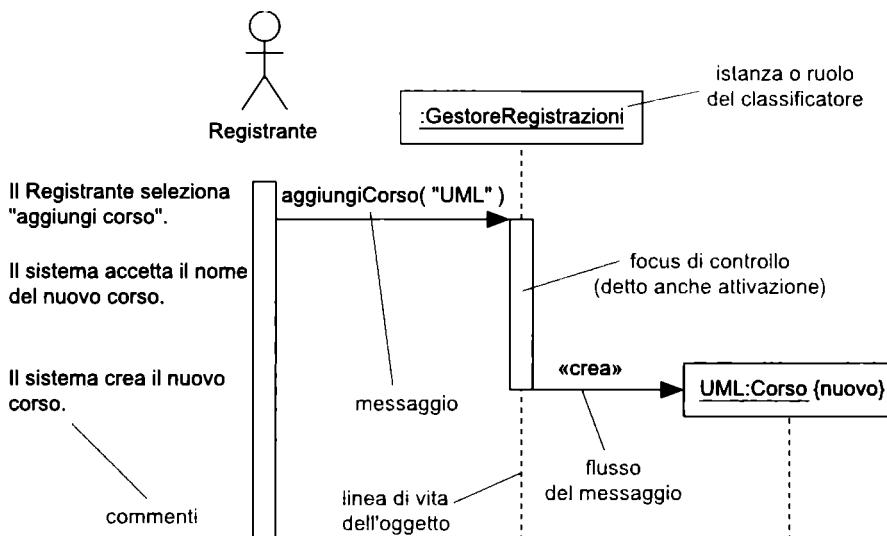


Figura 12.20

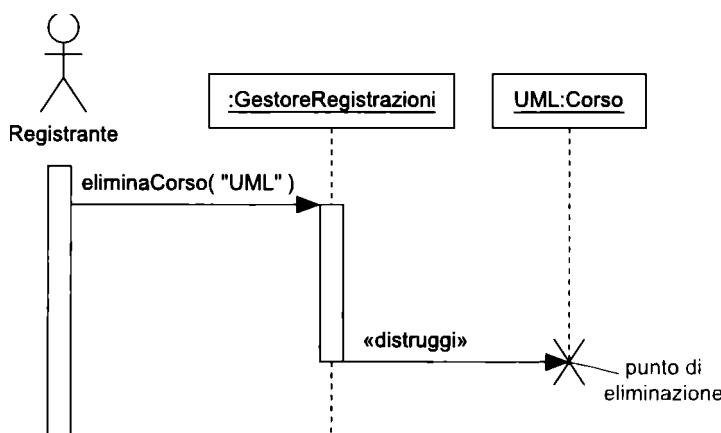


Figura 12.21

Nei diagrammi precedenti non ci si è preoccupati di mostrare la terminazione delle operazioni; si presuppone che ciascuna operazione, al termine della propria esecuzione, restituiscia il focus di controllo al proprio chiamante.

Tuttavia è possibile (e spesso desiderabile) mostrare esplicitamente il ritorno dalle operazioni; lo si può indicare nello stesso modo in cui lo si indica sui diagrammi di collaborazione: con una freccia tratteggiata che torna dal ricevente al mittente del messaggio. Se ne può vedere un esempio nella Figura 12.24.

Un aspetto molto apprezzato dei diagrammi di sequenza è la possibilità di riportare una sceneggiatura del diagramma tramite commenti aggiunti lungo il lato sinistro. Questo rende il diagramma molto più accessibile alle parti interessate che avessero minori capacità tecniche, come molti utenti ed esperti del dominio. La sceneggiatura può comprendere i passi della sequenza degli eventi del caso d'uso (come nella Figura 12.20), o può anche essere solo una descrizione testuale di ciò che sta succedendo nel diagramma.

12.8.1 Iterazione

In un diagramma di sequenza l'iterazione viene modellata includendo in un unico rettangolo i messaggi che devono essere ripetuti e riportando l'espressione di iterazione sotto di esso. La Figura 12.22 ne illustra un esempio.

L'UML 1.4 al momento non supporta la modellazione di multioggetti nei diagrammi di sequenza. Questo rende difficile visualizzare chiaramente un'iterazione sugli elementi di un insieme multioggetto. La Figura 12.22 è in realtà il diagramma di sequenza della corrispondente allo stesso modello da cui è tratto il diagramma di collaborazione della Figura 12.23.

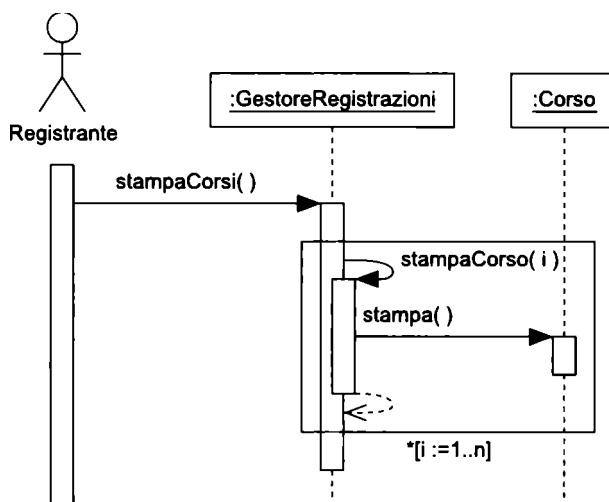


Figura 12.22

Sebbene la versione come diagramma si sequenza risulti abbastanza chiara e comprensibile, si osservi come, nel diagramma di collaborazione, la presenza del qualificatore e del multioggetto consenta di specificare il meccanismo di iterazione con una precisione decisamente superiore. Questi meccanismi non sono disponibili nei diagrammi di sequenza.

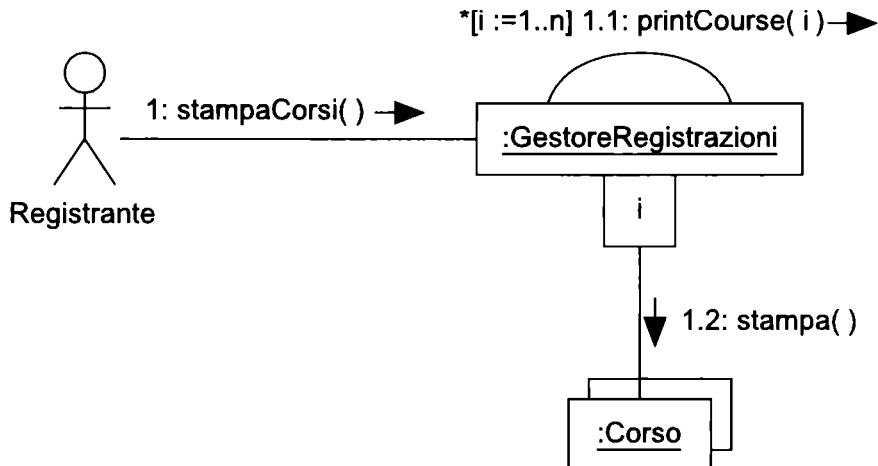


Figura 12.23

12.8.2 Ramificazione e auto-delegazione

I diagrammi di sequenza illustrano la ramificazione in modo molto più leggibile che non i diagrammi di collaborazione. Questo perché i primi si concentrano sull'aspetto temporale della sequenza degli eventi, mentre i secondi si occupano soprattutto delle istanze e delle relazioni.

La Figura 12.24 riporta il diagramma di sequenza corrispondente al diagramma di collaborazione visto nella Figura 12.12. Si osservi come vi siano modellate la ramificazione, l'auto-delegazione e il focus di controllo annidato.

Per modellare la ramificazione, si spezza il focus di controllo di un oggetto in due o più parti, quindi si associa una condizione di guardia (mutuamente esclusiva) al primo messaggio di ciascun ramo. Se non si usano condizioni di guardia mutuamente esclusive su ciascun ramo, allora si ottiene un'esecuzione concorrente e non una ramificazione.

12.8.3 Concorrenza: oggetti attivi

Si esprime la concorrenza mediante gli oggetti attivi (proprio come già visto per i diagrammi di collaborazione), ma occorre anche un metodo per visualizzare i *thread* in esecuzione parallela. Questo si indica con una biforcazione del focus di controllo.

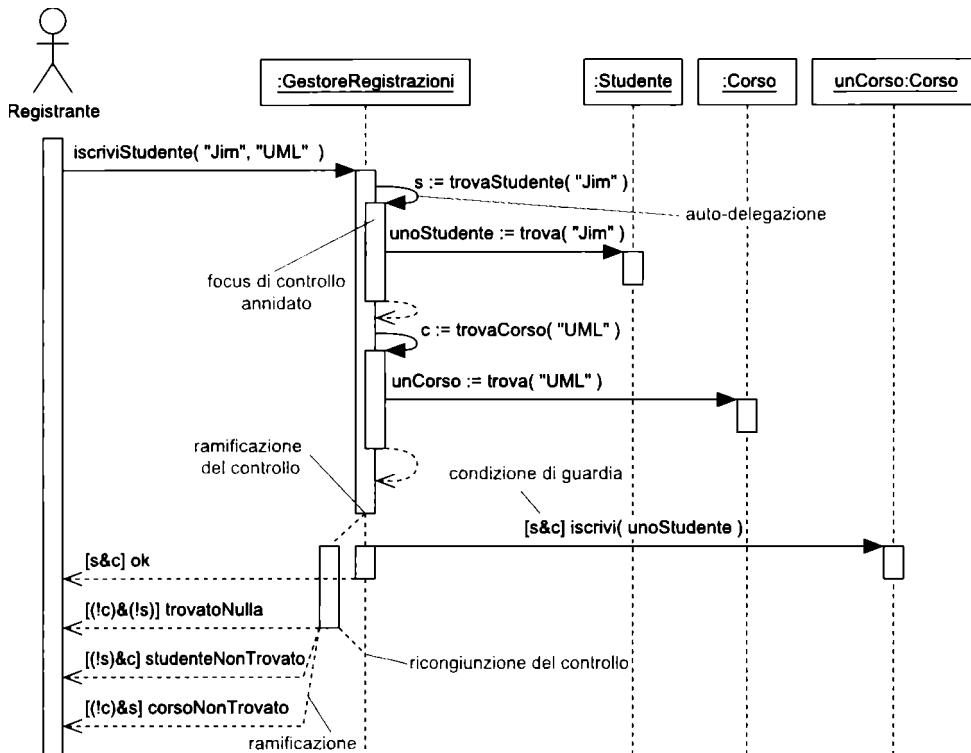


Figura 12.24

Nella Figura 12.25 (il diagramma di sequenza corrispondente al diagramma di collaborazione della Figura 12.17) si vede che quando termina uno dei *thread*, si ritorna allo stesso punto del focus di controllo originale di **:PannelloControlli** e quindi si invia il messaggio `suona(...)` alla **:Sirena**.

12.8.4 Stato di un oggetto e vincoli

I diagrammi di sequenza possono illustrare come un oggetto cambia stato nel tempo. Si collocano i diversi stati in punti appropriati della linea di vita dell'oggetto. Questo consente di visualizzare chiaramente quali eventi provocano una transizione di stato.

La Figura 12.26 mostra un esempio. Il messaggio `riceviPagamento(...)` provoca una transizione di stato dell'oggetto **:Ordine** dallo stato `nonPagato` allo stato `pagato`. Similmente, il messaggio `consegna()`, inviato dall'oggetto **:GestoreConsegne** a **:Ordine**, provoca una transizione dell'oggetto dallo stato `pagato` allo stato `consegnato`.

La Figura 12.26 mostra anche come sia possibile etichettare posizioni sull'asse temporale al fine di esprimere vincoli temporali.

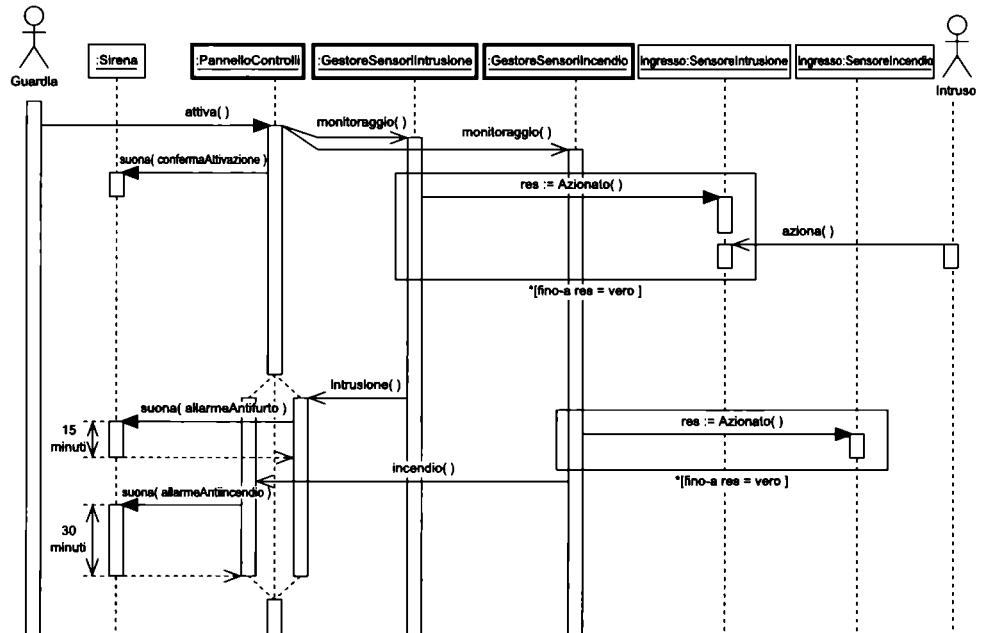


Figura 12.25

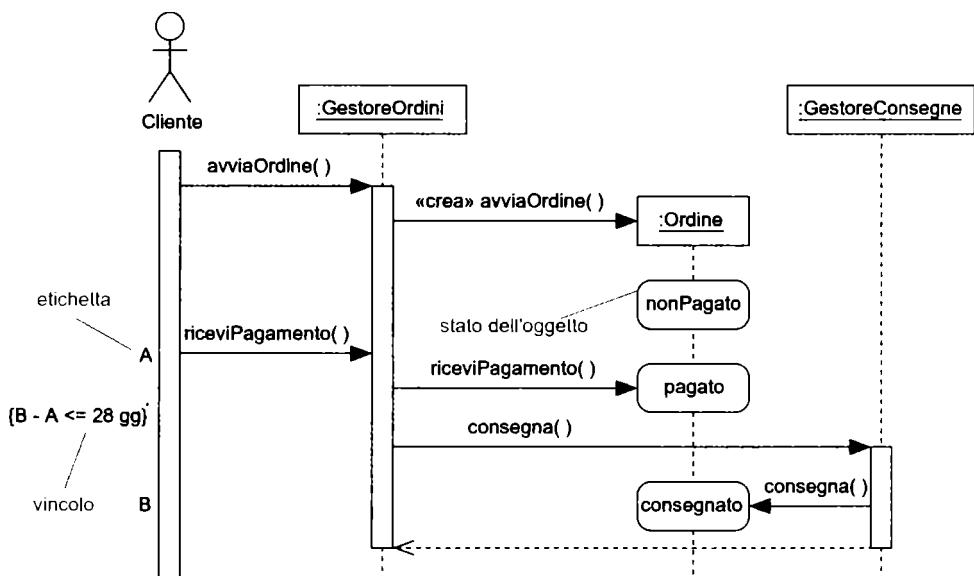


Figura 12.26

Si tratta effettivamente di una sintassi molto flessibile che consente di inserire sulla parte sinistra di un diagramma di sequenza, non solo vincoli temporali, ma anche qualunque altro tipo di vincolo.

Il vincolo temporale nella Figura 12.26 afferma che non devono intercorrere più di 28 giorni tra l'istante A e l'istante B. L'istante A indica la transizione dell'oggetto Ordine allo stato pagato, l'istante B indica la sua transizione allo stato consegnato. Di conseguenza, il diagramma si sequenza specifica che, nella parte sistema relativa alla gestione degli ordini, non devono passare più di 28 giorni tra il pagamento e la consegna di un ordine.

12.9 Riepilogo

Le realizzazioni di caso d'uso sono una parte essenziale del processo di analisi. Esse consentono di verificare la corrispondenza delle specifiche alla realtà implementabile, dimostrando come gli oggetti delle classi di analisi individuate possano interagire tra loro per produrre il comportamento richiesto al sistema. I diagrammi di interazione mostrano come classi e oggetti realizzano i requisiti specificati nei casi d'uso.

In questo capitolo sono stati appresi i seguenti concetti.

1. Le realizzazioni di caso d'uso si creano durante l'attività UP “Analizzare un caso d'uso”; questa attività produce una parte della vista dinamica del sistema.
2. Le realizzazioni di caso d'uso mostrano come le istanze delle classi di analisi possono realizzare i requisiti funzionali specificati nei casi d'uso.
 - Ogni realizzazione di caso d'uso realizza un unico caso d'uso.
 - Le realizzazioni di caso d'uso sono costituite da:
 - diagrammi delle classi di analisi: dovrebbero “raccontare una storia” che riguarda uno o più casi d'uso;
 - diagrammi di interazione: dovrebbero dimostrare come gli oggetti interagiscono per realizzare il comportamento specificato nel caso d'uso;
 - requisiti aggiuntivi: se ne scoprono sempre di nuovi durante la realizzazione di caso d'uso e devono essere registrati;
 - rifinitura dei casi d'uso: potrebbe essere necessario modificare un caso d'uso esistente, mentre lo si realizza.
3. I diagrammi di interazione possono essere in forma descrittore o in forma istanza.
 - I diagrammi d'interazione in forma descrittore contengono:
 - ruoli di classificatore: ruoli che verranno interpretati dalle istanze dei classificatori;
 - ruoli di associazione: ruoli che vengono interpretati dalle associazioni;
 - messaggi e flusso dei messaggi.

- I diagrammi di interazione in forma istanza contengono:
 - istanze di classificatori (oggetti);
 - collegamenti;
 - si possono mostrare la creazione e distruzione di istanze e collegamenti:
 - {nuovo} : l'istanza o collegamento è creato durante l'interazione;
 - {distrutto} : l'istanza o collegamento è distrutto durante l'interazione;
 - {transiente} : l'istanza o collegamento è creato e distrutto durante l'interazione;
 - messaggi e flusso dei messaggi;
 - si visualizza il flusso di un messaggio con una freccia; ve ne sono tre tipi:
 - chiamata di procedura: il mittente aspetta che la chiamata termini l'esecuzione;
 - comunicazione asincrona: il mittente non attende che la chiamata termini l'esecuzione;
 - rientro da chiamata di procedura: il rientro è sempre implicito, quindi si può omettere di mostrarlo.
 - iterazione;
 - ramificazione.

4. Esistono due tipi isomorfi di diagrammi:

- diagrammi di collaborazione;
- diagrammi di sequenza.

5. I diagrammi di collaborazione enfatizzano le collaborazioni tra gli oggetti.

- Una collaborazione descrive un insieme statico di relazioni tra istanze e i ruoli che tali istanze svolgono nelle relazioni.
- I diagrammi di collaborazione possono contenere:
 - multioggetti; rappresentano un insieme di oggetti:
 - i qualificatori consentono di selezionare un oggetto specifico dal multioggetto usando un identificatore univoco;
 - si può presumere che un multioggetto metta implicitamente a disposizione almeno questi metodi:
 - `trova(identificatoreUnivoco)`: restituisce un oggetto specifico, corrispondente all'identificatore univoco passato come parametro;
 - `include(unOggetto)`: restituisce vero se il multioggetto contiene `unOggetto`;
 - `conteggio()`: restituisce il numero di elementi contenuti nel multioggetto;

- iterazione:
 - un messaggio inviato a un multioggetto, e preceduto dall'indicatore di iterazione, viene inoltrato a ogni elemento dell'insieme:
 - *: elaborazione sequenziale degli elementi contenuti nel multioggetto;
 - *//: elaborazione parallela degli elementi contenuti nel multioggetto;
 - dopo l'indicatore iterazione si può aggiungere un'espressione di iterazione per specificare il numero di iterazioni:
 - nulla; si itera sull'intero insieme;
 - [i := 1..n];
 - [fintantoché (espressione Booleana)];
 - [fino-a (espressione Booleana)];
 - [per-ogni (espressione che risulta in un insieme di oggetti)];
- ramificazione: si associa un'espressione Booleana al messaggio, il quale viene inviato solo se la condizione risulta vera;
- concorrenza, ogni oggetto attivo ha il suo *thread* di controllo:
 - gli oggetti attivi sono istanze di classi attive;
 - gli oggetti e le classi attive si disegnano con i bordi spessi, oppure includendo la proprietà {attivo};
 - si assegna un nome a ogni *thread*, per esempio, A e B; si aggiunge il nome del *thread* al numero di sequenza, per esempio, A1.1 e B1.1.
- lo stato di un oggetto può essere indicato tra parentesi quadre, dopo il nome della classe:
 - questo ci indica un oggetto con stato: un oggetto che si trova in uno stato specifico;
 - si usa la dipendenza «diviene» tra due oggetti con stato per mostrare come l'oggetto cambi stato nel tempo.

6. I diagrammi di sequenza enfatizzano le interazioni tra istanze.

- Un'interazione descrive la sequenza temporale di messaggi scambiati tra istanze.
- Il tempo viene indicato dall'alto verso il basso, i ruoli di classificatore, o le istanze, vengono riportati da sinistra verso destra.
- Possono contenere tutti gli elementi di modellazione validi per i diagrammi di collaborazione:
 - eccetto i multioggetti e i qualificatori;
 - più una sceneggiatura: magari la sequenza degli eventi del caso d'uso, lungo il lato sinistro del diagramma;

- più la linea di vita degli oggetti: una linea verticale tratteggiata che ne rappresenta l'esistenza nel tempo:
 - si può indicare il punto di creazione di un oggetto inviando un messaggio «crea» alla sua icona;
 - si può indicare il punto di distruzione di un oggetto inviando un messaggio «distruggi» alla grossa X su cui termina la linea di vita dell'oggetto;
- più il focus di controllo: un rettangolo sottile e allungato posizionato sulla linea di vita che indica quando l'oggetto è attivo.
- Iterazione:
 - si disegna un rettangolo attorno ai messaggi che devono essere ripetuti;
 - si indica l'espressione di iterazione appena sotto questo rettangolo.
- Ramificazione; la linea di vita dell'oggetto si divide in rami alternativi:
 - proteggere ogni ramo con una condizione di guardia sul primo messaggio inviato;
 - tutte le condizioni di guardia devono essere mutuamente esclusive; altrimenti si ottiene una concorrenza, non una ramificazione.
- Concorrenza:
 - si usano gli oggetti attivi;
 - si creano biforcazioni e ricongiunzioni nel focus di controllo.
- Si visualizza lo stato di un oggetto inserendo icone di stato in punti appropriati della linea di vita.
- Si possono mettere vincoli ovunque nel diagramma, ma di solito si tengono sul lato sinistro.

Diagrammi di attività

13.1 Contenuto del capitolo

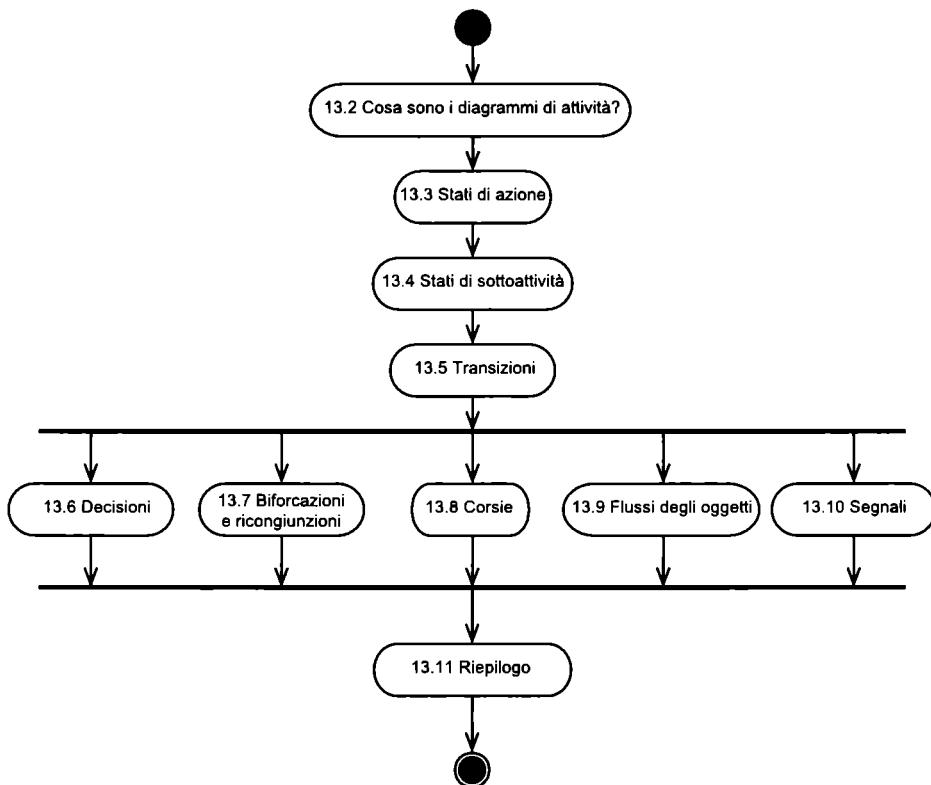
A chi non sa cosa sia un diagramma di attività si consiglia di leggere tutte le sezioni di questo capitolo, in ordine. Chi già conosce i diagrammi di attività può saltare la maggior parte del capitolo, anche se potrebbero, comunque, risultare interessanti il Paragrafo 13.9 (flussi degli oggetti) e il Paragrafo 13.10 (Segnali).

13.2 Cosa sono i diagrammi di attività?

I diagrammi di attività sono “diagrammi di flusso OO”. Consentono di modellare un processo come un insieme di attività e di transizioni tra queste attività. I diagrammi di attività sono in realtà soltanto dei tipi speciali di diagrammi di stato (vedere il Capitolo 19) in cui ogni stato ha un’azione di ingresso che specifica una procedura o funzione da eseguire quando si entra nello stato.

È possibile associare un diagramma di attività a *qualunque* elemento di modellazione, al solo fine di modellare il comportamento di tale elemento. I diagrammi di attività vengono tipicamente associati a:

- casi d’uso;
- classi;
- interfacce;
- componenti;
- nodi;
- collaborazioni;
- operazioni e metodi.

**Figura 13.1**

È, inoltre, possibile utilizzare i diagrammi di attività per modellare efficacemente i processi e i flussi di lavoro del *business*.

Anche se spesso i diagrammi di attività vengono utilizzati per illustrare il flusso di singole operazioni, è forse opportuno osservare che la rappresentazione migliore e più concisa di un'operazione è pur sempre il suo stesso codice sorgente! È, quindi, consigliabile usare giudizio.

L'essenza di un buon diagramma di attività è quella di comunicare uno specifico aspetto del comportamento dinamico di un sistema. A tal fine, è necessario che il suo livello di astrazione sia quello più adatto a comunicare il suo intento ai lettori. È molto importante che il diagramma contenga la quantità di informazioni minima indispensabile per comunicare il proprio intento. È piuttosto facile decorare i diagrammi di attività con stati e flussi degli oggetti o altri elementi, ma è sempre opportuno chiedersi se tali ornamenti aumentino o diminuiscano la comprensibilità del diagramma.

13.3 Stati di azione

I diagrammi di attività contengono stati di azione e stati di sottoattività. Gli stati di azione sono gli elementi costitutivi minimi dei diagrammi di attività e rappresentano azioni: attività che non possono essere scomposte in sottoattività.

Gli stati di azione vengono rappresentati come rettangoli coi lati arrotondati, come illustrato nella Figura 13.2.



Figura 13.2

L'espressione di azione indica l'azione di ingresso dello stato, ovvero la parte di lavoro che verrà eseguito in seguito all'ingresso nello stato di azione. Può essere scritta in pseudo-codice o in linguaggio naturale. Se è scritta in linguaggio naturale dovrebbe avere la forma di un verbo o di una frase con verbo, in quanto gli stati di azione fanno sempre qualcosa. Le azioni sono:

1. atomiche: non possono essere scomposte in pezzi più piccoli;
2. non interrompibili: quando ne inizia l'esecuzione si procede sempre fino al termine della stessa;
3. istantanee: in linea generale, si considera che il lavoro eseguito da uno stato di azione richieda una quantità di tempo trascurabile.

Secondo le specifiche dell'UML (versione 1.4), uno stato di azione è solo una versione semplificata di un elemento più generale che viene chiamato stato. In effetti, lo stato di azione è solo una forma abbreviata per indicare uno stato che ha un'azione di ingresso e almeno una transizione in uscita. La sintassi visuale per gli stati di azione è diversa da quella per gli stati: gli stati di azione vengono disegnati come rettangoli coi lati arrotondati, mentre gli stati vengono disegnati come rettangoli con gli angoli arrotondati ("rettangoli arrotondati"). Gli stati vengono trattati nel Capitolo 19.



Figura 13.3

L'UML fornisce una sintassi visuale specifica e una semantica semplificata per gli stati di azione. Questo perché, come vedremo più avanti, gli stati hanno, invece, una sintassi e una semantica molto ricca e complessa, sicuramente eccessiva per quel che serve alla modellazione dei diagrammi di attività.

Ogni diagramma di attività ha due stati speciali: lo stato iniziale e lo stato finale. Lo stato iniziale indica l'inizio del flusso di lavoro, mentre lo stato finale ne indica la fine. La Figura 13.3 illustra i simboli utilizzati per questi stati speciali.

13.4 Stati di sottoattività

Gli stati di sottoattività non sono atomici: possono essere scomposti in altri stati di sottoattività o stati di azione. Possono essere interrotti, e la loro esecuzione può richiedere una quantità di tempo non trascurabile. La Figura 13.4 illustra la sintassi visuale per gli stati di sottoattività.

Ogni stato di sottoattività richiama un intero grafo di attività che vi si trova annidato dentro. Questo grafo di attività annidato può contenere altri stati di azione o di sottoattività. In molti strumenti CASE UML, cliccando una o due volte su uno stato di sottoattività, si può aprire il diagramma di attività relativo al grafo di attività annidato.

Gli stati di sottoattività vengono solitamente utilizzati per modellare complessi processi di *business* sotto forma di flussi di lavoro. Ogni elemento del flusso di lavoro può essere modellato come uno stato di sottoattività, scomponibile in stati di sottoattività sempre più piccoli, fino a giungere agli stati di azione.

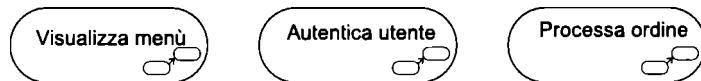


Figura 13.4

13.5 Transizioni

Quando uno stato di azione o di sottoattività ha terminato il proprio lavoro avviene una transizione in uscita dallo stato e un passaggio allo stato successivo. Questa viene detta transizione automatica. Nel Paragrafo 19.6, dove si discute dei diagrammi di stato UML, si vedrà che una transizione automatica non è altro che un transizione di stato che non è attivata da nessun evento particolare; in altre parole, succede semplicemente e automaticamente quando uno stato ha terminato il proprio lavoro.

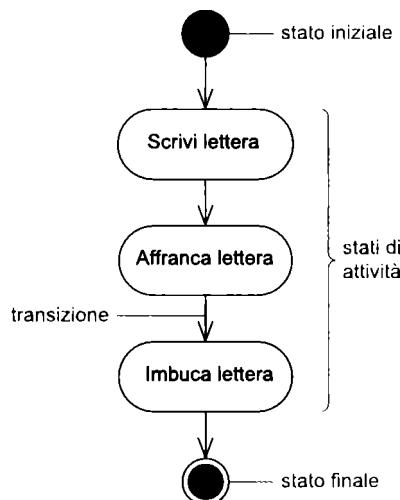


Figura 13.5

Nell'esempio di diagramma di attività illustrato nella Figura 13.5, si inizia (come d'obbligo) dallo stato iniziale e avviene subito una transizione automatica al primo stato di azione *Scrivi lettera*. Quando l'azione specificata è terminata, avviene una transizione automatica al successivo stato di azione. Questo tipo di processo continua fin quando non si raggiunge lo stato finale.

13.6 Decisioni

Una decisione specifica percorsi alternativi basati su qualche condizione di guardia Booleana. Il simbolo UML per la decisione è il rombo (Figura 13.6), proprio come in un normale diagramma di flusso. Lo stesso simbolo viene anche utilizzato per la fusione, laddove i due percorsi alternativi si riuniscono.

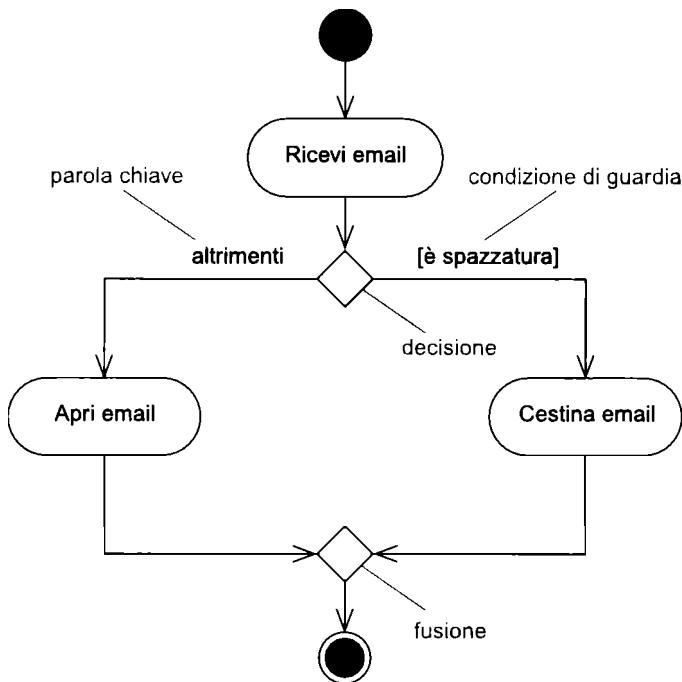


Figura 13.6

Una decisione può avere un unico percorso in ingresso, ma può avere molti percorsi in uscita. Ogni percorso in uscita da una decisione è protetto da una condizione di guardia. Il percorso può essere seguito se, e solo se, la condizione di guardia risulta vera. La parola chiave *altrimenti* indica il percorso che verrà preso se nessuna delle condizioni di guardia risultasse vera. Le condizioni di guardia devono essere formulate con cura, in modo che risultino mutuamente esclusive; in caso contrario, il diagramma di attività ne risulterebbe indeterminato!

Nell'esempio della Figura 13.6, avviene una transizione dallo stato iniziale allo stato di azione *Ricevi email*. Quando questo stato ha terminato il proprio lavoro, avviene una transizione alla decisione. Questa decisione ha due possibili transizioni in uscita. Se la email è posta-spazzatura allora [è spazzatura] risulta vero e, quindi, si attiva la transizione di destra e si raggiunge lo stato *Cestina email*, e in seguito si procede allo stato finale. In caso contrario, viene attivata la transizione di sinistra e si raggiunge lo stato *Apri email*, e in seguito si procede allo stato finale.

13.7 Biforazioni e ricongiunzioni

I diagrammi di attività sono ottimi per modellare flussi di lavoro concorrenti. È possibile spaccare un percorso in due o più flussi concorrenti utilizzando una biforcazione, quindi risincronizzare questi flussi concorrenti utilizzando una ricongiunzione.

Le biforcazioni hanno un'unica transizione in ingresso e due o più transizioni in uscita. Le ricongiunzioni hanno due o più transizioni in ingresso e un'unica transizione in uscita. La transizione in uscita può essere attivata soltanto quando *tutte* le transizioni in ingresso sono state attivate, ovvero solo quando tutti i flussi concorrenti di lavoro sono terminati.

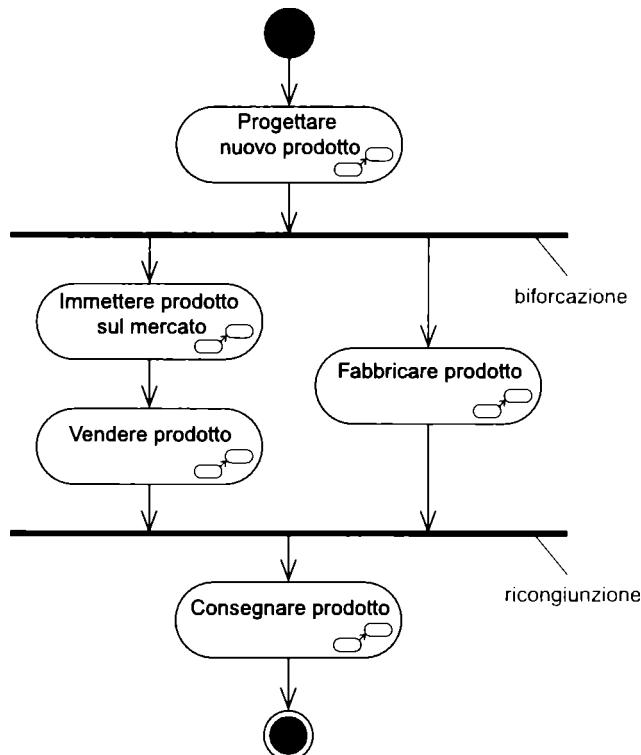


Figura 13.7

Dato che si può uscire da una ricongiunzione e passare allo stato successivo soltanto quando tutte le transizioni in ingresso sono state attivate, nel diagramma di attività la ricongiunzione è il punto in cui i flussi concorrenti vengono risincronizzati. Questi flussi concorrenti possono rappresentare flussi di lavoro concorrenti, processi di sistema concorrenti, *thread* concorrenti che vengono eseguiti in aree di memoria riservate o *thread* concorrenti dello stesso processo che vengono eseguiti in un'area di memoria condivisa.

La Figura 13.7 illustra un modello molto semplice con flussi di lavoro concorrenti. Questo diagramma di attività modella (a un livello di astrazione piuttosto alto) il processo di *business* richiesto per immettere sul mercato un nuovo prodotto. Ogni stato di questo diagramma di alto livello è uno stato di sottoattività che, a sua volta, contiene un grafo di attività annidato.

Si comincia dallo stato iniziale e avviene subito una transizione allo stato di sottoattività **Progettare nuovo prodotto**. Quando il prodotto è stato progettato, avviene una transizione automatica alla biforcazione dalla quale hanno origine due flussi di lavoro concorrenti, uno che riguarda il marketing e uno relativo alla fabbricazione. Nel flusso di lavoro del marketing si entra nello stato **Immettere prodotto sul mercato** in cui viene effettuata una campagna di marketing per il prodotto. Quindi si passa allo stato **Vendere prodotto** in cui si iniziano a prendere gli ordini per il prodotto. Parallelamente, nel flusso di lavoro della fabbricazione, si entra nello stato **Fabbricazione prodotto**. Quando si concludono entrambi i flussi di lavoro del marketing e della fabbricazione, avviene una transizione in uscita dalla ricongiunzione e si passa allo stato **Consegna prodotto**. Questo modello dice che, anche se è possibile fare del marketing e vendere un prodotto nuovo, non è possibile comunque consegnarlo veramente fin quando tale prodotto non sia stato anche fabbricato.

13.8 Corsie

L'UML non fornisce una semantica specifica per le corsie. Possono, quindi, essere utilizzate per partizionare i diagrammi di attività nel modo preferito! Le corsie vengono comunemente utilizzate per rappresentare:

- casi d'uso;
- classi;
- componenti;
- unità organizzative (nella modellazione di *business*);
- ruoli (nella modellazione dei flussi di lavoro).

Tuttavia non si è in alcun modo limitati a queste possibilità. Per esempio, nei modelli di progettazione relativi a sistemi distribuiti si possono utilizzare le corsie per modellare la distribuzione dei processi sulle macchine fisiche.

Spesso esiste una correlazione tra le corsie e i flussi di controllo concorrenti. Nell'esempio della Figura 13.8, le corsie rappresentano unità organizzative del *business*:

l'ufficio marketing, il reparto di produzione e il reparto della distribuzione. Il diagramma di attività mostra che l'ufficio marketing può portare avanti diverse sue attività, mentre il reparto di produzione si occupa della fabbricazione del prodotto. Il reparto della distribuzione, invece, dipende dagli altri due e deve aspettare che essi abbiano terminato il proprio lavoro. Capita spesso che reparti o unità organizzative diversi possano eseguire alcune attività in parallelo, ma che debbano poi essere sincronizzati in qualche punto del flusso. Le corsie dei diagrammi di attività consentono di modellare queste situazioni in modo ottimale.

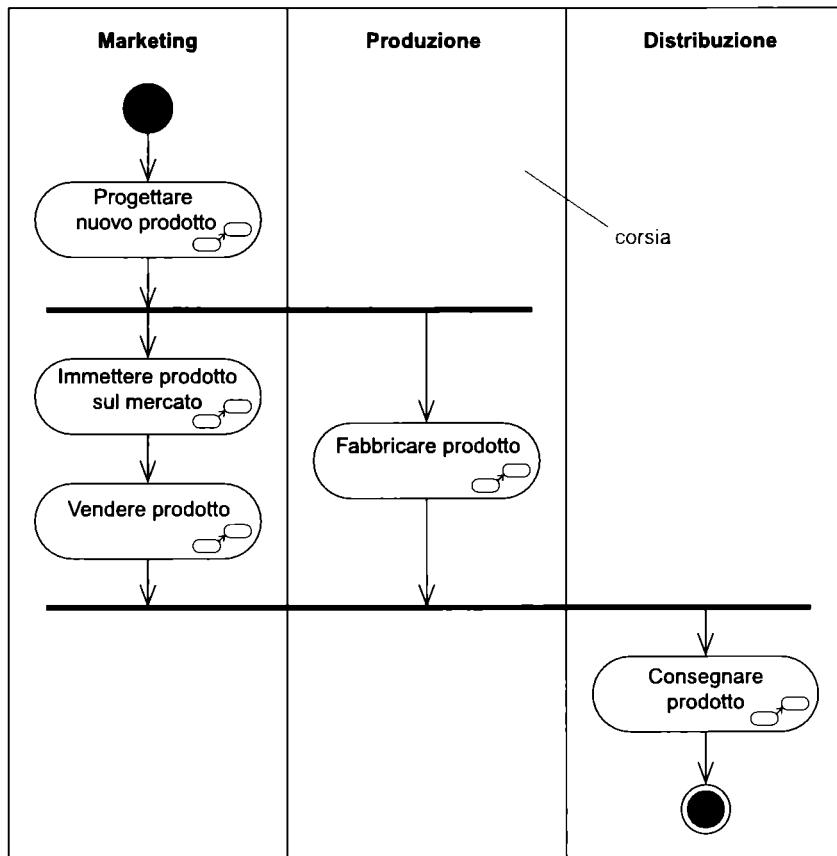


Figura 13.8 .

13.9 Flussi degli oggetti

Le attività possono avere oggetti in input e in output e possono modificarne lo stato. È possibile mostrare questo comportamento su un diagramma di attività tramite i flussi degli oggetti. Dato che in un sistema OO le attività possono interagire con molti oggetti, è consigliabile riportare sul flusso degli oggetti solo gli oggetti importanti.

Nella Figura 13.9 ci sono due oggetti importanti: l'oggetto `:SpecificheProdotto` che viene generato dallo stato di sottoattività `Progettare nuovo prodotto`, e che va in input allo stato di sottoattività `Fabbricazione prodotto`, e l'oggetto `:Ordine` che consente di fissare i dettagli relativi a un pagamento e una consegna e che fornisce informazioni storiche sul processo di vendita.

È possibile indicare lo stato dell'oggetto, via via che questo attraversa le diverse attività del flusso, scrivendo il nome dello stato, tra parentesi quadre, sotto il nome dell'oggetto. Il nome dello stato riportato deve corrispondere a uno stato definito nel diagramma di stato della classe dell'oggetto (i diagrammi di stato vengono trattati in dettagli nei Capitoli 19 e 20).

Un'altra tecnica per indicare le modifiche subite dallo stato dell'oggetto è quella di riportare i valori effettivi dei suoi attributi. In effetti, questo è l'unico modo per illustrare cambiamenti di stato nel caso che non esista un diagramma di stato per la classe dell'oggetto. I valori degli attributi possono essere riportati in un'apposita sottosezione sotto il nome dell'oggetto.

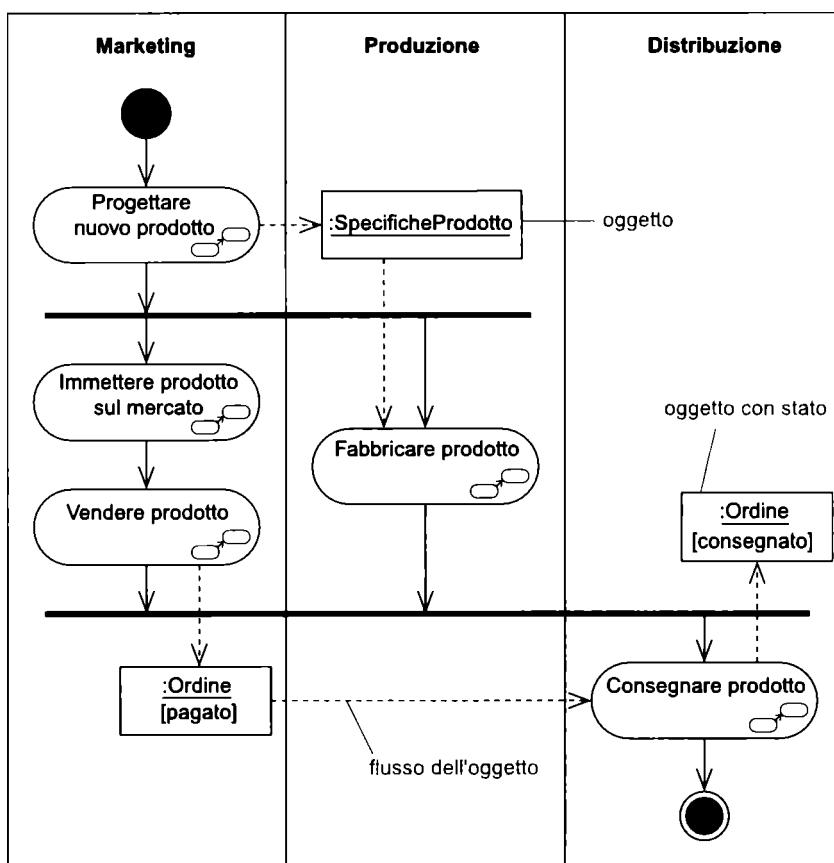


Figura 13.9

13.10 Segnali

Il segnale consente di rappresentare un pacchetto di informazioni che viene comunicato in modo asincrono tra due oggetti. Un evento di segnale avviene quando un oggetto riceve un segnale.

L'evento è un qualcosa che succede. La ricezione di un evento può provocare una transizione di stato. Fatta eccezione per gli eventi di segnale, di solito non si riportano eventi sui diagrammi di attività, ma solo sui diagrammi di stato. Gli eventi verranno approfonditi nel Paragrafo 19.7.

L'invio di segnale viene modellato sul diagramma di attività con un pentagono convesso etichettato con il nome del segnale che viene inviato. L'invio di segnale ha una transizione in ingresso e una transizione in uscita. Questo stato non è associato ad alcuna azione; tutto ciò che fa è inviare il segnale. Si può, ma non è obbligatorio, disegnare una freccia tratteggiata tra l'invio di segnale e l'oggetto che riceve il segnale.

Nell'esempio della Figura 13.10, il segnale Ordine viene inviato all'oggetto esterno :SocietaVenditaCorrispondenza.

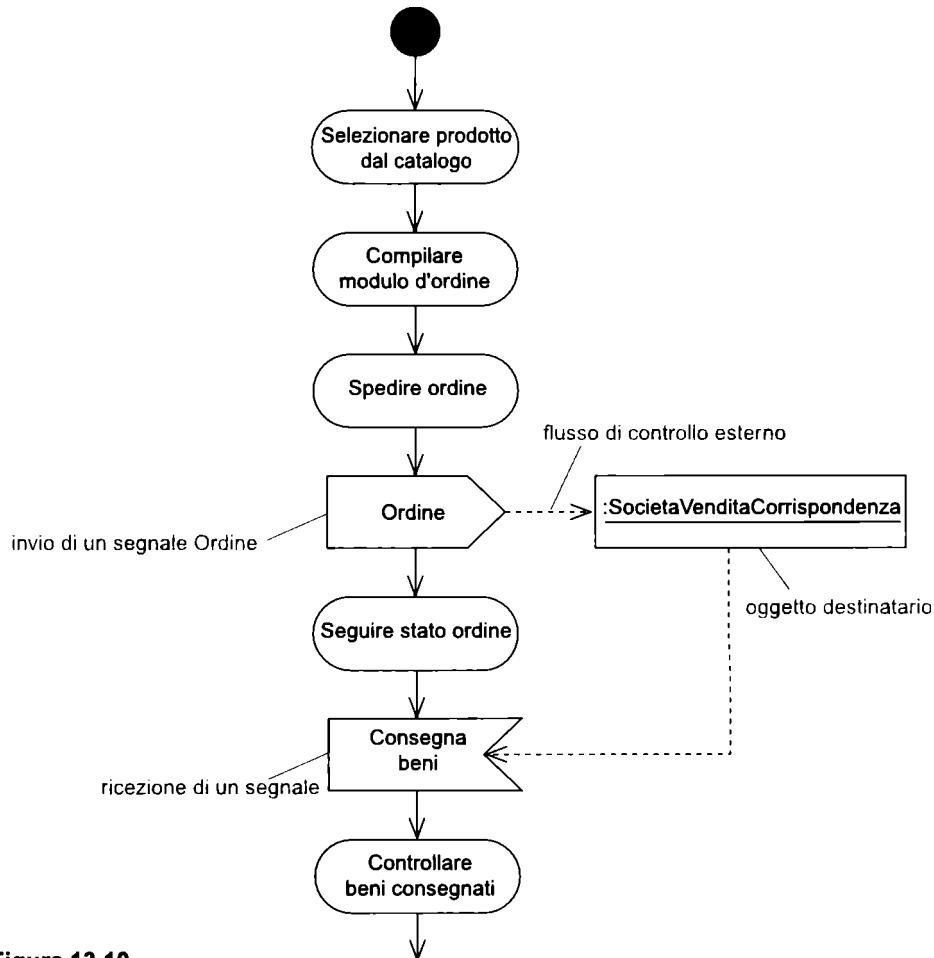


Figura 13.10

La ricezione di segnale viene modellata come un pentagono concavo etichettato con il nome del segnale ricevuto. La ricezione di segnale è uno stato con nessuna azione associata: serve solo per ricevere il segnale. Nella Figura 13.10 si resta in attesa per la ricezione di un segnale Consegna beni, il quale ci viene inviato dall'oggetto esterno :SocietaVenditeCorrispondenza.

I segnali sono classificatori e possono essere modellati come classi stereotipate. I segnali servono per consentire la comunicazione asincrona di *informazioni* tra oggetti diversi e, quindi, a differenza delle classi normali, hanno *soltanto* attributi e un'unica operazione隐式的 *invia(elencoDestinatari)*, la quale consente l'invio del segnale a un insieme di oggetti destinatari. Dato che questa operazione è implicita, non viene mai indicata sui diagrammi di attività: si presume che ciascun segnale abbia questa operazione. Gli attributi del segnale definiscono il contenuto informativo del segnale.

L'esempio della Figura 13.11 mostra la gerarchia di generalizzazione di eventi GUI di basso livello, modellati come segnali.

Dato che i segnali hanno solo attributi, ma nessuna operazione di interesse, sono in qualche modo simili a tipi di dati più tradizionali e possono esclusivamente servire per modellare i dati che vengono passati da un oggetto a un altro. Questo significa che in realtà non sono molto utili per modellare sistemi OO distribuiti. Tuttavia possono, comunque, risultare validi per modellare le informazioni scambiate tra un sistema OO e un sistema non OO.

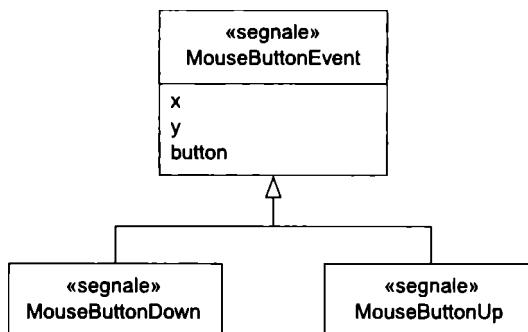


Figura 13.11

13.11 Riepilogo

In questo capitolo è stato considerato come i diagrammi di attività possano essere utilizzati per modellare molti diversi tipi di processo. Sono stati spiegati i seguenti concetti.

1. I diagrammi di attività sono diagrammi di flusso OO:
 - possono essere utilizzati per modellare qualunque tipo di processo;
 - possono essere associati a *qualunque* elemento di modellazione, per fissarne il comportamento;
 - un buon diagramma di attività illustra un aspetto specifico del comportamento di un sistema.

2. Gli stati di azione sono delle forme semplificate di stati che hanno un unico punto di ingresso. Essi sono:
 - atomici;
 - non interrompibili;
 - istantanei.
3. Lo stato iniziale indica l'inizio del diagramma di attività.
4. Lo stato finale indica la fine del diagramma di attività.
5. Gli stati di sottoattività contengono un grafo di attività completo annidato. Essi sono:
 - non atomici;
 - interrompibili;
 - richiedono, per l'esecuzione, una quantità di tempo finita.
6. Le transizioni indicano il passaggio da uno stato a un altro; le transizioni automatiche avvengono non appena uno stato ha terminato il proprio lavoro.
7. Le decisioni consentono di modellare punti di diramazione:
 - il flusso principale si divide in due o più percorsi alternativi;
 - ciascun percorso è protetto da una condizione di guardia Booleana;
 - tutte le condizioni di guardia devono essere mutuamente esclusive;
 - la parola chiave altrimenti indica il percorso che verrà seguito qualora nessuna delle condizioni di guardia risulti vera.
8. La biforcazione spacca un percorso in due o più flussi concorrenti; una biforcazione ha un'unica transizione in ingresso e due o più transizioni in uscita.
9. Le ricongiunzioni sincronizzano due o più flussi concorrenti:
 - una ricongiunzione ha una transizione in uscita e due o più transizioni in ingresso;
 - la transizione in uscita avviene soltanto quando sono avvenute tutte le transizioni in ingresso.
10. Le corsie consentono di partizionare i diagrammi di attività. È possibile definire una semantica personalizzata per le corsie. Possono rappresentare:
 - divisioni organizzative;
 - casi d'uso;
 - classi;
 - processi;
 - altro.
11. I flussi degli oggetti illustrano come gli oggetti abbiano funzione di input o di output in relazione agli stati di azione e di sottoattività, o come ne vengano modificati.

12. Un segnale è un tipo di evento.

- Un segnale consente di rappresentare un pacchetto di informazioni che viene comunicato in modo asincrono tra due oggetti.
- Un evento di segnale avviene quando un oggetto riceve un segnale.
- I segnali sono modellati come classi con lo stereotipo standard “segnale”:
 - possono avere solo attributi;
 - hanno un metodo implicito `invia(elencoDestinatari)` il quale consente di inviare il segnale a un insieme di oggetti destinatari.
- L’invio di segnale è uno speciale stato che invia un segnale.
- La ricezione di segnale è uno speciale stato che riceve il segnale; la transizione automatica in uscita dalla ricezione di segnale avviene solo quando il segnale è stato ricevuto.

Parte 4

Progettazione

Il flusso di lavoro della progettazione

14.1 Contenuto del capitolo

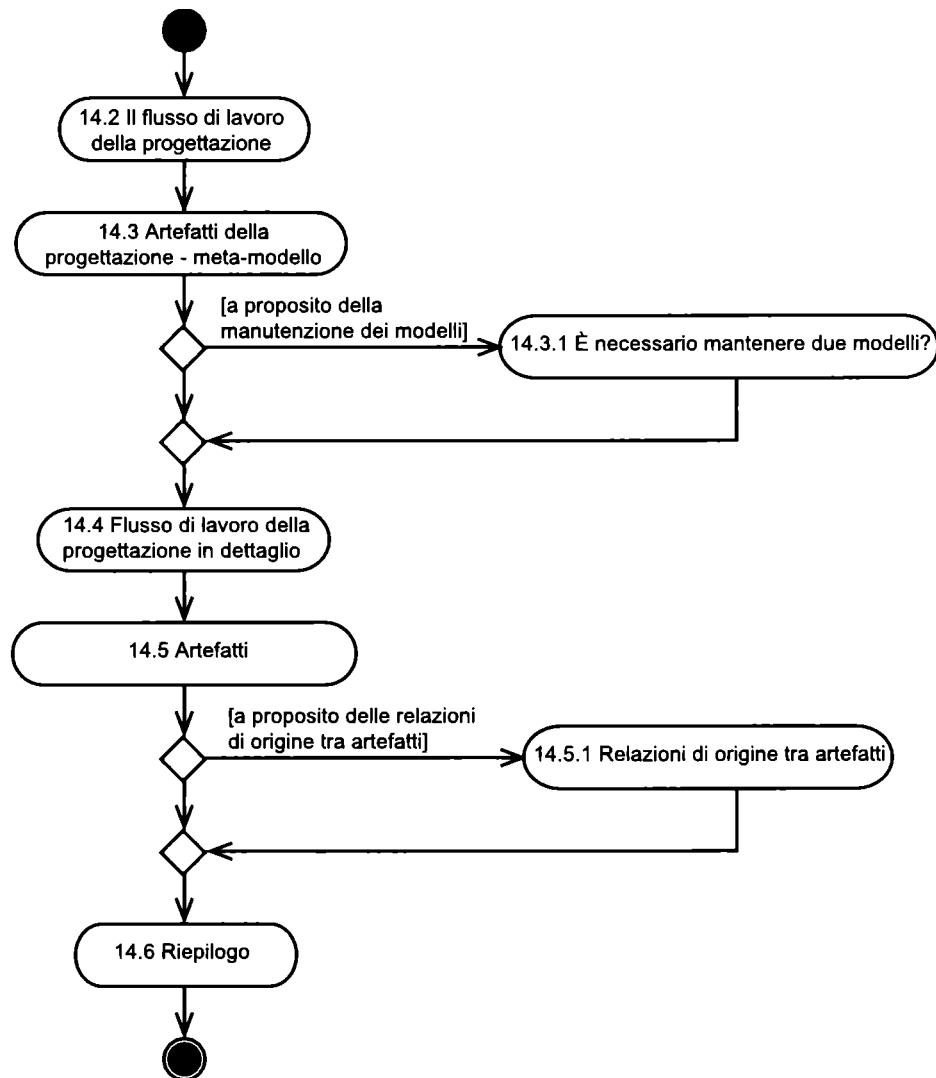
Il capitolo tratta del flusso di lavoro della progettazione, così come definito dall'UP. Uno degli argomenti principali è come il modello dell'analisi si evolva in un modello della progettazione, e se sia necessario o meno mantenere aggiornati entrambi i modelli; di questo importante punto si discuterà nel Paragrafo 14.3.1. Il resto del capitolo descrive in dettaglio il flusso di lavoro della progettazione e i relativi artefatti.

14.2 Il flusso di lavoro della progettazione

Il flusso di lavoro della progettazione costituisce la principale attività di modellazione durante l'ultima parte della fase di Elaborazione e la prima metà della fase di Costruzione. Come illustra la Figura 14.2, le prime iterazioni si concentrano sui requisiti e sull'analisi. Man mano che le attività di analisi si avvicinano alla conclusione, l'enfasi della modellazione si sposta sempre più sulla progettazione. In molti, casi l'analisi e la progettazione possono essere effettuate in parallelo. Tuttavia, come si vedrà più avanti, è molto importante distinguere chiaramente gli artefatti di questi due flussi di lavoro: il modello dell'analisi e il modello della progettazione.

Piuttosto di avere un gruppo di analisti e un gruppo di progettisti, l'UP consiglia di organizzare i gruppi di lavoro in modo verticale. Ciascun gruppo diventa responsabile di tutto il percorso di un artefatto (come per esempio un caso d'uso) dall'individuazione dei requisiti, all'analisi e alla progettazione, fino all'implementazione. Al posto di organizzare il gruppo di lavoro per competenze professionali, l'UP preferisce adottare una logica incentrata sugli artefatti consegnabili e sulle *milestone*. L'UP si concentra sugli "obiettivi" e non sulle "competenze".

Il lavoro di analisi si concentra sulla creazione di un modello logico del sistema che fissi le funzionalità che il sistema deve fornire per soddisfare i requisiti dell'utente.

**Figura 14.1**

Lo scopo delle attività di progettazione è quello di definire in modo completo come queste funzionalità debbano essere implementate. La differenza tra i due tipi di attività può essere compresa se si pensa al dominio del problema da una parte, e al dominio delle soluzioni dall'altra. I requisiti hanno origine nel dominio del problema e l'analisi può essere vista come l'esplorazione di questo dominio dal punto di vista delle parti interessate al sistema. La progettazione comporta lo studio e l'integrazione delle soluzioni tecniche (library di classi, meccanismi di persistenza ecc), che appartengono al dominio delle soluzioni, per la messa a punto di un modello del sistema (il modello della progettazione) che possa effettivamente essere implementato.

Durante la progettazione, i progettisti OO devono prendere delle decisioni in merito a questioni tecniche strategiche, quali la persistenza e la distribuzione degli oggetti, e creare un modello della progettazione appropriato. Il capoprogetto e l'architetto dovrebbero, inoltre, definire degli standard per risolvere le eventuali questioni tecniche minori.

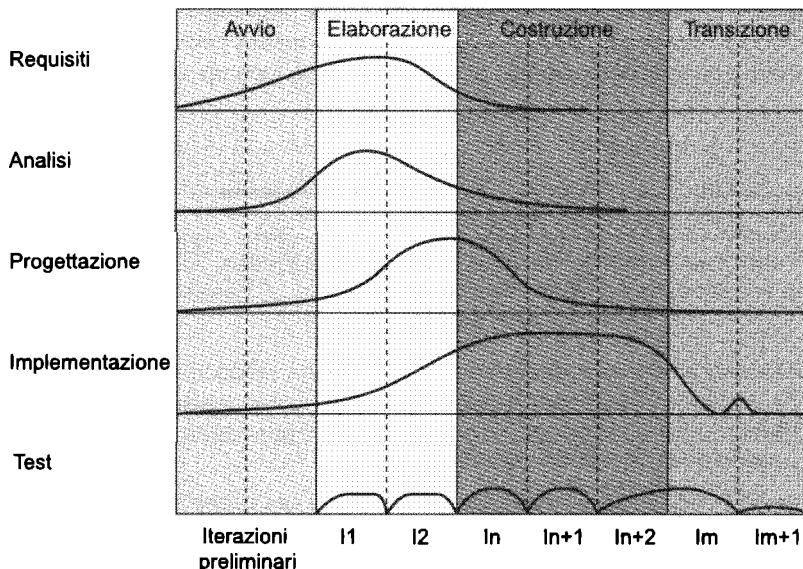


Figura 14.2 Adattata da Figura 1.5 [Jacobson 1], con il consenso della Addison-Wesley.

14.3 Artefatti della progettazione: meta-modello

La Figura 14.3 illustra il meta-modello del modello della progettazione. Il modello della progettazione contiene un unico sistema della progettazione, il quale contiene molti sottosistemi di progettazione (ne abbiamo mostrati solo cinque nella figura). Nella progettazione si utilizza il termine “sottosistema” al posto di “package”, per evidenziare che si è passati dalla vista puramente concettuale propria dell’analisi, a un modello fisico del sistema.

Nonostante possa capitare che già durante l’analisi siano state individuate diverse interfacce chiave del sistema, è proprio durante le attività di progettazione che ci si concentra sulle interfacce. Questo perché, in ultima analisi, il sistema sarà tenuto insieme proprio dalle interfacce tra i diversi sottosistemi di progettazione. Le interfacce, dunque, hanno un ruolo architettonurale molto importante durante la progettazione, e gran parte del lavoro di progettazione è dedicato alla individuazione e modellazione delle interfacce principali.

Esiste una semplice relazione di «origine» tra il modello dell’analisi e il modello della progettazione: il modello della progettazione si basa sul modello dell’analisi, e può essere considerato una sua evoluzione o elaborazione (come illustrato nella Figura 14.4). Un ragionamento simile vale anche per la semplice relazione biunivoca esistente tra il sistema dell’analisi e il sistema della progettazione.

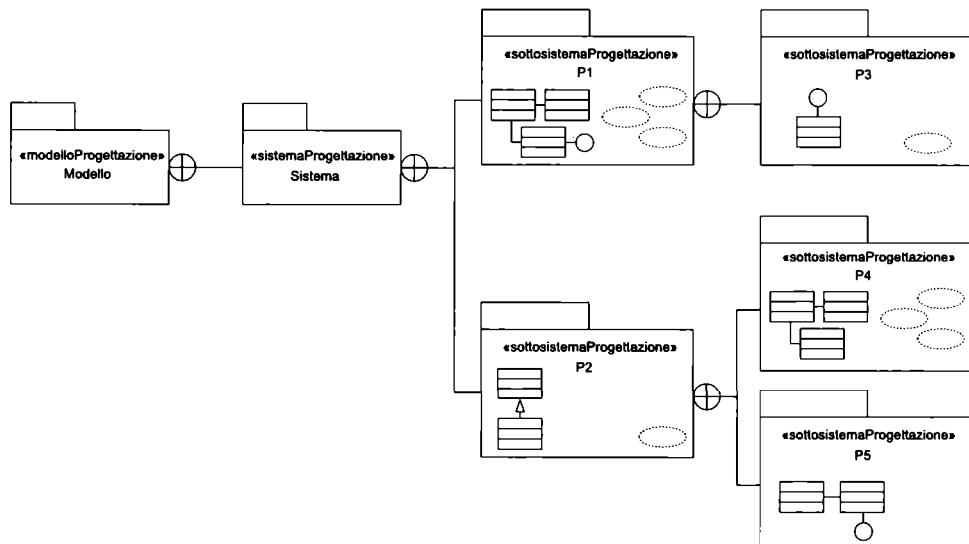


Figura 14.3

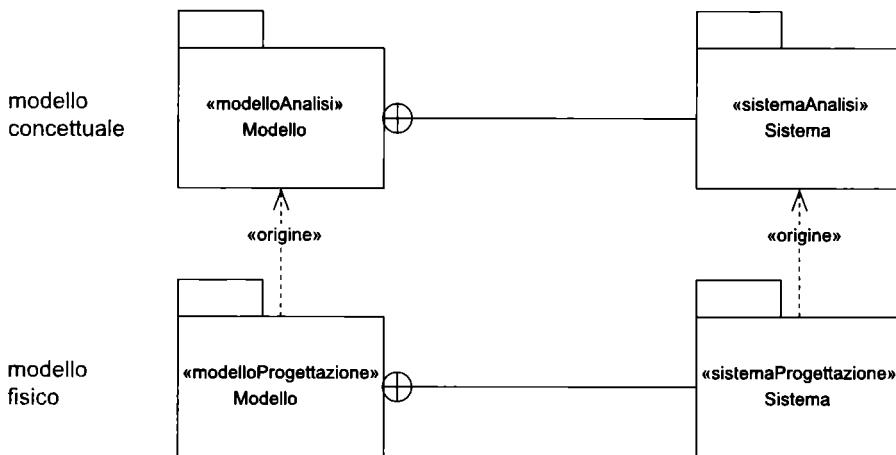


Figura 14.4

La relazione tra i *package* di analisi e i *package* di progettazione è leggermente più complessa. Spesso un sottosistema di progettazione ha come «origine» un *package* di analisi, ma non è sempre così. Possono esistere valide motivazioni tecniche o di architettura per scomporre un unico *package* di analisi in diversi sottosistemi di progettazione. Per esempio, se si pratica lo sviluppo basato sui componenti (*component-based development*, CBD), si potrebbe decidere di rappresentare ciascun macro-componente con un suo sottosistema di progettazione. In questo caso un *package* di analisi potrebbe corrispondere a parecchi sottosistemi di progettazione, a seconda anche del livello di dettaglio dei singoli componenti.

14.3.1 È necessario mantenere due modelli?

In un mondo ideale esisterebbe un unico modello del sistema, e lo strumento CASE utilizzato sarebbe in grado di generare da quello stesso modello, sia la vista di analisi, sia la vista di progettazione. In realtà questo requisito è molto più complesso di quanto non possa sembrare, e nessuno strumento CASE esistente è in grado di fornire una buona vista di analisi e una buona vista di progettazione di uno stesso modello sottostante. Restano, dunque, quattro possibili strategie: queste sono illustrate nella Tabella 14.1.

Tabella 14.1

Strategia	Conseguenze
1. Trasformare il modello dell'analisi, riferendolo, nel modello della progettazione.	È sufficiente mantenere aggiornato il solo modello della progettazione, ma il modello dell'analisi è perso.
2. Trasformare il modello dell'analisi, riferendolo, nel modello della progettazione, quindi utilizzare uno strumento CASE per recuperarne una "vista di analisi".	È sufficiente mantenere aggiornato il solo modello della progettazione, ma la "vista di analisi" recuperabile dallo strumento CASE potrebbe non essere soddisfacente.
3. Congelare il modello dell'analisi a un certo punto della fase di Elaborazione: effettuare una copia del modello dell'analisi e trasformarla, riferendola, nel modello della progettazione.	Esistono entrambi i modelli, ma non sono sincronizzati.
4. Mantenere due modelli distinti: il modello dell'analisi e il modello della progettazione.	Esistono entrambi i modelli – sono sincronizzati, ma è necessario tenerli allineati manualmente.

Non esiste una strategia migliore: dipende molto dal progetto. La domanda fondamentale da porsi è: "Abbiamo bisogno di mantenere la vista di analisi del sistema?". La vista di analisi fornisce il quadro d'insieme di un sistema. La vista di analisi può contenere anche solo tra l'1% e il 10% delle classi presenti nella più dettagliata vista di progettazione, ed è quindi tipicamente molto più comprensibile. Rimane dunque molto utile per:

- introdurre nuove risorse sul progetto;
- comprendere il funzionamento del sistema anche a distanza di mesi o anni dal rilascio;
- comprendere come le funzionalità del sistema soddisfino e corrispondano ai requisiti dell'utente;
- pianificare interventi di manutenzione e migliorie;
- comprendere l'architettura logica del sistema;
- appaltare la costruzione del sistema a risorse esterne.

Se si prevede di dover effettuare una delle suddette attività, allora è fondamentale preservare la vista di analisi. Solitamente è consigliato preservare comunque la vista di analisi per tutti i sistemi di grandi dimensioni, complessi o strategici, o che abbiano una lunga aspettativa di vita. In questi casi, è necessario scegliere la strategia 3 o la strategia 4. La decisione se lasciare o meno che il modello dell'analisi e il modello della progettazione si disallineino deve essere ben ponderata. Bisogna chiedersi se sia veramente accettabile per il progetto che questo succeda.

Se il sistema è piccolo (meno di 200 classi di progettazione), allora il modello della progettazione è di per sé sufficientemente ridotto da essere comprensibile. In questo caso, potrebbe non essere necessario preservare anche il modello dell'analisi. Il mantenimento dei due modelli separati potrebbe essere eccessivo anche nel caso in cui il sistema non sia strategico, o sia previsto che abbia una vita breve. In queste situazioni, bisogna scegliere tra la strategia 1 e la strategia 2, e le capacità dello strumento CASE utilizzato costituiscono un fattore determinante per questa scelta. Alcuni strumenti CASE gestiscono il solo modello della progettazione, ma consentono di filtrare e occultare alcune informazioni nel tentativo di ricostruire una vista di "analisi" del modello della progettazione. Questa vista è un compromesso accettabile per molti sistemi di medie dimensioni, ma potrebbe non essere sufficiente per gestire sistemi molto grandi.

Un'ultima considerazione: sarebbe saggio ricordarsi che moltissimi sistemi restano in vita molto più a lungo di quanto inizialmente previsto!

14.4 Flusso di lavoro della progettazione in dettaglio

La Figura 14.5 illustra il flusso di lavoro della progettazione previsto dall'UP. Le principali risorse coinvolte nella progettazione sono: l'architetto, l'ingegnere dei casi d'uso e l'ingegnere dei componenti. Nella maggior parte dei progetti OO, il ruolo di architetto è coperto da uno o più individui dedicati, mentre un'unica persona, in tempi diversi, può ricoprire entrambi i ruoli di ingegnere dei casi d'uso e ingegnere dei componenti.

Uno degli obiettivi dell'UP è che ciascun partecipante al progetto diventi esperto e responsabile di una parte del sistema su tutta la linea, dall'analisi all'implementazione.

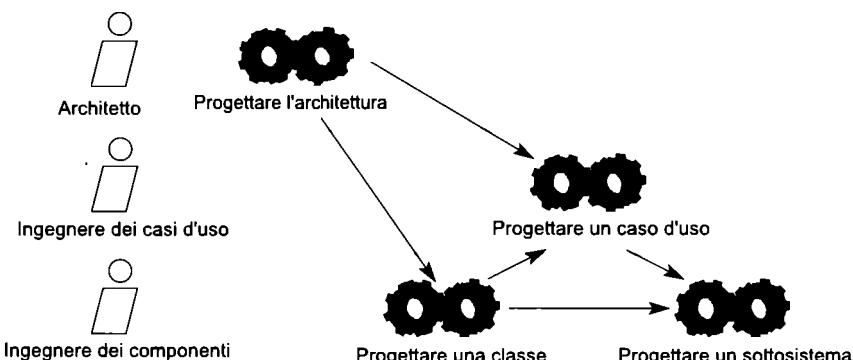


Figura 14.5 Riprodotta dalla Figura 9.16 [Jacobson 1], con il consenso della Addison-Wesley.

Quindi la persona, o il gruppo, che inizialmente effettua l'analisi OO di un particolare del sistema, sarà spesso anche responsabile di rifinire tale analisi con attività di progettazione e, magari aggiungendo qualche risorsa più esperta di programmazione, trasformarla in codice. Il vantaggio di questo approccio è che impedisce la pratica dello "scarica-barile" tra analisti, progettisti e programmatore, pratica che può essere piuttosto comune nei progetti OO.

14.5 Artefatti

Il modello della progettazione può essere considerato un'evoluzione del modello dell'analisi, al quale sono stati aggiunti un maggiore dettaglio e soluzioni tecniche specifiche. Il modello della progettazione contiene gli stessi tipi di elementi del modello dell'analisi, ma tutti gli artefatti sono più completi e includono dettagli relativi all'implementazione. Per esempio, una classe di analisi può essere poco più uno schizzo contenente qualche attributo e alcune delle operazioni più importanti. D'altra parte, una classe di progettazione deve, invece, essere completamente definita: devono essere specificati tutti gli attributi e tutte le operazioni (compresi il tipo restituito e l'elenco di tutti i parametri).

Il modello della progettazione è composto da:

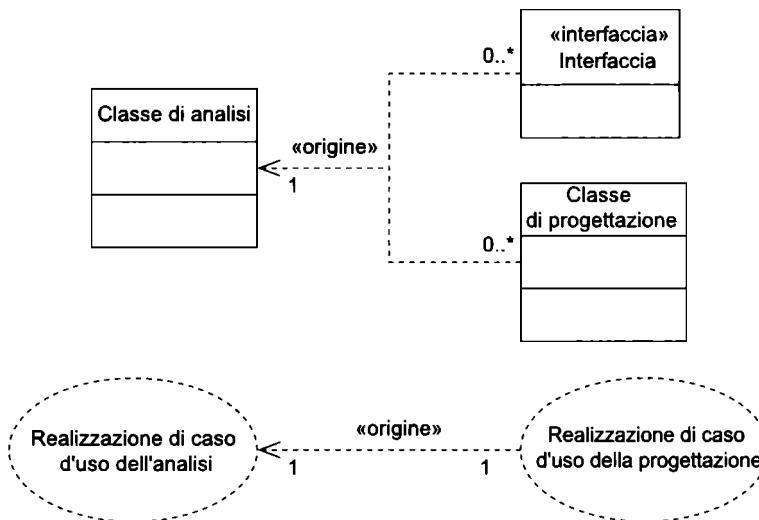
- sottosistemi di progettazione;
- classi di progettazione;
- interfacce;
- realizzazioni di caso d'uso della progettazione;
- un *diagramma di deployment*.

Uno degli artefatti più importanti prodotti durante la progettazione sono le interfacce. Il Capitolo 17 spiega come queste consentano di eliminare le interdipendenze esistenti nel sistema e di creare dei sottosistemi che possano essere sviluppati in parallelo.

La progettazione produce anche una prima bozza del *diagramma di deployment*, il quale illustra come il sistema software verrà distribuito sui nodi computazionali fisici. Questo diagramma è ovviamente importante e strategico. Tuttavia, dato che la maggior parte del lavoro sul *diagramma di deployment* viene effettuato durante l'implementazione, questo artefatto non sarà affrontato fino al Capitolo 23.

14.5.1 Relazioni di origine tra artefatti

Una classe di analisi può risolversi in una o più interfacce o classi di progettazione. Questo succede perché le classi di analisi costituiscono una rappresentazione concettuale, di alto livello, delle classi del sistema. Quando si giunge, infine, alla modellazione fisica (progettazione), ci si potrebbe rendere conto che queste classi concettuali richiedono l'implementazione di una o più classi o interfacce fisiche (o di progettazione), così come illustrato nella Figura 14.6.

**Figura 14.6**

La realizzazione di caso d'uso dell'analisi ha, invece, una semplice relazione «origine» biunivoca con la realizzazione di caso d'uso della progettazione. Nella progettazione, la realizzazione di caso d'uso ha semplicemente un maggiore dettaglio.

14.6 Riepilogo

Il flusso di lavoro della progettazione serve per determinare come debbano essere implementate le funzionalità individuate e specificate dal modello dell'analisi. Sono stati spiegati i seguenti concetti.

- Il flusso di lavoro della progettazione comprende le principali attività di modellazione effettuate durante l'ultima parte della fase di Elaborazione e la prima parte della fase di Costruzione.
 - Le attività di analisi e di progettazione possono essere in parte effettuate in parallelo.
 - Un singolo individuo o gruppo di lavoro dovrebbe occuparsi di ciascun artefatto, dall'analisi alla progettazione.
 - I progettisti OO dovrebbero concentrarsi sulle questioni strategiche, quali l'architettura distribuita dei componenti; si possono introdurre standard e linee guida per risolvere le questioni più tattiche.
- Il modello della progettazione contiene:
 - un unico sistema della progettazione;

- i sottosistemi di progettazione;
 - le realizzazioni di caso d'uso della progettazione;
 - le interfacce;
 - le classi di progettazione;
 - il *diagramma di deployment* (prima bozza).
3. Esistono relazioni di origine tra:
- il modello della progettazione e il modello dell'analisi;
 - il sistema della progettazione e il sistema dell'analisi;
 - uno o più sottosistemi di progettazione e un *package* di analisi.
4. È consigliabile mantenere sia il modello dell'analisi, sia il modello della progettazione se il sistema è:
- di grandi dimensioni;
 - complesso;
 - strategico;
 - soggetto a frequenti revisioni e modifiche;
 - di lunga vita, almeno secondo le previsioni;
 - sviluppato da risorse esterne.

Classi di progettazione

15.1 Contenuto del capitolo

Il capitolo tratta delle classi di progettazione. Queste sono i principali elementi che costituiscono il modello della progettazione, ed è fondamentale per un progettista OO capire come modellare queste classi in modo efficace.

Il capitolo prima descrive la struttura di una classe di progettazione, quindi, nel Paragrafo 15.4, spiega quali sono le caratteristiche di una classe di progettazione ben formata. Illustra i requisiti di completezza e sufficienza, essenzialità, massima coesione e minima interdipendenza. Infine, si concentra sull'ereditarietà e le sue alternative, quali l'aggregazione.

15.2 Cosa sono le classi di progettazione?

Le classi di progettazione sono classi le cui specifiche sono talmente complete che possono essere effettivamente implementate.

Nell'analisi le classi nascono dal dominio dei problemi, il quale è costituito dall'insieme dei requisiti che descrivono il problema che si vuole risolvere. Come considerato, le classi di analisi possono avere come origine i casi d'uso, l'SRS, i glossari o qualunque altra informazione pertinente.

Le classi di progettazione hanno due origini.

- Il dominio dei problemi, attraverso un processo di rifinitura delle classi di analisi; questa rifinitura comporta l'aggiunta dei dettagli relativi all'implementazione. Durante questa attività, capita spesso di dover scomporre una classe di analisi di alto livello in due o più distinte classi di progettazione. Esiste una relazione «origine» tra una classe di analisi e la classe, o le classi, di progettazione che ne descrivono l'implementazione.

- Il dominio delle soluzioni: il dominio delle soluzioni è l'insieme di tutte le librerie di classi di utilità e di componenti riutilizzabili quali classi per gestire i tipi comuni Time, Date, String, classi contenitore, ecc. Vi trovano posto anche i prodotti *middleware*, quali quelli di comunicazione o di *database* (relazionali o a oggetti), e i *framework* di componenti, quali DCOM, CORBA o Enterprise JavaBean. Il dominio delle soluzioni comprende anche i *framework GUI* (Graphical User Interface, Interfaccia Utente Grafica). Questo dominio fornisce tutti gli strumenti tecnici che consentono di implementare il sistema.

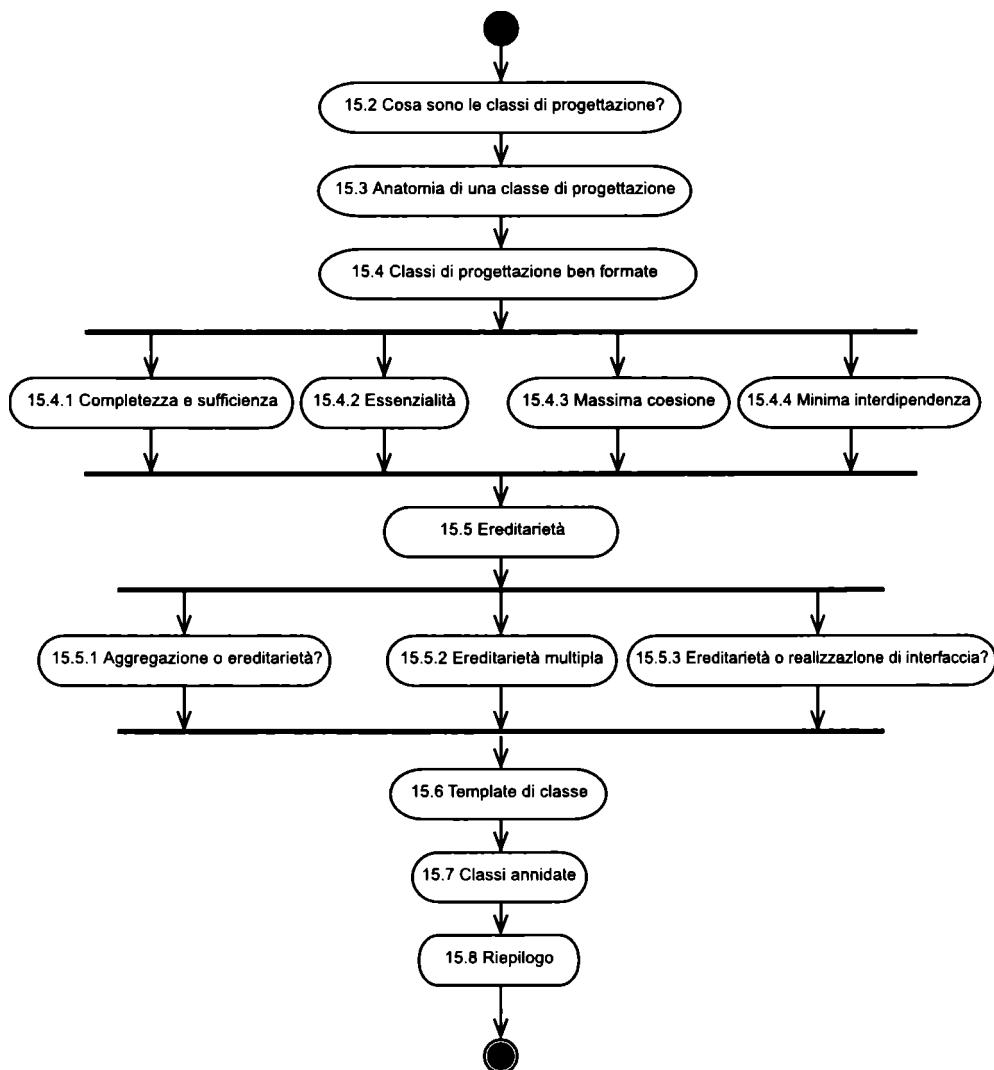


Figura 15.1

L'analisi serve a modellare *cosa* il sistema dovrebbe fare. La progettazione serve a modellare *come* tale comportamento debba essere implementato.

Perché una classe di analisi può risolversi in una o più classi di progettazione o interfacce? Il motivo è legato al fatto che la classe di analisi è definita a un livello di astrazione molto alto. Durante l'analisi non ci si preoccupa di definire l'insieme completo degli attributi, e l'insieme delle operazioni individuate vuole solo essere una bozza che illustra i servizi principali che la classe offre al resto del sistema. Quando si passa la stessa classe in progettazione, diviene necessario specificarne tutte le operazioni e tutti gli attributi; capita così spesso di scoprire che la classe è cresciuta troppo. In questo caso, è consigliabile scomporla in due o più classi più piccole. La progettazione richiede che le classi definite siano piccole unità autosufficienti e coesive, in grado di svolgere uno o due compiti molto bene. Bisogna evitare a tutti i costi di progettare enormi classi "tuttofare", in grado di svolgere qualunque operazione per chiunque.

15.3 Anatomia di una classe di progettazione

Durante l'analisi si tenta di fissare il comportamento richiesto al sistema, senza preoccuparsi affatto di come questo comportamento verrà in seguito implementato. La progettazione deve specificare esattamente come queste classi assolveranno le loro responsabilità. Per raggiungere questo obiettivo è necessario:

- completare l'insieme degli attributi della classe, includendo tutte le specifiche, ovvero: nome, tipo, visibilità e (opzionalmente) valore predefinito;
- ricavare dalle operazioni specificate in analisi l'insieme completo dei metodi della classe.

Questo processo di rifinitura è illustrato con un esempio molto semplice nella Figura 15.2.

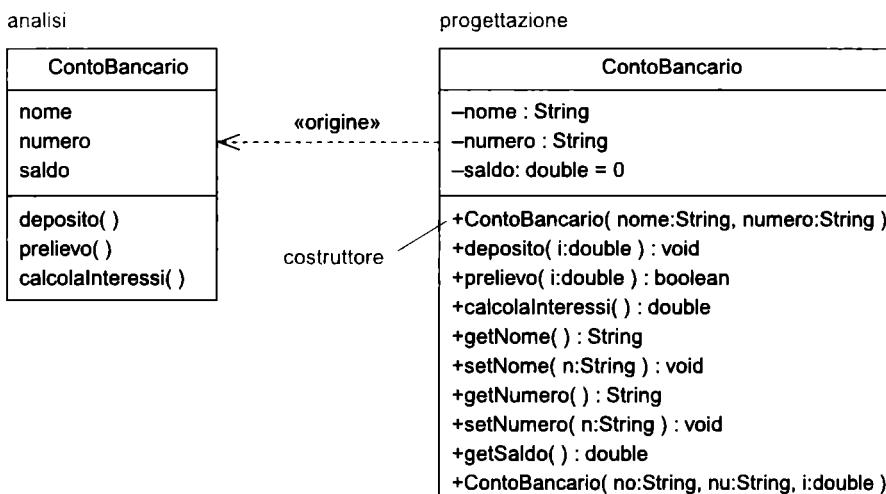


Figura 15.2

Come spiega il Capitolo 9, un'operazione è la specifica logica ad alto livello di un frammento di funzionalità offerto da una classe. Un metodo è, invece, una funzione completamente definita, pronta a essere implementata nel codice sorgente. Una singola operazione può, dunque, essere risolta con uno o più metodi implementabili.

Per illustrare questo punto si prenda in considerazione il seguente esempio. In un sistema per una compagnia aerea, durante l'analisi viene specificata un'operazione di alto livello chiamata `checkIn()`. Tuttavia, come sa chiunque abbia mai effettuato il check-in per prendere un volo, il `check-in` è in realtà un processo di *business* moderatamente complesso che comprende: farsi dare dal passeggero un certo numero di informazioni, verificarle, prendere i suoi bagagli e assegnargli un posto sul velivolo. Sembra ragionevole, quindi, pensare che l'operazione di alto livello `checkIn()` definita durante l'analisi venga successivamente, durante le attività di progettazione del processo, scomposta in una serie di metodi di più basso livello. Può darsi che venga comunque mantenuto un metodo di alto livello `checkIn()`, ma avendo questo metodo troppe responsabilità, dovrà delegarne alcune a una serie di metodi ausiliari, o a metodi di altre classi. Può anche succedere che il processo di *check-in* sia talmente complesso da richiedere la definizione di nuove classi ausiliarie.

15.4 Classi di progettazione ben formate

Quando si progetta una classe è sempre importante considerarla dal punto di vista dei suoi potenziali clienti. Come vedranno questa classe: è troppo complessa? Manca qualcosa? È molto dipendente da altre classi o no? Sembra fare ciò che il suo nome suggerisce? Si tratta di considerazioni importanti, sintetizzabili nelle seguenti quattro caratteristiche minime che una classe di progettazione deve avere, per poter essere considerata ben formata:

- completezza e sufficienza;
- essenzialità;
- massima coesione;
- minima interdipendenza.

Il modello della progettazione verrà dato ai programmatore che produrranno il codice sorgente finito, oppure il codice potrebbe essere generato direttamente dal modello stesso, se lo strumento CASE lo supporta. Le classi di progettazione devono, dunque, essere completamente definite. Proprio durante questo processo di definizione, diventa necessario stabilire se le classi siano ben formate o meno.

15.4.1 Completezza e sufficienza

I metodi pubblici di una classe definiscono un contratto tra la classe e i suoi clienti. È importante che questo contratto sia chiaro, ben definito e accettabile per tutte le parti interessate, proprio come per un normale contratto di affari.

Una classe è completa se fornisce ai suoi clienti tutto ciò che essi si aspettano dalla classe. I clienti inferiscono dal nome della classe quali debbano essere i metodi che la classe mette a loro disposizione. Per fare un esempio relativo al mondo reale: comprando una macchina nuova si può ragionevolmente presupporre che abbia delle ruote! Lo stesso meccanismo è applicabile anche alle classi; quando si dà un nome a una classe e se ne descrive la semantica, i clienti di quella classe, basandosi su queste informazioni, si aspetteranno che metta a loro disposizione un certo insieme di metodi. Per esempio, se la classe ContoBancario mette a disposizione un metodo prelievo(...), si può presupporre che abbia anche un metodo deposito(...). E poi ancora, se si progetta una classe chiamata CatalogoProdotti, i clienti potrebbero ragionevolmente presupporre che questa classe consenta loro di aggiungere, rimuovere e trovare Prodotti del catalogo. Queste operazioni sono praticamente implicite nel nome stesso della classe. La completezza è la capacità che una classe ha di soddisfare le ragionevoli richieste dei suoi clienti.

Una classe viene detta, invece, sufficiente se tutti i suoi metodi sono esclusivamente finalizzati alla realizzazione dell'unico scopo che la classe deve avere. Una classe non deve mai sorprendere i propri clienti. Deve contenere i metodi che ci si aspetta che contenga, e solo quelli. Un tipico errore commesso da progettisti poco esperti è quello di prendere una classe semplice e sufficiente, quale per esempio ContoBancario, e aggiungervi metodi per processare carte di credito o polizze assicurative ecc. La sufficienza di una classe misura quanto essa sia semplice e concentrata sulle sue poche responsabilità.

La regola d'oro della completezza e della sufficienza è che una classe deve fare esattamente ciò che i suoi utenti si aspettano: nulla di più e nulla di meno.

15.4.2 Essenzialità

I metodi devono essere progettati per offrire un unico servizio primitivo e atomico. Una classe *non* deve mettere a disposizione più di un modo per effettuare la stessa operazione, in quanto così creerebbe confusione tra i suoi clienti, e potrebbe anche portare a un aumento dei costi di manutenzione e a problemi di consistenza.

Se, per esempio, la classe ContoBancario ha un metodo primitivo per effettuare un deposito, non dovrebbe avere anche un metodo più complesso che consente di effettuare contestualmente una serie di depositi. Questo perché lo stesso risultato può essere ottenuto chiamando ripetutamente il metodo primitivo. Il punto è che le classi progettate devono sempre rendere disponibile l'insieme di metodi il più essenziale e semplice possibile.

Anche se questa caratteristica, detta appunto essenzialità, è solitamente desiderabile, esistono situazioni reali in cui può costituire un vincolo troppo rigido. Una ragione comune per cui a volte si sacrifica parzialmente l'essenzialità, è l'esigenza di aumentare le prestazioni del codice. Se, per esempio, effettuare i depositi bancari in serie risultasse molto più efficiente che non effettuarli singolarmente, allora avrebbe senso sacrificare l'essenzialità e aggiungere alla classe ContoBancario un metodo deposito(...) più complesso che possa gestire contestualmente più transazioni. Tuttavia, bisogna *sempre* iniziare da una situazione di massima essenzialità, ovvero con l'insieme di metodi più primitivo possibile. Solo successivamente, e solo in presenza di un motivo valido e comprovato, si può effettivamente pensare di rilassare il vincolo di essenzialità, aggiungendo complessità.

15.4.3 Massima coesione

Ogni classe dovrebbe modellare un unico concetto astratto e dovrebbe avere un insieme di metodi a supporto di questo suo unico compito: questa è la coesione. Se una classe comincia ad avere molte responsabilità diverse, allora alcune di queste devono essere implementate da classi ausiliarie (dette anche classi “*helper*”). La classe principale delega alcune responsabilità alle classi ausiliarie.

La coesione è una delle caratteristiche più desiderabili in una classe. Classi molto coese sono generalmente facili da comprendere, riusare e mantenere. Una classe coesa ha un insieme ridotto di responsabilità, molto correlate tra loro. Ogni metodo, attributo e associazione della classe è progettato esclusivamente per implementare questo insieme ridotto e concentrato di responsabilità.

La Figura 15.3 illustra il modello di un sistema di prenotazioni e vendite che ci ha lasciati piuttosto perplessi. È costituito da una classe BeanAlbergo, una classe BeanAuto e una classe BeanAlbergoAuto; i “*bean*” sono Enterprise JavaBean (EJB). BeanAlbergo è responsabile per la vendita di pernottamenti in albergo, BeanAuto per la vendita di servizi di autonoleggio, BeanAlbergoAuto per la vendita di pacchetti combinati di pernottamenti e autonoleggio. Ovviamente questo modello contiene diversi errori.

- I nomi delle classi sono discutibili: sarebbe molto più indicato chiamarle PernottamentoAlbergo e Autonoleggio.
- Il prefisso “Bean” non è necessario in quanto fa riferimento a un particolare dettaglio relativo all’implementazione.
- La classe BeanAlbergoAuto soffre di poca coesione: ha due responsabilità principali (vendere pernottamenti in albergo e vendere servizi di autonoleggio) le quali, tra l’altro, risiedono già in altre classi.

Dal punto di vista della coesione, BeanAlbergo e BeanAuto sono accettabili (ma devono comunque essere rinominate), ma BeanAlbergoAuto è semplicemente assurda.

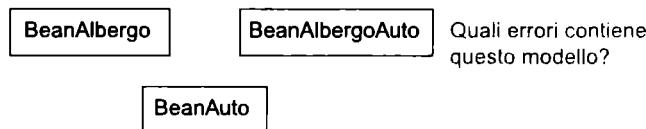


Figura 15.3

15.4.4 Minima interdipendenza

Una data classe dovrebbe essere associata all’insieme *minimo* di altre classi che le consente di realizzare le proprie responsabilità. Due classi dovrebbero essere associate soltanto se tra loro esiste una correlazione semantica: questa viene detta interdipendenza minima.

Uno degli errori comuni dei progettisti OO alle prime armi, è quello di connettere ogni elemento del modello a una moltitudine di altri elementi, a seconda delle esigenze che vengono progressivamente individuate. L'interdipendenza è proprio il nemico peggiore della modellazione a oggetti. Bisogna sforzarsi attivamente e continuamente per ridurre il numero di relazioni tra le classi, minimizzando quindi l'interdipendenza.

Un modello a oggetti molto interconnesso è l'equivalente del "codice-spaghetti" del mondo non-OO, e può produrre un sistema difficile da comprendere e da mantenere. I sistemi OO con un alto grado di interdipendenza sono spesso il risultato di progetti in cui non viene effettuata alcuna attività formale di progettazione, e dove il sistema viene lasciato crescere secondo le esigenze del momento.

I progettisti con poca esperienza devono stare attenti a non mettere in relazione due classi solo perché una delle due contiene un frammento di codice che potrebbe essere utile anche all'altra. Questa è la forma peggiore di riuso del codice, in quanto si sacrifica l'integrità dell'architettura del sistema per ottenere un piccolo risparmio di tempo di implementazione. In effetti, ogni relazione tra classi andrebbe ponderata con molta cura.

Molte associazioni tra classi presenti nel modello della progettazione nascono direttamente sul modello dell'analisi, ma molte altre vengono introdotte per rispettare vincoli di implementazione o per riusare codice. Questo secondo tipo di associazioni è quello che richiede maggiore attenzione.

Ovviamente un certo livello di interdipendenza è accettabile e desiderabile. All'interno di un *package* l'interdipendenza può anche essere molto alta, ma va bene così perché significa che il *package* è fortemente coeso. È l'interdipendenza tra *package* diversi che compromette l'architettura del sistema, ed è questa che deve essere attivamente individuata e ridotta al minimo.

15.5 Ereditarietà

Durante le attività di progettazione, l'ereditarietà tra classi assume un maggior rilievo.

Nel modello dell'analisi, si utilizza l'ereditarietà *soltanto* se tra due classi di analisi esiste un'ovvia e non ambigua relazione "è un/una". In progettazione, invece, si può anche decidere di utilizzare l'ereditarietà per riusare il codice di classe.

Si tratta di una strategia diversa: l'ereditarietà in questo caso serve per facilitare l'implementazione di una classe discendente e non per rappresentare una relazione di *business* tra due classi.

Le sezioni che seguono approfondiscono alcune strategie di progettazione che consentono di utilizzare l'ereditarietà in modo efficace.

15.5.1 Aggregazione o ereditarietà?

L'ereditarietà è una tecnica molto potente: è il meccanismo che consente di avere polimorfismo nei linguaggi fortemente tipati, quali Java, C#, e C++. Tuttavia i progettisti OO poco esperti spesso ne abusano. È necessario capire che l'ereditarietà ha anche alcune caratteristiche non desiderabili.

Tali caratteristiche sono le seguenti.

- È in assoluto la forma di interdipendenza più forte tra due o più classi.
- All'interno di una gerarchia di classi l'incapsulazione risulta debole. Modifiche alla classe base si propagano alle sottoclassi. Questo causa il problema noto come “fragilità della classe base”, per cui modifiche alla classe base possono avere un impatto notevole su molte altre classi del sistema.
- È un tipo di relazione poco flessibile. In tutti i linguaggi OO comunemente utilizzati, la relazione di ereditarietà non è modificabile a *run-time*. È possibile modificare gerarchie di aggregazione o composizione a *run-time*, creando e distruggendo relazioni, ma le gerarchie di ereditarietà restano immutabili. Si tratta in assoluto della forma meno flessibile di relazione tra classi.

La Figura 15.4 illustra una soluzione, tipica di un progettista poco esperto, al problema di modellazione dei ruoli in una società. A un primo esame sembra plausibile, tuttavia nasconde qualche problema. Per esempio, supponiamo di avere un oggetto *john* di tipo Programmatore, e di volerlo promuovere al tipo Manager. Come possiamo fare? Non è possibile cambiare la classe dell'oggetto *john* a *run-time*, e quindi l'unico modo di ottenere l'effetto desiderato è di creare un nuovo oggetto Manager (chiamato *john:Manager*), copiarvi tutte le informazioni di interesse contenute nell'oggetto *john:Programmatore* e, infine, distruggere l'oggetto *john:Programmatore* per preservare la consistenza dell'applicazione. Ovviamente questa procedura è complessa e ben diversa da ciò che succede nel mondo reale.

In realtà, il modello della Figura 15.4 nasconde un errore semantico fondamentale. È più corretto dire che l'impiegato è un impiego o che l'impiegato *ha* un impiego? Questa domanda suggerisce la soluzione del problema, riportata nella Figura 15.5.

Utilizzando l'aggregazione si riesce a correggere l'errore semantico: un *Impiegato ha* un *Impiego*. Questo modello è più flessibile e se fosse necessario consentirebbe a un *impiegato* di avere anche più di un *impiego*.

Sostituendo l'ereditarietà con l'aggregazione, come meccanismo per attribuire impieghi agli impiegati, si è ottenuto un modello molto più flessibile e semanticamente corretto. Questo esempio illustra un principio generale molto importante: le sottoclassi dovrebbero sempre rappresentare una relazione “è un tipo di”, e non una relazione “è un ruolo interpretato da”.

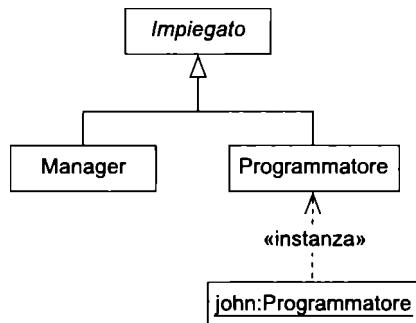
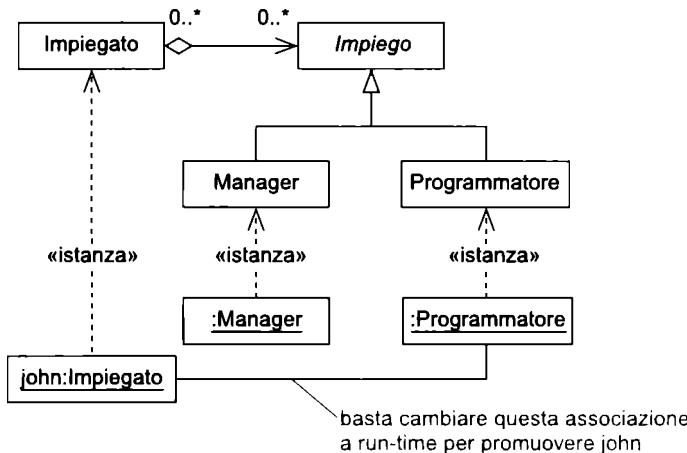


Figura 15.4

**Figura 15.5**

Quando si pensa alla semantica di *business* di società, impiegati e impieghi, diventa evidente che l'impiego è un *ruolo interpretato* da un impiegato e che non rappresenta, invece, un *tipo di* impiegato. Per questo motivo, l'ereditarietà è sicuramente una scelta sbagliata per modellare questo tipo di relazione di *business*. D'altra parte in una società esistono molti tipi di impieghi. Questo suggerisce che questi impieghi potrebbero essere modellati bene con una gerarchia ereditaria (basata sulla classe base astratta *Impiego*).

15.5.2 Ereditarietà multipla

A volte c'è l'esigenza di ereditare implementazione da più di una classe genitore. Questo meccanismo, detto ereditarietà multipla, non è supportato da tutti i linguaggi OO. Java e C#, per esempio, prevedono solo l'ereditarietà singola. In pratica però questa mancanza di supporto per l'ereditarietà multipla non costituisce un vero problema, dato che si può ottenere lo stesso effetto con l'uso combinato di ereditarietà singola e delegazione. Anche se l'ereditarietà multipla a volte offre la soluzione più elegante, deve essere usata solo se è supportata dal linguaggio che verrà utilizzato per l'implementazione. Seguono le principali considerazioni che concernono l'ereditarietà multipla.

- Le classi genitore devono risultare tutte semanticamente disgiunte tra loro. Se esiste una sovrapposizione semantica tra le classi base queste potrebbero interagire in modo imprevisto e la sottoclasse potrebbe comportarsi in modo strano. La formulazione corretta di questo vincolo è che le classi base devono essere tra loro ortogonali.
- Il principio di sostituibilità e la relazione “è un tipo di” devono essere applicabili tra la sottoclasse e ciascuna delle superclassi interessate.
- Le superclassi non dovrebbero avere antenati in comune. In caso contrario si viene a formare un anello nella gerarchia di ereditarietà delle classi per cui le stesse caratteristiche vengono ereditate da più percorsi, ovvero da diverse superclassi. Alcuni linguaggi che supportano l'ereditarietà multipla (come il C++) hanno appositi meccanismi per risolvere le ambiguità che nascono da anelli presenti nella gerarchia di ereditarietà delle classi.

Un caso in cui l'ereditarietà multipla si rileva semplice ed efficace è quello delle classi "mixin". Si tratta di classi che non sono autosufficienti, ma progettate specificatamente per essere incorporate in altre classi, utilizzando l'ereditarietà. La classe Dialer illustrata Figura 15.6 è una semplice classe mixin. Il suo unico comportamento è quello di comporre un numero telefonico, e non ha quindi una grande utilità come classe isolata. Tuttavia fornisce un insieme coeso di comportamenti che può essere riusato in modo diffuso in altre classi tramite l'ereditarietà multipla. Questa classe mixin è un esempio di classe di utilità generale, che potrebbe entrare a far parte di una libreria di utilità riusabili.

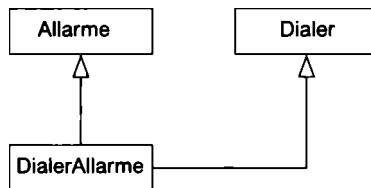


Figura 15.6

15.5.3 Ereditarietà o realizzazione di interfaccia?

Con l'ereditarietà una sottoclassa ottiene due cose:

- l'interfaccia: i metodi pubblici della classe base;
- l'implementazione: gli attributi, le associazioni e i metodi privati della classe base.

Con la realizzazione di interfaccia (vedere Capitolo 17) la sottoclassa ottiene una cosa sola:

- l'interfaccia: un insieme di operazioni pubbliche, senza implementazione alcuna.

L'ereditarietà e la realizzazione di interfaccia hanno qualcosa in comune, in quanto entrambe consentono di definire un contratto (un insieme di metodi) che le sottoclassi devono implementare. Tuttavia queste due tecniche hanno una semantica e un utilizzo molto diversi tra loro.

L'ereditarietà è necessaria solo se lo scopo è quello di ereditare anche alcuni dettagli implementativi (metodi, attributi, associazioni) dalla superclasse. Si tratta di una forma di riuso, e in effetti, quando i linguaggi OO fecero la loro prima comparsa, l'ereditarietà veniva considerata il principale meccanismo di riuso. Tuttavia, da allora è passata molta acqua sotto i ponti, e i progettisti si sono resi conto che l'ereditarietà impone anche dei vincoli, a volte inaccettabili. Il risultato è che oggi, entro certi limiti, l'ereditarietà viene utilizzata di meno.

La realizzazione di interfaccia è utile quando si vuole definire un contratto senza, però, essere disposti ad accettare i dettagli di implementazione ereditati. Se da un lato è vero che la realizzazione di interfaccia non offre alcuna possibilità di riuso del codice,

dall'altro, però, offre un meccanismo molto pulito per definire un contratto e garantire che certe classi lo rispettino. Dato che la realizzazione di interfaccia non consente di ereditare nulla, è sotto certi aspetti più flessibile e robusta dell'ereditarietà.

15.6 Template di classe

Fino a questo punto, per definire una classe di progettazione si è detto che è necessario specificare esplicitamente i tipi dei suoi attributi e i tipi dei parametri e del valore restituito di tutti i suoi metodi. Questo è giusto e funziona bene nella maggior parte delle situazioni, ma può a volte limitare la possibilità di riusare il codice.

La Figura 15.7 riporta un esempio in cui sono definite tre classi. Si tratta di tre classi di vettori dimensionati. La prima è un vettore dimensionato di elementi numerici interi (int), la seconda è un vettore dimensionato di elementi numerici a virgola mobile (double) e l'ultima è un vettore dimensionato di elementi alfanumerici (String). Esaminando queste tre classi si osserva che, se non fosse per il tipo di elemento contenuto nel vettore, sarebbero del tutto identiche. Eppure, nonostante questa affinità, si sono dovute definire tre classi distinte.

I template di classe consentono la parametrizzazione dei tipi. Questo significa che al posto di specificare il tipo effettivo degli attributi e dei parametri e valori restituiti dei metodi, è possibile definire una classe facendo riferimento a dei segnaposto o dei parametri. Questi possono essere sostituiti da dei valori effettivi per creare delle nuove classi. La Figura 15.8 illustra la definizione della classe *VettoreDimensionato* facendo riferimento ai parametri T (che è un classificatore, dato che non è specificato il tipo) e dim, che è un int.

<i>VettoreDimint</i>	<i>VettoreDimDouble</i>	<i>VettoreDimString</i>
Dim : int <code>elementi[] : int</code> <code>aggiungiElemento(e:int) : void</code> <code>getElemento(i:int) : int</code>	Dim : int <code>elementi[] : double</code> <code>aggiungiElemento(e:double) : void</code> <code>getElemento(i:int) : double</code>	Dim : int <code>elementi[] : String</code> <code>aggiungiElemento(e:String) : void</code> <code>getElemento(i:int) : String</code>

Figura 15.7

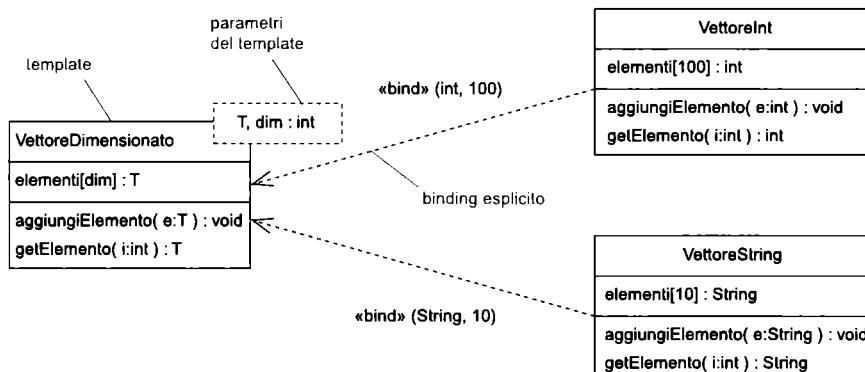


Figura 15.8

Associando a questi parametri formali dei valori specifici (operazione di “*binding*”) è possibile creare nuove classi: questa procedura viene detta istanziazione di un template. Si osservi che istanziando un template si ottiene una classe, mentre istanziando una classe si ottiene un oggetto.

È possibile istanziare un template utilizzando una relazione di dipendenza con lo stereotipo «bind»: in questo caso si parla di *binding esplicito*. Il Paragrafo 9.5, dove si discute delle dipendenze, conteneva un riferimento alla relazione «bind», rimandando però l’approfondimento a questo capitolo. Per istanziare un template è anche necessario specificare i valori effettivi che devono essere sostituiti ai parametri del template. Questi vengono elencati, tra parentesi graffe, subito dopo lo stereotipo «bind». Quando viene istanziato il template questi valori vengono sostituiti ai parametri del template e si ottiene così la nuova classe.

Si tratta ovviamente di un meccanismo di riuso molto potente: è possibile definire una classe molto generica come template, creando poi molte versioni specializzate di questa classe effettuando il *binding* con valori effettivi appropriati.

Nella Figura 15.8, il *binding* viene in realtà utilizzato in due diversi modi. Il parametro *T* richiede come valore un classificatore. Questo perché se non si associa alcun tipo al parametro di un template, si sottintende il tipo classificatore. Il parametro *dim* richiede, invece, un valore effettivo intero. Questo ci permette di specificare le dimensioni del vettore dimensionato come parametro del template.

I nomi dei parametri di un template sono locali al template stesso. Questo significa che se due template hanno entrambi un parametro *T*, si tratta di un diverso parametro *T* per ciascun template.

Esiste una sintassi alternativa per specificare l’istanziazione di un template, il *binding implicito*. Con questa sintassi non si utilizza una dipendenza esplicita di «bind», ma si usa un particolare formato nel nome della classe istanziata. Per istanziare un template implicitamente è sufficiente utilizzare al posto del nome della classe, il nome del template di classe, seguito dall’elenco dei valori effettivi dei parametri racchiusi tra parentesi angolari (<>), così come illustrato nella Figura 15.9. Lo svantaggio del *binding implicito* è che la classe istanziata non può veramente avere un suo nome.

Consigliamo l’utilizzo del *binding esplicito*, il quale consente alle classi istanziate da template di avere un loro nome descrittivo.

Anche se i template di classe sono una meccanismo molto potente, a oggi l’unico linguaggio OO comunemente usato che li supporti è il C++ (vedere la Tabella 15.1). Ovviamente i template non devono essere utilizzati durante la progettazione, se il linguaggio di sviluppo non li supporta.

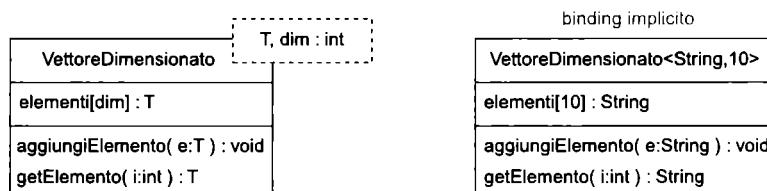


Figura 15.9

Tabella 15.1

Linguaggio	Supporto per i template
Java	No
C++"	Sì
Smalltalk	No
Python	No
Visual Basic	No
C#	No

15.7 Classi annidate

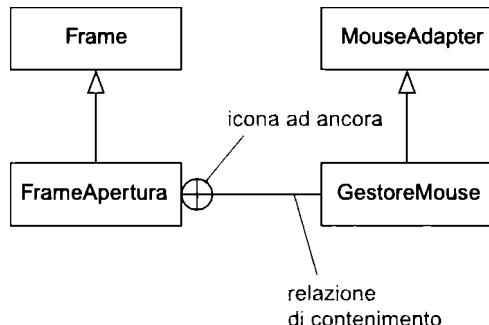
Alcuni linguaggi (Java, per esempio), consentono di definire una classe all'interno della definizione di un'altra classe. Questa prima classe viene chiamata classe annidata. In Java è anche nota come classe inner (interna).

Una classe annidata è dichiarata dentro alla dichiarazione di una seconda classe, detta classe esterna, ed è accessibile *solo* alla classe esterna e agli oggetti di quella classe. Solo la classe esterna o le sue istanze possono creare e utilizzare istanze di una classe annidata.

Le classi annidate possono essere utilizzate in Java per la gestione degli eventi. L'esempio in Figura 15.10 illustra una semplice classe di finestra di interfaccia utente chiamata FrameApertura. Eredita il comportamento da finestra dalla sua classe genitore Frame. FrameApertura ha una classe annidata GestoreMouse che eredita la capacità di gestire gli eventi del *mouse* dalla sua classe genitore MouseAdapter.

Ogni istanza di FrameApertura utilizza un'istanza di GestoreMouse per processare gli eventi del *mouse*. Per ottenere questo effetto, l'istanza di FrameApertura deve:

- creare un'istanza di GestoreMouse;
- impostare il GestoreMouse come proprio ascoltatore di eventi del *mouse*.

**Figura 15.10**

Questo approccio apporta notevoli benefici, in quanto tutto il codice di gestione del *mouse* si ritrova completamente encapsulato nella classe *GestoreMouse*.

15.8 Riepilogo

Le classi di progettazione sono gli elementi fondamentali del modello della progettazione. Sono stati spiegati i seguenti concetti.

1. Le classi di progettazione sono classi le cui specifiche sono talmente complete da permettere la loro implementazione.
2. Le classi di progettazione hanno due origini:
 - il dominio dei problemi:
 - sono una rifinitura delle classi di analisi;
 - una classe di analisi può evolversi in nessuna, una o più classi di progettazione;
 - il dominio delle soluzioni:
 - librerie di classi di utilità;
 - *middleware*;
 - librerie GUI;
 - componenti riusabili;
 - dettagli specifici dell'implementazione.
3. Le classi di progettazione hanno specifiche complete:
 - l'insieme completo degli attributi:
 - nome;
 - tipo;
 - valore predefinito (opzionale);
 - visibilità.
 - l'insieme completo dei metodi:
 - nome;
 - nomi e tipi di tutti i parametri;
 - valori dei parametri opzionali (opzionale);
 - tipo di valore restituito;
 - visibilità.

4. Le classi di progettazione ben formate devono avere le seguenti caratteristiche:
 - i metodi pubblici della classe definiscono un contratto tra la classe e i suoi clienti;
 - completezza: la classe fa almeno tutto ciò i suoi clienti si aspettano che faccia;
 - sufficienza: la classe non fa nulla oltre ciò che i suoi clienti si aspettano che faccia;
 - essenzialità: i servizi offerti sono primitivi, semplici, atomici e distinti;
 - massima coesione:
 - ogni classe deve rappresentare un unico concetto astratto ben definito;
 - tutti i metodi devono supportare l'unico compito della classe;
 - minima interdipendenza:
 - una classe non deve avere interdipendenze con altre classi oltre a quelle che le servono per occuparsi delle proprie responsabilità;
 - due classi devono essere interdipendenti solo se tra loro esiste una correlazione semantica;
 - l'interdipendenza finalizzata al riuso di codice deve essere evitata.
 - Una classe di progettazione deve sempre essere considerata dal punto di vista dei suoi clienti.
5. Ereditarietà.
 - L'ereditarietà deve essere utilizzata esclusivamente se tra due classi esiste un'ovvia relazione “è un/una” oppure per riusare codice (ma in questo caso è necessario evitare di introdurre interdipendenze).
 - Svantaggi:
 - l'ereditarietà è la forma più forte di interdipendenza tra due classi;
 - l'incapsulazione all'interno di una gerarchia ereditaria è debole e questo può causare il problema della “fragilità della classe base”: modifiche alla classe base impattano su tutta la gerarchia;
 - l'ereditarietà in molti linguaggi non è una relazione flessibile: la relazione viene decisa in fase di compilazione e non è modificabile a *run-time*.
 - Le sottoclassi dovrebbero rappresentare sempre una relazione “è un tipo di”, e non “è un ruolo interpretato da”: quest'ultima andrebbe rappresentata con un'aggregazione.
6. L'ereditarietà multipla consente a una classe di avere più di una classe genitore.
 - Di tutti i linguaggi OO comunemente usati, solo il C++ supporta l'ereditarietà multipla.

- Punti di attenzione:
 - le diverse classi genitore devono essere semanticamente disgiunte tra loro;
 - deve esistere una relazione “è un tipo di” tra la classe e ciascuna delle sue classi genitore;
 - il principio di sostituibilità deve essere applicabile tra la classe e ciascuna delle sue classi genitore;
 - le classi genitore non dovrebbero avere antenati in comune;
 - efficace per le classi mixin: semplici classi progettate per essere incorporate in altre classi utilizzando l’ereditarietà multipla; si tratta di una tecnica sicura e potente.

7. Ereditarietà e realizzazione di interfaccia.

- Ereditarietà:
 - fornisce l’interfaccia: i metodi pubblici;
 - fornisce l’implementazione: gli attributi, le associazioni e i membri protetti.
- Realizzazione di interfaccia: fornisce solo l’interfaccia.
- Usare l’ereditarietà quando si vuole ereditare anche l’implementazione.
- Usare la realizzazione di interfaccia quando si vuole solo definire un contratto.

8. Template di classe.

- Di tutti i linguaggi OO comunemente usati, solo il C++ supporta i template di classe.
- I template consentono la “parametrizzazione” di un tipo: si crea un template definendo un tipo sotto forma di parametri formali e si istanzia un template associando valori effettivi a tali parametri.
 - Il *binding* esplicito utilizza una dipendenza con stereotipo «bind»:
 - i valori effettivi sono riportati sulla relazione;
 - ogni classe istanziata dal template può avere un suo nome.
 - Il *binding*隐式的:
 - i valori effettivi sono riportati nel nome della classe, racchiusi tra parentesi angolari (<>);
 - le classi istanziate dal template non possono avere un loro nome: il nome è composto dal nome del template e dall’elenco degli argomenti.

9. Classi annidate:

- una classe annidata viene definita all’interno della definizione di un’altra classe;

- la classe annidata esiste solo all'interno della classe esterna: solo la classe esterna può creare e utilizzare istanze della classe annidata;
- le classi annidate sono anche note in Java come classi inner, e vengono comunemente utilizzate per gestire gli eventi all'interno di classi di interfaccia utente GUI.

Rifinitura delle relazioni di analisi

16.1 Contenuto del capitolo

Il capitolo descrive le tecniche per rifinire le relazioni di analisi e trasformarle in relazioni di progettazione. La prima parte del capitolo discute della conversione di relazioni di analisi in relazioni del tipo tutto-parte: aggregazione (Sezione 16.4) o composizione (Sezione 16.5). La seconda parte del capitolo spiega come gestire le molteplicità delle associazioni di analisi. Sono descritte tecniche specifiche per la rifinitura di associazioni di analisi con molteplicità uno-a-uno (Sezione 16.7), molti-a-uno (Sezione 16.8), uno-a-molti (Sezione 16.9) e molti-a-molti (Sezione 16.11.1). Sono, inoltre, trattate le associazioni bidirezionali (Sezione 16.11.2) e le classi associazione (Sezione 16.11.3).

16.2 Relazioni di progettazione

Quando si giunge alla progettazione, è necessario rifinire le relazioni tra le classi di analisi e trasformarle in relazioni tra classi di progettazione. Molte classi individuate e fissate durante l'analisi non sono direttamente implementabili, così come sono state definite. Per esempio, nessuno dei linguaggi OO comunemente usati supporta direttamente le associazioni bidirezionali, le classi associazione o le associazioni molti-a-molti. Nel modello della progettazione, è necessario specificare *come* questi tipi di associazioni debbano essere realizzati. Per ottenere le associazioni di progettazione a partire dalle associazioni di analisi, è necessario operare una rifinitura che prevede diverse procedure:

- trasformare le associazioni in relazioni di aggregazione o composizione dove applicabile;
- implementare le classi associazione;
- implementare le associazioni uno-a-uno;
- implementare le associazioni molti-a-molti;
- implementare le associazioni bidirezionali.

**Figura 16.1**

Tutte le associazioni di progettazione *devono* specificare:

- la navigabilità;
- la molteplicità di entrambi gli estremi. Tutte le associazioni di progettazione *dovrebbero*, inoltre, specificare il nome di un ruolo, almeno sull'estremo destinazione.

16.3 Aggregazione e composizione

Nella progettazione, una relazione di associazione può essere rifinita in una relazione di aggregazione o in una forma di aggregazione forte, nota come relazione di aggregazione compositiva. Normalmente, ci si riferisce all'aggregazione compositiva più semplicemente come composizione.

Per capire bene la differenza tra questi due tipi di aggregazione, è sufficiente ragionare su alcuni esempi del mondo reale.

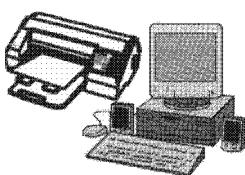
- Aggregazione: è una relazione tra oggetti poco forte: un computer e le sue periferiche sono un esempio di aggregazione.
- Composizione: è una relazione tra oggetti molto forte: un albero e le sue foglie sono un esempio di composizione.

Si esaminino più a fondo questi due esempi illustrati nella Figura 16.2. Un computer ha una relazione debole con le proprie periferiche. Queste periferiche possono essere o non essere collegate al computer o possono essere condivise da più computer e non fanno, quindi, propriamente parte di uno specifico computer: questa è aggregazione. D'altra parte, un albero ha invece una relazione molto più forte con le proprie foglie. Le foglie fanno parte di un solo albero e non possono essere condivise tra diversi alberi. Se un albero muore, si porta dietro anche le sue foglie: questa è composizione.

È consigliabile tenere a mente queste semplici analogie mentre si legge il resto del

L'UML definisce due tipi di associazioni

Aggregazione



Alcuni oggetti sono debolmente associati, come un computer e le sue periferiche

Composizione



Alcuni oggetti sono fortemente associati, come un albero e le sue foglie

Figura 16.2

capitolo, dove si tratterà in modo approfondito della semantica dell'aggregazione e della composizione.

16.4 Semantica dell'aggregazione

L'aggregazione è una relazione del tipo tutto–parte per cui un aggregato è costituito da molte parti. In una relazione tutto–parte, un oggetto (il *Tutto*) utilizza i servizi di un altro oggetto (la *Parte*). Il *Tutto* tende, quindi, a essere l'elemento che domina e controlla la relazione, mentre la *Parte* di solito si limita a servire le richieste del tutto, ed è quindi più passiva. In effetti, se la relazione è navigabile solo dal *Tutto* alla *Parte*, allora la *Parte* non sa neanche di essere in relazione con il *Tutto*.

L'aggregazione riportata nella Figura 16.3 mostra che:

- un Computer può essere collegato a 0 o più Stampanti;
- in *un qualunque istante* una Stampante è collegata a 0 o 1 Computer;
- *nel corso del tempo*, molti Computer possono utilizzare la stessa Stampante;
- la Stampante può esistere anche senza essere collegata a alcun Computer;
- la Stampante è indipendente dal Computer.

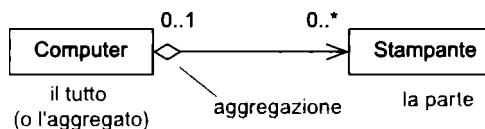


Figura 16.3

La semantica dell'aggregazione può essere così sintetizzata:

- l'aggregato può in alcuni casi esistere indipendentemente dalle parti, ma in altri casi no;
- le parti possono esistere indipendentemente dall'aggregato;
- l'aggregato è in qualche modo incompleto se mancano alcune delle sue parti;
- è possibile che più aggregati condividano una stessa parte.

L'aggregazione è transitiva. Esaminando la Figura 16.4, questo vuol dire che se C è una parte di B e B è una parte di A, allora C è anche una parte di A.

L'aggregazione è asimmetrica. Questo significa che un oggetto non potrà mai essere, direttamente o indirettamente, una parte di se stesso. Nell'esempio di aggregazione illustrato nella Figura 16.5, si vede che gli oggetti Prodotto possono aggregare altri oggetti Prodotto: questo va bene a patto che si tratti di oggetti *distinti*.

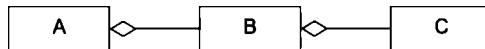
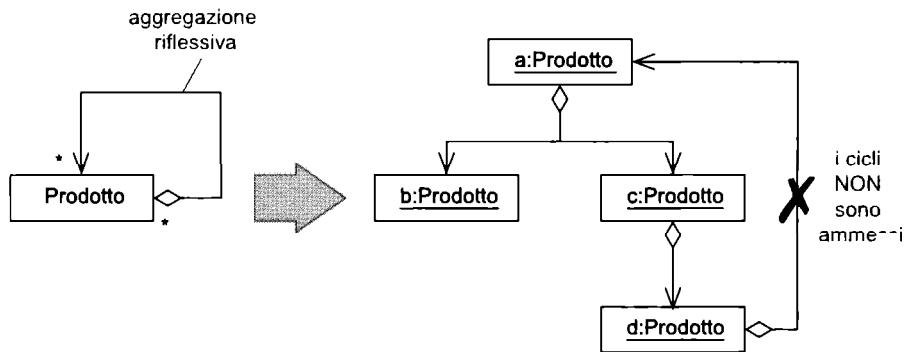
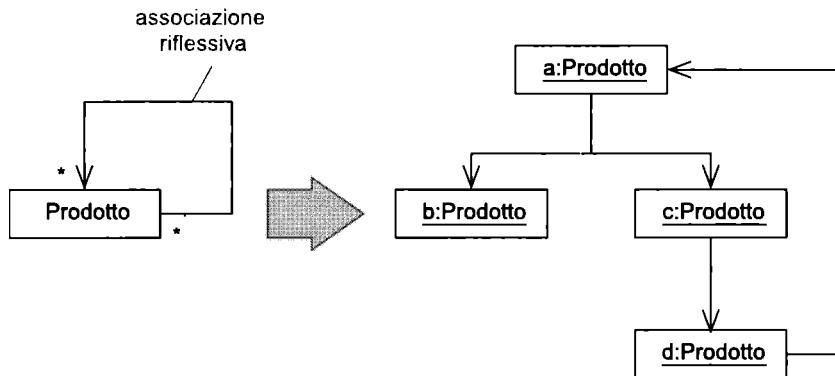
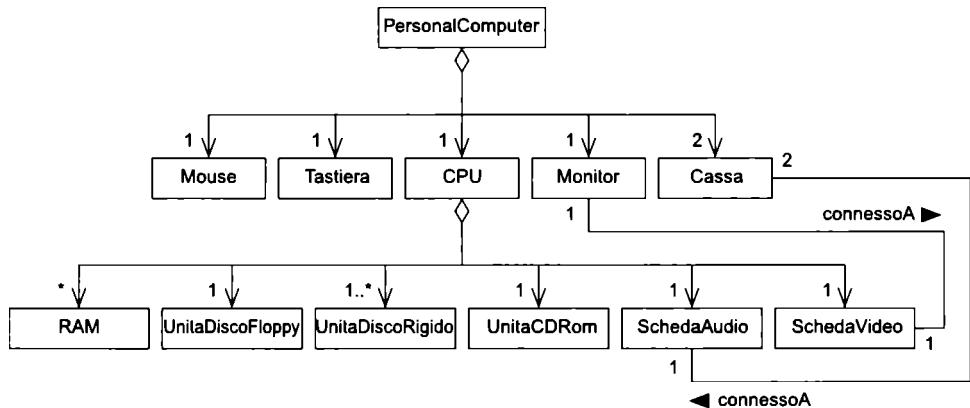


Figura 16.4



Prendendo ancora in esame la Figura 16.5, può succedere che sia necessario modellare la situazione illustrata, in cui l'oggetto d ha un riferimento all'oggetto a. Questo capita se l'oggetto d ha necessità di *referenziare* l'oggetto aggregato a, o di utilizzare alcuni dei suoi servizi. Ma come si può modellare questa situazione nel diagramma delle classi? La relazione di aggregazione riflessiva sulla classe Prodotto non va bene, in quanto il vincolo di asimmetria dell'aggregazione impedisce all'oggetto a di essere, direttamente o indirettamente, parte di se stesso. È, quindi, necessario introdurre un'*associazione riflessiva* generica sulla classe Prodotto che, in quanto generica, può gestire la relazione richiesta tra gli oggetti d e a, come illustrato nella Figura 16.6.



**Figura 16.7**

L'esempio della Figura 16.7 mostra un altro tipico esempio di aggregazione. È possibile modellare un personal computer (il *Tutto*) come un insieme di parti. Queste parti non hanno una relazione forte con il *Tutto*, in quanto sono intercambiabili con altri computer e, quindi, la semantica dell'aggregazione sembra adatta a questo modello. Il modello dice che un PersonalComputer può essere considerato un aggregato delle seguenti parti: un Mouse, una Tastiera, una CPU, un Monitor e due Casse. La CPU può a sua volta essere modellata come un aggregato di diversi componenti hardware, quali la RAM, una UnitaDiscoRigido ecc.

16.5 Semantica della composizione

La composizione è una forma più forte di aggregazione e ha una semantica del tutto simile (ma soggetta a vincoli più rigidi). Come l'aggregazione, si tratta di una relazione tutto-parte, transitiva e asimmetrica.

La differenza principale tra aggregazione e composizione è che nella composizione la Parte non ha una propria esistenza indipendente al di fuori del *Tutto*. Inoltre, nella composizione ogni Parte appartiene esclusivamente a un solo *Tutto*, mentre nell'aggregazione una Parte può essere condivisa da più *Tutti*.

Nell'esempio riportato nella Figura 16.8, gli oggetti Pulsante non hanno una propria esistenza indipendente dall'oggetto Mouse di cui fanno parte. Distruggendo l'oggetto Mouse, vengono distrutti anche i suoi oggetti Pulsante, che ne costituiscono parte integrante. Ogni oggetto Pulsante può appartenere esclusivamente a un solo oggetto Mouse. È proprio come per gli alberi e le foglie: l'esistenza della foglia è determinata dall'esistenza dell'albero, e ogni foglia può appartenere esclusivamente a un unico albero.

La semantica della composizione può essere così sintetizzata:

- ogni parte può appartenere a un solo composito per volta;
- il composito è l'unico responsabile di tutte le sue parti: questo vuol dire che è responsabile della loro creazione e distruzione;

- il composito può anche rilasciare una sua parte, a patto che un altro oggetto si prenda la relativa responsabilità;
 - se il composito viene distrutto, deve distruggere tutte le sue parti o cederne la responsabilità a qualche altro oggetto.

Dato che il composito è l'unico responsabile per il ciclo di vita delle sue parti, creando un composito esso creerà le sue parti. Similmente, distruggendo un composito esso deve distruggere tutte le sue parti o affidarle a un altro composito che ne accetti la responsabilità.

Un'altra differenza tra l'aggregazione e la composizione è che l'aggregazione può essere utilizzata per formare gerarchie *e* reti riflessive, mentre la composizione consente solo le gerarchie riflessive. Questo perché nella composizione un oggetto può far parte di *un solo* composito per volta.

L'UML fornisce un sintassi "annidata", alternativa, per la composizione, la quale illustra la natura della composizione in modo più grafico. Non esiste alcuna differenza semantica tra le due sintassi per la composizione. La Figura 16.9 le illustra entrambe. La sintassi ad albero è più diffusa, in quanto è più facile da comprendere e da disegnare.

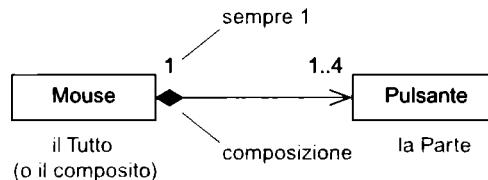


Figura 16.8

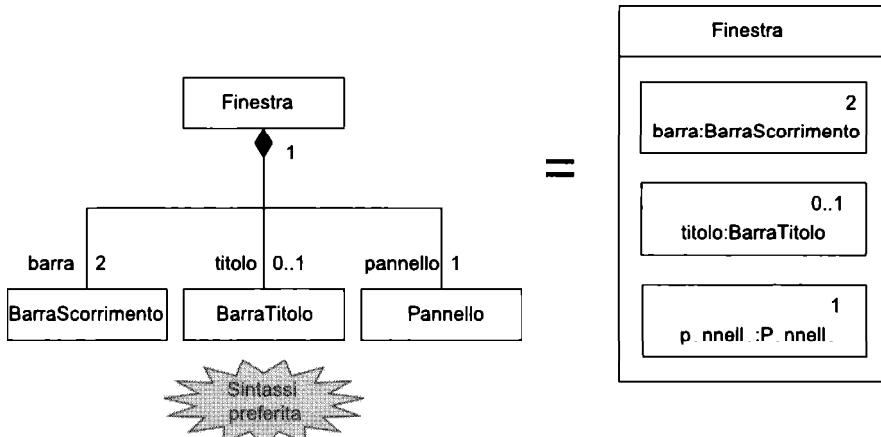


Figura 16.9

16.5.1 Composizione e attributi

Ragionando sulla semantica della composizione ci si rende conto di quanto sia simile alla semantica degli attributi. Sia le parti di una composizione, sia gli attributi, hanno un ciclo di vita controllato dall'oggetto a cui appartengono e non possono esistere al di fuori di esso. In effetti, gli attributi sono del tutto equivalenti a una relazione di composizione tra la classe a cui appartengono e la classe degli attributi stessi. Ma allora perché abbiamo bisogno di due modi per esprimere lo stesso concetto? Ci sono due motivi.

- Gli attributi possono essere dei tipi di dati primitivi. Alcuni linguaggi OO ibridi, come C++ e Java, hanno tipi di dati primitivi, come `int` e `double`, che non sono classi. Si potrebbe modellare questi tipi come classi con lo stereotipo «primitivo», ma, a essere sinceri, servirebbe solo a rendere il modello troppo affollato. Questi tipi primitivi dovrebbero *sempre* essere modellati come attributi.
- Esistono alcune classi di utilità, quali `Time`, `Date` e `String` che sono utilizzate in modo molto pervasivo. Se ogni uso di queste classi venisse modellato con una relazione di composizione alla classe stessa, il modello risulterebbe completamente illeggibile. È molto meglio modellare queste classi come attributi.

In ultima analisi, dovendo modellare un tipo primitivo o una classe di utilità, o anche solo una classe che non risulta interessante ai fini del modello o del diagramma in questione, è meglio utilizzare un attributo e non una relazione di composizione. Non esiste una regola aurea in questo caso, ma bisogna tenere sempre presente che il modello deve essere soprattutto chiaro, utile e leggibile.

16.6 Come rifinire le relazioni di analisi

In analisi si utilizzano spesso associazioni semplici, senza entrare nel merito della semantica della relazione (o di come essa verrà infine implementata). Durante la progettazione è, invece, necessario specificare completamente le relazioni, rifinendo ciascuna associazione, laddove possibile, in una delle forme di aggregazione. In realtà, l'unico caso in cui è lecito utilizzare un'associazione in progettazione è quando sia *necessario* per evitare un ciclo nel grafo delle aggregazioni (vedere il Paragrafo 16.4). Questo capita di rado e, quindi, la maggior parte delle associazioni individuate durante l'analisi vengono poi trasformate in aggregazioni o composizioni.

Dopo aver verificato che è possibile utilizzare un'aggregazione o una composizione per rifinire un'associazione di analisi, è necessario procedere nel seguente modo:

- qualora non siano presenti, aggiungere la molteplicità e i nomi dei ruoli all'associazione;
- decidere quale estremo dell'associazione è il *Tutto*, e quale è la *Parte*;
- controllare la molteplicità all'estremo del *Tutto*: se è esattamente 1, allora si può utilizzare una composizione, altrimenti è *necessario* utilizzare un'aggregazione;
- aggiungere la navigabilità *dal* *Tutto* *alla* *Parte*: in progettazione le associazioni devono essere unidirezionali.

A questo punto, l'associazione è stata rifinita in un'aggregazione o in una composizione.

Se la molteplicità a uno degli estremi è maggiore di 1, allora è necessario stabilire come verrà implementata. Questo è il passo successivo dell'operazione di rifinitura.

16.7 Associazioni uno-a-uno

Quasi tutte le associazioni uno-a-uno vengono rifinite in relazioni di composizione. In effetti, un'associazione uno-a-uno implica una relazione talmente forte tra le due classi che è meglio valutare se sia possibile fondere le due classi in un'unica classe, senza violare le regole che si applicano alle classi, di progettazione (vedere il Paragrafo 15.4). Se non si possono fondere le due classi occorre rifinire la relazione come illustrato nella Figura 16.10.

Se B non è una classe particolarmente importante, si può anche valutare di specificare B come attributo di A.

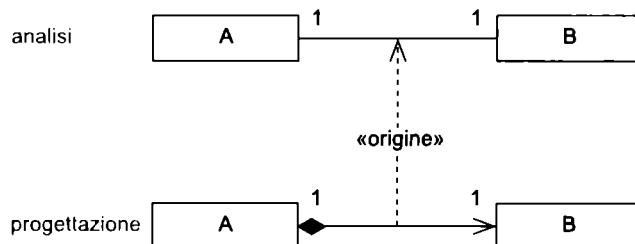


Figura 16.10

16.8 Associazioni molti-a-uno

Un'associazione molti-a-uno ha una molteplicità di molti all'estremo del Tutto, e di esattamente 1 all'estremo della Parte.

Dato che c'è una molteplicità di molti sul lato del Tutto, si capisce immediatamente che la composizione *non* è applicabile: la Parte è condivisa tra molti Tutti. Ma può, invece, essere applicabile l'aggregazione. Bisogna solo controllare che non nascano cicli nel grafo delle aggregazioni (vedere il Paragrafo 16.4): se così è, si può rifinire l'associazione di analisi come illustrato nella Figura 16.11.

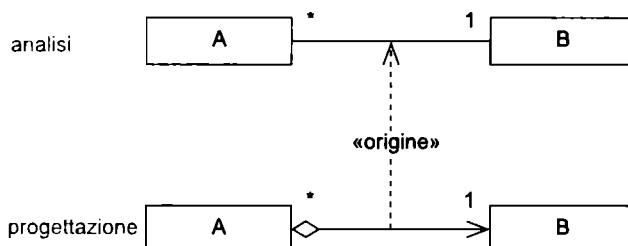


Figura 16.11

16.9 Associazioni uno-a-molti

In un'associazione uno-a-molti un *Tutto* ha un *insieme* omogeneo di oggetti *Parte*. Per poter implementare questo tipo di relazione, è necessario ricorrere a una classe contenitore, oppure utilizzare ciò che il linguaggio di programmazione offre come supporto nativo per gli insiemi di oggetti.

La maggior parte dei linguaggi OO offre poco supporto nativo per gestire insiemi di oggetti. In effetti, molti linguaggi mettono a disposizione dei programmatore soltanto i vettori (*array*). Un vettore è un insieme indicizzato di riferimenti a oggetti, solitamente limitato a una qualche dimensione massima. Il vantaggio dei vettori nativi è che sono solitamente molto veloci. Questa velocità ha però come contropartita una flessibilità decisamente inferiore a quella offerta da altri tipi di contenitore.

Le classi contenitore sono tipicamente molto più potenti e flessibili dei vettori e degli altri tipi di insiemi di oggetti nativi. Ne esistono di diverso tipo, con semantiche differenti. Il resto di questo capitolo approfondisce l'utilizzo delle classi contenitore nella progettazione.

16.10 Classi contenitore

Una classe contenitore è una classe le cui istanze sono specializzate nella gestione di un insieme di altri oggetti. Molti linguaggi mettono a disposizione una libreria standard che contiene classi contenitore (e altre classi di utilità).

Una delle chiavi per eccellere nella progettazione e nell'implementazione OO è la padronanza delle classi contenitore. Tutte le classi contenitore hanno metodi per:

- aggiungere oggetti all'insieme;
- rimuovere oggetti dall'insieme;
- recuperare il riferimento a un oggetto dell'insieme;
- iterare sull'insieme, ovvero ciclare su tutti gli oggetti dell'insieme, dal primo all'ultimo.

Esistono tante classi contenitore diverse e ciascuna è specializzata nel gestire un insieme di oggetti in un modo particolare. Un aspetto molto importante della progettazione e dell'implementazione OO, è saper scegliere la classe contenitore giusta per il compito giusto. La prossima sezione tratta questo argomento con maggior dettaglio.

Come esempio di uso di classe contenitore, la Figura 16.12 illustra un'associazione di analisi uno-a-molti che viene rifinita e implementata con una classe contenitore chiamata `Vector`. Si tratta di una classe Java contenuta nella libreria standard `java.util`. La relazione tra il *Tutto* (A) e la classe `Vector` è di solito una composizione, in quanto la classe `Vector` tipicamente è parte integrante dell'implementazione del *Tutto* e non ha alcun altro compito. Tuttavia la relazione tra la classe `Vector` e le parti (B) può essere o un'aggregazione o una composizione. Se A deve essere responsabile della creazione e della distruzione delle sue parti, allora è più indicato l'uso della composizione. In caso contrario si utilizza l'aggregazione.

Per quanto riguarda la modellazione con le classi contenitore, queste sono le strategie di base.

- Modellare esplicitamente anche la classe contenitore: questa è la situazione riportata nella Figura 16.12. Questa soluzione ha il vantaggio che è molto esplicita, ma ha anche lo svantaggio che le classi contenitore appesantiscono il modello della progettazione che risulta spesso poco leggibile. Sostituendo a ogni associazione uno-a-molti una classe contenitore, il modello si riempie molto in fretta. La scelta della classe contenitore è, inoltre, una decisione tattica che concerne l'implementazione e che può essere tranquillamente delegata agli sviluppatori. Si dovrebbe sostituire l'associazione uno-a-molti con una classe contenitore esplicita *solo* se la scelta della classe contenitore è effettivamente strategica.
- Specificare nello strumento CASE come debba essere implementata ciascuna associazione uno-a-molti. Molti strumenti CASE consentono di assegnare a ciascuna associazione uno-a-molti una specifica classe contenitore. Di solito, questo si fa aggiungendo valori etichettati all'associazione per specificare le proprietà per la generazione del codice della relazione. Questa tecnica è illustrata nella Figura 16.13, dove è stata aggiunta la proprietà {Vector} a un estremo della relazione. Si osservi che nel valore etichettato è stato specificato esclusivamente il nome: il valore non avrebbe alcun senso in questo caso.
- Specificare la semantica dell'insieme senza però specificare nessuna classe contenitore per l'implementazione. È importante non esagerare quando si devono modellare insiemi di oggetti. Come già detto, in molti casi la scelta del tipo specifico di contenitore da utilizzare è più tattica che strategica e può essere delegata al programmatore, che prenderà una decisione quando sarà giunto il momento di implementare la relazione. Tuttavia questa tecnica di solito non consente di avere il codice generato automaticamente.

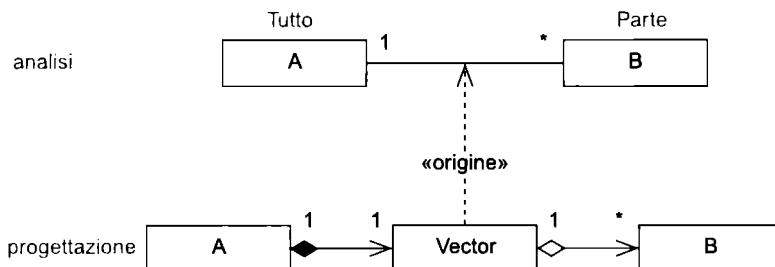
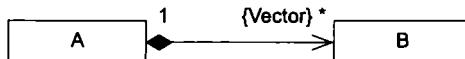
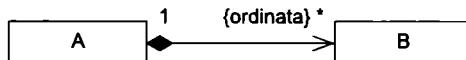


Figura 16.12

L'UML mette a disposizione due proprietà standard {ordinata} e {non-ordinata} che possono essere applicate agli estremi di una relazione per indicare se la classe contenitore debba mantenere l'insieme di oggetti ordinato o meno. Se, nell'esempio riportato nella Figura 16.13, servisse avere un insieme ordinato di oggetti di classe B, allora si potrebbe modellare la relazione come illustrato nella Figura 16.14.

**Figura 16.13****Figura 16.14**

Oltre a {ordinata} e {non-ordinata}, è possibile utilizzare altre proprietà non standard per indicare il comportamento richiesto alla classe contenitore. La Tabella 16.1 riporta alcuni esempi utili e significativi.

Dato che si tratta di proprietà non standard, è necessario assicurarsi che siano documentate altrove nel modello.

Tabella 16.1

Proprietà	Semantica
{ordinata per chiave}	L'insieme è ordinato secondo qualche chiave: è anche possibile specificare la chiave nella proprietà, per esempio {ordinata per nome}
{indicizzata}	Ogni elemento dell'insieme può essere indirizzato tramite un indice numerico
{insieme}	Il contenitore non ammette duplicati
{lifo} o {stack}	"Last in, first out" (Ultimo a entrare, primo a uscire) – uno stack – l'ultimo elemento collocato sullo stack è anche il primo che può esserne rimosso
{fifo} o {coda}	"First in, first out" (Primo a entrare, primo a uscire) – una coda – il primo elemento collocato nella coda è anche il primo elemento che può esserne rimosso

16.10.1 Classi contenitore semplici dell'OCL

Il Linguaggio dei Vincoli per gli Oggetti (Object Constraint Language, OCL) è un linguaggio formale per esprimere vincoli: fa parte delle specifiche dell'UML 1.4. Non è molto diffuso (e quindi non verrà trattato affatto in questo libro), ma offre un insieme generico di classi contenitore, un valido e utile esempio dei tipi di contenitore comune mente resi disponibili dai diversi linguaggi OO.

Le classi contenitore dell'OCL sono Set, Bag e Sequence. Può essere utile far riferimento alle classi contenitore dell'OCL verso la fine delle attività di analisi, o all'inizio di quelle di progettazione, quando è necessario specificare la semantica dei contenitori, ma non si vuole ancora introdurre dipendenze con il linguaggio di implementazione.

La Tabella 16.2 illustra la semantica delle diverse classi contenitore dell'UML.

Tabella 16.2

Classe contenitore	Ammette duplicati	Indicizzata	Ordinata
Set	No	No	No
Bag	Sì	No	No
Sequence	Sì	Sì	Sì

- Set si comporta in modo simile a un insieme matematico: non ammette la presenza di oggetti duplicati. Gli elementi dell'insieme non sono indicizzati e la loro posizione all'interno dell'insieme è indefinita.
- Bag è meno rigido di Set, in quanto accetta la presenza di oggetti duplicati. Per il resto la semantica è identica.
- Anche Sequence accetta oggetti duplicati, ma è un insieme indicizzato e ordinato. Questo significa che gli oggetti contenuti in un Sequence hanno un indice, come gli elementi di un vettore, e che la posizione relativa degli elementi dell'insieme non muta nel tempo.

16.10.2 La mappa

Un altro tipo di classe contenitore molto utile è la mappa, nota anche come dizionario. Queste classi si comportano in modo simile a una tabella di *database* con due sole colonne: la chiave e il valore. Le mappe sono strutturate in modo che, data una chiave, si può reperire velocemente il valore associato. La mappa è un ottima classe contenitore, quando c'è l'esigenza di avere un insieme di oggetti a cui si deve accedere utilizzando un valore chiave univoco, oppure quando si vuole costruire un indice per l'accesso veloce ad altri contenitori.

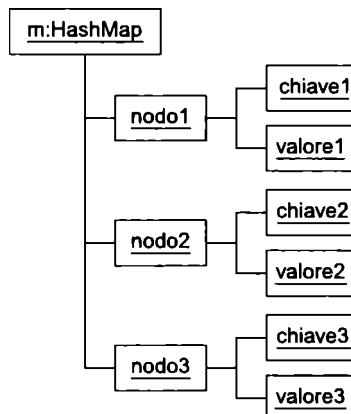


Figura 16.15

Sfortunatamente le mappe non fanno parte dell'OCL, quindi per usarle è necessario specificare un'implementazione dipendente dal linguaggio usato per l'implementazione.

Le mappe di solito si basano internamente su un insieme di nodi, dove a ogni nodo corrisponde una coppia di oggetti: l'oggetto chiave e l'oggetto valore. Sono ottimizzate in modo da riuscire a reperire velocemente un oggetto valore, dato l'oggetto chiave corrispondente.

La Figura 16.15 illustra una rappresentazione semplificata di una `HashMap` di Java. Il reperimento di un valore, data la corrispondente chiave, è molto veloce, in quanto l'insieme è indicizzato tramite l'uso di una *hash table*.

16.11 Relazioni reificate

Alcuni tipi di relazioni sono solo artefatti dell'analisi e non sono supportati da nessuno dei linguaggi OO comunemente utilizzati. La reificazione è il processo con cui si implementano queste relazioni di analisi. I seguenti tipi di relazioni di analisi richiedono una reificazione:

- associazioni molti-a-molti;
- associazioni bidirezionali;
- classi associazione.

16.11.1 Associazioni molti-a-molti

Le associazioni molti-a-molti non sono direttamente supportate da nessuno dei linguaggi OO comunemente utilizzati (anche se alcuni *databases* a oggetti le supportano direttamente) e devono, quindi, essere reificate in classi, aggregazioni, composizioni e dipendenze normali. Durante l'analisi si può essere piuttosto vaghi nel descrivere dettagli quali l'appartenenza e la navigabilità, ma quando si passa alla progettazione queste ambiguità devono essere risolte. Per fare questo, è necessario innanzitutto decidere quale delle due classi coinvolte sia il Tutto e quale sia la Parte, quindi si dovrà utilizzare l'aggregazione o la composizione in modo opportuno.

Nell'esempio della Figura 16.16, basandosi sui requisiti del sistema, si è deciso che la classe **Risorsa** è il Tutto. Questo perché il sistema ha come scopo quello di gestire la lavorazione associata alle Risorse, ovvero è incentrato sulle risorse. Se il sistema fosse stato incentrato sulle attività, allora avremmo utilizzato la classe **Attività** come Tutto, rovesciando la relazione della Figura 16.16. (Si osservi che non è buona abitudine utilizzare lettere accentate nei nomi delle classi e degli attributi, soprattutto durante la progettazione, in quanto, oltre a violare le regole di nomenclatura dell'UML, violano anche quelle della maggior parte dei linguaggi di implementazione - N.d.T.)

Si è anche deciso che la **Risorsa** sarà responsabile del ciclo di vita dei propri oggetti **Allocazione**, e si è quindi utilizzata una relazione di composizione.

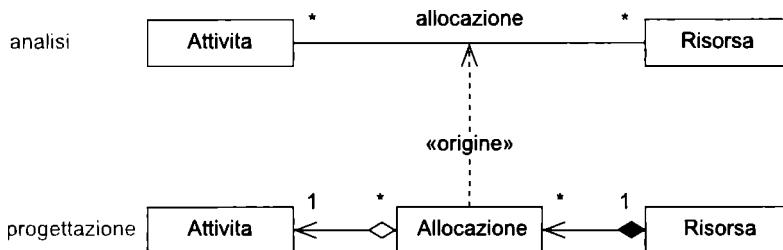


Figura 16.16

Se il sistema avesse un punto di vista più equidistante, allora si potrebbe dire che è incentrato sulle allocazioni (delle risorse alle attività). In questo caso, avremmo introdotto un nuovo oggetto (magari un *GestoreAllocazioni*), il quale avrebbe dovuto mantenere una lista di oggetti Allocazione ciascuno dei quali referenzia una Risorsa e una Attività.

16.11.2 Associazioni bidirezionali

Capita spesso di dover modellare la situazione in cui un oggetto a di classe A utilizza i servizi di un oggetto b di classe B, e che l'oggetto b ha esigenza a sua volta di effettuare un *callback* sull'oggetto a per usarne qualche servizio. Un tipico esempio è quello di un controllo GUI di classe Finestra che è composto da uno o più oggetti Pulsante, dove ciascun oggetto Pulsante ha bisogno di richiamare la Finestra di appartenenza per usufruire di qualche suo servizio.

In analisi questa è una situazione semplice da risolvere: è sufficiente modellare un'unica associazione bidirezionale. Tuttavia in progettazione è necessario trovare una soluzione migliore, in quanto nessuno dei linguaggi OO comunemente utilizzati supporta vere associazioni bidirezionali. Bisogna, quindi, reificare l'associazione bidirezionale in due distinte relazioni o dipendenze unidirezionali, come illustrato nella

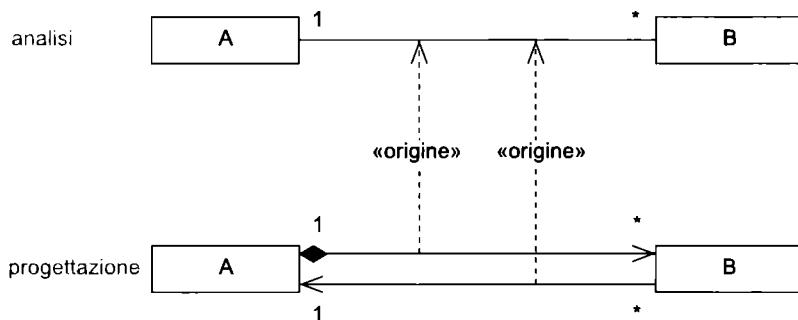


Figura 16.17

Figura 16.17.

Quando si modella un *callback* è necessario tener presente il vincolo di asimmetria a cui sono soggette le aggregazioni e le composizioni: un oggetto non può *mai* essere, direttamente o indirettamente, parte di se stesso. Questo vuol dire che se la classe A ha una relazione di aggregazione o composizione verso la classe B, la relazione di *callback* dalla classe B alla classe A deve essere modellata come un'associazione non rifinita. Se si usasse, infatti, una relazione di aggregazione anche dalla classe B alla classe A, allora l'oggetto *b* sarebbe una parte dell'oggetto *a* (per aggregazione o composizione) e l'oggetto *a* sarebbe una parte dell'oggetto *b* (per aggregazione). Questo ciclo nella gerarchia violerebbe in modo ovvio il vincolo di asimmetria dell'aggregazione.

Esistono anche associazioni bidirezionali in cui il Tutto passa un riferimento a se stesso come parametro di uno dei metodi della Parte, o dove la Parte istanzia direttamente il Tutto in uno dei propri metodi. Questi casi sono modellati più correttamente da una dipendenza che non da un'associazione.

16.11.3 Classi associazione

Le classi associazione sono degli artefatti puri dell'analisi che non trovano riscontro in nessuno dei linguaggi di programmazione OO comunemente utilizzati. Non corrispondono a nulla che si possa modellare in progettazione e devono, quindi, essere rimosse dal modello della progettazione.

Le classi associazione vengono reificate tramite l'uso di classi normali e una combinazione di associazioni, aggregazioni, composizioni o persino dipendenze,

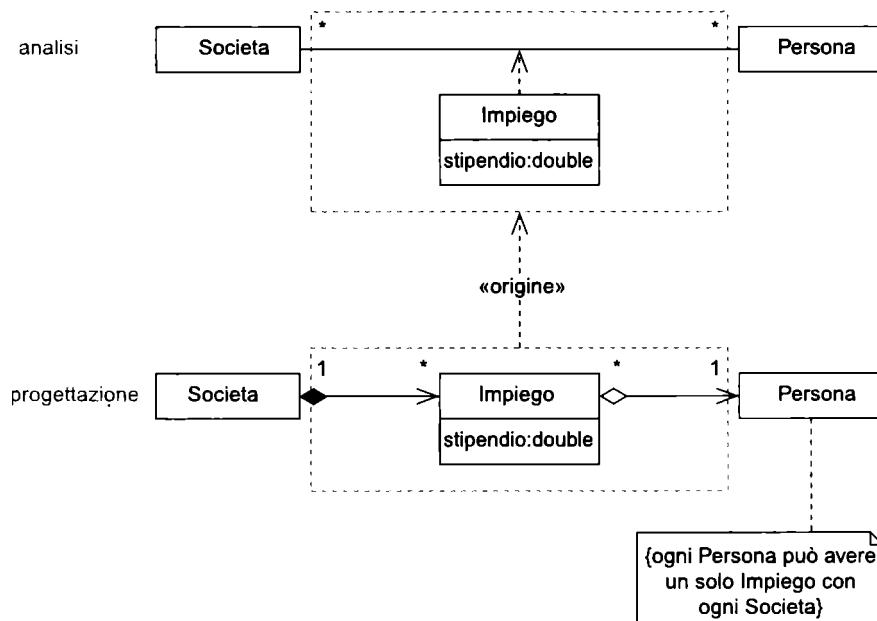


Figura 16.18

che ne fissi la semantica. Può essere necessario aggiungere dei vincoli al modello. Si comincia decidendo quale estremo dell'associazione costituisca il Tutto, quindi si specificano opportunamente composizione, aggregazione e navigabilità. La Figura 16.18 illustra un esempio.

Si osservi che quando si reifica una classe associazione, si perde parte della semantica. In particolare, si perde il vincolo che dichiara che i due oggetti associati devono formare una coppia univoca (vedere il Paragrafo 9.4.5). La Figura 16.18 illustra come sia possibile preservare questa semantica, aggiungendo una nota che riporta il vincolo appropriato.

16.12 Riepilogo

Questo capitolo ha spiegato come le relazioni dell'analisi vengono convertite in relazioni di progettazione, pronte per l'implementazione. Sono stati spiegati i seguenti concetti.

1. La rifinitura di un'associazione di analisi in un'associazione di progettazione comporta le seguenti attività:
 - rifinitura delle associazioni in relazioni di aggregazione o composizione, laddove possibile;
 - implementazione delle classi associazione;
 - implementazione delle associazioni uno-a-molti;
 - implementazione delle associazioni molti-a-molti;
 - implementazione delle associazioni bidirezionali;
 - aggiunta della navigabilità;
 - aggiunta della molteplicità a entrambi gli estremi dell'associazione;
 - aggiunta di un ruolo a entrambi gli estremi dell'associazione, o perlomeno all'estremo destinazione dell'associazione.
2. Aggregazione e composizione.
 - Si tratta di tipi di relazioni tutto-parte in cui gli oggetti di una classe hanno il ruolo del Tutto (o aggregato o composto), mentre gli oggetti dell'altra classe hanno il ruolo di Parte:
 - il Tutto usufruisce dei servizi delle Parti; le Parti servono le richieste del Tutto;
 - il Tutto domina la relazione e controlla la Parte; la Parte tende a essere più passiva.
 - Si tratta di relazioni transitive: se C è parte di B e B è parte di A, allora C è anche parte di A.
 - Si tratta di relazioni asimmetriche:
 - un Tutto non può mai essere, direttamente o indirettamente, Parte di se stesso;

- non devono comparire cicli nel grafo delle aggregazioni.
- Esistono due tipi di relazioni di aggregazione:
 - aggregazione;
 - aggregazione compositiva: abitualmente chiamata, semplicemente, composizione.

3. Aggregazione.

- Semantica dell'aggregazione:
 - l'aggregato può, in alcune situazioni, esistere indipendentemente dalle sue parti; in altre situazioni questo non è possibile;
 - le parti possono esistere indipendentemente dall'aggregato;
 - l'aggregato è in qualche modo incompleto se mancano alcune delle sue parti;
 - è possibile che aggregati distinti condividano una o più parti tra loro;
 - possono esistere sia gerarchie di aggregazione, sia reti di aggregazione;
 - il tutto conosce le sue parti, ma se la relazione è unidirezionale, dal Tutto alla Parte, allora le Parti non possono conoscere il Tutto a cui sono associate.
- L'aggregazione è simile alla relazione che esiste tra un personal computer e le proprie periferiche:
 - la relazione tra il computer e le proprie periferiche è debole;
 - le periferiche possono essere connesse oppure no;
 - le periferiche possono essere condivise tra diversi computer;
 - le periferiche non “appartengono” veramente a uno specifico computer.

4. Composizione.

- Si tratta di una forma di aggregazione più forte:
 - le parti appartengono a un solo composito per volta;
 - il composito è l'unico responsabile del ciclo di vita delle sue parti: questo significa che è responsabile della loro creazione e della loro distruzione;
 - il composito può anche rilasciare le proprie parti a patto che la responsabilità venga trasferita a un altro oggetto;
 - quando si distrugge un composito, questo deve a sua volta distruggere tutte le sue parti, oppure cederne la responsabilità a un altro oggetto;
 - ogni parte appartiene a un unico composito, dunque possono esistere gerarchie di composizione: non possono, invece, esistere reti di composizione.

- La composizione è simile alla relazione che esiste tra un albero e le sue foglie:
 - le foglie appartengono a un unico albero;
 - gli alberi non possono tra loro condividere le proprie foglie;
 - quando un albero muore, si porta dietro le proprie foglie.
- Una parte di un composito è equivalente a un attributo:
 - la composizione esplicita è più indicata per modellare le parti che sono importanti o interessanti;
 - gli attributi sono più indicati per modellare le parti che non sono né importanti, né interessanti.

5. Rifinitura delle associazioni di analisi.

- Le associazioni di analisi dovrebbero essere rifinite con relazioni di aggregazione in tutte le situazioni in cui è possibile. Se questo comporta la formazione di un ciclo nel grafo delle aggregazioni, allora non si può utilizzare una relazione di aggregazione, ma è invece necessario ricorrere a un'associazione o a una dipendenza.
- Procedura per la rifinitura delle associazioni con relazioni di aggregazione:
 - aggiungere molteplicità e nomi dei ruoli;
 - decidere quale classe è il Tutto e quale è la Parte;
 - controllare la molteplicità relativa al Tutto:
 - se è 1, allora si può forse utilizzare una composizione: controllare la semantica dell'associazione e, qualora corrisponda, applicare una composizione;
 - se non è 1, è *necessario* utilizzare una relazione di aggregazione;
 - aggiungere la navigabilità dal Tutto verso la Parte.

6. Tipi diversi di associazione:

- Associazione uno-a-uno: si risolve quasi sempre con una composizione. Tuttavia potrebbe anche risolversi nell'aggiunta di un attributo o nella fusione delle due classi.
- Associazione molti-a-uno: si utilizza l'aggregazione in quanto, esistendo molti Tutti non è possibile utilizzare la composizione.
- Associazione uno-a-molti:
 - la Parte è costituita da un insieme di oggetti;
 - si può implementare con un vettore (array) (supportati direttamente dalla maggior parte dei linguaggi OO): sono tipicamente poco flessibili, ma possono essere molto efficienti e veloci;

- si può implementare con una classe contenitore: sono più flessibili dei vettori e possono essere più veloci dei vettori quando si deve ricercare un elemento (ma sono tipicamente più lenti negli altri casi).
- Classi contenitore:
 - Si tratta di classi specializzate le cui istanze gestiscono un insieme di altri oggetti.
 - Tutte le classi contenitore hanno metodi per:
 - aggiungere oggetti all'insieme;
 - rimuovere oggetti dall'insieme;
 - recuperare il riferimento a un oggetto dell'insieme;
 - iterare sull'insieme: ciclare sull'insieme dal primo oggetto all'ultimo.
 - Opzioni per la modellazione delle classi contenitore:
 - modellare esplicitamente la classe contenitore;
 - usare lo strumento CASE per dichiarare il tipo di classe contenitore che si vuole utilizzare, aggiungendo una proprietà alla relazione: per esempio, {Vector};
 - illustrare al programmatore la semantica del tipo di classe contenitore richiesta aggiungendo una proprietà alla relazione:
 - {ordinata}: gli elementi dell'insieme sono soggetti ad un ordinamento rigido (proprietà standard dell'UML);
 - {non-ordinata}: gli elementi dell'insieme non sono soggetti a alcun ordinamento (proprietà standard dell'UML);
 - {ordinata per chiave}: gli elementi dell'insieme sono ordinati sulla base di qualche chiave;
 - {indicizzata}: ogni elemento dell'insieme ha un indice numerico, esattamente come gli elementi di un vettore;
 - {insieme}: l'insieme non può contenere elementi duplicati;
 - {lifo} o {stack}: stack: ultimo a entrare, primo a uscire;
 - {fifo} o {coda}: coda: primo a entrare, primo a uscire.
 - Non bisogna modellare in modo eccessivo: la scelta di una specifica classe contenitore è spesso una questione tattica, che può essere lasciata al programmatore.
 - Classi contenitore dell'OCL:
 - Set: duplicati non ammessi, non indicizzata, non ordinata;
 - Bag: duplicati ammessi, non indicizzata, non ordinata;
 - Sequence: duplicati ammessi, indicizzata, ordinata.

- La mappa:
 - anche detta dizionario;
 - data una chiave, il corrispondente valore può essere reperito in modo efficiente e veloce;
 - è assimilabile a una tabella di *database* con due colonne: una chiave e un valore;
 - le chiavi devono essere univoche.
- Relazioni da reificare:
 - Alcune relazioni sono artefatti astratti prodotti durante l'analisi e devono essere reificate, ovvero devono essere rese implementabili.
 - Associazioni molti-a-molti:
 - decidere quale classe sia il Tutto e quale sia la Parte;
 - sostituire la relazione con una classe intermedia: questo riduce la relazione molti-a-molti a due associazioni uno-a-molti.
 - Associazioni bidirezionali: sostituire con un'aggregazione o una composizione unidirezionale navigabile dal tutto alla parte, più un'associazione o dipendenza unidirezionale navigabile dalla parte al tutto.
 - Classi associazione:
 - decidere quale delle due classi associate sia il Tutto e quale sia la Parte;
 - sostituire la classe associazione con una classe (solitamente si mantiene il nome della classe associazione);
 - aggiungere una nota contenente il vincolo che le coppie di oggetti associati dalla relazione reificata devono essere univoche.

Interfacce e sottosistemi

17.1 Contenuto del capitolo

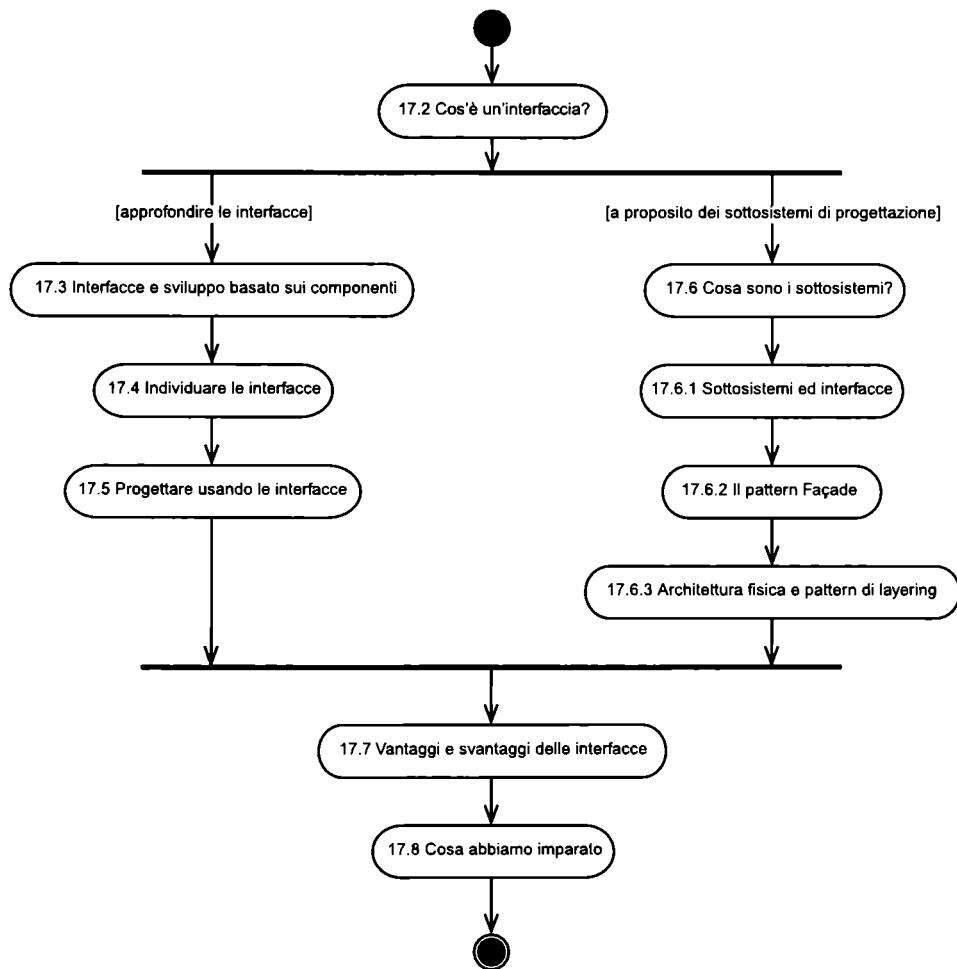
Il capitolo tratta due argomenti principali: le interfacce e i sottosistemi di progettazione. Vengono discussi insieme perché, come si vedrà nelle Sezioni 17.6.1 e 17.6.2, l'uso combinato di interfacce e sottosistemi di progettazione consente la creazione di architetture di sistema flessibili.

17.2 Cos'è un'interfaccia?

Un'interfaccia è un insieme di operazioni identificato da un nome. L'idea sottostante è quella di separare le *specifiche* di una funzionalità (l'interfaccia) dall'*implementazione* della stessa da parte di un classificatore, quale una classe o un sottosistema. L'interfaccia definisce un contratto che viene implementato dal classificatore.

Il classificatore può essere incluso fisicamente in un sottosistema o in un componente (i componenti sono trattati nel Capitolo 22). Se l'interfaccia è pubblica anche in questo sottosistema o componente, allora anche il componente o sottosistema realizzano tale interfaccia. Realizzare un'interfaccia significa accettare di rispettare il contratto che tale interfaccia definisce. Questo contratto è costituito dall'insieme delle specifiche di tutte le operazioni che compongono l'interfaccia.

Le interfacce possono avere un impatto notevole sulle attività di progettazione. Fino a questo punto è stata considerata quella parte della progettazione in cui si definiscono le classi e le si connettono tra loro: questa attività viene chiamata “progettare per implementazione”. Tuttavia, è più flessibile “progettare per contratto”, ovvero definire le interfacce e connetterle alle classi che le realizzano. Le librerie standard Java testimoniano la flessibilità e la potenza di questo approccio. Si può dire che un'interfaccia definisce un *servizio* offerto da una classe, un sottosistema o un componente. Le moderne architetture del software sono sempre più spesso incentrate sui servizi.

**Figura 17.1**

Dal punto di vista architettonale, le interfacce diventano molto importanti durante la progettazione, in quanto forniscono “le spine e le prese” che consentono di connettere tra loro sottosistemi di progettazione, senza però collegare le specifiche classi che risiedono in quei sottosistemi.

In un’interfaccia, ogni operazione *deve* avere:

- la segnatura completa: nome e tipo di tutti i parametri e tipo restituito;
- la semantica dell’operazione: sotto forma testuale o di pseudo-codice;
- optionalmente, uno stereotipo, vincoli e valori etichettati.

Le interfacce *non* possono avere:

- attributi;
- implementazione delle operazioni;
- relazioni navigabili *dall'interfaccia a nessun altro tipo di classificatore.*

È importante ricordarsi che l'interfaccia definisce la segnatura delle operazioni, la loro semantica, stereotipi, vincoli e valori etichettati, ma *nulla* di più. In particolare, non specifica o implica mai alcun dettaglio relativo all'implementazione: quest'ultima è interamente delegata alle classi, ai sottosistemi e ai componenti che realizzano l'interfaccia.

Un'interfaccia è molto simile a una classe astratta senza attributi, e con un'insieme di operazioni completamente astratte. Ciò che rende le interfacce sufficientemente dissimili da richiedere una notazione differente, è che esse sono soggette a un vincolo aggiuntivo che stabilisce che non possono esistere associazioni *da* un'interfaccia *a* nessun altro tipo di classificatore.

È forse il caso di osservare che in Java le interfacce si discostano leggermente dalla definizione che ne dà l'UML. Le interfacce Java possono, infatti, anche contenere valori costanti: valori che non possono mai essere modificati. Tipici esempi di valori costanti sono i giorni della settimana e i mesi dell'anno. La possibilità di incorporare valori costanti in un'interfaccia non ha alcun impatto sulla progettazione per interfacce, ma fornisce comunque ai programmati Java un meccanismo conveniente per raggruppare e rendere accessibili le loro costanti!

Anche se Java è il primo linguaggio comunemente utilizzato a introdurre un costrutto nativo per le interfacce, è comunque possibile programmare utilizzando le interfacce anche con gli altri linguaggi. Per esempio, in C++ si può definire una classe astratta con un insieme di operazioni completamente astratte, senza alcuna associazione ad altri classificatori.

La Figura 17.2 illustra le due diverse sintassi che l'UML formalizza per modellare le interfacce: la notazione stile “classe” (in cui si possono elencare le operazioni) e la notazione sintetica stile “palloncino” (in cui non si possono elencare le operazioni).

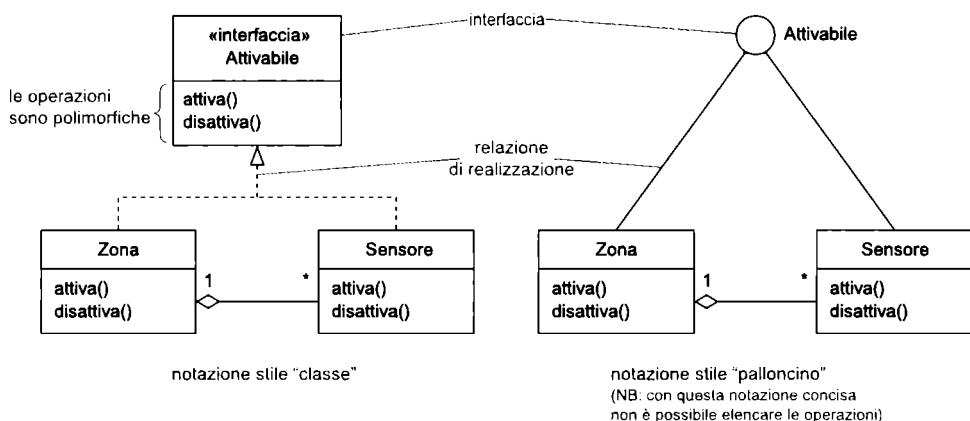


Figura 17.2

La relazione di realizzazione indica la relazione tra una specifica (in questo caso un'interfaccia) e ciò che realizza la specifica (in questo caso, le classi Zona e Sensore). Si vedrà più avanti che le interfacce non sono realizzate necessariamente da classi: anche i *package* e i componenti possono realizzarle.

Si osservi che nella notazione stile “classe” la relazione di realizzazione viene disegnata come una linea tratteggiata che termina con una punta a triangolo (generalizzazione), mentre con la notazione stile “palloncino” si utilizza una linea continua senza punta. La notazione stile “palloncino” vuole essere la più semplice e concisa possibile. Si utilizza la notazione stile “classe” quando si vuole specificare l’insieme delle operazioni, e la notazione stile “palloncino” quando invece non serve. Le due notazioni hanno tuttavia lo stesso identico significato.

Studiando la Figura 17.2, relativa a un sistema di allarme antifurto, si vede che l’interfaccia Attivabile specifica un servizio di attivazione/disattivazione. Questo tipo di servizio è ovviamente molto utile in un sistema di questo tipo. La suddetta interfaccia è realizzata da due classi: la classe Zona e la classe Sensore. I sistemi di allarme antifurto prevedono tipicamente molte Zone, ciascuna delle quali può essere attivata e disattivata indipendentemente dalle altre. All’interno di queste Zone esistono dei Sensori, e anche questi possono essere attivati e disattivati. Dal punto di vista del sistema ha senso che l’attivazione e la disattivazione delle Zone e dei Sensori avvenga nello stesso identico modo. La tecnica migliore per garantire questo risultato è quella di definire un’interfaccia per il servizio di attivazione/disattivazione: Attivabile. Fatto questo, tutto ciò che nel sistema necessita di un servizio di attivazione/disattivazione, deve semplicemente realizzare questa interfaccia.

Solitamente le interfacce hanno nomi simili a quelli delle classi. In Visual Basic e C# tuttavia esiste uno standard diffuso che prevede che il nome delle interfacce abbia come prefisso una I maiuscola, come per esempio `IAttivabile`.

17.3 Interfacce e sviluppo basato sui componenti

Le interfacce sono l’elemento chiave dello sviluppo basato sui componenti (component-based development, CBD). Questo tipo di sviluppo prevede che il software venga costruito utilizzando componenti o plug-in. L’unico modo per creare del software flessibile basato sui componenti, in cui sia sempre possibile aggiungere nuove implementazioni, è quello di utilizzare le interfacce sin dalla progettazione. Dato che un’interfaccia definisce solo un contratto, consente l’uso di un *qualsiasi numero* di diverse implementazioni, a patto che ciascuna rispetti il contratto.

Nella Figura 17.3 si può vedere come le interfacce consentano di collegare tra loro classi senza, però, introdurre interdipendenze.

In questo esempio la classe Stampante è in grado di stampare qualunque cosa implementi l’interfaccia Stampabile. Questa interfaccia definisce un’unica operazione astratta polimorfa chiamata `stamp(g : Graphics)`, che non viene mostrata nel diagramma, dato che utilizza la notazione sintetica stile “palloncino”. Questa operazione *dove* essere implementata da *tutte* le classi che realizzano Stampabile. La classe Stampante non ha, dunque, alcun tipo di interdipendenza con le classi che effettivamente implementano Stampabile: conosce esclusivamente l’interfaccia Stampabile.

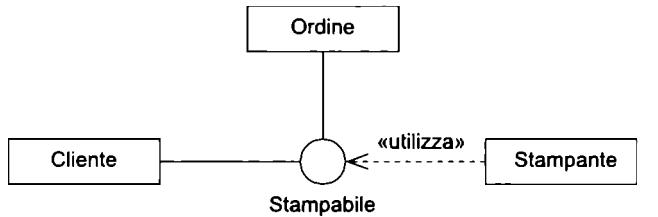


Figura 17.3

La Figura 17.4 illustra come le interfacce possano risultare efficienti e utili nella modellazione di sottosistemi. In questo esempio il sottosistema GUI conosce *esclusivamente* le interfacce GestoreClienti e GestoreConti. Non sa nulla di specifico dell’altro sottosistema. In linea di massima, è possibile sostituire del tutto il sottosistema LogicaBusiness con un altro diverso sottosistema, o anche con un insieme di sottosistemi, a patto che questi forniscano un’implementazione dello stesso insieme di interfacce richieste. Questo uso delle interfacce può offrire un’ottima flessibilità architetturale.

L'esempio riportato nella Figura 17.5 è basato sul caso reale di un sistema di sicurezza di un ufficio. Ogni piano del palazzo è una diversa Zona, che comprende un certo numero di Zone secondarie, oltre a dei sensori passivi a infrarossi e altri sensori di allarme. Ogni Zona ha un lettore di tessere magnetiche che può servire per attivare o disattivare una o più tessere di identificazione in quella Zona. Una tessera attivata su una Zona aggregata è anche attiva (valida) per tutte le sue Zone secondarie.

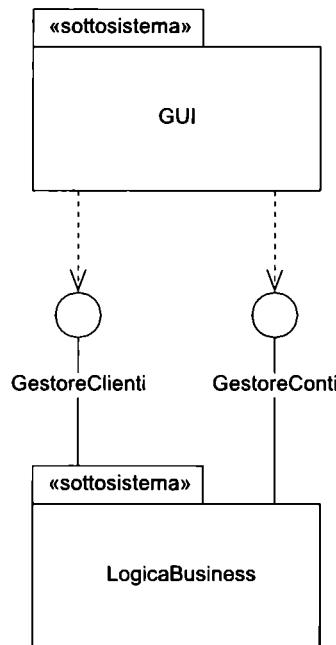
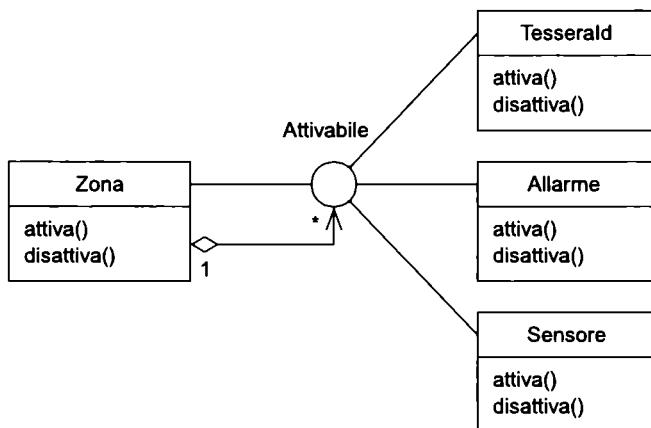


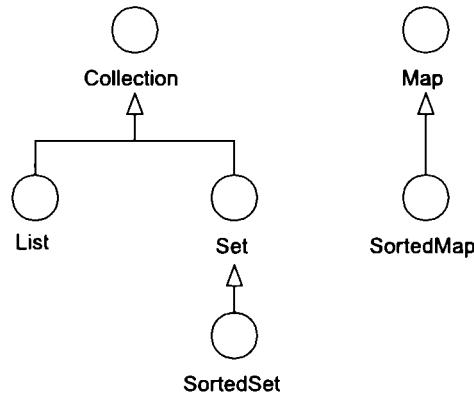
Figura 17.4

**Figura 17.5**

Nel nostro modello di questo sistema, ogni **Zona** contiene una collezione di oggetti che realizzano l’interfaccia **Attivable**. In questo modo, diventa molto semplice aggiungere nuovi tipi di apparati, quali rilevatori di fumo e lettori di tessere magnetiche.

Si osservi che la relazione di aggregazione esistente tra **Zona** e **Attivable** è unidirezionale. Questo è obbligatorio, perché anche se altri elementi possono essere associati a un’interfaccia, l’interfaccia stessa non può essere associata a nessun elemento. Si osservi anche come l’asimmetria dell’aggregazione vietи a una qualunque oggetto **Zona** di essere parte di se stesso.

Java fa un uso molto diffuso delle interfacce in tutte le librerie standard. Un esempio particolarmente valido è quello delle classi contenitore di Java (Figura 17.6). Anche se vengono definite solo sei interfacce, ne esistono in realtà ben dieci diverse implementazioni, ciascuna della quali offre caratteristiche diverse. Progettando per interfacce, il progettista Java può effettivamente rimandare la realizzazione dell’interfaccia a quando deve essere implementata, lasciando così la possibilità al programmatore di scegliere la soluzione più appropriata.

**Figura 17.6**

17.4 Individuare le interfacce

Quando si è a buon punto nella progettazione di un sistema, o di una sua parte, vale la pena di esaminare il modello alla ricerca di potenziali interfacce. È piuttosto facile da fare.

- Mettere in discussione ogni associazione: studiare ogni associazione e chiedersi se deve essere veramente limitata a una classe specifica o se non debba essere, invece, più flessibile. Se si giunge alla conclusione che deve essere più flessibile, allora si può prendere in considerazione l'uso di un interfaccia.
- Mettere in discussione ogni messaggio inviato: studiare ogni messaggio e chiedersi se debba essere veramente inviato solo agli oggetti di una specifica classe o se non debba essere invece più flessibile. Se deve essere più generico (se si riesce a pensare a qualche situazione in cui lo stesso messaggio potrebbe essere inviato a oggetti di un'altra classe), allora si può prendere in considerazione l'uso di un'interfaccia.
- Individuare gruppi di operazioni che possono essere riusate in altre classi. Se, per esempio, molte classi del sistema devono essere in grado di stamparsi su qualche periferica di *output*, allora forse è il caso di introdurre un'interfaccia Stampabile.
- Individuare gruppi di operazioni ripetute in classi diverse.
- Ricercare classi del sistema che sembrino avere un ruolo simile o identico: quel ruolo potrebbe forse diventare un'interfaccia.
- Individuare possibilità per future espansioni del sistema. A volte, con un minimo sforzo aggiuntivo, è possibile progettare sistemi che possono essere successivamente espansi o estesi con molta facilità. La domanda chiave è: "In futuro sarà necessario aggiungere classi al sistema?". Se la risposta è affermativa si può tentare di progettare una o più interfacce che definiscano il protocollo adatto per l'aggiunta di queste nuove classi.

17.5 Progettare usando le interfacce

È molto più facile progettare un sistema se le diverse parti si comportano in modo omogeneo e uniforme. L'utilizzo delle interfacce consente di definire un insieme comune di operazioni che molte diverse classi possono realizzare. Un buon esempio è un sistema su cui abbiamo lavorato, il quale doveva fornire un'interfaccia comune per diversi sistemi aziendali esistenti. Il problema era che ciascuno di questi sistemi utilizzava un protocollo di comunicazione differente. Questa complessità è stata nascondata introducendo un'unica interfaccia costituita dalle operazioni `apri()`, `leggi()`, `scrivi()` e `chiudi()`.

Ecco un secondo esempio. Si prenda in considerazione un sistema che modella un'organizzazione (in questo caso, un sistema per la gestione delle risorse umane),

esistono molte classi di oggetti che hanno un nome e un indirizzo: Persona, UnitaOrganizzativa, Ufficio. Tutte queste classi possono interpretare il ruolo comune di unitaIndirizzabile. È ovvio che queste classi dovrebbero implementare tutte un'interfaccia comune per la gestione delle informazioni relative al nome e all'indirizzo. Si può, quindi, definire un'interfaccia NomeIndirizzo che tutte devono realizzare. Questo problema può anche essere risolto con l'ereditarietà, ma la soluzione basata sulle interfacce è molto più flessibile.

Vale la pena di ricordare che le classi possono anche avere delle associazioni riflesive (con se stesse) e che esistono ruoli che sono del tutto interni a alcune classi. Anche questi sono buoni candidati per l'utilizzo di interfacce.

Le interfacce diventano uno strumento molto potente nel momento in cui vengono utilizzate per consentire l'inserimento di *plug-in* nel sistema. Una delle tecniche per rendere i sistemi flessibili e resistenti ai cambiamenti, è quella di progettarli in modo che sia possibile inserirvi facilmente delle estensioni. Per fare questo, è necessario utilizzare le interfacce. In un sistema progettato a interfacce, le associazioni e i messaggi inviati non sono più vincolati agli oggetti di una specifica classe, ma diventano vincolati a una specifica interfaccia. Questo rende più facile l'aggiunta di nuove classi al sistema, in quanto le interfacce definiscono il protocollo che le nuovi classi devono supportare per potersi inserire senza che il resto del sistema se ne accorga.

Un tipo di modulo software che è molto utile poter inserire e cambiare a piacere sono i plug-in contenenti algoritmi. Qualche tempo fa, lavorando su un sistema che eseguiva un calcolo complesso su una base dati molto grande, gli utenti richiedevano di poter sperimentare con l'algoritmo di calcolo per poter trovare la strategia ottimale. Però il sistema non era stato costruito tenendo conto di questa esigenza, e qualunque modifica dell'algoritmo, per quanto minima, richiedeva diversi giorni-uomo di lavoro, in quanto era necessario rivedere il codice e ricostruire l'intero sistema. Si è lavorato con uno dei progettisti per ristrutturare il sistema in modo da supportare un'interfaccia per algoritmi di calcolo intercambiabili. Terminato, era possibile provare i nuovi algoritmi nel giro di poche ore. In effetti, era addirittura possibile cambiare l'algoritmo mentre il sistema era in esecuzione.

17.6 Cosa sono i sottosistemi?

Un sottosistema è un *package* con lo stereotipo «sottosistema». I sottosistemi vengono utilizzati sia durante la progettazione, sia durante l'implementazione. Sono, quindi, chiamati sottosistemi di progettazione o sottosistemi di implementazione. In questo testo il termine «sottosistema» verrà spesso utilizzato da solo, dato che dal contesto si capisce facilmente di quale tipo di sottosistema si sta parlando.

I sottosistemi di progettazione contengono:

- classi e interfacce di progettazione;
- realizzazioni di caso d'uso;
- altri sottosistemi;
- elementi di specifica, quali i casi d'uso.

I sottosistemi servono per:

- separare compiti individuati durante la progettazione;
- rappresentare componenti di alto livello;
- incapsulare e nascondere la complessità dei sistemi esterni.

I sottosistemi di progettazione sono il primo passo verso la scomposizione del modello in componenti. I singoli *package* di analisi possono corrispondere a uno o più sottosistemi di progettazione, inoltre si possono introdurre sottosistemi che contengono esclusivamente artefatti provenienti dal dominio delle soluzioni, quali classi per l'accesso al *database* o classi per la comunicazione.

Come per i *package* di analisi, anche tra i sottosistemi di progettazione possono esistere dipendenze. In particolare, succede quando un elemento di un sottosistema è in qualche modo dipendente da un elemento di un altro sottosistema.

Come illustra la Figura 17.7, esistono diversi modi per disegnare graficamente un sottosistema. Si può utilizzare uno stereotipo (la tecnica più comune), o si può utilizzare l'icona a forcetta. Il nome dello stereotipo può essere posizionato nella linguetta o nel corpo del *package*, appena sopra il nome del sottosistema. L'icona a forcetta può essere collocata nella linguetta o nell'angolo in alto a destra del corpo del *package*.

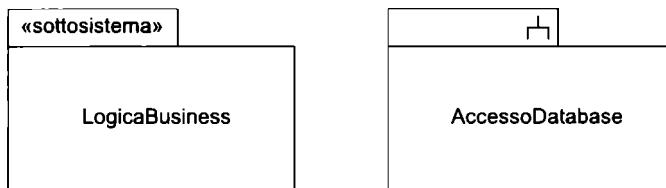


Figura 17.7

L'icona del sottosistema può essere suddivisa in una o più delle seguenti sottosezioni:

- operazioni;
- elementi della realizzazione;
- elementi della specifica.

Tutte le sottosezioni sono opzionali e in effetti è anche possibile lasciare l'icona vuota, oppure includere tutte le informazioni in un'unica sottosezione priva di etichetta. Nella Figura 17.8, viene riportato un esempio in cui sono utilizzate tutte e tre le sottosezioni.

In alto a sinistra, si trova la sottosezione Operazioni, la quale non ha mai alcuna etichetta. Questa sottosezione contiene le operazioni messe a disposizione dal sottosistema. Ciascuna di queste operazioni deve essere realizzata da un metodo pubblico di una classe pubblica del sottosistema. In questo esempio le tre operazioni del sottosistema sono tutte realizzate dalla classe GestoreRegistrazioni. Se ne può, dunque, dedurre che nel sottosistema la classe GestoreRegistrazioni deve essere pubblica e deve avere tre metodi pubblici: aggiungiCorso(...), rimuoviCorso(...) e trovaCorso(...), che realizzano le tre operazioni del sottosistema.

La sottosezione Elementi della specifica contiene elementi che specificano qualche aspetto del sottosistema, quali casi d'uso e interfacce. La sottosezione Elementi della realizzazione contiene gli elementi che realizzano il comportamento del sottosistema. La Figura 17.8 illustra come si possano anche indicare relazioni di realizzazione esplicite tra gli elementi di questa sottosezione e gli elementi delle altre due sottosezioni.

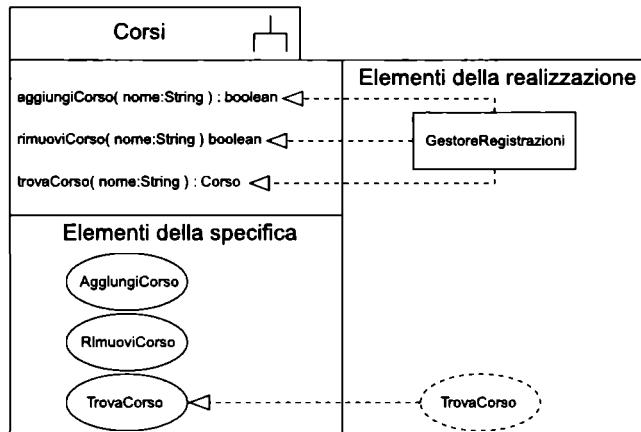


Figura 17.8

17.6.1 Sottosistemi e interfacce

È molto utile riuscire a progettare i sottosistemi in modo che si comportino da “scatole nere”, cioè che le classi del sottosistema siano completamente sconosciute e occultate a tutte le altre classi del sistema. È possibile raggiungere questo obiettivo, introducendo delle interfacce e rendendole pubbliche a livello di sottosistema. La maggior parte delle classi contenute nel sottosistema può, a questo punto, essere resa privata.

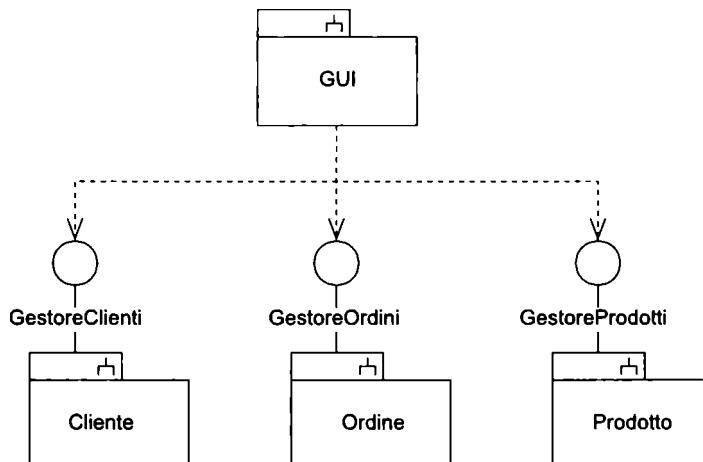


Figura 17.9

Si osservi che con l'introduzione delle interfacce, si ottiene un risultato sostanzialmente diverso dall'uso delle operazioni del sottosistema. Le operazioni del sottosistema richiedono la presenza di classi pubbliche che le realizzino. Questo comporta l'esposizione di alcune classi al resto del sistema.

Quando una classe pubblica di un sottosistema realizza un'interfaccia, si può dire che il sottosistema stesso realizza tale interfaccia. Si può associare direttamente il sottosistema all'interfaccia, come mostrato nella Figura 17.9.

Questo utilizzo delle interfacce consente di pensare al sottosistema come a un tipo di componente che fornisce un insieme di servizi, realizzando una o più interfacce.

17.6.2 Il pattern *Façade*

La tecnica per cui un sottosistema complesso viene nascosto dietro un semplice interfaccia ben-definita, è nota come *pattern di Façade* (Facciata), documentato in *Design Patterns* [Gamma 1]. Il libro in questione è una vera e propria miniera di *pattern* di progettazione potenti e riusabili, che possono essere utilizzati nei modelli di progettazione, in molti contesti differenti. A proposito del *pattern Façade*, Gamma dice: “La strutturazione di un sistema in sottosistemi aiuta a ridurre la complessità. Un obiettivo comune della progettazione è quello di minimizzare la comunicazione e le dipendenze tra sottosistemi. Una tecnica per raggiungere questo obiettivo, è quella di introdurre un oggetto *façade* che fornisca un'unica interfaccia semplificata alle funzionalità generali offerte da un sottosistema.”

Il *pattern Façade* consente di occultare informazioni e separare le responsabilità: si può usare per nascondere i dettagli complessi del funzionamento interno di un sottosistema, dietro un'interfaccia semplice. Si riduce così la complessità del sistema e si riesce a tenere sotto controllo e a gestire le interdipendenze tra i diversi sottosistemi.

Le interfacce di *façade* possono essere utilizzate per creare delle “fessure” nel sistema. Per creare tali interfacce si procede nel seguente modo:

- individuare parti del sistema tra loro coese;
- raggruppare tali parti in un *package* «sottosistema»;
- definire un'interfaccia per quel sottosistema.

17.6.3 Architettura fisica e *pattern di layering*

L'architettura fisica di un modello è costituita dall'insieme dei sottosistemi e delle interfacce di progettazione. Tuttavia, affinché questa architettura sia facile da comprendere e mantenere l'insieme di sottosistemi e interfacce deve comunque essere organizzata in maniera coerente. Questo è possibile applicando il *pattern* di architettura noto come *layering* (strutturazione a strati).

Il *pattern di layering* prevede la disposizione dei sottosistemi e delle interfacce di progettazione in strati, dove tutti i sottosistemi di uno stesso strato sono semanticamente coesi.

La creazione di un'architettura robusta a strati serve per controllare le interdipendenze tra sottosistemi:

- introducendo nuove interfacce laddove necessario;
- risistemando le classi creando nuovi sottosistemi, in modo da ridurre le interdipendenze tra sottosistemi.

Le dipendenze tra strati diversi devono essere controllate molto bene, in quanto sono interdipendenze poco desiderabili. È necessario assicurarsi che:

- le dipendenze tra due strati siano tutte unidirezionali e tutte nello stesso senso;
- tutte le dipendenze siano mediate da interfacce.

Esistono molte tecniche per produrre architetture a strati, e si possono creare quanti strati si ritiene necessario. Tuttavia, lo schema tipico prevede la separazione dei sottosistemi in tre strati: presentazione, logica di *business* e utilità. Come illustra la Figura 17.10, lo strato della logica di *business* viene spesso suddiviso ulteriormente.

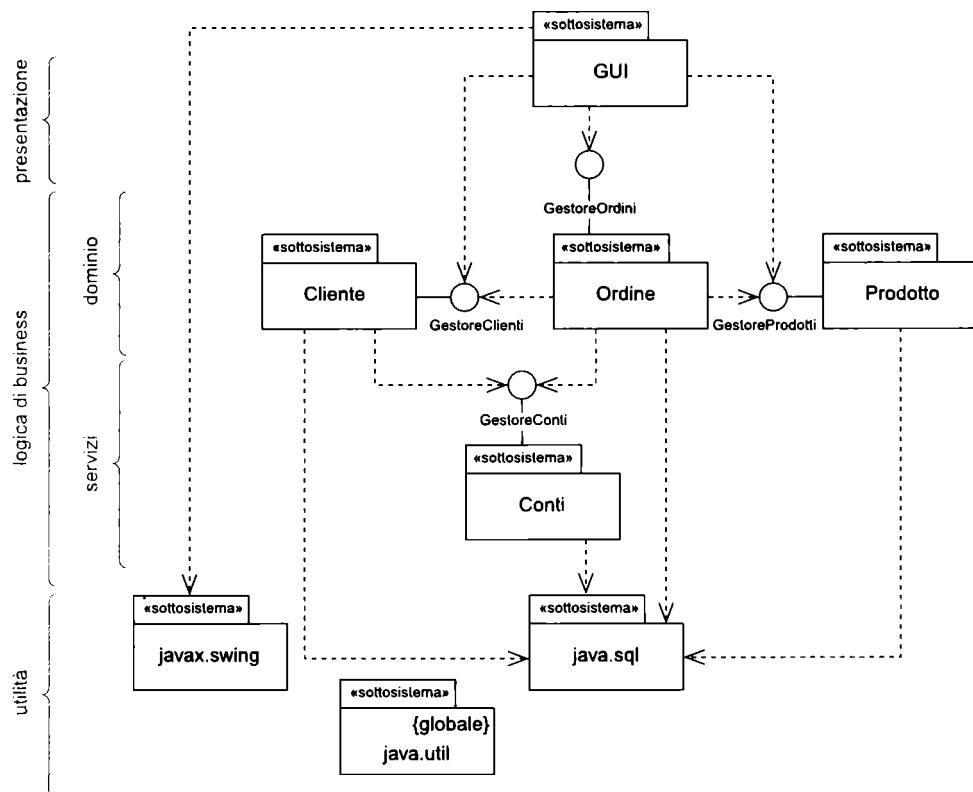


Figura 17.10

In questo esempio ci sono due sottostrati: dominio e servizi. Lo strato del dominio contiene i sottosistemi specifici di questa particolare applicazione, mentre lo strato dei servizi contiene i sottosistemi che sono riusabili anche in altre applicazioni.

Si consiglia di progettare, per quanto possibile, per interfacce. La Figura 17.10 illustra come tutti i sottosistemi da noi progettati siano connessi tra loro tramite interfacce. D'altra parte, i *package* Java sono invece connessi agli altri sottosistemi tramite dipendenze, nonostante tali *package* mettano a disposizione diverse interfacce. Il motivo per questa differenza è che è interessante e utile mostrare le interfacce specifiche del sistema, mentre mostrare quelle messe a disposizione dalle librerie standard Java non sembrerebbe avere alcuna utilità. Si osservi anche che il *package* `java.util`, che contiene componenti generici come le `String`, viene utilizzato da tutti i sottosistemi ed è quindi contrassegnato con la proprietà `{globale}`. Questo significa che tutto il contenuto pubblico del *package* è visibile in tutto il sistema.

17.7 Vantaggi e svantaggi delle interfacce

La progettazione per classi vincola il progetto a certe specifiche implementazioni. La progettazione per interfacce, invece, produce contratti che possono essere realizzati da molte diverse implementazioni. La progettazione per interfacce svincola il modello (e in ultima analisi anche il sistema) da dipendenze dell'implementazione e ne aumenta, dunque, la flessibilità e la estendibilità.

La progettazione per interfacce consente di ridurre il numero di dipendenze esistenti tra le classi, i sottosistemi e i componenti e permette, quindi, di avere sotto maggior controllo le interdipendenze presenti nel modello. L'interdipendenza è il peggior nemico dello sviluppatore a oggetti, in quanto i sistemi fortemente interdipendenti sono difficili da comprendere, da mantenere e da far evolvere. Un uso adeguato delle interfacce può aiutare a ridurre le interdipendenze e a separare il modello in sottosistemi coesi.

Esistono, comunque, anche degli svantaggi nell'utilizzo delle interfacce. In linea generale, rendere un sistema più flessibile comporta spesso anche un suo aumento di complessità. Chi usa la progettazione per interfacce deve, comunque, cercare sempre un buon compromesso tra flessibilità e complessità. In teoria, ogni metodo di ogni classe può essere trasformato in un'interfaccia, ma questo renderebbe del tutto incomprensibile un tale sistema! La flessibilità comporta, inoltre, spesso anche un costo in prestazioni, ma si tratta di una considerazione di poco conto rispetto all'aumento della complessità.

La progettazione di un sistema richiede la concretizzazione, sotto forma di software, di un insieme ben definito di requisiti di *business*. Alcuni di questi requisiti sono fluidi e cambiano molto in fretta, mentre altri risultano relativamente stabili. La flessibilità serve per affrontare gli aspetti fluidi. Una semplificazione del sistema può essere ottenuta sacrificando in parte la flessibilità, laddove i requisiti sono più stabili. In un certo senso, questo è uno dei segreti di una buona analisi e progettazione OO: distinguere le parti fluide e le parti stabili di un sistema e modellarle in modo appropriato.

A essere sinceri, la corretta modellazione di un sistema resta comunque più importante della sua modellazione flessibile. È sempre necessario innanzitutto modella-

re correttamente le principali semantiche di *business*, e solo dopo ci si può occupare della flessibilità. Infine, bisogna ricordarsi sempre di utilizzare interfacce semplici e utili.

17.8 Riepilogo

Le interfacce consentono di progettare il software specificando un contratto, e non una particolare implementazione. Sono stati spiegati i seguenti concetti.

1. Un'interfaccia definisce un insieme di operazioni.
 - Le interfacce separano la definizione delle funzionalità da quella dell'implementazione.
 - Le interfacce possono essere associate a classi, sottosistemi, componenti o a qualunque altro classificatore e definiscono i servizi che questi elementi hanno da offrire.
 - Se un classificatore pubblico che si trova in un sottosistema realizza una certa interfaccia pubblica, allora anche il sottosistema realizza la stessa interfaccia pubblica.
 - Qualunque elemento realizzi un'interfaccia, garantisce di rispettare il contratto definito dall'insieme di operazioni specificate in tale interfaccia.
2. Progettare per implementazione:
 - si connettono tra loro classi specifiche;
 - riduce la complessità del sistema (ma ne aumenta la rigidità).
3. Progettare per contratto:
 - le classi sono associate alle interfacce, ciascuna delle quali può avere molte diverse realizzazioni;
 - aumenta la flessibilità del sistema (ma ne aumenta anche la complessità).
4. Le interfacce sono costituite da un'insieme di operazioni. Ogni operazione deve avere:
 - una segnatura completa;
 - una specifica che descrive la semantica dell'operazione;
 - optionalmente uno stereotipo, dei vincoli e dei valori etichettati.
5. Le interfacce non possono mai avere:
 - attributi;
 - implementazione delle operazioni (metodi);

- relazioni navigabili dall’interfaccia a qualunque altro tipo di elemento.
6. Sintassi per le interfacce:
- utilizzare la notazione stile “classe” quando è necessario elencare le operazioni sul modello;
 - utilizzare la notazione concisa stile “palloncino” quando è sufficiente modellare l’interfaccia senza l’elenco delle operazioni.
7. Lo sviluppo basato sui componenti (component-based development, CBD) prevede la costruzione del software tramite l’utilizzo di componenti plug-in: l’uso di interfacce rende tali componenti interscambiabili; la progettazione a interfacce consente di utilizzare molti diversi componenti che realizzano in modo diverso le stesse interfacce.
8. Individuare le interfacce:
- mettere in discussione le associazioni;
 - mettere in discussione i messaggi inviati;
 - individuare e scorporare insiemi di operazioni riusabili;
 - individuare e scorporare insiemi di operazioni ripetute;
 - individuare classi diverse che interpretano lo stesso ruolo nel sistema;
 - valutare le possibilità di espansioni future.
9. I sottosistemi sono un tipo di *package*.
- I sottosistemi di progettazione contengono molti elementi di progettazione:
 - classi e interfacce di progettazione;
 - realizzazioni di caso d’uso;
 - altri sottosistemi.
 - I sottosistemi di implementazione contengono elementi relativi all’implementazione:
 - interfacce;
 - componenti;
 - altri sottosistemi.
 - Le icone dei sottosistemi possono opzionalmente avere le seguenti sottosezioni:
 - Operazioni: contiene le segnature di tutti i metodi messi a disposizione dal sottosistema (che devono essere realizzati da metodi pubblici di classi pubbliche del sottosistema);
 - Elementi della specifica: contiene elementi che specificano un qualche aspetto del sistema, quali i casi d’uso e le interfacce;
 - Elementi della realizzazione: contiene elementi che realizzano le specifiche.

- I sottosistemi possono essere utilizzati per:
 - separare i compiti individuati durante la progettazione;
 - rappresentare componenti di alto livello;
 - encapsulare sistemi aziendali esterni.
- Utilizzare le interfacce per nascondere i dettagli di implementazione dei sottosistemi:
 - il *pattern Façade* nasconde un'implementazione complessa dietro un'interfaccia semplice;
 - il *pattern di layering* organizza i sottosistemi in strati semanticamente coesi:
 - le dipendenze tra due diversi strati dovrebbero essere unidirezionali e navigabili tutte nello stesso senso;
 - tutte le interdipendenze tra strati devono essere mediate da un'interfaccia;
 - una suddivisione tipica prevede gli strati di presentazione, logica di *business* e utilità.

Realizzazione di caso d'uso della progettazione

18.1 Contenuto del capitolo

Il capitolo tratta delle realizzazioni di caso d'uso della progettazione. Questo processo serve a rifinire i diagrammi di interazione e i diagrammi di classe prodotti durante l'analisi, per illustrare gli artefatti della progettazione. Avendo già approfondito le classi di progettazione nel Capitolo 15, ci si concentra ora sui diagrammi di interazione. Spieghiamo inoltre come utilizzare nella progettazione i diagrammi di interazione per modellare le meccaniche centrali: le decisioni strategiche che devono essere prese durante la progettazione riguardo alla persistenza degli oggetti, alla loro distribuzione ecc. Infine, si vedrà come utilizzare i diagrammi di interazione per fissare le interazioni di alto livello interne al sistema, tramite la creazione di diagrammi di interazione dei sottosistemi.

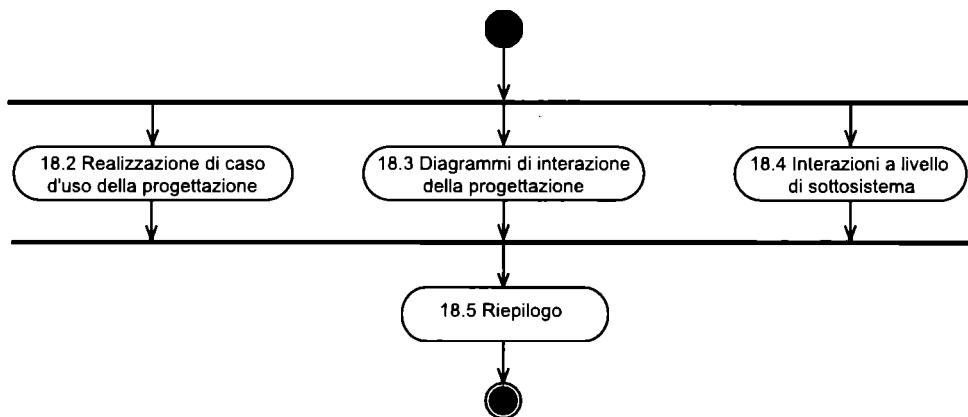


Figura 18.1

18.2 Realizzazione di caso d'uso della progettazione

Una realizzazione di caso d'uso della progettazione è una collaborazione tra oggetti e classi di progettazione che realizzano un caso d'uso. Esiste una relazione «origine» tra la realizzazione di caso d'uso dell'analisi e la realizzazione di caso d'uso della progettazione. La realizzazione di caso d'uso della progettazione specifica le decisioni relative all'implementazione, e implementa i requisiti non-funzionali. Una realizzazione di caso d'uso della progettazione è costituita da:

- diagrammi di interazione di progettazione;
- diagrammi di classe contenenti le classi di progettazione partecipanti.

Nelle realizzazioni di caso d'uso dell'analisi, ci si concentra su *cosa* il sistema deve fare. Nella progettazione ci si preoccupa di *come* il sistema lo farà. Quindi, a questo punto, è necessario specificare tutti quei dettagli di implementazione che sono stati ignorati durante l'analisi. Le realizzazioni di caso d'uso della progettazione sono, dunque, molto più dettagliate e complesse delle realizzazioni di caso d'uso originali dell'analisi.

È importante ricordarsi che la modellazione deve servire solo per comprendere il sistema che si vuole costruire. Durante la progettazione, ci si deve limitare a un livello di dettaglio che sia utile: questa si chiama progettazione strategica. Esiste anche la progettazione tattica, che però può essere tranquillamente rimandata alla fase di implementazione. In effetti, l'unico motivo per progettare in modo esaustivo è che si vuole generare la maggior parte del codice direttamente dal modello. Ma anche in questo caso, è piuttosto difficile che le realizzazioni di caso d'uso della progettazione abbiano un ruolo attivo nella generazione automatica del codice, e quindi vengono create solo per evidenziare qualche aspetto poco chiaro del comportamento del sistema.

18.3 Diagrammi di interazione della progettazione

I diagrammi di interazione sono una parte fondamentale delle realizzazioni di caso d'uso della progettazione. Dato che i diagrammi di sequenza hanno un maggiore contenuto informativo, di solito ci si concentra di più su questi che non sui diagrammi di collaborazione.

I diagrammi di interazione possono essere:

- una rifinitura dei principali diagrammi di interazione prodotti dall'analisi, con l'aggiunta di dettagli relativi all'implementazione;
- diagrammi completamente nuovi costruiti per illustrare questioni tecniche che sono state individuate durante le attività di progettazione.

Durante la progettazione viene introdotto un certo numero di meccaniche centrali, quali la persistenza degli oggetti, la distribuzione degli oggetti, le transazioni ecc. Spesso si creano nuovi diagrammi di interazione specifici per illustrare queste meccaniche. I diagrammi di interazione che illustrano le meccaniche centrali sono solitamente trasversali a diversi casi d'uso.

Per capire meglio il ruolo dei diagrammi di sequenza nella progettazione, si riprende il caso d'uso **AggiungiCorso** discusso precedentemente nel Paragrafo 12.8. La Figura 18.2 ripropone il caso d'uso **AggiungiCorso**.

La Figura 18.3 illustra il diagramma di interazione dell'analisi creato nel Paragrafo 12.8.

La Figura 18.4 mostra quello che potrebbe essere il diagramma di sequenza per il caso d'uso **AggiungiCorso** tipico delle fasi iniziali della progettazione. Si può vedere che è stato aggiunto lo strato GUI, anche se non è stato modellato con troppo dettaglio. Le operazioni di alto livello presenti nel diagramma di sequenza dell'analisi sono state risolte con specifici metodi di classe, completi di parametri. In particolare, la costruzione degli oggetti a questo punto è espressa esplicitamente tramite la chiamata di appositi metodi di costruzione.

Caso d'uso: AggiungiCorso
ID: UC8
Attori: Registrante
Precondizioni: Il Registrante è stato autenticato dal sistema.
Sequenza degli eventi: 1. Il Responsabile seleziona "aggiungi corso". 2. Il sistema accetta il nome del nuovo corso. 3. Il sistema crea il nuovo corso.
Postcondizioni: Un nuovo corso è stato aggiunto al sistema.

Figura 18.2

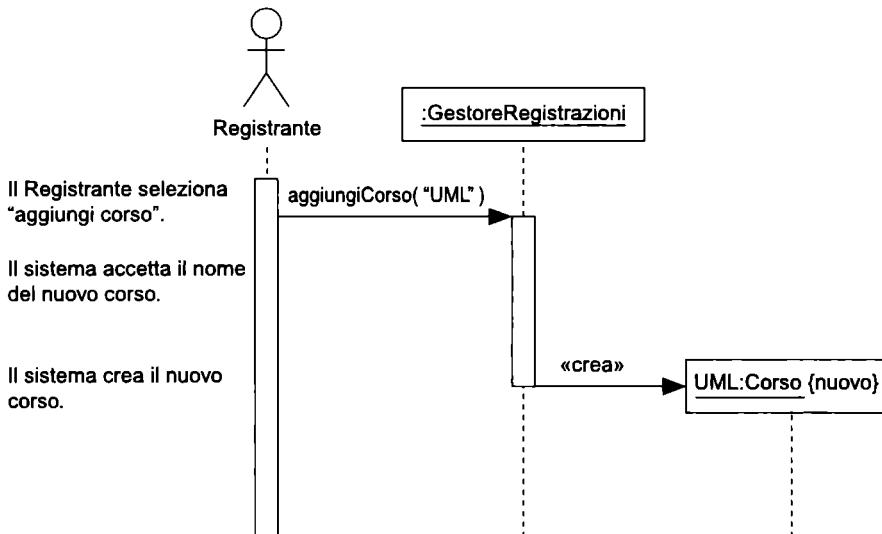
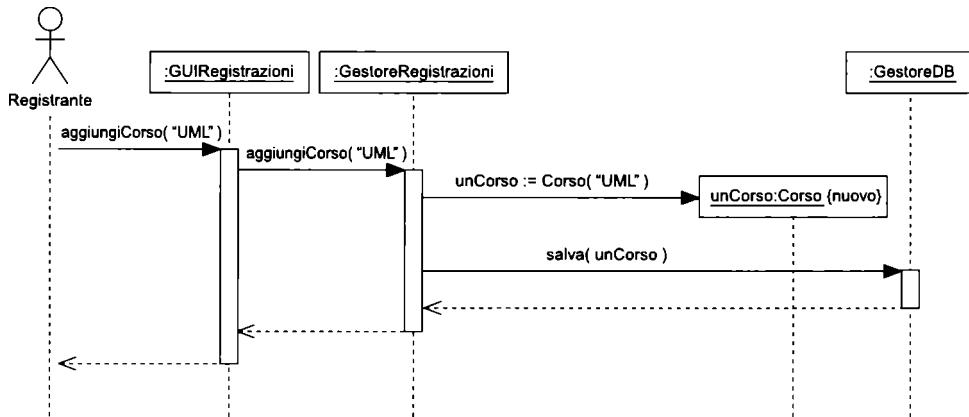


Figura 18.3

**Figura 18.4**

La Figura 18.4 risolve anche una delle meccaniche centrali: come verrà gestita la persistenza degli oggetti `Corso`. In questo caso, è stata scelta una meccanica di persistenza molto semplice: l'oggetto `:GestoreRegistrazioni` sfrutta i servizi dell'oggetto `:GestoreDB` per salvare gli oggetti `Corso` in un *database*. È essenziale che, una volta definita, questa stessa meccanica centrale venga utilizzata con consistenza in tutto il modello della progettazione. Una volta è capitato di lavorare su un sistema di grandi dimensioni che utilizzava almeno tre diverse meccaniche di persistenza: ovvero, almeno due di troppo!

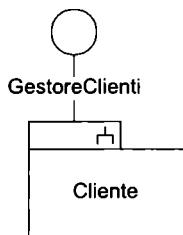
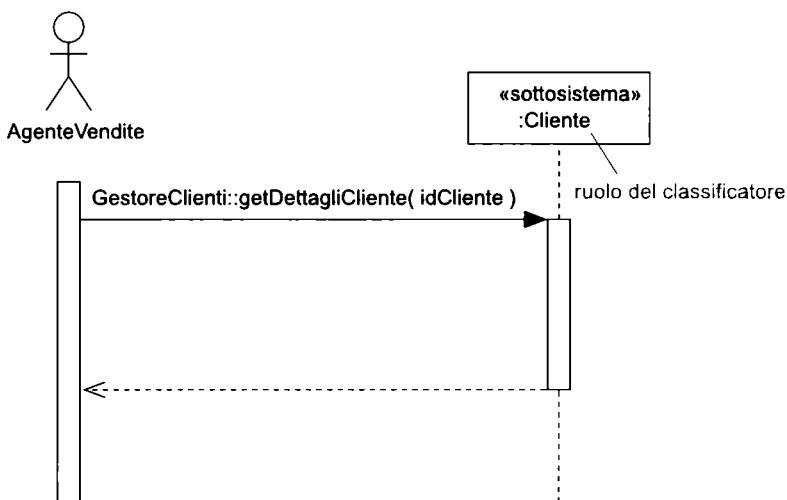
18.4 Interazioni a livello di sottosistema

Una volta che è stata definita l'architettura fisica dei sottosistemi e delle interfacce, può spesso risultare utile modellare le realizzazioni di caso d'uso non a livello di classe, ma a livello di sottosistema. Le interazioni a livello di sottosistema forniscono una vista di alto livello di come l'architettura realizzi i casi d'uso, senza entrare nel dettaglio di basso livello relativo all'interazione tra i singoli oggetti.

In questo tipo di diagramma, ogni sottosistema deve essere trattato come una scatola nera che fornisce servizi tramite le proprie operazioni o, meglio, tramite le proprie interfacce pubbliche. In questo modo, non c'è proprio bisogno di preoccuparsi delle interazioni tra gli oggetti interni del sottosistema. Tutto sommato, l'utilizzo di interfacce è preferibile all'utilizzo di operazioni, in quanto le interfacce sono maggiormente coese e forniscono una migliore encapsulazione.

La Figura 18.5 mostra il sottosistema Cliente che ha un'unica interfaccia chiamata `GestoreCliente`.

La Figura 18.6 riporta parte di un diagramma di sequenza relativo a un attore che interagisce con questo sottosistema. Dato che il sottosistema realizza tutte le proprie interfacce (e operazioni), nel diagramma di interazione i messaggi sono inviati direttamente al sottosistema. Volendo indicare l'interfaccia specifica che viene utilizzata in un'interazione, è possibile qualificare il nome del messaggio con il nome dell'interfaccia.

**Figura 18.5****Figura 18.6**

Per modellare le interazioni a livello di sottosistema, di solito si utilizzano diagrammi di interazione in forma descrittore (vedere Sezione 12.5). Questo perché i diagrammi di interazione in forma istanza modellerebbe l'interazione con istanze dei sottosistemi, ovvero a livello di singola esecuzione del programma. Di solito non si vuole, e non si deve, essere così specifici!

Il diagramma di sequenza riportato nella Figura 18.6 è in forma descrittore, come si può facilmente capire osservando che il nome del classificatore non è sottolineato. Il sottosistema *Cliente* viene visto come una scatola nera che soddisfa richieste di informazioni di dettaglio sui clienti, effettuate tramite la sua interfaccia *GestoreClienti*. Nel diagramma non è esplicitato, ma il risultato di queste richieste viene restituito sotto forma di oggetto di classe *DettagliCliente*. Questo significa che esistono interazioni tra gli oggetti interni del sottosistema *Cliente*, finalizzate alla raccolta delle informazioni richieste e alla costruzione di un oggetto *DettagliCliente*. Qualora fosse necessario, queste interazioni potrebbero essere mostrate su un altro diagramma di sequenza.

18.5 Riepilogo

La realizzazione di caso d'uso della progettazione è, in realtà, solo un'estensione della realizzazione di caso d'uso dell'analisi. Sono stati spiegati i seguenti concetti.

1. La realizzazione di caso d'uso della progettazione, è la collaborazione tra gli oggetti e le classi di progettazione che realizzano un caso d'uso. Comprendono:
 - diagrammi di interazione di progettazione: rifiniture dei diagrammi di interazione di analisi;
 - diagrammi delle classi di progettazione: rifiniture dei diagrammi delle classi di analisi.
2. I diagrammi di interazione di progettazione possono essere utilizzati per modellare meccaniche centrali, quali la persistenza degli oggetti; queste meccaniche sono trasversali a molti casi d'uso.
3. I diagrammi di interazione a livello di sottosistema mostrano l'interazione ad alto livello tra gli attori e le diverse parti del sistema:
 - possono contenere attori, sottosistemi e classi;
 - sono solitamente espressi in forma descrittore: la forma istanza in questo caso è troppo specifica.

Diagrammi di stato

19.1 Contenuto del capitolo

Questo capitolo tratta dei diagrammi di stato. Questi rappresentano uno strumento importante per la modellazione del comportamento dinamico di entità reattive. Il capitolo inizia con un'introduzione ai diagrammi di stato (Sezioni 19.2 e 19.3) per poi focalizzarsi sui componenti fondamentali dei diagrammi di stato: gli stati (Sezione 19.5), le transizioni (Sezione 19.6), e gli eventi (Sezione 19.7).

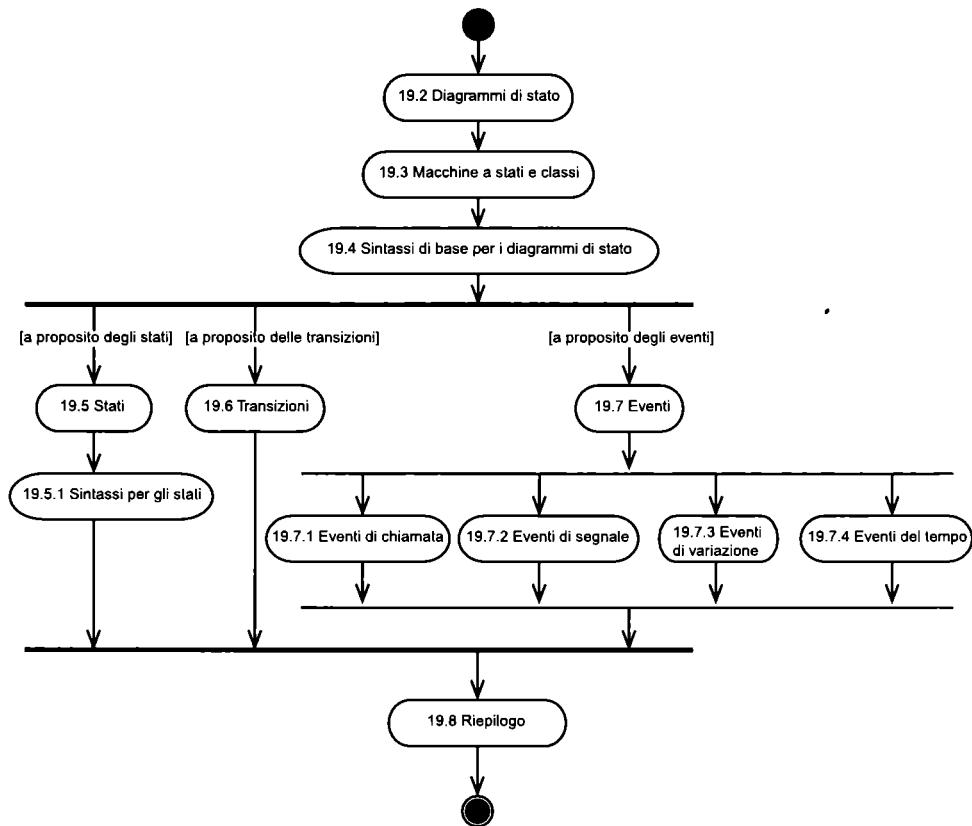
19.2 Diagrammi di stato

In realtà, in questo libro, sono già stati trattati dei diagrammi di stato. I diagrammi di attività sono un tipo speciale di diagrammi di stato in cui gli stati sono stati di azione o di sottoattività, e le transizioni sono attivate automaticamente al completamento delle azioni e delle attività di uno stato. Per questo motivo, i diagrammi di attività tipicamente utilizzano solo un piccolo sottoinsieme della sintassi UML per i diagrammi di stato.

Anche se entrambi i tipi di diagramma (di stato e di attività) modellano il comportamento dinamico di alcuni aspetti del sistema, tale modellazione ha scopi fondamentalmente diversi. I diagrammi di attività vengono solitamente utilizzati per modellare processi di *business* a cui partecipano diversi oggetti. D'altra parte, i diagrammi di stato vengono solitamente utilizzati per modellare il ciclo di vita di un'unica entità reattiva.

Un'entità reattiva è un oggetto (in senso lato) che fornisce il contesto a un diagramma di stato. Un'entità reattiva:

- risponde a eventi esterni (eventi che nascono al di fuori del contesto dell'entità);
- ha un ciclo di vita definito, che può essere modellato come una successione di stati, transizioni ed eventi;
- ha un comportamento corrente che dipende dai comportamenti precedenti.

**Figura 19.1**

Un diagramma di stato contiene una, e una sola, macchina a stati per ogni singola entità reattiva. Il mondo reale è pieno di esempi di entità reattive che possono essere modellate utilizzando macchine a stati. Nella modellazione OO le macchine a stati possono essere utilizzate per modellare il comportamento dinamico di entità reattive quali:

- classi;
- casi d'uso;
- sottosistemi;
- sistemi interi.

Tuttavia le macchine a stati sono solitamente utilizzate per modellare il comportamento dinamico di classi, ed è quindi questo il tipo di modellazione che sarà approfondito nelle seguenti sezioni.

19.3 Macchine a stati e classi

Per ogni classe può esistere *una* macchina a stati che modella tutte le transizioni di stato di tutti gli oggetti di quella classe, in risposta ai diversi tipi di evento. Gli eventi sono tipicamente dei messaggi inviati da altri oggetti, anche se in alcuni casi l'oggetto potrebbe generare internamente un evento di risposta al tempo che scorre.

Dato che gli oggetti di una classe possono essere presenti in molti casi d'uso, la macchina a stati di una classe modella il comportamento di oggetti di tale classe trasversalmente, per tutti i casi d'uso interessati.

Per spiegare le macchine a stati, si esamina un esempio molto semplice, preso dal mondo reale. Un oggetto molto comune nel mondo reale che cicla continuamente, e in modo ovvio, attraverso una macchina a stati, è la lampadina. La Figura 19.2 illustra come sia possibile inviare eventi a una lampadina utilizzando un interruttore. I due eventi che possono essere inviati sono **accendi** (questo evento modella l'attivazione della corrente elettrica che raggiunge la lampadina) e **spegni** (che modella l'interruzione della corrente).

Ogni macchina a stati ha uno stato iniziale (disco nero), che indica da dove parte la macchina, e uno stato finale (disco nero bordato, o bersaglio) che indica dove finisce la macchina. Tipicamente esiste una transizione automatica tra lo pseudo-stato iniziale e il primo "vero" stato della macchina a stati. Lo pseudo-stato iniziale viene di fatto utilizzato come semplice evidenziatore dell'inizio della serie di transizioni di stato.

Nella Figura 19.2, quando l'interruttore viene girato in posizione "On", alla lampadina viene inviato l'evento **accendi**. Nelle macchine a stati gli eventi sono da considerarsi istantanei. In altre parole, il tempo che trascorre tra quando l'interruttore invia l'evento e quando la lampadina lo riceve, è nullo. Gli eventi istantanei consentono di operare una notevole semplificazione della teoria delle macchine a stati, semplificazione che rende tale teoria molto più facilmente applicabile. Senza gli eventi istantanei potremmo facilmente ritrovarci in situazioni di concorrenza, in cui due eventi partono dalle rispettive fonti per raggiungere la medesima entità reattiva. La modellazione di queste situazioni di concorrenza richiederebbe delle macchine a stati infinitamente più complesse.

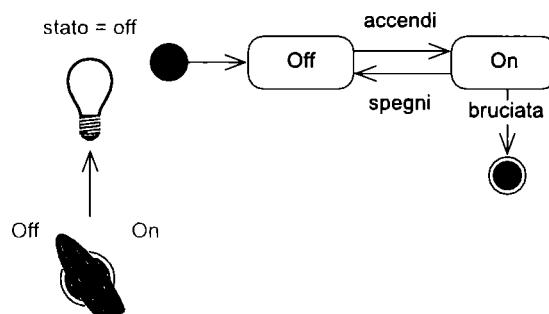


Figura 19.2

La lampadina riceve l'evento accendi e, in risposta all'evento, passa allo stato On. Ed è proprio questo l'elemento principale delle macchine a stati: gli oggetti possono cambiare stato quando ricevono un evento. Quando la lampadina riceve l'evento spegni, torna nello stato Off.

In un qualunque momento in cui la lampadina è in stato On, potrebbe ricevere l'evento bruciata (perché si può bruciare la lampadina): questo evento termina la macchina a stati.

19.4 Sintassi di base per i diagrammi di stato

La sintassi di base per i diagrammi di stato è semplice, come illustrato nella Figura 19.2.

- Gli stati sono rappresentati con rettangoli arrotondati, tranne lo stato iniziale (disco nero) e lo stato finale (disco nero bordato o bersaglio).
- Le transizioni indicano un possibile percorso tra due stati e sono modellate con frecce.
- Gli eventi sono scritti sopra la transizione che attivano.

La semantica di base è altrettanto semplice. Quando un'entità reattiva che si trova nello stato A riceve l'evento unEvento, può effettuare una transizione allo stato B. Ogni diagramma di stato dovrebbe avere uno stato iniziale (disco nero) che indica il primo stato della sequenza. A meno che esista un ciclo perpetuo di stati, i diagrammi di stato dovrebbero avere anche uno stato finale (disco nero bordato o bersaglio) il quale termina la sequenza delle transizioni.

Nelle prossime sezioni si discute in modo più approfondito di ciascuno dei tre elementi dei diagrammi di stato.

19.5 Stati

The UML Reference Manual [Rumbaugh 1] definisce uno stato come: “una condizione o situazione della vita di un oggetto durante la quale tale oggetto soddisfa una condizione, esegue un’attività o aspetta un qualche evento”. Lo stato di un oggetto varia nel corso del tempo, ma in un qualunque momento è determinato da:

- i valori dei suoi attributi;
- le relazioni che ha con altri oggetti;
- le attività che sta eseguendo.

Nel corso del tempo gli oggetti si scambiano messaggi, e questi messaggi sono eventi che possono provocare una transizione di stato degli oggetti. È importante decidere quale debba essere esattamente il significato di “stato”. Nell'esempio della lampadina, potremmo decidere (se fossimo fisici delle particelle) che ogni più piccola variazione

subita da uno degli atomi, o delle particelle subatomiche, della lampadina ne costituisce un cambio di “stato”. Tutto ciò sarebbe probabilmente preciso, ma ci obbligherebbe a gestire un’infinità di stati, la maggior parte dei quali sarebbero praticamente identici.

D’altra parte, dal punto di vista dell’utente della lampadina, gli unici stati che contano sono On (accesa) e Off (spenta). Questo è il segreto per una buona modellazione dei diagrammi di stato: è necessario individuare gli stati che rivestono *interesse* per il sistema e per i suoi utenti. Si consideri il seguente esempio:

```
class Colore
{
    int rosso;
    int verde;
    int blu;
}
```

Anche partendo dal presupposto che rosso, verde e blu possano ciascuno avere valori limitati all’intervallo 0–255, comunque, basandosi sui soli valori possibili degli attributi, gli oggetti di questa classe possono esistere in $256 \times 256 \times 256 = 16.777.216$ possibili stati! Certo di tratterebbe di un gran bel diagramma di stato! È, invece, importante porsi la seguente domanda: qual è la principale differenza semantica tra tutti questi stati? La risposta è che non c’è alcuna differenza semantica. Ciascuno dei 16.777.216 possibili stati rappresenta solo un colore, ed è tutto qui. In effetti, il diagramma di stato di questa classe è piuttosto poco interessante, dato che tutte le possibilità possono essere modellate con un unico stato.

Riassumendo, affinché sia rilevante modellare degli stati in una macchina a stati, deve esistere tra loro una “differenza che sia anche una differenza semantica”.

19.5.1 Sintassi per gli stati

La Figura 19.3 illustra la sintassi UML per gli stati.

Ogni stato può contenere zero o più azioni e attività. Le azioni sono considerate istantanee e non interrompibili, mentre l’esecuzione di un’attività richiede una quantità finita di tempo, per quanto piccola, e può essere interrotta. In uno stato, ogni azione è associata con una transizione interna attivata da un evento. Uno stato può contenere qualunque numero di azioni e transizioni interne.

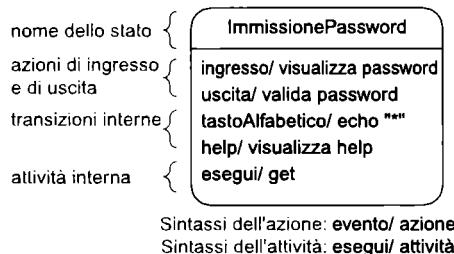


Figura 19.3

Le transizioni interne consentono di evidenziare che è successo qualcosa che vale la pena di modellare, ma che non provoca la transizione a un nuovo stato (o che non è sufficientemente importante da essere modellato in questo modo). Nell'esempio della Figura 19.3, la digitazione di un tasto alfabetico della tastiera è sicuramente un evento degno di modellazione, ma non provoca una transizione a uno stato diverso da `ImmissionPassword`. Viene, quindi, modellato con un evento interno, `tastoAlfabetico`, che provoca una transizione interna che attiva l'azione `echo "*"`.

Esistono poi due azioni speciali, l'azione di ingresso e l'azione di uscita, che sono associate agli eventi speciali ingresso e uscita. Questi due eventi hanno una semantica particolare. L'evento ingresso avviene istantaneamente e automaticamente all'ingresso nello stato: è la prima cosa che succede quando si entra in questo stato, e provoca l'esecuzione dell'azione di ingresso associata. L'evento uscita è l'ultimissima cosa che succede istantaneamente e automaticamente all'uscita dallo stato, e provoca l'esecuzione dell'azione di uscita associata.

L'esecuzione di un'attività, invece, dura un intervallo di tempo finito, ma non nullo, e può essere interrotta dalla ricezione di un evento. La parola chiave `esegui` indica un'attività. Mentre le azioni, in quanto atomiche, terminano sempre la loro esecuzione, è invece possibile interrompere un'attività prima che questa abbia finito l'esecuzione.

19.6 Transizioni

La Figura 19.4 illustra la sintassi UML per le transizioni.

Le transizioni hanno una sintassi semplice che può essere utilizzata per transizioni esterne (indicate con una freccia) e per transizioni interne (annidate all'interno di uno stato). Ogni transizione ha tre elementi opzionali:

- un evento: l'accadimento esterno o interno che attiva la transizione;
- una condizione di guardia: un'espressione Booleana che deve risultare vera perché la transizione possa avvenire;
- un'azione: una qualche conseguenza della transizione, che viene eseguita quando scatta la transizione.

La Figura 19.4 può essere letta nel seguente modo: “Quando avviene `unEvento`, se la `condizioneGuardia` risulta vera, allora `esegui unaAzione`, quindi entra immediatamente nello stato B.”

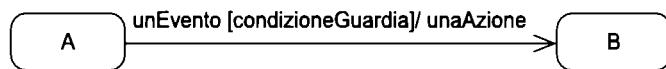


Figura 19.4

19.7 Eventi

L'UML definisce un evento come: "la specifica di un'occorrenza di interesse che ha una collocazione nel tempo e nello spazio." Gli eventi attivano le transizioni nelle macchine a stati. Gli eventi possono essere riportati esternamente sulle transizioni, come illustrato nella Figura 19.5 o internamente, dentro gli stati.

Esistono quattro tipi di evento, ognuno dei quali ha una diversa semantica:

- evento di chiamata;
- evento di segnale;
- evento di variazione;
- evento del tempo.

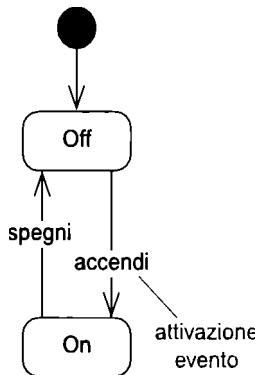


Figura 19.5

19.7.1 Eventi di chiamata

L'evento di chiamata è forse il tipo di evento più semplice. La ricezione di un evento di chiamata equivale a una richiesta di esecuzione di un insieme di azioni. Un evento di chiamata dovrebbe avere la segnatura identica a quella di uno dei metodi della classe che costituisce il contesto della macchina a stati. L'esempio nella Figura 19.6 riporta un frammento della macchina a stati relativa a una semplice classe ContoBancario. Gli eventi di chiamata deposito(...) e prelievo(...) corrispondono a metodi della classe ContoBancario.

È possibile associare a un evento di chiamata una sequenza di azioni, separate da punto e virgola. Queste azioni possono utilizzare attributi e metodi della classe di contesto. L'evento di chiamata può avere un valore restituito, ma questo deve corrispondere al tipo restituito dall'operazione corrispondente.

In altre parole, un evento di chiamata è solo una richiesta di esecuzione di un particolare metodo su un'istanza della classe di contesto.

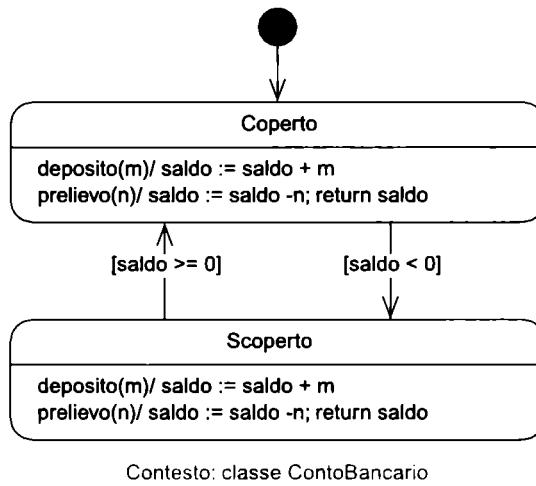


Figura 19.6

19.7.2 Eventi di segnale

Un segnale è un pacchetto di informazioni inviato in modo asincrono da un oggetto a un altro. Il segnale viene modellato come una classe con stereotipo, la quale ha attributi corrispondenti a tutte le informazioni comunicate. La Figura 19.7 illustra un esempio di segnale.

Dato che un segnale permette solo il passaggio di informazioni tra oggetti diversi, non può avere alcuna operazione, fatta eccezione per l'operazione implicita `invia(elencoDestinatari)` che consente l'invio del segnale a uno o più oggetti destinatari. L'uso dei segnali non è una pratica molto *object-oriented*, dato che in realtà i segnali non hanno altri comportamenti. L'invio di un segnale può essere indicato nel seguente modo:

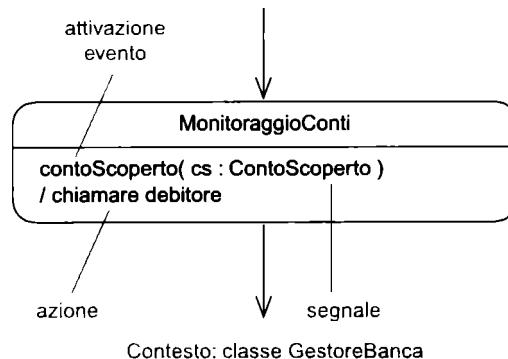
`ContoScoperto.invia(gestoreBanca)`

`ContoScoperto` è il nome del segnale, e il metodo implicito `invia(...)` in questo caso ha come argomento l'oggetto `gestoreBanca`. Questo oggetto è il destinatario di `invia(...)`. È possibile specificare una lista, separata da virgola, di oggetti destinatari. È anche possibile includere come argomento una o più *classi*, e in questo caso il segnale viene inviato a tutti gli oggetti di queste classi.

La ricezione di un segnale da parte di un'entità reattiva può essere modellata come un evento di segnale. L'attivazione di un evento di segnale è un metodo che prende come parametro quel tipo di segnale, come illustrato nella Figura 19.8.



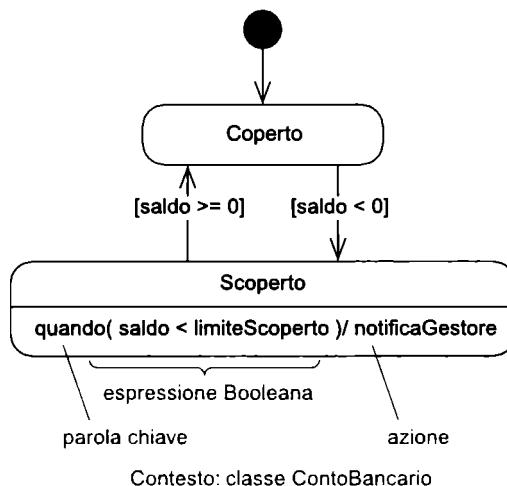
Figura 19.7

**Figura 19.8**

19.7.3 Eventi di variazione

L'evento di variazione contiene la parola chiave quando seguito da un'espressione Booleana; la Figura 19.9 ne illustra un esempio. L'azione associata a questo tipo di evento viene eseguita quando l'espressione Booleana risulta vera. Tutti i valori su cui si basa l'espressione Booleana devono essere attributi della classe di contesto. Dal punto di vista implementativo, un evento di variazione implica un ciclo di test perpetuo per tutta la durata dello stato.

Gli eventi di variazione sono attivati solo da cambiamenti. Questo significa che sono attivati quando la condizione Booleana, specificata nella clausola quando, *diventa* vera per la prima volta. Perché venga nuovamente attivato l'evento, la condizione Booleana deve prima tornare a essere falsa e, quindi, tornare a essere nuovamente vera.

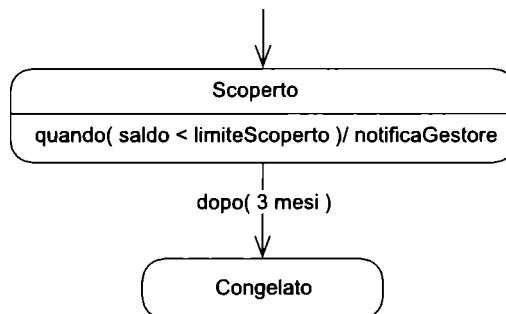
**Figura 19.9**

19.7.4 Eventi del tempo

Gli eventi del tempo sono indicati con le parole chiave quando e dopo. La parola chiave quando specifica un determinato *momento* nel tempo in cui verrà attivato l'evento; la parola chiave dopo specifica un *intervallo* di tempo al cui scadere verrà attivato l'evento. Per esempio, dopo(3 mesi) e quando(data = 07/10/2002).

È importante assicurarsi che le unità temporali (ore, giorni, mesi ecc.) siano sempre ben specificate per ciascun evento del tempo. Eventuali simboli contenuti nell'espressione (come data nell'esempio del paragrafo precedente) devono corrispondere ad attributi della classe di contesto.

La Figura 19.10 riporta un frammento della macchina a stati di una classe ContoBancario. In questo esempio, se un oggetto ContoBancario permane nello stato Scoperto per tre mesi, effettua una transizione allo stato Congelato.



Contesto: classe ContoBancario

Figura 19.10

19.8 Riepilogo

Questo capitolo ha spiegato come costruire diagrammi di stato semplici, utilizzando stati, azioni, attività, eventi e transizioni.

Sono stati spiegati i seguenti concetti.

1. I diagrammi di attività sono un tipo speciale di diagramma di stato in cui:
 - tutti gli stati sono stati di azione o di sottoattività;
 - le transizioni sono attivate automaticamente quando uno stato termina le proprie azioni e attività.
2. Un'entità reattiva fornisce il contesto a un diagramma di stato.
 - Le entità reattive:
 - rispondono a eventi esterni;
 - hanno un ciclo di vita definito che può essere modellato come una successione di stati, transizioni ed eventi;

- hanno un comportamento corrente che dipende dai comportamenti precedenti.
 - Le macchine a stati modellano il comportamento dinamico delle entità reattive: ogni entità reattiva corrisponde a una, e una sola, macchina a stati.
 - Esempi di entità reattive:
 - classi (più comune);
 - casi d'uso;
 - sottosistemi;
 - sistemi interi.
3. Azioni: frammenti di lavoro la cui esecuzione è istantanea e che non può essere interrotta:
- possono avvenire all'interno di uno stato, se associati a una transizione interna;
 - possono avvenire all'esterno di uno stato, se associati a una transizione esterna.
4. Attività: frammenti di lavoro la cui esecuzione richiede una quantità finita di tempo e che può essere interrotta; possono avvenire esclusivamente all'interno di uno stato.
5. Stato: una possibile condizione di un oggetto, semanticamente significativa.
- Lo stato di un oggetto è determinato da:
 - i valori degli attributi dell'oggetto;
 - le relazioni con altri oggetti;
 - le attività che l'oggetto sta eseguendo.
 - Sintassi per gli stati:
 - azione di ingresso: eseguita immediatamente, all'ingresso nello stato;
 - azione di uscita: eseguita immediatamente, all'uscita dallo stato;
 - transizioni interne: sono provocate da eventi che non sono sufficientemente significativi da attivare una transizione dell'oggetto in un nuovo stato; l'evento è processato da una transizione interna allo stato;
 - attività interna: un frammento di lavoro la cui esecuzione richiede una quantità finita di tempo e che può essere interrotta.
6. Transizione: il passaggio da uno stato a un altro.
- Sintassi per le transizioni:
 - evento: l'evento che attiva la transizione;
 - condizione di guardia: un'espressione Booleana che deve risultare vera perché possa avvenire la transizione;

- azione: l'azione che viene eseguita immediatamente, all'attivazione della transizione.
7. Evento: qualcosa di significativo che accade a un'entità reattiva. Esistono i seguenti tipi di evento:
- evento di chiamata:
 - una richiesta di esecuzione di un insieme di azioni;
 - la chiamata a un metodo dell'oggetto;
 - evento di segnale:
 - la ricezione di un segnale: un segnale è una comunicazione asincrona a senso unico tra due oggetti;
 - evento di variazione:
 - avviene quando una qualche condizione Booleana passa dall'essere falsa all'essere vera;
 - evento del tempo:
 - avviene allo scadere di un determinato intervallo di tempo (dopo);
 - avviene in un certo momento (quando).

Diagrammi di stato: tecniche avanzate

20.1 Contenuto del capitolo

Il capitolo inizia affrontando la questione degli stati composti. Si tratta di stati che a loro volta contengono una macchina a stati annidata. Il Paragrafo 20.2 introduce il concetto di macchine a stati annidate, o sotto-macchine. Si discute poi di due tipi di stati composti: lo stato composto sequenziale (20.3) e lo stato composto concorrente (20.4).

Quando si hanno due o più sotto-macchine concorrenti, è spesso necessario stabilire una qualche forma di comunicazione tra loro. Questo argomento viene trattato nel Paragrafo 20.5, che introduce anche due strategie di comunicazione: l'uso di attributi (Sezione 20.5.1) o l'uso di stati di sincronizzazione (Sezione 20.5.2).

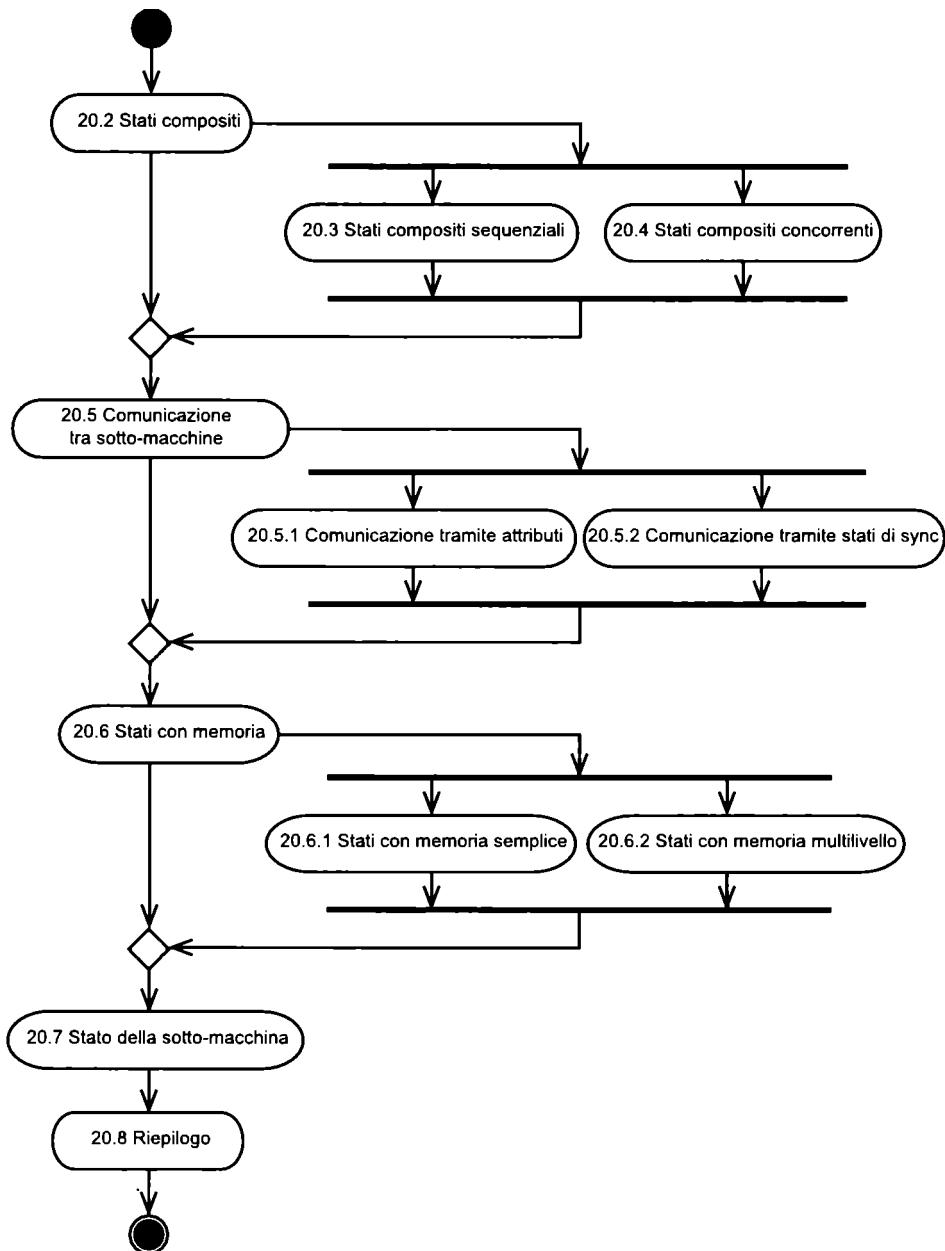
Nel Paragrafo 20.6 si discute del concetto di storia, ovvero dare a un superstato una "memoria" del suo sottostato finale prima di una transizione a uno stato esterno. Nelle Sezioni 20.6.1 e 20.6.2 vengono trattate due varianti di questa tecnica: memoria semplice e memoria multilivello.

Infine, nel Paragrafo 20.7, si parla di come sia possibile riepilogare il funzionamento di sotto-macchine annidate in un diagramma di stato, utilizzando gli stati della sotto-macchina.

20.2 Stati composti

Uno stato composto è uno stato che contiene una o più macchine a stati annidate: note anche come sotto-macchine. Nei diagrammi di stato, è possibile evidenziare gli stati composti aggiungendo l'indicatore di scomposizione all'icona dello stato, così come illustrato nella Figura 20.2.

È anche possibile indicare esplicitamente la scomposizione, riportando le macchine a stati annidate in una nuova sottosezione dell'icona dello stato: in questo caso, l'indicatore di scomposizione non serve.

**Figura 20.1**

Gli stati composti sono costituiti da stati componenti, o sottostati, dei quali lo stato composto è superstato. Ogni sottostato eredita tutte le transizioni del suo superstato. Le seguenti sezioni illustrano diversi esempi.

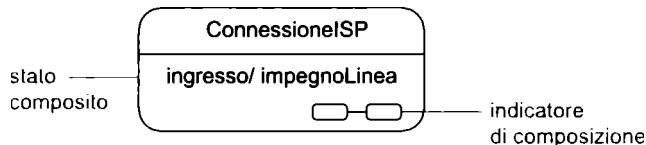


Figura 20.2

Ogni sottostato può a sua volta essere uno stato composito. Ovviamemente, esiste un limite al livello di annidamento, oltre il quale non ha senso inoltrarsi: per mantenere il modello comprensibile, non bisogna superare i due o tre livelli di annidamento, se non in casi del tutto eccezionali.

Se un superstato contiene un'unica macchina a stati annidata, allora si parla di stato composito sequenziale. Se, invece, un superstato contiene due o più macchine a stati, queste verranno eseguite parallelamente, e allora si parla di stato composito concorrente.

20.3 Stati composti sequenziali

Uno stato composito che contiene un'unica macchina a stati annidata viene chiamato stato composito sequenziale. La Figura 20.3 riporta l'esempio della macchina a stati di una classe chiamata ISPDialer. Questa classe è responsabile di comporre il numero telefonico e connettersi a un internet *service provider*.

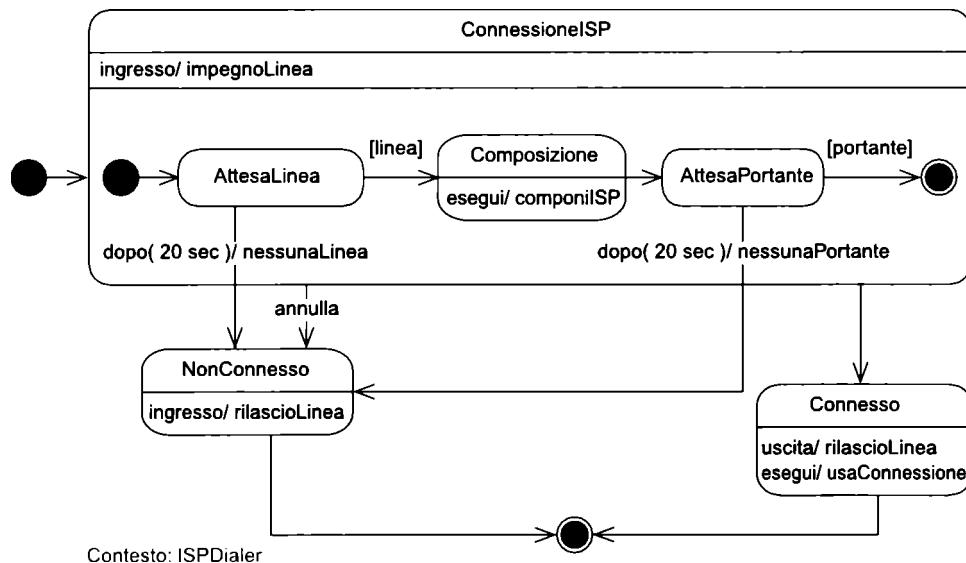


Figura 20.3

Ogni sottostato eredita tutte le transizioni del suo superstato, quindi, in questo caso, sia la transizione automatica dal superstato ConnessioneISP allo stato Connesso, sia la transizione attivata dall'evento **annulla**, vengono ereditate da ciascun sottostato. Questo è molto comodo, perché significa che ricevendo l'evento **annulla**, avverrà *sempre* una transizione dal sottostato corrente allo stato NonConnesso. L'uso di superstati e sottostati può semplificare enormemente un diagramma di stato.

Ecco lo schema di attraversamento completo della macchina a stati della classe ISPDialoger.

1. Entrando nel superstato ConnessioneISP viene immediatamente eseguita l'azione di ingresso: impegno linea.
2. Si entra nella macchina a stati annidata.
3. Si entra nello stato AttesaLinea.
 - 3.1. Si resta nello stato AttesaLinea per un massimo di 20 secondi.
 - 3.2. Se non si riesce a ottenere una linea entro questo tempo limite:
 - 3.2.1. viene eseguita l'azione nessunaLinea e avviene una transizione allo stato NonConnesso;
 - 3.2.2. entrando nello stato NonConnesso viene eseguito il rilascio della linea;
 - 3.2.3. avviene una transizione allo stato finale.
 - 3.3. Se si riesce ad avere una linea (la condizione di guardia [linea] risulta vera) entro 20 secondi:
 - 3.3.1. avviene una transizione allo stato Composizione in cui si esegue l'attività componiISP;
 - 3.3.2. appena termina l'attività componiISP, avviene una transizione allo stato AttesaPortante;
 - 3.3.3. si resta nello stato AttesaPortante per un massimo di 20 secondi.
 - 3.3.4. Se non si riesce a ottenere una portante entro 20 secondi:
 - 3.3.4.1. viene eseguita l'azione nessunaPortante e avviene una transizione allo stato NonConnesso;
 - 3.3.4.2. entrando nello stato NonConnesso viene eseguito il rilascio della linea;
 - 3.3.4.3. avviene una transizione allo stato finale.
 - 3.3.5. Se si riesce a ottenere una portante entro 20 secondi:
 - 3.3.5.1. avviene una transizione automatica dal superstato ConnessioneISP allo stato Connesso;
 - 3.3.5.2. viene eseguita l'azione usaConnessione fin quando non termina;

3.3.5.3. in uscita dallo stato Connesso viene eseguito il rilascio della linea;

3.3.5.4. avviene una transizione allo stato finale.

4. Se in *qualsiasi* momento in cui si è nel superstato ConnessioneISP, riceviamo l'evento annulla, avviene una transizione immediata allo stato NonConnesso.

4.1. Entrando nello stato NonConnesso viene eseguito il rilascio della linea.

4.2. Avviene una transizione allo stato finale.

20.4 Stati composti concorrenti

Uno stato composto concorrente è costituito da due o più sotto-macchine che eseguono parallelamente. Quando si entra in un superstato, tutte le sotto-macchine iniziano a eseguire in parallelo: questa viene detta biforcazione. A questo punto, esistono due modi per uscire dal superstato:

- tutte le sotto-macchine terminano l'esecuzione: in questo caso si parla di ricongiunzione;
- una delle sotto-macchine riceve un evento, o termina l'esecuzione, ed esegue una transizione esplicita a uno stato *esterno* al superstato. Questa *non* è una ricongiunzione: non avviene alcuna sincronizzazione delle sotto-macchine; le altre sotto-macchine vengono semplicemente terminate.

Per meglio studiare gli stati composti concorrenti, è necessario prendere in considerazione un sistema che esibisca un certo grado di parallelismo. Si modelli, quindi, un semplice sistema di sicurezza, costituito da un apparato di controllo, alcuni rilevatori di intrusione e di incendio e un allarme. La Figura 20.4 riporta la macchina a stati del sistema.

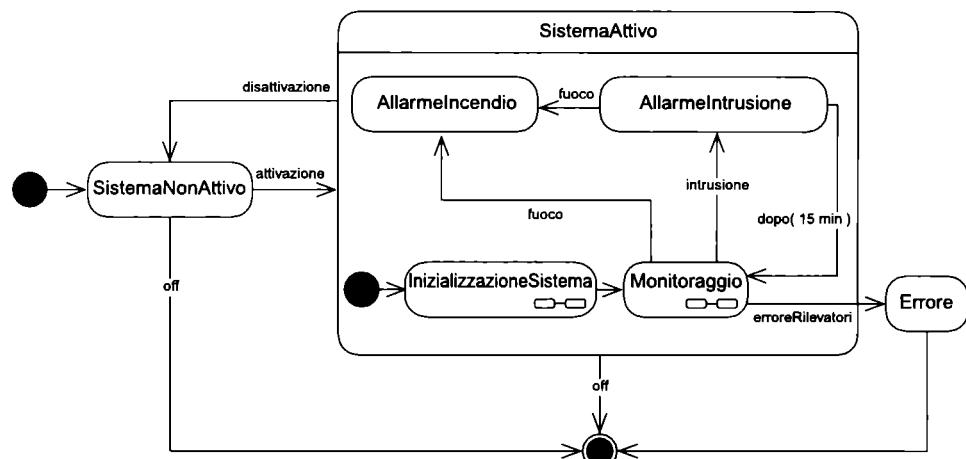


Figura 20.4

Nel sistema esistono due stati composti concorrenti, InizializzazioneSistema e Monitoraggio. La Figura 20.5 illustra il diagramma di stato dello stato InizializzazioneSistema. Quando si entra in questo stato, c'è una biforcazione in seguito alla quale due sotto-macchine distinte iniziano l'esecuzione in parallelo. Nella sotto-macchina illustrata in alto, lo stato InizializzazioneRilevatorilIncendio esegue il processo di inizializzazione dei rilevatori di incendio. Nella sotto-macchina illustrata in basso, lo stato InizializzazioneRilevatorIntrusione esegue la stessa operazione per i rilevatori di intrusione. Si può uscire dal superstato InizializzazioneSistema solo quando entrambe le sotto-macchine hanno terminato l'esecuzione. Questa è una ricongiunzione; le due sotto-macchine vengono sincronizzate in modo tale che non è possibile procedere oltre a meno che siano stati inizializzati sia i rilevatori di intrusione, sia quelli di incendio. Questo processo di inizializzazione dipende dal tipo di rilevatori utilizzati e può trattarsi, nei casi più semplici, anche solo di una breve fase di riscaldamento.

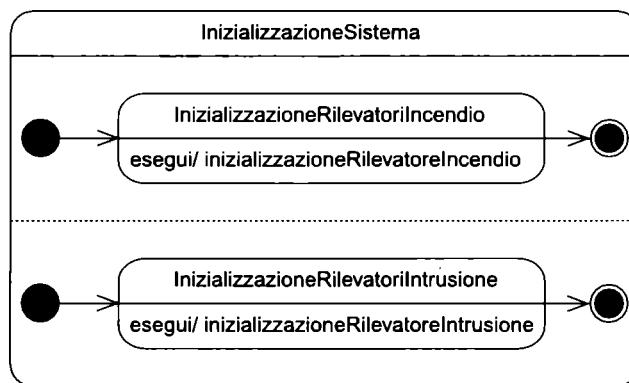


Figura 20.5

A volte si vuole far partire dei processi paralleli di controllo, senza però avere l'esigenza di sincronizzarli con una ricongiunzione quando sono terminati. Lo stato composito concorrente Monitoraggio, illustrato nel diagramma di stato della Figura 20.6, si comporta in questo modo.

Questo diagramma di stato ha alcune caratteristiche di interesse.

1. *Non esiste una sincronizzazione tra le due sotto-macchine:*
 - l'evento fuoco attiva una transizione esplicita da MonitoraggiolIncendio a AllarmeIncendio, la quale comporta l'uscita dal superstato Monitoraggio: l'altra sotto-macchina viene terminata;
 - similmente, l'evento intrusione attiva una transizione esplicita da MonitoraggiolIntrusione a AllarmeIntrusione, la quale comporta l'uscita dal superstato Monitoraggio: l'altra sotto-macchina viene terminata.
2. Quando scatta il rilevatore di incendio viene attivato l'allarme antincendio, il quale continua a suonare fin quando il sistema non viene reinizializzato manualmente.

3. Quando scatta il rilevatore di intrusione viene attivato l'allarme antifurto, il quale continua a suonare per 15 minuti, passati i quali il sistema torna nello stato Monitoraggio: per rispettare le leggi locali sull'inquinamento acustico.
4. Se subentra un evento fuoco mentre l'allarme antifurto sta suonando, avviene una transizione immediata allo stato AllarmeIncendio, e l'allarme antincendio comincia a suonare. Questo significa che l'allarme antincendio ha la precedenza sull'allarme antifurto.

Questo esempio mostra come l'utilizzo di stati composti concorrenti, con o senza sincronizzazione, consenta di modellare il parallelismo in modo efficace.

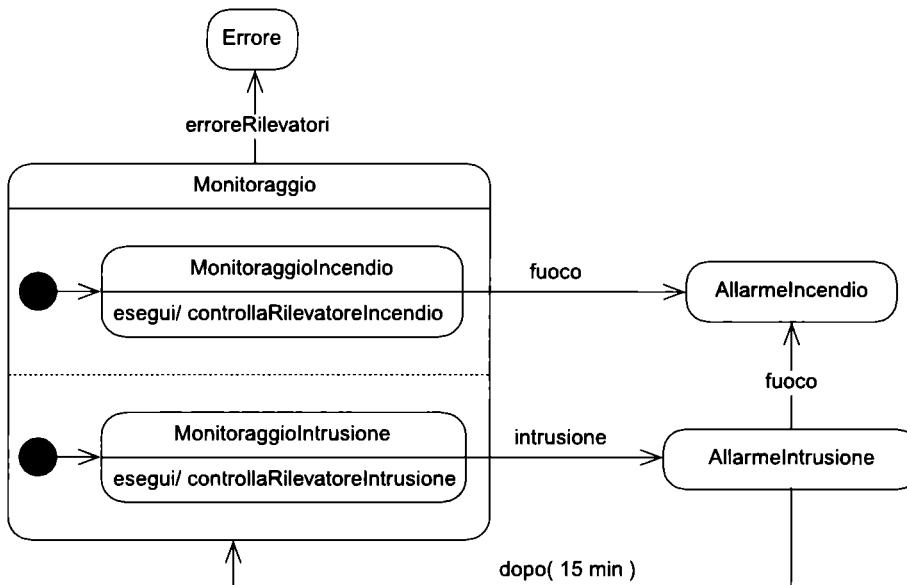


Figura 20.6

20.5 Comunicazione tra sotto-macchine

La Figura 20.5 ha mostrato come si possono utilizzare le biforazioni e le ricongiunzioni per creare sotto-macchine concorrenti e per risincronizzarle. Si tratta di una forma di comunicazione sincrona tra sotto-macchine: le sotto-macchine concorrenti aspettano fin quando non hanno *tutte* quante terminate l'esecuzione.

Capita spesso, invece, di avere l'esigenza di far comunicare due sotto-macchine, senza però volerle sincronizzare. In questo caso, si parla di comunicazione asincrona.

L'UML prevede la comunicazione asincrona consentendo a una sotto-macchina di lasciarsi dietro dei "messaggi" o degli "indicatori", pur continuando l'esecuzione. Le altre sotto-macchine possono leggere questi indicatori quando hanno tempo per farlo.

Nell'UML esistono due meccanismi per modellare questo tipo di comunicazione: gli attributi dell'entità reattiva e gli *stati di sync*. Le seguenti due sezioni trattano in modo più approfondito questi due meccanismi.

20.5.1 Comunicazione tramite attributi

Il meccanismo di comunicazione tramite attributi prevede che l'entità reattiva che si modella abbia un insieme di attributi, ai cui valori le sue sotto-macchine possono accedere in lettura e scrittura. La strategia su cui si basa la comunicazione tramite attributi è che una sotto-macchina imposta i valori degli attributi, mentre le altre sotto-macchine referenziano tali valori nelle condizioni di guardia delle loro transizioni.

Nello stato *ProcessoOrdini*, illustrato nella Figura 20.7, non si può sapere se di un dato ordine verrà completata prima la verifica del pagamento o la preparazione. La preparazione di alcuni ordini potrebbe richiedere l'attesa di una fornitura, mentre per altri la merce potrebbe essere già tutta disponibile. La verifica di alcuni pagamenti potrebbe essere più o meno istantanea (per esempio, di quelli effettuati tramite carta di credito), mentre in altri casi potrebbe essere necessario aspettare diversi giorni lavorativi (per esempio, per i pagamenti tramite assegno). Esiste una regola di *business* che crea una dipendenza logica tra le due sotto-macchine: l'ordine non può entrare in consegna se non è stato prima preparato o se il pagamento non è stato ancora verificato.

Nella Figura 20.7, nella sotto-macchina in alto, in uscita dallo stato *VerificaPagamento* avviene una transizione allo stato *Pagato*, in cui il valore dell'attributo *pagato* viene impostato a vero. Nella sotto-macchina in basso, quando si completa lo stato *PreparazioneOrdine*, può avvenire una transizione al nuovo stato *ConsegnaOrdine*, ma solo se l'attributo *pagato* risulta vero. Si è, quindi, stabilita una comunicazione asincrona tra le due sotto-macchine, utilizzando come indicatore un attributo che viene impostato da una delle sotto-macchine, e verificato dall'altra sotto-macchina: questo meccanismo è molto semplice e comune. Infine, entrambe le sotto-macchine terminano la loro esecuzione e si sincronizzano, e si può quindi uscire dallo stato *ProcessoOrdini*.

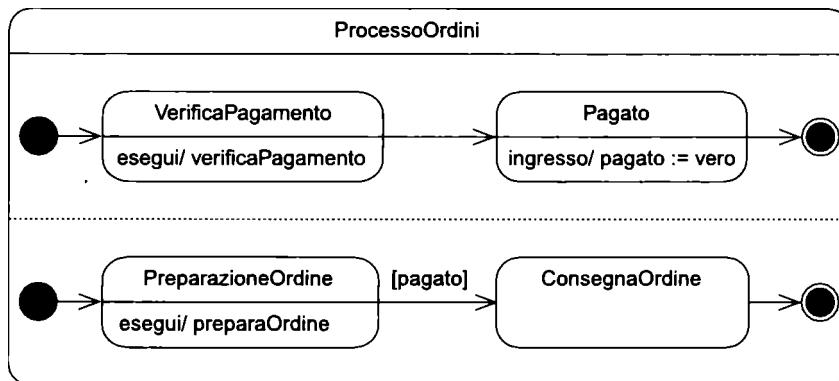


Figura 20.7

20.5.2 Comunicazione tramite *stati di sync*

Lo *stato di sync* è uno stato speciale il cui compito è quello di tenere traccia di ogni singola attivazione della sua unica transizione di input. Lo *stato di sync* è come una coda: si aggiunge un elemento alla coda ogni volta che viene attivata la transizione di input. Si rimuove un elemento ogni volta che avviene una transizione in uscita dallo *stato di sync*.

È possibile dichiarare che la coda deve avere un numero limitato di elementi, specificando all'interno dello *stato di sync* un numero intero positivo. Un asterisco all'interno dello *stato di sync* indica, invece, che esso può contenere un numero illimitato di elementi. La Figura 20.8 illustra la sintassi UML per gli *stati di sync*.

Gli *stati di sync* costituiscono un strumento molto flessibile per consentire alle sotto-macchine concorrenti di comunicare tra loro. Una sotto-macchina può “lasciare un messaggio” in uno *stato di sync* in modo che le altre sotto-macchine lo possano processare quando gli fa più comodo. Le sotto-macchine *non* devono necessariamente essere sincronizzate prima che la comunicazione possa avvenire: la comunicazione tramite *stati di sync* è asincrona, proprio come quella tramite attributi. Una sotto-macchina lascia un messaggio in uno *stato di sync*, e continua per la sua strada. Le altre sotto-macchine leggono il messaggio quando sono pronte per farlo.

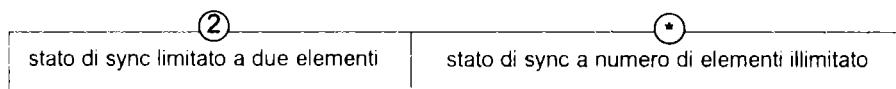


Figura 20.8

La Figura 20.9 riporta lo stesso esempio già visto nella Figura 20.7, in cui lo stesso identico comportamento viene modellato utilizzando uno *stato di sync*. Quando avviene la transizione dallo stato VerificaPagamento allo stato Pagato, questa transizione resta registrata nello *stato di sync*. Quando si conclude lo stato PreparazioneOrdine, si passa alla ricongiunzione. Se nello *stato di sync* è presente la registrazione della transizione dallo stato VerificaPagamento allo stato Pagato, la ricongiunzione avviene e si può procedere allo stato ConsegnatOrdine. In caso contrario, è necessario aspettare fin quando tale registrazione non compare nello *stato di sync*.

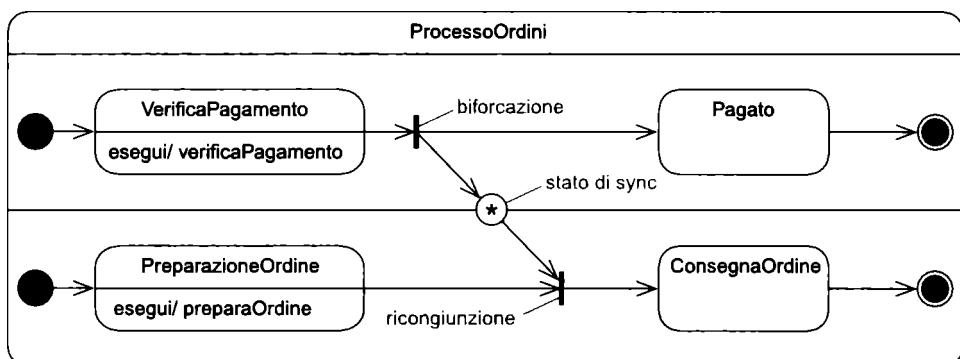


Figura 20.9

Uno dei vantaggi di utilizzare gli *stati di sync* al posto dei valori degli attributi, è che gli *stati di sync* non dicono nulla su *come* debba effettivamente essere implementata la comunicazione asincrona. Questa comunicazione può essere implementata utilizzando attributi, semafori, un *middleware* orientato ai messaggi (*message-oriented middleware*, MOM), oppure un *middleware* di tipo completamente diverso: lo *stato di sync* modella esclusivamente la semantica della comunicazione, e *non* i dettagli della sua implementazione. Inoltre, gli *stati di sync* consentono di modellare una coda che può contenere un numero limitato di elementi. Con i valori degli attributi sarebbe più difficile modellare questi tipi di comportamento.

20.6 Stati con memoria

Modellando macchine a stati capita spesso la seguente situazione.

- La macchina si trova nel sottostato A di uno stato composito.
- Avviene una transizione a uno stato esterno allo stato composito (e quindi anche al suo sottostato A).
- Si passa attraverso uno o più altri stati esterni.
- Avviene una transizione che riporta allo stato composito, *ma* si vorrebbe poter continuare l'esecuzione dal suo sottostato A dove è stata interrotta.

Come si può risolvere questa situazione? Pare ovvio che lo stato composito deve in qualche modo ricordarsi del sottostato in cui si trovava quando è avvenuta la transizione verso lo stato esterno. Questo requisito di poter riprendere l'esecuzione da dove era stata interrotta, è così comune che l'UML prevede un apposito elemento di sintassi: l'indicatore di stato con memoria.

Esistono due tipi di indicatori di stato con memoria: semplice e multilivello. Le prossime due sezioni li trattano in modo più approfondito.

20.6.1 Stati con memoria semplice

La Figura 20.10 mostra il diagramma di stato del caso d'uso ConsultaCatalogo di un sistema di *e-commerce*.

In questo esempio è possibile uscire dal superstato ConsultaCatalogo con tre diversi eventi:

- **uscita**: la macchina a stati viene terminata (non è necessario approfondire questo evento);
- **vaiACarrello**: transizione allo stato composito VisualizzazioneCarrello, in cui viene visualizzato il contenuto aggiornato del carrello della spesa;
- **vaiACassa**: transizione allo stato composito OperazioniCassa, in cui si presenta al cliente un riepilogo articoli ordinati e si conclude l'acquisto.

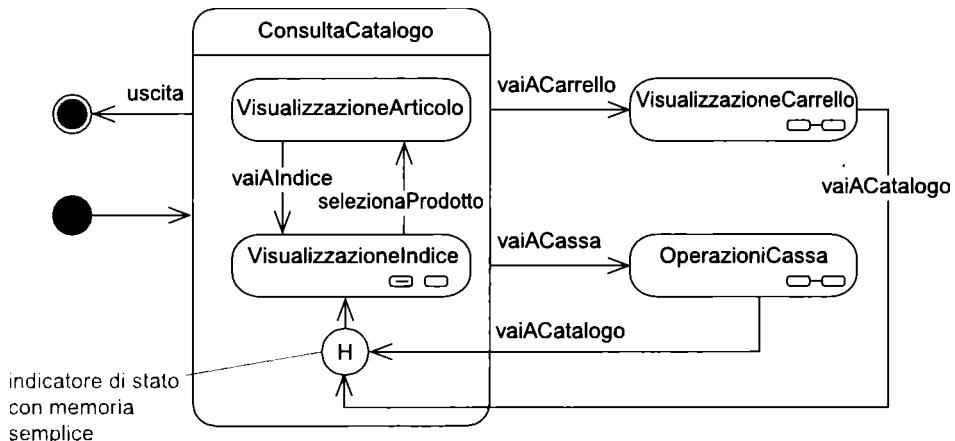


Figura 20.10

Tornando allo stato **ConsultaCatalogo** da **VisualizzazioneCarrello** o da **OperazioniCassa**, sarebbe preferibile per l’utente poter tornare sulla stessa pagina del catalogo da è partito: sembra un requisito ragionevole.

L’indicatore di stato può avere molte transizioni in ingresso, ma una sola in uscita. L’indicatore si ricorda del sottostato attivo quando è avvenuta l’ultima transizione in uscita dal superstato. In seguito, quando si torna da uno stato esterno allo stato con memoria, l’indicatore solitamente ridireziona la transizione sull’ultimo sottostato memorizzato. Se si tratta della prima volta che si entra nel superstato, allora non esiste memoria dell’ultimo sottostato attivo. In questo caso, viene attivata l’unica transizione in uscita dall’indicatore dello stato con memoria; nell’esempio illustrato si procederebbe, dunque, al sottostato **VisualizzazioneIndice**.

Gli stati con memoria consentono a un superstato di ricordarsi il sottostato attivo quando è avvenuta l’ultima transizione in uscita dal superstato. Tuttavia, come si può vedere nella Figura 20.10, anche il sottostato **VisualizzazioneIndice** è composito. Uno stato con memoria semplice non può ricordarsi dei sottostati di un suo *sottostato*: per questo ci vuole una memoria multilivello.

20.6.2 Stati con memoria multilivello

Uno stato con memoria multilivello può non solo ricordarsi l’ultimo sottostato attivo allo stesso livello dell’indicatore, ma anche l’eventuale ultimo sottostato attivo di questo sottostato attivo... e così via, per qualunque numero di stati annidati. Nell’esempio della Figura 20.11 non si torna, quindi, semplicemente all’indice o alla scheda articolo, ma anche sull’ultimo tipo di indice utilizzato (alfabetico o per categoria) e all’ultima pagina consultata. È possibile modellare lo stesso comportamento senza l’uso di stati con memoria multilivello, ma è estremamente più complesso.

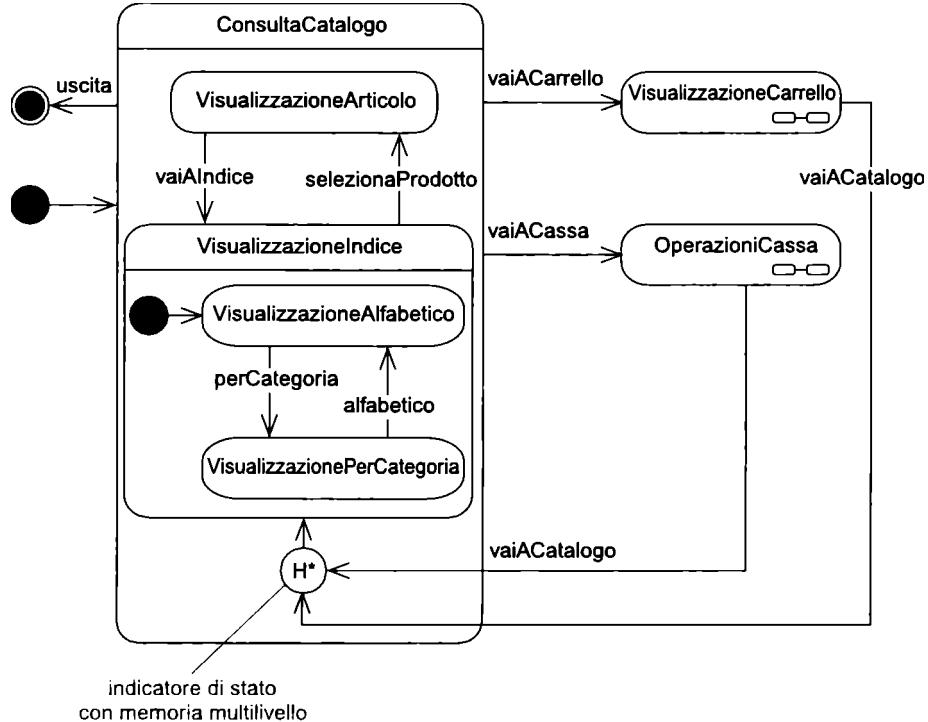


Figura 20.11

20.7 Stato della sotto-macchina

In situazioni in cui molti stati hanno una sotto-macchina, i diagrammi di stato possono diventare molto complessi piuttosto in fretta. Per semplificare i diagrammi di stato, è possibile utilizzare uno stato della sotto-macchina, il quale consente di far riferimento a una sotto-macchina che è stata completamente definita in un altro diagramma di stato.

Il frammento di diagramma di stato riportato nella Figura 20.12 modella il processo di autenticazione di un sito web, il quale potrebbe, o meno, riconoscere l'utente che lo visita.

Si tratta di un diagramma moderatamente complesso. È, tuttavia, possibile semplificarlo riportando i dettagli dello stato composito Autenticazione su un altro diagramma di stato. Nel diagramma principale si può, quindi, includere un riferimento a questo diagramma secondario utilizzando uno speciale tipo di stato chiamato stato della sotto-macchina. Per effettuare questo tipo di riferimento, si utilizza la parola chiave `include`, come illustrato nella Figura 20.13.

Lo stato della sotto-macchina è una notazione di comodo, molto utile per semplificare i diagrammi di stato. La semantica è piuttosto facile da comprendere: la presenza, in un diagramma, di uno stato della sotto-macchina è *del tutto equivalente* a includere l'intera macchina a stati indicata in quel punto del diagramma.

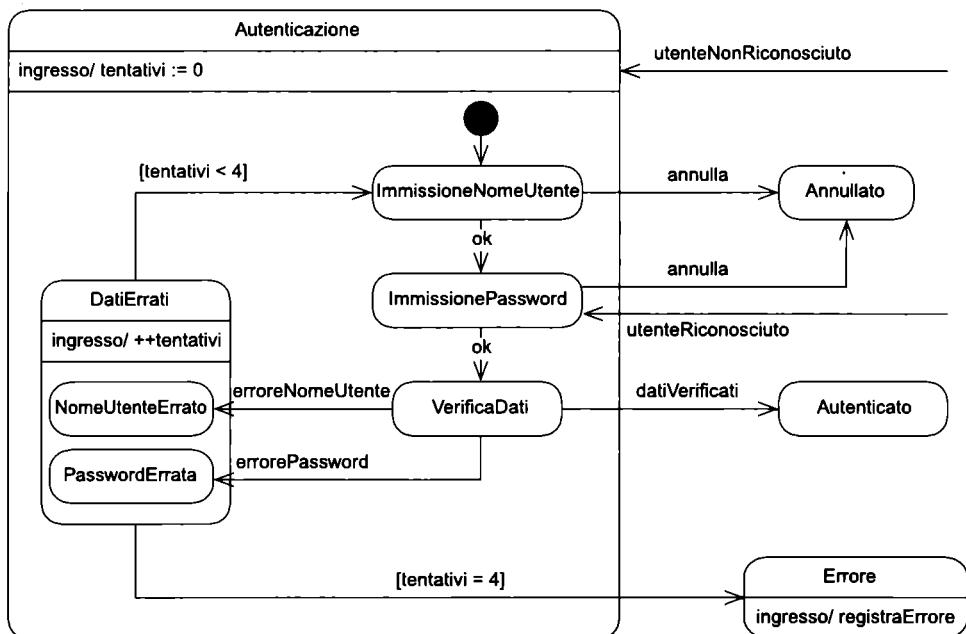


Figura 20.12

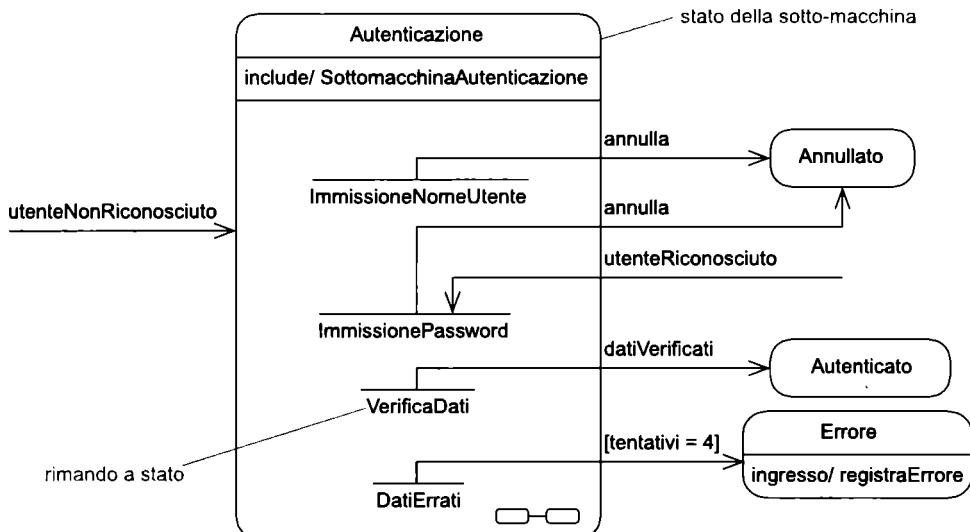


Figura 20.13

Un'altra notazione di comodo che può essere utilizzata in combinazione con lo stato della sotto-macchina, è il rimando a stato. Si tratta di un rimando, disegnato come una barra verticale o orizzontale, che indica un sottostato della sotto-macchina referenziata. La Figura 20.13 utilizza alcuni rimandi a stato.

20.8 Riepilogo

L'UML fornisce una sintassi per i diagrammi di stato che è ricca e che consente di fissare comportamenti complessi in diagrammi concisi. Sono stati spiegati i seguenti concetti.

1. Gli stati composti possono contenere una o più macchine a stati annidate: i sottostati ereditano tutte le transizioni del loro superstato.
2. Lo stato composito sequenziale contiene una sola macchina a stati annidata.
3. Lo stato composito concorrente contiene due o più sotto-macchine che vengono eseguite in parallelo.
 - Entrando nello stato avviene una biforcazione e tutte le sotto-macchine cominciano la loro esecuzione in parallelo.
 - Se tutte le sotto-macchine hanno uno stato finale, allora non si può uscire dal superstato fin quando tutte le sotto-macchine non hanno terminato la loro esecuzione: questa è una ricongiunzione.
 - Si può uscire dal superstato senza una ricongiunzione, se una sotto-macchina effettua una transizione esplicita a uno stato esterno.
4. Comunicazione asincrona tra sotto-macchine:
 - valore di attributo: una sotto-macchina imposta il valore di un attributo e le altre sotto-macchine lo leggono;
 - *stato di sync*: funziona come una coda in cui una sotto-macchina lascia un messaggio che può essere letto dalle altre sotto-macchine.
5. Uno stato con memoria consente al suo superstato di ricordare il sottostato attivo quando è avvenuta l'ultima transizione in uscita.
 - Uno stato con memoria semplice consente al suo superstato di ricordare il sottostato attivo prima dell'ultima transizione, ma solo tra i sottostati che si trovano *allo stesso livello* dell'indicatore:
 - quando viene attivata una transizione in ingresso sull'indicatore di stato con memoria semplice, la transizione viene ridirezionata sul sottostato memorizzato;
 - in caso di primo ingresso (nessun sottostato memorizzato), viene attivata l'unica transizione in uscita dall'indicatore di stato con memoria semplice.
 - Uno stato con memoria multilivello consente al suo superstato di ricordare il sottostato attivo prima dell'ultima transizione, *indipendentemente dal suo livello* di annidamento:
 - quando viene attivata una transizione in ingresso sull'indicatore di stato con memoria multilivello, la transizione viene ridirezionata sul sottostato memorizzato;
 - in caso di primo ingresso (nessun sottostato memorizzato) viene attivata l'unica transizione in uscita dall'indicatore di stato con memoria multilivello.
6. Lo stato della sotto-macchina consente di inserire nel diagramma di stato un riepilogo di una sotto-macchina:
 - utilizzare la parola chiave `include` per inserire un riferimento a una sotto-macchina descritta in un altro diagramma;
 - riepilogare la sotto-macchina utilizzando i rimandi a stato.

Parte 5

Implementazione

Il flusso di lavoro dell'implementazione

21.1 Contenuto del capitolo

Il flusso di lavoro dell'implementazione non prevede molte attività di analisi/progettazione OO, quindi questa è la parte più ridotta del libro. È, comunque, necessario dedicare qualche attenzione anche all'implementazione perché, anche se è vero che il flusso di lavoro dell'implementazione si concentra sulla produzione di codice, comunque comporta anche qualche elemento di modellazione UML.

21.2 Il flusso di lavoro dell'implementazione

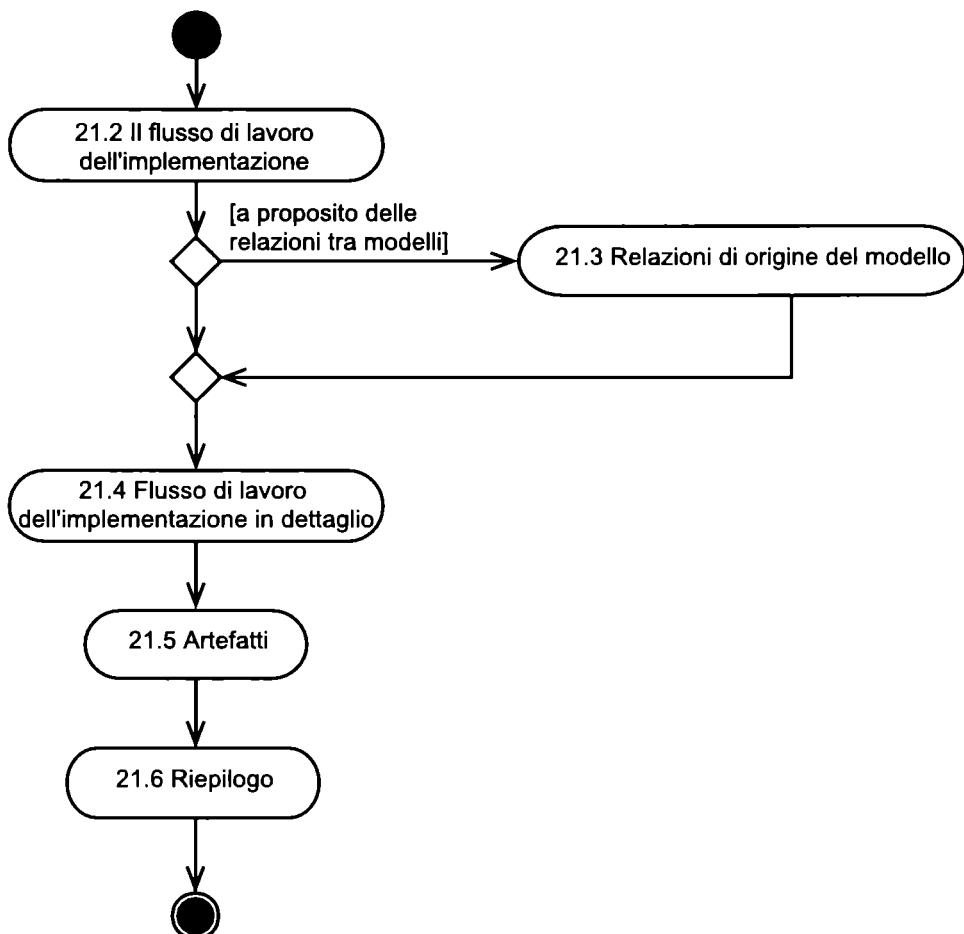
Il flusso di lavoro dell'implementazione comincia ad avere un certo peso nella fase di Elaborazione, per diventare poi il flusso di lavoro principale della fase di Costruzione (Figura 21.2).

L'implementazione riguarda la trasformazione del modello della progettazione in codice eseguibile. Dal punto di vista dell'analista/progettista, l'implementazione ha come scopo quello di produrre il modello dell'implementazione, sempreché sia richiesto.

Questo modello comporta l'allocazione (per lo più tattica) delle classi di progettazione ai componenti. Tale allocazione dipende in gran parte dal linguaggio di programmazione utilizzato per implementare il sistema.

Il flusso di lavoro dell'implementazione si concentra più che altro sulle attività finalizzate alla produzione di codice eseguibile. Il modello dell'implementazione può anche essere un sottoprodotto di queste attività, e non di attività di modellazione esplicita.

In effetti, molti strumenti CASE consentono l'estrapolazione, tramite *reverse engineering*, del modello dell'implementazione direttamente dal codice sorgente. In questo modo, la modellazione dell'implementazione viene, di fatto, delegata ai programmatorei.

**Figura 21.1**

Esistono tuttavia due casi in cui un'attività di modellazione esplicita, eseguita da analisti/progettisti OO esperti, potrebbe risultare molto importante.

- Se si ha intenzione di generare il codice direttamente dal modello (*forward engineering*), è necessario specificare esplicitamente alcuni dettagli, quali i file sorgente e i componenti da generare (salvo che si utilizzino i valori predefiniti dello strumento CASE).
- Se si utilizza il CBD per agevolare il riuso di componenti, l'allocazione delle classi e delle interfacce di progettazione ai componenti diventa una questione strategica. Potrebbe essere preferibile modellare questa allocazione in modo esplicito, senza delegarla ai singoli programmati.

Questo capitolo spiega cosa serve per mettere insieme un modello dell'implementazione.

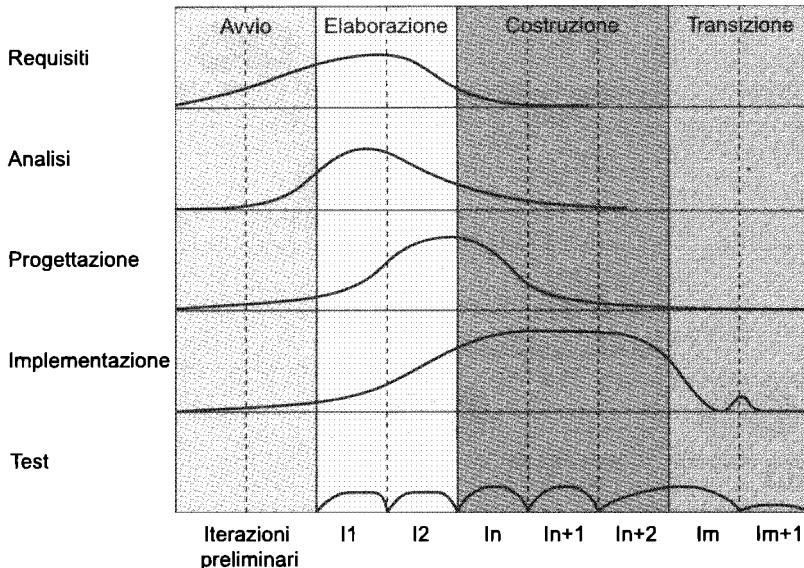


Figura 21.2 Adattata da Figura 1.5 [Jacobson 1], con il consenso della Addison-Wesley.

21.3 Relazioni di origine del modello

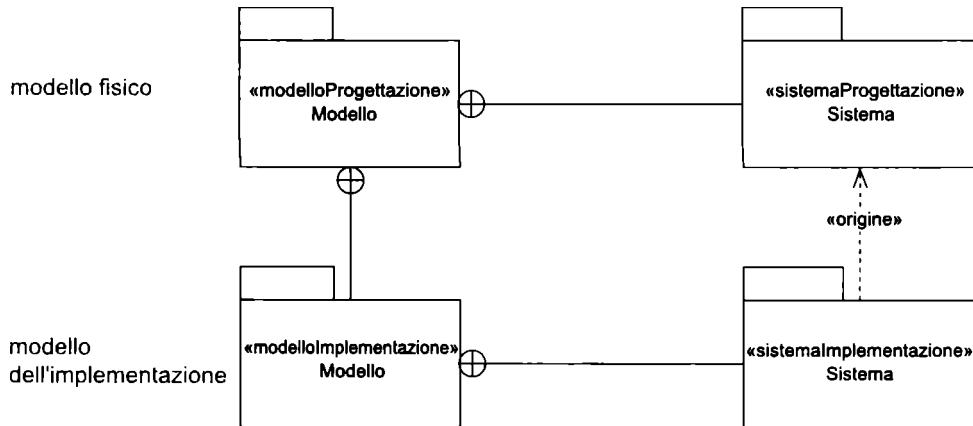
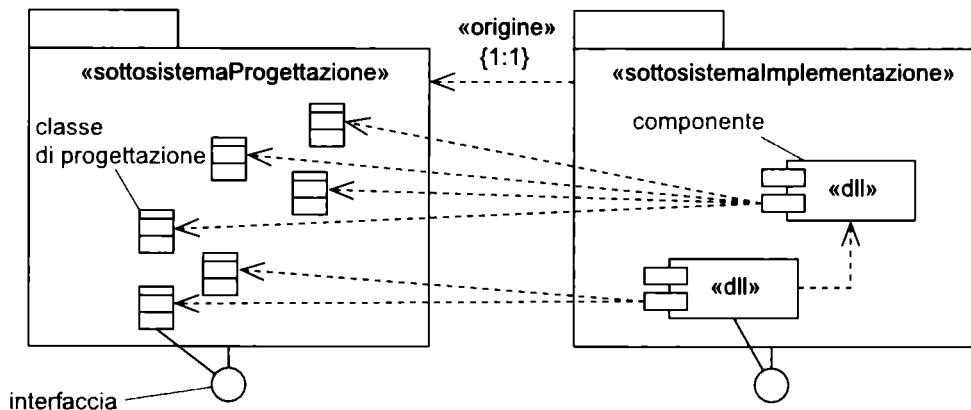
La relazione tra il modello dell'implementazione e il modello della progettazione è molto semplice. Il modello dell'implementazione è in realtà solo una vista di implementazione del modello della progettazione: ovvero *fa parte* del modello della progettazione, come illustrato nella Figura 21.3.

Ogni modello della progettazione corrisponde a un unico sistema della progettazione e, come ovvio, esiste una relazione biunivoca di «origine» tra questo sistema della progettazione e il corrispondente sistema dell'implementazione. Si ripete, si tratta in realtà solo viste diverse dello stesso modello.

Similmente, esistono relazioni biunivoco di «origine» tra ciascun sottosistema di progettazione e il corrispondente sottosistema di implementazione. Tuttavia, mentre i sottosistemi di progettazione contengono classi di progettazione, i sottosistemi di implementazione contengono i componenti in cui sono allocate tali classi. La Figura 21.4 illustra la relazione tra i sottosistemi di progettazione e i sottosistemi di implementazione.

Data la presenza di questa relazione biunivoca di "relazione", ogni sottosistema di implementazione realizza esattamente lo stesso insieme di interfacce del corrispondente sottosistema di progettazione. Nei sottosistemi di implementazione, tuttavia, queste interfacce sono realizzate da componenti.

I sottosistemi di progettazione sono entità logiche che raggruppano elementi di progettazione, mentre i sottosistemi di implementazione devono corrispondere ai meccanismi di raggruppamento fisici e reali del linguaggio di programmazione scelto per l'implementazione.

**Figura 21.3****Figura 21.4**

21.4 Flusso di lavoro dell'implementazione in dettaglio

Come si può vedere nella Figura 21.5, il flusso di lavoro dell'implementazione riguarda i ruoli di architetto, integratore di sistemi e ingegnere dei componenti. Questi ruoli del flusso di lavoro dell'implementazione possono essere coperti da analisti e progettisti, da soli o in piccoli gruppi di lavoro. Queste figure si possono concentrare sulla produzione del modello del *deployment* e dell'implementazione (attività comprese in “implementare l’architettura”). Le attività “integrare il sistema”, “implementare una classe”, “implementare un sottosistema” ed “eseguire unit test” vanno oltre lo scopo di questo libro: sono attività più di programmazione che non di analisi e progettazione.

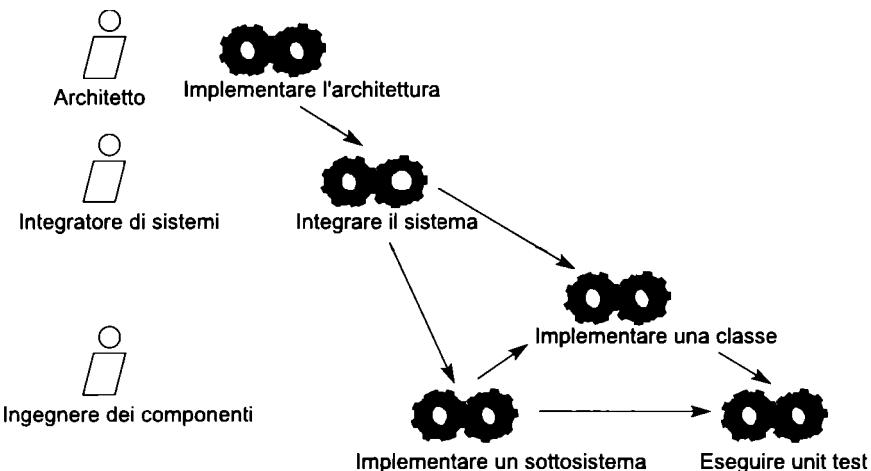


Figura 21.5 Riprodotta da Figura 10.16 [Jacobson 1], con il consenso di Addison-Wesley.

21.5 Artefatti

L'artefatto principale del flusso di lavoro dell'implementazione, dal punto di vista dell'analista/progettista OO, è il modello dell'implementazione. Questo modello è costituito da due nuovi tipi di diagramma:

- il diagramma dei componenti: modella le dipendenze tra i componenti software che costituiscono il sistema;
- il *diagramma di deployment*: modella i nodi computazionali fisici su cui verrà dislocato il software, e le relazioni tra questi nodi.

21.6 Riepilogo

L'implementazione si occupa principalmente della produzione di codice. Tuttavia l'analista/progettista OO può essere ancora utile per creare il modello dell'implementazione. Sono stati spiegati i seguenti concetti.

1. Nella fase di Costruzione, il flusso di lavoro principale è quello dell'implementazione.
2. L'implementazione si occupa della trasformazione del modello della progettazione in codice eseguibile.
3. La modellazione dell'implementazione è importante se:
 - si intende generare il codice dal modello (*forward engineering*);
 - si intende utilizzare CBD per poter riusare componenti.

4. Il modello dell'implementazione fa parte del modello della progettazione:
 - i sottosistemi di progettazione hanno una relazione di origine con i sottosistemi di implementazione;
 - le classi di progettazione vengono allocate ai componenti;
 - le interfacce sono realizzate dai componenti.
5. L'implementazione dell'architettura produce:
 - diagrammi dei componenti;
 - *diagrammi di deployment*.

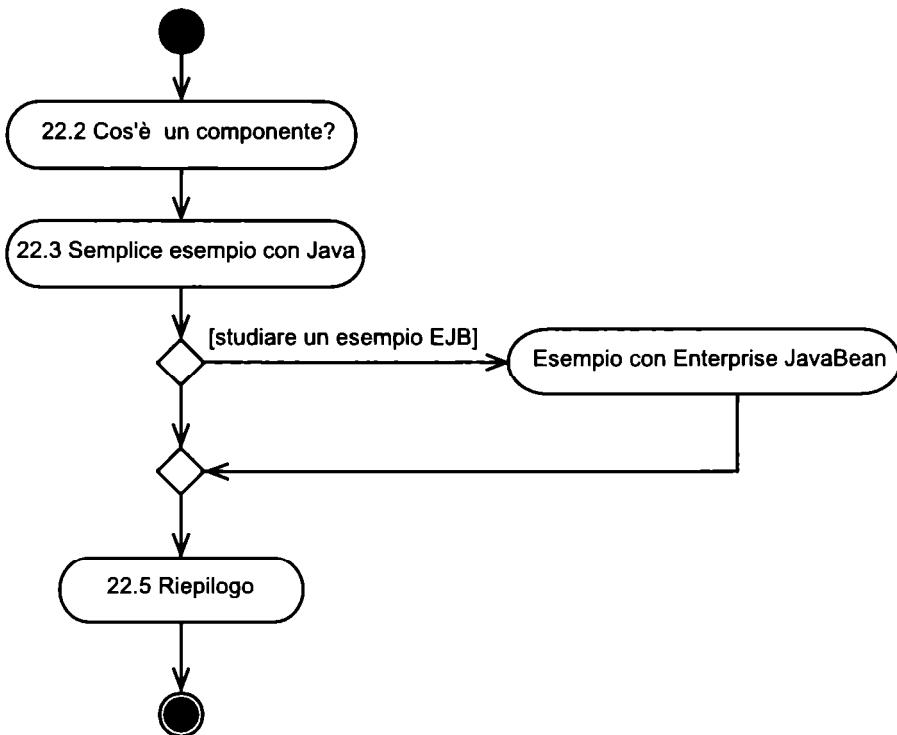
22.1 Contenuto del capitolo

In questo capitolo si discute di componenti. Si consiglia di leggere almeno le Sezioni 22.2 e 22.3. L'esempio del Paragrafo 22.4 è opzionale, ma è comunque interessante dato che illustra l'applicazione di una proposta di profilo UML standard per gli EJB. I profili servono ad adattare e personalizzare l'UML con l'aggiunta di stereotipi, vincoli e valori etichettati e descrivono come l'UML dovrebbe essere applicato alla modellazione in situazioni particolari.

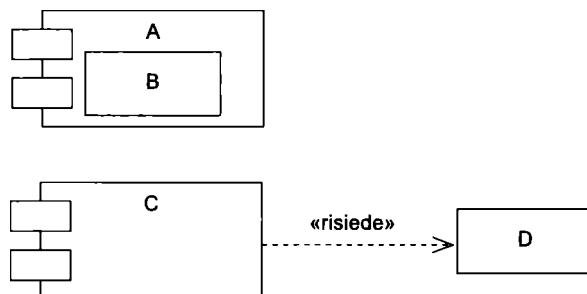
22.2 Cos'è un componente?

The UML Reference Manual [Rumbaugh 1] dice: “Un componente è una parte, fisica e sostituibile di un sistema, che contiene implementazione e che rispetta un insieme di interfacce e ne fornisce una realizzazione.” I componenti sono entità fisiche e tangibili quali gli Enterprise JavaBean (EJB) o i controlli ActiveX. I componenti sono unità di riuso, e sono definiti in modo piuttosto ampio. Uno qualunque degli artefatti elencati qui di seguito può essere considerato un componente. Ciascun componente può contenere molte classi e realizzare molte interfacce.

- un file di codice sorgente;
- un sottosistema di implementazione;
- un controllo ActiveX;
- un JavaBean;
- un Enterprise JavaBean;
- una servlet Java;
- una Java Server Page.

**Figura 22.1**

Il modello dei componenti documenta come le classi e le interfacce vengono combinate in componenti. L'assegnazione di una classe a un componente può essere indicata collocando l'icona della classe all'interno dell'icona del componente, oppure utilizzando una dipendenza «risiede» tra il componente e la classe. Questi due stili sono semanticamente equivalenti. La Figura 22.2 riporta esempi di queste due notazioni: la classe B è assegnata al componente A utilizzando la notazione di contenimento, mentre la classe D è assegnata al componente C utilizzando la dipendenza «risiede».

**Figura 22.2**

Il diagramma dei componenti può esistere esclusivamente in forma descrittore. Dato che i componenti sono parti fisiche del sistema, ha senso mostrare istanze di componenti solo in relazione all'hardware fisico: non possono esistere al di fuori di quell'hardware. Per questo motivo, le istanze di componenti possono comparire sul *diagramma di deployment*, dato che quest'ultimo modella l'hardware del sistema, ma non sul diagramma dei componenti. I *diagrammi di deployment* sono trattati nel Capitolo 23.

I componenti possono realizzare interfacce: in realtà si tratta di una forma abbreviata per dire che qualche classe *contenuta* nel componente può realizzare qualche interfaccia. I componenti possono dipendere da altri componenti. Quando si desidera ridurre l'interdipendenza tra componenti, è *necessario* introdurre interfacce a sostituzione delle dipendenze: la Figura 22.3 ne illustra un esempio.

In questo esempio, è stata accettata la dipendenza tra i componenti E e H, dato che questi risiedono nello stesso strato del modello e sono intimamente correlati. D'altra parte, i componenti E e G si trovano in strati diversi, ed è quindi stata introdotta l'interfaccia F per eliminare l'interdipendenza esistente tra loro.

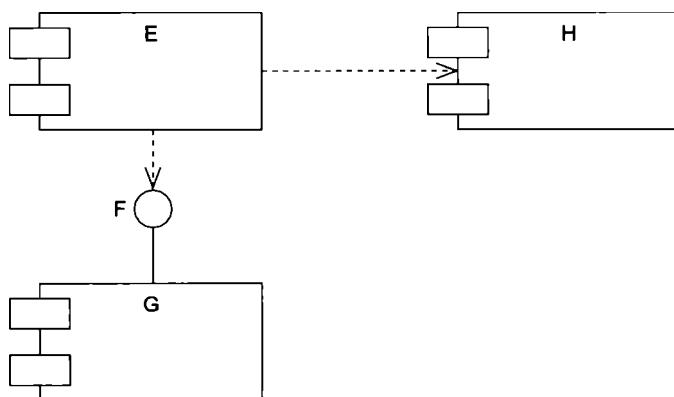


Figura 22.3

22.3 Semplice esempio con Java

In questa sezione si vuole illustrare la modellazione UML dei componenti utilizzando un esempio molto semplice in Java. La Figura 22.4 mostra il modello dei componenti di una semplice applicazione di *database* sviluppata in Java. Questa applicazione utilizza il *pattern Data Access Object* (Oggetto di Accesso ai Dati, DAO) descritto in *Core J2EE™ Patterns* [Alur 1]. L'idea sottostante questo *pattern* è quella di creare un DAO che contiene il codice per accedere a uno specifico database. Tale DAO viene, quindi, occultato dietro un'interfaccia che può essere utilizzata dai client. L'informazione viene passata dal DAO, tramite l'interfaccia, ai client sotto forma di un Value Object (Oggetto Valore, VO) che contiene una rappresentazione delle informazioni richieste non dipendente dal *database* sottostante.

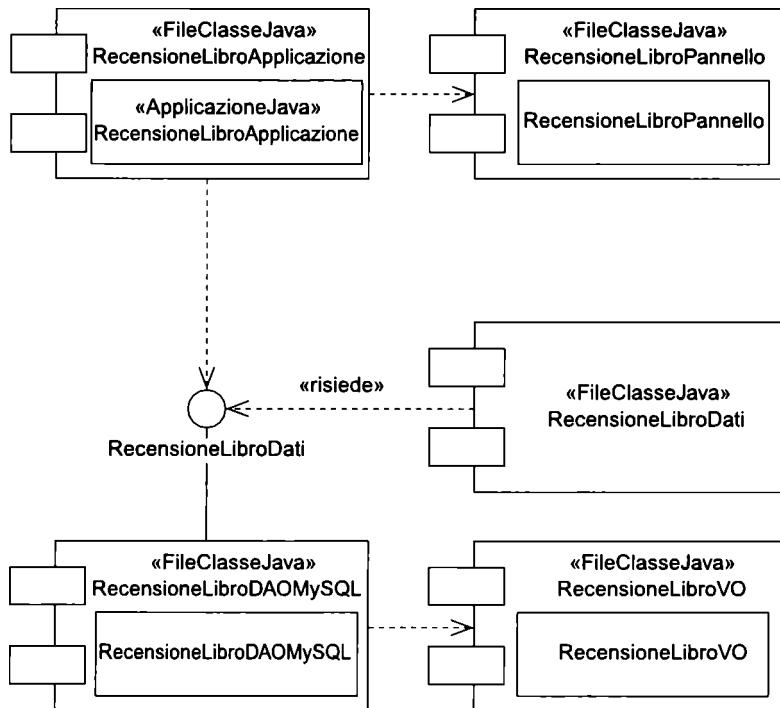


Figura 22.4

Uno dei vantaggi di questo *pattern* è che se si decide di spostare i dati su un *database* diverso, basta creare un nuovo DAO per gestire l'accesso a quel *database*. Il codice presente nel client, che utilizza esclusivamente l'interfaccia, e che quindi non conosce il DAO, non dovrebbe accorgersi della modifica.

I componenti sono file di classe Java contenenti bytecode compilato con Java. Per evidenziare questa scelta, è stato deciso di contrassegnare questi componenti con lo stereotipo `<<FileClasseJava>>`. Gli stereotipi sono utilizzati molto spesso nella modellazione dei componenti, dato che esistono moltissimi tipi di componenti.

In Java, i file di classe *tipicamente* contengono il codice di una singola classe Java; si semplifica, quindi, parecchio la mappatura tra classi e componenti. Nella Figura 22.4, il componente **RecensioneLibroApplicazione** contiene un'unica classe, chiamata anch'essa **RecensioneLibroApplicazione** (questa classe è mostrata con lo stereotipo `<<AplicazioneJava>>` per evidenziare che si tratta di un programma Java eseguibile). Questo componente usa il componente **RecensioneLibroPannello** (il quale contiene una classe con lo stesso nome), e fornisce la GUI all'applicazione. È stato deciso di usare una dipendenza diretta tra i due componenti, e non un'interfaccia; questo perché i due componenti sono intimamente correlati e appartengono allo stesso strato logico del sistema.

Il componente **RecensioneLibroApplicazione** usa l'interfaccia **RecensioneLibroDati**, che risiede nel componente **RecensioneLibroDati**.

L'interfaccia `RecensioneLibroDati` è implementata dal componente `RecensioneLibroDAOMySQL`. Questo componente è un DAO che fornisce accesso ai dati di recensioni libri archiviati in un *database* relazionale MySQL. Sarebbe possibile preparare diversi DAO per diversi *database* relazionali e, ammesso che implementino tutti l'interfaccia `RecensioneLibroDati`, dal punto di vista dell'applicazione client, sarebbero tutti intercambiabili.

Studiando la Figura 22.4, si capisce che questo modello è composto da tre strati distinti. Si possono descrivere questi strati, utilizzando un sottoinsieme dei cinque livelli che compongono il modello J2EE, così come trattato in [Alur 1]. Nel modello presentato ci sono il livello client, il livello di *business* e il livello di integrazione. È, inoltre, implicito anche il livello delle risorse, che in questo caso contiene il *database* relazionale. Possiamo raggruppare i nostri componenti e distribuirli in file Java ARchive (JAR), secondo questo schema, così come illustrato nella Figura 22.5.

Questo esempio dimostra come sia possibile avere dei componenti di alto livello (i file JAR) composti da componenti più piccoli (i file delle classi). I file JAR sono componenti adatti alle attività di distribuzione e di integrazione, mentre i file delle classi sono componenti adatti alle attività di sviluppo e di test. Quella di predisporre componenti di diverso livello è una pratica abbastanza comune.

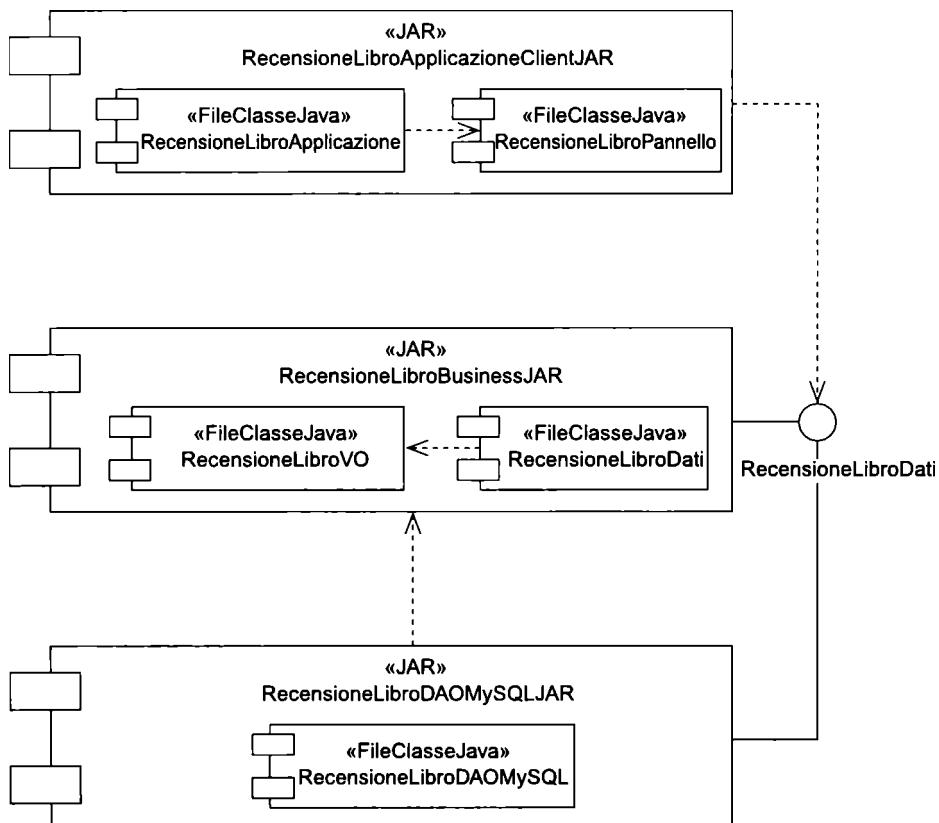


Figura 22.5

22.4 Esempio con Enterprise JavaBean

L'architettura Enterprise JavaBean (EJB) fornisce un modello di componenti distribuiti per Java. Un componente EJB viene dislocato in un contenitore EJB che risiede sul server applicativo. Questo contenitore fornisce al componente EJB molti servizi avanzati; servizi che altrimenti dovrebbero essere implementati manualmente. Questi servizi includono:

- distribuzione;
- transazioni;
- persistenza;
- sicurezza;
- condivisione di risorse.

Ogni componente EJB è costituito da almeno quattro parti: tre file di classe Java (con estensione .class) e un file XML (con estensione .xml). La Tabella 22.1 sintetizza la semantica di queste parti.

Tabella 22.1

Parte EJB	Semantica
Interfaccia home	Definisce l'interfaccia per cercare o creare istanze dell'EJB
Interfaccia remota	Definisce l'interfaccia dei metodi di <i>business</i> dell'EJB
Implementazione EJB	Implementa l'interfaccia home e l'interfaccia remota
Descrittore di <i>deployment</i>	Un file XML che descrive al contenitore EJB quali siano i servizi richiesti dal <i>bean</i> (per esempio, la persistenza)

Una caratteristica interessante dell'architettura EJB è che si *potrebbe* pensare che l'implementazione EJB implementi l'interfaccia remota e l'interfaccia *home*, mentre non è così! In realtà, queste due interfacce sono implementate rispettivamente da un oggetto EJB e da un oggetto *home* che vengono generati automaticamente dal contenitore EJB quando viene attivato. Ciononostante, l'implementazione EJB *deve* comunque includere i metodi definiti nell'interfaccia remota e nell'interfaccia *home*, anche se fisicamente non ha con esse una relazione di implementazione.

Tipicamente tutte le parti EJB vengono raggruppate in un unico file JAR, che è un file compresso che contiene le classi Java compilate e le altre parti. Quando questo file JAR viene dislocato in un contenitore EJB, tale contenitore genera un oggetto EJB e un oggetto *home*; può anche generare un JAR per i client, contenente gli stub che servono ai programmi client che volessero usare l'EJB.

Il JAR client fornisce l'implementazione lato client delle interfacce *home* e *remota* dell'EJB. Questa implementazione non fa altro che inoltrare messaggi al contenitore EJB, tramite il protocollo RMI (Remote Method Invocation) di Java.

Si tratta di una situazione piuttosto complessa. Quali sono i componenti coinvolti in questa architettura? Ragionando ad alto livello, si può dire di avere un componente EJB. Ma, a un più attento esame, possiamo vedere che questo componente ha un aspetto lato client (il file JAR client) e un aspetto lato server (il file JAR server). Guardando ancora più da vicino si vede che il JAR client realizza due interfacce, ma per il resto è piuttosto opaco, come lo è anche l'EJB che ne genera l'implementazione. Si osservà anche che il JAR server contiene almeno quattro file: l'interfaccia remota, l'interfaccia *home*, l'implementazione EJB e un *descrittore di deployment* in formato XML.

Ed ora si passi a un esempio concreto: un sistema Java molto semplice che utilizza un EJB per accedere a informazioni relative a libri. Il libro è stato modellato come un EJB entità. Questo particolare tipo di EJB è persistente, e non dobbiamo dunque preoccuparci dell'accesso al *database*: se ne occuperà il contenitore EJB. È stata, inoltre, modellata un'applicazione chiamata *LibroApplicazioneTest* che vuole essere una piattaforma di test per l'EJB.

Esistono diverse tecniche per modellare un EJB, ma la Rational Corporation ha proposto al Java Community Process una bozza di profilo UML per gli EJB (*JSR-000026, UML Profile for EJB [JSR 1]*), ed è questa la tecnica utilizzata. È, tuttavia, necessario tenere presente che questa bozza di profilo potrebbe ancora subire modifiche. La Figura 22.6 riporta il diagramma dei componenti del sistema presentato.

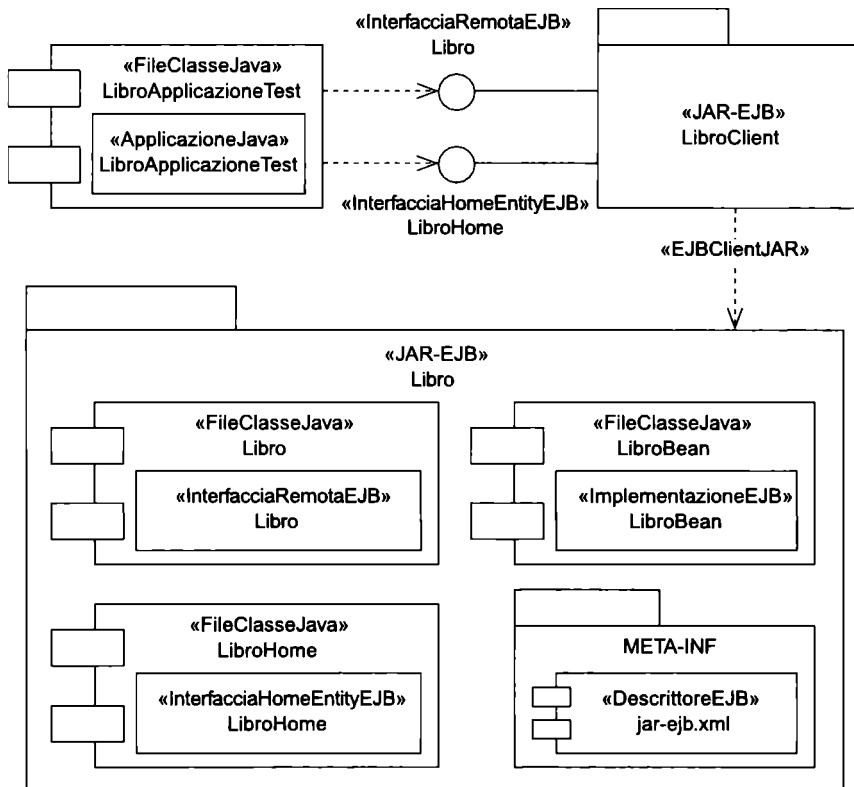


Figura 22.6

Il Profilo UML della Rational introduce diversi stereotipi standard per Java e per gli EJB, elencati nella Tabella 22.2.

Tabella 22.2

Stereotipo	Elemento stereotipato	Semantica
«FileClasseJava»	Componente	Un file che contiene il bytecode Java (binari): questi file hanno estensione .class
«DescrittoreEJB»	Componente	Un file XML che è il descrittore di sviluppo dell'EJB
«InterfacciaHomeEntitaEJB»	Classe	Un'interfaccia Java che è l'interfaccia <i>home</i> dell'entità EJB
«InterfacciaRemotaEJB»	Classe	Un'interfaccia Java che è l'interfaccia remota dell'EJB
«ImplementazioneEJB»	Classe	Una classe Java che fornisce l'implementazione dell'EJB
«JAR-EJB»	Package	Un JAR che contiene uno o più EJB: questi file hanno tipicamente estensione .jar
«JARClientEJB»	Dipendenza	Indica che il client è il JAR client dell'EJB

Come si può vedere dalla Tabella 22.2, il Profilo UML della Rational modella i JAR come dei *package* con stereotipo. È preferibile modellare i JAR come componenti con stereotipo, come mostrato nella Figura 22.5. Tuttavia, presupponendo che la semantica dei due stereotipi sia chiara (e lo è), sono accettabili entrambe le tecniche.

Il descrittore di *deployment* risiede in un *package* chiamato META-INF. Dato che i file JAR sono in realtà dei file zip (e possono essere creati e visualizzati con gli stessi strumenti), le informazioni relative al contenuto di ciascun componente vengono salvate quando si crea il file JAR. La specifica EJB richiede espressamente che il *descrittore di deployment* risieda sempre in una cartella di nome META-INF, e questo è molto facile da documentare sul nostro diagramma dei componenti.

22.5 Riepilogo

I componenti sono le unità di base per lo sviluppo di un sistema software. Il diagramma dei componenti illustra il modo in cui le classi e le interfacce sono state assegnate ai componenti.

Sono stati spiegati i seguenti concetti.

- I. Un componente è una parte fisica e sostituibile di un sistema.
 - I componenti raggruppano classi di progettazione. Questo può essere modellato nei seguenti modi:

- includendo l'icone della classe all'interno dell'icona del componente;
 - utilizzando una dipendenza «risiede» tra il componente e la classe.
- I componenti realizzano interfacce: che è solo un modo veloce per dire che le classi che risiedono in un componente realizzano delle interfacce.
2. Esempi di componenti:
- file di codice sorgente;
 - sottosistema di implementazione;
 - controllo ActiveX;
 - JavaBean;
 - Enterprise JavaBean;
 - servlet Java;
 - Java Server Page.
3. I componenti possono avere dipendenze con altri componenti: per ridurre le interdipendenze si può sostituire una dipendenza tra due componenti con un'interfaccia.

23.1 Contenuto del capitolo

Questa sezione spiega come produrre un *diagramma di deployment*. Si tratta di un diagramma che illustra come il software che si sta sviluppando verrà dislocato sull'hardware fisico, e come quell'hardware sarà interconnesso. Il Paragrafo 23.4 riporta un semplice esempio di *deployment EJB*.

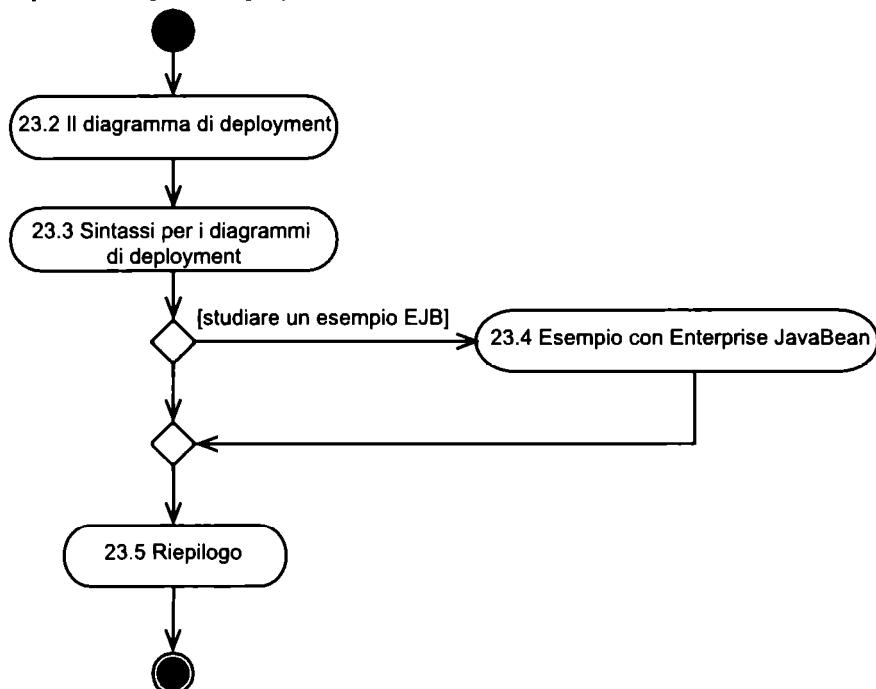


Figura 23.1

23.2 Il *diagramma di deployment*

Il *diagramma di deployment* mostra l'hardware su cui il sistema software verrà eseguito, e come tale software verrà dislocato sull'hardware.

Esistono due forme di *diagramma di deployment*:

- La forma descrittore: contiene nodi, relazioni tra nodi e componenti. Un nodo rappresenta un tipo di hardware (come, per esempio, un PC). Similmente, un componente rappresenta un tipo di software (come, per esempio, Microsoft Word).
- La forma istanza: contiene istanze di nodi, relazioni tra istanze di nodi e istanze di componenti. Un'istanza di nodo rappresenta un hardware specifico e identificabile (come, per esempio, il PC di Luca). Un'istanza di componente rappresenta una specifica istanza di un tipo di software (come, per esempio, la copia di Microsoft Word utilizzata per scrivere proprio questo libro). Se non si conoscono (o non interessa conoscere) i dettagli di specifiche istanze, si possono utilizzare istanze anonime.

Anche se ne stiamo discutendo come se fosse un'attività relativa all'implementazione, una prima bozza di *diagramma di deployment* viene spesso prodotta già durante la progettazione, quando si deve decidere l'architettura hardware del sistema. Si può iniziare con un *diagramma di deployment* in forma descrittore, limitato ai soli nodi e alle loro interconnessioni. Si può in seguito rifinire questo diagramma in uno o più *diagrammi di deployment* in forma istanza, che illustrino possibili disposizioni di istanze anonime di nodi. Infine, quando si conoscono i dettagli relativi all'hardware presente nel sito dove verrà dislocato il sistema, si potrà creare un *diagramma di deployment* in forma istanza che faccia riferimento alle macchine effettivamente utilizzate, sempreché sia richiesto.

La costruzione di un *diagramma di deployment* è, dunque, un processo a due tempi:

1. Nel flusso di lavoro della progettazione: si focalizza principalmente sui nodi, o sulle istanze di nodi, e sulle loro interconnessioni.
2. Nel flusso di lavoro dell'implementazione: si focalizza sull'assegnazione fisica delle istanze di componenti alle istanze di nodi (forma istanza) o dei componenti ai nodi (forma descrittore).

23.3 Sintassi per i *diagrammi di deployment*

La sintassi per i *diagrammi di deployment* in forma descrittore è piuttosto semplice, così come illustrato nella Figura 23.2.

I nodi vengono modellati come cubi etichettati con il *tipo* di hardware fisico che rappresentano. In questo esempio ci sono dei PC e delle Sparc Station. La relazione tra i nodi documenta una connessione tra i PC e le Sparc Station, e che tale connessione utilizza il protocollo HTTP.

È possibile dislocare componenti in ciascun nodo. In questo semplice esempio, i PC possono eseguire Internet Explorer 5, mentre le Sparc Station possono eseguire un web server Apache. La dipendenza tra i due componenti *dove* basarsi sul protocollo HTTP, dato che si tratta dell'unico modo in cui i due nodi sono interconnessi.

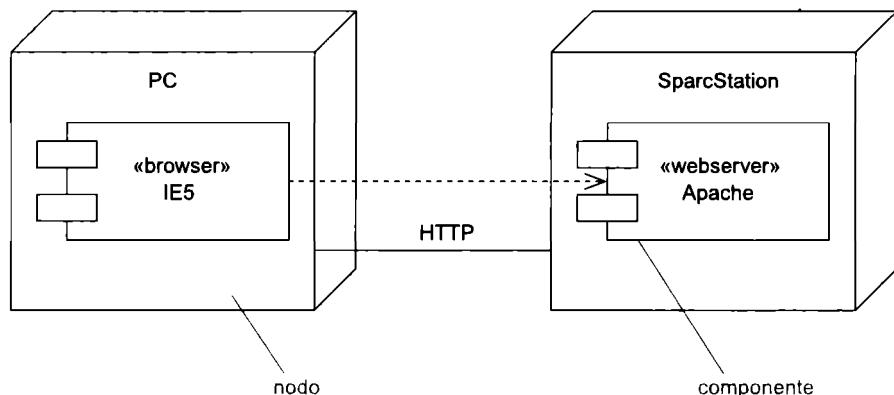


Figura 23.2

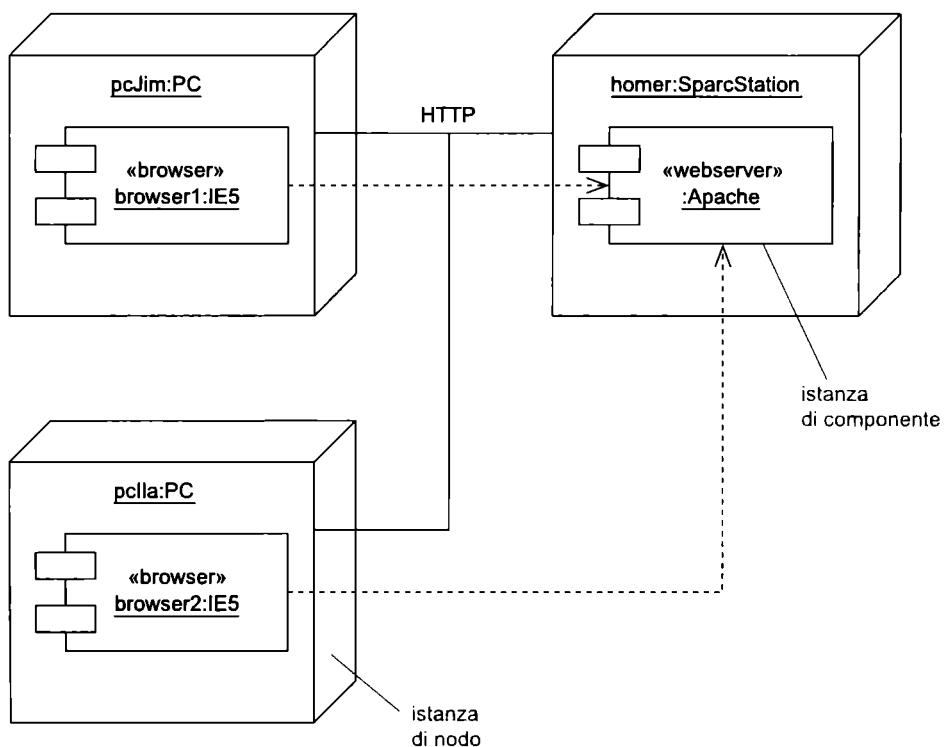


Figura 23.3

Il *diagramma di deployment* in forma istanza è molto simile. La Figura 23.3 mostra un diagramma in forma istanza dello stesso esempio della Figura 23.2. È necessario ricordarsi che i *diagrammi di deployment* in forma istanza modellano istanze *specifiche* di pezzi di hardware (istanze di nodi), su cui sono dislocate istanze *specifiche* di pezzi di software (istanze di componenti).

In questo *diagramma di deployment* in forma istanza ci sono due PC, quello di Jim e quello di Ila. Entrambi i PC stanno eseguendo una loro copia di IE5 e stanno comunicando con il server, homer, tramite HTTP. Il server sta eseguendo un'istanza anonima del web server Apache.

Il *diagramma di deployment* in forma istanza è quello più utilizzato, anche perché spesso serve modellare l'hardware specifico presente in un particolare sito client.

Secondo *The UML User Guide* [Booch 2], i *diagrammi di deployment* sono la parte dell'UML maggiormente soggetta all'uso di stereotipi. Dato che è possibile assegnare icone speciali agli stereotipi, questa prassi consente l'uso nel *diagramma di deployment* di simboli molto somiglianti all'hardware che si vuole modellare. In questo modo, i *diagrammi di deployment* possono essere resi facilmente leggibili. Per questo motivo, può essere utile avere una buona raccolta di clip-art! La Figura 23.4 riporta un *diagramma di deployment* in forma descrittore completamente stereotipato.

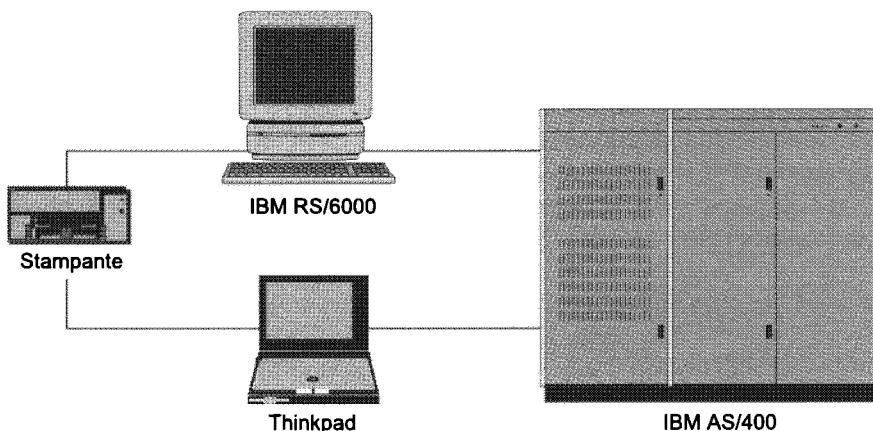


Figura 23.4

23.4 Esempio con Enterprise JavaBean

Si prenda ora l'esempio EJB del Paragrafo 22.4 e lo si dislochi su un hardware. Prima di proseguire, si consiglia la lettura del Paragrafo 22.4, seppure non sia già stata letta. La Figura 23.5 riporta il diagramma dei componenti creato per l'entità EJB Libro. Per poter dislocare questo sistema, è prima necessario prendere alcune decisioni.

- Che tipo di hardware utilizzerà il client?
- Che tipo di hardware utilizzerà il server?
- Che tipo di software verrà utilizzato come contenitore EJB?

Si è deciso di utilizzare dei PC Windows 2000, sia come client che come server, mentre verrà utilizzato JBoss come contenitore EJB: JBoss è un contenitore EJB *open source* disponibile su www.jboss.org.

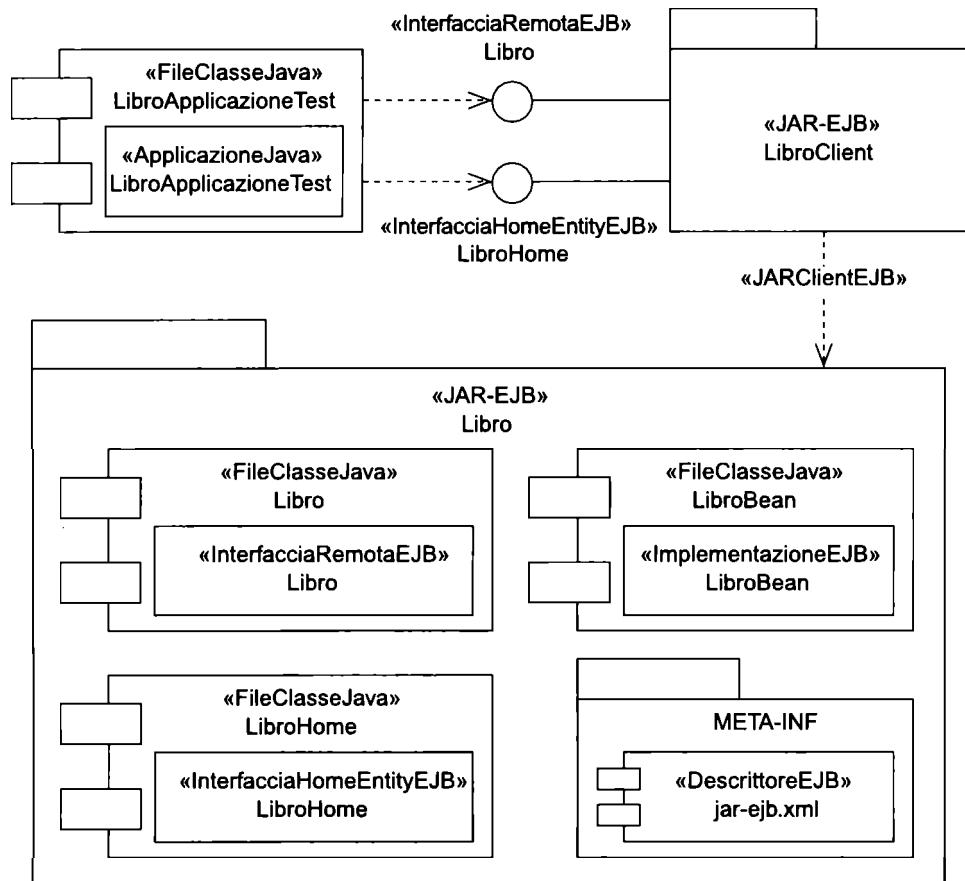


Figura 23.5

La Figura 23.6 riporta un *diagramma di deployment* in forma istanza del sistema. In questo modello ci sono due istanze di nodi, pcJim e homer.

Questi nodi sono connessi tramite il protocollo RMI (Remote Method Invocation) di Java, che lavora su una rete TCP/IP. La macchina pcJim esegue l'applicazione client, che usa il JAR client (:LibroClient) per accedere all'EJB. Il JAR server (:Libro) contiene l'EJB, ed è dislocato in un contenitore EJB JBoss sulla macchina homer.

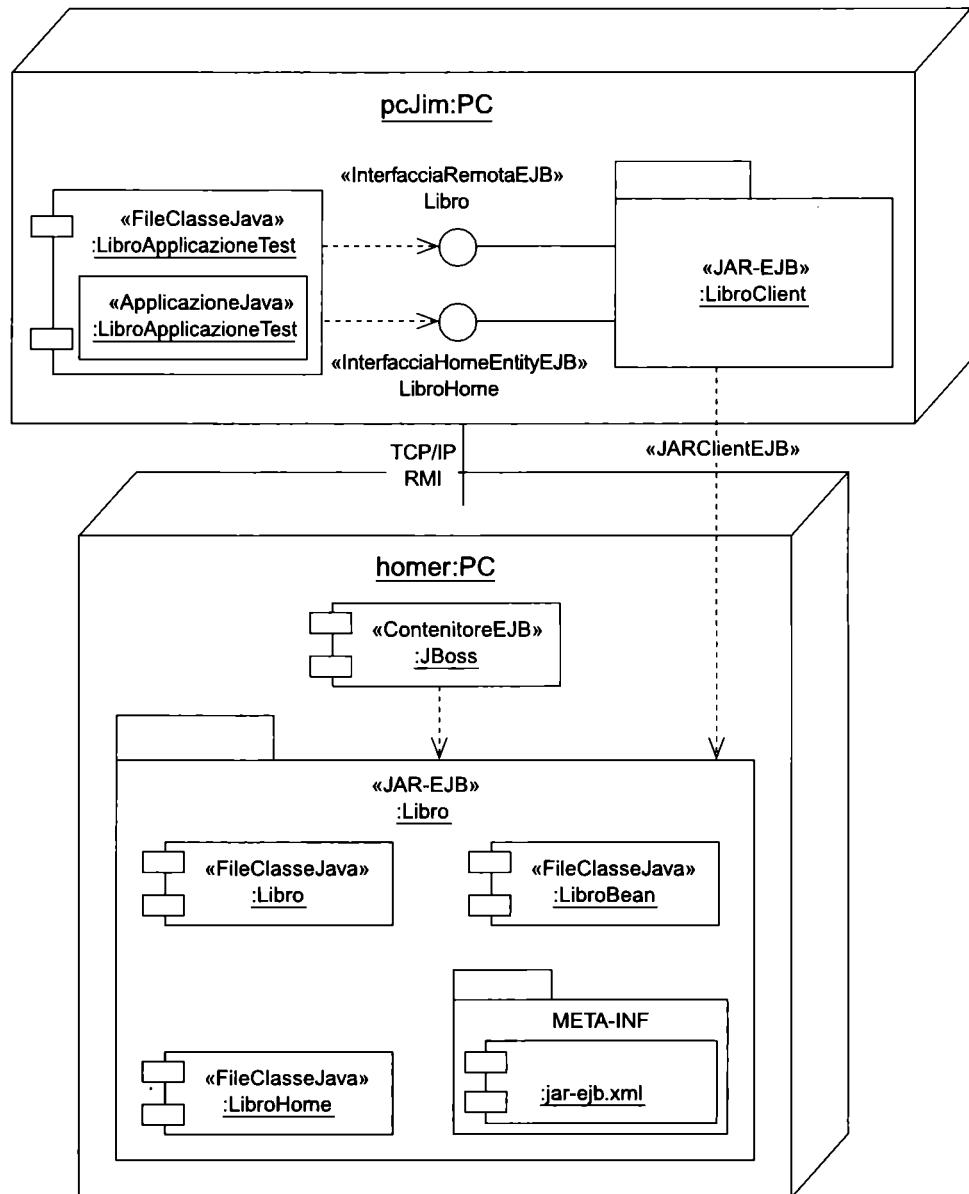


Figura 23.6

23.5 Riepilogo

I *diagrammi di deployment* consentono di modellare come un sistema software venga dislocato sull'hardware fisico. Si sono spiegati i seguenti concetti.

1. Il *diagramma di deployment* mostra la corrispondenza tra l'architettura software e l'architettura hardware.
2. Il *diagramma di deployment* in forma descrittore può essere utilizzato per modellare quali tipi di hardware, di software e di connessioni verranno utilizzati nella configurazione finale del sistema.
 - Descrive un insieme completo di possibili configurazioni.
 - Mostra:
 - nodi: i tipi di hardware utilizzati per eseguire il sistema;
 - relazioni: i tipi di connessioni esistenti tra i nodi;
 - componenti: i tipi di componenti software dislocati sui diversi tipi di nodi.
3. Il *diagramma di deployment* in forma istanza evidenzia una particolare configurazione del sistema relativa a un insieme di pezzi di hardware specifici e identificabili.
 - Descrive una particolare installazione del sistema, magari relativa a uno specifico sito utente.
 - Mostra:
 - istanze di nodi: pezzi di hardware specifici;
 - istanze di relazioni: relazioni specifiche tra istanze di nodi;
 - istanze di componenti: pezzi di software specifici e identificabili dislocati su istanze di nodi; per esempio, una particolare copia di Microsoft Office, con un numero di serie ben definito.
4. Nel flusso di lavoro della progettazione si può creare una prima bozza di *diagramma di deployment*, nel quale sono già stati individuati i nodi, o le istanze di nodi, e le loro interconnessioni; nel flusso di lavoro dell'implementazione si può rifinire questo diagramma aggiungendo i componenti, o le istanze di componenti.
5. È possibile utilizzare stereotipi, e le relative icone speciali, per rendere i *diagrammi di deployment* più somiglianti all'hardware modellato.

Appendice 1

Esempio di modello dei casi d'uso

A1.1 Introduzione

L'esperienza dice che i modelli UML non vanno troppo d'accordo con la carta. Chiunque abbia mai stampato un modello UML di grandi dimensioni, comprese le specifiche, sa di cosa si tratta. Per rendere i modelli UML più presentabili, è necessario utilizzare un supporto flessibile e ipertestuale. Oggi, questo significa utilizzare uno strumento CASE oppure un insieme di pagine web.

Si è, quindi, deciso di mettere a disposizione un esempio completo di modello UML sul nostro sito web (www.umlandtheunifiedprocess.com), invece di includerlo in questo libro. In questo esempio, seguiamo tutte le attività di analisi e di progettazione OO richieste, per creare una piccola applicazione di *e-commerce* via web. In questa appendice sono presentati solo alcuni punti di interesse del modello, semplificati, per offrire un piccolo assaggio di ciò che è disponibile sul sito web.

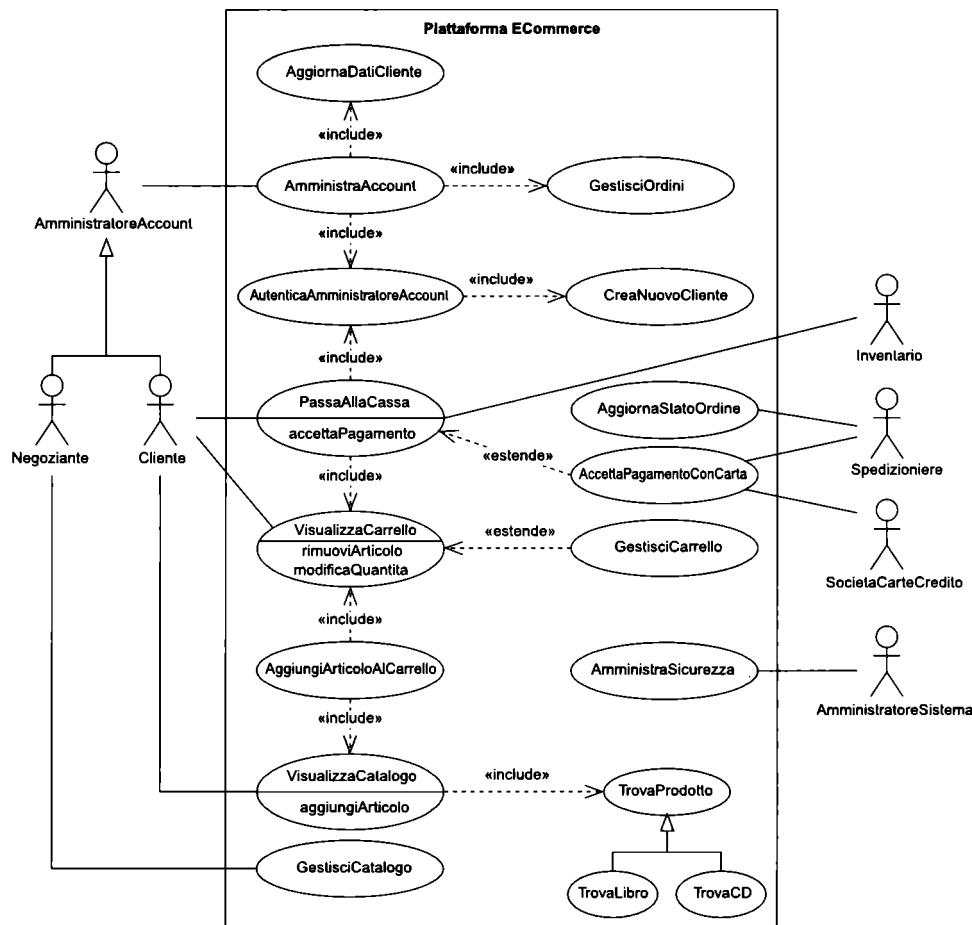
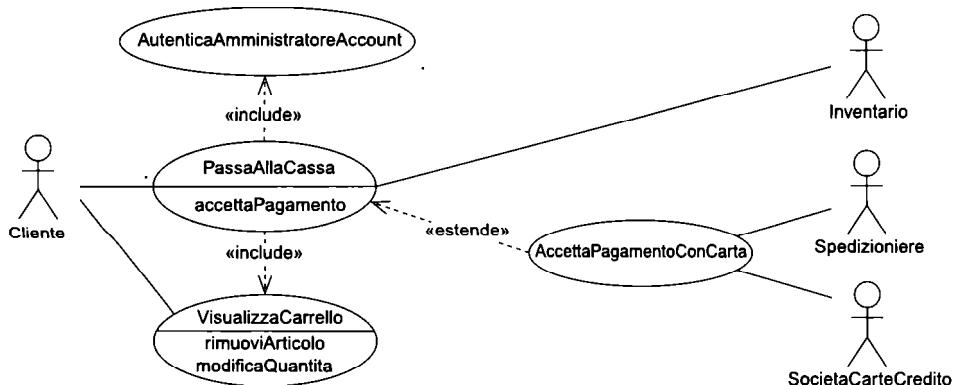
A1.2 Modello dei casi d'uso

Il modello dei casi d'uso è quello di un semplice sistema di *e-commerce* per la vendita di libri e CD. La Figura A1.1 riporta il risultato finale della modellazione dei casi d'uso.

Questo modello dei casi d'uso consente di farsi una buona idea di quello che fa il sistema, ma è consigliabile far riferimento al sito web per avere maggiori dettagli e qualche approfondimento.

A1.3 Esempi di casi d'uso

Si consideri ora un sottoinsieme del modello dei casi d'uso (vedere la Figura A1.2).

**Figura A1.1****Figura A1.2**

Caso d'uso: PassaAllaCassa	
ID: UC12	
Attori:	A2 Cliente
Include:	UC3 AutenticaAmministratoreAccount UC7 VisualizzaCarrello
Punti di estensione:	<accettaPagamento>
Precondizioni:	
Sequenza degli eventi:	<ol style="list-style-type: none"> 1. Il Cliente seleziona "Cassa". 2. include(AutenticaAmministratoreAccount) 3. include(MostraCarrello) 4. Se il cliente seleziona "procedi alla cassa" <ol style="list-style-type: none"> 4.1 Il sistema comunica con l'attore Inventario per sapere quali articoli presenti nell'ordine sono immediatamente disponibili. 4.2 Se qualche articolo dell'ordine non è immediatamente disponibile <ol style="list-style-type: none"> 4.2.1 Il sistema informa il Cliente che alcuni articoli non sono disponibili e che quindi devono essere rimossi dall'ordine. 4.3 Il sistema presenta al Cliente l'ordine finale e chiede conferma per l'addebito della carta. 4.4 Se il Cliente accetta l'ordine finale, <accettaPagamento>
Postcondizioni:	

Figura A1.3

Caso d'uso di estensione: AccettaPagamentoConCarta	
ID: UC1	
Estende:	UC12 PassaAllaCassa al punto <accettaPagamento>
Segmento inseribile:	<ol style="list-style-type: none"> 1. Il sistema recupera gli estremi della carta di credito del Cliente. 2. Il sistema invia un messaggio alla SocietaCarteCredito che contiene l'ID del venditore, l'autenticazione del venditore, gli estremi della carta di credito del Cliente e l'ammontare dell'ordine. 3. La SocietaCarteCredito convalida la transazione. 4. Se la transazione ha avuto successo <ol style="list-style-type: none"> 4.1 Il sistema notifica l'avvenuto addebito al Cliente. 4.2 Il sistema notifica al Cliente l'identificativo di riferimento dell'ordine per la tracciabilità. 4.3 Il sistema invia l'ordine allo Spedizioniere. 5. Se la transazione fallisce per credito non sufficiente <ol style="list-style-type: none"> 5.1 Il sistema informa il Cliente che non dispone di credito sufficiente per coprire il pagamento dell'ordine. 5.2 Il sistema offre al Cliente la possibilità di pagare utilizzando una carta di credito diversa. 6. Se la transazione fallisce per qualche altro errore. <ol style="list-style-type: none"> 6.1 Il sistema notifica al Cliente che l'ordine non può essere processato in questo momento e che è necessario riprovare più tardi.

Figura A1.4

Caso d'uso: AutenticaAmministratoreAccount
ID: UC3
Attori: A3 AmministratoreAccount
Include: UC6 CreaNuovoCliente
Punti di estensione:
Precondizioni: 1. L'AmministratoreAccount non è ancora stato autenticato dal sistema.
Sequenza degli eventi: 1. Se il sistema riconosce automaticamente l'AmministratoreAccount 1.1. Il sistema autentica l'AmministratoreAccount come Cliente o come Negoziante. 2. Se il sistema non riesce a riconoscere automaticamente l'AmministratoreAccount 2.1. Il sistema presenta le opzioni "cliente registrato" o "nuovo cliente". 2.2. Se l'AmministratoreAccount seleziona "cliente registrato" 2.2.1. Fintantoché l'AmministratoreAccount non è autenticato ed il numero di tentativi di autenticazione effettuati dall'AmministratoreAccount è inferiore o uguale a 3 2.2.1.1. Il sistema chiede all'AmministratoreAccount di inserire l'identificativo cliente e la password. 2.2.1.2. Il sistema autentica l'AmministratoreAccount come Cliente. 2.3. Se l'AmministratoreAccount seleziona "nuovo cliente" 2.3.1. include(CreaNuovoCliente) 2.3.2. Il sistema autentica l'AmministratoreAccount come Cliente.
Sequenza alternativa: L'AmministratoreAccount può annullare il processo in qualunque momento per spostarsi su un'altra parte del sistema, o per abbandonare del tutto il sistema.
Postcondizioni:

Figura A1.5

Caso d'uso: VisualizzaCarrello
ID: UC7
Attori: A2 Cliente
Punti di estensione: <rimuoviArticolo> <modificaQuantita>
Precondizioni:
Sequenza degli eventi: 1. Se non ci sono articoli nel carrello 1.1. Il sistema visualizza il messaggio "Carrello vuoto". 1.2. Il caso d'uso si conclude. 2. Il sistema visualizza un elenco di tutti gli articoli presenti nel carrello della spesa del Cliente. Questo elenco è composto da una riga per ogni articolo che contiene l'ID del prodotto, il nome dell'articolo, la quantità, il prezzo unitario ed il prezzo totale. <rimuoviArticolo> <modificaQuantita>
Sequenza alternativa: Il Cliente può abbandonare la pagina del carrello della spesa in qualunque momento.
Postcondizioni:

Figura A1.6

Nelle specifiche dei casi d'uso, sono stati inclusi tutti i dettagli importanti dei casi d'uso, omettendo però alcune informazioni generiche del documento (quali i dati relativi alla società, agli autori, alle versioni e altre proprietà). Queste informazioni dipendono tipicamente dal committente, e molti committenti definiscono intestazioni standard da applicare a tutta la documentazione di progetto.

Le specifiche dei casi d'uso possono anche essere gestite e archiviate direttamente con uno strumento CASE UML, ma il supporto per questo tipo di informazioni è, in molti strumenti, limitato al solo testo. Per questo motivo, molti modellatori preferiscono registrare le specifiche dei casi d'uso in un formato di documento più ricco, quale il Word o l'XML, magari inserendo nel modello dei casi d'uso, definito nello strumento CASE, dei collegamenti a questi documenti esterni. L'Appendice 2 illustra qualche idea su come si possa utilizzare l'XML per registrare le specifiche dei casi d'uso.

Appendice 2

L'XML e i casi d'uso

A2.1 Usare l'XML per i template di casi d'uso

Come già trattato, l'UML 1.4 non definisce alcuno standard per documentare i casi d'uso. I modellatori devono quindi utilizzare il supporto, solitamente limitato, offerto dagli strumenti CASE UML, oppure definirsi un loro standard. L'approccio più diffuso sembra essere quello di creare il modello dei casi d'uso con lo strumento CASE, e di inserire nel modello dei collegamenti a documenti esterni che contengono le specifiche dettagliate dei casi d'uso e degli attori. Questi documenti sono solitamente generati con un elaboratore di testi. Tuttavia un elaboratore di testi non è lo strumento ottimale per questo tipo di attività in quanto, nonostante consenta la formattazione e la strutturazione delle specifiche dei casi d'uso e degli attori, non permette di fissare la semantica di questa struttura.

Si ritiene che i documenti XML strutturati siano il formato più adatto per le specifiche dei casi d'uso. L'XML è un linguaggio di *markup* semantico, che separa quindi la struttura semantica del documento dalla sua formattazione. Una volta che una specifica di caso d'uso, o di attore, è stata fissata in un documento XML, è possibile trasformarla e formattarla in molti modi diversi, utilizzando i fogli di stile XSL [Kay 1]. È possibile trasformare tali specifiche in formato HTML, PDF o altri adatti agli elaboratori di testi, ma è anche possibile effettuare ricerche e selezioni su dette specifiche, per estrarne informazioni particolari.

La struttura di documenti XML può essere descritta utilizzando Document Type Definition (Definizione del Tipo di Documento, DTD) o il Linguaggio di Definizione degli Schema XML (XML Schema Definition Language). Sul sito web sono disponibili alcuni semplici schema XML per attori e casi d'uso. Possono essere scaricati e utilizzati gratuitamente, ma sono soggetti alla GNU General Public License (i cui dettagli sono consultabili su www.gnu.org). L'uso e il *feedback* di altri, consentono di apportare continui miglioramenti a questi schema XML.

Una descrizione dettagliata dell'XML e dell'XSL va ben oltre lo scopo sia di questo libro, sia del sito web. Tuttavia, sul nostro sito è possibile trovare dei *link* a risorse utili per chi vuole imparare o approfondire l'XML e l'XSL.

Bibliografia

È possibile consultare una bibliografia completa sul sito web www.umlandtheunifiedprocess.com. Si riporta qui una bibliografia sintetica dei soli testi che sono stati citati esplicitamente nel libro.

[Alur 1], *Core J2EE™ Patterns*, Deepak Alur, John Crupi, Dan Malks, Sun Microsystems Press, 2001, 0130648841

[Ambler 1], *The Unified Process Inception Phase*, Scott W. Ambler, Larry L. Constantine, CMP Books, 2000, 1929629109

[Ambler 2], *The Unified Process Elaboration Phase*, Scott W. Ambler, Larry L. Constantine, Roger Smith, CMP Books, 2000, 1929629052

[Ambler 3], *The Unified Process Construction Phase*, Scott W. Ambler, Larry L. Constantine, CMP Books, 2000, 192962901X

[Booch 1], *Object Solutions*, Grady Booch, Addison-Wesley, 1995, 0805305947

[Booch 2], *The Unified Modeling Language User Guide*, Grady Booch, Ivar Jacobson, James Rumbaugh, Addison-Wesley, 1998, 0201571684

[Chomsky 1], *Syntactic Structures*, Noam Chomsky, Peter Lang Publishing, 1975, 3110154129

[Gamma 1], *Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995, 0201633612

[Jacobson 1], *Unified Software Development Process*, Ivar Jacobson, Grady Booch, James Rumbaugh, Addison-Wesley, 1999, 0201571692

[JSR 1], *JSR-000026*, UML Profile for EJB, disponibile per il download su www.jcp.org, Rational Software Corporation

[Kay 1], *XSLT Programmer's Reference* 2nd Edition, Michael Kay, Wrox Press Inc, 2001, 1861005067

[Kruchten 1], *The Rational Unified Process, An Introduction*, Philippe Kruchten, Addison-Wesley, 2000, 0201707101

[Meyer 1], *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1997, 0136291554

[Pitts 1], *XML Black Book* 2nd Edition, Natalie Pitts, Coriolis Group, 2000, 1576107833

[Rumbaugh 1], *The Unified Modeling Language Reference Manual*, James Rumbaugh, Ivar Jacobson, Grady Booch, Addison-Wesley, 1998, 020130998X

Indice analitico

- abbreviazioni 107-108, 111-112, 116-119
«accede», stereotipo di dipendenza 154-159, 173-178, 182-187
accesso
 determinato dall'ambito 114
 dipendenza di 152-159
ActiveX 341-343, 348-351
aggregati coesi di classi 180-187
 aggregato 266-268, 279-285
aggregazione 258-270
 compositiva
 gerarchie/reti riflessive 266-268
 interdipendenza 180-187, 251-253
interfacce 288-291
multioggetti 198-200
relazione di - tra oggetti 103-106
rifinitura 270-285
riflessiva 266-270
 vedi anche composizione
alternativa, sequenza 62-73, 86-91
altrimenti (else), parola chiave 223-224, 229-235
ambito, accesso determinato dall'- 114
ambito di operazioni ed attributi 106, 112-119
 notazione UML per l'- 113-114
analisi
 artefatti dell'- 93-97, 187, 235-237, 242-245, 276-285
 diagramma delle classi 141-142, 166-168, 180-181, 187-190, 192-208, 215-219
 diagramma di interazione 187-192, 194-198, 215-219
 modellazione dell'- 91-94, 237-245
 modello dell'- 91-97
 relazione di - 263-285
 sistema dell'- 93, 237-240, 242-245
 vista dell'- 239-240, 242-245
 vedi anche classe di analisi, flusso di lavoro dei requisiti
analisi CRC 127-133
analista dei casi d'uso, attore UP 44-46
analista di sistema, attore UP 44-46
'Analizzare l'architettura', attività UP 93-97
'Analizzare un caso d'uso', attività UP 93-97, 119, 130-133, 187, 215-219
'Analizzare un package', attività UP 93-97
'Analizzare una classe', attività UP 93-97
annidato
 classe - 257-263
 flusso di controllo - 194-198
 focus del controllo - 208-215
 grafo di attività - 222
 macchina a stati - 319-321, 323-328, 330-335
 package di analisi - 93, 176-178, 180-187
 sintassi UML per la composizione 268-270
 spazio dei nomi - 173-175
antenato, *vedi* generalizzazione
applicativi, domini 6-7
«ApplicazioneJava», stereotipo di classe 343-348, 354-357
architetto, attore UP 32, 95-97, 235-237, 240-241, 338-339
architettura
 'Analizzare l'architettura' 93-97
 definizione di - 17-19
 dell'UML 7-8, 17-21
 dell'UP 27-41
diagramma di deployment 352-354, 357-359
Enterprise JavaBean 346-348, 354-357
fisica 295-297
hardware 27-28, 341-343, 351-359
'Implementare l'architettura' 338-339
'Progettare l'architettura' 240-241
Architettura del Ciclo di Vita 30-32, 34-35, 37-41
artefatto 3-5, 17-19, 25-30, 32-37
 dei requisiti 58-59
 della progettazione 235-245, 292-297
 dell'analisi 93-97, 187, 235-237, 242-245, 276-285
 dell'implementazione 339
 relazione di «origine» tra - 241-242
asimmetria 266-270, 277-285, 288-291
asincrona, comunicazione 215-219, 228-235, 314-319, 325-328
'Assegnare priorità ai casi d'uso', attività UP 44-46
'Assegnare priorità ai requisiti', attività non UP 44-46, 50-51
assiomi dell'UP 27-28
associazione 137-149
 classe - 146-148, 154-159, 263-265, 278-285
 «comunicazione» 57-58
 definizione di - 133-134, 137-149
 e interdipendenza 94-97
 istanza di un' - 137-149, 154-159
 molteplicità 138-148, 154-159

- molti-a-molti** 144-146, 148-149, 263-265, 276-285
molti-a-uno 148-149, 271-272, 279-285
 nome di - 138-139, 154-159
 qualificata 148-149, 154-159
 relazione 9, 133-134, 137-149, 154-159
 riflessiva 141-142, 266-268, 291-292
 ruolo di - 192-194
 simmetrica 266-268
 sintassi UML per le - 137-149
 tra attore e caso d'uso 57-58
uno-a-molti 144-146, 263-265, 272-276, 279-285
uno-a-uno 144-146, 271, 279-285
- astratto**
 attore 73-76, 86-91
 caso d'uso 76-79, 85-91
 classe 163-171, 251-253, 285-288
 operazione 163-171, 251-253, 285-291
- astrazione**
 ben definita 119-133
 classe di analisi 122-125, 128-133
 dipendenza di - 152, 154-159
 livello di - 164-165
- atomica**, azione 220-222, 229-235, 311-312
- attività** 226-228, 307-312, 316-319
 grafo di - annidato 222
vedi anche diagramma di attività
attività non UP 44-46, 50-51
- attività UP**
 analisi 93-97
'Analizzare un caso d'uso' 119, 130-133, 187, 215-219
 implementazione 338-339
 requisiti 44-46
'Integrare il sistema' 338-339
'Progettare l'architettura' 240-241
'Progettare un caso d'uso' 240-241
'Progettare un sottosistema' 240-241
'Progettare una classe' 240-241
- attivo, oggetto** 203-206, 212-213, 215-219
- attore** 13-15
 astratto 73-76, 86-91
 concreto 73-76, 86-91
 definizione di - 54-56
 generalizzazione tra - 73-76, 85-91
'Individuare attori e casi d'uso' 44-46, 50-51, 53-59, 70-73
 individuare gli - 44-46, 50-59, 70-73
 notazione UML per gli - 54-56
 relazione tra - 133-134
 tempo come - 56, 61-62, 70-73
- attore non UP**, ingegnere dei requisiti 44-46, 50-51
- attore UP**
 analista dei casi d'uso 44-46
 analista di sistema 44-46
 architetto 32, 95-97, 235-237, 240-241, 338-339
 ingegnere dei casi d'uso 240-241
 ingegnere dei componenti 240-241, 338
 integratore di sistemi 338
 progettista di interfaccia utente 44-46
- attributo**
 associazioni 144-146, 154-159
 candidati 122-123, 130-133
 classe associazione 146-148, 154-159
 classi di analisi 122-123, 126-128, 130
 composizione e - 270
 comunicazione tra sotto-macchine 325-328, 332-335
 con ambito di classe/istanza 112-119
 ereditarietà 161-165, 168-171
 sintassi UML per gli - 108-111
 sottosezione degli - 102-103, 106-112, 116-119, 226-228
 valore iniziale 108-111
 valori di - di un caso d'uso 76-79
 valori di - di un oggetto 97-119, 165-168, 192-208, 226-228, 247-248, 258-263, 310-312, 316-319, 325-328, 332
- ausiliaria**, classe 124-125, 247-248, 250
- auto-delegazione** 201-203, 212
- automatico**
 conversione - tra diagrammi di collaborazione/sequenza 190-191, 208-215
 generazione di codice - 144-146, 272-276, 302
 transizione - 222-223, 229-235, 307-310, 316-319
- Avvio**, *vedi* fase di Avvio
- azione** 219-220, 311-319
 di ingresso 219-222, 229-235, 311-312, 316-319, 321-323
 di uscita 311-312, 316-319, 321-323
 stato di - 220-223, 229-235, 307-309, 316-319
- Bag**, classe contenitore dell'OCL 274-275, 279-285
- baseline** 28-30, 32-37
- ben formate**
 classe di analisi - 123-124, 130-133
 classe di progettazione - 248-251, 258-263
- bidirezionale**
 associazione 143-144, 154-159, 171-173, 181-182, 263-265, 277-285
- collegamento** 134-137
- biforcazione**
 diagramma di attività 224-225, 229-235
 diagramma di sequenza 212-219
 diagramma di stato 323-328, 332-335
 «bind», stereotipo di dipendenza 255-263
- Booch**, Grady 5-6, 23-25, 124-125
- Booleano**, *vedi* condizione Booleana
- bozza**, *vedi* prima bozza
- brainstorming** 128, 130-133
- business**
 concetti di - 119-125, 128, 130-133
 regole di - 139-143
- C#** 6-7
 costruzione/distruzione di oggetti 114-116, 208-215
 ereditarietà 251-254
 interfacce 285-288
 visibilità degli attributi 107-108
- C++** 6-7, 258-263
 collegamenti 99-101
 costruzione/distruzione di oggetti 114-116, 208-215
 ereditarietà 251-254
 «friends», dipendenza 153-154
 interfacce 285-288
 segnatura delle operazioni 111-112, 162-163
 template 255-257
 tipi primitivi 270
 visibilità degli attributi 107-108
- CarmelCase** 102-103, 107-108, 111-112, 116-119, 138-139, 154-159, 194-198
- candidato**
 attributi - 122-123, 130-133
 casi d'uso - 56-57
 classi di analisi 125-133
 interfacce - 291-292
 package di analisi 180-181
- Capacità Operativa Iniziale** 30-32, 35-41
- CASE** 3-5, 10-12, 15-17
 classi contenitore 272-276, 279-285
 classi di analisi 108-112, 126-127, 129
 diagrammi di interazione 190-191, 208-215, 222
 flusso di lavoro dell'implementazione 335-337
 generazione di codice 248-251
 manutenzione doppio modello 239-240
 package 171-173, 181-182
 requisiti 66
 stereotipi 15-17
- caso d'uso** 13-15, 27-28, 37-41
'Analizzare un caso d'uso' 93-97, 119, 130-133, 187, 215-219

- 'Assegnare priorità ai casi d'uso' 44-46
astratto 76-79, 85-91
candidati 56-57
come classificatore 76-79
complessi 66-73
concreto 76-79, 85-91
definizione di - 56-58
'Descrivere un caso d'uso' 44-46, 50-51, 59-66, 70-73
«estende» 73, 81-91, 133-134
'Estrarre dai requisiti i casi d'uso' 44-46
flusso di lavoro dei requisiti 43-46, 50
generalizzazione tra - 76-79, 85-91
«include» 73, 79-91, 133-134
'Individuare attori e casi d'uso' 44-46, 50-51, 53-59, 70-73
individuare i - 44-46, 50-51, 53-66, 70-73
individuare le classi di analisi dai - 125-133
istanziazione di - 83
modellazione dei - 51-91
modello dei - 43-44
precondizioni/postcondizioni 59-66, 70-73, 83, 86-91
'Progettare un caso d'uso' 240
punto d'estensione 81-91
relazioni tra - 76-85
segmento inseribile 81-91
sistema dei - 43-44
specifiche di - 59-66, 70-73
'Strutturare il modello dei casi d'uso' 44-46, 50-51
valori di attributo di - 76-79
vista dei - 17-21, 23-25
XML 51-52, 359-365
vedi anche diagramma dei casi d'uso
CBD 23-25, 237-240, 288-291, 298-301, 335-337, 339-341
centrale, meccanica 302-304, 306-307
'chi', concetto - 25-27
«chiam», stereotipo di dipendenza 150-151, 154-159
chiamata, evento di - 313-319
Chomsky, Noam 48-50
ciclica, dipendenza - tra package 180-187
ciclo di vita
 a 'cascata' 23-25, 27-28
 Architettura del Ciclo di Vita 30-32, 34-35, 37-41
 degli oggetti 268-270, 276-277
 del progetto 29-37, 41-43, 50-51, 70-73
 dello sviluppo 3-7
 di un'entità reattiva 307-309, 316-319
modifiche del - 27
Obiettivi del Ciclo di Vita 30-33, 37-41
Class Responsibility Collaborator, *vedi*
analisi CRC
classe 13-15, 97-119
 ambito di - 106, 112-119
 'Analizzare una classe' 93-97
 annidata 257-263
 «ApplicazioneJava» 343-348, 354-357
 associazione riflessiva 141-142, 266-268, 291-292
 associazioni tra - 137-149, 154-159
 astratta 163-171, 251-253, 285-288
 ausiliaria 124-125, 247-248, 250
 concreta 163-171
 contenitore 144-146, 198-200, 245-247, 272-276, 279-285, 288-291
 definizione di - 103-106
 destinazione 134-139, 143-146, 148
 di analisi candidate 125-133
 di utilità 245, 253, 258, 270
 dipendenza tra - 149-154, 255-263
 ereditarietà 103-106, 161-171
 ereditarietà tra - di analisi 94-97, 124-125, 130-133
 ereditarietà tra - di progettazione 251-255, 258-263
 file di - 343-348
 generalizzazione tra - 159-165, 168-171
 'Implementare una classe' 338-339
 «ImplementazioneEJB» 346-348, 354
 individuare le - di analisi 119-133
 inner (Java) 257-263
 «InterfacciaHomeEntitaEJB» 346, 354
 «InterfacciaRemotaEJB» 346-348, 354
 istanziazione 105-106, 116-119
 macchina a stati 307-310
 metodo 97-106, 115-119, 122-123, 134-137, 149-159, 201-203, 219-220, 247-263, 298-301
 mixin 253-254, 258-263
 nascosta 126-127
 notazione UML per le - 106-112
 onnipotente 124-125, 130-133
 origine 134-137, 143-146
 package di analisi 179-187
 «primitivo» 270
 'Progettare una classe' 240-241
 pubblica 292-301
 realizzazione di caso d'uso 189-190, 302
 responsabilità 119-133, 247-251, 258
 sintassi UML per le - con stereotipo 112
 'classe', notazione stile - per le interfacce 285-288, 298-301
classe associazione 146-148, 154-159, 263-265, 278-285
classe contenitore dell'OCL 274-276, 279-285
classe di analisi 93-97, 119-133
 analisi CRC 127-133
 anatomia di una - 122-123
 astrazione 122-125, 128-133
 attributi 122-123, 126-128, 130-133
 ben formate 123-124, 130-133
 definizione di - 119-125
 e classi di progettazione 245, 258-263
 ereditarietà 94-97, 124-125, 130-133
 regole pratiche 124-125
 responsabilità 119-133
classe di progettazione 245-263
 anatomia di una - 247-248
 ben formate 248-251, 258-263
 definizione di - 245-247
 e classi di analisi 245-247, 258-263
 ereditarietà 251-255, 258-263
 responsabilità 247-251, 258-263
 template di classe 255-263
Classe Responsabilità Collaboratore, *vedi*
analisi CRC
classificatore 13-15, 19-21, 103-106, 149-154, 191-192
 caso d'uso come - 76-79
 classe come - 97-102
 ed interfacce 285-288, 298-301
 generalizzazione tra - 73-76, 159-161, 168-171, 190-191
 in un template di classe 255-257
 ruolo - 191-219
 segna come - 228-229
classificazione, schema di - 103-106, 116
classificatore, *vedi anche* descrittore
clip-art, raccolta 352-354
coesione
 di un layer 295-301
 di un package 175-176, 179-187
 di un sottosistema 295, 298-301
 di una classe 123-125, 130-133, 250, 258-263
Coleman, metodo Fusion 5-6
collaborazione 119, 154-161, 219-220, 302, 306-307
vedi anche diagramma di collaborazione
collegamento 133-149, 154-159, 266-268
 associazione riflessiva 141-142, 266-268, 291-292
 classe associazione 146-148, 154-159
 n-ario 135-136, 154-159
 diagrammi di interazione 191-219

- relazione 101-102, 106
 'come', concetto 25-27
 complessità delle interfacce 297-298
 complesso, casi d'uso - 66-73
 completezza
 del modello 11-12
 di una classe 248-249, 258-263
 component-based development, *vedi CBD*
 componente 13-15, 341-359
 definizione di - 341-343
 «DescrittoreEJB» 346-348, 354-357
 dipendenza tra - 341-346, 348-354
 «FileClasseJava» 343-348, 354-357
 flusso di lavoro dell'implementazione 335-341
 interfacce 285-288, 292-301
 istanza di un - 341-343, 352-354, 357
 notazione UML per i - 341-343
 vedi anche diagramma dei componenti,
 diagramma di deployment
 comportamentali, entità 8-9, 19-21
 comportamento
 dinamico 7, 219-220, 307-309, 316-319
 generalizzazione tra attori 73-76, 85-91
 generalizzazione tra casi d'uso 76-79, 85-91
 oggetti 97-102, 106-112, 116-119
 composito
 oggetto - 268-270, 279-285
 stato - 319-325, 328-335
 composizione 263-269
 interdipendenza 180-187, 251-253
 relazione di - tra oggetti 103-106
 rifinitura delle relazioni di analisi 270
 vedi anche aggregazione
 comuni
 distinzioni 13-15, 19-21, 103-106
 meccanismi 7-8, 11-17, 19-21
 comunicazione
 collegamento 133-137, 194-198
 tra sotto-macchine 325-328, 332-335
 tramite stato di sync 325-328, 332-335
 tramite valore di attributo 325-328, 332-335
 «comunicazione», stereotipo di associazione 57-58
 comunicazione asincrona 215-219, 228-235, 314-319, 325-328
 vedi anche segnale
 concetti di business 119-125, 128, 130-133
 concettuale, entità 128, 130-133
 concorrente
 flusso - su un diagramma di attività 224-226, 229-235
 sotto-macchine - 325-328
 stato composito - 319-321, 323-325, 332-335
 concorrenza
 diagramma di collaborazione 203-206, 215-219
 diagramma di sequenza 212-219
 diagramma di stato 319-321, 323-328, 332-335
 situazione di - 309-310
 concreto
 attore 73-76, 86-91
 caso d'uso 76-79, 85-91
 classe 163-171
 operazione 163-171
 condizione Booleana
 caso d'uso di estensione 85-91
 comunicazione tra sotto-macchine 326-327
 evento di variazione 315-316
 Fintantoché, caso 'uso 64-66, 70-73
 ramificazione in un diagramma di collaborazione 201-203, 215-219
 ramificazione in un diagramma di sequenza 212, 215-219
 sequenza alternativa degli eventi 62-64
 transizione di stato 312-313, 316-319, 321-323
 condizione di guardia
 sintassi UML per le - 201-203
 confine del sistema 52-59, 70-73
 contenitore
 classe 144-146, 198-200, 245-247, 272-276, 279-285, 288-291
 classi - dell'OCL 274-276, 279-285
 contratto, 'progettare per -' 285-288, 297-301
 controllo
 flusso di - 194-198
 focus di - 192-219
 controllo ActiveX 341-343, 348-351
 corsia, diagramma di attività 225, 229-235
 costituenti fondamentali dell'UML 8-11, 19-21
 costruttore 106, 114-119, 208-215, 302
 Costruzione, *vedi* fase di Costruzione
 CRC, *vedi* analisi CRC
 «crea», stereotipo di messaggio 208-219
 DAO, pattern 343-346
 Data Access Object, *vedi* DAO
 database 41-43, 50-51, 245-247, 275-277, 292-297, 302-304, 343-348
 dati
 di un oggetto 97-102
 opacità dei -, *vedi* incapsulazione
 dato, tipo di - 13-15
 decisione
 diagramma di attività 223-224, 229-235
 sintassi UML per le - 223-224
 definizione
 dell'UML 3-5, 19-21
 dell'UP 3-5, 19-23
 di architettura 17-19
 di associazione 133-134, 137-149
 di attore 54-56
 di caso d'uso 56-58
 di classe 103-106
 di classe di analisi 119-125
 di classe di progettazione 245-247
 di componente 341-343
 di diagramma di attività 219-220
 di dipendenza 105-106, 149-154
 di evento 313-316
 di flusso di lavoro 25-27
 di generalizzazione 159-161
 di interfaccia 285-288
 di oggetto 97-102
 di package 171-173
 di polimorfismo 165-168
 di realizzazione di caso d'uso 187-189
 di relazione 105-106, 133-134
 di rifinitura di un'associazione di analisi 279-285
 di sottosistema di progettazione 292
 di stato 206-208, 310-312
 di stereotipo 15-17
 Definizione del Tipo di Documento, *vedi* DTD
 deployment
 vista di - 17-21
 vedi anche diagramma di deployment
 «deriva-dai», stereotipo di dipendenza 152, 154-159
 «DescrittoreEJB», stereotipo di componente 346-348, 354-357
 descrittore, *vedi* forma descrittore
 'Describere un caso d'uso', attività UP 44-46, 50-51, 59-66, 70-73
 destinazione, classe 134-139, 143-146, 148-149
 diagramma degli oggetti 10-11, 17-19, 102-103, 135-136, 141-144, 154-159
 diagramma dei casi d'uso 10-11, 17-19, 55-66, 70-79, 81-91
 diagramma dei componenti 10-11, 17-19, 339-343, 346-351, 354-357
 diagramma delle classi 10, 17-19, 106-112
 di analisi 141-142, 166-168, 180-181,

- 187-190, 192-208, 215-219
di progettazione 266-268, 302, 306-307
- diagramma di attività 10-11, 17-19, 219-235, 307-309, 316-319
definizione di - 219-220
- diagramma di collaborazione 10-11, 17-19, 23-25, 189-219, 302-304
iterazione 200-201, 215-219
ramificazione 201-203, 215-219
sintassi UML per i - 194-198
- diagramma di deployment 10-11, 17-19, 241-245, 339-343, 352-354, 357-359
prima bozza di - 241-245, 352, 357-359
sintassi UML per i - 352-354, 357-359
- diagramma di flusso OO, *vedi* diagramma di attività
- diagramma di interazione
dei sottosistemi 304-306
della progettazione 302-307
dell'analisi 187-192, 194-198, 215-219
- diagramma di sequenza 10-11, 17-19, 23-25, 189-191, 201-203, 208-219, 302-306
iterazione 211-212, 215-219
ramificazione 212, 215-219
- diagramma di stato 10-11, 17-19, 23-25, 206-208, 219-220, 222-223, 226-229, 307-335
sintassi UML per i - 310
- diagrammi, costituenti fondamentali 10-11, 19-21
differenza semantica tra stati 310-312
- dimensionato, vettore 255-257
- dinamico
comportamento - 7, 219-220, 307-309, 316-319
modello - 10-11
struttura - 10-11
vista - 23-25, 187
- dipendenza
«accede» 152-159, 173-178, 182-187
«bind» 255-263
«chiama» 150-151, 154-159
ciclica tra package 180-187
definizione di - 105-106, 149-154
«deriva-da» 152, 154-159
di accesso 152-159
di astrazione 152, 154-159
di bind 149-154, 255-263
di instanziazione 105-106, 116-119, 137-149
di uso 150-152, 154-159
«diviene» 206-208, 215-219
«estende» 73, 81-91, 133-134
«friend» 153-159
«import» 153-159, 173-178, 182-187
- «include» 73, 79-91, 133-134
«invia» 151, 154-159
«istanzia» 105-106, 116-119, 137-159
«JARClientEJB» 346-348, 354-357
«origine» 25-27, 43-44, 152, 154-159, 173-175, 182-187, 237-242, 245-247, 302, 337-338
«parametro» 150-151, 154-159
«rifinisce» 152, 154-159
rifinitura delle associazioni di analisi 277-285
«risiede» 341-343, 348-351
tra classi 149-154, 255-263
tra componenti 341-346, 348-354
tra package 173-187
tra sottosistemi 292-301
«usa» 150-152, 154-159, 173-175, 182
- discendente, *vedi* generalizzazione
- disgiunzione semantica delle classi base 253-254, 258-263
- distinzioni comuni, meccanismo comune 13-15, 19-21, 103-106
- distorsione, filtro 48-51
- «distrugge», stereotipo di messaggio 215-219
- distruttore 115-119
- «diviene», stereotipo di dipendenza 206-208, 215-219
- dizionario, classe contenitore 275-276, 279-285
- Document Type Definition, *vedi* DTD
- domini applicativi 6-7
- dominio del problema 17-19, 94-97, 119-133, 235-237, 245-247, 258-263
- dominio delle soluzioni 94-95, 119-125, 130-133, 235-237, 245-247, 258-263, 292-297
- DOORS 47-48, 66
- dopo (after), parola chiave 316
- dovrà 47-48
- DTD 365
- EJB 245-247, 250, 341-343, 346-351, 354-357
- Elaborazione, *vedi* fase di Elaborazione
- elemento
della realizzazione 292-301
della specifica 292-301
- embedded, sistema 6-7, 70, 119-125, 128-129, 203-206
- Enterprise JavaBean, *vedi* EJB
- entità, costituenti fondamentali 8-9, 19-21
- entità concettuale 128, 130-133
- entità reattiva 307-310, 313-319, 325-328
- ereditarietà 103-106, 161-171
- incapsulazione 251-253, 258-263
- multipla 253-254, 258-263
- tra classi di analisi 94-97, 124-125, 130-133
- tra classi di progettazione 251-255, 258
- Ericsson, approccio della 21-25
- errori nella sequenza degli eventi 67, 69-73
- esegui (do), parola chiave 311-312
- 'Eseguire unit test', attività UP 338-339
- esplicito, binding 255-263
- espressione di iterazione
sequenza degli eventi 64-66
sintassi UML per le - 200-201
- essenzialità, di una classe 249-250, 258-263
- «estende», stereotipo di dipendenza 73, 81-91, 133-134
- estendibilità, meccanismi di 3-5, 15-17, 19-21, 105-106, 139-143
- estensione
caso d'uso di - 81-91
condizionale 85-91
punto d' - 81-91
- 'Estrarre dai requisiti i casi d'uso', attività non UP 44-46, 50-51
- etichettato, valore, *vedi* valore etichettato
- evento
definizione di - 313-316
del tempo 313-319
di chiamata 313-319
di segnale 228-235, 313-319
di variazione 313-319
diagramma di stato 307-319
- eXtensible Markup Language, *vedi* XML
- eXtensible Stylesheet Language, *vedi* XSL
- Façade, pattern 295, 298-301
- «façade», stereotipo di package 182-187
- fase di Avvio 23-25, 30-33, 37-41
flusso di lavoro dei requisiti 41-43, 50-51, 70-73
flusso di lavoro dell'analisi 91-97
- fase di Costruzione 23-25, 30-32, 35-41
flusso di lavoro della progettazione 235-237, 242-245
flusso di lavoro dell'implementazione 335-337, 339-341
- fase di Elaborazione 23-25, 30-35, 37-41
flusso di lavoro dei requisiti 41-43, 50-51, 70-73
flusso di lavoro della progettazione 235-237, 242-245
flusso di lavoro dell'analisi 91-97
flusso di lavoro dell'implementazione 335-337
- fase di Transizione 23-25, 30-32, 36-41

- fifo, proprietà classe contenitore 272-276, 279-285
figlio, *vedi* generalizzazione
 «FileClasseJava», stereotipo di componente 343-348, 354-357
file di classe 343-348
file di sorgenti 335-337, 341-343, 348-351
final 166-168
finale, stato 220-224, 229-235, 309-310, 321-323, 332-335
fino-a (until), parola chiave 200, 215-219
fintantoché (while), parola chiave 200-201, 215-219
Fintantoché (While), parola chiave 64-66, 70-73
fisico
 architettura 295-297
 modello 237-240
 oggetti 128-133
fissare i requisiti 17-19, 23-25, 33-35, 41-43, 48-50, 70
flessibilità delle interfacce 297-298
flusso concorrente, diagramma di attività 224-226, 229-235
flusso degli oggetti 226-235
flusso di controllo 194-198
flusso di lavoro 27-37
 analista dei casi d'uso, attore 44-46
 analista di sistema, attore 44-46
 architetto, attore 32, 95-97, 235-237, 240-241, 338-339
 come diagramma di attività 219-235
 definizione 25-27
 ingegnere dei casi d'uso, attore 240-241
 ingegnere dei componenti, attore 240-241, 338-339
 ingegnere dei requisiti, attore 44-46, 50
 integratori di sistemi, attore 338-339
 progettista di interfaccia utente, attore 44
flusso di lavoro dei requisiti 41-43, 50-51, 70-73, 95-97, 119-125
 'Assegnare priorità ai casi d'uso' 44-46
 'Assegnare priorità ai requisiti' 44-46, 50-51
 definizione dei requisiti, 46-50
 'Descrivere un caso d'uso' 44-46, 50-51, 59-66, 70-73
 'Estrarre dai requisiti i casi d'uso' 44-46, 50-51
 filtri 48-51, 61-62
 formulazione dei requisiti 47-48, 50-51
 importanza dei requisiti 46, 50-51
 in dettaglio 44-46
 'Individuare attori e casi d'uso' 44-46, 50-51, 53-59, 70-73
 'Individuare requisiti funzionali' 44-46, 50-51
 'Individuare requisiti non-funzionali' 44-46, 50-51
 modellazione dei casi d'uso 51-91
 nell'UP 29-30, 32, 34-41
 'Prototipare l'interfaccia utente' 44-46
 raccolta dei requisiti 44-46, 48-50
 requisiti del software 43-44
 'Strutturare il modello dei casi d'uso' 44-46, 50-51
vedi anche requisiti funzionali, requisiti non-funzionali
flusso di lavoro dei test, nell'UP 29-30, 32, 34-41
flusso di lavoro della progettazione 94-95, 114-116, 137-149, 235-237, 242-245, 352
 in dettaglio 240-241
 nell'UP 29-30, 32, 34-41
 'Progettare l'architettura' 240-241
 'Progettare un caso d'uso' 240-241
 'Progettare un sottosistema' 240-241
 'Progettare una classe' 240-241
flusso di lavoro dell'analisi 91-97, 119-125
 'Analizzare l'architettura' 93-97
 'Analizzare un caso d'uso' 93-97, 119, 130-133, 187, 215-219
 'Analizzare un package' 93-97
 'Analizzare una classe' 93-97
 in dettaglio 93-94
 nell'UP 29-30, 32, 34-41
flusso di lavoro dell'implementazione 94-95, 335-337, 339-341, 352, 357-359
 'Eseguire unit test' 338-339
 'Implementare l'architettura' 338-339
 'Implementare un sottosistema' 338-339
 'Implementare una classe' 338-339
 in dettaglio 338-339
 'Integrare il sistema' 338-339
 nell'UP 29-30, 32, 34-41
focus di controllo 192-219
 annidato 208-215
fondamentali, costituenti - dell'UML 8-11, 19-21
forcella, icona a - 292-297
forma descrittore
 diagramma dei componenti in - 341-343
 diagramma di collaborazione 192-208
 diagramma di deployment in - 352-354, 357-359
 diagramma di interazione in - 190-191, 215-219, 304-307
 diagramma di sequenza in - 208-215
formalizzazione dei requisiti 47-48, 50-51
forte coesione, di una classe 250, 258-263
 «framework», stereotipo di package 179, 182-187
 «friend», stereotipo di dipendenza 153
functoid 124-125, 130-133
funzione di un oggetto 97-102
Fusion, metodo (Coleman) 5-6
generalizzato, *vedi* generalizzazione
generalizzazione 159-161
 definizione di - 159-161
 filtro 48-51, 61-62
 gerarchia di - 94-97, 124-125, 159-165, 168-171, 180-187, 251-255, 258-263
 relazione di - 9-10, 133-134
 tra attori 73-76, 85-91
 tra casi d'uso 76-79, 85-91
 tra classi 159-165, 168-171
 tra package 178-187
generazione automatica del codice 144-146, 272-276, 302
genitore, *vedi* generalizzazione
gerarchia riflessiva
 di aggregazione 266-268
 di composizione 268-270
gerarchia
 di generalizzazione 94-97, 124-125, 159-165, 168-171, 180-187, 251-255, 258-263
 di oggetti 142-143
 di package 171-173, 182-187
globale, proprietà package 295-297
Glossario di Progetto 58-59, 70-73, 119-130
grafo di attività annidato 222
Gruppo di Lavoro sugli Oggetti, *vedi* OMG
guardia, *vedi* condizione Booleana
hardware 27-28, 119-125, 203-206, 341-343, 351-359
HashMap, classe contenitore Java 275-276
HTML 142-143
 nomi delle classi 107-108
 per i casi d'uso 51-52, 359-365
 per il Glossario di Progetto 58-59
HTTP, protocollo 352-357
HyperText Markup Language, *vedi* HTML
icona a forcella 292-297
icone 136-137

- identità di un oggetto 97-102, 106, 146
 'Implementare l'architettura', attività UP 338-339
 'Implementare un sottosistema', attività UP 338-339
 'Implementare una classe', attività UP 338
 implementazione
 artefatti dell'- 339
 classe di analisi 122-123
 componenti 341-351
 distinzione comune 15, 19-21
 operazioni 163-164
 polimorfismo 165-171
 'progettare per -' 285-288, 297-301
 vista di - 17-21
 vedi anche flusso di lavoro dell'implementazione
 «*ImplementazioneEJB*», stereotipo di classe 346-348, 354-357
 implicito, binding 255-263
 «importa», stereotipo di dipendenza 153-159, 173-178, 182-187
 encapsulato, spazio dei nomi 173-175, 182-187
 encapsulazione
 ereditarietà 251-253, 258-263
 97-102, 116-119, 295
 include (include), parola chiave 330-335
 Include (Include), parola chiave 79-81
 «include», stereotipo di dipendenza 73, 79-91, 133-134
 incompleto, modello 11-12
 inconsistente, modello 11-12
 incrementi 27-30, 35, 37-41
 indefinita, molteplicità 139-143, 154-159
 indicatore
 di stato con memoria 328-330, 332-335
 per la comunicazione tra sotto-macchine 325-328, 332-335
 indicizzata, proprietà classe contenitore 272-276, 279-285
 individuare
 attori 44-46, 50-51, 53-59, 70-73
 casi d'uso 44-46, 50-51, 53-66, 70-73
 classi di analisi 119-133
 interfacce 291
 package di analisi 180-187
 'Individuare attori e casi d'uso', attività UP 44-46, 50-51, 53-59, 70-73
 individuare il confine del sistema 54
 'Individuare requisiti funzionali', attività non UP 44-46, 50-51
 'Individuare requisiti non-funzionali', attività non UP 44-46, 50-51
 informative
 entità - 8-9, 19-21
 vedi anche annotazione
 ingegnere dei casi d'uso, attore UP 240-241
 ingegnere dei componenti, attore UP 240-241, 338-339
 ingegnere dei requisiti, attore non UP 44-46, 50-51
 ingegneria dei requisiti 41-44, 46-53, 58-59, 70-73
 strumenti di - 66
 vedi anche modellazione dei casi d'uso
 ingresso, azione di - 219-222, 229-235, 311-312, 316-319, 321-323
 iniziale
 Capacità Operativa Iniziale 30, 35
 stato 220-225, 229-235, 309-310
 inner, classe (Java) 257-263
 input, flusso degli oggetti 226-228
 inseribile, segmento 81-91
 insieme, proprietà classe contenitore 272-276, 279-285
 'Integrare il sistema', attività UP 338-339
 integratore di sistemi, attore UP 338-339
 interazione 17-19
 tra attori e casi d'uso 55-56, 70-76
 tra oggetti/classi/ruoli 116-119, 189-219
 tra sottosistemi 304-307
 vedi anche diagramma di interazione
 interdipendenza
 tra classi 94-97, 123-124, 130-133, 143-144, 159-161
 tra componenti 348-351
 tra package 171-173, 175-176, 179-187, 288-291, 297-298
 interfaccia 285-301
 candidate 291-292
 CBD (sviluppo basato sui componenti) 288-291
 complessità 297-298
 componenti 285-288, 292-301
 definizione di - 285-288
 distinzione comune 15, 19-21
 Enterprise JavaBean 346-348, 354-357
 flessibilità 297-298
 individuare le - 291
 modello dell'analisi/progettazione 237
 notazione UML per le - 285-291, 298
 progettare per - 291-292
 realizzazione di - 254-255, 258-263
 vantaggi/svantaggi 297-301
 «*InterfacciaHomeEntitaEJB*», stereotipo di classe 346-348, 354-357
 «*InterfacciaRemotaEJB*», stereotipo di classe 346-348, 354-357
 interno, rappresentazione di concetti - 6-7
 interrompibile, attività/sottoattività 222, 229-235, 311-312, 316-319
 «*invia*», stereotipo di dipendenza 151, 154-159
 istantaneo
 azione 220, 229-235, 311, 316
 evento 309-312, 316
 istanza 13-15, 19-21
 ambito di - 112-119
 di un caso d'uso 76-79
 di un componente 341-343, 352-354, 357-359
 di un nodo 352-359
 di una classe, oggetto 95-106, 114-119
 di un'associazione, collegamento 133-134, 137-149, 154-159
 vedi anche forma istanza
 «*istanzia*», stereotipo di dipendenza 105-106, 116-119, 137-159
 istanziazione
 classe/operazione astratta 163, 166-171
 del RUP su un progetto 25-27, 37-41
 dell'UP su un progetto 27, 37-41
 di un caso d'uso 83
 di un template 255-263
 di una classe, oggetto 105-106, 116-119
 di un'associazione, collegamento 137-149
 dipendenza di - 105, 116-119, 137-149
 valore iniziale di un attributo 108-111
 iterazione
 dell'UP 28-32, 37-41
 diagramma di collaborazione 200-201, 215-219
 diagramma di sequenza 211, 215-219
 sequenza degli eventi 64-66
 Jacobson, Ivar 5-6, 23-25
 JAR 343-348, 354-357
 «*JARClientEJB*», stereotipo di dipendenza 346-348, 354-357
 «*JAR-EJB*», stereotipo di package 346-348, 354-357
 Java 6-7
 classe inner 257-263
 classi contenitore 272-276
 collegamenti 134-137
 costruzione/distruzione di oggetti 114-116, 208-215
 ereditarietà 251-254
 final (parola chiave) 166-168
 interfacce 285-291, 295-297

- segnatura delle operazioni 111-112, 162-163
 servlet 341-343, 348-351
 tipi primitivi 270
 visibilità degli attributi 107-108
- Java ARchive, vedi JAR**
- Java Server Page** 341-343, 348-351
- JavaBean** 245-247, 250, 341-343, 346-351, 354-357
- JBoss** 354-357
- layering, pattern** 295-301
- lifo, proprietà classe contenitore** 272-276, 279-285
- lingua, vedi Glossario di Progetto**
- linguaggi di sviluppo** 6-7
- Linguaggio dei Vincoli per gli Oggetti, vedi OCL**
- Linguaggio per le Specifiche e la Descrizione, vedi SDL**
- livello di astrazione** 164-165
- macchina a stati** 307-332
 - annidata 319-321, 323-328, 330-335
 - vedi anche* diagramma di stato
- mappa, classe contenitore** 275, 279-285
- mappatura dei requisiti** 66, 70-73
- Matrice di Mappatura dei Requisiti** 70-73
- meccanica, centrale** 302-304, 306-307
- meccanismo**
 - comuni dell'UML 7-8, 11-17, 19-21
 - di estendibilità dell'UML 3-5, 15-17, 19-21, 105-106, 139-143
 - di persistenza 235, 302, 306
- memoria, stato con -** 328-330, 332-335
- messaggio**
 - «crea» 208-219
 - «distrugge» 215-219
 - interfacce** 291-292, 298-301
 - nei diagrammi di interazione 190-219, 304-306
 - nei diagrammi di stato 309-312
 - polimorfismo** 162-163, 165-171
 - sintassi UML per i - 194-198
 - tra oggetti 101-106, 111-112, 116-119, 134-137, 143-144, 154-159
- meta-modello** 7-8, 15-17, 43-44, 50-51, 93, 173-175, 237-240
 - dei requisiti del software 43-44
- metodo** 97-106, 115-119, 122-123, 134-137, 149-159, 201-203, 219-220, 247-263, 298-301
- metodo Fusion (Coleman)** 5-6
- minima interdipendenza, tra classi** 250-255, 258-263
- mixin, classe** 253-254, 258-263
- modellazione con molteplicità** 141
- modellazione dei casi d'uso** 51-91
 - casi d'uso complessi 66-73
 - tecniche avanzate 73-91
 - vedi anche* flusso di lavoro dei requisiti
- modellazione della progettazione** 237-240, 242-245
 - vedi anche* flusso di lavoro della progettazione
- modellazione dell'analisi** 91-94, 237-240, 242-245
 - vedi anche* flusso di lavoro dell'analisi
- «modello», stereotipo di package** 173-175
- modello dei casi d'uso** 43-44
 - esempio 359-365
 - individuare i package 180-181
- modello della progettazione** 237-242
- modello dell'analisi** 91-97
 - prima bozza di - 126-127, 129-133
- modello di documento standard** 27
- modello fisico** 237-240
- molteplicità** 94-95, 110, 263-265
 - associazioni 138-148, 154-159
 - modellazione con - 141
 - rifinitura delle associazioni di analisi 270-272, 279-285
- multi-a-molti**
 - associazione 144-146, 148-149, 263-265, 276-285
 - relazione - tra requisiti e casi d'uso 66
- multi-a-uno, associazione** 148-149, 271-272, 279-285
- multilivello, stato con memoria -** 329-330, 332-335
- multioggetto** 198-201, 211-212, 215-219
- multipla, ereditarietà** 253-254, 258-263
- n-ario, collegamento** 135-136, 154-159
- nascosto**
 - classe 126-127
 - modello parzialmente - 11-12
- navigabilità** 134-139, 143, 154-159, 171, 181-182, 263-268, 270-271, 278-285
- nodo**
 - computazionale 17-19, 219-220, 241-242, 339, 351-359
 - di una gerarchia 142-143
 - istanza di un - 352-359
- norme, sottosezione del -** 102-103, 106-112, 116-119
- norme di associazione** 138-139, 154-159
- norme di attore** 55-56
- norme di attributo** 103, 108-111, 116-119, 122-123, 144-146, 247-248, 258-263
- norme di caso d'uso** 56-66, 70-73
- nome di classe** 12-13, 102-103, 106-112, 114-125, 127-128, 130-133, 163-164, 248-251, 255-257, 314-315
- nome di interfaccia** 285-288, 304-306
- nome di istanza** 13-15, 194
- nome di messaggio** 304-307
- nome di metodo** 258-263
- nome di oggetto** 43-44, 102-103, 116-119, 194-198, 206-208, 226-228, 292-297
- nome di operazione** 101-102, 111-112, 114-119, 122-123, 161-165, 168-171
- nome di ruolo**
 - di una relazione 108-110
 - di un'associazione 138-139, 143-148, 154-159
 - nei diagrammi degli oggetti 135-136
 - nei diagrammi di collaborazione/sequenza 192-194
 - rifinitura delle classi di analisi 263-265, 270-271, 279-285
- nome di scenario di caso d'uso** 67
- nome di segnale** 228-229, 314-315
- nome di stato** 206-208, 226-228
- nome di stereotipo** 15-17, 112
- non atomico, attività/sottoattività** 222, 229-235
- non interrompibile, azione** 220-222, 229-235, 311-312, 316-319
- non-ordinata, proprietà classe contenitore** 272-276, 279-285
- notazione UML**
 - per gli attori 54-56
 - per gli oggetti 102-103, 116-119
 - per gli stati 206-208, 220-222, 311-312
 - per i componenti 341-343
 - per l'ambito di attributi/operazioni 113
 - per le classi 106-112, 116-119
 - per le interfacce 285-291, 298-301
- null, valore** 110, 116-119
- numero di serie** 97-102
- Obiettivi del Ciclo di Vita** 30-33, 37-41
- Object Constraint Language, vedi OCL**
- Object Management Group, vedi OMG**
- Object Modeling Technique, vedi OMT**
- Objectory, processo** 23-25
- obliquo, percorso** 136-137, 154-159
- OCL** 274-276, 279-285
- oggetto** 97-119
 - aggregato 266-268, 279-285
 - attivo 203-206, 212-213, 215-219
 - collaborazione 119, 154-159, 189-208, 215-219, 302, 306-307
 - collegamento 133-149, 154-159, 266
 - composito 268-270, 279-285

- costruzione 106, 114-119, 208-215, 302-304
 definizione di - 97-102
 differenza semantica tra stati 310-312
 distruzione 115-119
 fisico 128-133
 flusso degli - 226-235
 gerarchia/rete di - 142-143
 identità 97-102, 106, 146-148
 istanziazione 105-106, 116-119
 notazione UML per gli - 102-103, 116
 polimorfismo 165-171
 realizzazione di caso d'uso 302
 ruolo di un - 135-136, 138-139, 154-159
 stato di un - 97-102, 116-119, 206-208, 215-219
 valori di attributo 97-119, 165-168, 192-208, 226-228, 247-248, 258-263, 310-312, 316-319, 325-328, 332-335
- Oggetto di Accesso ai Dati, *vedi* DAO 343
- Oggetto Valore, *vedi* VO
- omonimi 58-59, 70-73, 126-127, 129-133
- onnipotente, classe 124-125, 130-133
- opacità dei dati, *vedi* incapsulazione
- Operativa, Capacità - Iniziale 30-32, 35-41
- operazione
 astratta 163-171, 251-253, 285-291
 classe associazione 146-148, 154-159
 classi di analisi 119-125, 130-133
 con ambito di classe/istanza 106, 112
 concreta 163-171
 costruttore 106, 114-119, 208-215, 302
 dipendenza 149-159
 distruttore 115-119
 ereditarietà 161-165, 168-171
 oggetti 97-102, 116-119
 polimorfismo 165-171
 segnatura di 101-102, 111-112, 116-119, 162-163, 165-171, 194-198, 285-288, 298-301, 313-314
 sintassi UML per le - 111-112
 sottosezione delle - 106-112, 116-119, 292-301
- ordinata, proprietà classe contenitore 272-276, 279-285
- ordinata per chiave, proprietà classe contenitore 272-276, 279-285
- «origine», stereotipo di dipendenza 25-27, 43-44, 152, 154-159, 173-175, 182-187, 237-242, 245-247, 302, 337
- origine
 classe 134-137, 143-146
 relazione di - 25, 43, 154-159, 173-175, 182-187, 237-247, 302, 337-341
- ornamento
 meccanismo comune 12-13, 19-21
 opzionale per le classi 106-112, 122-123
 visibilità 108-110
- ortogonale, percorso 136-137, 154-159
- output, flusso degli oggetti 226-228
- override, *vedi* ridefinire
- package 8-9, 19-21, 43-44, 93, 171-187
 aggregati coesi di classi 180-187
 analisi dell'architettura 179-182
 'Analizzarne un package' 93-97
 annidato 93, 176-178, 180-187
 candidati 180-181
 classi di analisi 179-187
 definizione di - 171-173
 dipendenza tra - 180-187
 «façade» 179, 182-187
 figli 178-179
 «framework» 179, 182-187
 generalizzazione tra - 178-187
 gerarchia di - 171-173, 182-187
 individuare i - di analisi 180-187
 «JAR-EJB» 346-348, 354-357
 «modello» 173-175
 sintassi UML per i - 93, 171, 176
 «sistemi» 179, 182-187
 «sottosistemi» 179, 182-187, 292-297
 stereotipi di - 179, 182-187
 «stub» 179, 182-187
 «topLevel» 171-173, 182-187
 visibilità 171-173, 181-187
- 'palloncino', notazione stile - per le interfacce 285-291, 298-301
- «parametro», stereotipo di dipendenza 150-151, 154-159
- parola chiave
 altrimenti (else) 223-224, 229-235
 dopo (after) 316
 dovrà 47-48
 esegui (do) 311-312
 final 166-168
 fino-a (until) 200-201, 215-219
 fintantoché (while) 200-201, 215-219
 fintantoché (While) 64-66, 70-73
 include (include) 330-335
 Include (Include) 79-81
 Per (For) 64, 70-73
 per-ogni (for) 200-201, 215-219
 quando (when) 315-316
 Se (If) 62, 70-73
- parte, relazione tutto- 263-285
- parti interessate 19-21, 41-43, 46, 58-59, 85-91, 94-97, 125-129, 235-237
- passo della sequenza degli eventi 61-62
- pathname 152-153, 173-178
- pattern
 DAO 343-346
 di layering 295-301
 Façade 295, 298-301
- Per (For), parola chiave 64, 70-73
- percorso, collegamento 136-137, 154-159
- per-ogni (for), parola chiave 200-201, 215-219
- persistenza, meccanismo di 235-237, 302-304, 306-307
- plug-in, *vedi* CBD
- polimorfismo 159-171
 definizione di - 165-168
- postcondizioni, sequenza degli eventi 59-66, 70-73, 83, 86-91
- precondizioni, sequenza degli eventi 59-66, 70-73, 83, 86-91
- predicato
 nominale/verbale 126-127, 130-133, 138-139, 154-159
 verbale 56-57, 111-112, 126-127, 130-133, 138-139, 154-159, 220-222
- prima bozza
 di diagramma di deployment 241-245, 352, 357-359
 di modello dell'analisi 126-127, 129-133
 di sistema 33-34
- «primitivo», stereotipo di classe 270
- principale, scenario 67, 70-73
- principio di sostituibilità 73-76, 86-91, 159-171, 178-179, 253-254, 258-263
- privata, visibilità 108-110, 153-154, 171-173, 182-187
- problema, dominio del 17-19, 94-97, 119-133, 235-237, 245-247, 258-263
- processo, vista dei 17-21
- processo di ingegneria del software, *vedi* SEP
- processo di sviluppo del software, *vedi* SDP
- processo Objectory 23-25
- Processo Rational Objectory, *vedi* ROP
- Processo Unificato per lo Sviluppo del Software, *vedi* USDP
- Processo Unificato Rational, *vedi* RUP
- Prodotto, Rilascio del 30-32, 36-41
- 'Progettare l'architettura', attività UP 240-241
- 'progettare per contratto' 285-288, 297-301
- 'progettare per implementazione' 285-288, 297-301
- 'Progettare un caso d'uso', attività UP 240

- 'progettazione
 artefatti della- 235-245, 292-297
 diagramma delle classi 266-268, 302, 306-307
 diagramma di interazione 302-307
 modello della - 237-240, 242-245
 realizzazione di caso d'uso della - 241
 relazione di - 263-285
 sistema della - 237-240, 242-245, 337
 strategica 235-237, 242-245, 302
 tattica 235-237, 242-245, 302
- progettista di interfaccia utente, attore UP 44-46
- Progetto, Glossario di 58, 70, 119-130
- proprietà 17, 19-21
 attivo, oggetto 203-206, 215-219
 classe contenitore 272-276, 279-285
 globale, package 295-297
- protetta, visibilità 108-110, 171-173, 182-187
- protocollo HTTP 352-357
- 'Prototipare l'interfaccia utente', attività UP 44-46
- pubblica, visibilità 108-110, 152-153, 171-173, 182-187
- punto di estensione 81-91
 sottosezione dei - 81-85
- punto di inserimento, sintassi UML per i - 79-81
- qualificata, associazione 148-149, 154-159
- 'quando', concetto 25-27
- quando (when), parola chiave 315-316
- quantificatore universale 48-51
- queue, proprietà classe contenitore 272-276, 279-285
- raccolta clip-art 352-354
- raggruppamento
 entità di 8-9, 19-21
vedi anche package
- ramificazione
 diagramma di collaborazione 201-203, 215-219
 diagramma di sequenza 212, 215-219
 sequenza degli eventi 62-64
- rappresentazione di concetti interni 6-7
- Rational Corporation 5-6, 21-27, 47, 346
- Rational Objectory Process, *vedi* ROP
- Rational Requisite Pro 47, 66
- Rational Unified Process, *vedi* RUP
- realizzazione
 elementi della - 292-301
 relazioni di - 9-10
- realizzazione di caso d'uso 93-97, 171-173, 187-189, 215-219
- definizione di - 187-189
 della progettazione 241-245
 diagramma di collaborazione 190-208, 215-219
 diagramma di interazione 187-190, 302
 diagramma di sequenza 190-191, 208-219
 elementi 189-191, 215-219
 reattiva, entità 307-310, 313-319, 325-328
 regole di business 139-143
 regole pratiche
 classi di analisi 124-125
 modello dell'analisi 94-97
- reificare una relazione 146-148, 154-159, 276-285
- relazione 9-10, 133-134, 154-159
 associazione 9-10, 133-134, 137-149, 154-159
 classe associazione 146-148, 154-159, 263-265
 collegamento 101-102, 106
 costituenti fondamentali 9-10, 19-21
 definizione di - 105-106, 133-134
 di analisi 263-285
 di generalizzazione 9-10
 di origine 25-27, 43, 152, 154-159, 173-175, 182-187, 237-247, 302, 337-341
 di progettazione 263-285
 di realizzazione 9-10
 ereditarietà tra classi 161-165, 168-171
 molteplicità 138-148, 154-159
 navigabilità 134-139, 143-144, 154-159, 171-173, 181-182, 263-268, 270-271, 278-285
 reificare una - 146, 154-159, 276-285
 rifinitura delle - di analisi 270-285
 tra attori 73-76
 tra casi d'uso 76-85
 tra classi ed oggetti 105-106
 transitività 175-176
 tutto-parte 263-285
- Requisite Pro 47, 66
- requisiti
 artefatti dei 58-59
 'Assegnare priorità ai requisiti' 44-46, 50-51
 'Estrarre dai requisiti i casi d'uso' 44-46, 50-51
 formulazione dei - 47-48, 50-51
 importanza dei - 46, 50-51
 'Individuare requisiti funzionali' 44-46, 50-51
 'Individuare requisiti non-funzionali' 44-46, 50-51
- meta-modello dei - del software 43-44
 raccolta dei - 44-46, 48-50
- requisiti funzionali
 classi di analisi 119-125
 confine del sistema 54
 fissare i - 33-34
 flusso di lavoro dei requisiti 41-51
 mappatura dei - 66
 modellazione dei casi d'uso 70-73
 realizzazione di caso d'uso dell'analisi 187-189, 215-219
- requisiti non-funzionali
 classi di analisi 119-125
 confine del sistema 54
 flusso di lavoro dei requisiti 41-51
 modellazione dei casi d'uso 70-73
 realizzazione di caso d'uso della progettazione 302
- responsabilità
 classe di analisi 119-133
 classe di progettazione 247-251, 258
- rete
 di oggetti 142-143
 riflessiva di aggregazione 266-268
- ricongiunzione
 diagramma di attività 224-225, 229-235
 diagramma di sequenza 215-219
 diagramma di stato 323-328, 332-335
- ridefinire
 ereditarietà tra classi 162-163, 168-171
 ereditarietà tra package 178-179
 operazioni concrete 166-168
- «rifinisce», stereotipo di dipendenza 152, 154-159
- rifinitura
 dei diagrammi di interazione dell'analisi 302-304, 306-307
 delle relazioni di analisi 270-285
 realizzazione di caso d'uso 189-190
- riflessiva
 aggregazione 266-270
 associazione 141-142, 266, 291
 composizione 268-270
 gerarchia - di composizione 268-270
 gerarchia/rete - di aggregazione 266-268
- Rilascio del Prodotto 30-32, 36-41
- rimando a stato 330-335
- rimozione, filtro 48-51, 61-62
- rischio 23-25, 27-28, 32-34, 37-41
- «risiede», stereotipo di dipendenza 341-343, 348-351
- ROP 23-25
- Rumbaugh, James 5-6, 23-25

- ruolo**
- classificatore 191-219
 - corsie 225-226
 - degli oggetti 135, 138, 154-159
 - di associazione 192-194
 - diagramma di interazione 190-192
 - ereditarietà 251-253, 258-263
 - nei diagrammi di collaborazione/sequenza 190-219
 - RUP 21, 23-27, 37-41, 44-46
- scenari**
- caso d'uso 52-53, 66-73
 - sequenza degli eventi 66-70
- schema di classificazione** 103-106, 116-119
- SDL** 23-25
- SDP** 21-23
- Se (If), parola chiave** 62, 70-73
- secondario, scenario** 67-73
- segmento inseribile** 81-91
- segnale** 23-25, 151, 154-159, 228-235, 313-319
 - come classificatore 228-229
 - evento di - 228-235, 313-319
- segnatura di un'operazione** 101-102, 111-112, 116-119, 162-163, 165-171, 194-198, 285-288, 298-301, 313-314
- semantico**
 - differenza - tra stati 310-312
 - substrato - del modello 11-12
- semplisce, stato con memoria** - 328-329, 332-335
- SEP** 21-27, 32-33, 37-41
- Sequence, classe contenitore dell'OCL** 274-275, 279-285
- sequenza, vedi diagramma di sequenza**
- sequenza alternativa** 62-64, 66-73, 86-91
- sequenza degli eventi** 59-73, 76-91
 - espressione di iterazione 64-66
 - iterazione 64-66
 - passo 61-62
 - precondizioni/postcondizioni 59-66, 70-73, 83, 86-91
 - ramificazione 62-64
 - scenari 66-70
- sequenziale, stato composito** - 319-323, 332-335
- serie, numero di** 97-102
- servlet Java** 341-343, 348-351
- Set, classe contenitore dell'OCL** 274-275, 279-285
- simmetria** 266-268
- sincronizzazione** 203-206, 224-226, 229-235, 323-328
- sinonimi** 58-59, 70-73, 126-127, 129-133
- sintassi UML**
 - per gli attributi 108-111
 - per gli stati di sync 327-328
 - per i diagrammi di collaborazione 194
 - per i diagrammi di deployment 352
 - per i diagrammi di stato 310
 - per i messaggi 194-198
 - per i package 93, 171-173, 176-178
 - per i punti di inserimento 79-81
 - per i template 255-257
 - per i valori etichettati 17
 - per la composizione 268-270
 - per le associazioni 137-149
 - per le classi con stereotipo 112
 - per le condizioni di guardia 201-203
 - per le decisioni 223-224
 - per le espressioni di iterazione 200-201
 - per le operazioni 111-112
 - per le transizioni 312-313
 - vedi anche* notazione UML
- sistema**
 - confine del - 52-59, 70-73
 - dei casi d'uso 43-44
 - della progettazione 237-240, 242-245, 337-338
 - dell'analisi 93, 237-240, 242-245
 - '*Integrare il sistema*' 338-339
 - prima bozza di - 33-34
 - «*sistema*», stereotipo di package 179, 182-187
 - sistema embedded 6-7, 70, 119-125, 128-129, 203-206
 - situazione di concorrenza 309-310
 - software development process, *vedi* SDP
 - software engineering process, *vedi* SEP
 - soluzioni, dominio delle 94, 119, 130, 235, 245, 258, 292
 - sorgenti, file di 335-337, 341-343, 348-351
 - sostantivo
 - analisi -/verbo 125-133
 - 126-127, 130-133, 138-139, 154-159
 - sostituibilità, principio di 73, 86-91, 159-171, 178, 253
 - sottoattività, stato di 220-235, 307-309, 316-319
 - sottoclasse 159-171, 251-255, 258-263
 - sottolineatura, notazione UML 102-103, 116-119
 - sotto-macchina 319-328, 330-335
 - comunicazione tra - 325-328, 332-335
 - sottosezione delle - 319-321
 - stato della - 330-335
 - «*sottosistema*», stereotipo di package 179, 182-187, 292-297
- sottosezione** 11-12
 - analisi CRC 127-128, 130-133
 - degli attributi 102-103, 106-112, 116-119, 226-228
 - degli elementi della realizzazione 292-301
 - degli elementi della specifica 292-301
 - degli stati delle sotto-macchine 319-321
 - dei punti di estensione 81-85
 - del nome 102-103, 106-112, 116-119, 292-301
 - per gli attori 55-56
 - per gli oggetti 102-103, 116-119, 226
 - per gli stati 319-321
 - per i sottosistemi 292-301
 - per le classi 106-112, 116-119
- sottosistema**
 - diagramma di interazione dei - 304-306
 - dipendenza tra - 292-301
 - '*Implementare un sottosistema*' 338-339
 - interazione tra - 304-307
 - '*Progettare un sottosistema*' 240-241
- sottosistema di implementazione** 292-297, 335-337, 339-343, 348-351
 - definizione di - 292-297
- sottosistema di progettazione** 237-245, 292-301
 - definizione di - 292-297
 - interfacce e - 285-291, 294-301
- sottostato** 319-323, 328-335
- spazio dei nomi**
 - annidato 173-175
 - 152-159, 171-178, 182-187, 257-263
- specializzazione** 159-161
- specificità, elementi della** 292-301
- Specification and Description Language, vedi** SDL
- specifiche** 41-43, 46-51
 - meccanismo comune 11-12, 19-21
- Specifiche dei Requisiti di Sistema, vedi** SRS
- specifiche di caso d'uso** 59-66, 70-73
- SRS** 47-48, 52-53, 66
- stack, proprietà classe contenitore** 272-276, 279-285
- statica, struttura** 7, 10-11, 187
- statico**
 - modello - 10-11, 23-25, 180-181
 - struttura - 187
- stato** 219-235, 307-335
 - composito 319-325, 328-335
 - definizione di - 206-208, 310-312
 - della sotto-macchina 330-335

- di azione 220-222, 229-235
 di sottoattività 220-235, 307-309, 316
 di sync 325-328, 332-335
 differenza semantica tra - 310-312
 finale 220-224, 229-235, 309-310, 321-323, 332-335
 iniziale 220-225, 229-235, 309-310
 memoria 328-330, 332-335
 notazione UML per gli - 206-208, 220-222, 311-312
 rimando a - 330-335
 sottostato 319-323, 328-335
 superstato 319-325, 328-330, 332-335
 transizione di - 97-102, 116-119, 206-208, 213-215, 219-235, 307-335
vedi anche diagramma di stato
- stereotipo**
 «accede», dipendenza 152-159, 173-178, 182-187
 «ApplicazioneJava», classe 343-348, 354-357
 «bind», dipendenza 255-263
 «chiama», dipendenza 150, 154-159
 «comunicazione», associazione 57-58
 «crea», messaggio 208-219
 definizione di - 15-17
 «deriva-dà», dipendenza 152, 154-159
 «DescrittoreEJB», componente 346, 354
 di una classe di analisi 122-123
 «distrugge», messaggio 215-219
 «diviene», dipendenza 206, 215-219
 «estende», dipendenza 73, 81-91, 133
 «façade», package 179, 182-187
 «FileClasseJava», componente 343-348, 354-357
 «framework», package 179, 182-187
 «friend», dipendenza 153-159
 «ImplementazioneEJB», classe 346-348, 354-357
 «importa», dipendenza 153-159, 173-178, 182-187
 «include», dipendenza 73, 79-91, 133
 «InterfacciaHomeEntitàEJB», classe 346-348, 354-357
 «InterfacciaRemotaEJB», classe 346-348, 354-357
 «invia», dipendenza 151, 154-159
 «istanzia», dipendenza 105-106, 116-119, 137-159
 «JARClientEJB», dipendenza 346-348, 354-357
 «JAR-EJB», package 346-348, 354-357
 meccanismo di estendibilità 15-17, 105
 «modello», package 173-175
- «origine», dipendenza 25-27, 43-44, 152, 154-159, 173-175, 182-187, 237-242, 245-247, 302, 337-338
 «parametro», dipendenza 150, 154-159
 «primitivo», classe 270
 «rifinisce», dipendenza 152, 154-159
 «risiede», dipendenza 341, 348-351
 sintassi UML per le classi con - 112
 «sistema», package 179, 182-187
 «sottosistema», package 179, 182-187, 292-297
 «stub», package 179, 182-187
 «topLevel», package 171-173, 182-187
 «usa», dipendenza 150-152, 154-159, 173-175, 182-187
 strategica, progettazione 235-237, 242-245, 302
 struttura
 dell'UML 7-8
 dell'UP 28-30
 statica 7, 10-11, 187
 strutturali, entità 8-9, 19-21
 'Strutturare il modello dei casi d'uso', attività UP 44-46, 50-51
 «stub», stereotipo di package 179, 182-187
 substrato semantico del modello 11-12
 sufficienza, di un classe 248-249, 258-263
 superclasse 159-168, 251-255
 superstato 319-325, 328-330, 332-335
 sviluppo, linguaggi di - 6-7
 sviluppo basato sui componenti, *vedi* CBD
 sync, stato di - 325-328, 332-335
 System Requirements Specification, *vedi* SRS
 tattica, progettazione 235, 242, 302
 Tecnica di Modellazione degli Oggetti, *vedi* OMT
 tecniche avanzate, per i diagrammi di stato 319-335
 template
 di classe 255-263
 istanziazione 255-263
 sintassi UML per i - 255-257
 vettore dimensionato 255-257
 XML per i - di caso d'uso e di attore 51-52, 365
 tempo
 come attore 56, 61-62, 70-73
 evento del 313-319
 terminologia, *vedi* Glossario di Progetto tipo
 di dato 13-15
 parametrizzato 255-263
- «topLevel», stereotipo di package 171-173, 182-187
 transitività 175-176, 182-187, 266-270, 279-285
 Transizione, *vedi* fase di Transizione
 transizione di stato 97-102, 116-119, 206-208, 213-215, 219-235, 307-335
 sintassi UML per le - 312-313
 tutto-parte
 relazione - 263-285
vedi anche aggregazione, composizione
 UML, definizione dell'- 3-5, 19-21
 unidirezionale
 associazione 143-144, 171-173, 181-182, 270-271, 277-285, 288-291
 collegamento 134-137, 143-144
 Unified Software Development Process, *vedi* USD^P
 universale, quantificatore 48-51
 uno-a-molti, associazione 144-146, 263-265, 272-276, 279-285
 uno-a-uno
 associazione 144-146, 271, 279-285
 relazione di «origine» tra artefatti 237-242
UP
 definizione dell'- 3-5, 19-23
 fasi 32-41
 istanziazione dell'- su un progetto 27, 37-41
 «usa», stereotipo di dipendenza 150-152, 154-159, 173-175, 182-187
 uscita, azione di 311-312, 316-319, 321-323
 USD^P 21-23
 uso, dipendenza di 150-152, 154-159
 utilità, classe di 245-247, 253-254, 258-263, 270
 valore di attributo
 di un caso d'uso 76-79
 di un oggetto 97-119, 165-168, 192-208, 226-228, 247-248, 258-263, 310-312, 316-319, 325-328, 332-335
 valore etichettato
 classi 110-112, 122-123, 272-276
 interfacce 285-288
 meccanismo di estendibilità 15, 19
 profili UML 341
 requisiti 66
 sintassi UML per i - 17
 valore null 110, 116-119
 Value Object, *vedi* VO
 variazione, evento di 313-319
 verbo 56-57, 111-112, 126-127, 130-133,

- 138-139, 154-159, 220-222
- verbo, analisi sostantivo/- 125-133
- vettore 110, 116-119, 144-146
 - array 272, 279-285
 - dimensionato 255-257
- vincolo
 - classe associazione 278-285
 - costruzione/distruzione di oggetti 194-198
 - diagramma di sequenza 213-215
 - ereditarietà di classe 161-165, 168-171
 - interfacce 285-288, 298-301
 - meccanismo di estendibilità 15-17
 - molteplicità 139-143
 - profili UML 341
- visibilità
 - attributo 108-110, 122-123, 247-248, 258-263
 - ornamento 108-110
 - package 171-173, 181-187
- vista dei casi d'uso 17-21, 23-25
- vista dei processi 17-21
- vista dell'analisi 239-240, 242-245
- vista di deployment 17-21
- vista di implementazione 17-21
- vista logica 17-21
- Visual Basic 6-7
 - interfacce 285-288
 - visibilità degli attributi 107-108
- VO 343-346
- vuoto, valore null 110, 116-119
- XML 142-143
 - descrittore di deployment EJB 346-348
 - nomi delle classi 107-108
 - per gli attori 51-52
 - per i casi d'uso 51-52, 359-365
 - per il Glossario di Progetto 58-59
- XSL 51-52, 365