Leszek A. Maciaszek

Sviluppo di sistemi informativi con UML

Analisi dei requisiti e progetto di sistema

ADDISON-WESLEY

CAGLIARI

UBO/01714906
SSD 6619

HFN = 8867
AMS: 68N15.87
(Linguaggi di programmazione)

Sviluppo di sistemi informativi con UML

Analisi dei requisiti e progetto di sistema

DIPARTIMENTO DI MATEMATICA
BIBLIOTECA
UNIVERSITÀ CAGLIARI
AMS
68N15
87

Sviluppo di sistemi informativi con UML

Analisi dei requisiti e progetto di sistema

LESZEK A. MACIASZEK
Macquarie University, Sydney, Australia

Inventario n. 6619
SEMINARIO SCIENTIFICO

ADDISON-WESLEY

Pearson Education Italia S.r.l.
A Pearson Education Company

Copyright © 2002 Pearson Education Italia S.r.l.
Via Fara, 28
20124 Milano
Tel. 02/67 39 76 392 Fax 02/67 39 76 503
E-mail: info@addison.it
Web: http://www.addison.it/

Authorized translation from the English language edition, entitled:
Requirements Analysis and System Design – Developing Information Systems with UML, 1st Edition, by Leszek A. Maciaszek
Published by Pearson Education Limited publishing as Addison Wesley
Copyright © 2001 by Pearson Education Limited

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata all'Autore, alla Pearson Education Italia o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.
LA FOTOCOPIATURA DEI LIBRI È UN REATO. L'editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume. Le richieste di riproduzione vanno inoltrate ad AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), Via delle Erbe, 2 - 20121 Milano - Tel. e Fax 02/80.95.06.

Traduzione: Giuseppe Berio
Revisione tecnica: Antonio Di Leva
Copy editing: Felicia Barbaro
Composizione: TOTEM di Astolfi Andrea
Grafica di copertina: Gianni Gilardoni
Stampa: Arti Grafiche Battaia – Zibido S. Giacomo (MI)

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

ISBN 88-7192-138-0

Printed in Italy

1a edizione aprile 2002

A mia madre

Alla memoria di mio padre

Alla mia famiglia

Sommario

Presentazione dell'edizione italiana xix

Prefazione xxi

Argomento del testo xxi

Caratteristiche distintive del testo xxii

A chi si rivolge xxiii

Organizzazione del testo xxiv

Materiale supplementare xxv

Ringraziamenti xxvi

Diagrammi d'attività del testo xxviii

Acquisti OnLine (Sviluppo guidato) xxix

Immatricolazioni universitarie (Caso di studio) xxxi

Video Store (Caso di studio) xxxiii

Gestione Contatti con i Clienti (CMS) (Caso di studio) xxxiv

Telemarketing (Caso di studio) xxxvi

1 Il processo di sviluppo del software 1

1.1 La natura dello sviluppo del software 1

1.1.1 Il principio di sviluppo del software 2

1.1.2 Stakeholder 3

1.1.3 Processo 4

1.1.3.1 Processo iterativo e incrementale 5

1.1.3.2 Capability maturity model 5

1.1.3.3 ISO 9000 7

1.1.4 Linguaggi e strumenti di modellazione 7

1.1.4.1 Unified modeling language 8

1.1.4.2 Strumenti CASE e miglioramento del processo 9

1.2 Pianificazione di sistema 9
1.2.1 Approccio SWOT 10
1.2.2 Approccio VCM 11
1.2.3 Approccio BPR 12
1.2.4 Approccio ISA 13
1.2.5 Livelli gestionali di un'organizzazione 14
1.3 Fasi del ciclo di vita del software 15
1.3.1 Fase di determinazione dei requisiti 16
1.3.2 Fase di specifica dei requisiti 16
1.3.3 Fase di progetto architetturale 17
1.3.4 Fase di progetto di dettaglio 17
1.3.5 Fase d'implementazione 18
1.3.6 Fase d'integrazione 18
1.3.7 Fase di manutenzione 19
1.3.8 Pianificazione di progetto nel ciclo di vita del software 20
1.3.9 Metriche nel ciclo di vita del software 21
1.3.10 Testing nel ciclo di vita del software 22
1.4 Approcci allo sviluppo del software 23
1.4.1 Approccio strutturato 23
1.4.2 Approccio object-oriented 24
Sommario 25
Domande 26
2 Fondamenti di analisi dei requisiti 29
2.1 Fondamenti della tecnologia a oggetti 29
2.1.1 Oggetto istanza 30
2.1.1.1 Notazione per gli oggetti 30
2.1.1.2 Come collaborano gli oggetti? 31
2.1.1.3 Come un oggetto identifica altri oggetti? 32
2.1.2 Classe 34
2.1.2.1 Attributo 35
2.1.2.2 Operazione 36
2.1.3 Associazione 37
2.1.3.1 Grado di un'associazione 38
2.1.3.2 Molteplicità di un'associazione 38
2.1.3.3 Legame ed estensione di un'associazione 39
2.1.3.4 Classe di associazione 40

2.1.4 Aggregazione e composizione 40
2.1.5 Generalizzazione 41
2.1.5.1 Polimorfismo 42
2.1.5.2 Ereditarietà 43
2.1.5.3 Classe astratta 44
2.1.6 Oggetto classe 45
2.2 Analisi guidata di un caso di studio 46
2.2.1 Il caso di studio: Acquisti OnLine 47
2.2.2 Modellazione dei casi d'uso 47
2.2.2.1 Attori 48
2.2.2.2 Casi d'uso 49
2.2.2.3 Diagramma dei casi d'uso 49
2.2.2.4 Documentare i casi d'uso 52
2.2.3 Modellazione delle attività 52
2.2.3.1 Attività 54
2.2.3.2 Diagramma di attività 54
2.2.4 Modellazione delle classi 56
2.2.4.1 Classi 56
2.2.4.2 Attributi 58
2.2.4.3 Associazioni 60
2.2.4.4 Aggregazioni 61
2.2.4.5 Generalizzazioni 62
2.2.4.6 Diagramma delle classi 62
2.2.5 Modellazione dell'interazione 64
2.2.5.1 Interazioni 64
2.2.5.2 Operazioni 66
2.2.5.3 Diagramma di sequenza 67
2.2.6 Modellazione con statechart 68
2.2.6.1 Stati e transizioni 70
2.2.6.2 Diagrammi statechart 71
2.3 Descrizioni dei casi di studio 72
2.3.1 Immatricolazioni universitarie 73
2.3.2 Video Store 74
2.3.3 Gestione Contatti con i Clienti (CMS) 74
2.3.4 Telemarketing 75
Sommario 76
Domande 77
Esercizi 78

3 Determinare i requisiti 79

3.1 Principi di determinazione dei requisiti 79

3.2 Raccolta dei requisiti 80

3.2.1 Metodi tradizionali per la raccolta dei requisiti 81
3.2.1.1 Interviste 81
3.2.1.2 Questionari 82
3.2.1.3 Osservazioni 83
3.2.1.4 Studio dei documenti e dei sistemi software 83

3.2.2 Metodi moderni di raccolta dei requisiti 84
3.2.2.1 Prototipi 84
3.2.2.2 Sviluppo cooperativo delle applicazioni (JAD) 84
3.2.2.3 Sviluppo rapido delle applicazioni (RAD) 85

3.3 Negoziazione e validazione dei requisiti 86

3.3.1 Requisiti fuori dal contesto 86
3.3.2 Matrice di dipendenza dei requisiti 87
3.3.3 Priorità e rischi associati ai requisiti 87

3.4 Gestione dei requisiti 88

3.4.1 Identificazione e classificazione dei requisiti 88
3.4.2 Gerarchie di requisiti 89
3.4.3 Gestione del cambiamento 90
3.4.4 Tracciabilità dei requisiti 90

3.5 Modello dei requisiti di business 91

3.5.1 Modello dello scopo del sistema 91
3.5.1.1 Esempio di modellazione dello scopo di sistema 92

3.5.2 Modello dei casi d'uso di business 93
3.5.2.1 Esempio di modellazione dei casi d'uso di business 94

3.5.3 Modello delle classi di business 95
3.5.3.1 Esempio di modellazione delle classi di business 95

3.6 Documento dei requisiti 96

3.6.1 Schemi di documento 97
3.6.2 Premesse del progetto 98
3.6.3 Servizi del sistema 98
3.6.4 Vincoli del sistema 99
3.6.5 Aspetti progettuali 99
3.6.6 Appendici 100

Sommario 100

Domande 101

Esercizi 102

4 Specifica dei requisiti 103

4.1 Principi di specifica dei requisiti 103

4.2 Specifiche dello stato 104

4.2.1 Modellazione delle classi 105
4.2.1.1 Trovare le classi 105
4.2.1.2 Specificare le classi 112

4.2.2 Modellazione delle associazioni 120
4.2.2.1 Determinare le associazioni 121
4.2.2.2 Specificare le associazioni 121
4.2.2.3 Esempio di specifica di associazioni 122

4.2.3 Modellazione delle relazioni di aggregazione e composizione 123
4.2.3.1 Determinare aggregazioni e composizioni 124
4.2.3.2 Specificare aggregazioni e composizioni 125
4.2.3.3 Esempio di specifica di aggregazioni e composizioni 125

4.2.4 Modellazione delle relazioni di generalizzazione 126
4.2.4.1 Determinare le generalizzazioni 127
4.2.4.2 Specificare le generalizzazioni 127
4.2.4.3 Esempio di specifica di generalizzazioni 127

4.2.5 Modellazione degli oggetti 129
4.2.5.1 Specificare gli oggetti 129
4.2.5.2 Esempio di specifica di oggetti 129

4.3 Specifiche del comportamento 130

4.3.1 Modellazione dei casi d'uso 130
4.3.1.1 Determinare i casi d'uso 131
4.3.1.2 Specificare i casi d'uso 132
4.3.1.3 Esempi di specifica di casi d'uso 132

4.3.2 Modellazione delle attività 136
4.3.2.1 Determinare le attività 138
4.3.2.2 Specificare le attività 138
4.3.2.3 Esempio di specifica di attività 138

4.3.3 Modellazione delle interazioni 139
4.3.3.1 Determinare le sequenze di messaggi 140
4.3.3.2 Specificare le sequenze di messaggi 140
4.3.3.3 Esempio di specifica di sequenze di messaggi 140

4.3.4 Modellazione delle interfacce pubbliche 142
4.3.4.1 Determinare le operazioni delle classi 142
4.3.4.2 Specificare le operazioni delle classi 143
4.3.4.3 Esempio di specifica di operazioni delle classi 143

4.4 Specifiche del cambiamento di stato 144

4.4.1 Modellazione dell'evoluzione dello stato degli oggetti 144
4.4.1.1 Determinare gli stati degli oggetti 144
4.4.1.2 Specificare gli stati degli oggetti e la loro evoluzione 145
4.4.1.3 Esempio di specifica di diagrammi statechart 145

Sommario 145

Domande 147

Esercizi 148

5 Approfondimento dell'analisi 151

5.1 La modellazione delle classi 151

5.1.1 Stereotipi 151
5.1.2 Vincoli 152
5.1.3 Note e tag 154
5.1.4 Visibilità e incapsulamento 155
5.1.4.1 Visibilità protetta 156
5.1.4.2 Visibilità delle proprietà ereditate nelle classi 157
5.1.4.3 Classi "amiche" 158
5.1.5 Informazione derivata 160
5.1.5.1 Attributi derivati 161
5.1.5.2 Associazioni derivate 162
5.1.6 Associazioni qualificate 163
5.1.7 Classi d'associazione e classi materializzate 164
5.1.7.1 Esempio di un database del personale 165
5.1.7.2 Modellazione con classi d'associazione 165
5.1.7.3 Modellazione con classi materializzate 166

5.2 Stratificazione delle classi 167

5.2.1 Complessità delle reti 168
5.2.2 Complessità delle gerarchie 169
5.2.3 Package 170
5.2.3.1 Notazione per i package 170
5.2.3.2 Diagramma dei package 171
5.2.4 L'approccio BCE 172

5.3 Generalizzazione ed ereditarietà 173

5.3.1 Generalizzazione e sostituibilità 173
5.3.2 Ereditarietà e incapsulamento 174
5.3.3 Ereditarietà dell'interfaccia 174
5.3.4 Ereditarietà dell'implementazione 175
5.3.4.1 Uso corretto dell'ereditarietà dell'implementazione – ereditarietà per estensione 176
5.3.4.2 Uso problematico dell'ereditarietà dell'implementazione - ereditarietà per restrizione 177
5.3.4.3 Uso improprio dell'ereditarietà dell'implementazione - ereditarietà per convenienza 177
5.3.4.4 I rischi nell'ereditarietà dell'implementazione 178

5.4 Modellazione di aggregazione e delegazione 181

5.4.1 Semantica dell'aggregazione 182
5.4.1.1 L'aggregazione "possiede in modo esclusivo" 182
5.4.1.2 L'aggregazione "possiede" 183
5.4.1.3 L'aggregazione "ha" 183
5.4.1.4 L'aggregazione "raggruppa" 184
5.4.2 Aggregazione come alternativa alla generalizzazione 184
5.4.2.1 Delegazione e prototipi 185
5.4.2.2 Delegazione ed ereditarietà 186
5.4.3 Aggregazione e oloni - alcune considerazioni 187

Sommario 188

Domande 189

Esercizi 190

6 Elementi di base del progetto di sistema 191

6.1 Architettura software 191

6.1.1 Architettura distribuita 192
6.1.2 Architettura a tre strati 193
6.1.3 Programmazione del database 194
6.1.3.1 Interazione applicazione - database 194
6.1.3.2 Approccio BCED 195
6.1.3.3 Sistema software 196
6.1.4 Strategie di riuso 197
6.1.4.1 Riuso di librerie di classi 197
6.1.4.2 Riuso di framework 197
6.1.4.3 Riuso di pattern 198
6.1.5 Componenti 198
6.1.5.1 Notazione per i componenti 199
6.1.5.2 Diagramma dei componenti 199
6.1.5.3 Componenti e package 199
6.1.5.4 Componenti, classi e interfacce 200
6.1.6 Rilascio (deployment) 202
6.1.6.1 Notazione per il nodo 202
6.1.6.2 Diagramma di deployment 202
6.1.6.3 Nodi e componenti 203

6.2 Collaborazioni 204
6.2.1 Notazione per la collaborazione 204
6.2.2 Diagramma di collaborazione 204
6.2.2.1 Notazione per i messaggi 205
6.2.2.2 Struttura del messaggio 206
6.2.2.3 Tipi di messaggio 206
6.2.2.4 Riscrittura e sovraccarico 207
6.2.2.5 Messaggi iterativi e tipi parametrici 208
6.2.2.6 Messaggi "self" 209
6.2.2.7 Messaggi asincroni 209
6.2.2.8 Chiamate di ritorno 210
6.2.3 Diagrammi di sequenza e diagrammi di collaborazione 210
6.2.4 Realizzazione dei casi d'uso 211
6.2.4.1 Aspetti strutturali della collaborazione 211
6.2.4.2 Aspetti comportamentali della collaborazione 212
6.2.5 Realizzazione di operazioni 212
6.3 Progetto guidato di un caso di studio 215
6.3.1 Progetto dei package 216
6.3.1.1 Package di casi d'uso 216
6.3.1.2 Package di classi 217
6.3.2 Progetto dei componenti 219
6.3.2.1 Implementazione di applicazioni web 220
6.3.2.2 Diagramma dei componenti 221
6.3.3 Progetto di rilascio 222
6.3.3.1 Rilascio di applicazioni web 222
6.3.3.2 Diagramma di deployment 223
6.3.4 Progetto delle collaborazioni 224
6.3.4.1 Elaborazione dei casi d'uso 224
6.3.4.2 Struttura delle collaborazioni 229
6.3.4.3 Comportamento delle collaborazioni 231
Sommario 232
Domande 233
Esercizi 234

7 Progetto dell'interfaccia utente 237
7.1 Progetto dell'interfaccia come attività multidisciplinare 237
7.2 Dal prototipo all'implementazione dell'interfaccia 238
7.3 Linee guida per il progetto d'interfaccia centrato sull'utente 239
7.3.1 L'utente ha il controllo 240
7.3.2 Consistenza 241
7.3.3 Personalizzazione e configurazione 242
7.3.4 Tolleranza 242
7.3.5 Retroazione 242
7.3.6 Estetica e usabilità 243
7.4 Finestre d'interfaccia 243
7.4.1 Finestra primaria 243
7.4.1.1 Browser a riga 246
7.4.1.2 Browser ad albero 247
7.4.1.3 Pagina web 248
7.4.2 Finestre secondarie 249
7.4.2.1 Finestra di dialogo 250
7.4.2.2 Cartella a schede 252
7.4.2.3 Lista a tendina 254
7.4.2.4 Finestra di messaggio 254
7.5 Dipendenze tra finestre 255
7.5.1 Documenti e viste 256
7.5.2 Interfaccia a documento singolo 257
7.5.3 Interfaccia a documenti multipli 257
7.6 Navigazione nelle finestre 258
7.6.1 Stereotipare il diagramma d'attività per la navigazione nelle finestre 259
7.6.2 Diagramma di navigazione nelle finestre 261
Sommario 265
Domande 265
Esercizi 266

8 Progetto del database 269

8.1 Il database 269

8.1.1 Modelli dei dati 270

8.1.2 Corrispondenza tra oggetti e database 270

8.2 Modello di database a oggetti 272

8.2.1 Primitive di rappresentazione per ODB 272
8.2.1.1 Tipo letterale e tipo oggetto 272
8.2.1.2 Relazioni e relazioni inverse 274
8.2.1.3 Ereditarietà ISA ed EXTENDS 276
8.2.1.4 Operazioni predefinite 276

8.2.2 Trasformazione UML - ODB 277
8.2.2.1 Trasformazione delle classi entity 277
8.2.2.2 Trasformazione delle associazioni 279
8.2.2.3 Trasformazione delle aggregazioni 279
8.2.2.4 Trasformazione delle generalizzazioni 282

8.3 Modello di database object-relational 282

8.3.1 Primitive di rappresentazione per ORDB 284
8.3.1.1 Tipi strutturati e tipi distinti 284
8.3.1.2 Tabelle di oggetti 285
8.3.1.3 Tipi riga 285
8.3.1.4 Tipi riferimento 286
8.3.1.5 Colonne, campi e attributi 287
8.3.1.6 Ereditarietà OF e UNDER 287

8.3.2 Trasformazione UML - ORDB 289
8.3.2.1 Trasformazione delle classi entity 289
8.3.2.2 Trasformazione delle associazioni 290
8.3.2.3 Trasformazione delle aggregazioni 292
8.3.2.4 Trasformazione delle generalizzazioni 292

8.4 Modello di database relazionale 294

8.4.1 Primitive di rappresentazione per RDB 295
8.4.1.1 Colonne, domini e regole 296
8.4.1.2 Tabelle relazionali 297
8.4.1.3 Integrità referenziale 298
8.4.1.4 Trigger 300
8.4.1.5 Viste relazionali 301
8.4.1.6 Forme normali 302

8.4.2 Trasformazione UML - RDB 303
8.4.2.1 Trasformazione delle classi entity 304
8.4.2.2 Trasformazione delle associazioni 305
8.4.2.3 Trasformazione delle aggregazioni 305
8.4.2.4 Trasformazione delle generalizzazioni 308

Sommario 310

Domande 312

Esercizi 312

9 Progetto di programmi e transazioni 313

9.1 Progettare i programmi 313

9.1.1 Coesione e accoppiamento di classi 314
9.1.1.1 Tipologie di accoppiamento tra classi 315
9.1.1.2 La Legge di Demeter 315
9.1.1.3 Metodi di accesso e dipendenza di classi 316
9.1.1.4 Classificazione dinamica e classi disomogenee 318

9.1.2 Progettare la collaborazione client/server 321
9.1.2.1 Procedure memorizzate 322
9.1.2.2 Trigger 323

9.1.3 Progettare la persistenza 324
9.1.3.1 Collaborazione strutturale per la persistenza 324
9.1.3.2 Collaborazione comportamentale per stabilire la sessione con il database 326

9.2 Navigazione di programma 326

9.2.1 Stereotipare il diagramma d'attività per la navigazione di programma 326

9.2.2 Diagramma per la navigazione di programma 328

9.3 Progettare le transazioni 330

9.3.1 Transazione breve 331
9.3.1.1 Controllo pessimistico della concorrenza 331
9.3.1.2 Livelli di isolamento 331
9.3.1.3 Ripristino automatico 332
9.3.1.4 Ripristino programmabile 333
9.3.1.5 Progettare procedure memorizzate e trigger 334

9.3.2 Transazione lunga 335

9.4 Round-trip engineering 336

9.4.1 Round-trip engineering per i programmi client 336

9.4.2 Round-trip engineering nei database 337

9.4.3 Reingegnerizzare un database relazionale in un object-relational 338

Sommario 341

Domande 341

Esercizi 342

10 Testing e gestione del cambiamento 343

10.1 Testing dei servizi del sistema 343

10.1.1 Incontri di rassegna 344
10.1.2 Ispezioni 344
10.1.3 Testing delle specifiche 345
10.1.4 Testing del codice 345

10.2 Testing dei vincoli del sistema 346

10.2.1 Testing dell'interfaccia utente 346
10.2.2 Testing del database 347
10.2.3 Testing delle autorizzazioni 348
10.2.4 Testing di altri vincoli 348

10.3 Documentazione di test e di gestione del cambiamento 350

10.4 Gestire il cambiamento 351

10.4.1 Sottoporre una richiesta di cambiamento 352
10.4.2 Tenere traccia delle richieste di cambiamento 353

10.5 Tracciabilità 354

10.5.1 Dalle caratteristiche del sistema ai casi d'uso e ai requisiti dei casi d'uso 355
10.5.2 Dai piani di test ai casi di test e ai requisiti di test 356
10.5.3 Dai diagrammi UML ai documenti e ai requisiti 357
10.5.4 Dai requisiti dei casi d'uso ai requisiti di test 358
10.5.5 Dai requisiti di test ai difetti 358
10.5.6 Dai requisiti dei casi d'uso alle estensioni 360

Sommario 362

Domande 362

Bibliografia 363

Indice analitico 369

Presentazione dell'edizione italiana

Il testo del prof. Leszek Maciaszek illustra le fasi di analisi e progetto dei sistemi informativi, affrontando sia gli aspetti metodologici sia quelli modellistici. Gli aspetti legati all'implementazione del sistema vengono considerati solo per i riflessi che possono avere sul progetto del sistema stesso.

Il paradigma object-oriented adottato utilizza UML (Unified Modeling Language) come strumento unificante attraverso l'intero ciclo di vita del progetto, fino a illustrare i punti fondamentali della fase di implementazione. Questa impostazione è particolarmente adatta a illuminare la complessità dello sviluppo dei moderni sistemi informativi, basati su architetture a tre livelli in cui l'interfaccia grafica con l'utente gioca un ruolo essenziale. L'approfondita discussione sulle architetture di sistema introduce un realismo nella modellazione che è difficile trovare in testi analoghi.

Il valore didattico del testo è accresciuto dalla chiarezza espositiva e dall'utilizzo sistematico di una serie di esempi ben selezionati e presentati; questo aspetto è stato particolarmente apprezzato dagli studenti dei corsi sui Sistemi Informativi che, nell'ambito delle lauree e dei diplomi in Informatica presso l'Università degli Studi di Torino, hanno affrontato le problematiche legate al progetto di tali sistemi.

Il testo colma quindi una lacuna nella letteratura disponibile e risulterà certamente utile, oltre che agli studenti, ai professionisti del settore che intendono approfondire le proprie conoscenze in un'area dell'informatica essenziale per la gestione della complessità del mondo reale.

Antonio Di Leva
Dipartimento di Informatica
Università degli Studi di Torino

Torino, aprile 2002

Prefazione

Argomento del testo

Lo sviluppo di un sistema informativo, dalle origini del progetto fino al primo rilascio agli utenti, comprende tre fasi iterative e incrementali: analisi, progetto e implementazione. Il testo descrive i metodi e le tecniche usati nelle prime due fasi. I problemi d'implementazione sono trattati solo per quel che serve alla fase di progetto. Il testing e la gestione del cambiamento sono trattati nell'ultimo capitolo.

Il testo è orientato allo sviluppo object-oriented del software. L'Unified Modeling Language (UML) è usato per specificare gli elaborati della modellazione. L'enfasi è posta sullo sviluppo iterativo e incrementale, usando il medesimo linguaggio di modellazione (UML) in tutte le fasi del ciclo di vita del software. Analisti, progettisti e programmatori adottano sostanzialmente lo stesso linguaggio, sebbene possano usarne dialetti (profili) che meglio si adattano alle specifiche necessità.

Le prime applicazioni della tecnologia a oggetti miravano essenzialmente all'interfaccia grafica utente (GUI), concentrandosi sulla rapidità dello sviluppo di nuovi sistemi e la velocità d'esecuzione dei programmi. In questo testo, si enfatizza l'applicazione della tecnologia a oggetti allo sviluppo dei sistemi informativi. La sfida consiste nella gestione di un grande volume di dati, nella complessità delle strutture dati, nell'accesso condiviso all'informazione da parte di molti utenti, nell'elaborazione transazionale, nella modificabilità dei requisiti, etc. Il principale vantaggio della tecnologia a oggetti, nel contesto dei sistemi informativi, è nella semplificazione dei problemi legati alla manutenzione e alla scalabilità del software.

Lo sviluppo di sistemi informativi è sinonimo di analisi e progetto "in grande". Per questo un progetto IS non può aver successo senza il rispetto di rigidi processi di sviluppo e senza la comprensione delle sottostanti architetture software. Il processo di sviluppo è di tipo object-oriented, iterativo e incrementale. L'architettura software è basata su una soluzione client/server, ove il client è una stazione di lavoro, con la gestione di un'interfaccia grafica, e il server contiene un database. Il client e il server sono eseguiti da processi distinti e comunicano mediante scambio di messaggi. Il database può essere relazionale, object-relational, o object-oriented puro.

Il testo, usando UML, propone un approccio dettagliato all'analisi e al progetto di sistemi informativi, e discute i modi per:

1. dominare la complessità di grandi sistemi;
2. migliorare le architetture software;
3. facilitare la leggibilità, la manutenibilità e la scalabilità del software;
4. promuovere la strutturazione stratificata degli oggetti;
5. gestire l'integrazione dei componenti;
6. migliorare i modelli di collaborazione tra interfaccia grafica e oggetti persistenti del database, etc.

Caratteristiche distintive del testo

Questo testo ha diverse caratteristiche che lo contraddistinguono nella letteratura esistente. L'approccio di base è "l'insegnamento attraverso gli esempi". Le discussioni più importanti sono basate su casi di studio e su un esempio di sviluppo guidato, relativi a cinque domini, *Immatricolazioni universitarie*, *Video Store*, *Gestione del Contatto con i Clienti*, *Telemarketing* e *Acquisti OnLine*. L'esempio e i casi di studio sono tra loro indipendenti e possono essere estesi o variati, svolgendo gli esercizi proposti al termine di buona parte dei capitoli. Alcuni esercizi fanno riferimento a un sesto dominio applicativo, la *Valutazione delle Spese Pubblicitarie*.

Per facilitare l'autoapprendimento, lo sviluppo guidato (Acquisti OnLine) e i casi di studio sono formulati e discussi procedendo per "domanda-e-risposta". Una sezione a parte, all'inizio del testo, intitolata "Diagrammi d'attività del testo", presenta alcuni diagrammi che legano i passi, organizzati a "domanda-e-risposta", usati nello sviluppo guidato e nei casi di studio. Tali diagrammi d'attività possono anche servire come indice alternativo per gli esempi sparsi nel testo.

Il testo tratta dei principi, dei metodi e delle tecniche per una buona analisi e un buon progetto. In particolare, il progetto è trattato con attenzione, non come una semplice trasformazione delle specifiche dell'analisi. Il testo riconosce le difficoltà legate allo sviluppo object-oriented di sistemi client/server, cercando comunque di suggerire nuovi punti di vista per affrontare il "progetto in grande", lo sviluppo iterativo e incrementale di grandi sistemi, e le limitazioni di strumenti e metodi nella produzione di software di grandi dimensioni.

La scelta degli argomenti è basata sull'esperienza ed è caratterizzata dal bilanciamento tra spiegazione pratica e aspetti teorici. Argomenti non rilevanti nel settore industriale o di esclusivo interesse della ricerca sono stati esclusi. Soprattutto si sono volute evitare complicazioni non necessarie pur senza perdere di rigore.

Il testo è aggiornato ai più recenti sviluppi: usa UML (l'ultimo standard nella modellazione di sistemi), tratta gli ultimi modelli nella tecnologia dei database (inclusi i database object-relational), e tiene conto dello spostamento (guidato da Internet) verso client "leggeri" associati a server con database. I principi di analisi e progetto discussi nel testo si applicano egualmente bene alle convenzionali soluzioni client/server e alle moderne applicazioni distribuite a componenti.

Lo sviluppo del software non è riconducibile a soluzioni nette: buone soluzioni software provengono da analisti e progettisti/programmatori validi, non da algoritmi applicati ciecamente. Il testo intende *avvertire* il lettore sulle *potenziali* difficoltà che possono non essere risolte completamente dall'approccio presentato: di conseguenza, si auspica che i lettori applichino la conoscenza acquisita con attenzione, senza attese eccessive sulla facilità di applicazione dell'approccio stesso.

In sintesi, le caratteristiche distintive del testo sono:

1. Il testo mette in relazione le teorie con la pratica, sotto forma di problemi reali e limitazioni che devono essere considerati nell'applicazione dell'approccio "sul campo".
2. Il testo riserva una particolare attenzione alla fase di progetto. Il progetto *non* è trattato come una trasformazione diretta dei risultati dell'analisi, riconoscendo le difficoltà dello sviluppo client/server su larga scala.
3. Il testo e il materiale supplementare sono ricchi di esempi ed esercizi non banali, corredati dalle rispettive soluzioni. Il manuale dell'insegnante non è un'aggiunta, ma è stato scritto contemporaneamente al testo ed è meticoloso.

A chi si rivolge

In accordo alla crescente domanda di corsi universitari che tengano conto delle esigenze industriali, questo testo è orientato sia a studenti sia a professionisti del settore. È stato un compito difficile che, si spera, sia stato portato a termine con successo. Per assicurare un *beneficio educativo durevole*, gli aspetti implementativi dello sviluppo del software non sono stati discussi in termini di specifici prodotti commerciali (sebbene siano stati presentati strumenti CASE commerciali nelle illustrazioni e nelle soluzioni).

Il testo è orientato a corsi di studi di informatica e di sistemi informativi. Poiché comprende sia argomenti ad "alto livello" di modellazione di sistemi sia problemi di "basso livello" quale il progetto di interfacce utente e database, potrebbe essere d'interesse per corsi di *analisi di sistemi*, *ingegneria del software*, *basi di dati*, *tecnologia a oggetti*, *progetto di sviluppo del software*, che richiedano agli studenti di sviluppare progetti di sistemi sull'intero ciclo di vita: dalla determinazione dei requisiti all'implementazione dell'interfaccia e del database. Il testo è pensato per un corso di un semestre ma potrebbe essere usato per due corsi semestrali, uno sull'analisi dei requisiti e l'altro sul progetto dei sistemi.

Per i professionisti, le teorie presentate sono in relazione diretta con la realtà: la maggior parte dei casi di studio, degli esempi e degli esercizi derivano dall'esperienza di consulenza dell'autore, e si è adottata una politica che non nasconde al lettore le difficoltà potenziali o le limitazioni degli approcci proposti. Le seguenti tipologie di professionisti probabilmente possono trarre il maggior beneficio dal testo: *analisti di business e di sistema*, *programmatori*, *architetti di sistema*, *capi progetto*, *revisori*, *esecutori di test*, *redattori di documentazione tecnica* e *formatori nel settore industriale*.

Organizzazione del testo

Il volume è composto da dieci capitoli che coprono completamente l'analisi e il progetto object-oriented dei sistemi informativi. Gli argomenti sono presentati in un ordine coerente con i moderni processi di sviluppo. I primi cinque capitoli trattano i problemi dell'analisi, gli ultimi cinque trattano il progetto e gli aspetti correlati.

Lettori con conoscenze eterogenee dovrebbero comunque potersi ritrovare. Due capitoli sono infatti dedicati alla spiegazione degli elementi di base dell'analisi e del progetto. I rimanenti capitoli danno per scontato che il lettore abbia acquisito tali elementi di base. Il lettore può scegliere di studiare i capitoli sugli elementi di base, approfonditamente oppure come un semplice ripasso.

Il testo ha molte caratteristiche tendenti a migliorare la chiarezza dell'esposizione e interromperne la monotonia, tra cui:

- Le sezioni sono brevi.
- Gli esempi e le descrizioni dei casi di studio sono stati evidenziati ponendoli in parti distinte dal testo normale.
- Un'icona grafica (si veda qui di seguito) stabilisce l'inizio di ogni esempio e identifica le domande e gli esercizi al termine di ogni capitolo.

icona

Esempi

Domande

Esercizi

Materiale supplementare

Nei siti web associati al testo è possibile trovare un insieme completo di materiale aggiuntivo. La maggior parte dei documenti su web è liberamente disponibile per i lettori ma alcune sezioni, protette con parola chiave, sono a beneficio dei docenti che hanno adottato questo testo come supporto ai loro corsi. La pagina iniziale del testo è rintracciabile a partire dai due indirizzi seguenti:

http://www.booksites.net/maciaszek

http://www.comp.mq.edu.au/books/maciaszek

L'insieme di materiali su web, sotto forma di pagine o file, comprende:

1. **Manuale dell'insegnante** con:
   - *Slide per le lezioni* in formato Microsoft PowerPoint e Acrobat Reader (.pdf).
   - Manuale delle *Risposte e soluzioni* contenente le risposte e le soluzioni commentate per tutte le domande ed esercizi al termine di ogni capitolo. L'organizzazione del manuale corrisponde alla struttura del testo, e riporta anche le domande cui fanno seguito risposte e soluzioni.
2. **Risorse per lo studente** con slide delle lezioni in formato Acrobat Reader.
3. **Risorse per l'autoapprendimento** con i file dei modelli per Rational Rose (.mdl) e PowerDesigner (.pdm), contenenti le soluzioni all'esempio di sviluppo guidato, ai casi di studio e a tutti gli altri esempi di modellazione presentati nel testo.
4. **Errata-corrige** dedicata alla correzione di errori e omissioni del testo originale inglese.
5. **Pagina di approfondimento** per orientare il lettore sulle ultime idee e tendenze nelle aree trattate dal testo, e per elencare i corsi che lo hanno adottato e che mettono a disposizione materiale condivisibile, attraverso il web, con altri lettori.

I vostri commenti, correzioni, suggerimenti per miglioramenti, contributi, etc. sono molto apprezzati e possono essere direttamente inviati all'autore (in inglese), usando i seguenti riferimenti:

Leszek A. Maciaszek
Department of Computing
Macquarie University
Sydney
NSW 2109
Australia
leszek@ics.mq.edu.au
http://www.comp.mq.edu.au/~leszek/
telefono:+61 2 9850-9519
fax: +61 2 9850-9551
indirizzo postale: North Ryde, Herring Road, Bld. E6A, Room 319

Ringraziamenti

Lo sviluppo di questo testo non sarebbe stato possibile senza l'interazione dell'autore con amici, colleghi, studenti, esperti e tutti coloro che, volutamente o meno, hanno formato la conoscenza dell'autore in quest'area. Sono veramente in debito con tutti loro. Un tentativo di elencare tutti i loro nomi sarebbe impraticabile e chiedo loro di accettare semplicemente il mio "grazie". Tuttavia, anche nel breve spazio disponibile per i ringraziamenti, alcuni devono essere menzionati:

- Keith Mansfield, Mary Lince e lo staff di Pearson Education - per aver riconosciuto il potenziale di questo testo, portandolo nel mercato internazionale.
- Stephen Bills, e il suo staff, di ACNielsen AdEx, Sydney, Australia, specialmente: Steven Grotte, Kevin Mathie, Cameron Murray, James Rees, Jovan Spoa e Eric Zurcher - per avermi concesso una piattaforma di test con un esempio sviluppato completamente per il testo.
- I miei amici e colleghi del Department of Computing alla Macquarie University, Sydney, Australia - per il loro supporto durante la redazione di questo testo.
- I revisori formali (che mi sono sconosciuti) e informali - per i loro inestimabili suggerimenti di cui il testo ha grandemente beneficiato.
- Gli staff di Rational, Oracle e Sybase - per avermi fornito il software database e CASE indispensabile per lo sviluppo dei casi di studio e degli esempi.

La casa editrice ringrazia i seguenti soggetti per i permessi di riproduzione accordatele:

La Figura 2.34 è riprodotta con il permesso di Gateway, Inc. © Gateway, Inc.; le Figure 7.1, 7.2, 7.4, 7.6, 7.7, 7.8, 7.9, 7.11, 7.12, 7.13, 7.14, 7.15, 7.16, 7.17, 7.19, 7.20, 10.6, 10.7, 10.8, 10.9, 10.10, 10.11 e 10.12 sono riprodotte con il permesso di ACNielsen; la Figura 7.10 è riprodotta con il permesso della Macquarie University.

Un sito web associato accompagna il testo ***Sviluppo di sistemi informativi con UML*** di Leszek A. Maciaszek.

Si consiglia di visitare il sito web associato al testo ***Sviluppo di sistemi informativi con UML*** all'indirizzo:

www.booksites.net/maciaszek

in cui è possibile trovare materiale interessante sia per l'insegnamento sia per lo studio.

Per gli insegnanti:

- Slide delle lezioni in formato PowerPoint o Acrobat.
- Manuale delle soluzioni contenente risposte annotate e soluzioni a tutte le domande ed esercizi al termine di ogni capitolo.

Per gli insegnanti e gli studenti:

- File di modelli in Rational Rose e PowerDesigner contenenti le soluzioni allo sviluppo guidato, ai casi di studio e a tutti gli altri esempi di modellazione nel testo.
- Pagina degli aggiornamenti con i legami alle ultime tendenze e ad altri lavori nell'area d'interesse del testo.

Diagrammi d'attività del testo

L'analisi dei requisiti e il progetto di sistema riguardano soprattutto conoscenze pratiche. Come tali, non possono essere apprese solo leggendo libri. L'unico modo efficace per apprenderle è facendo pratica, commettendo errori, analizzandoli e provando a evitarli nel futuro. Questo testo è stato scritto con questa idea di fondo.

Il testo è ricco di esempi, includendo un esempio di sviluppo guidato e quattro casi di studio. Tuttavia, gli esempi sono stati formulati per permettere al lettore l'autoapprendimento e la sperimentazione, di commettere errori o di arrivare a soluzioni alternative. Il lettore è sempre invitato a tentare una propria soluzione prima di guardare la soluzione già fornita. Dopo tutto, "Un buon giudizio viene dall'esperienza. L'esperienza viene da un giudizio negativo" (Jim Horning).

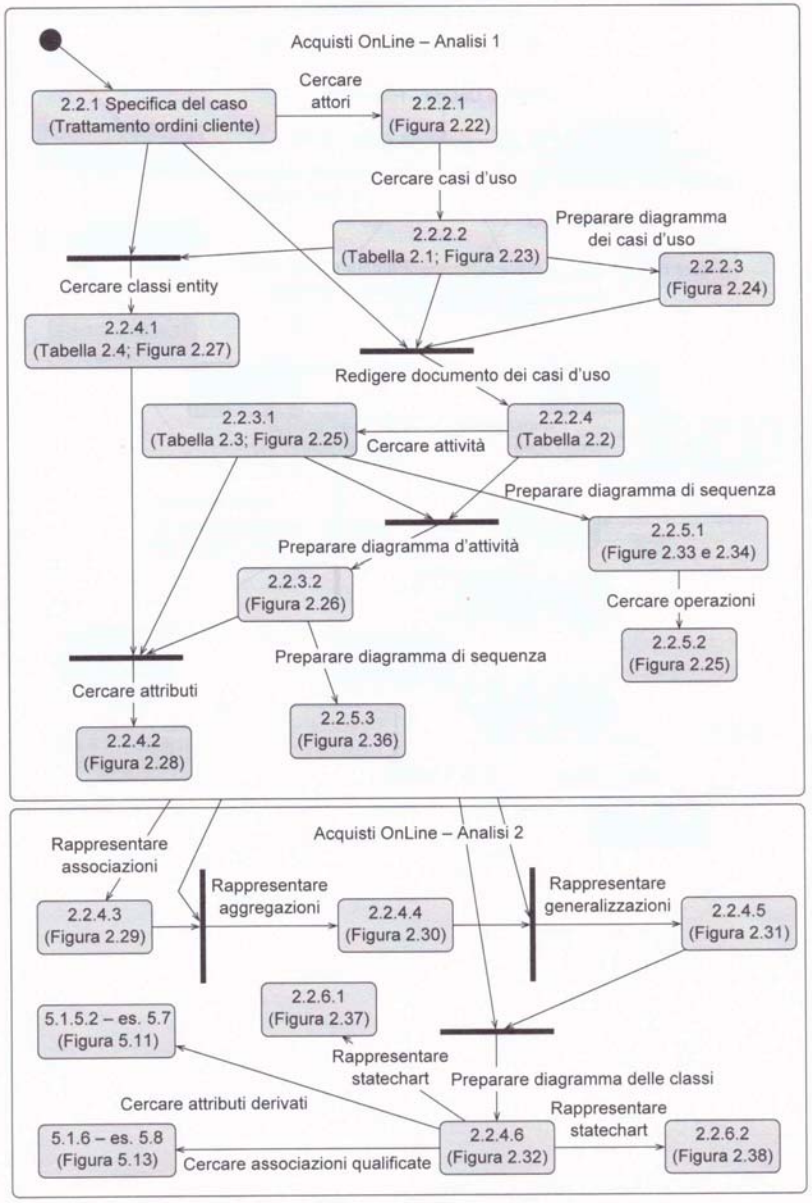
I diagrammi d'attività del testo forniscono le mappe che collegano tabelle e figure relative a ciascun esempio. Una tecnica di modellazione formale (Diagrammi di attività UML) è stata usata nella costruzione di questi diagrammi, ma nessuna conoscenza tecnica è necessaria per poter comprendere e usare questi diagrammi. Ogni *rettangolo arrotondato* (uno stato UML) si riferisce a una sezione del testo che contiene un esempio. Le *frecce* che (direttamente o indirettamente) puntano verso un rettangolo, provengono da *esempi prerequisito*. Le etichette su tali frecce definiscono le attività che dovrebbero essere state portate a termine prima di tentare di comprendere l'esempio.

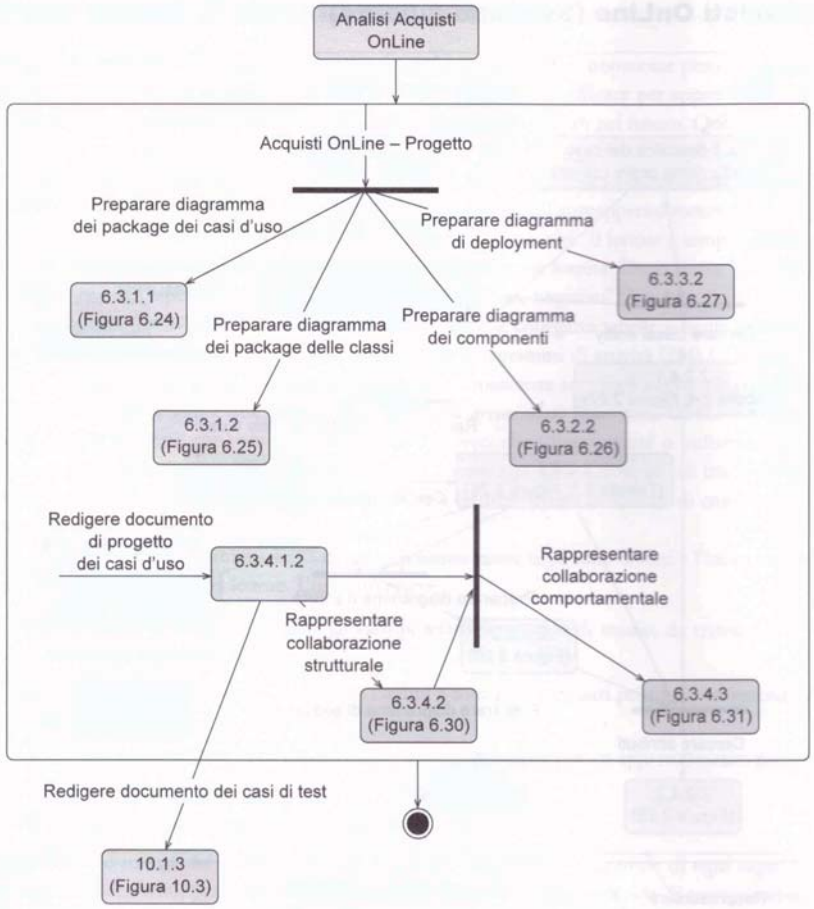
I modi in cui i diagrammi d'attività del testo possono essere usati sono diversi e l'unica limitazione è l'immaginazione del lettore. L'uso può essere:

- per comporre in un'unità coerente gli esempi relativi a un caso di studio, da trattarsi in un corso, in una lezione etc.;
- per localizzare rapidamente i precedenti esempi e soluzioni necessari prima di intraprendere l'analisi dei successivi;
- per scegliere alcune attività di modellazione e ignorarne altre, per un apprendimento mirato e specializzato;
- per sviluppare progetti da assegnare agli studenti, etc.

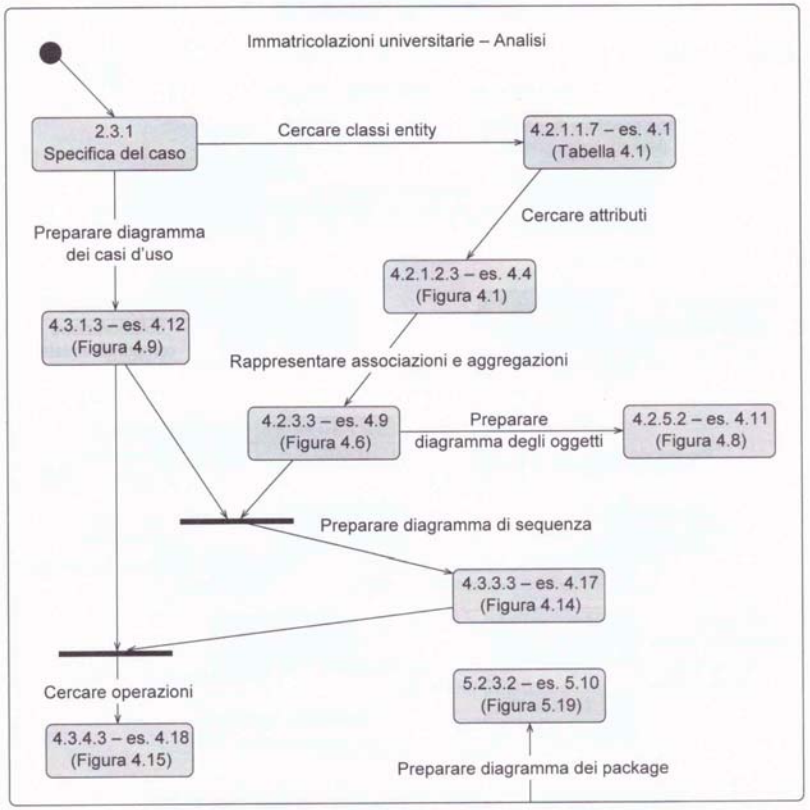
I diagramma d'attività del testo sono integrati con gli esercizi che, al termine di ogni capitolo, estendono le idee sviluppate negli esempi contenuti nel testo. Il miglior modo di trovare esempi cui un esercizio si riferisce, è di prendere visione del diagramma d'attività interessato.

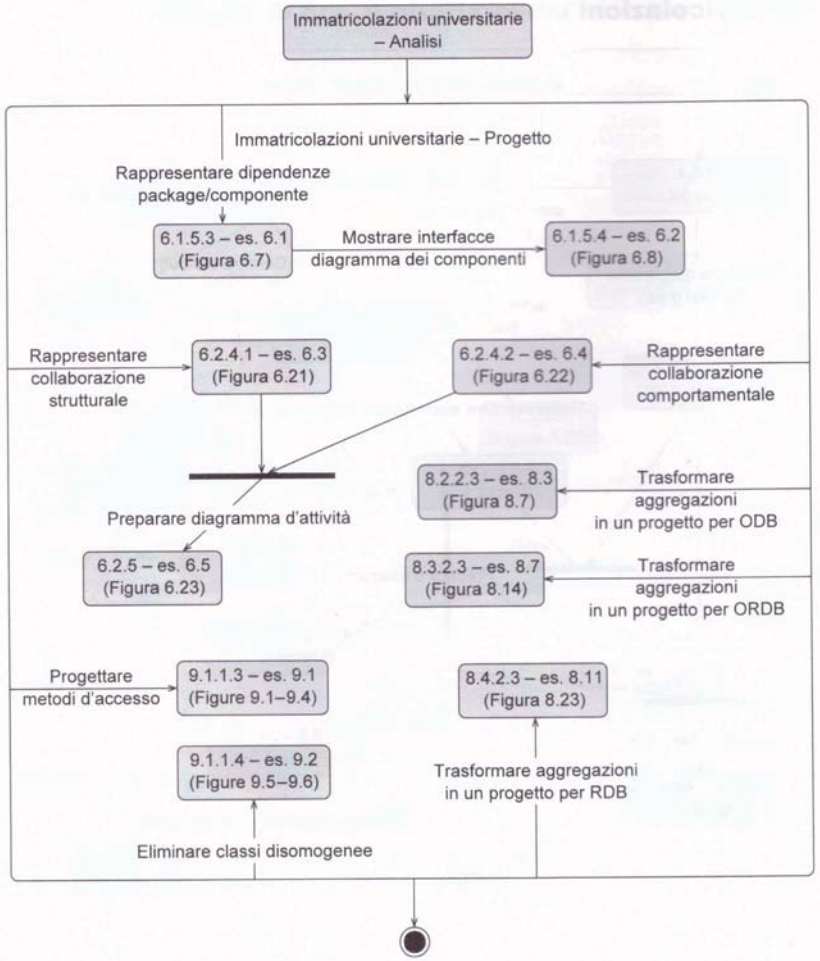
Acquisti OnLine (Sviluppo guidato)



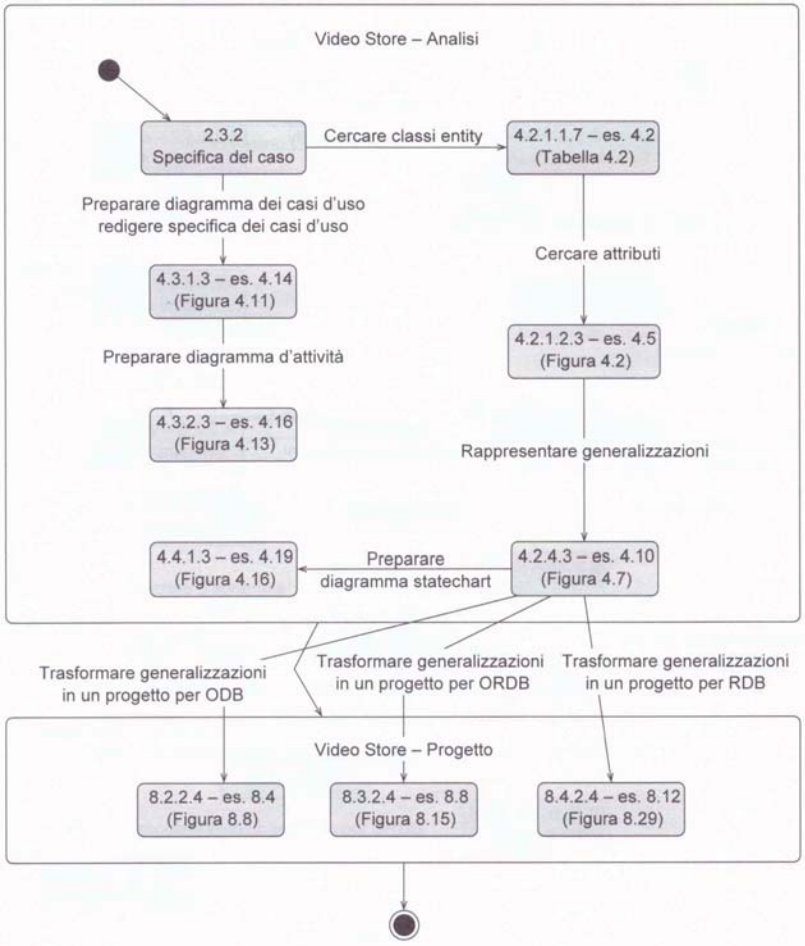


Immatricolazioni universitarie (Caso di studio)

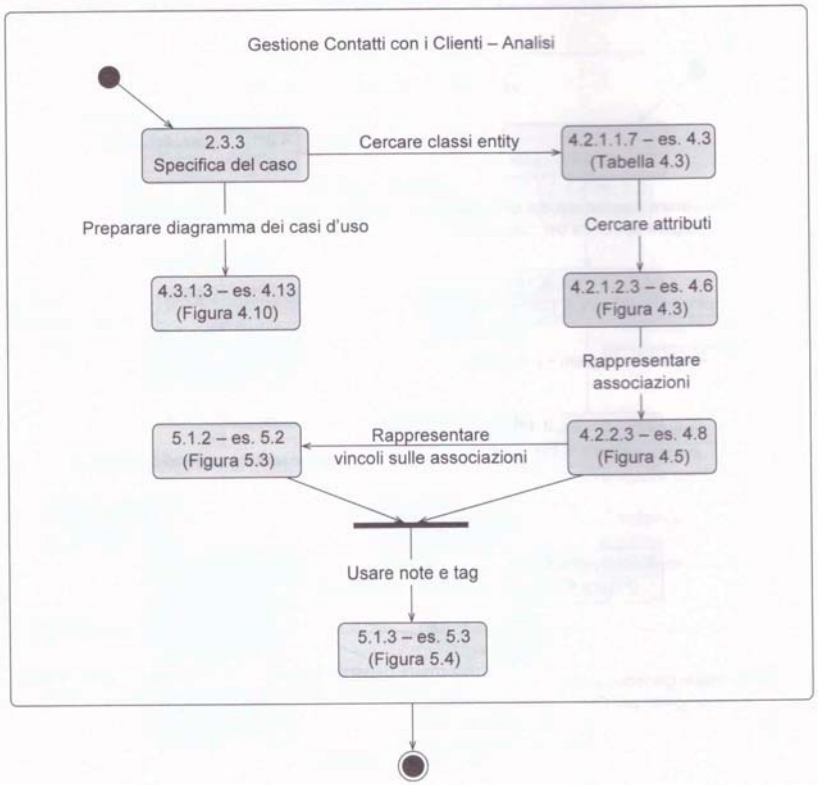


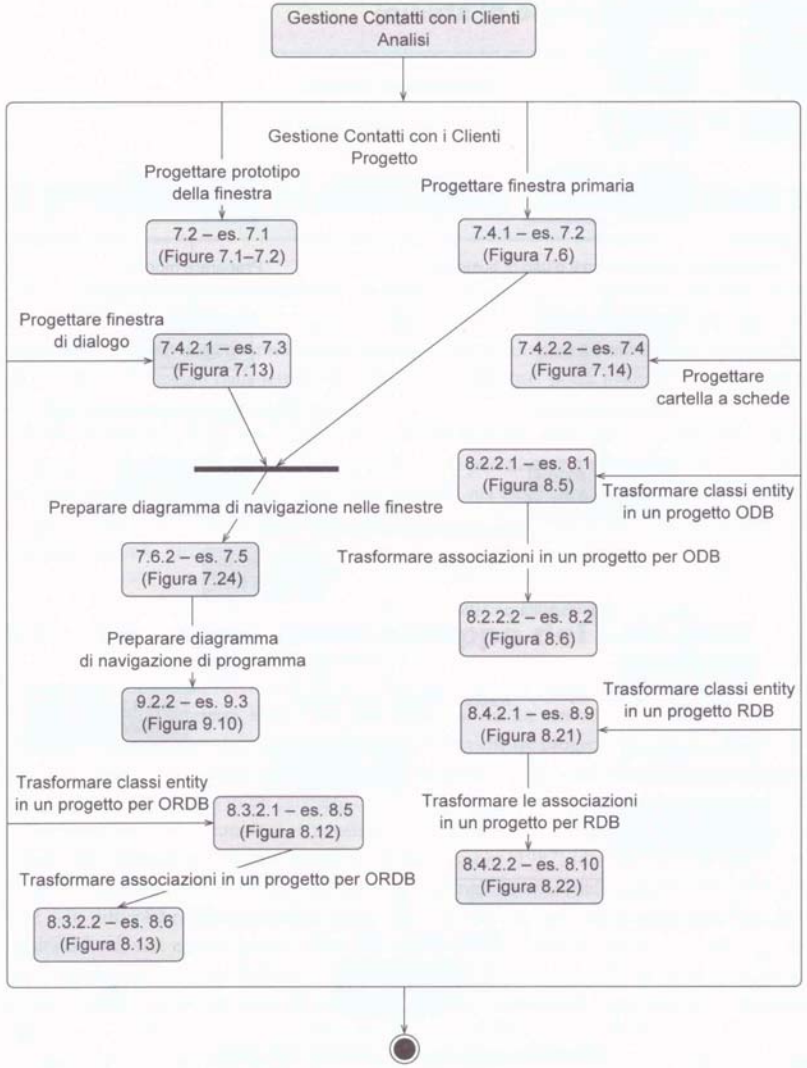


Video Store (Caso di studio)

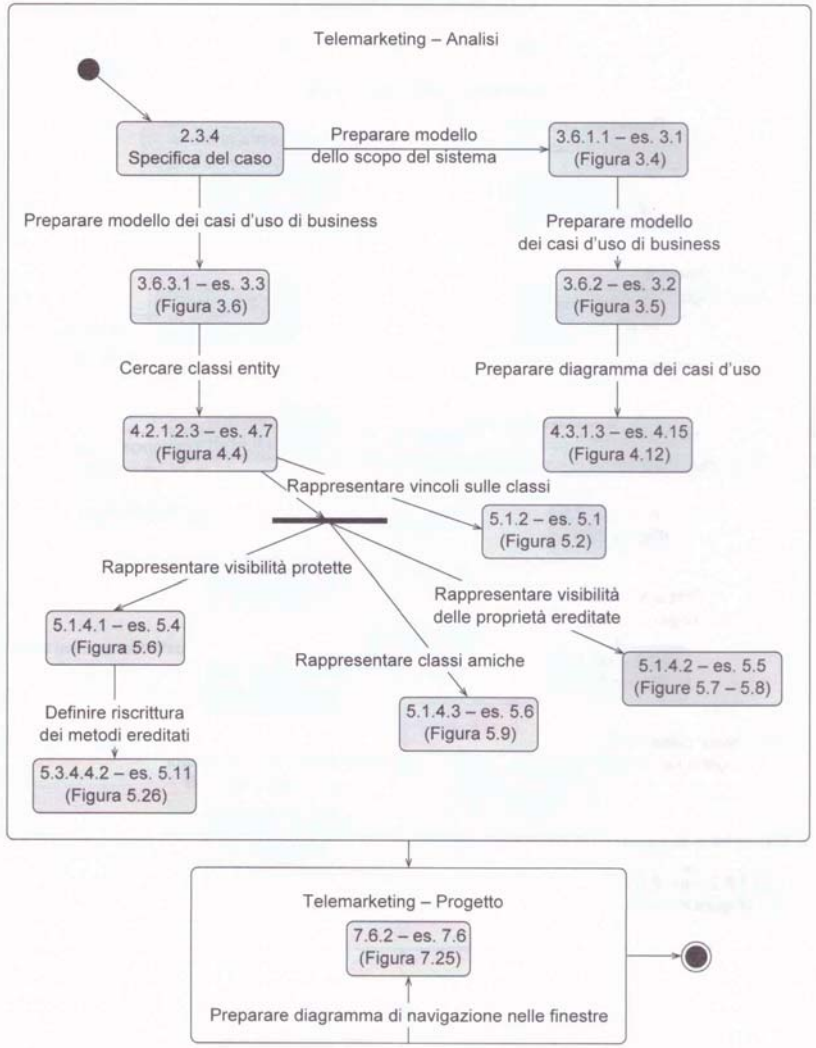


Gestione Contatti con i Clienti (CMS) (Caso di studio)





Telemarketing (Caso di studio)



Capitolo 1

Il processo di sviluppo del software

Lo scopo di questo capitolo è delineare alcuni elementi strategici nel processo di sviluppo del software. Tuttavia la presentazione è solo introduttiva e alcuni di questi elementi non sono unanimemente accettati: il lettore non deve necessariamente condividere l'impostazione data dall'autore per trarre vantaggio dalla lettura del libro (eventualmente, potrà rivalutare le sue opinioni una volta terminata la lettura stessa).

Il valore didattico di questo capitolo è di introdurre il lettore ai processi e agli approcci che compongono l'attuale sviluppo del software. Il lettore può avere già familiarità con molte delle idee e dei temi discussi in questo capitolo, in base alla sua esperienza, all'utilizzo quotidiano del computer o alla consultazione della letteratura in materia. Tale lettore può scegliere di iniziare la lettura dal capitolo successivo.

1.1 La natura dello sviluppo del software

La letteratura sui sistemi informativi è ricca d'esempi relativi a progetti falliti, tempi e costi non rispettati, soluzioni scorrette, sistemi non gestibili e così via. La versione 1998 del rapporto di ricerca CHAOS, preparato dallo Standish Group, afferma che tre progetti su quattro falliscono per una o più delle suddette ragioni. Le domande sono quindi: quali sono le cause e quali i sintomi della presenza di problemi in un progetto e come devono essere trattati?

Per rispondere a tali questioni, è necessario comprendere anzitutto la natura del processo di sviluppo del software. In un articolo ormai classico, Brooks (1987) identificava l'essenza e gli incidenti nell'ingegneria del software (software engineering). L'*essenza* dell'ingegneria del software è parte delle difficoltà relative al software stesso. Queste difficoltà sono una conseguenza della complessità del software e possono solo essere riconosciute. Esse conducono a un principio sullo sviluppo del software, il quale stabilisce semplicemente che il software è il prodotto di un atto creativo di sviluppo, un'arte, nel senso di un'attività svolta da un artigiano piuttosto che da un artista. Tipicamente, il software non è il risultato di un'attività manifatturiera ripetitiva.

Una volta compreso il principio, dovrebbero essere analizzati i problemi che ne derivano per l'ingegneria del software; in altre parole, le difficoltà dovute alla pratica nella produzione di software e riconducibili all'intervento umano. Tali difficoltà possono essere raggruppate in tre categorie:

1. Stakeholder (persone interessate al progetto).
2. Processo.
3. Linguaggio e strumenti di modellazione.

1.1.1 Il principio di sviluppo del software

Il software è *sviluppato* piuttosto che *fabbricato* (Pressman 1997). Ovviamente, non si può negare che i progressi dell'ingegneria del software abbiano portato maggiore certezza nella pratica di produzione ma (diversamente che nell'ingegneria tradizionale) il successo di un progetto di sviluppo non può essere garantito.

Algoritmi, librerie di codice, classi riusabili, componenti software etc., costituiscono soluzioni incomplete alle necessità che sorgono durante lo sviluppo di sistemi informativi. La sfida è nel mettere insieme piccole parti in un *sistema aziendale* coerente, un sistema in grado di soddisfare tutti i requisiti di complessi processi di business.

La pratica del software incoraggia lo sviluppo di sistemi a partire da package (pacchetti) adattabili, come le soluzioni COTS (*Commercial of the Shelf*, componenti disponibili in commercio) oppure i sistemi ERP (*Enterprise Resource Planning*, pianificazione delle risorse aziendali). Un package può fornire una procedura per la contabilità, per la manifattura o per gestire un sistema di risorse umane. L'enfasi è quindi spostata dalla costruzione del software all'adattamento di software preesistente, ma il processo di produzione è ugualmente faticoso.

Costrutti concettuali (modelli) devono essere creati per ogni sistema sviluppato da zero in modo che possano soddisfare le necessità proprie di un'organizzazione. D'altra parte, le funzionalità di un qualunque package devono essere adattate alle necessità. In entrambi i casi, l'attività di programmazione può essere diversa ma l'analisi dei requisiti e il progetto del sistema sono simili. Dopo tutto, un modello rimane lo stesso anche se le sue rappresentazioni (implementazioni) sono diverse.

Inoltre, è improbabile che un'organizzazione possa trovare sul mercato un package per automatizzare le sue attività fondamentali (di *core business*). Un'attività di core business di una compagnia telefonica è la gestione delle telefonate, non la gestione delle risorse umane o della contabilità. Ciò che rende un'organizzazione funzionante (e competitiva) deve essere sviluppato da zero (o nuovamente sviluppato a partire da un *sistema ereditato* preesistente). Come notato da Szyperski (1998): "Un package standard crea un ambiente comune, la competizione deve venire dalle altre aree".

In ogni caso, il processo di sviluppo può trarre vantaggio dall'uso della *tecnologia a componenti* (Szyperski, 1998; Allen e Frost, 1998). Un componente è un'unità di software eseguibile con funzionalità ben definite (servizi) e protocolli di comunicazione (interfacce) verso altri componenti. I componenti devono poi essere configurati al fine di soddisfare i requisiti delle applicazioni. Attualmente le tecnologie a componenti con maggiore influenza sono:

- *Common Object Request Broker Architecture* (CORBA) di Object Management Group (OMG).
- *Distributed Component Object Model* (DCOM) di Microsoft.
- *Enterprise JavaBeans* (EJB) di Sun.

Package, componenti e tecniche simili non modificano l'essenza della produzione del software. In particolare, i principi e le attività relativi all'analisi dei requisiti e al progetto del sistema rimangono invariati. Un prodotto software finito può essere assemblato usando componenti standard opportunamente adattati, ma il processo di assemblaggio è ancora un'arte. Francamente, come osserva Pressman (1987), non abbiamo ancora "ricambi" software per sostituire "in corso d'opera" i componenti non funzionanti.

1.1.2 Stakeholder

Gli *stakeholder* sono persone che hanno un ruolo in un progetto software. Qualunque persona influenzata dal sistema o che ha influenza sullo sviluppo del sistema è uno stakeholder. Ci sono due principali gruppi di stakeholder:

- Clienti (utilizzatori o proprietari del sistema);
- Sviluppatori (analisti, progettisti, programmatori, etc.).

Sarebbe preferibile usare (in una prospettiva di sviluppo del sistema) il termine *cliente* piuttosto che *utente*. Anzitutto, il cliente è la persona che "paga" per lo sviluppo ed è responsabile delle decisioni. Inoltre, sebbene il cliente non sia sempre coerente, i suoi requisiti non possono essere arbitrariamente modificati o rifiutati dagli sviluppatori; requisiti in conflitto, non realizzabili oppure illegali devono essere rinegoziati con i clienti.

Ciò detto, bisogna riconoscere che il termine *utente* (per indicare il *cliente*) è più diffuso, per cui continuerà a essere utilizzato nel testo. Sarà limitato però l'uso del termine *utente finale* (*end-user*) il quale, storicamente, si riferisce a un *rappresentante dei clienti* scelto per interagire con gli sviluppatori (al posto di *utente reale*).

I sistemi informativi sono *sistemi sociali*. Sono sviluppati da persone (sviluppatori) per delle persone (clienti). Il successo di un progetto software è determinato da fattori sociali, la tecnologia è secondaria. Esistono molti esempi di sistemi tecnicamente poveri che funzionano e sono utili per i clienti, mentre l'inverso non è vero. Un sistema che non avvantaggia il cliente sarà abbandonato, non importa quanto tecnicamente valido sia.

In genere, le principali cause di fallimento del software possono essere ricondotte a fattori legati ai clienti; da questo punto di vista, i progetti falliscono perché (vedi Pfleeger, 1998):

- le necessità dei clienti sono mal comprese oppure comprese insufficientemente;
- i requisiti del cliente cambiano troppo frequentemente;
- i clienti non forniscono sufficienti risorse ai progetti;
- i clienti non vogliono cooperare con gli sviluppatori;
- i clienti hanno attese non realistiche;
- il sistema non porta più benefici ai clienti.

I progetti falliscono anche perché gli *sviluppatori* possono non essere all'altezza delle attività in cui sono coinvolti. Con l'incremento della complessità del software, si sta sempre più riconoscendo che capacità e conoscenza degli sviluppatori sono fattori critici. Buoni sviluppatori possono fornire una soluzione, ma ottimi sviluppatori possono fornire soluzioni migliori, più velocemente e a minor costo. Come asserito in (Brooks, 1987): "Ottimi progetti sono dovuti a ottimi sviluppatori".

L'eccellenza e l'impegno degli sviluppatori sono i fattori che maggiormente contribuiscono alla qualità del software e alla produttività. Per essere sicuri che un prodotto software sia un successo, fornendo un valido contributo al cliente, un'organizzazione per lo sviluppo del software dovrebbe seguire alcune ovvie indicazioni (Brooks, 1987; Yourdon, 1994):

- assumere gli sviluppatori migliori;
- fornire agli sviluppatori una formazione continua;
- incoraggiare lo scambio di conoscenze tra gli sviluppatori al fine di ottenere uno stimolo e un sostegno reciproco;
- motivare gli sviluppatori rimuovendo gli ostacoli e incanalando i loro sforzi in attività produttive;
- offrire un ambiente di lavoro stimolante (per le persone ciò può essere molto più importante rispetto ad aumenti di stipendio occasionali);
- allineare gli obiettivi delle persone con quelli dell'organizzazione;
- enfatizzare il lavoro di gruppo.

1.1.3 Processo

Il *processo* di sviluppo del software definisce le attività e le procedure organizzative per incrementare la collaborazione nel gruppo di sviluppo in modo da fornire al cliente un prodotto di qualità. Un modello di processo:

- stabilisce un ordine di esecuzione delle attività;
- specifica quali elaborati dello sviluppo sono forniti e quando;
- assegna le attività e gli elaborati da sviluppare agli sviluppatori;
- fornisce criteri per monitorare il progresso del processo, per misurarne i risultati e per pianificare i progetti futuri.

I processi di sviluppo non possono essere standardizzati o codificati in modo da poter essere automaticamente adottati da un'organizzazione. Ogni organizzazione dovrebbe sviluppare il proprio modello di processo o configurarlo a partire da un generico schema di processo, come lo schema *Rational Unified Process* (fornito dalla Rational Software Corp., Kruchten, 1999).

Il processo adottato da un'organizzazione deve essere in linea con il suo sviluppo culturale, la dinamica sociale, la conoscenza e la capacità degli sviluppatori, le aspettative dei clienti, le dimensioni dei progetti e anche i tipi di ambienti applicativi in cui sviluppa il software. Poiché tutti questi fattori sono soggetti a cambiamento, un'organizzazione può avere la necessità di diversificare il proprio modello di processo, creandone varianti per ogni tipologia di progetto software. Per esempio, in base alla familiarità degli sviluppatori con metodi e strumenti di modellazione, corsi speciali di apprendimento possono essere inclusi nel processo.

La *dimensione del progetto* ha probabilmente la maggiore influenza sul processo. In un progetto limitato (con una decina di sviluppatori), un processo formalizzato non è mai necessario. Un gruppo così ridotto preferisce comunicare e reagire in modo informale ai cambiamenti. In progetti più grandi, una rete informale di comunicazione non è sufficiente ed è necessario un processo di sviluppo ben definito.

1.1.3.1 Processo iterativo e incrementale

I moderni processi di sviluppo del software sono invariabilmente *iterativi* e *incrementali*. Modelli del sistema sono raffinati e trasformati durante le fasi di analisi, progetto e implementazione, maggiori dettagli sono aggiunti in *iterazioni* successive, modifiche e migliorie sono introdotte se necessario, rilasci incrementali (*incremental release*) di moduli software permettono di mantenere la soddisfazione dell'utente e forniscono importanti ritorni a moduli in fase di sviluppo.

Come indicato nel *Rational Unified Process*: "Un processo iterativo include la gestione di un flusso di rilasci eseguibili. Un processo incrementale comprende integrazioni continue nell'architettura di sistema per produrre tali rilasci, ciascuno dei quali contiene migliorie rispetto ai precedenti" (Booch *et al.*, 1999, p. 33.).

Il successo di un processo iterativo e incrementale è basato sulla identificazione iniziale dei *moduli costituenti l'architettura di sistema*. I moduli dovrebbero avere dimensioni comparabili, essere molto coesi e avere una minima sovrapposizione (accoppiamento). Anche l'ordine nel quale i moduli devono essere implementati è importante. Alcuni moduli non possono essere *rilasciati* se dipendono da informazioni relative ad altri moduli ancora da sviluppare. A meno che lo sviluppo iterativo e incrementale non sia pianificato e controllato, esso può degenerare verso uno sviluppo casuale, senza alcun controllo sul reale progresso del progetto.

1.1.3.2 Capability maturity model

Una delle principali sfide per qualsiasi organizzazione impegnata nella produzione di software è migliorare il proprio processo di sviluppo. Allo scopo, l'organizzazione deve conoscere quali sono i problemi del processo attualmente usato. Il CMM (*Capability Maturity Model*, modello per la valutazione del livello di maturità delle capacità aziendali) è un metodo comune per la valutazione e il miglioramento del processo (CMM, 1995).

IL CMM è stato definito dal Software Engineering Institute (SEI) presso la Carnegie Mellon University di Pittsburgh, USA, e originariamente usato dal DoD (Dipartimento della Difesa, USA) per valutare le capacità delle organizzazioni che proponevano contratti; ora è largamente usato dalle industrie dell'IT (*Information Technology*, tecnologia dell'informazione) in America e altrove.

Il CMM è essenzialmente un *questionario* che un'organizzazione deve riempire e che subisce poi un processo di verifica e l'assegnazione all'organizzazione di uno dei cinque livelli del modello. Quanto più alto è il livello, maggiore è la maturità di processo nell'organizzazione. La Figura 1.1 presenta i cinque livelli, con una breve descrizione delle principali caratteristiche di ciascun livello, e le principali aree di miglioramento da considerare per raggiungere un livello più alto.

Arthur (1992) chiama i livelli di maturità "la scala verso l'eccellenza nel software". I cinque gradini della scala sono: *iniziale* (situazione caotica), *ripetibile* (gestione del progetto ripetibile), *definito* (uso di metodi e strumenti per sviluppo e gestione del processo), *gestito* (possibilità di valutazione del processo), e *ottimizzato* (miglioramento continuo del processo).

L'esperienza ha mostrato che possono essere necessari svariati anni per progredire di un livello nella scala della maturità. La maggior parte delle organizzazioni sono a Livello 1, alcune a Livel-

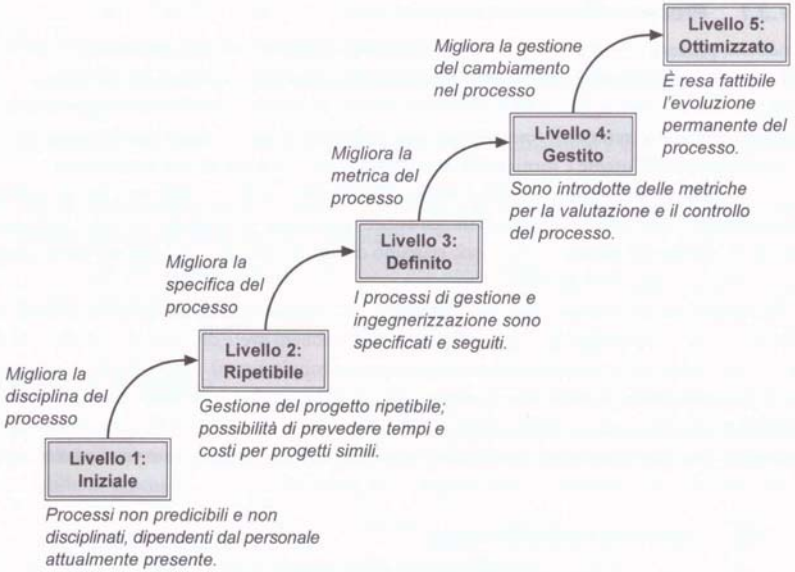


FIGURA 1.1 Livelli di maturità del processo nel CMM.

lo 2, pochissime a Livello 5. Le seguenti domande dimostrano le difficoltà per classificarsi a Livello 2. Un'organizzazione che vuole essere a Livello 2 del CMM deve fornire risposta positiva a ciascuna delle seguenti domande (e anche altre) (Pfleeger, 1998):

- La funzione di SQA (*Software Quality Assurance*, garanzia di qualità del software) prevede un canale informativo separato dalla gestione dello sviluppo del software?
- È presente una funzione di controllo della configurazione del software per ogni progetto che includa sviluppo di software?
- È presente un processo formale per la verifica del software?
- È presente una procedura formale per determinare le tempistiche di sviluppo del software?
- Esistono procedure formali per stimare il costo di sviluppo del software?
- Sono raccolte le statistiche sugli errori nel codice e sui test?
- Il gestore ha un meccanismo di controllo periodico dello stato dei progetti di sviluppo software?
- È presente un meccanismo per controllare le modifiche nei requisiti del software?

1.1.3.3 ISO 9000

Oltre al CMM, esistono altri modelli di miglioramento del processo. Di particolare interesse è la serie di standard di qualità ISO 9000, sviluppata da International Organization for Standardization (ISO). Lo standard ISO si applica sia alla gestione della *qualità* sia al *processo* di produzione di un prodotto. Lo standard è generico, si può applicare in qualsiasi industria e in tutte le tipologie di business, compreso lo sviluppo del software.

La principale premessa dello standard ISO 9000 è che se il processo è corretto allora anche il suo risultato (prodotto o servizio) lo sarà. "L'obiettivo della gestione della qualità è produrre prodotti di qualità costruendo la qualità piuttosto che valutando la qualità nei prodotti." (Schmauch, 1994, p. 1.)

Lo standard ISO 9000 non impone restrizioni né specifica processi. Lo standard prevede modelli di *cosa* deve essere realizzato, non di *come* le attività devono essere svolte. Un'organizzazione che richiede la certificazione ISO (chiamata anche registrazione) deve dire cosa fa, fare ciò che dice di fare e dimostrare che questo è stato fatto (Schmauch, 1994).

Un test significativo per un'organizzazione certificata ISO è che tale organizzazione dovrebbe essere capace di fornire un prodotto (o un servizio) di qualità anche se la sua intera forza-lavoro venisse rimpiazzata. A tale scopo, l'organizzazione deve *documentare e registrare* tutte le sue attività, per ciascuna delle quali devono essere definite procedure scritte, incluso il trattamento degli errori e i reclami dei clienti.

Come nel caso del CMM, la certificazione ISO può essere concessa solo dopo una revisione da parte di un certificatore ISO. Queste revisioni sono poi ripetute a intervalli regolari.

1.1.4 Linguaggi e strumenti di modellazione

Gli stakeholder e i processi sono due elementi in un "triangolo per il successo". Il terzo elemento consiste nel disporre di un linguaggio e di strumenti di modellazione: infatti, gli artifatti della modellazione devono essere comunicati (linguaggio) e documentati (strumenti).

Gli sviluppatori hanno bisogno di un *linguaggio* per costruire modelli (visuali e non) da discutere con i clienti e altri sviluppatori. Il linguaggio dovrebbe permettere la costruzione di modelli a vari livelli di astrazione in modo da presentare le soluzioni proposte a vari livelli di dettaglio.

Il linguaggio dovrebbe avere una forte componente *visuale* ("una figura è meglio di mille parole" recita un detto popolare). Dovrebbe anche possedere una robusta *semantica dichiarativa*, cioè permettere di rappresentare le procedure attraverso frasi assertive, in modo da comunicare indicando quali sono i bisogni piuttosto che come procedere per trattarli.

Gli sviluppatori hanno anche bisogno di *strumenti* CASE (*Computer-Assisted Software Engineering*, ingegneria del software assistita dall'elaboratore). Uno strumento CASE permette di memorizzare i modelli in un database o deposito (*repository*) centrale, di visualizzarli e di manipolarli in modo testuale o grafico. Idealmente, un repository dovrebbe permettere un accesso multi-utente ai modelli. Funzioni tipiche di un repository associato a uno strumento CASE sono:

- coordinare l'accesso ai modelli;
- semplificare la collaborazione tra gli sviluppatori;

- memorizzare più versioni di uno stesso modello, identificando anche le differenze tra le diverse versioni;
- permettere di condividere uno stesso concetto tra più modelli;
- controllare la consistenza dei modelli;
- generare i rapporti e la documentazione di progetto;
- generare le strutture dati e il codice (forward engineering);
- generare modelli da implementazioni esistenti (reverse engineering), etc.

Si noti che un programma generato da un CASE è frequentemente uno *scheletro* di codice, mentre gli algoritmi devono essere codificati dai programmatori seguendo le procedure ordinarie.

1.1.4.1 Unified modeling language

"Il linguaggio UML (*Unified Modeling Language*, linguaggio unificato di modellazione) è un linguaggio visuale di modellazione per scopi generali che è usato per specificare, costruire e documentare gli elaborati di un sistema software." (Rumbaugh *et al.*, 1999). UML fu sviluppato dalla Rational Software Corp. per unificare le migliori caratteristiche delle notazioni e metodi esistenti e, nel 1997, il comitato OMG (Object Management Group) lo approvò come linguaggio standard di modellazione. Da allora, UML è stato ulteriormente sviluppato ed è ampiamente adottato nel settore IT.

UML è indipendente da qualunque processo di sviluppo del software, sebbene Rational abbia più tardi proposto una corrispondenza con il processo chiamato *Rational Unified Process* (Kruchten, 1998). Chiaramente, un processo che adotta UML deve seguire un *approccio object-oriented* allo sviluppo del software. UML non è appropriato agli approcci strutturati di vecchio tipo che forniscono un supporto per sistemi implementati con i tradizionali linguaggi di programmazione procedurali, come il COBOL.

UML è anche indipendente dalle tecnologie di implementazione (purché esse siano object-oriented). Ciò rende UML talvolta carente come supporto alla fase di progetto dettagliato nel ciclo di sviluppo, ma, per lo stesso motivo, UML risulta piuttosto flessibile rispetto alle modifiche nelle piattaforme di implementazione.

I costrutti del linguaggio UML permettono la modellazione della struttura statica e del comportamento dinamico di un sistema. Il sistema è rappresentato come un insieme di *oggetti* che collaborano (moduli software) i quali reagiscono a eventi esterni per eseguire attività a beneficio dei clienti (utilizzatori). Certi modelli UML enfatizzano alcuni aspetti del sistema e ignorano aspetti che sono enfatizzati da altri modelli. Insieme, tutti i modelli forniscono una descrizione completa del sistema e possono essere classificati in tre gruppi:

- *modelli dello stato* (descrivono le strutture statiche dei dati);
- *modelli del comportamento* (descrivono le collaborazioni tra oggetti);
- *modelli del cambiamento di stato* (descrivono gli stati permessi dal sistema nel tempo).

UML contiene anche alcuni *costrutti architetturali* che permettono la modularizzazione del sistema ai fini dello sviluppo iterativo e incrementale.

1.1.4.2 Strumenti CASE e miglioramento del processo

Il miglioramento del processo è molto più che l'introduzione di nuovi metodi e tecniche. Infatti, l'introduzione di nuovi metodi e tecniche in un'organizzazione con una bassa maturità di processo può portare più danni che benefici.

Un esempio di ciò è la tecnologia CASE. Uno strumento CASE *integrato* può permettere a più sviluppatori di collaborare e condividere le informazioni di progetto al fine di produrre nuovi elaborati progettuali. Un tale strumento impone certi processi cui il gruppo di sviluppo deve adeguarsi onde trarre vantaggio dall'impiego di tale tecnologia, ma se il gruppo di sviluppo non è riuscito a migliorare i suoi processi, difficilmente assimilerà il processo dettato dallo strumento. Il risultato è che la produttività potenziale e la maggiore qualità offerta dalla nuova tecnologia non si realizzeranno.

La precedente discussione non deve portare il lettore a credere che la tecnologia CASE sia un "*business rischioso*"; rappresentare i risultati del progetto con carta e matita ha senso solo nella didattica, mai in progetti reali. Semplicemente, è probabile che tale tecnologia non porti i benefici attesi se si intende usarla per guidare il gruppo di sviluppo e il gruppo non è preparato a seguire i passi necessari. Comunque, gli stessi metodi e tecniche potrebbero migliorare produttività e qualità per singolo sviluppatore che usa la tecnologia localmente sulla propria stazione di lavoro.

1.2 Pianificazione di sistema

I progetti dei sistemi informativi devono essere *pianificati*, cioè devono essere identificati, classificati, ordinati per importanza, e selezionati per poi svilupparli, migliorarli o, eventualmente, eliminarli. La domanda è: quale delle tecnologie e applicazioni nel campo IS (*Information System*, sistema informativo) permetteranno maggiori benefici? Teoricamente, la decisione su come procedere dovrebbe essere basata su una *strategia di business* e su un'accurata e sistematica pianificazione (Bennett *et al.*, 1999; Hoffer *et al.*, 1999; Maciaszek, 1990).

La strategia di business può essere determinata attraverso vari processi conosciuti come *pianificazione strategica, modellazione del business, reingegnerizzazione dei processi di business, allineamento strategico, gestione delle risorse informative*, o simili. Non è nostro scopo spiegare le differenze tra i vari approcci: è sufficiente dire che tutti gli approcci sottintendono lo studio dei processi di business dell'organizzazione per avere una visione a lungo termine e quindi dare la priorità a quegli elementi cui può essere data risposta con la Tecnologia dell'Informazione.

Ciò detto, esistono molte organizzazioni, in particolare quelle "piccole", con una strategia di business non chiara. Tali organizzazioni decidono sullo sviluppo dei sistemi informativi semplicemente identificando i problemi attuali più pressanti che necessitano di una soluzione. Se l'ambiente esterno o le condizioni interne del business mutano, i sistemi informativi esistenti devono, a loro volta, essere modificati. Mentre questo approccio ha ovvi svantaggi, permette alle piccole

organizzazioni di rifocalizzarsi velocemente sulla situazione corrente, o per trarre vantaggio da nuove opportunità o per allontanarsi da situazioni pericolose.

Le grandi organizzazioni non possono affrontare continue modifiche alle proprie direzioni di business. In realtà, spesso esse dettano le direzioni per altre organizzazioni. In qualche modo, esse possono (entro certi limiti) plasmare l'ambiente ai propri *attuali* bisogni. Comunque, le grandi organizzazioni devono accuratamente "*guardare al futuro*", usando un approccio basato sulla pianificazione per identificare i progetti da sviluppare. Questi sono tipicamente grandi progetti che necessitano di molto tempo per essere completati.

La pianificazione di sistema può essere fatta in modi differenti. Un approccio tradizionale è soprannominato SWOT (*Strengths, Weaknesses, Opportunities, Threats,* punti di forza, di debolezza, opportunità e pericoli). Un'altra popolare strategia è basata sul VCM (*Value Chain Model,* modello della catena del valore). Approcci più moderni per lo sviluppo di una strategia di business sono noti come BPR (*Business Process Reengineering,* reingegnerizzazione dei processi di business). I bisogni informativi di un'organizzazione possono anche essere stimati usando gli schemi dell'approccio ISA (*Information System Architecture,* architettura del sistema informativo); tali schemi sono stati ottenuti per analogia con approcci che hanno avuto successo in altre discipline non necessariamente relative al settore IT (ad esempio, l'industria delle costruzioni).

Tutti gli approcci alla pianificazione di sistema hanno un importante comune denominatore: tutti riguardano l'*efficacia* (fare la cosa giusta) piuttosto che l'*efficienza* (fare bene le cose). Risolvere in modo efficiente un problema sbagliato non è un gran risultato!

1.2.1 Approccio SWOT

L'approccio SWOT permette l'identificazione, classificazione, ordinamento e selezione dei progetti di sviluppo del sistema informativo in modo che questi siano *allineati* con i punti di forza, di debolezza, le opportunità e i pericoli in cui può incorrere l'organizzazione. L'approccio è di tipo top-down e ha come punto di partenza la definizione della *missione* aziendale. La missione è la "ragion d'essere" dell'organizzazione e ne sintetizza il carattere unico; a partire dalla missione è necessario specificare una *visione* futura (la "realtà auspicata") ponendo, generalmente, l'enfasi sui bisogni del cliente piuttosto che sui prodotti o servizi che l'organizzazione stessa fornisce.

La missione e la visione aziendale (e la strategia di business che ne deriva) prendono in considerazione i *punti di forza e di debolezza interni all'azienda* nelle aree della gestione, produzione, risorse umane, finanza, marketing, ricerca e sviluppo, etc. Tali punti di forza e debolezza devono essere accuratamente analizzati, essere oggetto di consenso e di valutazione al fine di stabilire fra loro delle priorità.

L'identificazione dei punti di forza e debolezza interni all'azienda è una condizione necessaria ma non sufficiente per una pianificazione di successo del business. Inoltre, un'organizzazione non agisce nel vuoto: essa dipende da fattori economici, sociali, politici e tecnologici esterni, perciò deve individuare le opportunità esterne (onde trarne un vantaggio) e i pericoli che possono arrivare dall'esterno (per cercare di evitarli). Questi sono fattori che un'organizzazione non può controllare ma la loro conoscenza è essenziale nella determinazione dei suoi obiettivi e scopi.

Le organizzazioni perseguono in qualunque momento uno o alcuni *obiettivi*. Gli obiettivi sono normalmente a lungo termine (da tre a cinque anni) o addirittura "senza tempo". Esempi tipici di obiettivi sono: migliorare la soddisfazione del cliente, introdurre nuovi servizi, incrementare il controllo sui fornitori, etc. Ogni obiettivo strategico deve essere associato a *scopi* specifici, normalmente espressi su base annuale. Per esempio, l'obiettivo "migliorare la soddisfazione del cliente" può essere supportato dallo scopo "gestire gli ordini cliente più velocemente" (ad esempio, in due settimane).

Gli obiettivi e gli scopi richiedono *strategie* di gestione e specifiche *politiche* per l'implementazione di dette strategie. Tali strumenti gestionali dovrebbero adattare le strutture organizzative, allocare le risorse e definire progetti di sviluppo, compresi quelli riguardanti i sistemi informativi.

1.2.2 Approccio VCM

Il modello VCM stima il vantaggio competitivo analizzando l'intera sequenza delle attività in un'organizzazione, dalle materie prime ai prodotti finiti, venduti e inviati ai clienti. La metafora della catena enfatizza il principio che un singolo legame debole può causare la rottura dell'intera catena.

Il modello ha come scopo il valutare quale configurazione della catena del valore fornirà il vantaggio competitivo maggiore. I progetti di sviluppo dell'IS possono quindi indirizzarsi verso quei segmenti, operazioni, canali di distribuzione, approcci di marketing etc. che danno il maggior vantaggio.

Nell'approccio VCM originale (Porter, 1985), le attività organizzative sono classificate in *attività primarie* e *attività di supporto*. Le attività primarie creano o aggiungono valore a un prodotto finale. Esse sono distinte in cinque passi successivi: (1) logistica in ingresso, (2) operazioni, (3) logistica in uscita, (4) vendita e marketing, (5) servizi. Le attività di supporto non aggiungono valore, almeno direttamente. Sono comunque essenziali ma non arricchiscono il prodotto. Le attività di supporto comprendono: (1) amministrazione e infrastruttura, (2) gestione delle risorse umane, (3) ricerca e sviluppo e, non sorprendentemente, (4) sviluppo dell'IS.

Mentre il VCM è un utile strumento per la pianificazione dell'IS, l'informatizzazione dell'azienda può facilitare modifiche del business che a loro volta possono creare vantaggi competitivi. In altre parole, *la tecnologia dell'informazione può trasformare la catena del valore dell'organizzazione,* quindi fra tecnologia e catena del valore può instaurarsi un "circolo virtuoso" utile per entrambe.

Porter e Millar (1985) identificano cinque passi che un'organizzazione può seguire per sfruttare le opportunità offerte dalla IT:

1. Stimare il contenuto informativo di prodotti e processi.
2. Stimare il ruolo della IT nella struttura industriale.
3. Identificare e graduare i modi in cui la IT potrebbe creare un vantaggio competitivo.
4. Considerare come la IT potrebbe creare nuovi business.
5. Sviluppare un piano per trarre vantaggio dalla IT.

1.2.3 Approccio BPR

L'approccio BPR alla pianificazione dei sistemi è fondato sull'ipotesi che le organizzazioni moderne devono abbandonare la scomposizione funzionale, le strutture gerarchiche e i principi operativi attualmente in uso e *reinventarsi*.

Tale concetto fu introdotto nel 1990 da Hammer (1990) e da Davenport e Short (1990), generando immediatamente sia interessi sia controversie. Una descrizione estesa del BPR può essere trovata nei libri relativi (Hammer e Champy, 1993; Davenport, 1993).

Oggi, la maggior parte delle organizzazioni sono strutturate in *unità verticali* focalizzate su funzioni, prodotti o settori. Tali strutture e stili di lavoro risalgono al 18° secolo e ad Adam Smith con il suo principio di divisione del lavoro e, conseguentemente, frammentazione delle attività. Secondo quest'impostazione, nessun impiegato o dipartimento è responsabile per un *processo di business* che è definito come "... una collezione di attività che ricevono uno o più tipi di ingressi e creano un risultato utile per il cliente" (Hammer e Champy, 1993).

Il BPR sfida i principi industriali di Smith di divisione del lavoro, controllo gerarchico ed economie di scala. Nel mondo d'oggi, le organizzazioni devono essere capaci di adattarsi velocemente alle variazioni del mercato, alle nuove tecnologie, ai fattori competitivi, alla domanda, etc. Strutture organizzative rigide, nelle quali i processi di business devono essere spezzati tra molti dipartimenti, sono obsolete. Le organizzazioni devono focalizzarsi sui processi di business piuttosto che su singole mansioni, attività, persone, o funzioni dipartimentali. Questi processi attraversano *orizzontalmente* il business e terminano ai punti di contatto con il cliente. "La differenza più marcata tra un'organizzazione per processi e una tradizionale è l'esistenza di un responsabile di processo" (Hammer e Stanton, 1999).

L'obiettivo principale del BPR è di riprogettare radicalmente i processi di business di un'organizzazione (quindi il BPR è talvolta noto come *process redesign*). I processi di business devono essere identificati, analizzati e migliorati. Un processo è documentato con un diagramma di *workflow* (flusso di lavoro) che specifica il flusso di eventi, documenti e informazioni, e può essere usato per valutare tempi, risorse e costi necessari per eseguire le attività del processo stesso.

Il maggior ostacolo all'implementazione del BPR nelle organizzazioni risiede nella necessità di inserire una visione orizzontale del processo nella tradizionale gestione verticale. Una seria iniziativa di BPR richiede quindi il cambiamento dell'organizzazione intorno a gruppi di sviluppo considerati come unità organizzative principali. Questi gruppi sono responsabili per uno o più processi integrati di business.

Talvolta una modifica radicale è inaccettabile. Le strutture tradizionali non possono essere modificate "in una notte". Una spinta radicale può incontrare forti resistenze e i benefici potenziali del BPR possono essere in tal modo compromessi. Sotto certe condizioni, un'organizzazione può trarre benefici da un continuo miglioramento "per piccoli passi" dei processi di business piuttosto che da una loro radicale reingegnerizzazione: è questo il concetto del BPI (*Business Process Improvement*, miglioramento continuo dei processi) (Allen e Frost, 1998).

Una volta che i processi di business sono definiti, i responsabili dei processi potrebbero richiedere che la IT fornisca un ulteriore supporto per aumentare la loro efficienza. Di conseguenza, lo sviluppo del SI potrebbe concentrarsi sull'implementazione del workflow identificato. La combinazione di *efficacia* del BPR e di *efficienza* della IT possono produrre notevoli benefici su tutti gli indici di prestazione dell'organizzazione, come la qualità del servizio, l'efficienza, il costo, il vantaggio competitivo, la flessibilità etc.

1.2.4 Approccio ISA

Diversamente dagli approcci descritti, ISA è un approccio bottom-up che fornisce uno *schema architetturale neutrale* per soluzioni IS. Esso non comprende una metodologia di pianificazione di sistema, semplicemente offre uno schema di riferimento in grado di inquadrare la maggior parte delle strategie di business.

L'approccio ISA è stato introdotto dall'articolo fondamentale di Zachman (1987) e più tardi esteso da Sowa e Zachman (1992). Una versione leggermente modificata dell'articolo originale è stata pubblicata sempre da Zachman (1999).

Lo schema di riferimento è rappresentato come una tabella di trenta caselle organizzate in cinque righe (etichettate da 1 a 5) e sei colonne (etichettate da A a F). Le righe rappresentano le differenti *prospettive* usate nella costruzione di un complesso prodotto di ingegneria, come può considerarsi un sistema informativo. Queste prospettive sono quelle dei cinque maggiori *attori* coinvolti nell'IS:

1. Pianificatore (definisce lo scopo del sistema);
2. Responsabile (definisce il modello concettuale);
3. Progettista (specifica il modello fisico del sistema);
4. Programmatore (definisce le soluzioni tecnologiche dettagliate);
5. Sottocontraente (fornisce componenti del sistema).

Le sei colonne rappresentano sei differenti *modelli di descrizione* o *architetturali* con cui ciascuno dei partecipanti lavora. Come per le prospettive, le descrizioni sono piuttosto differenti tra loro ma sono allo stesso tempo intrinsecamente legate. Le descrizioni rispondono a sei domande concernenti ciascun partecipante:

A. *Di cosa sono fatte le cose?* (cioè i dati nel caso dell'IS);
B. *Come le cose funzionano?* (cioè i processi di business);
C. *Dove le cose si trovano?* (cioè la posizione dei componenti di processo);
D. *Chi lavora con le cose?* (cioè gli utilizzatori);
E. *Quando le cose accadono?* (cioè la programmazione di eventi e stati);
F. *Perché le cose si verificano?* (cioè le motivazioni dell'azienda).

La combinazione di prospettive e descrizioni in trenta caselle fornisce una potente tassonomia per un'architettura completa di sviluppo dell'IS. Le prospettive differiscono per come sono impiegate e per l'uso di rappresentazioni modellistiche differenti. Differenti modelli enfatizzano differenti punti di vista dei partecipanti. Allo stesso modo, le descrizioni sono indirizzate verso differenti obiettivi, e ciascuna risponde a differenti domande.

La caratteristica più interessante dell'approccio ISA sta nel fatto che lo schema è sufficientemente flessibile per adattarsi a modifiche future delle condizioni di business e delle risorse. Questo perché una soluzione ISA non è derivata da una particolare strategia di business, è solo uno schema per una descrizione completa dell'IS.

1.2.5 Livelli gestionali di un'organizzazione

Il discorso sulla pianificazione di sistema è associato al fatto che un'organizzazione è generalmente gestita a tre diversi livelli decisionali:

1. Strategico;
2. Tattico;
3. Operativo.

I tre livelli sono caratterizzati da ottiche diverse sulle decisioni aziendali, da insiemi distinti di applicazioni che l'IS è chiamato a gestire, e da supporti specifici richiesti alla IT. È responsabilità della pianificazione di sistema definire l'insieme di applicazioni dell'IS e soluzioni IT che siano le più efficaci per l'organizzazione in un particolare momento. La Tabella 1.1 definisce gli elementi coinvolti nei tre livelli decisionali, le applicazioni dell'IS e le soluzioni IT (Jordan e Machesky, 1990; Robson, 1994).

Le applicazioni dell'IS e le soluzioni IT che offrono i maggiori ritorni a un'organizzazione sono quelle a *livello strategico*. Tuttavia, queste sono anche le soluzioni più difficili da implementare e richiedono una progettazione competente e specializzata. Dopo tutto, questi sono i sistemi che danno a un'organizzazione un reale vantaggio competitivo sul mercato.

All'altro capo dello spettro, i sistemi di supporto al *livello di gestione operativa* sono quelli consueti, usano una tecnologia convenzionale per database, e sono frequentemente configurati a partire da soluzioni preesistenti. Questi sistemi non determinano un vantaggio competitivo, ma senza di loro l'organizzazione non è in grado di funzionare correttamente.

Ogni organizzazione moderna possiede un insieme completo di sistemi a livello operativo ma solo le organizzazioni meglio gestite possiedono un insieme integrato di applicazioni di IS a livello strategico. La principale tecnologia per memorizzare e recuperare i dati per decisioni strategiche e tattiche è nota come *data warehouse* (Kimball, 1996).

TABELLA 1.1 Supporto fornito dall'IS e dall'IT per i diversi livelli decisionali.

Livello decisionale	Focalizzazione delle decisioni	Tipiche applicazioni dell'IS	Tipiche soluzioni IT
Strategico	Strategie di supporto agli obiettivi organizzativi di lungo periodo	Analisi del mercato e delle vendite, Pianificazione di prodotto, Valutazione delle prestazioni	Data mining, Gestione della conoscenza aziendale
Tattico	Politiche di supporto agli obiettivi di breve periodo e all'allocazione delle risorse	Analisi di bilancio, Previsioni retributive, Ordinamento delle giacenze, Servizi al cliente	Data warehouse, Elaborazione analitica, Fogli elettronici
Operativo	Attività giornaliere di staff e supporto alla produzione	Stipendi, Fatturazione, Acquisti, Contabilità	Database, Elaborazione transazionale, Generatori di applicazioni

1.3 Fasi del ciclo di vita del software

Lo sviluppo del software segue un'evoluzione nel tempo, generalmente indicata come *ciclo di vita*. Il ciclo di vita è un insieme ordinato di attività gestite nel corso di ogni progetto di sviluppo e viene realizzato mediante opportuni processi e metodi, organizzati in *fasi* che il prodotto software attraversa dal principio fino alla sua dismissione.

Il ciclo di vita dello sviluppo del software può essere presentato a differenti livelli di dettaglio delle fasi. A un *livello di massima*, il ciclo di vita è rappresentabile con sole tre fasi:

1. Analisi.
2. Progetto.
3. Implementazione.

La *fase di analisi* è concentrata sui requisiti del sistema. I requisiti sono individuati e specificati. I modelli delle funzioni e dei dati del sistema sono sviluppati e integrati e sono presi in considerazione anche i requisiti non-funzionali e altri vincoli di sistema.

La *fase di progetto* è separata in due maggiori sottofasi: progetto architetturale e progetto di dettaglio. In particolare, viene generalmente approfondito il progetto di programmi client/server che integrano interfaccia utente e oggetti del database. Vari elementi progettuali che influenzano la comprensibilità, la manutenibilità e la scalabilità del sistema devono essere considerati e documentati.

La *fase di implementazione* comprende la codifica dei programmi per le applicazioni client e il database server. Processi incrementali e iterativi di implementazione sono enfatizzati. Per realizzare un buon prodotto, è importante procedere in modo ciclico, dalla specifica dei modelli all'implementazione delle applicazioni client e del database server, e viceversa (*round-trip engineering*).

Per riassumere, l'analisi riguarda quello che si deve fare, il progetto riguarda come farlo usando la tecnologia disponibile, e l'implementazione riguarda l'attuazione pratica (nel senso di rilasciare al cliente il prodotto software finale).

A un *livello di granularità più fine*, il ciclo di vita può essere suddiviso nelle seguenti sette fasi:

1. Determinazione dei requisiti.
2. Specifica dei requisiti.
3. Progetto architetturale.
4. Progetto di dettaglio.
5. Implementazione.
6. Integrazione.
7. Manutenzione (ed eventuale dismissione).

Alcuni autori includono anche la *Pianificazione delle attività* e il *Testing* come fasi aggiuntive; queste due importanti attività *non* dovrebbero essere considerate come fasi separate poiché riguardano l'intero ciclo di vita. Infatti, dovrebbe essere introdotto fin dall'inizio un piano di gestione del progetto, arricchendolo successivamente nella fase di specifica, e facendolo continuamente evolvere durante il resto del ciclo di vita. Analogamente, il testing è cruciale dopo l'implementazione ma dovrebbe essere applicato agli elaborati del progetto in una qualunque altra fase.

1.3.1 Fase di determinazione dei requisiti

Kotonya e Sommerville (1998) definiscono un *requisito* come "Un'asserzione riguardante un servizio del sistema o un vincolo". Un'asserzione di servizio descrive come il sistema dovrebbe comportarsi rispetto a un utente individuale o a un'intera comunità di utenti. Nell'ultimo caso, l'asserzione definisce una *regola di business* che deve essere sempre rispettata (ad esempio, "I salari settimanali sono pagati tutti i venerdì"). Un'asserzione di servizio può anche riferirsi a qualche procedura che il sistema deve applicare (ad esempio, "Il calcolo della commissione di un venditore è basato sulle vendite dell'ultimo mese usando la formula ...").

Un *vincolo* esprime una restrizione sul comportamento del sistema o sullo sviluppo del sistema. Un esempio del primo tipo può essere il vincolo sulla sicurezza: "Solo i manager diretti possono vedere le informazioni sullo stipendio dei propri dipendenti". Un esempio del secondo tipo può essere: "È necessario usare XXX come strumento di sviluppo".

Si noti che talvolta la distinzione tra un vincolo sul comportamento del sistema e una regola di business è vaga. Ciò non costituisce un problema se tutti i requisiti sono comunque identificati e le ripetizioni eliminate.

La fase di determinazione dei requisiti deve individuare, analizzare e negoziare i requisiti con i clienti. La fase può utilizzare varie tecniche di raccolta dell'informazione come: interviste strutturate e non strutturate di utenti, questionari, studio di documenti e moduli, registrazioni video etc. Una tecnica moderna è lo sviluppo di un prototipo (*rapid prototyping*) della soluzione in modo che requisiti complessi possano essere rapidamente chiariti evitando incomprensioni.

La negoziazione tra sviluppatori e clienti è anche necessaria per eliminare requisiti in contraddizione e in sovrapposizione tra loro, e per allineare periodicamente il progetto al budget e alle scadenze.

Il prodotto della fase di determinazione dei requisiti è un *documento dei requisiti* costituito in gran parte da un testo in linguaggio naturale con alcuni diagrammi e tabelle informali. Nessun modello formale è usato se non l'adozione eventuale di poche semplici notazioni comuni che possano essere facilmente comprese onde semplificare la comunicazione con i clienti.

1.3.2 Fase di specifica dei requisiti

La fase di specifica dei requisiti inizia quando gli sviluppatori passano alla modellazione dei requisiti con una particolare tecnica (come UML), usando uno strumento CASE per inserire, analizzare e documentare i modelli in costruzione. Come risultato, il documento dei requisiti è arricchito con modelli visuali e rapporti generati dallo strumento. In pratica, un *documento delle specifiche* rimpiazza il documento dei requisiti.

Le due più importanti tecniche di specifica nell'analisi object-oriented sono il *diagramma delle classi* e il *diagramma dei casi d'uso* per rappresentare dati e funzionalità del sistema. Un tipico documento delle specifiche conterrà anche altri requisiti, riguardanti, ad esempio, i problemi politici e legali, le prestazioni, l'usabilità, la manutenibilità e la sicurezza.

I differenti *modelli di specifica* permettono di vedere la situazione da molte e differenti angolature, in modo che aspetti specifici possano essere enfatizzati e analizzati. Anche la consistenza e la completezza dei requisiti devono essere accuratamente controllate.

Idealmente, i modelli di specifica dovrebbero essere indipendenti dalla piattaforma hardware/software sulla quale il sistema deve essere implementato. Considerare i problemi implementativi impone pesanti restrizioni sul vocabolario (quindi sull'espressività) del linguaggio di modellazione. Inoltre, il vocabolario può essere difficilmente comprensibile per i clienti, con la conseguenza di inibire la comunicazione tra sviluppatori e clienti.

Ciò detto, alcuni dei vincoli potrebbero, di fatto, influenzare le scelte hardware e software degli sviluppatori. Inoltre, gli stessi clienti potrebbero esprimere propri requisiti in relazione a una particolare tecnologia hardware/software, o anche richiedere un particolare strumento. La conclusione è: evitare considerazioni hardware/software, se possibile.

1.3.3 Fase di progetto architetturale

Il documento di specifica, elencando tutti i requisiti che il prodotto software deve soddisfare, è simile a un *contratto* tra sviluppatori e clienti per la fornitura di un prodotto. Nel progetto architetturale le specifiche vengono prese in consegna da architetti e progettisti di sistema per sviluppare modelli a più basso livello, riguardanti l'architettura di sistema e i suoi comportamenti interni. Il progetto è sviluppato tenendo conto della piattaforma hardware/software sulla quale il sistema sarà implementato.

La descrizione del sistema in termini dei suoi moduli è chiamato *progetto architetturale*, e deve specificare le decisioni sulle strategie di soluzione relative agli aspetti client e server del sistema.

La descrizione dei comportamenti interni di ciascun modulo (caso d'uso) è compito della fase successiva, il *progetto di dettaglio*, ove si sviluppano algoritmi e strutture dati per ciascun modulo tenendo conto dei vincoli imposti dalla sottostante piattaforma implementativa.

Il progetto architetturale riguarda la scelta di una strategia di soluzione insieme a una modularizzazione del sistema. Tale strategia deve decidere in merito agli aspetti relativi all'interfaccia utente (client) e al database (server), così come sui programmi (*middleware*) necessari per collegare il server ai client. La decisione sui blocchi di base (moduli) è relativamente indipendente sebbene un progetto dettagliato debba essere conforme alla soluzione client/server scelta.

I modelli client/server sono spesso estesi per permettere *un'architettura a tre livelli (three-tier)* dove la logica applicativa costituisce un livello separato. Il livello intermedio è un livello logico e come tale potrebbe o meno essere supportato da un hardware separato. La logica applicativa è un processo che si esegue o sul lato client o sul lato server (cioè può essere compilato col processo client o server) e implementato mediante un DLL (*Dynamic Link Library*, libreria dinamica di collegamento), un API (*Application Programming Interface*, interfaccia di programmazione applicativa), oppure con RPC (*Remote Procedure Call*, chiamata di procedura remota), etc. (Next, 1996).

1.3.4 Fase di progetto di dettaglio

Il progetto architetturale descrive il prodotto in termini dei suoi moduli. Il *progetto di dettaglio* descrive ciascun modulo. Generalmente, i moduli sono allocati o in un componente client o in un componente server, i progettisti applicativi sono responsabili per i primi, i progettisti del database dovrebbero sviluppare i secondi.

Il progetto *dell'interfaccia utente* (client) deve adeguarsi alle linee guida fornite dal produttore (Windows, Motif, Macintosh) di una particolare interfaccia GUI (*Graphical User Interface*, interfaccia grafica utente). Tali linee guida sono normalmente fornite online come parte della documentazione elettronica del GUI (ad esempio, Windows 2000).

Un principio basilare per il progetto object-oriented di un'interfaccia GUI è che l'*utente ha il controllo* (e non il programma). Il programma reagisce agli eventi casuali generati dall'utente e fornisce i servizi software necessari. I restanti principi di progetto dell'interfaccia sono conseguenza di tale fatto (ovviamente, tale principio non deve essere preso alla lettera, il programma dovrebbe, ad esempio, validare i privilegi dell'utente e non permettere certe azioni).

Il *progetto del database* definisce gli oggetti nel database server, generalmente un server relazionale (o eventualmente object-relational). Alcuni di questi oggetti sono contenitori di dati (tabelle, viste etc.), altri sono procedurali, come *procedure memorizzate* (stored procedure), azioni eseguite automaticamente dal sistema al verificarsi di certe condizioni (*trigger*) etc.

1.3.5 Fase d'implementazione

L'implementazione di un sistema informativo comprende *l'installazione* di software acquistato e la *codifica* di software sviluppato per il cliente. Comprende anche altre importanti attività, come il caricamento dei database di test e di produzione, il testing, l'addestramento degli utilizzatori, etc.

Generalmente in un gruppo di implementazione si distinguono due insiemi di programmatori: uno è responsabile per la programmazione lato client e l'altro per la programmazione del database server. I programmi client implementano finestre, logica applicativa e chiamate ai programmi del database server (procedure memorizzate) quando necessario. La responsabilità per la consistenza del database e per la correttezza transazionale è legata ai programmi lato server.

Nel quadro di uno sviluppo iterativo e incrementale, il progetto delle *interfacce utente* è talvolta soggetto a modifiche nell'implementazione. I programmatori applicativi possono optare per una diversa forma delle finestre per adeguarsi ai principi dei fornitori di GUI, anche per facilitare la programmazione o per incrementare la produttività dell'utente.

Analogamente, l'implementazione del *database server* può forzare modifiche ai documenti di progetto. Problemi inaspettati riguardanti il database, le difficoltà con la programmazione di procedure memorizzate e trigger, gli aspetti della concorrenza, l'integrazione con i processi client, l'ottimizzazione delle prestazioni, etc., sono solo alcune delle ragioni per cui il progetto potrebbe richiedere delle modifiche.

1.3.6 Fase d'integrazione

Lo sviluppo incrementale implica l'*integrazione incrementale* di moduli software man mano che questi vengono prodotti. Tale attività non è semplice. Per grandi sistemi, l'integrazione dei moduli può richiedere maggior tempo e sforzo che ogni altra precedente fase del ciclo di vita, compresa l'implementazione. Come osservò Aristotele: "Il tutto è più che la somma delle sue parti".

L'integrazione dei moduli deve essere chiaramente pianificata fin dall'inizio del ciclo di vita del software. Le unità di software, da implementare singolarmente, devono essere identificate nei passi iniziali dell'analisi di sistema e riviste accuratamente durante il progetto architetturale. La sequenza di implementazione deve permettere l'integrazione incrementale nel modo più semplice possibile.

La difficoltà principale nell'integrazione incrementale deriva dalle mutue dipendenze tra i moduli. In un sistema ben progettato tale *accoppiamento (coupling)* è minimizzato. Tuttavia, accade spesso che due moduli siano mutuamente dipendenti, per cui nessuno dei due può funzionare da solo.

Che cosa fare se è necessario rilasciare un modulo prima che un altro sia pronto? La risposta è la scrittura di uno speciale codice per colmare temporaneamente "il vuoto" così che tutti i moduli possano essere integrati. I programmi che simulano l'attività di un modulo mancante sono chiamati *emulatori* (stub).

A differenza di sistemi software convenzionali, basati sulla nozione di programma principale, nei moderni sistemi object-oriented guidati da eventi non esiste un'intelligenza centrale (il programma principale). Non c'è una struttura d'integrazione chiaramente definita e le convenzionali strategie d'integrazione top-down e bottom-up sono poco utilizzabili.

I sistemi object-oriented devono essere *progettati per l'integrazione.* Ciascun modulo deve, per quanto possibile, essere indipendente. Le dipendenze tra moduli dovrebbero essere identificate e minimizzate nelle fasi di analisi e progetto. Idealmente, ciascun modulo dovrebbe corrispondere a un processo che viene eseguito in risposta a una particolare necessità del cliente. L'uso di emulatori per rimpiazzare le operazioni dovrebbe essere evitato quando possibile: se non correttamente progettati, la fase di integrazione potrebbe risultare caotica mettendo a rischio l'intero progetto di sviluppo.

1.3.7 Fase di manutenzione

La *manutenzione* segue al rilascio presso il cliente del prodotto software completo o di alcuni suoi moduli. Essa non è solo una parte integrante del ciclo di vita del software, ma è anche la più pesante in termini di tempi e risorse assorbiti. Schach (1996) stima che il 67% del tempo del ciclo di vita è speso nella manutenzione del software.

La manutenzione consiste di tre attività distinte (Maciaszek, 1990):

1. gestione,
2. manutenzione adattativa,
3. manutenzione evolutiva.

La *gestione* riguarda la manutenzione ordinaria necessaria per mantenere il sistema operativo e accessibile agli utenti. La *manutenzione adattativa* comprende il monitoraggio e la verifica delle operazioni del sistema, l'adattamento delle sue funzionalità alle modifiche dell'ambiente, e l'ottimizzazione delle prestazioni del sistema per portarlo ai livelli richiesti. La *manutenzione evolutiva* si riferisce alla riprogettazione e alla modifica del sistema per trattare nuovi requisiti o modificare sostanzialmente quelli esistenti.

La fase di manutenzione di un sistema software può diventare a un certo punto non più sostenibile e il sistema deve essere dismesso. La *dismissione* potrebbe accadere per ragioni che poco hanno a che vedere con *l'utilità* del software. Il software è probabilmente ancora utile ma è diventato "non manutenibile". Schach (1996) elenca quattro ragioni possibili per la dismissione del software:

1. le modifiche proposte oltrepassano le capacità immediate della manutenzione evolutiva;
2. il sistema è fuori dal controllo dei responsabili della manutenzione e gli effetti delle modifiche non possono essere previsti;
3. la documentazione sulla quale basare le possibili estensioni del software è carente o inesistente;
4. la sostituzione della piattaforma hardware/software con l'assenza di una ragionevole strategia di migrazione.

1.3.8 Pianificazione di progetto nel ciclo di vita del software

Una massima familiare recita che "ciò che non si può pianificare non si può realizzare". La pianificazione riguarda tutto il ciclo di vita del software. Inizia quando le attività di *pianificazione di sistema* hanno determinato la strategia di business per l'organizzazione e i progetti software sono stati identificati. La *pianificazione di progetto* è un'attività di valutazione di risultati del progetto, costi, tempi, rischi, scadenze, e risorse da assegnare al progetto stesso. Comprende anche la scelta di metodi di sviluppo, strumenti, standard, tecniche di organizzazione dei gruppi di lavoro, etc.

La pianificazione di progetto è un obiettivo in continua evoluzione. Non è qualcosa che è fatto una volta e mai modificato. Ci sono pochi *vincoli fissi*, i piani di progetto evolvono con il ciclo di vita.

Vincoli tipici sono *tempo* e *denaro*: ogni progetto dovrebbe avere una precisa scadenza e un budget definito. Una delle prime attività nella pianificazione è stimare se il progetto sia fattibile rispettando tempi, costi e altri vincoli. Se è fattibile allora i vincoli sono documentati e possono essere modificati solo tramite un processo formale di approvazione.

La *fattibilità di un progetto* è stimata attraverso molti fattori (Hoffer *et al.*, 1999; Whitten e Bentley, 1998):

- *Fattibilità operazionale:* riesamina gli argomenti originariamente considerati nella pianificazione di sistema (quando il progetto venne identificato) e analizza quale sarà l'impatto del sistema proposto sulle strutture organizzative, sulle procedure e sul personale.
- *Fattibilità economica:* stima costi e benefici del progetto (è anche nota come analisi costi-benefici).
- *Fattibilità tecnica:* stima la possibilità di realizzare le soluzioni tecniche proposte e la disponibilità di capacità, esperienze e risorse per farlo.
- *Fattibilità temporale:* stima la ragionevolezza delle scadenze temporali del progetto.

Non tutti i vincoli sono noti o possono essere valutati all'inizio del progetto. Vincoli aggiuntivi (come vincoli legali, contrattuali, politici, e di sicurezza) possono essere scoperti durante la fase di analisi dei requisiti e devono essere in qualche modo presi in considerazione negli studi di fattibilità.

Dopo la stima di fattibilità, il *piano di progetto* sarà definito e costituirà la guida per la gestione del progetto e del processo. Gli elementi esaminati nel piano di progetto comprendono (Whitten e Bentley, 1998):

- Campo di azione del progetto;
- Compiti del progetto;
- Direzione e controllo del progetto;
- Gestione della qualità;
- Misurazione e metriche;
- Schedulazione del progetto;
- Allocazione delle risorse (persone, materiali, strumenti);
- Gestione delle persone.

1.3.9 Metriche nel ciclo di vita del software

Misurare non solo il tempo e lo sforzo di sviluppo ma anche altri parametri (*metriche*) è una parte importante nel processo di *gestione del progetto*. Spesso tale parte non è trattata nelle organizzazioni con basso livello di maturità di processo, ma il prezzo che si "paga" è alto. Senza una misura del passato, l'organizzazione non è capace di pianificare accuratamente il suo futuro.

Le metriche sono normalmente trattate nel contesto della *qualità e complessità del prodotto software* (Henderson-Sellers, 1996; Pressman, 1997). Esse sono usate per misurare fattori di qualità come la correttezza, l'affidabilità, l'efficienza, l'integrità, l'usabilità, la manutenibilità, la flessibilità e la testabilità. Per esempio, l'affidabilità del software può essere valutata misurando la frequenza e la complessità dei guasti, l'accuratezza dei risultati, la possibilità di recupero da guasti, etc.

Altrettanto importante è l'applicazione delle metriche per il monitoraggio dei risultati nelle differenti fasi del ciclo di vita. Ad esempio, metriche possono essere usate per stimare l'efficacia del processo e migliorare la qualità del lavoro.

Tipiche metriche che si applicano al *processo software* e che possono essere valutate nelle varie fasi del ciclo di vita sono (Schach, 1996):

- La volatilità dei requisiti (percentuale di requisiti modificati o scartati dalla fine della fase di analisi dei requisiti). Questo può riflettere la difficoltà di ottenere i requisiti dai clienti o può dipendere da una scarsa qualità nella documentazione.
- La dimensione del documento di specifica generato dallo strumento CASE, e altre metriche più dettagliate fornite dal repository dello strumento, come il numero di classi nel modello delle classi. Se valutate sui progetti passati di cui sono noti tempi e costi, queste metriche forniscono un utile insieme di dati per prevedere durata e sforzo del progetto in esame.
- Le statistiche sugli errori (quando furono introdotti nel prodotto, come furono scoperti e corretti) che possono far riflettere sulla qualità dei passi di revisione e di testing.
- Numero medio di test prima che un modulo sia considerato accettabile per l'integrazione e il rilascio ai clienti. Questo può far riflettere sulle procedure di debugging dei programmatori.

1.3.10 Testing nel ciclo di vita del software

Come la pianificazione di progetto o la valutazione delle metriche, il *testing* è un'attività relativa a tutto il ciclo di vita del software (non si tratta di una fase successiva all'implementazione, sarebbe troppo tardi!). Infatti, l'incremento dei costi di correzione di errori introdotti nelle fasi iniziali del ciclo di vita diventa insostenibile (Schach, 1996).

Le attività di testing dovrebbero essere accuratamente pianificate. I casi di test che definiscono i passi da intraprendere nel tentativo di "scoprire i guasti" nel prodotto software dovrebbero essere preparati dall'inizio.

I casi di test dovrebbero essere definiti per ciascun modulo funzionale o caso d'uso descritto nel documento dei requisiti. Mettere in relazione i casi di test con i casi d'uso stabilisce un legame di *tracciabilità* tra i test e i requisiti utente: per essere testabile, un elaborato software deve essere "tracciabile".

Ogni sviluppatore controlla i prodotti del proprio lavoro. Tuttavia, per essere efficace, il testing dovrebbe essere condotto da una terza parte. La responsabilità della qualità del prodotto può essere assegnata al gruppo di SQA che dovrebbe comprendere alcuni dei migliori sviluppatori dell'organizzazione.

Più accurato è il testing nelle fasi iniziali dello sviluppo, maggiore sarà il ritorno in termini economici e temporali. Requisiti, specifica, e qualunque documento (compreso il codice sorgente) riguardante il software prodotto possono essere controllati durante *revisioni formali* chiamate *walkthrough* (incontri di rassegna) e *ispezioni*.

Le revisioni sono riunioni accuratamente preparate che esaminano una parte della documentazione o del sistema. Un revisore incaricato studia un documento precedentemente disponibile e presenta varie questioni agli sviluppatori. Durante la riunione si discutono gli eventuali errori senza necessariamente individuare delle soluzioni. Premesso che le riunioni sono amichevoli, la sinergia di gruppo può condurre alla scoperta precoce e alla correzione di molti errori.

Una volta che i prototipi del software e le prime versioni del prodotto siano resi disponibili, può essere effettuato un *testing funzionale*. Ci sono due tipi di test funzionale:

- test delle specifiche (black-box testing).
- test del codice (white-box o glass-box testing).

Il *test delle specifiche* tratta un programma come una scatola nera di cui nulla è noto eccetto che fornendole degli ingressi essa produce delle uscite. Al programma vengono quindi sottoposti determinati ingressi e le uscite risultanti sono analizzate per verificare l'eventuale presenza di errori. Il test delle specifiche è particolarmente utile per scoprire requisiti mancanti.

Il *test del codice* analizza la logica del programma per determinare gli ingressi necessari per seguire i vari cammini di esecuzione del programma stesso. Il test del codice integra il test delle specifiche perché i due test tendono alla scoperta di differenti categorie di errore.

Lo sviluppo incrementale non solo implica l'integrazione incrementale dei moduli software ma anche un *test incrementale o di regressione*. Il test di regressione è la riesecuzione, su un modulo di software esteso, dei precedenti test col medesimo *insieme di dati*. L'ipotesi da verificare col test è che le funzionalità preesistenti del modulo non dovrebbero essere negativamente influenzate dall'estensione.

Il test di regressione può essere facilmente automatizzato memorizzando in un database i dati di ingresso e uscita dei test. La principale difficoltà è dovuta al fatto che lo sviluppo incrementale può estendere non solo la logica procedurale di un programma ma anche le strutture dati sottostanti, complicando in tal modo il confronto fra i risultati.

1.4 Approcci allo sviluppo del software

La "rivoluzione del software" ha introdotto sostanziali modifiche nel modo in cui i prodotti software funzionano. In particolare, il software è divenuto molto più *interattivo*, adattandosi dinamicamente alle richieste dell'utente.

La logica procedurale di un vecchio programma COBOL del passato era inflessibile e poco reattiva. Una volta iniziato, il programma era eseguito fino al completamento in un modo più o meno deterministico. Occasionalmente, il programma poteva richiedere alcune informazioni all'utente e seguire di conseguenza distinti cammini d'esecuzione. In generale, tuttavia, l'interazione con l'utente era limitata ad un numero prefissato di cammini d'esecuzione: il controllo era nel programma, non nell'utente!

Con l'avvento delle moderne interfacce utente grafiche (GUI) le cose sono notevolmente cambiate. I programmi di interfaccia sono guidati dagli eventi (*event-driven*) e sono eseguiti in modo imprevedibile e casuale in funzione degli eventi generati dall'utente attraverso una tastiera, un mouse, o altri dispositivi di ingresso. Dietro ogni evento, c'è un *oggetto* software che sa come gestire quell'evento nello stato corrente di esecuzione del programma. Una volta che la gestione è completata, il controllo ritorna all'utente.

Stili differenti di programmazione richiedono differenti approcci allo sviluppo del software. Il software convenzionale è stato ben supportato dal cosiddetto *approccio strutturato*. I moderni sistemi GUI richiedono la programmazione a oggetti e *l'approccio object-oriented* è il miglior modo per progettare tali sistemi.

1.4.1 Approccio strutturato

L'*approccio strutturato* allo sviluppo dei sistemi divenne uno standard "de facto" negli anni '80. Esso è basato su due tecniche: DFD (*Data Flow Diagrams*, diagrammi di flusso dei dati) per la modellazione di processi e ERD (*Entity Relationship Diagrams*, diagrammi entità-relazioni) per la modellazione dei dati.

L'approccio strutturato è *centrato sulle funzioni* e usa i DFD come elemento caratterizzante dello sviluppo. Recentemente, e come diretto risultato della popolarità del modello relazionale dei dati, l'approccio è divenuto più *centrato sui dati* con maggiore enfasi posta sugli ERD.

La combinazione di DFD e ERD permette la costruzione di modelli di analisi relativamente completi che rappresentano i dati e le funzioni del sistema a un livello di astrazione ragionevole e indipendente da considerazioni software/hardware. Il modello di analisi deve quindi essere trasformato in un modello di progetto espresso, generalmente, in termini di modello relazionale dei dati e implementato su un database relazionale.

L'approccio strutturato è basato su alcune caratteristiche (delle quali non tutte sono perfettamente allineate con la moderna ingegneria del software):

- L'approccio tende a essere *sequenziale ed evolutivo* piuttosto che iterativo e incrementale (cioè l'approccio non facilita lo sviluppo del progetto attraverso l'iterazione e la produzione incrementale del software).
- L'approccio tende a produrre soluzioni poco flessibili che rispondono a un insieme definito di funzioni ma risultano difficilmente estensibili in futuro.
- L'approccio parte "da zero" e non supporta il riuso di componenti preesistenti.

La natura sequenziale dell'approccio introduce il rischio di interpretare scorrettamente gli originari requisiti durante lo sviluppo. Il rischio è accresciuto dalla necessità di compromessi tra la semantica relativamente dichiarativa dei modelli di analisi e le soluzioni procedurali nei modelli di progetto e nel codice implementativo (i modelli di analisi sono semanticamente più "ricchi" dei modelli di progetto e implementazione).

1.4.2 Approccio object-oriented

L'*approccio object-oriented* allo sviluppo dei sistemi divenne popolare negli anni '90. Il comitato OMG ha approvato lo standard UML nel 1997.

Confrontato con l'approccio strutturato, l'approccio object-oriented è maggiormente *centrato sui dati* ed evolve attorno a modelli di classi. Nella fase di analisi, non è necessario definire le operazioni sulle classi, ma solo gli attributi. La crescente importanza dei *casi d'uso* in UML ha provocato un certo spostamento dell'enfasi dai dati alle funzioni.

Esiste attualmente la convinzione che gli sviluppatori usino l'approccio a oggetti a causa dei vantaggi "tecnici" dell'approccio, come l'astrazione, l'incapsulamento, il riuso, l'ereditarietà, lo scambio di messaggi, il polimorfismo, etc. Tali proprietà possono favorire la riusabilità del codice, un minor tempo di sviluppo, un incremento della produttività dei programmatori, una maggior qualità del software, etc., ma non è detto che tutto questo si realizzi sempre nella pratica. Nonostante ciò, oggi gli oggetti si usano e verranno usati ancora nel futuro per ragioni legate al nuovo stile di programmazione guidato da eventi e supportato dalle moderne interfacce utente.

Altre ragioni della popolarità dell'approccio object-oriented riguardano la possibilità di contrastare più efficacemente l'accumularsi di progetti inevasi (application backlog) in campi che stanno assumendo un ruolo sempre più importante. Esempi di nuove categorie di applicazioni che richiedono la tecnologia a oggetti sono il *lavoro cooperativo* e i *sistemi multimedia*.

Lo sviluppo di sistemi con l'approccio object-oriented si adatta bene a un processo *iterativo e incrementale*. Un singolo modello (o un singolo documento di progetto) è "elaborato" attraverso le fasi di analisi, progetto e implementazione, con l'introduzione di maggiori dettagli in iterazioni successive, rilasciando versioni incrementali di moduli onde mantenere alta la soddisfazione dell'utente e permettere ritorni d'informazione per altri moduli in sviluppo.

Tutto questo è possibile perché tutti i modelli nelle varie fasi di sviluppo (analisi, progetto e implementazione) sono semanticamente ricchi e basati sul medesimo "linguaggio", e il vocabolario è essenzialmente lo stesso (classi, attributi, metodi, ereditarietà, polimorfismo etc.). Si noti, comunque, che se l'implementazione è basata su un database relazionale, è ancora necessaria una

complessa e rischiosa trasformazione (poiché la semantica del modello relazionale è piuttosto povera rispetto a quella dei modelli a oggetti).

L'approccio object-oriented risponde ai problemi presentati dall'approccio strutturato ma introduce alcuni nuovi problemi:

- La fase di analisi è condotta su un modello a livello di astrazione elevato e se l'implementazione è prevista su un server relazionale il *salto semantico* tra un concetto e la sua implementazione diventa significativo e può causare problemi. Se la piattaforma di implementazione è costituita da un database a oggetti, o *object-relational*, la trasformazione dei risultati di analisi e progetto è molto più semplice.
- La gestione del progetto è più complessa. I responsabili valutano lo sviluppo del progetto per mezzo di ben definite strutture di suddivisione del lavoro, rapporti di progetto, e scadenze. Nello sviluppo a oggetti non ci sono chiare frontiere tra le fasi e la documentazione di progetto evolve continuamente. Una soluzione al problema è legata alla suddivisione del progetto in "piccoli" moduli e alla gestione dell'avanzamento per mezzo di frequenti versioni eseguibili (alcune versioni possono essere interne, altre possono essere rilasciate ai clienti).
- Un ulteriore problema riguarda l'accresciuta *complessità* delle soluzioni che, a sua volta, influenza la manutenibilità e la scalabilità del software.

Le difficoltà nell'approccio object-oriented non cambiano comunque il fatto che le nuove applicazioni software sono molto più complesse da costruire e l'approccio strutturato è chiaramente inadeguato. L'approccio object-oriented è attualmente il solo metodo noto utilizzabile nello sviluppo del nuovo software altamente interattivo.

Sommario

In questo capitolo abbiamo passato in rassegna gli aspetti strategici del processo di sviluppo del software. Per alcuni lettori il contenuto di questo capitolo può risultare piuttosto scontato. Per i lettori con una certa esperienza di sviluppo del software, il capitolo potrebbe comunque fornire alcuni elementi di riflessione. Per tutti i lettori, lo scopo era di servire da introduzione alle discussioni più complete che verranno presentate in seguito.

La *natura* dello sviluppo del software è essenzialmente *artistica*. Il risultato di un progetto software non può essere completamente determinato all'origine. La principale *difficoltà accidentale* nello sviluppo del software riguarda gli *stakeholder*, un prodotto software deve portare beneficio alle persone interessate, altrimenti fallirà. Il *triangolo per il successo* comprende, oltre al fattore stakeholder, un *processo di sviluppo* ben fondato e il supporto di *linguaggi e strumenti di modellazione*.

Lo sviluppo del software riguarda la fornitura di un prodotto *efficiente*. La *pianificazione di sistema* precede lo sviluppo del software e determina quali prodotti possono essere i più *efficaci* per l'organizzazione. Ci sono vari modi in cui la pianificazione di sistema può essere condotta. Quattro di questi approcci sono stati discussi: SWOT, VCM, BPR e ISA. I sistemi che supportano il livello decisionale *strategico* promettono la maggior efficacia ma sono anche quelli che creano le maggiori sfide agli sviluppatori di software.

Lo sviluppo del software segue un *ciclo di vita*. Questo libro è centrato su due fasi del ciclo di vita: l'*analisi* e il *progetto*. Le altre fasi comprendono l'implementazione, l'integrazione e la manutenzione. Lo sviluppo del software riguarda inoltre altre importanti attività come la pianificazione di progetto, la raccolta di metriche, il testing e la gestione del cambiamento; non sono state considerate come fasi separate perché esse devono permeare l'intero ciclo di vita.

In passato, i prodotti software erano *procedurali*; una procedura programmata eseguiva le sue attività in modo più o meno sequenziale e deterministico, e quindi terminava. L'*approccio strutturato* allo sviluppo poteva essere usato con successo nella produzione di tali sistemi.

I prodotti software moderni sono intrinsecamente *object-oriented*, un programma consiste di oggetti che sono eseguiti in modo casuale e imprevedibile, e il programma non termina fino a quando non viene chiuso dall'utente. Gli oggetti attendono il verificarsi di eventi generati dall'utente per iniziare una computazione, possono richiedere dei servizi ad altri oggetti per completare la propria attività e quindi tornano in attesa di altri eventi generati dall'utente o di richieste da altri oggetti. Le moderne applicazioni dell'IS, di tipo client/server e con interfaccia GUI, sono tipicamente object-oriented e l'*approccio object-oriented* allo sviluppo è il più adeguato per la produzione di tali applicazioni. Il seguito del libro è centrato quindi su questo approccio.

Domande

D1 Sulla base delle vostre esperienze con prodotti software, come interpretereste l'osservazione di Fred Brooks che l'essenza dell'ingegneria del software è determinata dalla complessità software? In cosa l'ingegneria del software differisce dall'ingegneria tradizionale, quale l'ingegneria civile o meccanica?

D2 È stato ipotizzato che la produzione del software sia un'arte. Questa osservazione può essere completata con la citazione: "L'arte è una collaborazione fra Dio e l'artista, e meno fa l'artista, meglio è" (André Gide). Quale lezione, se è il caso, possono trarre gli sviluppatori del software da questa citazione? Siete d'accordo con essa?

D3 Spiegare la differenza fra pacchetti di programmi (package) e componenti. Quali pensate siano le future prospettive per queste due tecnologie?

D4 Si ricordi la definizione di stakeholder. Un fornitore di software o una persona tecnica di supporto possono essere uno stakeholder? Spiegare.

D5 Quale livello CMM di maturità è necessario affinché l'organizzazione possa rispondere con successo a una situazione di crisi? Spiegare.

D6 Nello spiegare l'approccio SWOT alla pianificazione di sistema è stato rilevato che "la missione dell'organizzazione dovrebbe porre l'enfasi sulle necessità del cliente piuttosto che sui prodotti o sui servizi che essa fornisce". Spiegare ed esemplificare come l'indicazione su prodotti e servizi nella missione dell'organizzazione possa non soddisfare l'obiettivo di efficacia nella pianificazione di sistema.

D7 Il BPR fa una distinzione netta fra processo di business e funzione di business. Qual è questa distinzione? Fornire un esempio di un processo di business che attraversa orizzontalmente un'organizzazione.

D8 Confrontare i concetti della catena di valore e di processo di business.

D9 Perché la comprensione dell'architettura ISA è importante per lo sviluppo del sistema?

D10 Che cosa sono i tre livelli di gestione? Considerate un'applicazione bancaria che controlla i modi d'uso di una carta di credito da parte del suo proprietario al fine di bloccare automaticamente la carta quando la banca ritiene fondato il sospetto di un abuso (furto, frode, etc.). Di quale livello di gestione fa parte una tale applicazione? Spiegarne i motivi.

D11 Spiegare la differenza fra le fasi di determinazione dei requisiti e di specifica dei requisiti.

D12 Spiegare il rapporto delle due fasi di progetto (progetto architetturale e progetto dettagliato) con le prime due fasi del ciclo di vita, la determinazione dei requisiti e la specifica dei requisiti.

D13 Come interpretate il fatto che un sistema object-oriented dovrebbe essere progettato per l'integrazione?

D14 La pianificazione di sistema e le metriche del software sono inerentemente correlate. Spiegare questa supposizione.

D15 Spiegare il rapporto fra la tracciabilità e il testing.

D16 Quali sono le principali tecniche di modellazione nell'approccio strutturato allo sviluppo del software?

D17 Quali sono le ragioni principali per uno spostamento dall'approccio strutturato a quello object-oriented?

Capitolo 2

Fondamenti di analisi dei requisiti

Questo libro tratta la produzione di software object-oriented: per questo è necessaria una profonda comprensione della tecnologia a oggetti, senza la quale uno sviluppatore non sarà in grado di usare UML correttamente come linguaggio di modellazione.

La difficoltà principale, nell'apprendimento della tecnologia a oggetti, è legata all'assenza di un evidente punto di partenza e alla mancanza di un ben definito cammino di ricerca. Non esistono approcci top-down o bottom-up nel senso comunemente noto. Per necessità, l'approccio deve essere una sorta di "dal centro verso l'esterno". Non importa quanto si avanzi nel processo d'apprendimento, con l'emergere di nuovi elementi sembrerà sempre di essere a "metà". Una prima verifica, che segnala un progresso indicativo nell'apprendimento, si ha quando il lettore comprende bene che in un sistema object-oriented "ogni cosa è un oggetto".

Questo capitolo cercherà di fornire un giusto livello informativo al lettore prima di approfondire l'argomento. Esso riguarderà le premesse per l'analisi dei requisiti, considerandoli da un duplice punto di vista. Anzitutto, saranno spiegati i fondamenti della tecnologia a oggetti, quindi, seguendo il metodo dell'insegnamento per esempi (*teach-by-example*), sarà presentata l'analisi dei requisiti per un familiare caso di studio, gli Acquisti OnLine via Internet.

2.1 Fondamenti della tecnologia a oggetti

Un buon modo di spiegare l'orientamento a oggetti nei sistemi informativi è di presentare un'analogia con gli oggetti concreti del mondo reale. Il mondo che ci circonda consiste di *oggetti* che possono trovarsi in uno *stato* particolare, definito dai valori degli *attributi* degli oggetti stessi. Ad esempio, una tazza da caffè sulla scrivania è in uno stato `pieno` perché è in grado di contenere liquidi e contiene ancora del caffè. Quando non c'è più caffè, lo stato della tazza è `vuoto`. Se la tazza cade e si rompe, il suo stato è `rotto`.

La tazza da caffè è piuttosto passiva, non ha un *comportamento* proprio. Invece, la stessa passività non vale per un cane o per un albero. Il cane abbaia, l'albero cresce etc., quindi alcuni oggetti del mondo reale mostrano un comportamento.

Tutti gli oggetti del mondo reale possiedono un'*identità*, una proprietà immutabile attraverso cui un oggetto si distingue da un altro. Se avessimo avuto sulla scrivania due tazze da caffè di uno stesso insieme di tazze, avremmo potuto affermare che le due tazze sono *uguali* ma non sono *identiche*. Sono uguali perché hanno gli stessi valori per tutte le loro proprietà (la stessa dimensione e forma, sono bianche e sono vuote). Nel gergo dell'object-oriented, le due tazze non sono identiche poiché sono due ed è necessario scegliere quale usare.

Gli oggetti del mondo reale possiedono tre caratteristiche (stato, comportamento, identità) che permettono di definirli come *sistemi a comportamento naturale*. I sistemi naturali sono da sempre i sistemi più *complessi* che noi conosciamo, nessun sistema computerizzato ha raggiunto la complessità di un animale o di una pianta. I sistemi naturali mostrano dei comportamenti interessanti, possono adattarsi a modifiche interne ed esterne, possono evolvere nel tempo etc. La lezione è evidente: forse per costruire *sistemi artificiali* bisognerebbe emulare la struttura e il comportamento di quelli naturali (vedi Maciaszek *et al.*, 1996b).

I sistemi artificiali sono modelli della realtà. Una tazza da caffè sullo schermo di un computer è un modello della "cosa" reale, come lo è un cane o un albero sullo schermo. Una tazza da caffè può essere rappresentata anche con proprietà comportamentali. Può, per esempio, cadere sul pavimento se urtata, rompendosi quando tocca il suolo. L'azione "cadere" può essere rappresentata come un'*operazione* (proprietà comportamentale) del modello "tazza". La maggior parte degli oggetti (se non tutti) in un sistema computerizzato "sono vivi", mostrano cioè un comportamento.

2.1.1 Oggetto istanza

Un oggetto è un'istanza di una "cosa", e può essere una delle molte istanze della stessa "cosa". La mia "tazza da caffè" è un'istanza nell'insieme delle possibili tazze.

Una generica descrizione di una "cosa" è chiamata *classe*. Quindi, un oggetto è un'istanza di una classe. Ma, come vedremo nella Sezione 2.1.6, anche una classe può essere il risultato di un'istanziazione, cioè essa stessa può essere un oggetto. Per tale ragione, è necessario distinguere tra un *oggetto istanza* e un *oggetto classe*. Ogni cosa in un sistema object-oriented è un oggetto, sia esso un oggetto istanza sia esso un oggetto classe, e un sistema object-oriented consiste di oggetti che collaborano.

Per brevità, un oggetto istanza è generalmente chiamato *oggetto* o *istanza*. È fuorviante chiamare un oggetto: istanza oggetto. In conseguenza della discussione precedente, non sarà sostenuto l'uso del termine "classe oggetto". Vero è che una classe è uno "stampo" per oggetti aventi i medesimi attributi e operazioni, ma una classe può essere a sua volta frutto di un'istanziazione (e non è piacevole chiamare il risultato di tale creazione "oggetto classe oggetto").

2.1.1.1 Notazione per gli oggetti

La notazione UML per un oggetto è un rettangolo suddiviso in due aree separate. L'area soprastante racchiude il nome di un oggetto e il nome della classe dalla quale l'oggetto proviene. La sintassi è:

`nomeoggetto: nomeclasse.`

L'area sottostante contiene la lista dei nomi di attributi e i relativi valori. Il tipo di attributo può essere riportato usando la sintassi:

`nomeattributo: tipo = valore.`

La Figura 2.1 mostra un oggetto, chiamato `c1`, della classe `Corso`. L'oggetto ha due attributi i cui tipi non sono mostrati, essendo specificati nella definizione della classe.

FIGURA 2.1 Oggetto istanza.

È importante osservare che la notazione per gli oggetti non prevede un'area per elencare le *operazioni* che l'oggetto può eseguire. Questo perché le operazioni sono uguali per tutti gli oggetti e sarebbe ridondante ripeterle in ciascun oggetto. Le operazioni possono essere inserite nella *classe* o essere associate agli oggetti con altri meccanismi (implementati nel sottostante sistema software object-oriented).

2.1.1.2 Come collaborano gli oggetti?

Il numero di oggetti di una classe può essere molto grande, per cui non è né pratica né realizzabile la visualizzazione di molti di essi in un diagramma. Gli oggetti sono rappresentati solo per esemplificare un sistema in un certo momento o per mostrare come *collaborano* nel tempo per eseguire certi lavori. Ad esempio, per ordinare dei prodotti, una *collaborazione* tra un oggetto `Magazzino` e un oggetto `Acquisto` potrebbe rendersi utile.

Le attività del sistema sono eseguite da insiemi di oggetti che invocano delle *operazioni* su altri oggetti scambiandosi dei *messaggi*. I messaggi attivano operazioni degli oggetti che possono dar luogo a modifiche di stato e invocare a loro volta altre operazioni.

La Figura 2.2 mostra il flusso di messaggi tra quattro oggetti. Le parentesi dopo i nomi di messaggio indicano che il messaggio accetta dei parametri (come nelle chiamate di funzione della programmazione tradizionale). L'oggetto `Ordine` chiede all'oggetto `Spedizione` di consegnare la

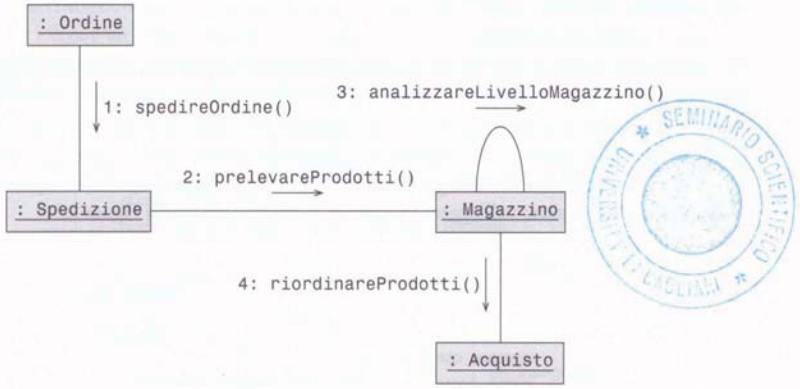


FIGURA 2.2 Collaborazione fra oggetti.

merce ordinata. Per fare questo, `Spedizione` chiede all'oggetto `Magazzino` di sottrarre le appropriate quantità di prodotti. L'oggetto `Magazzino` analizza quindi i nuovi livelli d'inventario e, se si è sotto una determinata soglia, chiede all'oggetto `Acquisto` di riordinare altri prodotti.

Sebbene sia stata illustrata la collaborazione tra oggetti come una sequenza di attività e i messaggi siano stati numerati, in generale, il flusso dei messaggi non impone uno stretto ordine temporale sull'attivazione degli oggetti. Ad esempio, `analizzareLivelloMagazzino` e `riordinaProdotto` possono eseguirsi in qualunque ordine, eventualmente in modo indipendente da `spedireOrdine` e `prelevareProdotti`. Per tali ragioni, sarà abbandonata la numerazione dei messaggi quando si discuterà la collaborazione tra oggetti a livello di progetto (Sezione 6.2).

2.1.1.3 Come un oggetto identifica altri oggetti?

La domanda riguarda il modo in cui un oggetto conosce l'*identità* di un altro oggetto al quale vuole inviare un messaggio. Come un oggetto `Ordine` conosce il "suo" oggetto `Spedizione` in modo che il messaggio `spedireOrdine` raggiunga la destinazione voluta?

La risposta è che a ogni oggetto è assegnato un OID (*Object Identifier*, identificatore di oggetto) all'atto della sua creazione. L'OID è un *riferimento*, un numero unico che rimane associato all'oggetto per la sua intera esistenza. Se un oggetto X vuol inviare un messaggio a un oggetto Y, deve in qualche modo conoscere l'OID di Y.

Ci sono due soluzioni pratiche per stabilire *legami* OID tra oggetti. Le soluzioni sono:

- legami OID persistenti,
- legami OID transitori.

La distinzione tra questi due tipi di legame ha a che vedere con la *longevità* degli oggetti. Alcuni oggetti vivono solo durante l'esecuzione di un'applicazione: sono creati dall'applicazione e distrutti durante o al termine dell'esecuzione della stessa. Questi sono oggetti *transitori*. Altri oggetti sopravvivono all'esecuzione dell'applicazione; quando l'applicazione termina sono memorizzati su disco, cioè in memoria persistente, e sono disponibili per la prossima esecuzione dell'applicazione. Questi sono oggetti *persistenti*.

2.1.1.3.1 Legami persistenti

Un legame *persistente* è un riferimento (o un insieme di riferimenti) contenuto in un oggetto della memoria persistente che lega tale oggetto a un altro sempre contenuto nella memoria persistente (o a un insieme di altri oggetti). Di conseguenza, per legare in modo persistente un oggetto di `Corso` al suo oggetto di `Insegnante`, l'oggetto di `Corso` deve contenere un attributo di legame il cui valore sia l'OID dell'oggetto di `Insegnante`. Il legame è persistente poiché l'OID è fisicamente memorizzato nell'oggetto di `Corso`, come mostrato nella Figura 2.3.

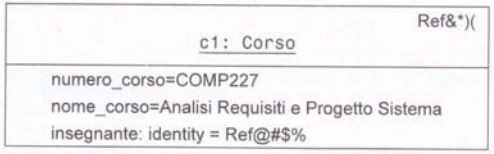


FIGURA 2.3 Implementazione di un legame persistente.

L'OID dell'oggetto `c1` è mostrato come `Ref&*)(`. L'oggetto contiene un attributo di legame chiamato `insegnante`. Il tipo di tale attributo è `identity`. Il suo valore è il numero "magico" `Ref@#$%`, indirizzo del disco dove l'oggetto di `Insegnante` (che insegna nel corso `c1`) è memorizzato in modo *persistente*.

Una volta che gli oggetti di `Corso` e di `Insegnante` sono trasferiti nello spazio di lavoro dell'applicazione il valore dell'attributo `insegnante` sarà sostituito (*swizzling*, scambio dei puntatori) da un puntatore in tale spazio di lavoro, stabilendo così una collaborazione in memoria tra gli oggetti (*swizzling* non è un termine UML, è usato nei database a oggetti, si veda Sezione 8.2, dove sono frequenti i trasferimenti di oggetti tra la memoria persistente e quella locale).

La Figura 2.3 illustra come i legami persistenti sono generalmente implementati. Tuttavia, in UML i legami (*link*) sono indicati come in Figura 2.4: i legami sono *istanze* di un'*associazione* fra gli oggetti di `Corso` e `Insegnante`.

Normalmente, i legami permettono la navigazione in una qualunque delle due direzioni. Ogni oggetto di `Corso` è legato al proprio oggetto di `Insegnante` e da un oggetto di `Insegnante` si può navigare verso il corrispondente oggetto di `Corso`. È possibile, sebbene non frequente, permettere la navigazione in una sola direzione.

Ad esempio, un *messaggio* inviato da un oggetto di `Insegnante` per ottenere il nome di un oggetto di `Corso` potrebbe essere simile al seguente:

```
rif_crs.ottieniNomeCorso(out nome_crs)
```

Nell'esempio, l'oggetto di `Corso` che eseguirà l'operazione `ottieniNomeCorso` (implementata nella classe `Corso`) è riferito dal valore corrente della variabile `rif_crs`. Alla variabile `nome_crs` sarà assegnato il valore ottenuto dall'operazione `ottieniNomeCorso` specificata nella classe `Corso`.

2.1.1.3.2 Legami transitori

Cosa sarebbe accaduto se non avessimo definito un legame persistente tra `Corso` e `Insegnante`? Avremmo ancora potuto inviare un messaggio dall'insegnante `t1` al corso `c1` per invocare l'operazione `ottieniNomeCorso`? L'applicazione deve avere altri meccanismi per trovare l'identità dell'oggetto `c1` e creare un legame transitorio dall'oggetto `t1` all'oggetto `c1` (Riel, 1996).

Fortunatamente, un programmatore ha molte tecniche per inizializzare la variabile `rif_crs` con l'OID di un oggetto `c1` residente nella memoria locale. Tanto per iniziare, è possibile che in

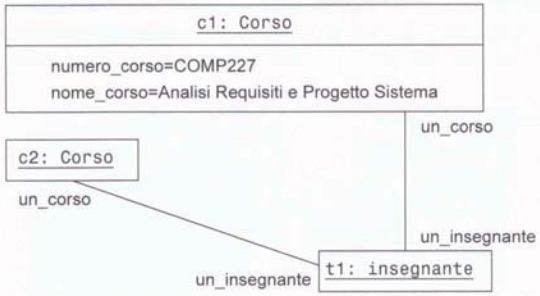


FIGURA 2.4 Legami persistenti nel modello UML degli oggetti.

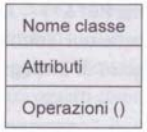


FIGURA 2.5. Aree della classe.

precedenza sia stato stabilito nell'applicazione un legame tra gli oggetti `c1` e `t1` e che `rif_crs` contenga ancora il corretto OID (ad esempio, l'applicazione aveva eseguito un'operazione di ricerca sulla disponibilità dell'insegnante `t1` e sull'orario dei corsi, decidendo che l'insegnante `t1` dovesse insegnare nel corso `c1`).

Una possibilità alternativa è che il programma abbia accesso a una tabella persistente che assegna i numeri di corso ai nomi degli insegnanti. Può ricercare tutti i corsi insegnati da `t1` e chiedere all'utente di scegliere il corso cui il messaggio `ottieniNomeCorso` deve essere inviato.

È anche possibile che l'obiettivo del programma sia di creare corsi e insegnanti prima di memorizzarli in un database. Non esistono legami persistenti tra corsi e insegnanti ma l'utente inserisce le informazioni necessarie affinché ogni corso identifichi chiaramente l'insegnante incaricato. Il programma può allora memorizzare i legami transitori in variabili (come `rif_crs`) che saranno usate nel seguito dell'esecuzione per inviare messaggi tra gli oggetti di `Insegnante` e di `Corso`.

In breve, ci sono tecniche di programmazione per stabilire legami transitori tra oggetti che non siano stati legati in modo persistente attraverso associazioni tra classi. I legami transitori sono variabili che contengono, come valori, OID di oggetti, oggetti momentaneamente nella memoria locale dell'applicazione. La mappatura (*swizzling*) tra OID persistenti e transitori dovrebbe essere responsabilità del sottostante ambiente di programmazione, come un DBMS (*DataBase Management System*, sistema di gestione di database) a oggetti.

2.1.2 Classe

Una *classe* è il descrittore di un insieme di oggetti con gli stessi attributi e operazioni. Le *operazioni* e gli *attributi* (con i rispettivi *tipi*) sono definiti nella classe che può anche essere considerata come uno "stampo" per la creazione dei suoi oggetti. Ogni oggetto creato dallo stampo può contenere valori degli attributi conformi ai tipi definiti, e può invocare operazioni.

Graficamente, una classe è rappresentata come un rettangolo suddiviso in tre aree distinte non sovrapposte, come mostrato in Figura 2.5. L'area soprastante contiene il nome della classe. L'area centrale contiene le dichiarazioni degli attributi della classe. L'area sottostante contiene la definizione delle operazioni.

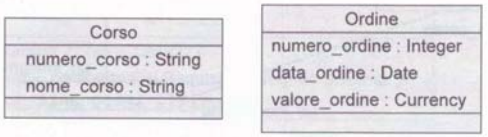


FIGURA 2.6 Attributi.

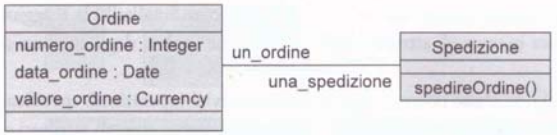


FIGURA 2.7 Nomi di ruolo che indicano classi (modello di analisi).

2.1.2.1 Attributo

Un attributo è una coppia *tipo-valore*. Le classi definiscono il tipo degli attributi e gli oggetti contengono il valore degli attributi stessi. La Figura 2.6 presenta due classi con attributi e relativi nomi e tipi.

Un tipo per l'attributo può essere un tipo *primitivo* o può essere un'altra *classe*. Un tipo primitivo è un tipo reso direttamente disponibile dal sottostante ambiente software object-oriented. Tutti i tipi degli attributi nella Figura 2.6 sono primitivi.

2.1.2.1.1 Classi come tipo di attributo

Una classe può essere usata come tipo per un attributo. Un oggetto della classe in cui un tale attributo è definito contiene come valore dell'attributo l'OID di un oggetto della classe tipo dell'attributo in questione. Nei modelli UML, gli attributi con tipo basato su classi (piuttosto che tipo primitivo) non sono elencati nell'area centrale: per rappresentarli si usano le *associazioni* tra classi. La Figura 2.7 mostra un'associazione tra classi.

I due nomi sulla linea che rappresenta l'associazione (`una_spedizione` e `un_ordine`) sono i cosiddetti *nomi dei ruoli*. Un ruolo identifica il significato che la relativa classe riveste nell'associazione e viene usato per navigare verso un oggetto dell'altra classe.

Nell'implementazione, il nome di ruolo diventa un attributo della classe opposta, il cui tipo è la classe relativa al ruolo. La Figura 2.8 mostra la forma che le due classi di Figura 2.7 potrebbero assumere in una loro possibile implementazione.

2.1.2.1.2 Visibilità degli attributi

Come osservato nella Sezione 2.1.1.2, gli oggetti collaborano inviando messaggi ad altri oggetti. Un messaggio invoca un'operazione della classe e l'operazione fornisce il servizio richiesto accedendo ai valori degli attributi dell'oggetto. Affinché questo sia possibile, le operazioni devono essere visibili agli oggetti esterni (i messaggi devono vedere le operazioni). Tali operazioni devono quindi avere una *visibilità pubblica*.

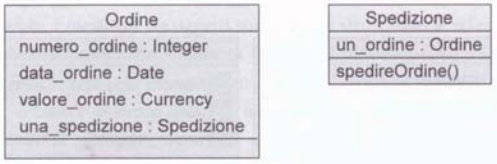


FIGURA 2.8 Classi come tipo di attributi (modello d'implementazione).

In un sistema object-oriented puro (come nell'ambiente Smalltalk) la maggior parte delle operazioni è pubblica mentre gli attributi sono *privati*, cioè i valori degli attributi sono nascosti alle altre classi.

Gli oggetti di una classe possono solo richiedere servizi (operazioni) specificati nell'interfaccia pubblica di un'altra classe e non possono manipolare direttamente gli attributi di altri oggetti. Si può affermare che le operazioni *incapsulano* gli attributi. Si noti, comunque, che l'incapsulamento si applica alle classi.

La visibilità è normalmente indicata con un simbolo + o – :

+ per indicare la visibilità pubblica,

– per indicare la visibilità privata.

Questi simboli sono sostituiti in alcuni strumenti CASE da icone grafiche. La Figura 2.9 mostra due rappresentazioni grafiche per evidenziare la visibilità degli attributi.

2.1.2.2 Operazione

Un oggetto contiene dati (attributi) e algoritmi (operazioni) che manipolano tali dati. Un'operazione è dichiarata in una classe e la procedura che implementa l'operazione è chiamata *metodo*.

Un'operazione (o metodo, per essere precisi) è invocata da un messaggio (il nome del messaggio e il nome dell'operazione sono gli stessi). L'operazione può contenere una lista di parametri in ingresso cui saranno assegnati valori specifici nella chiamata e può restituire un valore all'oggetto chiamante.

Il nome di operazione insieme alla lista dei tipi dei parametri formali è chiamata la *segnatura* (signature) di un'operazione. La segnatura deve essere unica all'interno di una classe. Questo significa che una classe può avere più operazioni con lo stesso nome, purché la lista dei tipi dei parametri formali sia sempre diversa.

2.1.2.2.1 Operazioni come supporto alla collaborazione tra oggetti

Un'applicazione object-oriented opera reagendo agli eventi casuali generati dall'utente. Gli eventi provengono dalla tastiera, dal mouse, dalle voci di menu, dai bottoni di azione e da qualunque altro tipo di dispositivo di ingresso. Un *evento* generato dall'utente viene convertito in un *messaggio* verso un *oggetto*. Per raggiungere un obiettivo dell'applicazione molti oggetti possono essere chiamati a collaborare, invocando le operazioni di altri oggetti (si veda la Sezione 2.1.1.2).

La Figura 2.10 mostra le operazioni delle classi necessarie per realizzare la collaborazione tra oggetti presentata in Figura 2.2. Ogni messaggio nella Figura 2.2 richiede un'operazione della classe indicata nella destinazione del messaggio. La classe `Ordine` non ha nessun'operazione in questo semplice esempio. Un oggetto della classe `Ordine` inizia una collaborazione tra oggetti ri-

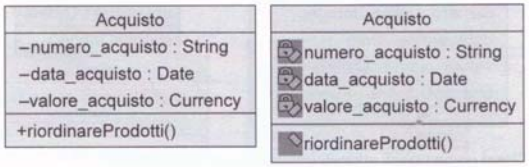


FIGURA 2.9 Attributi privati e operazioni pubbliche.

FIGURA 2.10 Operazioni a supporto di una collaborazione d'oggetti.

chiedendo a un oggetto della classe `Spedizione` di eseguire la consegna vera e propria della merce. Come risultato della spedizione, il magazzino potrebbe necessitare di un riapprovvigionamento della merce.

2.1.2.2.2 Visibilità e applicabilità di un'operazione

La *visibilità* delle operazioni segue le stesse regole di visibilità degli attributi (si veda la Sezione 2.1.2.1.2). La visibilità di un'operazione determina se un'operazione è visibile agli oggetti di classi diverse da quella in cui è stata dichiarata. Se è visibile allora la sua visibilità è *pubblica*, altrimenti è *privata*. Le icone davanti ai nomi delle operazioni nella Figura 2.10 denotano una visibilità pubblica.

Perché un oggetto possa fornire un servizio al mondo esterno le sue operazioni devono essere visibili, per cui nei sistemi object-oriented la visibilità è generalmente pubblica. Tuttavia molti oggetti hanno anche problemi di gestione interna; pertanto le relative operazioni devono essere private e accessibili solo agli oggetti della classe in cui sono definite.

La visibilità di un'operazione deve essere distinta dall'*applicabilità* (*scope*) dell'operazione. L'operazione può essere invocata su un oggetto istanza (Sezione 2.1.1) o può essere invocata su un oggetto classe (Sezione 2.1.6). Nel primo caso, si parla di applicabilità a livello di istanza dell'operazione. Nel secondo caso, di applicabilità a livello di classe. Ad esempio, l'operazione che trova l'età di un impiegato ha un'applicabilità a livello di istanza mentre l'operazione che calcola l'età media di tutti gli impiegati ha un'applicabilità a livello di classe.

2.1.3 Associazione

Un'*associazione* è una tipologia di relazione tra classi. Altre tipologie di relazione comprendono: generalizzazione, aggregazione, dipendenza e alcune altre.

Un'associazione lega oggetti di classi. Gli oggetti che necessitano di comunicare possono usare tali legami. Tipicamente, i messaggi tra oggetti sono inviati sfruttando le relazioni di associazione.

La Figura 2.11 mostra la relazione chiamata `OrdSped` tra le classi `Ordine` e `Spedizione`. La relazione permette di legare uno stesso oggetto di classe `Ordine` a più di un oggetto di classe `Spedizione` (questo fatto è indicato dall'asterisco *). Anche un oggetto di classe `Spedizione` può trasportare (è legato a) più di un oggetto di classe `Ordine`.

razi
altr

pub
può
to s

+
–

Qu
due

2.1
Un
zior

gio
ing
chia

(sig
gnil
ram

2.1.
Un'
ti p
altr
gio
chia

ogg
clas
que

FIG

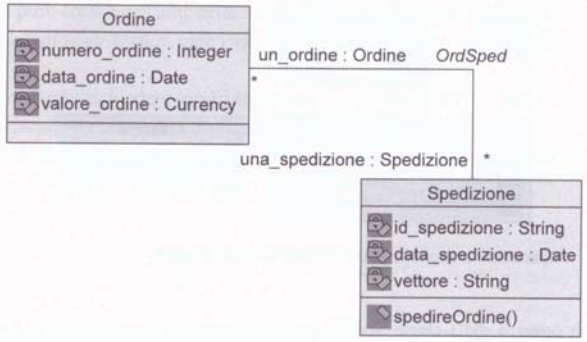


FIGURA 2.11 Associazione.

2.1.3.1 Grado di un'associazione

Il *grado* di un'associazione definisce il numero di classi da essa connesse. Generalmer associazioni di grado due, chiamate *associazioni binarie*. L'associazione nella Figura 2.1

Le associazioni possono essere definite anche su una singola classe, e sono in tal ca *associazioni unarie* o *ricorsive* (Maciaszek, 1990). Le associazioni ricorsive stabiliscon oggetti di una stessa classe.

La Figura 2.12 mostra un tipico esempio di associazione ricorsiva rappresentante gerarchica del personale. Un oggetto di `Impiegato` è `gestito_da` un altro oggett desima classe o da nessuno (ad esempio, l'Amministratore Delegato non è `gestito` no). Un oggetto di `Impiegato` è il `gestore_di` molti impiegati, fino al livello pi personale che non ha la responsabilità di alcun altro impiegato.

Anche le associazioni di grado tre (associazioni ternarie) sono possibili, sebbene non sia raccomandato (Maciaszek, 1990).

2.1.3.2 Molteplicità di un'associazione

La *molteplicità di un'associazione* specifica quanti oggetti di una classe (identificata d ruolo) possono essere associati con un singolo oggetto dell'altra classe partecipante all'a

La molteplicità è rappresentata come un intervallo di interi `n1..n2`. L'intero `n1` numero minimo di oggetti connessi mentre `n2` stabilisce il numero massimo (il nur mo può essere anche indicato con un asterisco * se non si è a conoscenza del valo Se non si vuole specificare il numero minimo ma si suppone che potrebbero essere co ti oggetti, semplicemente non si indica tale numero minimo (come in Figura 2.11).

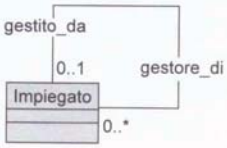


FIGURA 2.12 Associazione unaria.

Le molteplicità più frequenti sono:

`0..1`

`0..*`

`1..1`

`1..*`

`*`

La Figura 2.13 mostra due associazioni tra le classi `Insegnante` e `CorsoOfferto`. Un'associazione rappresenta l'assegnazione corrente degli insegnanti ai corsi attualmente offerti. L'altra determina quale insegnante è incaricato del corso. Un insegnante può insegnare in diversi corsi o nessuno (per esempio, se l'insegnante è in congedo per impegni di lavoro), e può essere incaricato di più corsi offerti o di nessuno. Uno o più insegnanti possono insegnare in un corso, ma uno e uno solo uno di questi è incaricato del corso.

In UML la molteplicità di associazione è un termine di cui si abusa. Le molteplicità minime "0" e "1" possono essere viste come una differente nozione semantica chiamata appartenenza (*membership*) o anche partecipazione (*participation*) (Maciaszek, 1990). La molteplicità minima "0" significa una partecipazione *opzionale* di un oggetto in un'associazione. La molteplicità minima "1" significa una partecipazione *obbligatoria*. Ad esempio, un oggetto di classe `CorsoOfferto` deve essere `gestito_da` un oggetto di classe `Insegnante`.

La proprietà di partecipazione ha altri interessanti aspetti semantici. Ad esempio, una partecipazione obbligatoria potrebbe ulteriormente implicare che l'appartenenza è *fissa*, cioè una volta che un oggetto è legato a un altro nell'associazione, non può essere riconnesso, in un momento diverso, a un altro nella stessa associazione.

2.1.3.3 Legame ed estensione di un'associazione

Un legame è un'istanza di associazione. In generale, un legame è una n-upla di riferimenti a oggetti. La n-upla può essere un *insieme* di riferimenti o una *lista* (insieme ordinato) di riferimenti. Spesso, la n-upla contiene un solo riferimento. L'*estensione* è un insieme di legami (e quindi è un insieme di insiemi).

La Figura 2.14 è una particolare istanziazione dell'associazione `OrdSped` della Figura 2.11. Ci sono cinque legami nella Figura 2.14. Quindi la cardinalità dell'estensione dell'associazione è cinque.

Comprendere il significato dei legami e dell'estensione di un'associazione è un elemento importante nella comprensione del concetto stesso di associazione.

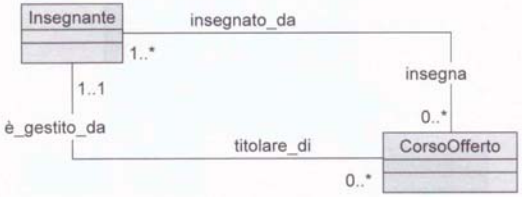


FIGURA 2.13 Molteplicità dell'associazione.

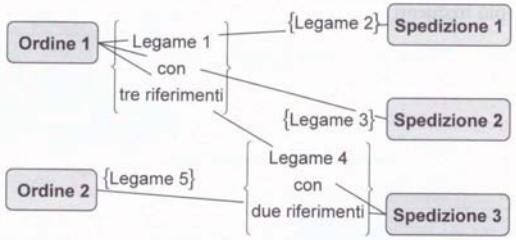


FIGURA 2.14 Legame ed estensione.

2.1.3.4 Classe di associazione

Se un'associazione possiede attributi (e/o operazioni) propri deve essere rappresentata co classe (poiché gli attributi possono essere definiti solo in una classe). Ogni oggetto di una *associazione* ha legami con gli oggetti delle classi associate. Poiché una *classe di associazion* classe, può essere associata ad altre classi nel modo consueto.

La Figura 2.15 mostra una classe di associazione `Valutazione`. Un oggetto di tal contiene una lista di punteggi, un punteggio totale e un giudizio ottenuti da uno `Stude` un `CorsoOfferto`.

Il tipo dell'attributo `punteggio` è `List(Number)`. Il tipo `List` è chiamato tipo *pa co*. `Number` è un parametro di `List`, dove `List` definisce un insieme ordinato di valori. buto `punteggio` contiene la lista di tutti i punteggi che uno studente ha ottenuto in u (ad esempio, se lo studente "Rossi" segue il corso "COMP227" la lista dei punteggi che ha ottenuto per quel corso sarà contenuta in un oggetto della classe `Valutazione` che senta il legame tra "Rossi" e "COMP227").

2.1.4 Aggregazione e composizione

L'*aggregazione* è una relazione tra una parte e l'insieme delle parti (*whole-part*). Essa si st tra una classe rappresentante la collezione dei componenti (*classe superset*) e le classi rappre ti i componenti stessi (*classi subset*), per cui si può dire che un oggetto composto *contiene* a getti, i suoi componenti.

La proprietà di contenimento può essere forte (*aggregazione per valore*) o debole (*aggre per referenza*). In UML l'aggregazione per valore è chiamata *composizione* mentre l'aggre per referenza è chiamata semplicemente *aggregazione*.

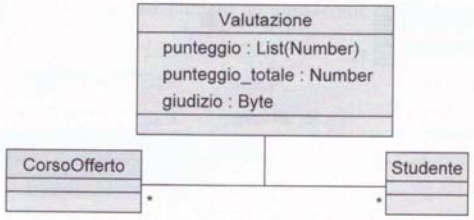


FIGURA 2.15 Classe d'associazione.

Dal punto di vista della modellazione del sistema, l'aggregazione è una tipologia speciale di associazione con una semantica aggiuntiva. In particolare, l'aggregazione è transitiva e asimmetrica. *Transitività* significa che se la classe A contiene la classe B e la classe B contiene la classe C, allora anche A contiene C. *Asimmetricità* significa che se A contiene B allora B non può contenere A.

La composizione possiede un'ulteriore proprietà di *dipendenza esistenziale*. Un oggetto di una classe subset non può esistere senza che sia legato a un oggetto della classe superset. Ciò implica che se un oggetto della classe superset viene distrutto allora devono essere distrutti tutti gli oggetti delle classi subset a esso associati.

Per rappresentare la composizione si usa graficamente un *rombo pieno* posto al termine della linea di associazione dalla parte della classe superset. Un'aggregazione che non sia una composizione è rappresentata con un *rombo vuoto*. Si noti che il rombo vuoto può essere anche usato per non dover fare in un certo momento un'esplicita scelta tra aggregazione e composizione (l'aggregazione potrebbe essere successivamente raffinata in una composizione).

La Figura 2.16 mostra una composizione sulla sinistra e una normale aggregazione sulla destra. Qualunque oggetto di classe `Libro` è una composizione di oggetti di classe `Capitolo` a loro volta composti da oggetti di classe `Sezione`. Un oggetto di classe `Capitolo` non può avere un'esistenza indipendente; esiste solo all'interno di un oggetto di classe `Libro`. Lo stesso non può essere affermato per un oggetto di classe `BottigliaBirra` che può esistere indipendentemente dal suo contenitore, un oggetto di classe `Cassa`.

2.1.5 Generalizzazione

La *generalizzazione* è una relazione "tipo-di" (kind-of) tra una classe più generica (*superclasse* o *genitore*) e una classe più specifica della precedente (*sottoclasse* o *figlio*). La sottoclasse è una tipologia della superclasse. Un oggetto della sottoclasse può essere usato ovunque un oggetto della superclasse è permesso.

La generalizzazione permette di non ridefinire delle proprietà già definite. Gli attributi e le operazioni già introdotti nella superclasse possono essere *riusati* nella sottoclasse. Si dice che una sottoclasse *eredita* gli attributi e i metodi della classe genitore. Osserviamo che anche la generalizzazione è transitiva e asimmetrica. Essa facilita la specifica incrementale, l'uso di proprietà comuni tra classi e una migliore localizzazione delle modifiche.

FIGURA 2.16 Composizione e aggregazione.

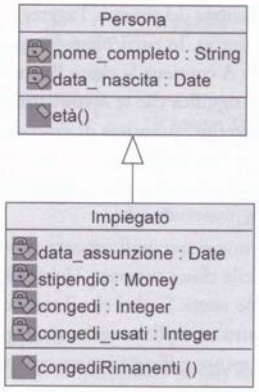


FIGURA 2.17 Generalizzazione.

Una generalizzazione viene graficamente rappresentata come un triangolo vuoto posto al termine della relazione dalla parte della superclasse. Nella Figura 2.17, `Persona` è superclasse e `Impiegato` è la sottoclasse, per cui `Impiegato` eredita tutti gli attributi e operazioni della classe `Persona`. Le proprietà ereditate non vengono mostrate nella sottoclasse (la generalizzazione "forza" l'ereditarietà).

Si noti che di ereditarietà si parla in relazione alle classi, non agli oggetti (si applica ai tipi, non ai valori). La classe `Impiegato` eredita la definizione degli attributi `nome_completo` e `data_nascita`. Un oggetto della classe `Impiegato` può essere istanziato con valori per tali attributi (poiché questi esistono, per ereditarietà, negli oggetti di `Impiegato`) oltre che con valori per gli attributi specifici `data_assunzione`, `stipendio`, `congedi`, `congedi_usati`.

2.1.5.1 Polimorfismo

Un metodo ereditato da una sottoclasse è spesso usato così come è stato definito nella superclasse. L'operazione `età()` funziona allo stesso modo per gli oggetti di classe `Persona` e per quelli di classe `Impiegato`, ma, a volte, un'operazione deve essere modificata (*overridden*) o completamente riscritta in una sottoclasse al fine di adeguarsi alle variazioni semantiche della sottoclasse stessa.

Ad esempio, `Impiegato.congediRimanenti()` è valutata sottraendo `congedi_usati` da `congedi` (Figura 2.17). Tuttavia, un impiegato che sia un manager ottiene annualmente dei `congedi_extra`. Se la classe `Manager` è aggiunta alla gerarchia di generalizzazione (come mostrato in Figura 2.18), l'operazione `Manager.congediRimanenti()` deve modificare l'operazione `Impiegato.congediRimanenti()` (come indicato in Figura 2.18 attraverso la duplicazione del nome dell'operazione nella sottoclasse).

Poiché l'operazione `congediRimanenti()` è stata modificata, nella sottoclasse esisteranno due implementazioni (due metodi) per l'operazione, e inviando il messaggio `congediRimanenti()` a un oggetto di classe `Impiegato` o di classe `Manager`, il metodo appropriato verrà eseguito (un oggetto sa a quale classe appartiene). L'operazione `congediRimanenti()` è quindi *polimorfa*.

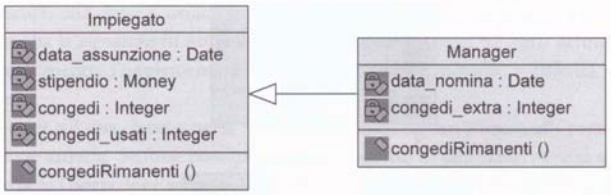


FIGURA 2.18 Polimorfismo.

2.1.5.2 Ereditarietà

Il *polimorfismo* serve a poco in assenza di ereditarietà. L'*ereditarietà* permette la definizione incrementale di una sottoclasse riusando ed estendendo le definizioni della superclasse. L'operazione `Manager.congediRimanenti()` è implementata probabilmente invocando le funzionalità di `Impiegato.congediRimanenti()` e quindi aggiungendo `congedi_extra` al valore da questa restituito.

Più classi in una gerarchia di ereditarietà possono dichiarare la stessa operazione. Tale operazione possiede diverse implementazioni (metodi) ma la medesima segnatura (il nome, il numero e il tipo dei suoi parametri). Il comportamento polimorfo è legato all'ereditarietà.

2.1.5.2.1 Ereditarietà multipla

Una sottoclasse può ereditare da più di una superclasse. L'*ereditarietà multipla* può portare a conflitti di ereditarietà che devono essere risolti esplicitamente dal programmatore.

Nella Figura 2.19, la classe `Tutore` eredita dalle classi `Insegnante` e `StudenteDottorato`. `Insegnante` a sua volta eredita da `Persona` e lo stesso accade per `StudenteDottorato` (attraverso `Studente`). Il risultato è che `Tutore` dovrebbe ereditare due volte gli attributi e le operazioni di `Persona` se il programmatore non impone all'ambiente di programmazione di ereditare una sola volta usando il cammino destro o quello sinistro (oppure se l'ambiente di programmazione stesso usa alcuni comportamenti di default, accettabili per il programmatore, che eliminano la duplicazione).

2.1.5.2.2 Classificazione multipla

Nella maggior parte degli ambienti di programmazione object-oriented, un oggetto può essere istanza di una sola classe. Questa è una restrizione fastidiosa poiché, in realtà, gli oggetti possono appartenere a più classi.

La *classificazione multipla* è diversa dall'ereditarietà multipla. Nella classificazione multipla un oggetto è simultaneamente l'istanza di due o più classi. Nell'ereditarietà multipla una classe può avere più superclassi ma solo una classe deve essere definita per ciascun oggetto.

Nell'esempio sull'ereditarietà multipla di Figura 2.19, ogni oggetto di classe `Persona` (come `Mary` o `Peter`) appartiene a una singola classe (la classe più *specifica* che si applica a esso). Se `Mary` è uno `StudenteDottorato` ma non un `Tutore` allora la classe di `Mary` è `StudenteDottorato`.

Il problema sorge se `Persona` è specializzata in alcune gerarchie ortogonali. Ad esempio, una `Persona` può essere un `Impiegato` o uno `Studente`, un `Maschio` o una `Femmina`, un `Bambino` o un `Adulto`. Senza la classificazione multipla, sarebbe necessario definire classi per ogni combinazione corretta tra le gerarchie ortogonali per ottenere, ad esempio, un oggetto di

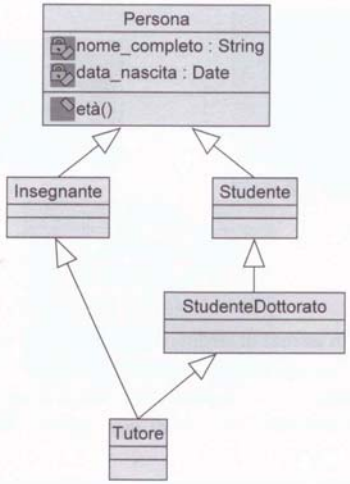


FIGURA 2.19 Ereditarietà multipla.

classe `Persona` che sia uno studente, bambino e femmina (la classe potrebbe essere chiamata `BambinoFemminaStudente`) (Fowler e Scott, 2000). Si osservi che in tal modo bisognerebbe introdurre otto sottoclassi!

2.1.5.2.3 Classificazione dinamica

Nella maggior parte degli ambienti di programmazione object-oriented, un oggetto non può cambiare la propria classe dopo che è stato istanziato (creato). Questa è un'altra fastidiosa restrizione poiché in realtà gli oggetti possono cambiare dinamicamente classe.

La *classificazione dinamica* è strettamente legata alla classificazione multipla. Un oggetto non solo appartiene a più classi ma può cambiare classe durante la sua esistenza.

Con la classificazione dinamica, un oggetto di classe `Persona` può essere prima un `Impiegato` e poi divenire un `Manager` (rimanendo comunque un impiegato). Senza la classificazione dinamica le modifiche a livello business, come la promozione degli impiegati, sono difficili (o anche impossibili) da rappresentare in modo dichiarativo nel diagramma delle classi. Dovrebbero essere rappresentate proceduralmente con diagrammi di stato o con tecniche simili di modellazione.

Purtroppo UML non fornisce alcun supporto per modellare la classificazione dinamica o la classificazione multipla; d'altra parte, lo stesso accade, in genere, per gli ambienti di programmazione.

2.1.5.3 Classe astratta

La *classe astratta* è un concetto importante di modellazione che segue dalla nozione di ereditarietà. Una classe astratta è una classe genitore che non ha direttamente oggetti istanza: solo le sue sottoclassi possono essere istanziate!

Generalmente, una classe è astratta perché almeno una delle sue operazioni è astratta. Un'*operazione astratta* ha la *segnatura* (il nome e la lista dei suoi argomenti formali) definita nella classe astratta genitore mentre l'implementazione dell'operazione (il metodo) è differita nelle classi figlie "concrete".

La ragione per cui una classe astratta non può istanziare oggetti è legata alla presenza, almeno, di un'operazione astratta. Se fosse possibile creare un oggetto di una classe astratta allora un messaggio che invochi l'operazione astratta dovrebbe causare un errore in esecuzione (poiché non esiste l'implementazione di quell'operazione nella classe di quell'oggetto).

Una classe può essere astratta solo se è una superclasse completamente partizionata in sottoclassi. La partizione è completa se le sottoclassi contengono tutti gli oggetti che possono essere istanziati nella gerarchia di ereditarietà. Non ci sono oggetti "smarriti" nella gerarchia (Page-Jones, 2000). La classe `Persona` di Figura 2.19 non può essere astratta ammesso che sia possibile creare oggetti di `Persona` che non siano né insegnanti né studenti. È anche possibile aggiungere in seguito altre sottoclassi a `Persona` (come `ImpiegatiAmministrativi`).

La Figura 2.20 mostra la classe astratta *`Video`* (in UML, il nome di una classe astratta è mostrato in corsivo). La classe contiene l'operazione astratta `spesaAffitto()`. Presumibilmente, le spese d'affitto sono calcolate diversamente per i videonastri e i DVD. Ci sono quindi due diverse implementazioni dell'operazione `spesaAffitto()` rispettivamente nelle classi `CassettaVideo` e `DiscoDVD`.

Le classi astratte non possono aver oggetti ma sono molto utili nella modellazione. In un certo senso, esse definiscono un "vocabolario" di alto livello che arricchisce il linguaggio della modellazione.

2.1.6 Oggetto classe

In precedenza si è distinto tra *oggetti istanza* e *oggetti classe*. Un *oggetto classe* è un oggetto con attributi e/o operazioni applicabili non agli oggetti della classe ma alla classe stessa nella sua globalità (per cui si parla di attributi e/o operazioni *globali*).

Poiché in un sistema object-oriented puro gli oggetti sono gli unici elementi di memorizzazione, è necessario avere esplicitamente un oggetto classe per contenere attributi e operazioni globali. I più comuni attributi globali sono attributi che contengono valori predefiniti (di default) o valori aggregati (come somme, contatori, medie). Le più comuni operazioni globali creano e distruggono oggetti istanza oppure calcolano valori aggregati.

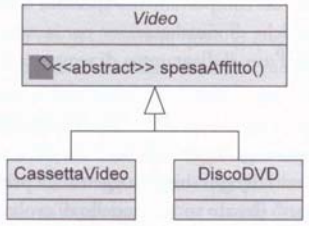


FIGURA 2.20 Classe astratta con operazione astratta.

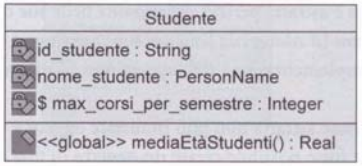


FIGURA 2.21 Classe con attributi e operazioni applicabili a livello classe.

La Figura 2.21 mostra la classe `Studente` con un attributo globale (`max_corsi_per_semestre`) e un'operazione globale (`mediaEtàStudenti()`). Ogni studente ha il medesimo numero massimo di corsi per semestre e di conseguenza tale numero dovrebbe essere memorizzato nell'oggetto classe. L'operazione per calcolare l'età media degli studenti ha un'applicabilità globale poiché necessita l'accesso individuale all'età degli studenti (oggetti istanza di `Studente`) al fine di determinare l'età media su tutti gli studenti.

UML impone che gli attributi e le operazioni globali siano evidenziati nel rettangolo di una classe. Nella Figura 2.21 un carattere "$" prefisso al nome di un attributo sta a significare che l'attributo è globale (o *statico*). La notazione UML di *stereotipo* con il nome `global` tra parentesi (<<>>) è stata usata per identificare un'operazione globale (o *statica*).

2.2 Analisi guidata di un caso di studio

Questa sezione presenta un'analisi guidata della modellazione visuale con UML usando un semplice esempio. Lo scopo è mostrare i vari diagrammi UML e come questi siano correlati. Ogni diagramma UML enfatizza una vista particolare del sistema. Per comprendere un sistema nella sua globalità, svariati diagrammi UML, rappresentanti diverse viste, devono essere sviluppati e integrati.

Genericamente, si possono distinguere tre tipologie di modelli UML, ognuno con il proprio insieme di diagrammi e i relativi costrutti:

1. Il *modello dello stato*, il quale rappresenta la *vista statica* del sistema e modella i *requisiti informativi*. Il modello dello stato (essendo lo stato definito attraverso il valore corrente degli attributi e delle associazioni di ciascun oggetto) specifica le strutture dei dati e le loro relazioni. La principale tecnica di visualizzazione è il diagramma delle classi.
2. Il *modello del comportamento*, il quale rappresenta la *vista operativa* del sistema e modella i *requisiti funzionali*. Il modello del comportamento specifica transazioni, operazioni e algoritmi sui dati. Ci sono molte tecniche di visualizzazione per la modellazione del comportamento, come il diagramma dei casi d'uso, il diagramma di sequenza, il diagramma di collaborazione e il diagramma di attività.
3. Il *modello del cambiamento di stato*, il quale rappresenta la *vista dinamica* del sistema e modella l'evoluzione nel tempo degli oggetti. Il modello del cambiamento di stato mostra le possibili transizioni fra gli stati di ogni singolo oggetto. La principale tecnica di visualizzazione è il diagramma "statechart" (basato su un modello di evoluzione degli stati di un oggetto).

2.2.1 Il caso di studio: Acquisti OnLine

Internet ha rivoluzionato i modi in cui fare business. Per restare competitive, le organizzazioni devono essere presenti su Internet e devono estendere le proprie applicazioni per considerare anche il commercio elettronico (*e-commerce*). Le modifiche sono ampie per quanto riguarda il "front-end" (lato client) delle applicazioni mentre nel database di "back-end" (lato server di sistema) permane il convenzionale trattamento delle transazioni.

L'analisi prende in esame un'applicazione per effettuare acquisti on-line. Si analizzerà principalmente il back-end dell'applicazione, come l'accettazione di un ordine on-line del cliente, il trattamento e l'evasione degli ordini, la fatturazione e la spedizione delle merci al cliente.

Scopo dell'analisi è mostrare come procede la modellazione visuale con UML. Anche la sequenza delle attività di modellazione è semplificata per mostrare le dipendenze tra i diversi diagrammi UML, per cui tale sequenza non deve essere interpretata come un *processo* raccomandato. In realtà, il processo è *iterativo* e *incrementale* (Sezione 1.1.3.1). Nello studio si è posto l'accento sulla modellazione dei casi d'uso: la modellazione delle classi potrebbe essere eseguita in modo parallelo alla modellazione dei casi d'uso.

Specifica del caso: Acquisti OnLine
(Trattamento degli ordini cliente)

Un produttore di computer offre la possibilità di acquistare computer via Internet. Il cliente può selezionare un computer sulla pagina web del produttore. I computer sono classificati in server, desktop e portatili. Il cliente può selezionare un configurazione standard o può costruire interattivamente la configurazione desiderata. I componenti configurabili (come le memorie) sono presentati in una lista di opzioni disponibili. Per ogni configurazione, il cliente può valutarne il costo.

Per ordinare, il cliente deve fornire le informazioni per la consegna e per il pagamento. Le modalità di pagamento accettate sono: carta di credito e assegno. Una volta che l'ordine è stato inviato, il sistema manda un e-mail di conferma al cliente con i dettagli dell'ordine. Il cliente, mentre attende l'arrivo del computer, può controllare on-line lo stato dell'ordine in ogni momento.

Il trattamento di back-end consiste nei passi di: verifica della solvibilità del cliente, richiesta al magazzino della configurazione ordinata, stampa di una fattura, e richiesta al magazzino di spedire il computer al cliente.

2.2.2 Modellazione dei casi d'uso

Il *comportamento del sistema* è ciò che il sistema fa quando reagisce a eventi esterni. In UML, il comportamento esteriormente visibile e controllabile del sistema è descritto dai casi d'uso. Un *caso d'uso* rappresenta una funzione di business che è *esternamente visibile* a un attore e che può essere *controllata* singolarmente nel processo di sviluppo.

Un *attore* rappresenta chiunque o qualunque cosa (persona, macchinario, etc.) interagisca (tramite un caso d'uso) con il sistema per ottenere un servizio o un qualche risultato utile dal sistema stesso.

Un *diagramma dei casi d'uso* è una rappresentazione visuale di attori e casi d'uso con specifiche e definizioni aggiuntive. Un diagramma dei casi d'uso non è solo un diagramma ma piuttosto un

modello documentato del comportamento atteso del sistema. La stessa considerazione vale anche per gli altri diagrammi UML. Se non diversamente precisato, *diagramma UML* sarà usato come sinonimo di *modello UML*.

2.2.2.1 Attori

Gli *attori* e i casi d'uso sono determinati dall'analisi dei requisiti funzionali. Le funzioni del sistema sono "materializzate" nei casi d'uso. I casi d'uso soddisfano i requisiti fornendo un risultato di valore per un attore. Non importa se l'analista sceglie di identificare prima gli attori e poi i casi d'uso o viceversa. In questo esempio, si è scelto di identificare prima gli attori.

Una tipica immagine grafica per un attore è un "omino stilizzato" (vedere di seguito). In generale, un attore può essere mostrato con un simbolo rettangolare di classe. Come discuteremo in seguito (si vedano Sezioni 2.2.5.3 e 3.5.2) potrebbe essere necessario rappresentare gli attori all'interno del sistema (ad esempio, per controllarne i permessi di accesso) per cui un attore sarà rappresentato come una *classe* e potrà avere attributi e operazioni (associate a eventi che invia o riceve).

Analisi: Passo 1 (Acquisti OnLine)

Con riferimento alla specifica del caso precedente (vedi sopra), si considerino i seguenti requisiti (opportunamente adattati) per trovare gli attori nell'applicazione di Acquisti OnLine:

1. Il cliente usa la pagina web del produttore per vedere la configurazione standard del computer (server, desktop o portatile) scelto. Il costo è esplicitamente mostrato.
2. Il cliente sceglie di vedere i dettagli della configurazione, per l'acquisto oppure per definire un'altra configurazione più adatta. Il costo di qualunque configurazione può essere calcolato su richiesta del cliente.
3. Il cliente può scegliere di ordinare il computer direttamente online oppure può richiedere un incontro con il venditore prima di confermare l'ordine, per ottenere maggiori spiegazioni sui dettagli dell'ordine, una negoziazione del costo, etc.
4. Per ordinare, il cliente deve riempire online un modulo con i dettagli della spedizione, l'indirizzo di fatturazione e i dettagli di pagamento (carta di credito o assegno).
5. Dopo che l'ordine del cliente è stato inviato e confermato, il venditore invia elettronicamente una richiesta al magazzino con i dettagli della configurazione ordinata.
6. I dettagli della transazione, compresi un numero d'ordine e un numero cliente, sono inviati via e-mail al cliente, permettendo a quest'ultimo di controllare lo stato dell'ordine.
7. Il magazzino riceve la fattura dal venditore e spedisce il computer al cliente.

La Figura 2.22 mostra tre attori che sono esplicitamente presenti nei requisiti. Gli attori sono `Cliente`, `Venditore`, `Magazzino`.

FIGURA 2.22 Attori (Acquisti online).

2.2.2.2 Casi d'uso

Un *caso d'uso* rappresenta un'unità funzionale che fornisce valore a un attore. Un attore che non comunica con almeno un caso d'uso è privo d'interesse mentre il contrario non è necessariamente vero (un caso d'uso che non comunica con alcun attore è permesso, e può avere senso). Possono esistere, infatti, casi d'uso che generalizzano o specializzano altri casi d'uso e che non interagiscono direttamente con attori; essi possono essere usati internamente da altri casi d'uso per fornire certi risultati agli attori.

I casi d'uso possono essere dedotti dall'identificazione dei compiti degli attori. La domanda da porsi è: "Quali sono le responsabilità e le aspettative dell'attore verso il sistema?". I casi d'uso possono anche essere determinati da un'analisi diretta dei requisiti funzionali. In molti casi, un *requisito funzionale* corrisponde direttamente a un *caso d'uso*.

Analisi: Passo 2 (Acquisti OnLine)

Con riferimento al Passo 1 di analisi si trovino i casi d'uso dell'applicazione Acquisti OnLine.

Per affrontare tale problema, si può costruire una matrice in cui i requisiti di funzionalità sono associati agli attori e ai casi d'uso. Si noti che alcune potenziali funzioni di business possono non essere incluse nel contesto dell'applicazione, per cui non devono essere trasformate in casi d'uso.

La Tabella 2.1 mostra l'assegnazione dei requisiti funzionali elencati al Passo 1 ad attori e casi d'uso. I compiti del magazzino riguardanti la configurazione del computer e la sua spedizione al cliente sono funzioni *esterne al contesto applicativo* e quindi non saranno prese in considerazione.

La Figura 2.23 mostra la notazione UML per i casi d'uso. Un caso d'uso è disegnato come un'ellisse con un nome all'interno o immediatamente sottostante. Poiché un caso d'uso rappresenta un'interazione fra un attore e il sistema, potrebbe essere necessario rappresentare l'interazione all'interno del sistema (ad esempio, a fini statistici), per cui un caso d'uso potrà essere anche rappresentato tramite una *classe* (e quindi potrà avere attributi e operazioni).

2.2.2.3 Diagramma dei casi d'uso

Il diagramma dei casi d'uso assegna i casi d'uso agli attori. Permette anche di stabilire delle relazioni tra casi d'uso, se necessario. Tali relazioni saranno analizzate nel Capitolo 4 (Sezione 4.3.1.2).

Il diagramma dei casi d'uso è la tecnica di visualizzazione principale per un modello comportamentale del sistema. Gli elementi del diagramma (casi d'uso e attori) devono essere ulteriormente descritti per ottenere un *modello dei casi d'uso* completo (si veda la Sezione 2.2.2.4).

Analisi: Passo 3 (Acquisti OnLine)

Con riferimento ai passi precedenti, disegnare un diagramma dei casi d'uso per l'applicazione Acquisti OnLine.

TABELLA 2.1 Assegnazione dei requisiti ad attori e casi d'uso (Acquisti OnLine).

Req.	Requisito	Attore	Caso d'uso
1	Il cliente usa la pagina web d'acquisti online del produttore per selezionare una configurazione standard del server, desktop o computer portatile che potrebbe interessargli. Il prezzo viene mostrato.	Cliente	Mostrare Configurazione Computer Standard
2	Il cliente esamina i dettagli della configurazione, per acquistarla direttamente o per costruire una configurazione più adatta. Il prezzo di ogni configurazione può essere calcolato su richiesta del cliente.	Cliente	Costruire Configurazione Computer
3	Il cliente potrebbe scegliere di ordinare il computer online oppure richiedere il contatto di un venditore per una spiegazione dei dettagli dell'ordine, negoziare il prezzo, etc., prima che l'ordine sia emesso.	Cliente, Venditore	Ordinare Computer Configurato, Richiedere Contatto Commerciale
4	Per emettere l'ordine, il cliente deve riempire un modulo online con gli indirizzi di spedizione e di fatturazione e con i dettagli del pagamento (carta di credito o assegno).	Cliente	Ordinare Computer Configurato, Verificare e Accettare Pagamento Cliente
5	Dopo che un ordine cliente è registrato nel sistema, il venditore invia al magazzino un e-mail con i dettagli della configurazione ordinata.	Venditore, Magazzino	Informare Magazzino su Ordine
6	I dettagli della transazione, compresi numero d'ordine e numero contabile cliente, sono inviati via e-mail al cliente, in modo che questi possa controllare online lo stato dell'ordine.	Cliente, Venditore	Ordinare Computer Configurato, Aggiornare Stato Ordine
7	Il magazzino ottiene la fattura dal venditore e spedisce il computer al cliente.	Venditore, Magazzino	Stampare Fattura

Una soluzione a questo passo può essere direttamente ottenuta dalle informazioni contenute nei passi precedenti. La sola considerazione aggiuntiva può riguardare le relazioni tra i casi d'uso. Il diagramma è presentato nella Figura 2.24. Il significato della relazione `<<extend>>` è che il caso d'uso `Ordinare Computer Configurato` può essere esteso dal cliente con il caso d'uso `Richiedere Contatto Commerciale`.

FIGURA 2.23 Casi d'uso (Acquisti OnLine).

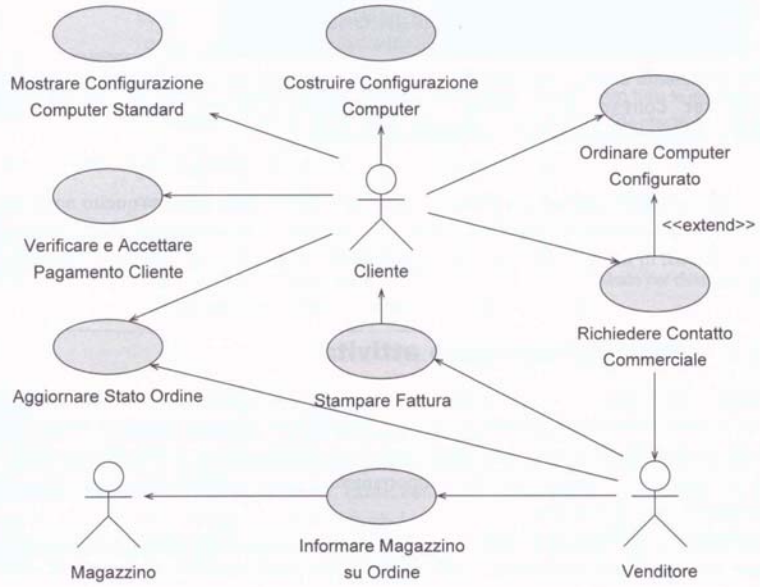


FIGURA 2.24 Diagramma dei casi d'uso (Acquisti OnLine).

2.2.2.4 Documentare i casi d'uso

Ciascun caso d'uso deve essere descritto in un documento testuale che definisce cosa il sistema deve fare quando l'attore attiva un caso d'uso. La struttura di un *documento di caso d'uso* può variare ma una tipica descrizione potrebbe contenere (vedi Quatrani, 2000):

- *Breve descrizione*
- *Attori* coinvolti
- *Precondizioni* necessarie per l'inizio del caso d'uso
- *Descrizione dettagliata* del flusso di eventi o passi che specificano il caso; essa comprende:
  - *Flusso principale* degli eventi, suddiviso a sua volta in:
    - *Sottoflussi* degli eventi (i sottoflussi possono essere ulteriormente suddivisi in sottoflussi più semplici per migliorare la leggibilità del documento)
  - *Flussi alternativi* per descrivere situazioni eccezionali
- *Postcondizioni* per definire lo stato del sistema dopo il termine del caso d'uso.

Il documento dei casi d'uso evolve con l'avanzamento del processo di sviluppo. Nella fase iniziale della determinazione dei requisiti, viene introdotta solo una breve descrizione dei casi, riservandosi di completare il documento in modo iterativo e inc rementale. Il documento completo emerge alla fine della fase di specifica dei requisiti. A quel punto, i modelli dell'interfaccia GUI possono essere aggiunti al documento. Più avanti, il documento dei casi d'uso verrà usato per produrre la documentazione utente del sistema implementato.

Analisi: Passo 4 (Acquisti OnLine)

Con riferimento ai passi precedenti, si scriva un documento di caso d'uso per `Ordinare Computer Configurato`. Usare la propria conoscenza di un tipico sistema di trattamento degli ordini per derivare dettagli non definiti nei requisiti.

La soluzione di questo passo è presentata in forma tabellare (Tabella 2.2), ma questo non è il modo tipico per documentare i casi d'uso. I documenti dei casi d'uso possono contenere molte pagine (una decina, in media) e di norma dovrebbero essere redatti come un normale documento con indice.

2.2.3 Modellazione delle attività

Il *modello delle attività* permette di rappresentare graficamente il flusso di eventi di un caso d'uso. Introdotti solo nelle ultime versioni di UML, i modelli delle attività colmano il vuoto tra una rappresentazione di alto livello del comportamento di sistema, con i *modelli dei casi d'uso*, e la rappresentazione a più basso livello del comportamento, con i *modelli di interazione* (diagrammi di sequenza e di collaborazione).

Il *diagramma delle attività* mostra i passi di una computazione. Ogni passo è uno *stato* in cui si esegue "qualcosa". Per tale motivo, i passi di esecuzione sono chiamati anche *stati di attività*.

TABELLA 2.2 Specifica testuale per il caso d'uso "Ordinare computer configurato" (Acquisti OnLine).

Caso d'uso	Ordinare computer configurato
Breve descrizione	Questo caso d'uso permette al `Cliente` di emettere un ordine d'acquisto. Ciò comprende la necessità di fornire gli indirizzi di consegna e fatturazione così come i dettagli del pagamento.
Attori	`Cliente`
Precondizioni	Il cliente usa un browser Internet per accedere alle pagine web del produttore. La pagina di inserimento ordine mostra i dettagli di un computer configurato con il suo prezzo.
Flusso principale	Il caso d'uso inizia quando il `Cliente` decide di ordinare il computer configurato scegliendo sullo schermo una funzione `Continua` (o simile) mostrata quando i dettagli dell'ordine sono visualizzati. Il sistema chiede che il `Cliente` inserisca i dettagli dell'acquisto, comprendendo: nome del venditore (se conosciuto), dettagli della spedizione (nome cliente e indirizzo), dettagli della fattura (se differenti dai dettagli della spedizione), metodo di pagamento (carta di credito o assegno), qualsiasi commento. Il `Cliente` sceglie una funzione `Acquisto` (o simile) per inviare l'ordine al produttore. Il sistema assegna un numero d'ordine univoco e un numero contabile cliente all'ordine d'acquisto e memorizza le informazioni sull'ordine nel database. Il sistema invia un e-mail al `Cliente` (con il numero d'ordine, il numero cliente e tutti i dettagli dell'ordine) come conferma d'accettazione dell'ordine.
Flussi alternativi	Il `Cliente` attiva la funzione di `Acquisto` prima di aver fornito tutte le informazioni necessarie. Il sistema visualizza un messaggio d'errore e richiede che le informazioni mancanti siano fornite. Il `Cliente` sceglie la funzione `Ricomincia` (o simile) per avere un modulo d'acquisto vuoto. Il sistema permette al `Cliente` di inserire le informazioni un'altra volta.
Postcondizioni	Se il caso d'uso ha avuto successo, l'ordine d'acquisto è registrato nel database del sistema. Altrimenti, lo stato del sistema non viene modificato.

Il diagramma fa vedere quali passi (o attività) sono eseguiti in sequenza e quali concorrentemente. Il flusso di controllo da un'attività alla seguente è chiamato *transizione*.

Se il documento dei casi d'uso è stato completato, allora le attività possono essere dedotte dalla descrizione dei *flussi principali* e *alternativi*. Esiste, comunque, un'importante distinzione tra la descrizione dei casi d'uso e il modello delle attività. Le descrizioni dei casi d'uso sono "scritte" dalla *prospettiva* di un *attore esterno*. I modelli di attività sono piuttosto il *punto di vista interno del sistema*.

I modelli delle attività, oltre alla descrizione dei casi d'uso, possono avere altri impieghi nello sviluppo del sistema (Fowler e Scott, 2000). Possono essere usati per comprendere i processi di business a un alto livello di astrazione prima ancora della definizione di un qualsiasi caso d'uso. In alternativa, possono essere usati a un più basso livello di astrazione per progettare complessi algoritmi sequenziali oppure nel progetto di applicazioni concorrenti.

2.2.3.1 Attività

Se il modello delle attività è usato per visualizzare la sequenza di attività in un caso d'uso, le attività possono essere definite a partire dal documento del caso d'uso. Come osservato in precedenza, le attività dovrebbero essere identificate partendo dal punto di vista del sistema, non degli attori.

Un'attività è rappresentata in UML da un rettangolo con angoli arrotondati. Per essere precisi, lo stesso simbolo grafico è usato per visualizzare uno *stato di attività* e uno *stato di azione*. La distinzione tra l'attività e l'azione è la scala temporale. L'attività necessita di tempo per essere completata; l'azione è così veloce che può essere considerata "istantanea" in rapporto a una scala temporale "normale". Di conseguenza, nei modelli statechart (Sezione 2.2.6) le attività possono essere definite solo all'interno di *stati di oggetto* mentre le azioni possono apparire sulle transizioni tra stati di oggetto.

Analisi: Passo 5 (Acquisti OnLine)

Con riferimento al passo 4 di analisi, si considerino nell'applicazione Acquisti OnLine i flussi principale e alternativi del documento del caso d'uso `Ordinare Computer Configurato` e si trovino le attività.

La Tabella 2.3 elenca le specifiche dei flussi principale e alternativi del documento di caso d'uso e identifica gli stati di attività. Si notino i nomi delle attività, espressi dal punto di vista del sistema (non degli attori). Le attività identificate nella Tabella 2.3 sono rappresentate nella Figura 2.25.

2.2.3.2 Diagramma di attività

Il *diagramma di attività* mostra le transizioni tra le attività. A meno che esso non rappresenti un ciclo, il diagramma avrà uno stato iniziale e uno o più stati finali. Un cerchio pieno rappresenta lo stato iniziale, uno *stato finale* è mostrato usando un cerchio con all'interno un cerchio pieno più piccolo.

Le transizioni si possono *separare* (*branch*) e *fondere* (*merge*). Ciò crea *cammini (threads)* di computazione *alternativi*. Un rombo mostra la condizione di separazione.

FIGURA 2.25 Attività per il caso d'uso "Ordinare computer configurato" (Acquisti OnLine).

Le transizioni possono anche *biforcarsi* (*fork*) e *riunirsi* (*join*). Questo crea *cammini* di computazione *concorrenti* (paralleli). Le biforcazioni/riunioni delle transizioni sono rappresentate da *barre* (segmenti spessi). Si noti che un diagramma di attività senza processi concorrenti assomiglia a un *diagramma di flusso* convenzionale (nell'esempio non si presentano comportamenti concorrenti; nella Sezione 4.3.2 è discusso un esempio in cui è evidenziato un comportamento concorrente).

Analisi: Passo 6 (Acquisti OnLine)

Con riferimento ai Passi 4 e 5 di analisi dell'applicazione Acquisti OnLine, si rappresenti un diagramma d'attività per il caso d'uso `Ordinare Computer Configurato`.

TABELLA 2.3 Ricerca delle attività nei flussi principale ed alternativi del caso d'uso "Ordinare computer configurato" (Acquisti OnLine).

N.	Asserzioni contenute nel caso d'uso	Attività
1	Il caso d'uso inizia quando il `Cliente` decide di ordinare il computer configurato scegliendo sullo schermo una funzione Continua (o simile) mostrata quando i dettagli dell'ordine sono visualizzati.	Mostrare Attuale Configurazione; Ottenere Richiesta Ordine
2	Il sistema chiede che il `Cliente` inserisca i dettagli dell'acquisto, comprendendo: nome del venditore (se conosciuto), dettagli della spedizione (nome cliente e indirizzo), dettagli della fattura (se differenti dai dettagli della spedizione), metodo di pagamento (carta di credito o assegno), qualsiasi commento.	Ottenere Dettagli Acquisto; Mostrare Modulo Acquisto
3	Il `Cliente` sceglie una funzione `Acquisto` (o simile) per inviare l'ordine al produttore.	Mostrare Modulo Acquisto
4	Il sistema assegna un numero d'ordine univoco e un numero contabile cliente all'ordine d'acquisto e memorizza le informazioni sull'ordine nel database.	Memorizzare Ordine
5	Il sistema invia un e-mail al `Cliente` (con il numero d'ordine, il numero cliente e tutti i dettagli dell'ordine) come conferma d'accettazione dell'ordine.	Email Dettagli Ordine
6	Il `Cliente` attiva la funzione di `Acquisto` prima di aver fornito tutte le informazioni necessarie. Il sistema visualizza un messaggio d'errore e richiede che le informazioni mancanti siano fornite.	Ottenere Dettagli Acquisto; Mostrare Modulo Acquisto
7	Il `Cliente` sceglie la funzione `Ricomincia` (o simile) per avere un modulo d'acquisto vuoto. Il sistema permette al `Cliente` di inserire le informazioni un'altra volta.	Ottenere Richiesta Ordine

Per disegnare il diagramma, le attività identificate al Passo 5 devono essere connesse con transizioni, come mostrato in Figura 2.26. `Mostrare Attuale Configurazione` è l'attività iniziale; la *transizione ricorsiva* su di essa rappresenta il fatto che la configurazione è continuamente aggiornata fino a quando una successiva transizione provoca l'esecuzione di un'altra attività (`Ottenere Richiesta Ordine`).

Nello stato `Mostrare Modulo Acquisto`, la condizione `[tempolimite]` provoca la fine esecuzione del modello di attività (si è superato il tempo limite). In alternativa, lo stato `Ottenere Dettagli Acquisto` è attivato (l'attività viene eseguita). Se i dettagli d'acquisto sono incompleti, il sistema entra ancora nello stato `Mostrare Modulo Acquisto` altrimenti si porta nello stato `Memorizzare Ordine` e in seguito nello stato `Email Dettagli Ordine` (quindi va allo stato finale).

Si noti che sono mostrate solo le condizioni di separazione verificate all'uscita da un'attività. Le condizioni di separazione interne all'attività non sono esplicitate e possono essere dedotte dalla presenza di più transizioni di uscita, sulle quali è possibile indicare delle *condizioni di guardia* tra parentesi quadre (come `[tempolimite]` all'uscita di `Mostrare Modulo Acquisto`).

2.2.4 Modellazione delle classi

Lo *stato del sistema* rappresenta il contenuto informativo del sistema a un dato istante, ed è costituito dall'insieme corrente di oggetti istanza (si veda la Sezione 2.1.1).

La definizione dello stato del sistema è data per mezzo di *modelli delle classi*. Gli elementi di modellazione comprendono le classi stesse, attributi e operazioni delle classi, associazioni, aggregazioni e composizioni, e generalizzazioni (si vedano le Sezioni 2.1.2, 2.1.3, 2.1.4, 2.1.5). Un *diagramma delle classi* offre una rappresentazione grafica integrata di tali elementi di modellazione.

Si ricordi che sebbene in questo esempio si discuta la modellazione delle classi dopo quella dei casi d'uso, in pratica tali attività di modellazione sono condotte in parallelo (si veda la Sezione 3.2). I due modelli si completano mutuamente: i casi d'uso facilitano la scoperta delle classi e, viceversa, i modelli delle classi possono portare alla scoperta di casi d'uso sfuggiti in un primo tempo.

2.2.4.1 Classi

Finora le classi sono state usate per definire gli *oggetti di business*. Gli esempi forniti hanno fatto riferimento a entità di business persistenti, come `Ordine`, `Spedizione`, `Cliente`, `Studente` etc. Queste classi definiscono anche il *modello del database* che costituirà il supporto al dominio applicativo e, per tale ragione, sono spesso indicate come *classi entità* (rappresentano gli oggetti persistenti nel database; nel seguito del libro saranno chiamate anche *classi entity*).

Le *classi entity* definiscono il nucleo del sistema informativo. L'analisi dei requisiti si interessa principalmente alle classi entità. Tuttavia, affinché il sistema funzioni, vi è bisogno di altre classi come, ad esempio, quelle necessarie per definire gli oggetti dell'interfaccia GUI fra gli utenti e il sistema, chiamate anche *classi di confine* (*classi boundary*) o di presentazione. Il sistema necessita inoltre di classi da usare nel controllo della logica di programma, le *classi di controllo* (*control classes*) (si veda la Sezione 5.2.3).

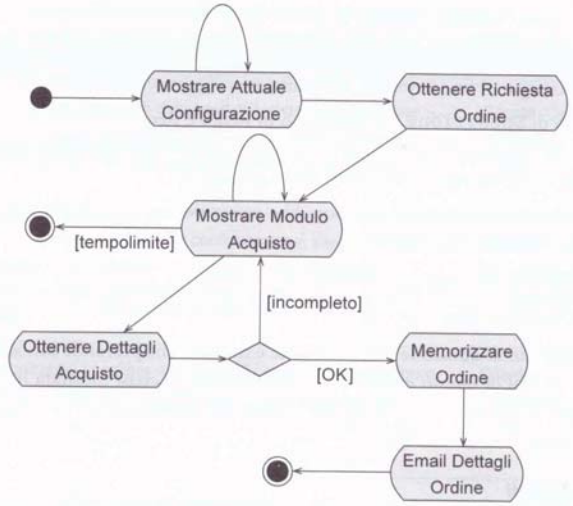


FIGURA 2.26 Diagramma di attività per il caso d'uso "Ordinare computer configurato" (Acquisti OnLine).

A seconda dell'approccio di modellazione prescelto, le *classi boundary* e le *classi control* possono, o non, essere trattate nell'analisi dei requisiti. La modellazione di tali classi può, in genere, essere ritardata fino alla fase di progetto del sistema.

Seguendo l'approccio usato per scoprire attori e casi d'uso (si veda la Tabella 2.1), è possibile costruire una tabella di ausilio per la ricerca delle classi nell'analisi dei requisiti funzionali. La Tabella 2.4 associa i requisiti funzionali elencati nel Passo 1 alle classi entity.

Analisi: Passo 7 (Acquisti OnLine)

Con riferimento ai requisiti definiti nella specifica del caso (Sezione 2.2.1) e al Passo 1 (Sezione 2.2.2.1), si trovino le classi entity per l'applicazione Acquisti OnLine.

La ricerca delle classi è un compito iterativo e la lista iniziale delle classi candidate può modificarsi. Per determinare se un concetto contenuto nei requisiti è una classe candidata, si può far riferimento ad alcune semplici domande:

1. Il concetto è un "contenitore" di dati?
2. Possiede degli attributi che possono assumere valori distinti?
3. Potrebbe comprendere molti oggetti istanza?
4. Appartiene al contesto applicativo in esame?

La lista di classi della Tabella 2.4 suggerisce ancora molte domande. Ad esempio:

1. Qual è la differenza tra `ComputerConfigurato` e `Ordine`? Dopo tutto, non ci saranno computer configurati fino a quando un ordine non verrà inviato, o no?
2. Il significato di `Spedizione` è il medesimo in Req.4 e Req.7? Probabilmente no. La classe `Spedizione` è necessaria sapendo che la spedizione è responsabilità del magazzino e di conseguenza fuori dal contesto applicativo? (Sezione 2.2.2.2)
3. `ElementoConfigurazione` non potrebbe essere ridotto a un insieme di attributi della classe `ComputerConfigurato`?
4. `StatoOrdine` è una classe o un attributo di `Ordine`?
5. `Venditore` è una classe o un attributo di `Ordine` e `Fattura`?

Rispondere a tali o simili domande non è semplice e richiede una profonda conoscenza dei requisiti applicativi. Ai fini di questo esempio, si è scelta la lista di classi mostrata in Figura 2.27. Si noti che la classe `Cliente` è già apparsa come *attore* nel diagramma dei casi d'uso, il che spiega l'annotazione "dalla vista dei casi d'uso".

2.2.4.2 *Attributi*

La struttura di una classe è definita dai suoi *attributi* (Sezione 2.1.2.1). L'analista deve avere una qualche conoscenza sugli attributi quando, inizialmente, introduce una classe. In pratica, gli attributi principali sono di norma definiti contestualmente all'introduzione di una classe nel modello.

Analisi: Passo 8 (Acquisti OnLine)

Con riferimento ai Passi 5,6,7, si introducano gli attributi per le classi di Figura 2.27, considerando solo attributi con tipo primitivo (Sezione 2.1.2.1).

FIGURA 2.27 Classi (Acquisti OnLine).

TABELLA 2.4 Assegnazione dei requisiti a classi entity (Acquisti online)

Req.	Requisito	Classi entity
1	Il cliente usa la pagina web d'acquisti online del produttore per selezionare una configurazione standard del server, desktop o computer portatile che potrebbe interessargli. Il prezzo viene mostrato.	Cliente, Computer (Prodotto, ConfigurazioneStandard)
2	Il cliente esamina i dettagli della configurazione, per acquistarla direttamente o per costruire una configurazione più adatta. Il prezzo di ogni configurazione può essere calcolato su richiesta del cliente.	Cliente, ComputerConfigurato, ElementoConfigurazione
3	Il cliente potrebbe scegliere di ordinare il computer online oppure richiedere il contatto di un venditore per una spiegazione dei dettagli dell'ordine, negoziare il prezzo, etc., prima che l'ordine sia emesso.	Cliente, ComputerConfigurato, Ordine, Venditore
4	Per emettere l'ordine, il cliente deve riempire un modulo online con gli indirizzi di spedizione e di fatturazione e con i dettagli del pagamento (carta di credito o assegno).	Cliente, Ordine, Spedizione, Fattura, Pagamento
5	Dopo che un ordine cliente è registrato nel sistema, il venditore invia al magazzino un e-mail con i dettagli della configurazione ordinata.	Cliente, ComputerConfigurato, Ordine, Venditore, ElementoConfigurazione
6	I dettagli della transazione, compresi numero d'ordine e numero contabile cliente, sono inviati via e-mail al cliente, in modo che questi possa controllare online lo stato dell'ordine.	Cliente, Ordine, StatoOrdine
7	Il magazzino ottiene la fattura dal venditore e spedisce il computer al cliente.	Spedizione, Fattura

La Figura 2.28 mostra le classi con attributi primitivi. Solo gli attributi più interessanti sono mostrati. Gli attributi di `ElementoConfigurazione` necessitano di una breve spiegazione. L'attributo `tipo_elemento` avrà valori come: `processore`, `memoria`, `schermo`, `disco_rigido` etc. L'attributo `descr_elemento` descrive con più dettagli il tipo dell'elemento. Ad esempio, il processore nella configurazione può essere: Intel 600Mhz con 256k di cache.

Comprensibilmente, c'è un significativo ammontare di decisioni da prendere per la definizione degli attributi di Figura 2.28. Diverse interpretazioni sono possibili e possono essere evidenziate nei tentativi di soluzione che il lettore può condurre per proprio conto.

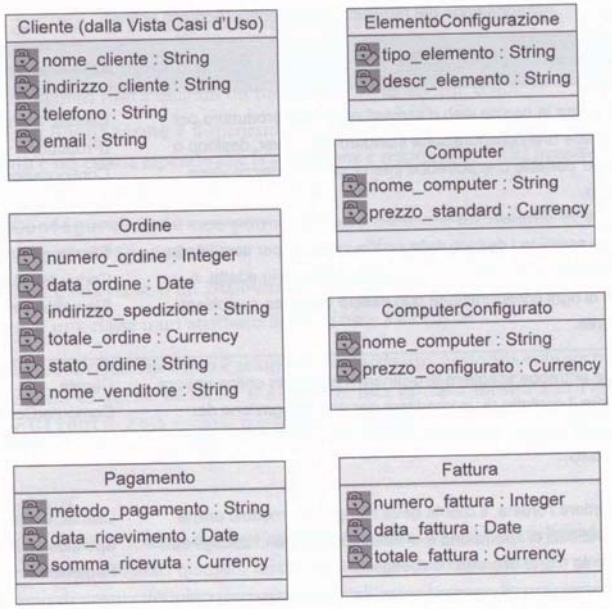


FIGURA 2.28 Attributi primitivi (Acquisti OnLine).

2.2.4.3 Associazioni

Le *associazioni* tra classi stabiliscono legami per le collaborazioni tra oggetti (Sezione 2.1.3). Nell'implementazione del sistema, le associazioni saranno rappresentate con attributi i cui tipi sono le classi partecipanti alla singola associazione (Sezione 2.1.2.1.1).

Analisi: Passo 9 (Acquisti OnLine)

Con riferimento ai passi precedenti, si considerino le classi di Figura 2.28. Si analizzino i cammini di accesso tra dette classi necessari per i casi d'uso. Si aggiungano le associazioni al modello delle classi.

La Figura 2.29 mostra le associazioni più evidenti tra le classi. Nella Sezione 2.1.3.2 erano state introdotte alcune ipotesi per la determinazione della *molteplicità* delle associazioni. L'ordine viene emesso da un singolo `Cliente` ma un `Cliente` può fare più `Ordini`. Un `Ordine` non viene accettato fino a quando il `Pagamento` non sia stato configurato (quindi, un'associazione 1 a 1). Un `Ordine` può non avere una `Fattura` associata ma la `Fattura` è sempre relativa a un solo `Ordine`. Un `Ordine` si riferisce a uno o più `ComputerConfigurato`. Un `ComputerConfigurato` (cioè una certa configurazione di elementi) può essere ordinato più volte (o mai).

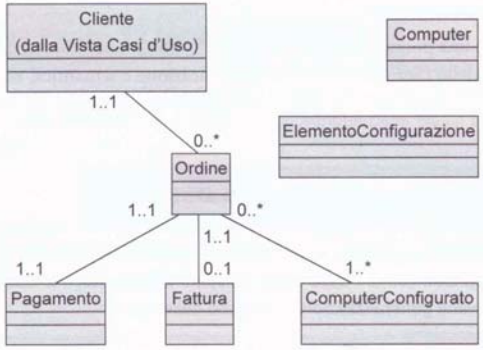


FIGURA 2.29 Associazioni (Acquisti OnLine).

2.2.4.4 Aggregazioni

Aggregazione e composizione sono forme forti di associazione, con una semantica propria (Sezione 2.1.4). Nei comuni ambienti commerciali di programmazione, aggregazioni e composizioni sono rappresentate come associazioni, con attributi aventi per tipo le classi partecipanti alle singole associazioni (Sezione 2.1.2.1.1).

Analisi: Passo 10 (Acquisti OnLine)

Con riferimento ai passi precedenti, si considerino i modelli di Figura 2.28 e 2.29. Si aggiungano le aggregazioni al modello delle classi.

La Figura 2.30 contiene due relazioni di aggregazione. Un `Computer` possiede uno o più `ElementoConfigurazione` e un `ComputerConfigurato` consiste di uno o più `ElementoConfigurazione`.

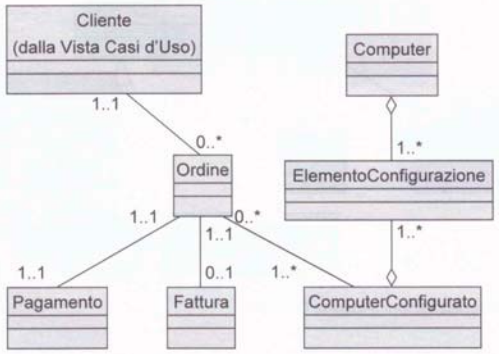


FIGURA 2.30 Aggregazioni (Acquisti OnLine).

2.2.4.5 Generalizzazioni

La *generalizzazione* è una potente tecnica di riuso del software ma può anche semplificare e rendere più chiaro un modello (Sezione 2.1.5). La semplificazione è semantica, non grafica. La generalizzazione crea di solito nuove classi generiche (eventualmente *astratte*) nella gerarchia di generalizzazione.

Analisi: Passo 11 (Acquisti OnLine)

Con riferimento ai passi precedenti, si considerino i modelli delle Figure 2.28 e 2.30. Si analizzi come estrarre attributi comuni dalle classi esistenti assegnandoli a classi a livello più alto. Aggiungere le derivanti generalizzazioni al modello delle classi.

La Figura 2.31 mostra il modello modificato con una generica classe `Computer` astratta e con due sottoclassi concrete `ComputerStandard` e `ComputerConfigurato`. `Ordine` e `ElementoConfigurazione` sono ora legate a `Computer` e `Computer` può essere sia `ComputerStandard` sia `ComputerConfigurato`.

2.2.4.6 Diagramma delle classi

Il diagramma delle classi è "il cuore e l'anima" di un sistema object-oriented. In questo esempio viene considerato solo l'aspetto di *modellazione statica* ottenibile con il modello delle classi, per cui non sono state definite operazioni per le classi stesse. Le operazioni appartengono più al campo del progetto che a quello dell'analisi. Quando le operazioni sono incluse nelle classi, il modello definisce implicitamente il *comportamento* del sistema.

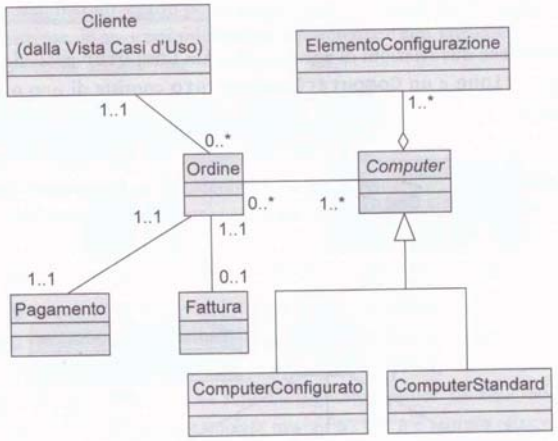


FIGURA 2.31 Generalizzazioni (Acquisti OnLine).

Analisi: Passo 12 (Acquisti OnLine)

Con riferimento ai passi precedenti, si integrino i modelli di Figura 2.28 e 2.31 mostrando un diagramma delle classi completo. Si modifichino gli attributi delle classi come conseguenza dell'introduzione della gerarchia di generalizzazione.

La Figura 2.32 è il diagramma delle classi per l'applicazione Acquisti OnLine. Non può comunque considerarsi una soluzione completa: per esempio, ulteriori attributi potrebbero essere ancora necessari in un'implementazione pratica.

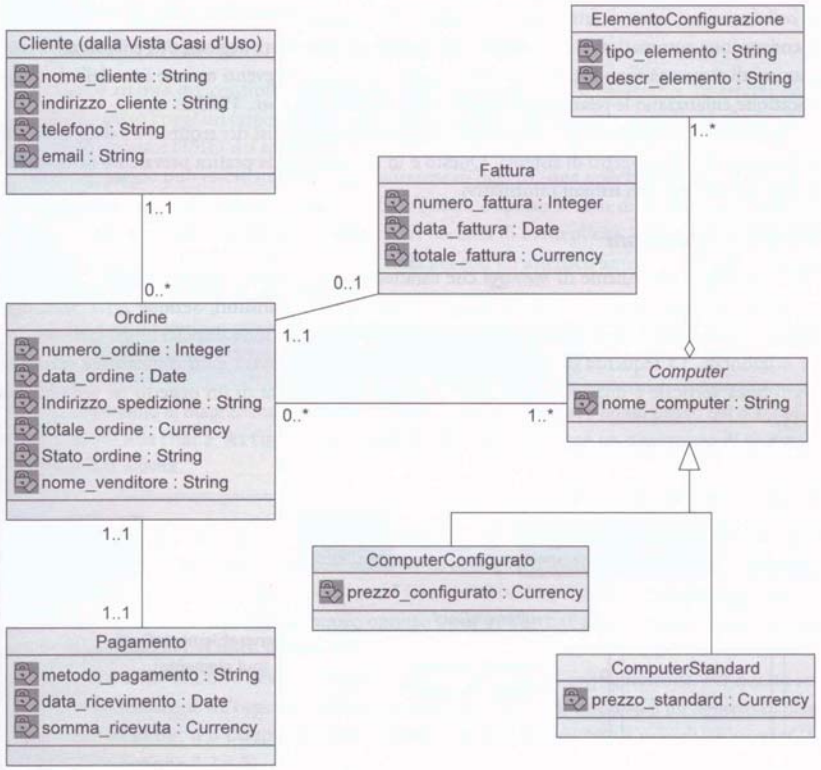


FIGURA 2.32 Diagramma delle classi (Acquisti OnLine).

2.2.5 Modellazione dell'interazione

La *modellazione dell'interazione* considera le interazioni tra oggetti come la tecnica per eseguire i casi d'uso. I modelli di interazione sono usati negli stadi più avanzati dell'analisi dei requisiti, quando un modello delle classi di base è noto, in modo che i riferimenti agli oggetti siano fondati su tale modello.

L'osservazione precedente sottolinea la principale distinzione tra modellazione delle attività (si veda la Sezione 2.2.3) e modellazione dell'interazione. Entrambe catturano il comportamento di un singolo caso d'uso, ma la *modellazione delle attività* è fatta a un alto livello di astrazione, mostrando la sequenza di eventi senza che tali eventi siano assegnati a oggetti, mentre la *modellazione dell'interazione* mostra la sequenza di eventi (messaggi) tra gli oggetti durante la collaborazione.

Esistono due tipologie di diagramma d'interazione, il *diagramma di sequenza* e *il diagramma di collaborazione*. Sono di fatto intercambiabili e, di conseguenza, molti strumenti CASE offrono la conversione automatica da un modello all'altro. La differenza sta negli aspetti enfatizzati. I diagrammi di sequenza sono focalizzati sulle sequenze temporali di eventi mentre i modelli di collaborazione enfatizzano le relazioni tra oggetti (vedi Rumbaugh *et al.*, 1999).

In questo libro si useranno i diagrammi di sequenza nell'analisi dei requisiti e i diagrammi di collaborazione nel progetto di sistema. Questo è in accordo con la pratica prevalente nelle metodologie di sviluppo dei sistemi informativi.

2.2.5.1 Interazioni

Un'*interazione* è un insieme di *messaggi* che caratterizza un qualche comportamento; i messaggi vengono scambiati tra *oggetti* utilizzando *legami* (persistenti o transitori, Sezione 2.1.1.3). Il diagramma di sequenza è un diagramma bidimensionale. Gli oggetti sono disposti lungo la dimensione orizzontale. La sequenza di messaggi è mostrata dall'alto al basso nella dimensione verticale. Ogni linea verticale è una linea temporale chiamata *linea di vita* di un oggetto (si veda Figura 2.33).

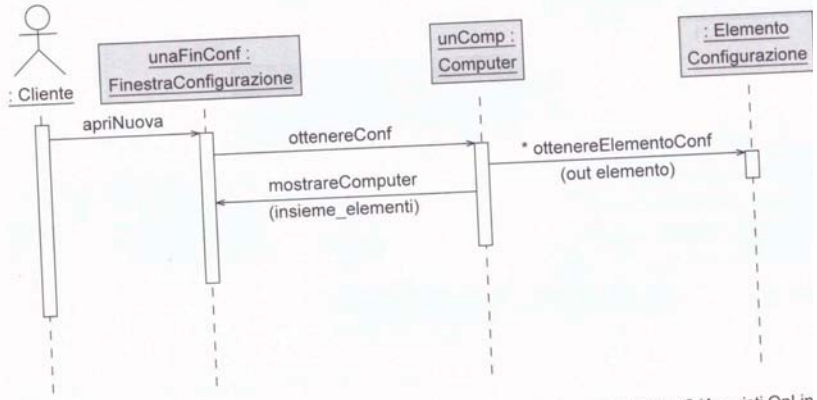


FIGURA 2.33 Diagramma di sequenza per l'attività "Mostrare attuale configurazione" (Acquisti OnLin

Una freccia rappresenta un messaggio da un oggetto chiamante (*sender*) a un'operazione (metodo) dell'oggetto chiamato (*target*). Il messaggio ha un nome e può avere una lista di *argomenti attuali* attraverso cui informazioni (anche di controllo) possono essere scambiate fra i due oggetti. Gli argomenti attuali corrispondono a quelli *formali* del metodo definiti nella classe target.

Un argomento attuale può essere un *argomento di input* (dal sender al target) o un *argomento di output* (ritornato dal target al sender). L'argomento di input può essere identificato con la parola chiave `in` (se non è presente alcuna parola chiave, si assume l'argomento come in input). L'argomento di output è identificato con la parola chiave `out`. Sono anche possibili argomenti allo stesso tempo di input e di output (identificati con la parola chiave `inout`) ma sono rari nelle soluzioni object-oriented. Ad esempio, il messaggio `ottieniNomeCorso` (si veda la Sezione 2.1.1.3.1), inviato a un oggetto identificato dalla variabile `rif_crs`, ha un argomento di output e nessun argomento di input:

```
rif_crs.ottieniNomeCorso(out nome_crs)
```

Mostrare il *ritorno* del controllo dall'oggetto target al sender non è necessario. La freccia con il messaggio verso l'oggetto target implica automaticamente il ritorno del controllo al sender: il target infatti conosce l'OID del sender.

Un messaggio può essere inviato a una *collezione* di oggetti (una collezione potrebbe essere un insieme, una lista, un vettore di oggetti etc.). Questo è spesso il caso di un oggetto chiamante connesso con un legame a più oggetti destinatari (a causa di molteplicità 1 a molti o molti a molti dell'associazione). Una *marca d'iterazione*, indicata con un asterisco anteposto al nome del messaggio, indica l'iterazione su una collezione.

Analisi: Passo 13 (Acquisti OnLine)

Con riferimento al diagramma di attività di Figura 2.26, si consideri il primo passo del diagramma ovvero `Mostrare Attuale Configurazione`. Si costruisca un diagramma di sequenza per tale attività.

Un diagramma di sequenza per "mostrare attuale configurazione" è presentato in Figura 2.33. L'attore esterno (`Cliente`) richiede di mostrare la configurazione di un computer. Il messaggio `apriNuova` è inviato a un oggetto di classe `FinestraConfigurazione`. Il messaggio provoca la *creazione* (*istanziazione*) di un nuovo oggetto `unaFinConf` (`FinestraConfigurazione` è una *classe boundary*, si veda Sezione 2.2.4.1).

L'oggetto `unaFinConf` deve "mostrare se stesso" con i dati di configurazione. Per questo scopo, invia un messaggio all'oggetto `unComp:Computer`. In realtà, `unComp` è un oggetto di classe `ComputerStandard` o `ComputerConfigurato` poiché *`Computer`* è una classe astratta (Figura 2.31 in Sezione 2.2.4.5).

L'oggetto `unComp` usa l'argomento di output `elemento` per "comporre se stesso" a partire da oggetti di `ElementoConfigurazione`. Manda quindi gli elementi della configurazione a `unaFinConf` attraverso l'argomento `insieme_elementi` del messaggio `mostrareComputer`. L'oggetto `unaFinConf` può ora "mostrare se stesso". La visualizzazione sullo schermo sarà simile a quella in Figura 2.34.

Si noti che i campi di scelta nella Figura 2.34 (come il campo `Memory`) mostrano all'inizio gli elementi standard di configurazione. Le liste di scelta contengono altri elementi non standard che il cliente può selezionare al fine di creare una nuova configurazione. Il diagramma di sequenza di Figura 2.33 non rappresenta come e quando i dati sono inseriti nei campi di scelta.

2.2.5.2 Operazioni

Sebbene l'introduzione di *operazioni* nelle classi sia spesso ritardata alla fase di progetto, si illustrerà in questo esempio come l'analisi delle interazioni possa condurre alla scoperta di operazioni. La dipendenza tra le interazioni e le operazioni è molto semplice: ogni *messaggio* invoca l'*operazione* dell'oggetto target che ha lo stesso nome del messaggio.

Ovviamente, questa corrispondenza biunivoca tra *messaggi* nei modelli d'interazione e *operazioni* implementate nella classe può essere ragionevolmente sostenuta se il modello d'interazione

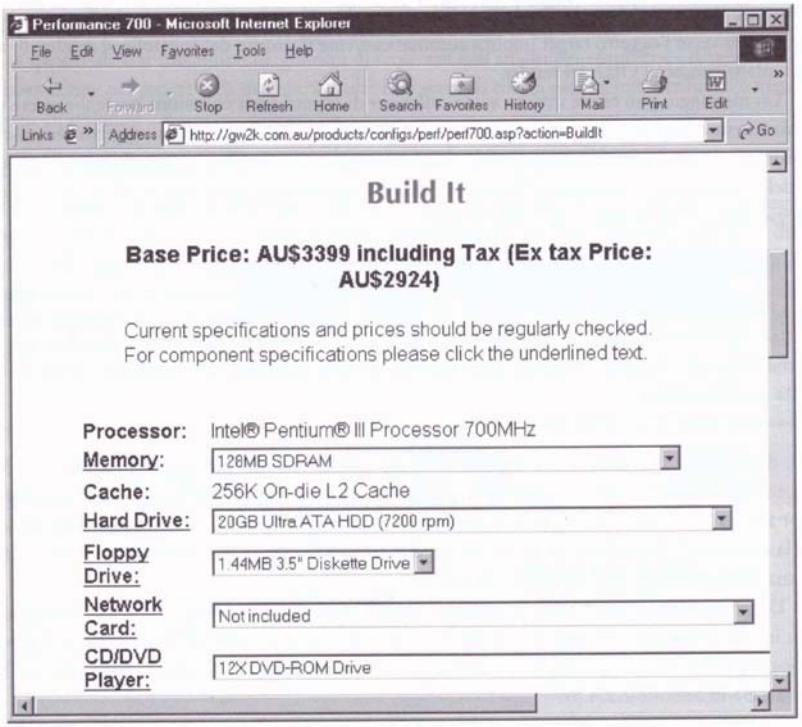


FIGURA 2.34 Esempio di finestra di configurazione (Acquisti OnLine) (Per gentile concessione di Gateway, Sydney, Australia).

costituisce un dettagliato progetto tecnico: questo, in genere, non è né possibile né desiderabile nella fase di analisi.

A lato, si osservi che una relazione simile esiste tra *messaggi* e *associazioni* ogni qualvolta un messaggio è scambiato tra oggetti *persistenti* utilizzando un legame per individuare l'oggetto target (Sezione 2.1.1.3.1). Di conseguenza, la presenza di questo tipo di messaggi in un diagramma di sequenza implica la presenza di un'associazione nel diagramma delle classi.

Analisi: Passo 14 (Acquisti OnLine)

Con riferimento al diagramma delle classi in Figura 2.32 e al diagramma di sequenza in Figura 2.33, si aggiunga, per ogni messaggio d'interazione nel diagramma di sequenza, un'operazione nella classe corrispondente specificata nel diagramma delle classi (non si ridisegni l'intero diagramma di classe, si mostrino solo le classi estese dalle operazioni).

Il risultato di questo semplice passo è mostrato in Figura 2.35. `FinestraConfigurazione` è una classe boundary. Le altre due classi sono classi entity rappresentanti oggetti persistenti del database. L'operazione `apriNuova` è stereotipata come *operazione di creazione* `<<constructor>>` e verrà implementata come un metodo di creazione che istanzia nuovi oggetti della classe. La classe `Computer` è *astratta* e `ottenereConf` è un'operazione astratta specificata nelle sottoclassi (`ComputerConfigurato` e `ComputerStandard`). Tali sottoclassi contengono la propria implementazione di `ottenereConf`.

2.2.5.3 Diagramma di sequenza

Per ogni caso d'uso può essere costruito, in genere, uno specifico diagramma di sequenza. Poiché ogni caso d'uso può anche essere espresso con un diagramma di attività, un diagramma di sequenza può essere definito per ogni diagramma di attività. Fornire svariati punti di vista correlati tra loro per descrivere un sistema è una buona pratica di modellazione.

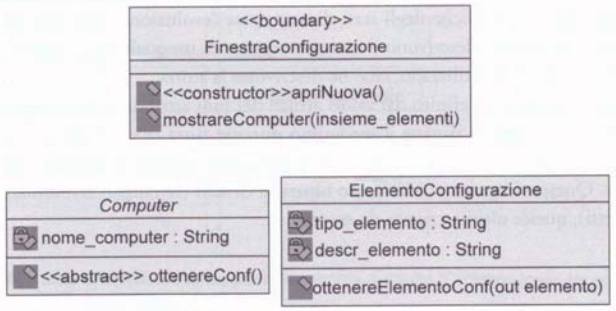


FIGURA 2.35 Uso delle interazioni per aggiungere operazioni alle classi (Acquisti OnLine).

Analisi: Passo 15 (Acquisti OnLine)

Con riferimento al diagramma di attività in Figura 2.26, si costruisca il diagramma di sequenza corrispondente. Per semplicità, non si mostri lo scambio di messaggi tra gli oggetti di `Computer` e di `ElementoConfigurazione` con gli oggetti delle sottoclassi (si assuma che un oggetto di `Computer` sia di `ComputerStandard` oppure di `ComputerConfigurato`).

Si mostrino esclusivamente i messaggi di attivazione. I ritorni sono impliciti. Non è necessario specificare gli argomenti delle operazioni o altre informazioni di controllo.

La Figura 2.36 è una possibile risposta alle precedenti richieste. Il diagramma di sequenza è stato suddiviso in due parti per migliorarne la leggibilità (le linee di vita di `Ordine` e `FinestraOrdine` sono ripetute nelle due parti).

Il diagramma di sequenza è di facile interpretazione. I primi due messaggi sono stati spiegati al Passo 13. Il messaggio `accettareConf` produce l'invio del messaggio `prepararеOrdine`. Ciò crea un oggetto temporaneo `:Ordine` che è mostrato dall'oggetto `:FinestraOrdine`.

Se i dettagli dell'ordine sono accettati dal cliente (`inviareOrdine`), `:FinestraOrdine` chiede la memorizzazione (`memorizzareOrdine`) dell'oggetto `:Ordine` in modo persistente. Tale oggetto `:Ordine` collega quindi se stesso al computer ordinato `:Computer` e ai corrispondenti oggetti `:Cliente` e `:Pagamento`. Una volta che tali oggetti sono collegati (nel database), l'oggetto `:Ordine` invia un messaggio `emailOrdine` all'attore esterno `Cliente`.

Si noti la duplice veste in cui appare il concetto di cliente. Esso fa riferimento sia a un attore esterno (che interagisce col sistema) sia a una classe entità che, internamente al sistema, memorizza le caratteristiche dei clienti affinché il sistema stesso possa riconoscerli: una tale dicotomia si presenta sovente nella modellazione di sistemi reali.

2.2.6 Modellazione con statechart

Un *modello d'interazione* fornisce la specifica dettagliata di un caso d'uso. Un *modello statechart* è un'estensione dei modelli basati su automi a stati finiti e permette una dettagliata descrizione delle possibili modifiche dinamiche degli stati di una classe (evoluzione degli oggetti della classe). Tali modifiche tipicamente descrivono il comportamento di un qualunque oggetto della classe a fronte dei casi d'uso che lo utilizzano, cioè ne descrivono la storia.

Lo *stato* di un oggetto è definito dai valori attuali dei suoi attributi (attributi primitivi e attributi di legame). Un oggetto rimane il medesimo durante tutta la sua esistenza (la sua identità non muta mai, Sezione 2.1.1.3) ma il suo stato può modificarsi coerentemente con un *diagramma statechart*. Questo diagramma è un grafo bipartito di *stati* (rettangoli arrotondati) e *transizioni* (archi diretti), queste ultime causate da *eventi*.

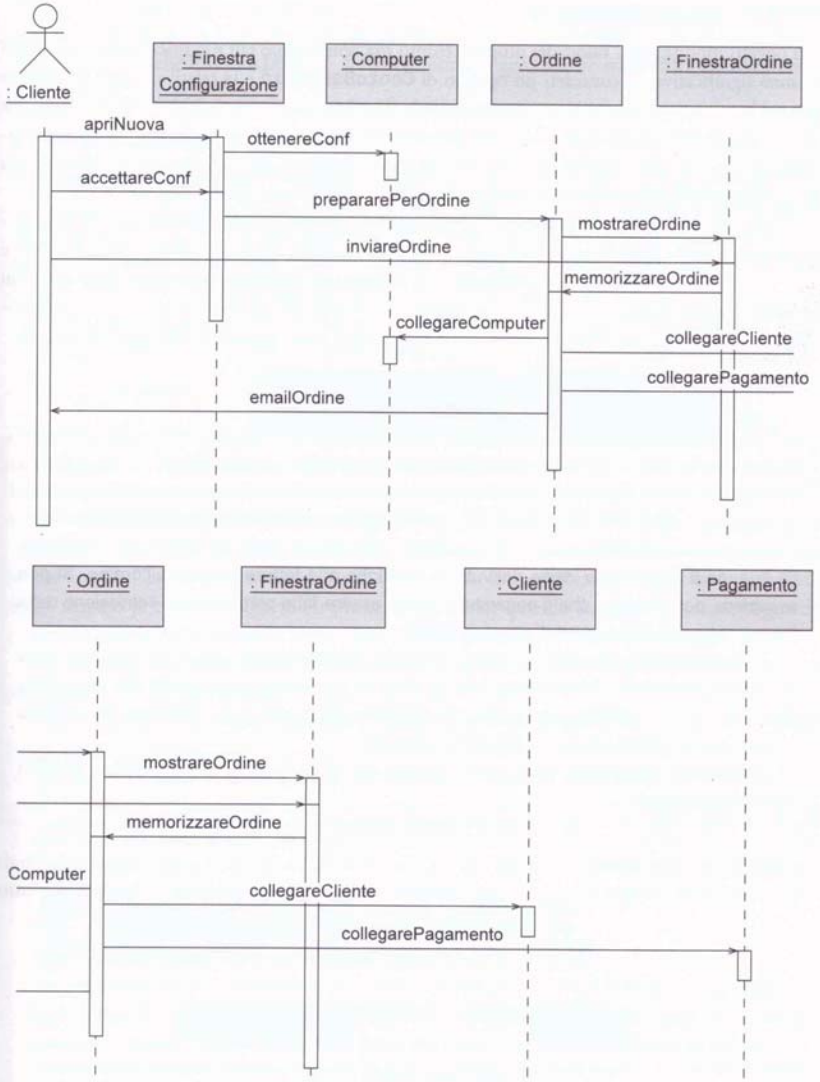


FIGURA 2.36 Diagramma di sequenza per il diagramma di attività "Ordinare computer configurato" (Acquisti OnLine).

2.2.6.1 Stati e transizioni

Gli oggetti modificano i valori dei propri attributi ma non sempre tali modifiche sono *transizioni di stato* significative. Si consideri un oggetto di `ContoBancario` e la relativa regola di business per cui le spese su un conto sono azzerate quando il `saldo` è superiore a 100 000$. Potremo dire che in queste condizioni l'oggetto di `ContoBancario` entra in un uno stato `privilegiato`, altrimenti è in uno stato `normale`. Il `saldo` del conto cambia dopo ogni versamento/prelievo ma lo *stato* muta solo nel momento in cui il saldo supera o scende sotto la soglia di 100 000$.

L'esempio fatto illustra l'essenza della modellazione degli stati. Si definiscono modelli degli stati per le modifiche "significative" dello stato, non per modifiche qualsiasi. Cosa sia "significativo" e cosa non lo sia è un problema modellistico. Un diagramma statechart può essere visto come un modello di *regole di business*: le regole non cambiano (almeno per un certo tempo) e sono indipendenti dai particolari casi d'uso. Infatti, i casi d'uso devono essere conformi alle regole di business.

Analisi: Passo 16 (Acquisti OnLine)

Si consideri la classe `Fattura` dell'applicazione Acquisti OnLine. Dal modello dei casi d'uso è noto che un cliente specifica il metodo di pagamento (carta di credito o assegno) per il computer quando riempie il modulo d'acquisto e lo invia. Questo conduce alla generazione di un ordine e, in seguito, alla preparazione di una fattura. Tuttavia, il modello dei casi d'uso non specifica quando il pagamento viene ricevuto in relazione alla fattura relativa all'ordine. Si può assumere, per esempio, che il pagamento possa essere fatto prima o dopo l'emissione della fattura e che pagamenti parziali siano permessi.

Dal modello delle classi è noto che la fattura per l'ordine è preparata da un venditore ma è gestita dal magazzino. Quest'ultimo invia la fattura al cliente contestualmente alla spedizione del computer. È importante che lo stato di pagamento della fattura sia mantenuto nel sistema in modo che le fatture siano correttamente annotate.

Si disegni il diagramma statechart che rappresenta i possibili stati della fattura tenendo conto dei pagamenti.

La Figura 2.37 è un modello statechart per la classe `Fattura`. Lo stato iniziale di un oggetto di `Fattura` è `Non pagata`. Ci sono due possibili transizioni dallo stato `Non pagata`: l'evento

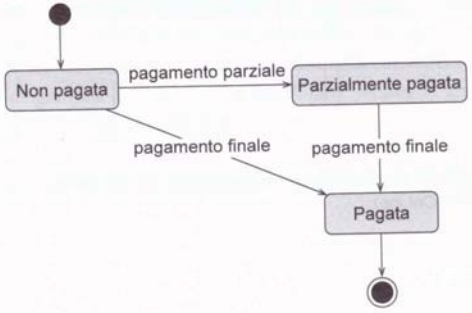


FIGURA 2.37 Stati ed eventi per la classe Fattura (Acquisti OnLine).

`pagamento parziale` determina la transizione dell'oggetto allo stato `Parzialmente pagata` (solo un pagamento parziale è concesso), e l'evento `pagamento finale` (sia dallo stato `Non pagata` che `Parzialmente pagata`) porta l'oggetto nello stato `Pagata`, che è anche lo stato finale.

2.2.6.2 Diagrammi statechart

Un *diagramma statechart* viene normalmente associato a una classe sebbene, in generale, possa essere associato ad altri concetti, ad esempio un caso d'uso. Se associato a una classe, il diagramma indica come gli oggetti di quella classe reagiscono a determinati eventi, cioè quale *azione* fa un oggetto all'accadere di un certo evento. L'azione eseguita dall'oggetto dipende, in generale, dallo stato in cui esso si trova, e determina di solito una modifica dello stato.

La descrizione completa di una *transizione* consiste di tre parti (ogni parte è opzionale, è possibile ometterle tutte se il significato di una transizione è evidente):

```
evento (parametri) [guardia] / azione.
```

Un evento è il verificarsi, in un certo istante, di un fatto che interessa un qualsiasi oggetto. Può avere dei parametri, ad esempio: `tasto_mouse_premuto (tasto_destro)`. L'evento può essere condizionato tramite una condizione, o *guardia*, ad esempio: `tasto_mouse_premuto (tasto_destro) [interno_finestra]`. Solo quando la condizione è vera l'`azione` viene eseguita.

La distinzione tra un evento e una guardia non è sempre evidente. Tale distinzione sta nel fatto che l'*evento* "accade" ma non è detto che l'azione sia immediatamente eseguita, l'evento può essere sospeso prima che l'oggetto sia pronto per gestirlo; quando l'oggetto è pronto, la guardia viene valutata per determinare se la transizione debba scattare (e l'azione essere eseguita).

Mentre un'*azione* è una "breve" computazione eseguita quando la transizione scatta, lo stato può contenere computazioni anche complesse, cioè delle *attività*.

Uno stato può anche essere composto a partire da stati più semplici (*stati annidati*) connessi a loro volta da transizioni. Lo *stato composto* è uno stato astratto: a livello di dettaglio può avere un comportamento complesso ma a livello più generale si presenta come uno stato qualsiasi. Una transizione uscente dai confini dello stato composto significa che può scattare a partire da uno qualunque degli stati annidati. La stratificazione degli stati migliora la chiarezza e la concisione del diagramma.

Analisi: Passo 17 (Acquisti OnLine)

Con riferimento ai passi precedenti, si consideri la classe `Ordine` e si analizzino gli stati nei quali un oggetto di `Ordine` può trovarsi, partendo dalla sua conferma al sistema fino alla sua evasione. Si osservi che un computer ordinato può essere già pronto oppure deve essere configurato per soddisfare i requisiti imposti. Si può anche indicare il giorno in cui si vuole ricevere il computer, anche se questo è già pronto.

È possibile cancellare un ordine fino alla spedizione del computer. Non è necessario rappresentare la penalità che potrebbe essere prevista nel caso di cancellazione tardiva.

Si disegni un diagramma statechart per la classe `Ordine`.

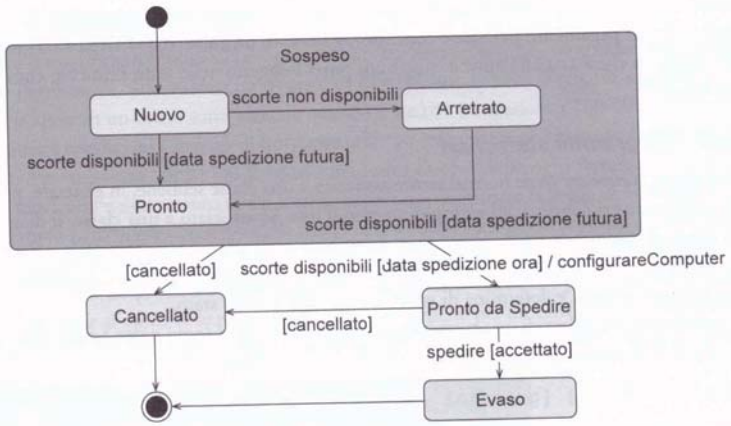


FIGURA 2.38 Diagramma statechart per la classe Ordine (Acquisti OnLine).

Il diagramma statechart per `Ordine` è mostrato in Figura 2.38. Lo stato iniziale è `Nuovo`. Questo è uno degli stati dello stato composto `Sospeso`, gli altri sono `Arretrato` e `Pronto`. Ci sono due transizioni possibili da uno qualunque dei tre stati annidati nello stato `Sospeso`: la transizione verso lo stato `Cancellato` (controllata dalla condizione `[cancellato]`) e la transizione verso lo stato `Pronto da Spedire`. La transizione allo stato `Pronto da Spedire` è etichettata con una descrizione completa contenente l'evento (`scorte disponibili`), la guardia (`data spedizione ora`), e l'azione (`configurareComputer`). Sarebbe anche possibile, senza violare le regole del modello, sostituire le guardie `[cancellato]` con eventi `cancella`.

2.3 Descrizioni dei casi di studio

Per illustrare varie attività di modellazione, si useranno nel testo quattro casi di studio, presentati con lo stesso stile usato per il caso Acquisti OnLine, cioè elencando delle domande cui verranno associate delle soluzioni. Questo permetterà al lettore di tentare una propria soluzione e quindi confrontarla con quelle suggerite. I casi di studio sono:

1. Immatricolazioni universitarie.
2. Video Store.
3. Contatti con i Clienti (CMS).
4. Telemarketing.

2.3.1 Immatricolazioni universitarie

Le Immatricolazioni universitarie costituiscono un classico esempio didattico (vedi Quatrani, 2000; Stevens e Pooley, 2000). È un dominio di applicazione inaspettatamente complesso con un ricco insieme di regole di business che cambiano con una frequenza relativamente alta, e quindi con informazioni storiche da conservare insieme alle loro relazioni con le regole di business valide in quel periodo.

Non ci sono due università che si comportino allo stesso modo. Ciascuna ha le proprie peculiarità che possono essere usate nei casi di studio per enfatizzare aspetti particolari dell'analisi e del progetto del sistema. L'enfasi nel nostro caso di studio è sulla complessità della modellazione di stato rispetto alla gestione della dimensione temporale e alla rappresentazione delle regole di business con strutture dati.

Descrizione del caso 1: Immatricolazioni universitarie

Un'università di medie dimensioni offre svariati curriculum (gradi) di laurea, di master e di dottorato a studenti a tempo pieno e a tempo parziale. La struttura educativa dell'università è composta da divisioni. Le divisioni contengono più dipartimenti. Mentre una singola divisione amministra ogni curriculum, il curriculum può comprendere corsi di altre divisioni. Infatti, l'università vanta un'ampia libertà di scelta dei corsi da parte degli studenti per comporre i loro curriculum.

La flessibilità nella selezione dei corsi richiede uno sforzo da parte del sistema di gestione delle immatricolazioni. Piani di studio liberi e individuali non devono contraddire le regole che governano il curriculum, come i corsi di prerequisito e quelli obbligatori. La scelta del piano può essere limitata da sovrapposizioni d'orario, numero massimo di studenti per corso, etc.

La flessibilità della didattica offerta dall'università è stata la principale causa del costante incremento del numero di studenti. Comunque, per mantenere la sua tradizionale efficienza, il sistema di immatricolazione, ancora parzialmente manuale, deve essere sostituito con una nuova soluzione software. La ricerca preliminare per un package già disponibile sul mercato non ha avuto successo. Il sistema d'immatricolazioni universitarie è talmente particolare da richiedere uno sviluppo specifico.

Il sistema deve essere di supporto alle attività di preimmatricolazione e alle procedure di gestione dell'immatricolazione. Le attività di preimmatricolazione devono comprendere l'invio agli studenti dei risultati degli esami dell'ultimo semestre insieme alle istruzioni per l'immatricolazione. Durante il periodo di immatricolazione, il sistema deve verificare il piano proposto dagli studenti a fronte dei prerequisiti, delle eventuali sovrapposizioni d'orario, del numero massimo di studenti iscritti al corso, etc. La soluzione ad alcuni problemi può richiedere la consultazione con i tutori accademici o con i docenti dei corsi.

2.3.2 Video Store

Il secondo caso di studio è una normale applicazione di business, tipica dei commerci al minuto, che riguarda il supporto alle operazioni di un piccolo Video Store. Il Video Store gestisce in magazzino un'ampia scelta di nastri o dischi video di film attuali e popolari. La principale operazione del Video Store è l'affitto dei video.

Un tipico sistema computerizzato a supporto del Video Store potrebbe essere derivato da software commerciale già disponibile sul mercato o da qualche altro software proprietario preesistente. Il sistema dovrebbe usare un DBMS commerciale disponibile su personal computer ed essere inizialmente posto su una macchina singola. Lo sviluppo della interfaccia GUI dovrebbe essere fatto usando un semplice linguaggio di quarta generazione (4GL) con gestione dello schermo, possibilità di generazione codice e connessione al database.

Un aspetto caratteristico del Video Store come caso di studio è la lunga catena d'attività, dall'ordine dei video con la gestione del magazzino alla contabilità associata con l'affitto dei video. In qualche modo, si tratta di una *catena del valore* su piccola scala (si veda la Sezione 1.2.2).

Descrizione del caso 2: Video Store

Un nuovo Video Store intende offrire in affitto nastri e dischi video a un vasto pubblico. La gestione aziendale vuole disporre del supporto di un sistema computerizzato per le sue attività e dispone delle specifiche di svariati package software che potrebbero essere sviluppati e adattati per configurare il sistema. Per la selezione dei package, Video Store ha assunto un analista il cui compito è la determinazione e la specifica dei requisiti.

Video Store inizialmente dispone di un magazzino di circa un migliaio di nastri e cinquecento dischi video. La merce è stata ordinata da un fornitore ma in futuro saranno contattati più fornitori. Tutti i nastri e dischi video saranno codificati con un codice a barre in modo da integrare un lettore di codici nel sistema per gestire l'affitto e i rientri. Anche le carte clienti verranno codificate a barre.

I clienti potranno riservare i video per il ritiro in una data specifica. Il sistema deve essere dotato di un flessibile motore di ricerca per rispondere alle richieste dei clienti, incluse domande su film che Video Store non ha in magazzino (e può ordinare su richiesta).

2.3.3 Gestione Contatti con i Clienti (CMS)

La Gestione Contatti con i Clienti, spesso nota con l'acronimo CMS (Contact o Customer Management System), è un dominio applicativo "attuale", ed è una componente importante di ogni sistema per la gestione di un'azienda di tipo ERP.

I sistemi ERP sono complessi "package" applicativi ampiamente configurabili che automatizzano le cosiddette transazioni di *back-office* (applicazioni dedicate alla gestione e non all'interazione con l'ambiente esterno). Tre componenti tipiche di un sistema ERP sono: contabilità, produzione e risorse umane (il CMS appartiene alla componente risorse umane). Evidentemente, anche la componente CMS di una soluzione ERP è in genere molto complessa, per cui in questo caso di studio ne verranno trattati solo alcuni aspetti.

Le applicazioni CMS sono caratterizzate da interfacce complesse attraverso cui gli impiegati nel dipartimento che si occupa delle relazioni con i clienti possono gestire i contatti. Il sistema agisce come un'agenda per registrare le attività e gli eventi in relazione al cliente, tenendo traccia dell'avanzamento dei contatti stessi.

L'agenda deve essere basata su un database per permettere l'ordinamento dinamico e il monitoraggio di attività ed eventi da parte di più impiegati. Come la maggior parte di sistemi di gestione delle risorse umane, le applicazioni CMS richiedono schemi sofisticati di autorizzazione per il controllo dell'accesso alle informazioni riservate.

Descrizione del caso 3: CMS

Una società di ricerche di mercato ha un archivio di organizzazioni che acquistano rapporti di analisi di mercato. Alcuni grandi clienti hanno anche acquistato dalla società del software specializzato per la generazione di rapporti. Tali clienti sono stati forniti di dati grezzi e pre-aggregati per generare in proprio i rapporti.

La società è costantemente alla ricerca di nuovi clienti, anche se questi possono essere solo interessati a sporadici rapporti su commissione e per mercati di nicchia. Poiché i potenziali clienti non sono ancora clienti, la società preferisce chiamarli contatti, da cui il nome del sistema CMS (i contatti sono diretti a clienti potenziali, attuali e passati).

Un nuovo sistema CMS deve essere sviluppato internamente ed essere disponibile a tutti gli impiegati della società ma con vari livelli di accesso. Gli impiegati del dipartimento Servizio Clienti saranno i gestori del sistema. Il sistema permetterà una programmazione flessibile delle attività legate ai contatti in modo che gli impiegati possano collaborare fruttuosamente per conquistare nuovi clienti e per rafforzare le relazioni esistenti.

2.3.4 Telemarketing

Molte organizzazioni "piazzano" i loro prodotti e servizi attraverso il telemarketing, cioè prendendo contatto direttamente con il cliente per telefono. Un sistema di telemarketing deve fornire il supporto per un complesso processo di programmazione delle chiamate che gli impiegati devono effettuare, per facilitare la conversazione e per registrare i risultati delle conversazioni stesse.

I sistemi di telemarketing si basano essenzialmente sulle capacità del database di gestire le code delle chiamate, permettendo di riordinarle in modo dinamico e supportando più conversazioni concorrenti. Un altro aspetto interessante è la capacità di chiamare automaticamente i numeri telefonici in lista di attesa.

Descrizione del caso 4: Telemarketing

Un'organizzazione caritatevole vende biglietti di lotteria per raccogliere fondi attraverso "campagne" pubblicitarie di supporto alla vendita. La società ha una lista dei contribuenti passati (sostenitori). Per ogni nuova campagna, un sottoinsieme di tali sostenitori è pre-selezionato per il telemarketing e/o per un contatto postale.

La società usa alcuni schemi innovativi per guadagnare nuovi sostenitori. Tali schemi comprendono speciali campagne con "bonus" per ricompensare i sostenitori per i loro acquisti, per aver attratto nuovi sostenitori etc. La società non sceglie casualmente i potenziali sostenitori da elenchi telefonici o mezzi simili.

Per supportare il proprio lavoro, la società ha deciso di far sviluppare una nuova applicazione di telemarketing. Il nuovo sistema dovrà supportare fino a cinquanta impiegati simultanei. Il sistema deve essere in grado di ordinare nelle liste le chiamate sulla base di priorità predefinite e altri vincoli.

Il sistema deve chiamare i numeri telefonici nell'ordinamento fornito. Connessioni fallite devono essere rimesse in coda e riprovate successivamente. Anche le chiamate dei sostenitori devono essere trattate. I risultati delle conversazioni, compresi gli ordini di biglietti e le modifiche ai dati dei sostenitori, devono essere gestite.

Sommario

Questo capitolo ha trattato molti elementi e concetti di base. Si sono introdotti e spiegati i concetti e la terminologia fondamentale della tecnologia a oggetti. Sono stati fatti molti esempi per spiegare la modellazione a oggetti, compreso un completo studio di caso. Per un lettore non familiare all'argomento lo sforzo compiuto dovrebbe essere ripagato nei prossimi capitoli.

Un sistema a oggetti consiste di *oggetti* che collaborano. Ogni oggetto ha uno stato, un comportamento e un'identità. Quest'ultima può essere la caratteristica più essenziale per una corretta comprensione dei sistemi a oggetti, è anche la più difficile da apprezzare per le persone con un qualche bagaglio d'esperienza nelle applicazioni computerizzate convenzionali. Navigare attraverso *legami* è il "modus operandi" della tecnologia a oggetti, qualcosa che un lettore formato sulla tecnologia relazionale avrà qualche difficoltà a digerire.

Una classe è uno *stampo* per la creazione di oggetti. Definisce gli *attributi* che un oggetto contiene e le *operazioni* che su un oggetto possono essere invocate. Gli attributi possono avere un tipo primitivo o possono avere classi come tipo. Questi ultimi definiscono delle *associazioni*. L'associazione è una tipologia di relazione tra classi. Altre tipologie di relazione sono l'*aggregazione* e la *generalizzazione*.

La relazione di generalizzazione fornisce la base per il *polimorfismo* e l'*ereditarietà*. Gli ambienti commerciali di programmazione possono fornire un supporto all'*ereditarietà multipla* ma non è comune che lo stesso supporto si abbia per la *classificazione multipla* o *dinamica*. In relazione all'ereditarietà, esiste la nozione di *classe astratta*. Una classe può avere attributi o operazioni applicabili solo alla classe stessa non alle sue istanze. Tali attributi e operazioni necessitano della nozione di *oggetto classe*.

I concetti della tecnologia a oggetti sono stati enfatizzati nell'esempio centrato sullo sviluppo dell'applicazione Acquisti OnLine. Il caso di studio ha introdotto l'intera collezione di tecniche modellistiche di UML: modellazione dei casi d'uso, delle attività, delle classi, dell'interazione e di statechart. Le tecniche sono state presentate a un livello relativamente astratto e molti dettagli sono stati omessi (verranno ripresi nei capitoli seguenti).

Infine, sono state introdotte le specifiche di quattro *casi di studio* che verranno usati nel seguito (insieme al caso Acquisti OnLine) per esemplificare e spiegare analisi e progetti più sofisticati. I casi di studio riguardano i seguenti domini d'applicazione: immatricolazione universitarie, gestione di un video store, gestione dei contatti con i clienti, e telemarketing.

Domande

D1 Perché è necessario distinguere tra oggetto istanza e oggetto classe?

D2 Che cos'è un identificatore d'oggetto?

D3 Qual è la differenza tra un oggetto transitorio e uno persistente?

D4 Cosa sono un legame transitorio e un legame persistente? Come vengono usati durante l'esecuzione di un programma?

D5 Cosa significa che il tipo di un attributo è una classe? Fare un esempio.

D6 Perché, in un buon modello a oggetti, molti attributi sono privati mentre molte operazioni sono pubbliche?

D7 Qual è la differenza tra la visibilità di un'operazione e la sua applicabilità?

D8 In quali situazioni deve essere usata una classe d'associazione? Dare un esempio.

D9 Qual è la differenza tra la composizione e l'aggregazione?

D10 Si spieghi l'affermazione "in un tipico ambiente di programmazione a oggetti l'ereditarietà si applica alle classi e non agli oggetti".

D11 Qual è il legame tra la modifica delle operazioni e il polimorfismo?

D12 Che cos'è la segnatura di un'operazione?

D13 In che modo la classificazione multipla si differenzia dall'ereditarietà multipla?

D14 Quali benefici si possono trarre dall'uso di una classe astratta?

D15 Spiegare principali caratteristiche e differenze tra modello dello stato, modello del comportamento e modello del cambiamento di stato.

D16 Un attore può avere attributi e operazioni? Spiegare.

D17 Spiegare il ruolo dei diagrammi di attività nella modellazione di sistema.

D18 Qual è la differenza tra separazione (branch) e biforcazione (fork) nel diagramma d'attività? Fare un esempio.

D19 Cosa sono le classi entità?

D20 Cosa sono un argomento attuale e un argomento formale?

D21 Qual è la differenza tra un'azione e un'attività nel diagramma statechart? Fare un esempio.

Esercizi

E1 Si consideri la Figura 2.13 (Sezione 2.1.3.2). Si ipotizzi che un corso offerto comprenda lezioni e seminari e che sia possibile che un docente sia incaricato della parte istituzionale e un altro, distinto, sia incaricato della parte seminariale. Modificare il diagramma di Figura 2.13 per rappresentare il precedente fatto.

E2 Si consideri la Figura 2.15 (Sezione 2.1.3.4). Si fornisca un modello alternativo che non usi né una classe d'associazione né un'associazione ternaria. Si descrivano le differenze semantiche, se ne esistono, tra il modello in Figura 2.15 e il nuovo modello.

E3 Si consideri la Figura 2.19 (Sezione 2.1.5.2.1). Estendere l'esempio aggiungendo opportuni attributi alle classi `Insegnante`, `Studente`, `StudenteDottorato` e `Tutore`.

E4 Acquisti OnLine - Si consideri il Passo 2 (Sezione 2.2.2.2). Il punto 6 nella Tabella 2.1 stabilisce che al cliente debba essere inviato un e-mail in modo che possa controllare online lo stato di un ordine. Non c'è un caso d'uso che consideri tale fatto. Dovrebbe esserci? Spiegare.

E5 Acquisti OnLine - Si consideri il Passo 2 (sezione 2.2.2.2) e il Passo 3 (sezione 2.2.2.4). Si scriva un documento di caso d'uso per il caso `Aggiornare Stato Ordine`. Il caso d'uso deve permettere a un cliente di controllare lo stato dell'ordine del suo computer. Il nome di tale caso d'uso riflette la sua funzionalità? Conviene cercare un nome diverso?

E6 Acquisti OnLine - Si consideri il Passo 9 (Sezione 2.2.4.3). Nella Figura 2.29 la classe `Cliente` non è direttamente associata con le classi `Pagamento`, `Fattura`, `ComputerConfigurato`. Dovrebbe essere associata? Se così fosse, modificare il diagramma. Spiegare.

E7 Acquisti OnLine - Si consideri il Passo 11 (Sezione 2.2.4.5). La Figura 2.31 è un esempio, relativamente semplice, che illustra la generalizzazione. Quali complicazioni sorgerebbero se vi fossero delle differenze tra le informazioni sulle fatture per un `ComputerStandard` e per un `ComputerConfigurato`? Per esempio, potrebbero esserci costi aggiuntivi per modifiche richieste ai sistemi `ComputerConfigurato` e uno sconto per acquisti all'ingrosso di sistemi `ComputerStandard`. Modificare il diagramma per riflettere tali complicazioni. Spiegare brevemente.

E8 Acquisti OnLine - Si consideri il Passo 13 (Sezione 2.2.5.1) e il Passo 6 (Sezione 2.2.3.2). Si rappresenti un diagramma di sequenza per l'attività `Mostrare Modulo Acquisto` (Figura 2.26).

E9 Acquisti OnLine - Si consideri una soluzione dell'Esercizio 8 precedente. Si aggiungano le operazioni alle classi denotate dagli oggetti nel diagramma di sequenza.

E10 Acquisti OnLine - Si consideri il Passo 16 (Sezione 2.2.6.1). Il diagramma statechart in Figura 2.37 è coerente con il vincolo secondo cui solo un pagamento parziale è permesso. Si ipotizzi che questo non sia più vero e più pagamenti parziali siano permessi. Modificare conseguentemente il diagramma, fornendo due soluzioni: la prima per il caso in cui i pagamenti parziali sono a priori specificati come parziali, la seconda per il caso in cui è il sistema che deve verificare se il pagamento è parziale o meno.

Capitolo 3

Determinare i requisiti

La determinazione dei requisiti richiede capacità sociali, di comunicazione e di gestione non indifferenti. Questa è la fase meno tecnica dello sviluppo di un sistema ma, se non accuratamente svolta, può avere conseguenze più serie che per altre fasi. Nelle fasi successive, i costi per gestire requisiti omessi o non correttamente interpretati possono diventare insostenibili.

Questo capitolo presenta un ampio spettro di problemi nella determinazione dei requisiti. La prima parte del capitolo riguarda i metodi di raccolta, negoziazione (discussione) e validazione dei requisiti con gli utenti, ed anche la gestione dei requisiti stessi. Quest'ultimo aspetto comprende i problemi di tracciabilità (*traceability*) e gestione del cambiamento (*change management*), argomenti che saranno approfonditi nel Capitolo 10.

La seconda parte del capitolo introduce semplici tecniche di rappresentazione grafica per descrivere i modelli di business dell'organizzazione in esame. Viene anche trattata la struttura del documento dei requisiti.

3.1 Principi di determinazione dei requisiti

La determinazione dei requisiti è la prima fase del ciclo di vita per lo sviluppo di un sistema. Il sistema da sviluppare è definito dalle attività di *pianificazione di sistema* (Sezione 1.2). L'obiettivo della determinazione dei requisiti è di fornire una definizione informale dei requisiti, funzionali e non, che gli stakeholder si aspettano di ritrovare nel sistema implementato.

I requisiti definiscono i *servizi* attesi dal sistema e i *vincoli* cui il sistema deve sottostare. I servizi possono raggrupparsi in: *requisiti di sistema* (descrivono lo scopo del sistema), *requisiti funzionali* (descrivono le funzioni di business), e *requisiti informativi* (descrivono le strutture di dati). I vincoli possono essere classificati secondo le diverse categorie di restrizioni poste sul sistema quali i comportamenti voluti, la sicurezza e le prestazioni richieste, etc.

I requisiti devono essere ottenuti dai clienti (utenti e proprietari del sistema). L'attività di raccolta è svolta da un analista di business (o di sistema) utilizzando tecniche diverse, che possono spaziare dalle tradizionali interviste ai clienti andando (se necessario) fino alla costruzione di prototipi.

I requisiti raccolti devono essere soggetti a un'attenta analisi per eliminare duplicazioni e contraddizioni. Questo compito conduce, in genere, a *revisioni e rinegoziazioni* dei requisiti con i clienti che li hanno espressi.

Una volta accettati dai clienti, i requisiti sono definiti, classificati, numerati, e ordinati (secondo certe priorità) nel *documento dei requisiti*. Il documento è strutturato seguendo una traccia scelta dall'organizzazione per la documentazione dei requisiti stessi.

Sebbene il documento dei requisiti sia principalmente una specifica informale (in linguaggio naturale), esso contiene un modello schematico ad alto livello del business. Tale documento descrive lo *scopo del sistema* e comprende normalmente un *modello dei casi d'uso* e un *modello delle classi*.

Poiché i requisiti del cliente sono costantemente in evoluzione, è necessario prepararsi a gestire tale cambiamento. La *gestione dei requisiti* comprende quindi attività come la stima dell'impatto che la modifica di un requisito ha su altri requisiti e sul resto del sistema.

3.2 Raccolta dei requisiti

Un *analista di business* raccoglie i requisiti del sistema attraverso discussioni che coinvolgono, in genere, *clienti* ed *esperti* nel dominio del sistema in analisi. L'aiuto dell'esperto può non essere necessario se l'analista di business ha una sufficiente esperienza sul dominio. L'`Analista di Business` è allora una tipologia di `Esperto del Dominio`, come rappresentato con una relazione di generalizzazione nella Figura 3.1 (notare che la Figura 3.1 usa, per convenienza, una notazione informale che non ha nessun rapporto con modelli di casi d'uso!).

I requisiti dedotti dagli esperti del dominio costituiscono la *conoscenza del dominio*. In pratica, essi esprimono *regole di business* in genere ben note ed applicabili alla maggioranza delle organizzazioni di quel certo tipo. I requisiti dedotti dai clienti sono espressi negli scenari dei casi d'uso. Questi vanno oltre la conoscenza di base del dominio e rappresentano il carattere unico dell'organizzazione, il suo modo attuale di "fare business".

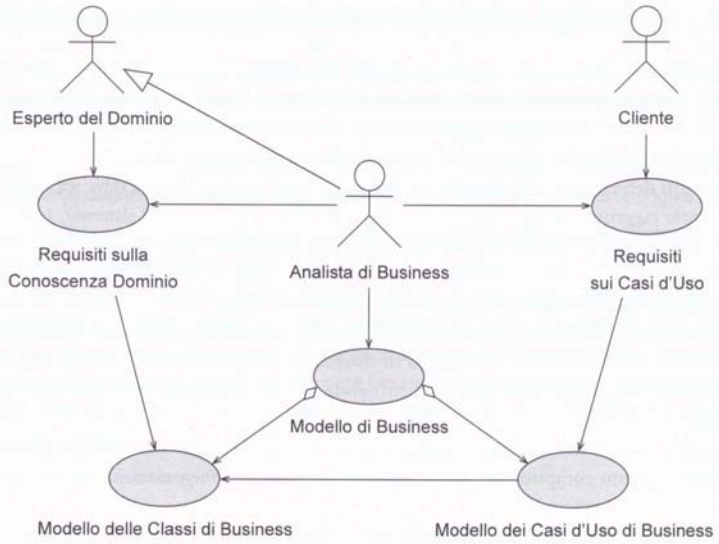


FIGURA 3.1 Influenze durante la determinazione dei requisiti.

Il compito di un analista di business è di combinare i due insiemi di requisiti in un `Modello di Business` che, come mostrato in Figura 3.1, comprende un `Modello delle Classi di Business` e un `Modello dei Casi d'Uso di Business`. Il `Modello delle Classi di Business` è un diagramma delle classi ad alto livello che identifica e mette in relazione gli *oggetti del business*. Il `Modello dei Casi d'Uso di Business` è un diagramma di caso d'uso ad alto livello che identifica le principali funzionalità del sistema.

In generale, le classi del dominio (oggetti del business) non devono essere necessariamente derivate dai casi d'uso (vedi Rumbaugh, 1994). In pratica, un `Modello delle Classi di Business` dovrebbe essere confrontato con il `Modello dei Casi d'Uso di Business`. Tale verifica potrebbe provocare aggiustamenti o estensioni nel `Modello delle Classi di Business`.

Seguendo Hoffer *et al.* (1996), si distinguerà tra metodi tradizionali e moderni per la ricerca di fatti e la raccolta di informazioni.

3.2.1 Metodi tradizionali per la raccolta dei requisiti

I metodi tradizionali per la raccolta dei requisiti comprendono *interviste*, *questionari*, *osservazioni* e *studio dei documenti* del business. Questi metodi sono semplici e redditizi, e la loro efficacia è "inversamente proporzionale" al *rischio* legato al progetto. Un alto rischio implica che il sistema è difficile da implementare, perfino i requisiti di alto livello possono essere poco chiari e, probabilmente, i metodi tradizionali non sono sufficienti.

3.2.1.1 Interviste

Le *interviste* sono una tecnica fondamentale per la ricerca di fatti e la raccolta d'informazioni. Le interviste, in genere, sono condotte con i clienti e forniscono principalmente informazioni sulle funzionalità del sistema in esame: per questo sono esprimibili attraverso un `Modello dei Casi d'Uso di Business` (Figura 3.1). Anche gli esperti del dominio possono essere intervistati se l'analista non ha una sufficiente conoscenza del dominio.

Le interviste con i clienti sono complesse (Kotonya e Sommerville, 1998; Sommerville e Sawyer, 1997). I clienti possono avere solo una vaga idea dei propri requisiti, essere poco disponibili alla cooperazione o non in grado di esprimere i propri requisiti in termini comprensibili. Potrebbero anche richiedere requisiti che creano uno sforamento dei costi di progetto o che non sono implementabili. Infine, è probabile che requisiti di clienti distinti presentino dei conflitti.

Ci sono due tipologie fondamentali di interviste: strutturate (formali) e non strutturate (informali). Un'*intervista strutturata* è preparata in anticipo, ha un'agenda e molte domande sono predeterminate. Alcune di queste domande possono essere *aperte* (non è possibile prevedere le risposte), altre *chiuse* (con risposta selezionabile da un insieme di possibili risposte già fornite).

Le interviste strutturate sono generalmente accompagnate da *interviste non strutturate* che assomigliano a incontri informali, senza domande anticipate o obiettivi predeterminati. Lo scopo è di incoraggiare il cliente a parlare di ciò che ha in mente, su cui l'analista potrebbe non aver riflettuto e su cui, di conseguenza, non avrebbe fatto domande.

Sia nelle interviste strutturate sia in quelle non strutturate si deve fornire un qualche punto di partenza ed un contesto per la discussione. Ciò può avvenire con un breve documento scritto inviato (ad esempio, con un e-mail) all'intervistato prima dell'incontro, spiegando gli obiettivi dell'intervistatore e/o ponendo alcuni quesiti di base.

In generale, dovrebbero essere *evitate* tre tipologie di domande (vedi Whitten e Bentley, 1998):

- Domande nelle quali l'intervistatore esprime (direttamente o indirettamente) la propria opinione sul problema ("dobbiamo fare le cose nel modo in cui le facciamo?").
- Domande prevenute, simili alle precedenti, eccetto che l'opinione dell'intervistatore è chiaramente espressa ("non intendi fare questo, non è vero?")
- Domande contenenti già la risposta ("devi fare le cose in questo modo, non è vero?").

Ci sono molti fattori per il successo di un'intervista ma, forse, il più importante è la capacità dell'intervistatore di *comunicare e stabilire rapporti interpersonali*. Oltre che porre domande, è importante ascoltare attentamente ed essere pazienti, in modo che l'intervistato sia a suo agio. Per mantenere un buon rapporto interpersonale, un memorandum che sintetizzi i risultati dell'intervista dovrebbe essere inviato all'intervistato uno o al massimo due giorni dopo, tenendo poi nel debito conto le osservazioni e le precisazioni che potrebbero emergere.

3.2.1.2 Questionari

I *questionari* sono un modo efficiente per raccogliere informazione da molti clienti. I questionari sono normalmente usati in aggiunta alle interviste, non al loro posto. Un'eccezione potrebbe essere un progetto a basso rischio con obiettivi ben definiti. In tal caso, la natura passiva e meno approfondita di un questionario potrebbe essere sufficiente.

In generale, i questionari sono meno produttivi delle interviste poiché nessun chiarimento dei quesiti o delle risposte è possibile. Questo è sia un vantaggio sia uno svantaggio. È un vantaggio perché chi risponde ha tempo per considerare le risposte e può rimanere anonimo. È uno svantaggio perché chi risponde non ha l'opportunità di discutere e approfondire i quesiti.

Un questionario dovrebbe essere progettato al fine di semplificare la risposta ai quesiti. In particolare, la maggior parte dei quesiti dovrebbe essere chiusa: bisognerebbe evitare i quesiti aperti. I *quesiti chiusi* possono assumere tre forme (Whitten e Bentley, 1998):

- *Quesiti a scelta multipla,* dove chi risponde deve scegliere una o più risposte tra un insieme di risposte date. Commenti aggiuntivi possono essere permessi.
- *Quesiti a punteggio,* dove è possibile esprimere la propria opinione su un'affermazione. Punteggi possibili possono essere: forte accordo, accordo, neutrale, disaccordo, forte disaccordo, e non so.
- *Quesiti con ordinamento,* dove le risposte fornite dovrebbero essere ordinate (per importanza) con numeri in sequenza, valori percentuali, o simili meccanismi di ordinamento.

Un questionario ben progettato, semplice da completare, incoraggerà chi risponde a restituire prontamente un documento completo. Comunque, valutando i risultati del questionario, l'analista dovrebbe considerare possibili distorsioni dovute al fatto che chi non ha risposto avrebbe probabilmente voluto dare risposte differenti (vedi Hoffer *et al.*, 1996).

3.2.1.3 Osservazioni

Ci sono situazioni nelle quali l'analista trova difficile ottenere un'informazione completa attraverso interviste o questionari. Il cliente potrebbe essere incapace di fornire direttamente l'informazione o potrebbe averne solo una conoscenza frammentata (rispetto, ad esempio, all'intero processo di business). In tali casi, l'*osservazione* può essere una tecnica efficace di ricerca dei fatti. Dopo tutto, il miglior modo di apprendere come annodare una cravatta è attraverso l'osservazione del processo!

L'*osservazione* può assumere due forme:

- *Osservazione passiva,* dove l'analista osserva le attività di business senza interruzioni o coinvolgimento diretto del cliente che lavora. In alcuni casi, una videocamera potrebbe essere usata per un'osservazione poco intrusiva.
- *Osservazione attiva,* dove l'analista partecipa direttamente alle attività ed entra a far parte del gruppo di lavoro.

Per essere significative, le osservazioni dovrebbero essere svolte per periodi prolungati, in differenti intervalli temporali e con differenti carichi di lavoro. La principale difficoltà che si incontra è che le persone tendono a comportarsi differentemente quando sono osservate. In particolare, tendono a lavorare seguendo regole e procedure formali. Questo distorce la realtà nascondendo qualsiasi "scorciatoia" (positiva o negativa) messa generalmente in atto, e quindi anche eventuali suggerimenti che tali comportamenti "disinvolti" potrebbero fornire.

3.2.1.4 Studio dei documenti e dei sistemi software

Lo *studio di documenti* e dei *sistemi software* esistenti nell'azienda è una tecnica importante per approfondire la conoscenza del dominio e per ricercare i requisiti legati ai casi d'uso (si veda la Sezione 3.2). La tecnica è sempre usata sebbene possa solo individuare aspetti particolari del sistema.

Diversi *requisiti dei casi d'uso* possono essere individuati attraverso l'analisi dei documenti organizzativi esistenti e dei moduli/rapporti di sistema (se già esiste una soluzione computerizzata, come accade normalmente nelle medio/grandi organizzazioni). Uno dei modi più utili per approfondire la comprensione dei requisiti dei casi d'uso è la consultazione dell'archivio (se esiste) dei difetti e delle richieste di modifica relativi al sistema esistente.

I *documenti organizzativi* da studiare comprendono: modulistica aziendale (compilata, se possibile), descrizioni delle procedure lavorative e delle politiche di intervento, piani di business, organigrammi, corrispondenza tra uffici, minute d'incontri, registrazioni contabili, corrispondenza esterna, lamentele clienti, etc.

I *moduli e rapporti di sistema* da studiare includono schermate e rapporti computerizzati (con la relativa documentazione), manuali operativi di sistema, documentazione utente, documentazione tecnica, modelli di analisi e progetto etc.

La conoscenza del dominio può essere approfondita tramite la consultazione di riviste e libri di riferimento. Gli studi di pacchetti software proprietari, come i sistemi ERP per la gestione dell'azienda, possono fornire anch'essi una ricca conoscenza del dominio. Ovviamente, lo studio di questi sistemi richiede la consultazione di materiali illustrativi presso i produttori: la rete Internet permette di portare a termine molte di queste "visite" senza lasciare l'ufficio.

3.2.2 Metodi moderni di raccolta dei requisiti

I metodi moderni di raccolta dei requisiti comprendono l'uso di prototipi software, lo sviluppo cooperativo delle applicazioni (JAD - Joint Application Development) e lo sviluppo rapido delle applicazioni (RAD - Rapid Application Development). Queste tecniche offrono una migliore comprensione dei requisiti ma con costi e sforzi decisamente superiori, che potrebbero comunque essere ampiamente ripagati a lungo termine.

I metodi moderni sono tipicamente impiegati quando il *rischio* del progetto è alto. I fattori che rendono il progetto ad alto rischio sono molti e comprendono: obiettivi non chiari, procedure non documentate, requisiti instabili, incerta perizia dell'utente, sviluppatori inesperti, impegno dell'utente insufficiente, etc.

3.2.2.1 Prototipi

Lo *sviluppo di prototipi* è il più usato metodo moderno per la raccolta dei requisiti. I prototipi software sono costruiti per visualizzare il sistema, o solo una sua parte, ai clienti, al fine di valutarne la reazione.

Un *prototipo* è un sistema dimostrativo, un modello "sporco e veloce" della soluzione che presenta un'interfaccia grafica (GUI) e simula il comportamento del sistema relativo a vari eventi utente. Il contenuto informativo delle schermate GUI è direttamente codificato nel prototipo piuttosto che dinamicamente ottenuto dal database.

La complessità (e le crescenti aspettative del cliente) delle moderne interfacce GUI rendono il prototipo un elemento importante nello sviluppo del software. La fattibilità e l'utilità del sistema possono essere valutate attraverso prototipi prima che la reale implementazione sia intrapresa.

In generale, un prototipo di sistema è un modo efficace di dedurre requisiti difficilmente ottenibili dai clienti con altri meccanismi. Ciò accade spesso per sistemi che devono fornire nuove funzionalità di business. È il caso di usarli anche con requisiti in conflitto o quando esistono problemi di comunicazione tra clienti e sviluppatori.

Esistono due tipologie di prototipo (vedi Kotonya e Sommerville, 1998):

- *Prototipo "usa e getta"*, scartato quando la raccolta dei requisiti è completata (è specifico della fase di determinazione dei requisiti e riguarda i requisiti difficilmente comprensibili).
- *Prototipo evolutivo*, che è usato anche dopo la fase di determinazione dei requisiti per costruire il prodotto finale. Il prototipo evolutivo punta a incrementare la velocità di fornitura del prodotto. Tipicamente si concentra sui requisiti meglio compresi in modo che la prima versione del prodotto possa essere fornita velocemente (sebbene con funzionalità incomplete).

Un'argomentazione in favore di prototipi "usa e getta" è che si evita il rischio di ritrovare soluzioni "sporche" o inefficienti nel prodotto finale. Comunque, la potenza e la flessibilità degli attuali strumenti di produzione del software hanno indebolito tale argomentazione: attualmente, in un progetto ben gestito, soluzioni prototipali inefficienti possono essere trattate adeguatamente.

3.2.2.2 Sviluppo cooperativo delle applicazioni (JAD)

Riguarda un tipo di sviluppo delle applicazioni che "mette insieme" tutti gli stakeholder (clienti e sviluppatori) (Wood e Silver, 1995). Sebbene si sia incluso JAD nei moderni metodi di raccolta dei requisiti, tale tecnica fu introdotta (da IBM) sul finire degli anni '70.

Ci sono molte società di consulenza che offrono un servizio di organizzazione e gestione di una sessione JAD. Un incontro JAD può essere di poche ore, di qualche giorno o anche di due settimane. Il numero di partecipanti non dovrebbe superare un limite compreso tra 25 e 30. I partecipanti all'incontro sono (vedi Hoffer *et al.*, 1996; Whitten e Bentley, 1998):

- Il *Leader*, è la persona che dirige e modera l'incontro. Deve avere eccellenti capacità di comunicazione e una buona conoscenza del dominio del business (ma non necessariamente una buona conoscenza dello sviluppo del software). Il leader non è uno stakeholder (a parte il fatto di essere il leader JAD).
- Il *Segretario*, è la persona che registra la sessione JAD su computer. Dovrebbe avere capacità di scrittura al terminale e un'ottima conoscenza dello sviluppo del software. Il segretario può usare strumenti CASE per documentare la sessione e sviluppare modelli iniziali di soluzione.
- I *Clienti* (*utenti e manager*), sono i principali partecipanti, i quali mettono in comune e discutono requisiti, prendono decisioni, approvano obiettivi di progetto, etc.
- Gli *Sviluppatori*, sono gli analisti di business e altri membri del gruppo di sviluppo. Ascoltano piuttosto che intervenire, partecipano all'incontro per scoprire fatti e raccogliere informazione, non per influenzare il processo.

La tecnica JAD è basata sulle dinamiche di gruppo. La "sinergia di gruppo" probabilmente produce migliori soluzioni ai problemi. I gruppi incrementano la produttività, apprendono più velocemente, producono giudizi più equilibrati, eliminano più facilmente gli errori, possono prendere le decisioni più rischiose (questo non sempre è positivo!), orientano l'attenzione dei partecipanti sui problemi più importanti, integrano le persone, etc. Quando condotte "a regola d'arte", le sessioni JAD tendono a fornire risultati sorprendentemente buoni. Ma è necessario agire con attenzione: "...Ford Motor Co. nel 1950 visse un'esperienza disastrosa di vendite con Edsel, un'auto progettata da un comitato" (Wood e Silver, 1995, p. 176.)

3.2.2.3 Sviluppo rapido delle applicazioni (RAD)

Il RAD è molto più che un metodo di raccolta dei requisiti, è un vero e proprio approccio allo sviluppo del software (Hoffer *et al.*, 1996). Come dice il nome, il RAD punta a dare velocemente delle soluzioni. L'eccellenza tecnica è secondaria rispetto alla velocità di fornitura di una soluzione. Secondo Wood e Silver (1995), il RAD combina cinque tecniche:

- *Prototipo evolutivo* (Sezione 3.2.2.1).
- *Strumenti CASE* con possibilità di generazione di codice e sviluppo integrato tra modelli di progetto e codice (*round-trip engineering*).
- Un *gruppo di specialisti con strumenti avanzati di sviluppo* (SWAT, *Specialists with Advanced Tools*): è il gruppo di sviluppo RAD che dovrebbe comprendere i migliori analisti, progettisti e programmatori dell'organizzazione. Il gruppo lavora con tempi ridotti e a stretto contatto con gli utenti.
- *JAD Interattivo*, sessione JAD (Sezione 3.2.2.2) durante la quale il segretario è sostituito dal gruppo SWAT con strumenti CASE.
- *Timeboxing*, metodo di gestione progetti che impone un periodo di tempo fissato (timebox) al gruppo SWAT per completare il progetto (se il progetto è in ritardo, gli obiettivi della soluzione sono ridotti per permettere comunque il completamento del progetto nei tempi previsti).

L'approccio RAD potrebbe essere una proposta attraente per molti progetti, in particolare per i più piccoli che non riguardano l'area del core business dell'organizzazione e, di conseguenza, non sono necessari per altri progetti di sviluppo. Soluzioni rapide non sono probabilmente ottimali o sostenibili per le aree di base, legate al core business. I problemi associati al RAD comprendono:

1. progetto dell'interfaccia GUI inconsistente;
2. soluzioni specializzate piuttosto che soluzioni generali (queste ultime facilitano il riuso del software);
3. scarsa e incompleta documentazione;
4. software difficile da mantenere e da estendere, etc.

3.3 Negoziazione e validazione dei requisiti

I requisiti di clienti distinti potrebbero sovrapporsi o essere in conflitto. Alcuni requisiti potrebbero essere ambigui o non realistici. Altri requisiti potrebbero rimanere ignorati. Per tali ragioni, i requisiti devono essere negoziati e validati con i clienti prima di essere riportati nel documento dei requisiti.

In realtà, la *negoziazione e validazione dei requisiti* è svolta in parallelo alla raccolta. Appena i requisiti sono acquisiti, sono assoggettati a un'indagine applicata automaticamente nelle tecniche moderne di raccolta che implicano l'uso della *dinamica di gruppo*. Tuttavia, una volta che i requisiti siano stati collezionati, la possibile presenza di conflitti implica una *negoziazione* basata su una bozza del documento dei requisiti. Scopo del passo è migliorare la comprensione dei requisiti elencati nella bozza, modificandoli se necessario. Requisiti irrilevanti sono rimossi. Nuovi requisiti eventualmente scoperti sono aggiunti.

La *validazione dei requisiti* richiede una versione più completa del documento dei requisiti, con tutti i requisiti chiaramente identificati e classificati. Gli stakeholder leggono il documento e procedono a effettuare incontri di revisione formale. Le revisioni sono una forma di *testing* (Sezione 10.1.1).

3.3.1 Requisiti fuori dal contesto

La scelta dei progetti e, di conseguenza, dei sistemi da implementare è fatta durante le attività di *pianificazione di sistema* (Sezione 1.2). Tuttavia, i dettagli delle interdipendenze tra i sistemi possono essere scoperte solo durante la fase di *analisi dei requisiti*. È compito dell'analisi dei requisiti determinare il *contesto del sistema* in modo che ogni "uscita di contesto" possa essere valutata il più presto possibile.

Per poter decidere se un qualsiasi requisito sia o no all'interno del contesto in cui deve essere sviluppato il sistema è necessario un modello di riferimento. Nel passato, tale modello era rappresentato per mezzo di un *diagramma di contesto*, il diagramma di livello più alto nella tecnica di modellazione strutturata chiamata DFD. Sebbene il DFD sia stato sostituito in UML dal diagramma dei casi d'uso, il diagramma di contesto è ancora un metodo molto efficace per stabilire il contesto del sistema (Sezione 3.5.1).

Tuttavia, potrebbero esistere altre ragioni per cui un requisito può essere considerato come esterno al contesto (vedi Sommerville e Sawyer, 1997). Ad esempio, un requisito può essere troppo complesso da implementare in un sistema computerizzato e quindi lasciato a un processo umano. Oppure il requisito potrebbe avere una bassa priorità e, quindi, essere escluso dalla prima versione del sistema. Infine, un requisito potrebbe essere implementato nell'ambiente esterno e, quindi, uscire dal contesto del sistema in costruzione.

3.3.2 Matrice di dipendenza dei requisiti

Assumendo che tutti i requisiti siano chiaramente identificati e numerati (Sezione 3.4.1), è possibile costruire una *matrice di dipendenza* degli stessi (o *matrice d'interazione*) (vedi Sommerville e Sawyer, 1997; Kotonya e Sommerville, 1997). La matrice riporta in modo ordinato gli identificatori dei requisiti sia sulle righe sia sulle colonne, come mostrato in Figura 3.2.

Siccome la matrice è, evidentemente, simmetrica, solo la sua parte inferiore (sotto la diagonale principale) è usata. Le celle indicano se due requisiti qualsiasi si sovrappongono, sono in conflitto, o sono indipendenti (celle vuote). I *requisiti in conflitto* dovrebbero essere discussi con i clienti e riformulati ove possibile per risolvere o ridurre i conflitti (la traccia del conflitto dovrebbe essere conservata per poter essere riconsiderata nel successivo processo di sviluppo). Anche i *requisiti sovrapposti* dovrebbero essere riscritti per eliminare le sovrapposizioni.

La matrice di dipendenza dei requisiti è una tecnica semplice ma efficace per ricercare i conflitti e le sovrapposizioni quando il numero dei requisiti è relativamente piccolo. Quando questo non si verifica, la tecnica può essere ancora usata se i requisiti sono raggruppati in categorie (Sezione 3.4.1) e quindi confrontati separatamente all'interno di ciascuna categoria.

3.3.3 Priorità e rischi associati ai requisiti

Una volta che l'insieme di requisiti rivisti è stato prodotto (risolvendo conflitti e sovrapposizioni), è necessario sottoporlo a un'analisi del rischio e una valutazione della priorità. L'*analisi del rischio* identifica i requisiti che potrebbero causare difficoltà nello sviluppo. La *valutazione della priorità* è necessaria per permettere una semplice ritaratura del progetto rispetto ai ritardi (ad esempio, "scaricando" i requisiti con bassa priorità per rispettare tempi di progetto).

I requisiti possono essere "rischiosi" a causa di una varietà di fattori. Tipologie comuni di *rischio* sono (vedi Sommerville e Sawyer, 1997):

- *Rischio tecnico*, quando un requisito è difficile da implementare tecnicamente.
- *Rischio di prestazione*, quando un requisito una volta implementato può far diminuire in modo non accettabile il tempo di risposta del sistema.

Requisito	R1	R2	R3	R4
R1	X	X	X	X
R2	Conflitto	X	X	X
R3			X	X
R4		Sovrapposto	Sovrapposto	X

FIGURA 3.2 Matrice di dipendenza dei requisiti.

- *Rischio di sicurezza*, quando un requisito una volta implementato può esporre il sistema a pericoli di intrusione.
- *Rischio d'integrità del database*, quando un requisito può causare inconsistenza nei dati e non può essere verificato agevolmente.
- *Rischio nel processo di sviluppo*, quando un requisito richiede l'uso di strumenti di sviluppo non convenzionali e con cui gli sviluppatori non hanno familiarità (ad esempio, metodi di specifica formale).
- *Rischio politico*, quando un requisito potrebbe risultare difficile da soddisfare a causa di ragioni politiche interne.
- *Rischio legale*, quando un requisito potrebbe violare leggi attualmente vigenti.
- *Rischio di volatilità*, quando un requisito è probabilmente soggetto a modifiche frequenti durante il processo di sviluppo.

In linea di principio, le priorità dei requisiti sono espresse da singoli clienti durante il processo di raccolta dei requisiti. Tali priorità devono quindi essere negoziate negli incontri di gruppo tenendo conto dei fattori di rischio.

Per eliminare eventuali ambiguità e semplificare l'assegnazione delle priorità, il numero di classi possibili di priorità dovrebbe essere ridotto. Da tre a cinque classi di priorità è il caso più comune: "alta", "media", "bassa", "non nota". Un elenco alternativo potrebbe essere: "essenziale", "utile", "poco interessante", "da decidere".

3.4 Gestione dei requisiti

I requisiti devono essere gestiti. La *gestione dei requisiti* è in realtà parte della gestione globale di progetto e riguarda tre punti principali:

1. Identificare, classificare, organizzare e documentare i requisiti.
2. Gestire le modifiche dei requisiti (e come esse sono proposte, negoziate, validate e documentate).
3. Gestire la tracciabilità dei requisiti (ovvero come sono trattate le relazioni di dipendenza sia tra i requisiti e altri elaborati dello sviluppo sia tra i requisiti stessi) (si veda Capitolo 10).

3.4.1 Identificazione e classificazione dei requisiti

I requisiti sono descritti con *asserzioni in linguaggio naturale*, quali:

- "Il sistema, su richiesta dell'impiegato, dovrebbe preparare la chiamata al prossimo cliente".
- "Il sistema dovrebbe effettuare automaticamente il numero telefonico e simultaneamente visualizzare sullo schermo le informazioni del cliente comprendenti: numero telefonico, numero cliente, nome cliente".
- "Se la connessione è riuscita, il sistema dovrebbe visualizzare un testo d'introduzione che l'impiegato potrebbe usare per iniziare la conversazione con il cliente".

Un comune sistema potrebbe consistere di centinaia o migliaia di asserzioni sui requisiti simili alle precedenti. Per gestire propriamente un tale numero di requisiti, questi devono essere numerati seguendo un qualche schema d'identificazione. Lo schema potrebbe comprendere una *classificazione* dei requisiti in categorie più facilmente gestibili.

Esistono varie tecniche d'identificazione e classificazione dei requisiti (Kotonya e Sommerville, 1998):

- *Identificatori unici*, normalmente numeri assegnati in sequenza manualmente o generati dallo strumento CASE (cioè dal database o *repository* in cui lo strumento memorizza gli elaborati dell'analisi e del progetto).
- *Numeri generati in base alla struttura del documento dei requisiti* (ad esempio, il settimo requisito nella terza sezione del secondo capitolo potrebbe essere numerato 2.3.7).
- *Numeri sequenziali rispetto alla categoria del requisito*, assegnati in aggiunta a un nome mnemonico che identifica la categoria del requisito (dove le categorie possono essere: requisiti funzionali, requisiti informativi, requisiti di prestazione, requisiti di sicurezza, etc.).

Ogni metodo d'identificazione ha i propri pro e contro. Il più flessibile e meno incline all'errore è l'*identificatore unico generato dal database*. I sistemi di database hanno capacità proprie per la generazione di identificatori unici per ogni nuovo record (di dati), anche in un contesto multi-utente.

Alcuni database possono anche offrire un supporto alla gestione di versioni distinte dello stesso record (estendendo il valore dell'identificatore unico con il numero della versione). Infine, i database permettono di mantenere i legami d'*integrità referenziale* tra i modelli elaborati, requisiti compresi, e di conseguenza, offrono il supporto necessario per la gestione del cambiamento dei requisiti e della loro tracciabilità.

3.4.2 Gerarchie di requisiti

I requisiti possono essere strutturati gerarchicamente con relazioni simili a quelle di composizione (Sezione 2.1.4). Un requisito può essere composto da più sotto-requisiti e, a sua volta, può essere parte di un requisito a livello superiore.

Le relazioni gerarchiche introducono un livello ulteriore della classificazione dei requisiti, e questo può riflettersi nei numeri d'identificazione (usando la notazione a punto). Ad esempio, consideriamo il seguente insieme gerarchico di requisiti:

1. "Il sistema, su richiesta dell'impiegato, dovrebbe preparare la chiamata al prossimo cliente".
   - 1.1 "Il sistema dovrebbe attivare il bottone `Prossima_chiamata` quando l'impiegato entra nel modulo `Controllo Telemarketing` oppure quando la precedente chiamata è terminata".
   - 1.2 "Il sistema dovrebbe rimuovere la chiamata attivata dalla coda delle chiamate in attesa".
   - 1.3 etc.

Le gerarchie di requisiti permettono la definizione di requisiti che sono a differenti *livelli di astrazione*. Ciò è coerente con il principio generale di modellazione incrementale secondo cui passando al livello successivo d'astrazione si aggiungono sistematicamente dettagli ai modelli.

3.4.3 Gestione del cambiamento

I requisiti cambiano, possono essere rimossi, nuovi requisiti possono essere aggiunti, in una fase qualsiasi del ciclo di sviluppo. Il cambiamento è la norma, non l'eccezione, ma può diventare un grosso problema se non opportunamente gestito.

Più lo sviluppo è avanzato, più il cambiamento è costoso. Modificare un requisito appena creato, e non legato ad altri, è un semplice esercizio di editing. Modificare lo stesso requisito dopo che è stato implementato nel software potrebbe essere molto costoso, tanto da risultare proibitivo per la necessità di "tracciare" all'indietro (e modificare) tutte le parti di progetto sviluppate a partire dal requisito originario.

Una modifica potrebbe essere legata a un errore umano ma è spesso causata da variazioni di politica interna o da fattori esterni quali la competizione, i mercati globali, i progressi tecnologici, la legislazione, etc. Qualunque sia la ragione, rigide politiche di gestione sono necessarie per documentare la *richiesta* di cambiamento, per stimare il suo *impatto* nel progetto, e per effettuare le modifiche.

Poiché i cambiamenti ai requisiti sono costosi, una procedura formale deve essere sviluppata per gestire ogni richiesta di cambiamento. Una modifica accettata deve essere analizzata per valutarne la fattibilità tecnica, l'impatto sul resto del progetto, e il costo. Una volta approvata, la modifica è integrata nei modelli in cui è rilevante e quindi implementata nel software.

La gestione del cambiamento coinvolge la "tracciatura" di una mole rilevante di informazioni: sarebbe velleitario pensare di non usare uno strumento di supporto adatto allo scopo. In teoria, le modifiche ai requisiti dovrebbero essere memorizzate e tracciate da uno strumento di *gestione della configurazione* del software (*software configuration management tool*) usabile dagli sviluppatori per gestire le versioni di modelli e programmi durante tutto il ciclo di sviluppo. Un buono strumento CASE dovrebbe avere proprie capacità di gestione della configurazione oppure essere facilmente collegabile a uno strumento specifico.

3.4.4 Tracciabilità dei requisiti

La *tracciabilità dei requisiti* è solo un elemento, per quanto importante e critico, della *gestione del cambiamento*. Un modulo per gestire la tracciabilità dei requisiti deve conservare le relazioni di tracciabilità, per propagare le modifiche a/da un requisito lungo tutto il ciclo di sviluppo.

Si consideri il requisito: "Il sistema, su richiesta dell'impiegato, dovrebbe preparare la chiamata al prossimo cliente". Tale requisito potrebbe essere rappresentato con un diagramma di sequenza, attivato dall'interfaccia GUI attraverso un bottone d'azione etichettato "Prossima Chiamata", e programmato come un trigger del database. Nel caso in cui esista una relazione di tracciabilità tra tutti questi elementi, una modifica a uno qualunque di essi renderà tutta la "traccia" soggetta a modifica (per usare il linguaggio di uno degli strumenti esistenti sul mercato, la traccia diventa *sospetta*).

Una relazione di tracciabilità può legare modelli prodotti in fasi distinte dello sviluppo. Solo legami adiacenti di tracciabilità possono essere modificati direttamente. Per esempio, se un elemento *A* è legato a un elemento *B*, e *B* è legato a un elemento *C*, la modifica della relazione deve essere fatta in due passi (il Capitolo 10 tratta più approfonditamente la tracciabilità e la gestione del cambiamento).

3.5 Modello dei requisiti di business

La fase di *determinazione dei requisiti* cattura i requisiti descrivendoli (per la maggior parte) come asserzioni in linguaggio naturale. Una modellazione formale dei requisiti usando UML è condotta in seguito nella fase di *specifica dei requisiti*. In ogni modo, anche in questa fase è generalmente prodotta una rappresentazione grafica dei requisiti, a un alto livello di astrazione, chiamata *modello dei requisiti di business*.

Modelli grafici ad alto livello sono molto utili per determinare lo scopo del sistema, per identificare i casi d'uso principali e per stabilire le classi di business più importanti. La Figura 3.3 mostra le dipendenze tra questi tre modelli e i modelli nelle restanti fasi del ciclo di vita.

Il ruolo centrale dei casi d'uso nel ciclo di vita è illustrato nella Figura 3.3 in cui i casi di test, la documentazione utente, i piani di progetto sono tutti derivati dal modello dei casi d'uso. Inoltre, il modello dei casi d'uso e il modello delle classi sono il punto di partenza per ogni altro modello di progetto e di implementazione.

3.5.1 Modello dello scopo del sistema

La principale inquietudine che affligge gli analisti nello sviluppo di un sistema è, forse, lo *slittamento dello scopo* dovuto al cambiamento continuo dei requisiti. Mentre alcune modifiche ai requisiti sono inevitabili, è necessario assicurare che le modifiche richieste non vadano oltre lo scopo condiviso e accettato del progetto.

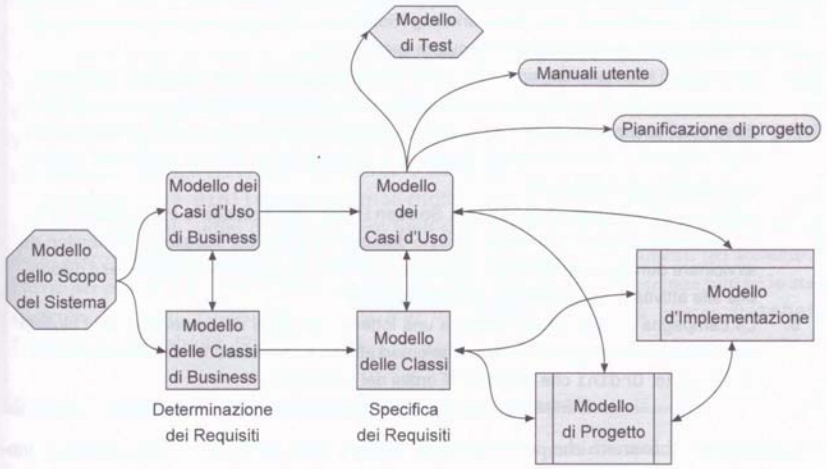


FIGURA 3.3 Modelli di business nel ciclo di vita.

La domanda è: "Come dobbiamo definire lo scopo del sistema?" Rispondere a tale questione non è semplice poiché qualsiasi sistema è solo una parte di un ambiente più esteso, una parte dell'insieme di sistemi che costituiscono l'ambiente stesso. I sistemi collaborano scambiandosi informazioni e invocando servizi. Quindi, la precedente domanda potrebbe essere interpretata come: "Dovremmo implementare i requisiti oppure la funzionalità richiesta è responsabilità di un altro sistema?".

Per rispondere alle domande sullo scopo, è necessario conoscere il *contesto* nel quale il sistema opera. È necessario conoscere le *entità esterne*, altri sistemi, organizzazioni, persone, macchine, etc. che attendono alcuni servizi dal sistema o forniscono servizi al sistema, servizi che si traducono in flussi di informazioni (*data flow*) da e per il sistema.

Lo scopo del sistema, di conseguenza, può essere determinato attraverso l'identificazione delle entità esterne e dei flussi di ingresso/uscita tra esse e il sistema stesso. Il sistema riceve delle informazioni in ingresso, esegue le azioni necessarie e produce delle informazioni in uscita.

UML non fornisce un buon modello grafico per definire lo scopo del sistema. Di conseguenza, lo storico *diagramma di contesto* (*context diagram*) dei DFD (Sezione 3.3.1) è spesso usato per portare a termine tale compito. La Figura 3.4 mostra il diagramma di contesto per l'applicazione di telemarketing.

3.5.1.1 Esempio di modellazione dello scopo di sistema

Esempio 3.1 (Telemarketing)

Si consideri la specifica del caso Telemarketing (Sezione 2.3.4) e si costruisca per esso un diagramma di contesto, considerando le seguenti osservazioni:

1. Il servizio di telemarketing opera per delle "campagne pubblicitarie per opere caritatevoli"; le campagne sono pianificate su indicazione degli amministratori della società che decidono quali cause sostenere. Le campagne devono essere approvate dal governo locale. Il loro progetto e la pianificazione relativa è supportato da un sistema applicativo separato, il `Database Campagna`.
2. Esiste un altro sistema, il `Database Sostenitori`, che gestisce le informazioni su tutti i sostenitori, passati e presenti. Questo database è usato per scegliere i sostenitori da avvicinare durante una specifica campagna. L'insieme di sostenitori scelto è reso disponibile alle attività di telemarketing per la campagna.
3. La campagna è organizzata intorno a una lotteria. Gli ordini dei sostenitori per i biglietti della lotteria sono registrati durante il telemarketing e quindi elaborati dal sistema di `Elaborazione Ordini` che gestisce gli ordini dei sostenitori.

Un diagramma di contesto che potrebbe essere costruito come soluzione a quest'esempio è mostrato in Figura 3.4. La "bolla" al centro del diagramma rappresenta il sistema. I rettangoli attorno alla bolla indicano le entità esterne. Le frecce rappresentano i flussi di dati. Il contenuto informativo dettagliato dei flussi non è visibile sul diagramma, esso è definito separatamente e memorizzato nel repository dello strumento CASE.

FIGURA 3.4 Modello dello scopo del sistema - diagramma di contesto (Telemarketing).

Il sistema `Telemarketing` ottiene l'informazione sulla campagna corrente dall'entità esterna `Database Campagna`. Tale informazione comprende il numero e i prezzi dei biglietti, i premi della lotteria, la durata della campagna, etc. Analogamente, i dettagli dei sostenitori sono ottenuti dal `Database Sostenitori`. Durante una chiamata, una nuova informazione su un sostenitore potrebbe emergere (ad esempio, il sostenitore ha cambiato indirizzo), quindi il `Database Sostenitori` deve essere aggiornato di conseguenza (il flusso di dati `Dettagli Sostenitore` è bidirezionale).

La principale attività avviene tra `Telemarketing` e `Sostenitore`. Il flusso `Conversazione` contiene l'informazione che è scambiata durante la conversazione telefonica. La risposta del sostenitore all'offerta dell'impiegato di comprare biglietti della lotteria è trasferita lungo il flusso `Risultato`. Un flusso distinto, chiamato `Collocamento Biglietti`, è usato per registrare i dettagli sui biglietti ordinati dal sostenitore.

Il flusso `Ordine Biglietti` è diretto all'entità esterna `Elaborazione Ordine`. Ulteriori elaborazioni degli ordini di biglietti sono fuori dallo scopo del sistema. Si può assumere che dopo aver registrato gli ordini, altre entità esterne possono gestire i pagamenti dei sostenitori, l'invio per posta dei biglietti, la consegna dei premi, etc. Questo non è d'interesse purché lo stato corrente di ordini, pagamenti, etc. sia reso disponibile dalle entità esterne `Database Campagna` e `Database Sostenitori`.

3.5.2 Modello dei casi d'uso di business

Un *modello dei casi d'uso di business* (Kruchten, 1999) è un modello molto astratto (Sezione 2.2.2) che identifica i processi di business di alto livello, i *casi d'uso di business*. Un caso d'uso in questo contesto corrisponde a ciò che è talvolta chiamata una *caratteristica di sistema* (le caratteristiche di sistema sono spesso specificate in un *documento visione*; se la "visione" dell'azienda è presente allora potrebbe essere usata come sostituto del modello dei casi d'uso di business).

L'aspetto centrale di un diagramma dei casi d'uso di business è l'*architettura* dei processi di business. Il diagramma fornisce una "vista a volo d'uccello" del comportamento desiderato del si-

stema. La descrizione testuale per ciascun caso d'uso di business è concisa, "orientata" al business, e centrata sul flusso principale di attività. Un modello dei casi d'uso di business non è adeguato per comunicare agli *sviluppatori* "cosa" il sistema dovrebbe fare.

I *casi d'uso di business* sono trasformati in *casi d'uso* nella fase di specifica dei requisiti. È in quest'ultima fase che i casi d'uso sono identificati e specificati in dettaglio, le descrizioni iniziali sono estese per comprendere sotto-processi e processi alternativi, possono essere costruiti modelli dimostrativi di alcune schermate GUI, e le relazioni tra i casi d'uso sono definite.

Gli *attori* in un diagramma dei casi d'uso di business differiscono dalle *entità esterne* nel diagramma di contesto per il modo con cui gli attori stessi interagiscono con il sistema. Gli attori sono operativi, attivano i casi d'uso inviando loro degli eventi. Le linee di comunicazione tra attori e casi d'uso non sono solo flussi di dati, rappresentano il flusso di eventi dagli attori e il flusso delle risposte dai casi d'uso.

Come osservato in precedenza (sezione 2.2.5.3) esiste un'interessante dicotomia riguardante gli attori. Gli attori possono essere sia *esterni* sia *interni* a un sistema. Sono *esterni* perché interagiscono con il sistema dall'esterno inviandogli degli eventi. Possono essere *interni* perché il sistema potrebbe aver bisogno di gestire delle informazioni sugli attori per interagire con essi. La specifica del sistema deve descrivere il sistema e anche prendere in considerazione il suo ambiente. Poiché l'ambiente contiene gli attori, il sistema potrebbe trovarsi nella necessità di avere due modelli concernenti gli attori, un modello degli attori e un modello di cosa il sistema conosce degli attori.

3.5.2.1 Esempio di modellazione dei casi d'uso di business

Esempio 3.2 (Telemarketing)

Si consideri la specifica del caso e il diagramma di contesto per il sistema Telemarketing (Sezioni 2.3.4 e 3.5.1.1) e si costruisca un diagramma dei casi d'uso di business.

Un possibile diagramma dei casi d'uso di business è presentato in Figura 3.5. Vi sono due attori: `Impiegato` e `Sostenitore`; l'`Impiegato` richiede al sistema una chiamata telefonica un `Sostenitore` e, supposto che la connessione sia stabilita, il `Sostenitore` è coinvolto nella chiamata come attore. Il caso d'uso di business `Programmare Connessione Telefonica` (che include in questo caso l'esecuzione della chiamata) diventa una funzionalità visibile per entrambi gli attori.

Durante una conversazione, un Impiegato potrebbe aver bisogno di accedere e modificare i dettagli della campagna e dei sostenitori: questa funzionalità è rappresentata dal caso d'uso di business `Gestione Dati` in cui sono usate le operazioni primitive di creazione, lettura, modifica e cancellazione di oggetti, operazioni indicate a volte con l'acronimo CRUD (Create, Read, Update e Delete).

Infine, il caso d'uso di business `Inserire Risultato Conversazione` serve per l'inserimento dei risultati dell'azione di telemarketing. Questo caso d'uso è visibile a entrambi gli attori.

L'omissione della relazione tra i casi d'uso `Gestione Dati` e `Inserire Risultato Conversazione` è arbitraria. In generale, molte relazioni tra casi d'uso di business possono essere soppresse per evitare confusione e affollamento del diagramma. Infatti, i casi d'uso tendono ad avere una qualche sorta di relazione con la maggior parte degli altri casi d'uso.

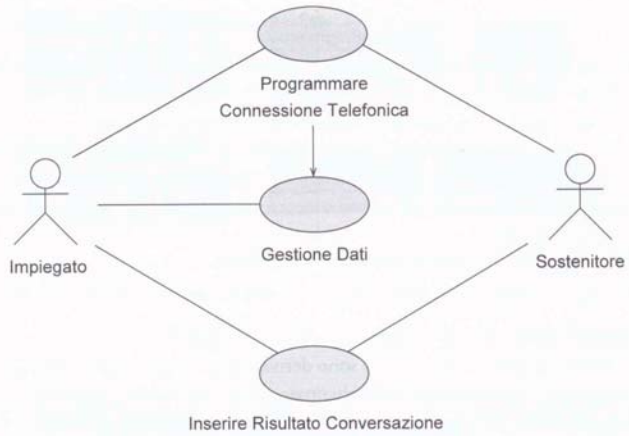


FIGURA 3.5 Modello dei casi d'uso di business (Telemarketing).

3.5.3 Modello delle classi di business

Un *modello delle classi di business* è un *modello delle classi*. Come per i casi d'uso, la differenza è nel livello d'astrazione. Un modello delle classi di business identifica i principali "oggetti di business" nel sistema. A questo livello, si è poco interessati agli attributi delle classi, il nome e una breve descrizione potrebbero essere sufficienti.

È interessante osservare che spesso gli *attori* di un modello dei caso d'uso di business sono rappresentati come *classi* nel modello delle classi di business. Ciò è coerente con l'osservazione precedente che gli attori sono spesso sia esterni sia interni al sistema (Sezione 3.5.2).

3.5.3.1 Esempio di modellazione delle classi di business

Esempio 3.3 (Telemarketing)

Si consideri la specifica del caso, il diagramma di contesto e il diagramma dei casi d'uso di business per il Telemarketing (Sezione 2.3.4, 3.5.1.1 e 3.5.2.1), e si costruisca un diagramma delle classi del business. I seguenti suggerimenti potrebbero aiutare:

1. L'enfasi nel sistema è sull'ordinamento delle chiamate nelle code di attesa. Tale ordinamento (programmazione) è un problema algoritmico, comunque le code delle chiamate e i risultati delle chiamate devono essere memorizzati in qualche struttura di dati.
2. Come precedentemente discusso, l'informazione sugli attori potrebbe essere memorizzata in oggetti del sistema.

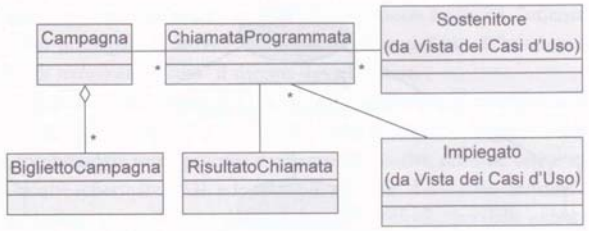


FIGURA 3.6 Modello delle classi di business (Telemarketing).

Un primo modello delle classi di business è mostrato in Figura 3.6. Il diagramma contiene sei classi; due (`Sostenitore` e `Impiegato`) sono derivate dagli attori nel modello dei casi d'uso di business. L'algoritmo d'ordinamento delle chiamate ottiene un numero telefonico e altre informazioni dalla classe `Sostenitore` e assegna la chiamata a un'istanza della classe `Impiegato` attualmente disponibile. L'algoritmo sarà probabilmente implementato nel database del sistema come una procedura memorizzata (Sezione 9.1.2.1).

La classe `ChiamataProgrammata` contiene la coda attuale delle chiamate, ciascuna delle quali fornisce dei risultati che sono registrati in `RisultatoChiamata` e propagati ad altre classi come `Campagna` o `Sostenitore`.

La classe `Campagna` aggrega oggetti di `BigliettoCampagna` e associa molte istanze di `ChiamataProgrammata`. Allo stesso modo, `Sostenitore` e `Impiegato` potrebbero avere molte istanze di `ChiamataProgrammata`. L'associazione tra `ChiamataProgrammata` e `RisultatoChiamata` è 1:1 (biunivoca).

3.6 Documento dei requisiti

Il *documento dei requisiti* è il risultato tangibile della fase di determinazione dei requisiti. La maggior parte delle organizzazioni produce un documento dei requisiti seguendo uno *schema* predefinito che definisce la struttura (indice) e lo stile del documento.

Il corpo principale del documento dei requisiti consiste di asserzioni sui requisiti stessi. Come discusso in Sezione 3.1, i requisiti possono essere raggruppati in *asserzioni di servizio* e *asserzioni di vincolo* (*vincoli*). Le asserzioni di servizio possono ulteriormente classificarsi in *requisiti informativi* e *requisiti funzionali*.

Il documento dei requisiti deve descrivere inoltre gli *aspetti progettuali*, trattati generalmente all'inizio del documento e poi ancora nelle conclusioni. Nella parte introduttiva del documento è trattato il contesto del progetto, comprendente lo scopo del progetto, la descrizione degli stakeholder, e i vincoli principali. Nella parte conclusiva sono discussi i restanti problemi, compresi i tempi, i costi, i rischi, etc.

3.6.1 Schemi di documento

Gli *schemi* per i documenti dei requisiti sono in genere recuperabili da libri di testo, organizzazioni di standardizzazione (IEEE, ANSI, etc.), pagine web di società di consulenza, fornitori di strumenti CASE, etc. A tempo debito, ogni organizzazione dovrebbe sviluppare uno standard idoneo alle proprie pratiche organizzative, alla propria cultura, ai lettori previsti, etc.

Uno schema definisce la struttura del documento, fornendo linee guida dettagliate su cosa deve essere scritto in ogni sezione del documento stesso. Le linee guida potrebbero riguardare il contenuto, le motivazioni, gli esempi, e ulteriori considerazioni (vedi Robertson e Robertson, 2000).

La Figura 3.7. mostra un tipico indice per il documento dei requisiti. Le spiegazioni sul contenuto saranno discusse nelle prossime sezioni.

Documento dei requisiti
Indice

1. Premesse del progetto
1.1 Obiettivi e scopo del prodotto
1.2 Contesto di business
1.3 Stakeholder
1.4 Soluzioni preliminari
1.5 Sintesi del documento
2. Servizi del sistema
2.1 Scopo del sistema
2.2 Requisiti funzionali
2.3 Requisiti informativi
3. Vincoli di sistema
3.1 Requisiti di interfaccia
3.2 Requisiti di prestazione
3.3 Requisiti di sicurezza
3.4 Requisiti operativi
3.5 Requisiti politici e legali
3.6 Altri vincoli
4. Aspetti progettuali
4.1 Problemi aperti
4.2 Programnma preliminare
4.3 Previsione costi
Appendici
Glossario
Documenti e moduli di business
Riferimenti

FIGURA 3.7 Documento dei requisiti - Indice.

3.6.2 Premesse del progetto

Le *premesse* inserite nel documento riguardano i manager e i decisori i quali, probabilmente, non studieranno in dettaglio l'intero documento. Gli *obiettivi* e lo *scopo* del progetto devono essere spiegati chiaramente, seguiti dalla descrizione del contesto di business.

Il documento dei requisiti deve costruire un *caso di business* per il sistema. In particolare, devono essere considerati tutti gli sforzi di *pianificazione di sistema* (Sezione 1.2). Il documento dei requisiti dovrebbe spiegare anche come il sistema proposto contribuirà agli obiettivi e alle finalità di business dell'organizzazione.

Gli *stakeholder* (Sezione 1.1.2) del sistema devono essere identificati. È importante che un cliente non sia un generico dipartimento o ufficio, per cui dovrebbero essere elencati i nomi delle persone.

Sebbene il documento dei requisiti dovrebbe essere sufficientemente lontano da soluzioni tecniche, è comunque importante discutere le *linee generali di soluzione*. Qualsiasi soluzione già disponibile è di speciale interesse: in genere, è più sensato comprare un prodotto piuttosto che svilupparlo da zero.

Il documento dei requisiti dovrebbe fornire una lista di package software e di componenti che dovrebbero essere ulteriormente analizzati come potenziali soluzioni. Si noti che scegliere un soluzione preesistente modifica i processi di sviluppo ma non dispensa dall'analisi dei requisiti e dal progetto di sistema.

Infine, è una buon'idea inserire una *sintesi* del resto del documento nelle premesse per facilitare la comprensione del contenuto. La sintesi dovrebbe anche spiegare la metodologia di analisi e di progetto adottata dagli sviluppatori.

3.6.3 Servizi del sistema

La parte principale del documento dei requisiti è dedicata alla definizione dei servizi del sistema (Sezioni 1.3.1 e 3.1). Tale parte comprenderà probabilmente più della metà dell'intero documento e sarà anche l'unica a contenere modelli ad alto livello, cioè i modelli dei requisiti di business (Sezione 3.5).

Lo *scopo del sistema* può essere modellato con un *diagramma di contesto* (Sezione 3.5.1) attraverso cui i "confini" del progetto proposto devono essere chiaramente definiti.

I *requisiti funzionali* possono essere rappresentati con un *diagramma dei casi d'uso di business* (Sezione 3.5.2); il diagramma dovrebbe anche fornire un inquadramento di alto livello per un dettagliato elenco dei requisiti funzionali (come discusso nella Sezione 3.4, ciascun requisito deve essere identificato, classificato e definito).

I *requisiti informativi* possono essere rappresentati con un *diagramma delle classi di business* (Sezione 3.5.3). Come per i requisiti funzionali, il diagramma delle classi di business non è una definizione completa delle strutture dati. Comunque, gli attributi della classe potrebbero essere brevemente descritti e gli identificatori dovrebbero essere determinati (altrimenti, non è possibile discutere correttamente le associazioni).

3.6.4 Vincoli del sistema

I *servizi del sistema* definiscono *cosa* il sistema deve fornire. I *vincoli* (Sezioni 1.3.1.e 3.1) descrivono le limitazioni cui il sistema deve sottostare quando esegue i propri servizi e possono riguardare:

- Requisiti d'interfaccia.
- Requisiti di prestazione.
- Requisiti di sicurezza.
- Requisiti operativi.
- Requisiti legali e politici, etc.

I *requisiti d'interfaccia* definiscono come il prodotto si rapporta con gli utenti. Nel documento dei requisiti si specifica solo come si presenterà l'interfaccia. Il progetto iniziale dell'interfaccia stessa sarà condotto durante la specifica dei requisiti e sviluppato più avanti durante il progetto di sistema.

I *requisiti di prestazione* possono assumere un ruolo centrale per il successo di un progetto. In senso stretto, specificano i *tempi di risposta* del sistema per i vari compiti che esso deve eseguire. In senso più allargato, i requisiti di prestazione comprendono altri vincoli, riguardanti l'affidabilità, la disponibilità, l'efficienza, etc.

I *requisiti di sicurezza* descrivono i privilegi d'accesso degli utenti all'informazione controllata dal sistema. Agli utenti possono essere imposte restrizioni per l'accesso ai dati e/o limitazioni per l'esecuzione di certe operazioni sui dati stessi.

I *requisiti operativi* specificano l'ambiente hardware/software nel quale il sistema opererà. Tali requisiti potrebbero determinare un impatto su altri aspetti del progetto, come l'addestramento degli utenti e la manutenzione del sistema.

I *requisiti politici* e *i requisiti legali* restano spesso impliciti piuttosto che essere stabiliti esplicitamente nel documento dei requisiti. Questo può rivelarsi un errore molto costoso. Tali requisiti devono essere messi in giusta evidenza perché potrebbero provocare il rifiuto del prodotto.

Altre categorie di *vincoli* sono possibili. Ad esempio, alcuni sistemi potrebbero imporre ulteriori richieste sulla semplicità d'uso del sistema (*requisiti d'usabilità*) o sulla semplicità di manutenzione del sistema (*requisiti di manutenibilità*).

Una definizione non ambigua dei vincoli di sistema riveste un ruolo cruciale nel progetto di un sistema informativo: sono numerosi gli esempi di progetti falliti a causa di vincoli omessi o scorrettamente interpretati. Il problema è delicato sia rispetto ai clienti sia rispetto agli sviluppatori. Sviluppatori senza scrupoli potrebbero giocare la carta della poca chiarezza nei vincoli di sistema nel tentativo di eludere le proprie responsabilità.

3.6.5 Aspetti progettuali

La parte finale del documento dei requisiti riguarda aspetti propriamente di progetto non trattati altrove. Una sezione importante di questa parte potrebbe chiamarsi *Problemi Aperti* e descrivere qualsiasi problema possa avere un impatto sul successo del progetto. Ad esempio, la sezione potrebbe evidenziare dei requisiti che possono non essere presi inizialmente in considerazione ma che potrebbero assumere in seguito un certo rilievo, oppure includere problemi potenziali o pratiche negative che il sistema potrebbe instaurare.

Un *programma preliminare* dei principali compiti del progetto dovrebbe essere sviluppato (Sezione 1.3.8) con un'allocazione iniziale di risorse (umane e non). Uno strumento di gestione di progetto può essere usato a questo punto per produrre diagrammi standard di pianificazione, come diagrammi PERT o carte di Gantt (Maciaszek, 1990). Un risultato importante di questo sforzo è la possibilità di fornire una stima preliminare dei costi di sviluppo.

3.6.6 Appendici

Nella struttura del documento, le appendici contengono in genere altre informazioni utili alla comprensione dei requisiti. Un'importante appendice potrebbe essere costituita dal *glossario* di progetto che definisce termini, acronimi e abbreviazioni usati nel documento dei requisiti. L'importanza di un buon glossario non può essere sottostimata poiché interpretazioni errate della terminologia possono causare molto danno.

Uno degli aspetti spesso trascurati nel documento dei requisiti è la rilevanza che dovrebbe assumere un'analisi dei *documenti* e dei *moduli* usati nell'organizzazione del lavoro (modulistica aziendale). Sarebbe auspicabile che esemplari (possibilmente compilati) della modulistica aziendale siano allegati al documento.

Il documento dei requisiti dovrebbe infine contenere una sezione di *riferimenti* alle fonti usate nella preparazione del documento stesso. Tali fonti dovrebbero comprendere, oltre a testi e pubblicazioni varie (manuali, documenti di standardizzazione, etc.), le minute degli incontri, i memorandum e altri documenti interni utilizzati durante la raccolta dei requisiti.

Sommario

In questo capitolo è stata discussa la fase di determinazione dei requisiti che precede la specifica dei requisiti stessi. Determinare i requisiti vuol dire raccoglierli e documentarli usando tecniche per lo più testuali (e quindi informali). La specifica dei requisiti, discussa nel prossimo capitolo, introdurrà modelli più formali.

La *raccolta dei requisiti* utilizza due linee di indagine, la conoscenza del dominio e i casi d'uso, che si complementano mutuamente e portano alla determinazione di un modello di business per il sistema da sviluppare.

Ci sono vari *metodi di raccolta dei requisiti*: interviste a clienti ed esperti del dominio, questionari, osservazioni, studio dei documenti e dei sistemi software, prototipi, JAD e RAD.

I requisiti forniti da clienti diversi potrebbero sovrapporsi o essere in conflitto. È compito dell'analista risolvere sovrapposizioni e conflitti attraverso la *negoziazione e validazione dei requisiti*. Per svolgere correttamente quest'attività, l'analista dovrebbe costruire una *matrice di dipendenza dei requisiti* e assegnare *rischi* e *priorità* ai requisiti stessi.

Grandi progetti devono *gestire* grandi volumi di requisiti. È essenziale in tali progetti che i requisiti siano *identificati* e *classificati*. *Gerarchie di requisiti* possono essere introdotte per accrescere la comprensibilità. La gestione deve assicurare un'efficiente *tracciabilità dei requisiti* durante l'intero ciclo di vita del progetto.

Sebbene la fase di determinazione dei requisiti non comprenda una modellazione formale del sistema, potrebbe essere costruito un semplice *modello dei requisiti di business*. Il modello di business dovrebbe comprendere tre diagrammi generici, il diagramma di contesto, il diagramma dei casi d'uso di business e il diagramma delle classi di business.

Il documento risultante dalla determinazione dei requisiti inizia con una sezione riservata al management aziendale, le *premesse del progetto*, che descrivono (ad alto livello) gli *obiettivi* e lo *scopo* del progetto, e il contesto di business. Le parti centrali del documento descrivono i *servizi del sistema* e i *vincoli*. La parte finale del documento tratta altri elementi progettuali, come il programma preliminare e una previsione dei costi.

Domande

D1 Durante la raccolta dei requisiti si cerca di riconciliare i requisiti sul dominio e i requisiti sui casi d'uso. Spiegare la differenza tra queste due tipologie di requisito. Una di queste tipologie dovrebbe essere prioritaria all'altra nel processo di determinazione dei requisiti? Spiegare.

D2 Discutendo le interviste si è raccomandato che opinioni degli intervistatori o risposte preconfezionate siano evitate. Credete che in alcune situazioni domande di questo tipo possano essere utili per il progetto?

D3 Cosa significa costruire un prototipo? In quale modo è utile alla raccolta dei requisiti?

D4 Cosa si intende per JAD e quali sono i suoi principali vantaggi rispetto agli altri metodi di raccolta dei requisiti?

D5 Cos'è lo slittamento dello scopo del sistema? Come può essere tenuto in considerazione durante la raccolta dei requisiti?

D6 Elencare e definire i rischi dei requisiti.

D7 Perché i requisiti dovrebbero essere numerati?

D8 Cos'è un traccia "sospetta"?

D9 In che senso gli attori in un diagramma di caso d'uso del business sono differenti dalle entità esterne nel diagramma di contesto?

D10 Descrivere la struttura di un tipico indice per il documento dei requisiti.

Esercizi

Specifica del caso: Valutazione delle Spese Pubblicitarie

Si consideri il caso di un'organizzazione per la Valutazione delle Spese Pubblicitarie (VSP):

1. L'organizzazione VSP raccoglie dati sulla pubblicità da varie fonti: stazioni televisive e radio, giornali, riviste, pubblicità murali, al cinema e in Internet.
2. Tali dati forniscono la base per diverse tipologie di servizio verso i clienti di VSP. Alcuni clienti potrebbero richiedere un rapporto sull'adeguatezza della pubblicità prodotta ai requisiti forniti all'agenzia pubblicitaria. Altri clienti potrebbero richiedere un rapporto sulle spese sostenute da un'agenzia pubblicitaria o per uno specifico prodotto pubblicizzato, raggruppate secondo vari criteri (tempi, regioni geografiche, fonti, etc.).
3. La stesura dei rapporti di spesa è il "core business" di VSP. Il portafoglio clienti di VSP comprende pubblicitari e inserzionisti individuali, agenzie di pubblicità, consulenti nel settore pubblicitario, responsabili di marketing, pianificatori di campagne pubblicitarie, compagnie radio, televisive, etc.
4. VSP ha contratti con molte fonti di erogazione di pubblicità per ricevere regolarmente da esse i file con informazioni sugli interventi pubblicitari erogati. L'informazione è trasferita sul database di VSP e soggetta a un'attenta verifica, in parte manuale e in parte automatica. Il compito della verifica è assicurare che tutti i dettagli sugli interventi siano validi e coerenti nel contesto dell'informazione erogata. L'inserimento manuale (e il monitoraggio) di interventi pubblicitari per cui non esistono informazioni computerizzate costituisce una componente rilevante dell'attività di VSP.
5. Una volta inserita e verificata l'informazione, gli interventi pubblicitari sono soggetti alla valorizzazione (processo d'assegnamento di una spesa stimata agli interventi stessi).

E1 Valutazione delle Spese Pubblicitarie – Si consideri la specifica del caso e si prepari un diagramma di contesto per il sistema VSP. Spiegare il modello prodotto.

E2 Valutazione delle Spese Pubblicitarie – Si consideri la specifica del caso e si prepari un diagramma dei casi d'uso di business per il sistema VSP. Spiegare il modello prodotto.

E3 Valutazione delle Spese Pubblicitarie – Si consideri la specifica del caso e si prepari un diagramma delle classi di business per il sistema VSP. Spiegare il modello prodotto.

Capitolo 4

Specifica dei requisiti

I sistemi complessi presentano molteplici aspetti: per ottenere la specifica completa di tali sistemi sono in genere necessari diversi modelli, modelli che possono essere di natura grafica o formale. A tal fine, UML offre un insieme di tecniche integrate di rappresentazione per assistere l'analista di sistema. L'uso di strumenti CASE è essenziale perché la modellazione abbia successo.

La specifica dei requisiti produce tre categorie di modelli: modelli dello stato, modelli del comportamento e modelli del cambiamento di stato. Questo capitolo spiega ed esemplifica tutte le principali tecniche che UML mette a disposizione per ogni categoria.

Sebbene nel capitolo siano dapprima presentati i modelli dello stato seguiti dai modelli del comportamento e del cambiamento di stato, questo non riflette la sequenza con cui la modellazione è condotta. Molti modelli sono sviluppati in parallelo e si "alimentano" mutuamente. Ciò è particolarmente vero per i due modelli principali, il modello delle classi e il modello dei casi d'uso.

4.1 Principi di specifica dei requisiti

La specifica dei requisiti s'interessa alla *modellazione* dei requisiti del cliente, definiti durante la *determinazione dei requisiti*. Durante questa fase sono presi in considerazione essenzialmente i servizi attesi dal sistema (Sezione 3.1) mentre i vincoli sono ulteriormente sviluppati solo per gli aspetti che possono derivare dalle iterazioni nel ciclo di sviluppo del progetto.

La specifica dei requisiti riceve in ingresso i requisiti del cliente e costruisce dei modelli di specifica (Sezione 2.2) che forniscono una definizione più formale per i vari aspetti (viste) del sistema. Le due categorie principali di requisiti del cliente, *requisiti funzionali* e *requisiti informativi*, sono entrambe considerate.

Uno dei risultati della fase di specifica è il *documento delle specifiche* ottenuto dal documento dei requisiti (Sezione 3.6) ampliato ed elaborato. La struttura del documento originario non è cambiata ma il contenuto è significativamente esteso nei capitoli che definiscono i requisiti del cliente. Il documento delle specifiche rimpiazza il documento dei requisiti ai fini del progetto e dell'implementazione (per quanto continui spesso a essere indicato come documento dei requisiti).

I modelli di specifica possono essere classificati in tre gruppi:

1. Modelli dello stato.
2. Modelli del comportamento.
3. Modelli del cambiamento di stato.

I modelli dello *stato* costituiscono un'elaborazione dei requisiti informativi. I modelli del *comportamento* forniscono specifiche dettagliate per i requisiti funzionali. I modelli del *cambiamento di stato* spaziano sulle due tipologie di requisito, spiegando come le funzioni causano le modifiche ai dati.

I modelli possono essere presentati mediante diagrammi UML e, in genere, un singolo diagramma riguarda una sola delle tre finalità di modellazione: lo stato, il comportamento, o il cambiamento di stato. Un'eccezione degna di nota è il diagramma delle classi che accanto agli aspetti di modellazione dello stato prende in considerazione anche altri aspetti legati al comportamento degli oggetti.

Ciascun diagramma enfatizza una prospettiva particolare (*vista*) del sistema e, insieme, permettono a sviluppatori e utenti di osservare la soluzione proposta da varie "angolazioni", rilevando alcuni aspetti e trascurandone altri. Nessun diagramma, preso singolarmente, fornisce una definizione completa del sistema. Il sistema può essere compreso solo considerando l'insieme dei diagrammi correlati.

La costruzione dei diagrammi non avviene in modo sequenziale, costruendo un diagramma dopo l'altro. Alcuni diagrammi sono sviluppati in parallelo e maggiori dettagli sono aggiunti in successive iterazioni; i dettagli possono riguardare il progetto (non solo l'analisi) dell'applicazione, per cui inizialmente sono espressi in forma testuale. A meno che un rigido processo di sviluppo sia imposto agli sviluppatori, la decisione di quali modelli costituiscano il nucleo iniziale intorno al quale sviluppare il processo dipende principalmente dalle preferenze personali degli analisti. I due modelli più importanti, i diagrammi dei casi d'uso e i diagrammi delle classi, dovrebbero essere costruiti in parallelo affinché, in ogni momento, ciascuno di essi possa usare l'informazione e le idee introdotte nell'altro.

4.2 Specifiche dello stato

Lo *stato* di un oggetto è determinato dal valore dei propri attributi e associazioni. Per esempio, uno stato di un oggetto `ContoBancario` potrebbe essere "a debito" quando il valore dell'attributo `saldo` è negativo. Poiché gli stati di un oggetto sono determinati dai dati in esso contenuti, i modelli delle strutture dati sono chiamati *specifiche dello stato*.

Le specifiche dello stato forniscono una *vista statica* del sistema (quindi, la modellazione dello stato è talvolta indicata come modellazione statica). Il compito principale della modellazione statica è definire le *classi* per il dominio applicativo in esame, i loro *attributi* e le *relazioni* con altre classi. Inizialmente, le *operazioni* delle classi non sono prese in considerazione e saranno derivate in seguito dai modelli di comportamento.

Generalmente s'identificano prima le classi *di sistema* (cioè le classi *entity*) le quali definiscono il dominio applicativo e che avranno una presenza persistente nel database del sistema. Tali classi sono talvolta chiamate "*oggetti di business*". Le classi collegate a eventi del sistema (classi *controllo* o *control*) e le classi che rappresentano gli oggetti nell'interfaccia del sistema (classi *presentazione* o *boundary*) non saranno identificate fino a quando le caratteristiche comportamentali del sistema non saranno analizzate.

4.2.1 Modellazione delle classi

Il modello delle classi è l'elemento fondamentale nello sviluppo object-oriented di un sistema, in base al quale il comportamento del sistema può essere analizzato. Sfortunatamente, il processo per identificare le classi "rilevanti" e le loro proprietà è sempre complesso e dipende dalla "sensibilità" dell'analista: due analisti distinti non producono mai due insiemi identici di classi (e proprietà) per il medesimo dominio applicativo. Sebbene i modelli delle classi possano essere, in una certa misura, differenti fra loro, il risultato e la soddisfazione utente potrebbero essere ugualmente positivi (o ugualmente negativi).

La modellazione delle classi non è un processo deterministico. Non esiste ricetta per come trovare le classi "buone". Per un buon progetto delle classi, che comunque dovrebbe svilupparsi in modo *iterativo* e *incrementale*, sono determinanti le seguenti caratteristiche dell'analista:

1. conoscenza delle tecniche di modellazione delle classi;
2. comprensione del dominio applicativo;
3. esperienza in progetti simili conclusi in modo soddisfacente;
4. capacità di "guardare avanti" per valutare le conseguenze delle decisioni assunte rispetto alle fasi successive del progetto;
5. disponibilità a rivedere il modello per eliminare imperfezioni, etc.

L'ultimo punto dipende dalla possibilità di utilizzare strumenti CASE. Tali strumenti incrementano certamente la produttività personale degli sviluppatori; tuttavia, l'applicazione su larga scala della tecnologia CASE potrebbe complicare (piuttosto che facilitare) lo sviluppo del sistema in organizzazioni immature (sezione 1.1.4.2).

4.2.1.1 Trovare le classi

Due analisti non producono mai due identici modelli delle classi per lo stesso dominio applicativo, così come due analisti non usano mai gli stessi processi mentali durante la ricerca delle classi. La letteratura è ricca di approcci raccomandati per la ricerca delle classi e gli analisti potrebbero anche inizialmente seguire uno di tali approcci, ma le successive iterazioni quasi certamente coinvolgono meccanismi non convenzionali e piuttosto arbitrari.

Bahrami (1999) commenta i punti principali dei quattro più comuni approcci per l'identificazione delle classi:

1. Approccio basato sulle frasi nominali.
2. Approccio basato sulle strutture comuni (*pattern*).
3. Approccio guidato dai casi d'uso.
4. Approccio CRC (*Class-Responsibility-Collaborators*, classi-responsabilità-collaborazioni).

Nei paragrafi che seguono, tali approcci saranno sinteticamente analizzati e inoltre si presenteranno alcuni esempi che usano un *approccio misto*.

4.2.1.1.1 Approccio basato sulle frasi nominali

L'approccio basato sulle frasi nominali consiglia all'analista di cercare nel documento dei requisiti le *frasi nominali*, cioè i sostantivi o le frasi in cui il nome ha una prevalenza sulla parte verbale (sono in genere frasi di tipo assertivo). Ogni sostantivo (ovvero frase nominale) è considerato una *classe candidata*. L'elenco delle classi candidate trovate è quindi suddiviso in tre gruppi:

1. classi rilevanti;
2. classi incerte;
3. classi irrilevanti.

Le *classi irrilevanti* sono quelle esterne al dominio del problema. I sostantivi corrispondenti appaiono nel discorso ma non è possibile formulare un'asserzione che descriva il loro scopo nel contesto del sistema in analisi. Analisti con esperienza probabilmente non includeranno le classi irrilevanti nella lista originale delle classi candidate. In tal modo, i passi d'identificazione ed eliminazione delle classi irrilevanti sono evitati.

Le *classi rilevanti* sono quelle che manifestamente appartengono al dominio del problema. Le frasi nominali rappresentanti i "nomi" di tali classi appaiono con frequenza nel documento dei requisiti. In più, è possibile confermare l'importanza e la finalità di tali classi dalla conoscenza generale del dominio applicativo e dall'analisi di sistemi simili, testi, documenti, e così via.

Le *classi incerte* sono quelle che non è possibile classificare con sicurezza e unanimemente come rilevanti. Tali classi costituiscono la sfida maggiore: è necessario analizzarle ulteriormente per includerle (o escluderle) nella lista delle classi rilevanti. È quest'analisi che permette di ottenere un modello delle classi migliore di un altro.

L'approccio basato sulle frasi nominali presume che il documento dei requisiti sia completo e corretto (nella realtà, ciò è vero solo raramente). Anche se questo fosse vero, la ricerca in grandi quantità di testo diventa ben presto pesante e non conduce necessariamente a un risultato completo e accurato.

4.2.1.1.2 Approccio basato sulle strutture comuni (*pattern*)

L'approccio basato su *pattern* deriva le classi candidate dalla teoria generale di classificazione degli oggetti. La teoria della classificazione è una parte della scienza che s'interessa al partizionamento del mondo di oggetti in gruppi significativi (pattern) che si ritrovano con una certa frequenza nel discorso e su cui è più agevole basare dei ragionamenti.

Bahrami (1999) elenca i seguenti pattern per la ricerca delle classi candidate:

- *Concetti*. Un concetto è una nozione condivisa, su cui esiste un accordo da parte di un'ampia comunità di persone. Senza concetti le persone non possono comunicare in modo soddisfacente. Per esempio, `Prenotazione` è una classe in un sistema di prenotazione aerea.
- *Eventi*. Un evento è un fatto che accade in un certo istante di tempo o ha una durata trascurabile rispetto a una scala temporale di riferimento. Per esempio, `Arrivo` è una classe in un sistema di prenotazione aerea.
- *Organizzazioni*. Un'organizzazione è una qualunque tipologia di raggruppamento o collezione di cose che ha una precisa finalità nel contesto in esame. Per esempio, `AgenziaViaggi` è una classe in un sistema di prenotazione aerea.

- *Persone*. "Persona" in questo caso è da intendersi non come una persona fisica ma piuttosto come un ruolo che una persona riveste nel sistema. Per esempio, `Passeggero` è una classe in un sistema di prenotazione aerea.
- *Posti*. I posti sono locazioni fisiche rilevanti per il sistema. `UfficioViaggi` è una classe in un sistema di prenotazione aerea.

Questi pattern suggeriscono le classi candidate per il progetto, la cui rilevanza deve comunque essere analizzata con attenzione.

Rumbaugh *et al.* (1999) propongono un differente schema di classificazione:

- Classe *fisica* (ad esempio, `Aereo`).
- Classe di *business* (ad esempio, `Prenotazione`).
- Classe *logica* (ad esempio, `OrarioVolo`).
- Classe *applicativa* (ad esempio, `TransazionePrenotazione`).
- Classe *computer* (ad esempio, `Indice`).
- Classe *comportamentale* (ad esempio, `CancellazionePrenotazione`).

L'approccio basato sui pattern fornisce un'utile guida ma non offre alcun processo sistematico attraverso il quale un insieme di classi affidabile e completo possa essere ricavato. Tale approccio potrebbe essere usato con successo per determinare l'insieme iniziale di classi o per verificare se alcune classi (derivate con altri mezzi) debbano essere presenti o meno.

Un particolare pericolo proprio di quest'approccio riguarda la possibilità di una scorretta interpretazione dei nomi delle classi. Ad esempio, cosa significa un `Arrivo`? Potrebbe significare: arrivo sulla pista d'atterraggio (tempo d'atterraggio), arrivo al terminal (tempo di sbarco), arrivo al ritiro bagagli (tempo di ritiro bagagli), etc. Qual è il significato corretto?

4.2.1.1.3 Approccio guidato dai casi d'uso

Nell'ambito di UML l'approccio *guidato dai casi d'uso* è evidenziato, se non raccomandato, da diverse metodologie (ad esempio, dal Rational Unified Process, RUP). Il modello grafico dei casi d'uso è completato con descrizioni testuali e con diagrammi di sequenza o collaborazione per i singoli casi d'uso (Sezione 2.2 e nel seguito di questo capitolo). Tali descrizioni e diagrammi addizionali definiscono i passi (e gli oggetti) necessari allo svolgimento di ciascun caso d'uso, e quindi costituiscono un buon punto di partenza per dedurre le classi candidate.

L'approccio guidato da casi d'uso è intrinsecamente bottom-up. Dopo aver determinato i casi d'uso e specificato, almeno parzialmente, la *vista d'interazione* del sistema mediante i diagrammi di sequenza, gli oggetti usati in tali diagrammi portano alla scoperta delle classi.

In realtà, l'approccio ha alcune similitudini con l'approccio basato sulle frasi nominali: infatti, a questo stadio nel ciclo di vita, la maggior parte dei casi d'uso è rappresentata testualmente, senza diagrammi d'interazione. La base comune sta nel fatto che i casi d'uso specificano i requisiti, e quindi entrambi gli approcci studiano le asserzioni contenute nel documento dei requisiti per trovare le classi candidate.

L'approccio guidato da casi d'uso soffre di lacune simili a quelle dell'approccio basato sulle frasi nominali. Essendo un approccio bottom-up, si affida per la sua accuratezza alla completezza e correttezza dei modelli dei casi d'uso, e quindi può condurre a uno sbilanciamento nello sviluppo verso i requisiti funzionali. Per questo motivo si dice che l'approccio è *guidato dalle funzioni* (i proponenti dell'object oriented preferiscono chiamarlo *guidato dal problema*).

4.2.1.1.4 Approccio CRC

L'approccio CRC è più che una tecnica per la ricerca delle classi, è un modo interessante per interpretare, comprendere e illustrare i requisiti. Rebecca Wirfs-Brock e i suoi colleghi (Wirfs-Brock e Wilkerson, 1989; Wirfs-Brock *et al.*, 1990) hanno reso popolare tale approccio.

L'approccio CRC prevede lo svolgimento di riunioni di gruppo, semplificate dall'uso di schede costituite da tre sezioni: il *nome classe*, scritto nella sezione più in alto, le *responsabilità* della classe, elencate nella sezione di sinistra, i *collaboratori*, elencati nella sezione di destra. Le responsabilità sono i servizi (operazioni) che la classe è in grado di fornire alle altre classi. Tali responsabilità possono richiedere la collaborazione (servizi) di altre classi, le quali sono elencate nella sezione "collaboratori".

L'approccio CRC è un processo dinamico durante il quale gli sviluppatori si scambiano le schede, compilandole con i nomi di classi e assegnando le responsabilità e le collaborazioni durante "l'esecuzione" di uno scenario d'elaborazione (ad esempio, un caso d'uso). Allorquando siano necessari servizi non coperti da classi esistenti, è creata una nuova classe cui sono assegnate le appropriate responsabilità e collaborazioni. Se una classe diventa "troppo appesantita", è suddivisa in un numero di classi più "piccole".

A differenza di altri, l'approccio CRC identifica le classi analizzando lo scambio di messaggi tra gli oggetti per realizzare le funzionalità del sistema. L'enfasi è posta su una distribuzione uniforme dell'"intelligenza" nel sistema e alcune classi potrebbero essere derivate da tale necessità "tecnica", piuttosto che emergere come "oggetti di business". In tal senso, l'approccio CRC potrebbe essere adatto alla verifica di classi già scoperte per mezzo di altri metodi e alla determinazione delle loro proprietà (implicate da responsabilità e collaborazioni).

4.2.1.1.5 Approccio misto

In pratica, il processo di scoperta delle classi segue probabilmente approcci differenti in diversi momenti. Spesso sono coinvolti elementi di tutti gli approcci spiegati in precedenza, insieme ad altri fattori come la conoscenza generale, l'esperienza e l'intuizione degli analisti. Tale processo di ricerca delle classi è chiamato *approccio misto* e non è né top-down né bottom-up, piuttosto può essere caratterizzato come middle-out (dal centro verso l'esterno).

Un possibile scenario è il seguente. L'insieme iniziale delle classi è derivato in base alla conoscenza generale e all'esperienza degli analisti. L'approccio basato su pattern fornisce una guida ulteriore. Altre classi potrebbero essere aggiunte dall'analisi di descrizioni d'alto livello del dominio applicativo usando l'approccio basato sulle frasi nominali. Se sono disponibili diagrammi dei casi d'uso, l'approccio guidato da casi d'uso può essere usato per ricercare nuove classi e verificare quelle esistenti. Infine, l'approccio CRC permette una riflessione conclusiva sulla lista delle classi finora trovate.

4.2.1.1.6 Linee guida per determinare le classi

La selezione delle classi candidate da parte di un analista potrebbe avvenire seguendo alcune *linee guida* o regole empiriche (si ricordi che, a tal punto, l'interesse è solo per le *classi entity*):

1. Ciascuna classe deve avere una chiara *finalità* nel sistema.
2. Ogni classe descrive uno "stampo" per un *insieme* di oggetti. Classi *singolari*, per le quali è possibile immaginare solo un oggetto singolo, non sono probabilmente tra gli "oggetti di business". Tali classi costituiscono in genere una "conoscenza comune" all'applicazione e sono direttamente codificate nei programmi applicativi. Ad esempio, se il sistema è progettato per una singola organizzazione, l'esistenza di una classe `Organizzazione` non ha molto senso.
3. Ogni classe deve essere specificata tramite un *insieme d'attributi*. È buona norma determinare fra questi l'attributo o gli attributi d'identificazione (*chiavi*) che permettono di individuare un oggetto fra gli altri oggetti della classe. Si ricordi, comunque, che una classe non necessita di una chiave definita dall'utente poiché gli *identificatori d'oggetto* (OIDs) già identificano gli oggetti delle classi (Sezione 2.1.1.3).
4. Classi e attributi sono elementi distinti di modellazione. Se un certo concetto debba essere modellato come una classe o come un attributo dipende dal dominio applicativo. Ad esempio, `Colore` di un automobile è normalmente percepito come un attributo della classe `Automobile`. Tuttavia, in una carrozzeria, `Colore` è una classe con i propri attributi (lucentezza, densità, trasparenza, etc.).
5. Ogni classe comprende un *insieme di operazioni*. Comunque, a tale stadio non c'è interesse per l'identificazione delle operazioni. Le operazioni (i servizi che la classe fornisce nel sistema) sono dichiarate nell'*interfaccia* della classe e implicate dalle finalità della classe (Punto1).

4.2.1.1.7 Esempi di ricerca delle classi

Esempio 4.1 (Immatricolazioni universitarie)

Considerare i seguenti requisiti per il sistema di Immatricolazioni universitarie e identificare le classi candidate:

1. Ogni curriculum universitario ha un numero di corsi obbligatori e un numero di corsi a scelta.
2. Ogni corso è a un dato livello e ha un valore in punti di credito.
3. Un corso può essere parte di un numero qualsiasi di curriculum.
4. Ogni curriculum specifica un numero minimo totale di punti di credito richiesto per raggiungerlo (ad esempio, il diploma in Sistemi Informativi richiede 68 punti di credito inclusi i corsi obbligatori).
5. Gli studenti possono combinare i corsi offerti in piani di studio adatti alle proprie necessità e che permettono di ottenere il curriculum nel quale lo studente è immatricolato.

Analizzando i requisiti al fine di trovare le classi candidate, si può osservare che, nella prima asserzione, le classi rilevanti sono `Curriculum` e `Corso`. Queste due classi sono conformi alle cinque linee guida elencate precedentemente. Non si può ancora essere certi se e come la classe `Corso` può essere specializzata nelle classi `CorsoObbligatorio` e `CorsoScelta`. È chiaro, ad esempio, che un corso può essere obbligatorio o a scelta rispetto a un certo curriculum. È possibile che la distinzione tra i corsi obbligatori e quelli a scelta sia rappresentata con un'associazione o anche con un attributo a livello di classe. Quindi, `CorsoObbligatorio` e `CorsoScelta` sono considerate classi probabili.

La seconda asserzione identifica solo attributi della classe `Corso`, `livello_corso` e `punti_credito`. La terza asserzione caratterizza un'associazione tra le classi `Corso` e `Curriculum`. La quarta asserzione introduce `totale_punti_credito` come un attributo della classe `Curriculum`.

L'ultima asserzione permette di considerare tre nuove classi: `Studente`, `CorsoOfferto` e `PianoStudio`. Le prime due sono senza dubbio classi rilevanti ma `PianoStudio` potrebbe ancora essere trasformato in un'associazione tra `Studente` e `CorsoOfferto`. Per tale ragione, `PianoStudio` è classificata come una classe probabile. Questa discussione è riassunta nella Tabella 4.1.

Esempio 4.2 (Video Store)

Considerare i seguenti requisiti per il sistema di Video Store e identificare le classi candidate:

1. Il Video Store ha in scorta un'estesa biblioteca di titoli di film attuali e popolari. Un film particolare potrebbe essere su video cassetta o video disco.
2. Le video cassette sono in formato "Beta" o "VHS"; i video dischi sono in formato "DVD".
3. Ogni film ha un certo periodo d'affitto (espresso in giorni), con una spesa d'affitto per quel periodo.
4. Il Video Store deve essere in grado di rispondere immediatamente a qualsiasi richiesta sulla disponibilità di un film nella scorta e quante cassette e/o dischi sono disponibili per l'affitto (la condizione attuale di ogni cassetta e disco deve essere nota e registrata).

Nella prima asserzione, Video Store non è una classe poiché l'intero sistema tratta di esso (ci sarebbe solo un oggetto nel database per quella classe, si tratta di una *classe singolare*). Allo stesso modo, le nozioni di scorta e biblioteca sono troppo generiche per essere considerate classi, almeno a questo stadio. Le classi rilevanti sembrano essere `TitoloFilm`, `CassettaVideo` e `VideoDisco`.

La seconda asserzione introduce un'ulteriore *specializzazione* su video cassette e dischi. Si possono proporre tre classi: `CassettaBeta`, `CassettaVHS` e `DiscoDVD`. Tuttavia, poiché c'è una sola tipologia di video disco, non c'è bisogno di mantenere sia `VideoDisco` sia `DiscoDVD` nel modello finale. Quale mantenere dipende dal livello finale di specializzazione applicato a cassette video e video dischi. Si noti che non vi è ancora alcuna certezza se `CassettaBeta` e `CassettaVHS` abbiano attributi che le differenziano (a parte il fatto che una è Beta e l'altra è VHS).

La terza asserzione afferma che ogni titolo di film ha condizioni d'affitto associate a esso. Tuttavia, non è chiaro cosa s'intenda per "film", titolo film o supporto film (cassetta o disco)? È necessario chiarire questo requisito con i clienti. Nel frattempo, è possibile dichiarare `CondizioniAffitto` come una classe probabile piuttosto che memorizzare informazioni su periodo d'affitto e costo d'affitto nella classe `TitoloFilm`.

TABELLA 4.1 Classi candidate (Immatricolazioni universitarie).

Classi rilevanti	Classi probabili
`Corso`	`CorsoObbligatorio`
`Curriculum`	`CorsoScelta`
`Studente`	`PianoStudio`
`CorsoOfferto`	

TABELLA 4.2 Classi candidate (Video Store).

Classi rilevanti	Classi probabili
`TitoloFilm`	`CondizioniAffitto`
`SupportoVideo`	
`CassettaVideo`	
`VideoDisco (o DiscoDVD)`	
`CassettaBeta`	
`CassettaVHS`	

L'ultima asserzione conferma che `CassettaBeta`, `CassettaVHS` e `DiscoDVD` (o `VideoDisco`) sono classi rilevanti: infatti, è necessario memorizzare informazioni sulla condizione corrente di ogni cassetta o disco. Attributi quale `condizione_video` possono essere genericamente dichiarati in una classe a più alto livello d'astrazione (diciamo `SupportoVideo`) ed ereditati dalle sottoclassi concrete (come `CassettaVHS`). Questa discussione è riassunta nella Tabella 4.2.

Esempio 4.3 (CMS)

Considerare i seguenti requisiti per il sistema CMS (Gestione Contatti con i Clienti) e identificare le classi candidate:

1. Il sistema supporta la funzione di "mantenere il contatto" con la base di clienti attuali e potenziali al fine di rispondere alle loro necessità e concludere nuovi contratti per i propri prodotti.
2. Il sistema memorizza i nomi, i numeri di telefono, gli indirizzi postali e di corriere, etc. delle organizzazioni e delle persone di contatto in tali organizzazioni.
3. Il sistema permette ai suoi impiegati di programmare i compiti e gli eventi che devono essere svolti rispetto alle persone di contatto. Gli impiegati programmano i compiti e gli eventi per altri impiegati o per se stessi.
4. Un compito è un insieme di eventi che si verificano per raggiungere un risultato come: convertire un cliente potenziale in un cliente attuale, organizzare la consegna dei prodotti, risolvere un problema di un cliente, etc. Tipi comuni d'eventi sono: chiamata telefonica, visita, invio di un fax, etc.

TABELLA 4.3 Classi candidate (CMS)

Classi rilevanti	Classi probabili
`Organizzazione`	`OrgAttuale`
`Contatto`	`OrgPotenziale`
`Impiegato`	`IndirizzoPostale`
`Compito`	`IndirizzoCorriere`
`Evento`	

La prima asserzione contiene i concetti di cliente, contratto e prodotto. La conoscenza generica e l'esperienza possono suggerire queste come classi entity. Tuttavia, contratto e prodotto non sono concetti appartenenti allo scopo del sistema CMS e quindi non dovrebbero essere considerati.

`Cliente` è una classe rilevante ma si potrebbe preferire il nome `Contatto` sapendo che non tutti i contatti sono clienti attuali. La distinzione tra cliente attuale e potenziale potrebbe giustificare l'introduzione delle classi `ClienteAttuale` e `ClientePotenziale`, ma poiché non vi è certezza, le due classi devono considerarsi classi probabili.

La seconda asserzione fa nuova luce sulla precedente discussione. È necessario distinguere tra un contatto con (una persona responsabile di) un'organizzazione e un contatto con persone. `Cliente` non sembra essere un buon nome per la classe. Dopo tutto, `Cliente` implica solo un cliente attuale e il nome può incorporare contatti con persone e organizzazioni. La nuova proposta è introdurre le classi: `Organizzazione`, `Contatto` (intendendo un contatto con persona), `OrgAttuale` (cioè un'organizzazione che è attualmente cliente) e `OrgPotenziale` (cioè un'organizzazione che è un cliente potenziale).

Nella seconda asserzione, solo alcuni attributi delle classi sono menzionati. Comunque, indirizzi postali e di corriere possono considerarsi attributi composti (parte di entrambe le classi: `Organizzazione` e `Contatto`). In alternativa, `IndirizzoPostale` e `IndirizzoCorriere` possono anche considerarsi come classi probabili.

La terza asserzione introduce tre classi rilevanti: `Impiegato`, `Compito` ed `Evento`. L'asserzione spiega l'attività di programmazione.

L'ultima asserzione rende più chiaro il significato delle precedenti classi e delle loro relazioni, ma non determina l'introduzione di nuove classi.

4.2.1.2 Specificare le classi

Una volta costruita la lista delle classi candidate, queste dovrebbero essere ulteriormente specificate includendole in un diagramma delle classi e definendone le proprietà. Alcune proprietà possono essere inserite all'interno delle icone grafiche rappresentanti le classi nei relativi diagrammi. Molte altre proprietà nella specifica delle classi hanno solo una rappresentazione testuale. Gli strumenti CASE forniscono semplici capacità di editing per inserire o modificare tali informazioni.

Come spiegato all'inizio di questo capitolo, le classi sono specificate a un certo livello d'astrazione. Capacità modellistiche più avanzate di UML non saranno qui usate ma discusse nel Capitolo 5 e seguenti.

4.2.1.2.1 Nominare le classi

A ogni classe deve essere dato un *nome*. In alcuni strumenti CASE è possibile assegnare un *codice* per la classe, possibilmente differente dal nome. Il codice dovrebbe conformarsi alle convenzioni sui nomi richieste dal linguaggio di programmazione o dal DBMS individuati per l'implementazione, ed è usato per la generazione del codice software dai modelli di progetto.

In precedenza, si è adottata una particolare convenzione per i nomi delle classi, secondo cui un nome di classe *inizia con una lettera maiuscola*. Per i nomi composti, la prima lettera di *ciascuna parola è maiuscola* (piuttosto che separata da un trattino). Si tratta solo di una convenzione raccomandata ma ha un certo seguito tra gli sviluppatori (nella Sezione 6.1.3.2, si raccomanderà di aggiungere una singola lettera di prefisso a ciascuna classe per identificare lo strato software al quale la classe appartiene).

Il nome di una classe dovrebbe essere *un nome singolare* (ad esempio `Corso`) o un nome singolare e un aggettivo (ad esempio `CorsoObbligatorio`) ogni volta ciò sia possibile. Poiché la classe è uno "stampo" per i suoi oggetti, usare frasi nominali plurali non aggiunge generalmente nuova informazione. Tuttavia può accadere che frasi nominali al plurale rappresentino meglio il significato di una classe (come `CondizioniAffitto` nell'Esempio 4.2).

Il nome di una classe dovrebbe essere *significativo*. Dovrebbe rappresentare la vera natura della classe ed essere derivato dal *vocabolario degli utenti* (non dal gergo degli sviluppatori!). È meglio usare un nome "lungo" piuttosto che creare un nome "criptico". Tuttavia, nomi troppo lunghi (ad esempio, più di *trenta caratteri*) sono pesanti, e alcuni ambienti di programmazione potrebbero non accettarli: tale osservazione è importante se lo strumento CASE lavora con i nomi invece che con i codici delle classi. Nomi lunghi potrebbero essere accettati dallo strumento in "aggiunta" al nome e al codice della classe.

4.2.1.2.2 Trovare e specificare gli attributi delle classi

L'icona grafica rappresentante una classe consiste di tre compartimenti (aree) (nome classe, attributi, operazioni) (Sezione 2.1.2). La specifica degli attributi delle classi è parte delle *specifiche dello stato* e quindi è discussa in questo paragrafo. La specifica delle operazioni delle classi è discussa più avanti, nelle *specifiche del comportamento* (Sezione 4.3).

La ricerca degli attributi avviene in parallelo alla ricerca delle classi. L'identificazione di attributi è un "effetto collaterale" della determinazione delle classi. Ciò non significa che la scoperta di attributi sia un'attività semplice. Al contrario, è un processo complesso ed altamente iterativo.

Nei modelli iniziali di specifica, si definiscono gli attributi essenziali alla comprensione degli *stati* nei quali possono trovarsi gli oggetti della classe. Altri attributi potrebbero temporaneamente essere ignorati (ma l'analista deve assicurarsi che tale informazione non vada perduta). È improbabile che tutti gli attributi delle classi siano menzionati nel documento dei requisiti, ma è importante nelle fasi iniziali includere solo attributi che siano implicati dai requisiti (altri attributi possono comunque essere aggiunti nelle susseguenti iterazioni).

La convenzione raccomandata per i nomi di attributo è d'usare le *lettere minuscole*. Le parole in un nome composto sono separate da un trattino di sottolineatura "_".

4.2.1.2.3 Esempi di specifica delle classi

Esempio 4.4 (Immatricolazioni universitarie)

Con riferimento all'Esempio 4.1, si considerino i seguenti ulteriori requisiti provenienti dal documento dei requisiti:

1. La scelta di corsi da parte di uno studente potrebbe essere limitata da sovrapposizioni d'orario e dal numero massimo di studenti che possono essere immatricolati nell'attuale offerta di corsi.
2. Il piano di studio proposto da uno studente è inserito nel sistema online d'immatricolazione. Il sistema controlla la consistenza del piano e riporta gli eventuali problemi. I problemi devono essere risolti con l'aiuto di un consigliere accademico. Il piano di studio finale è soggetto all'approvazione del delegato del Capo divisione e quindi inviato al segretario.

La prima asserzione fa menzione delle sovrapposizioni d'orario ma non è chiaro come rappresentare quest'elemento. È possibile che si stia parlando di un caso d'uso che proceduralmente determina le sovrapposizioni d'orario. La seconda parte della stessa asserzione può essere rappresentata aggiungendo l'attributo `quota_immatricolazioni` alla classe `CorsoOfferto`. `CorsoOfferto` dovrebbe avere inoltre gli attributi `anno` e `semestre` cui l'orario si riferisce.

La seconda asserzione "rafforza" la classe `PianoStudio`. Si può osservare che `PianoStudio` combina un certo numero di corsi offerti inclusi nell'offerta attuale. Di conseguenza, `PianoStudio` dovrebbe anch'essa avere gli attributi `anno` e `semestre`.

Un'analisi più approfondita delle classi probabili `CorsoObbligatorio` e `CorsoScelta` conduce all'osservazione che un corso è obbligatorio o a scelta *in relazione* a un curriculum accademico. Lo stesso corso potrebbe essere obbligatorio in relazione a un curriculum, a scelta in un altro curriculum e non ammesso per altri curriculum: se ciò è vero, `CorsoObbligatorio` e `CorsoScelta` non possono essere delle classi (si noti che non si sta trattando la modellazione con la generalizzazione (Sezione 2.1.5), argomento discusso in Sezione 4.2.4).

La Figura 4.1 rappresenta un modello delle classi conforme a ciò che è stato fin qui discusso. In più, i simboli (*stereotipi*) <<PK>> e <<CK>> davanti agli attributi indicano, rispettivamente, la *chiave primaria* e le *chiavi candidate* (Sezione 8.4.1.2). Le chiavi sono identificatori unici degli oggetti nelle classi interessate: fra queste n'è stata scelta una (la più rilevante) come primaria mentre le altre restano candidate. Anche i tipi di dato per gli attributi sono stati specificati.

Le classi `PianoStudio` e `CorsoOfferto` non hanno attributi identificanti. Questi potranno essere aggiunti successivamente, ad esempio all'atto dell'introduzione delle associazioni tra classi (Sezioni 2.1.3 e 4.2.2).

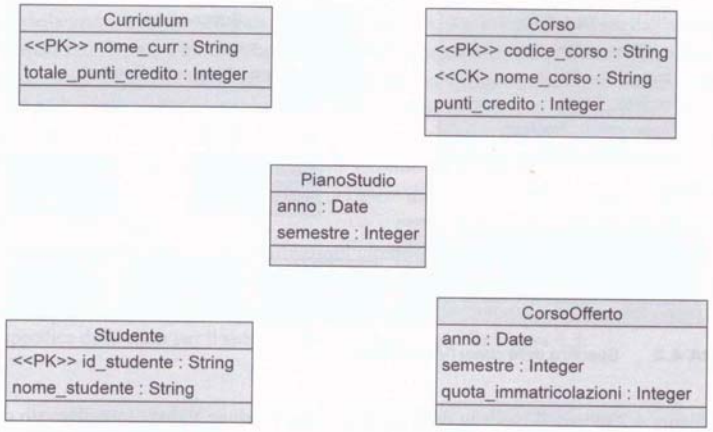


FIGURA 4.1 Specifica delle classi (Immatricolazioni universitarie).

Esempio 4.5 (Video Store)

Con riferimento all'Esempio 4.2, si assuma che il requisito 3 sulle condizioni d'affitto sia stato chiarito. I requisiti aggiuntivi sono i seguenti:

1. La spesa d'affitto è differente per supporti video differenti: cassetta o disco (ma è il medesimo per le due categorie di cassetta, Beta e VHS).
2. Sebbene il disco DVD sia il solo formato per dischi video attualmente presente, gli utenti richiedono che in futuro il sistema sia facilmente estensibile con altri formati per disco.
3. Gli impiegati del Video Store tendono a usare i codici dei film più popolari invece del titolo per identificare il film stesso. Questa è un utile pratica poiché lo stesso film potrebbe aver più di una versione, ognuna con differenti registi.

La prima asserzione spiega che le condizioni d'affitto sono differenti per `CassettaVideo` e `VideoDisco` pur contenenti lo stesso film. Ha senso, di conseguenza, avere `CondizioniAffitto` come classe distinta (da associare alle classi `CassettaVideo` e `VideoDisco`).

La seconda asserzione implica la necessità della coesistenza delle due classi `VideoDisco` e `DiscoDVD` (un `DiscoDVD` è un `VideoDisco` ma in futuro potrebbero essercene altri). Una gerarchia di generalizzazione con radice `SupportoVideo` è ora piuttosto ovvia ma si rimanda alla Sezione 4.2.4 per la discussione sulle relazioni di generalizzazione.

La terza asserzione aggiunge `codice_film` (come attributo chiave) e `regista` alla classe `TitoloFilm`. Altri attributi sono già stati discussi nell'Esempio 4.2.

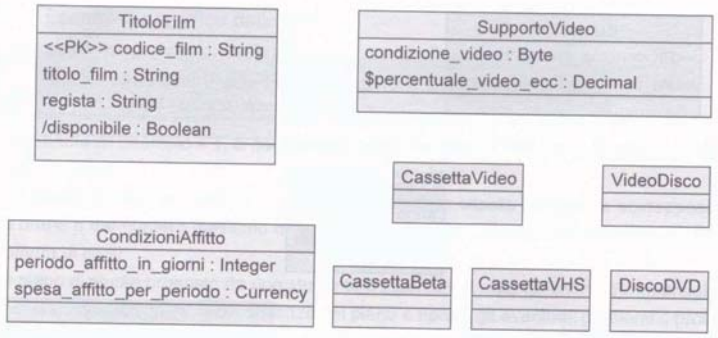


FIGURA 4.2 Specifica delle classi (Video Store).

La Figura 4.2 mostra il modello delle classi per l'applicazione Video Store, derivato dalle discussioni degli Esempi 4.2 e 4.5.

L'attributo booleano `disponibile` (si veda Esempio 4.2) della classe `TitoloFilm` è un attributo *derivato*. L'attributo `$percentuale_video_ecc` della classe `SupportoVideo` è un attributo di classe *(statico)* (Sezione 2.1.6) che conterrà la percentuale di oggetti per cui vale la condizione: `condizione_video="eccellente"` (benché non mostrata nel diagramma, un'operazione statica, chiamata ad esempio `$calcola_percentuale` dovrebbe essere associata all'attributo per calcolarne a richiesta il corrente valore).

Esempio 4.6 (CMS)

Con riferimento all'Esempio 4.3 si consideri la seguente informazione aggiuntiva:

1. Un cliente è considerato attuale se esiste un contratto con quel cliente per la consegna di prodotti o l'erogazione di servizi (la gestione del contratto è, comunque, esterna allo scopo del sistema).
2. Il sistema permette la produzione di vari rapporti sui contatti, basati su indirizzi postali e di corriere (ad esempio, trovare tutti i clienti per codice postale).
3. Data e ora della creazione di un compito è registrata. Il "valore monetario" atteso dal completamento del compito deve essere anch'esso memorizzato.
4. Gli eventi per l'impiegato sono visualizzati sullo schermo in pagine tipo calendario (un giorno per pagina). La priorità di ciascun evento (bassa, media e alta) è visivamente distinguibile sullo schermo.
5. Non a tutti gli eventi è associato un "tempo dovuto", alcuni sono "senza tempo" (a essi si può adempiere in qualsiasi momento durante la giornata per la quale sono stati programmati).
6. Il tempo di creazione di un evento non può essere modificato ma il tempo dovuto può subire modifiche.
7. Al termine di un evento, la data e l'ora di completamento sono registrati.
8. Il sistema memorizza anche gli identificatori degli impiegati che hanno creato i compiti e gli eventi, chi è programmato per dar seguito all'evento ("impiegato dovuto") e chi ha completato l'evento.

L'analisi della prima asserzione indica che la nozione di cliente attuale è derivata dall'associazione tra `Organizzazione` e `Contratto`. L'associazione potrebbe essere piuttosto dinamica. Quindi, le classi `OrgAttuale` e `OrgPotenziale` non sono molto significative. Inoltre, il sistema non è coinvolto nella gestione contrattuale e non è responsabile per la manutenzione della classe `Contratto`, per cui la soluzione più ragionevole è introdurre un attributo derivato `Organizzazione.è_attuale` modificabile secondo necessità.

La seconda asserzione fornisce la ragione per avere due classi indirizzo: `IndirizzoPostale` e `IndirizzoCorriere`.

Le asserzioni rimanenti danno informazioni ulteriori sugli attributi delle classi. Ci sono inoltre diversi suggerimenti sulle associazioni fra le classi e sui vincoli d'integrità; ad esempio, nell'asserzione 8 gli impiegati sono associati ai compiti e agli eventi (saranno discussi più avanti).

La specifica delle classi per il sistema CMS è presentato nella Figura 4.3.

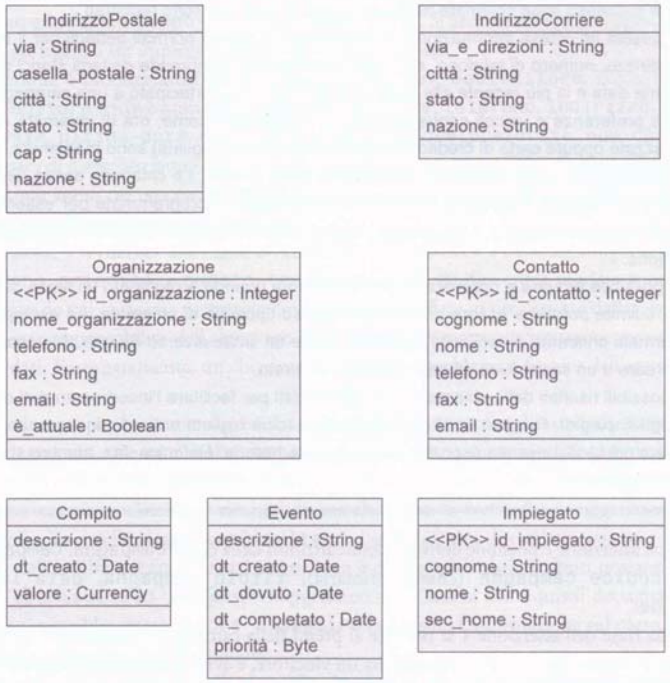


FIGURA 4.3 Specifica delle classi (CMS).

Esempio 4.7 (Telemarketing)

Con riferimento alla Sezione 2.3.4 e alla Sezione 3.5 che hanno presentato il modello dei requisiti di business per l'applicazione di Telemarketing (in particolare, il diagramma delle classi di business in Figura 3.6) si prenda in considerazione la seguente ulteriore informazione:

1. Una campagna ha un titolo, generalmente usato per riferirsi a essa, e un codice unico usato come riferimento interno. Ogni campagna ha una durata fissa. Al termine della campagna, i premi sono estratti e sono avvertiti i possessori di biglietti vincenti.
2. Tutti i biglietti sono numerati. I numeri sono unici per tutti i biglietti di una campagna. Il numero totale dei biglietti di una campagna, il numero di biglietti venduti dall'inizio e lo stato attuale di ogni biglietto sono noti (per esempio, disponibile, ordinato, vincitore premio, pagato).
3. Per determinare la prestazione degli impiegati della società, le durate delle chiamate e il loro successo (cioè chiamate risultanti in biglietti ordinati) sono registrati.
4. È gestita un'estesa informazione sui sostenitori. A parte i normali dettagli per il contatto (indirizzo, numero di telefono, etc.) tale informazione comprende dettagli storici come la prima data e la più recente alle quali un sostenitore ha partecipato a una campagna. Anche preferenze e vincoli conosciuti per un sostenitore (come, ore di chiamata non apprezzate oppure carta di credito usuale per l'acquisto di biglietti) sono mantenute.
5. L'elaborazione delle chiamate deve seguire certe priorità. Le chiamate senza risposta o dove si trova una segreteria telefonica devono essere riprogrammate per essere rieseguite successivamente. È importante variare le ore di chiamata durante i tentativi di ripetizione.
6. È possibile provare e riprovare le chiamate fino a quando è raggiunto un limite di tentativi. Il limite potrebbe essere diverso per diverse tipologie di chiamata. Ad esempio, una normale chiamata di "sollecito" potrebbe avere un limite diverso da una chiamata per ricordare a un sostenitore un pagamento in arretrato.
7. I possibili risultati delle chiamate sono classificati per facilitare l'inserimento dati da parte degli impiegati. Risultati tipici sono: successo (cioè biglietti ordinati), insuccesso, richiamare più tardi, nessuna risposta, impegnato, segreteria telefonica, fax, numero sbagliato, disconnesso.

Dalla prima asserzione è possibile derivare alcuni attributi della classe `Campagna`. `Campagna` contiene il `codice_campagna` (chiave primaria), `titolo_campagna`, `data_inizio` e `data_fine`.

L'ultima frase dell'asserzione 1 si riferisce ai premi della campagna e suggerisce l'introduzione della classe `Premio` (un premio è estratto, ha un vincitore, e avrà presumibilmente altre proprietà non esplicitamente indicate, come descrizione, valore, posizione rispetto ad altri premi della campagna). Tale classe è quindi aggiunta al modello delle classi con gli attributi `descr_premio`, `valore_premio` e `posizione_premio`. Si osservi che la data alla quale i premi sono estratti è la medesima per tutti i premi di una campagna. L'attributo `data_estrazione` è quindi aggiunto alla classe `Campagna`. Un vincitore di un premio è un sostenitore. Questo fatto si rappresenterà più avanti con un'associazione tra `Premio` e `Sostenitore`.

L'asserzione 2 indica che ogni biglietto ha un numero ma il numero non è unico su tutti i possibili biglietti (è unico solo all'interno di una determinata campagna). L'attributo `numero_biglietto` è introdotto ma non definito come chiave all'interno di `BigliettoCampagna`. Altri due attributi in `BigliettoCampagna` sono `valore_biglietto` e `stato_biglietto`. Il numero totale di biglietti e il numero di biglietti venduti (`num_biglietti` e `num_biglietti_venduti`) sono attributi di `Campagna`.

L'asserzione 3 mette in risalto alcuni problemi. Quali strutture dato sono necessarie per calcolare la prestazione degli impiegati? Per iniziare, è necessario disporre di misure per mezzo delle quali la prestazione possa essere espressa. Una possibilità è calcolare il numero medio di chiamate per ora e il numero medio di chiamate con successo per ora. Si può allora calcolare un indicatore di prestazione dividendo le chiamate di successo per il numero di chiamate. Alla classe `Impiegato` si aggiungono quindi i due attributi `media_ora` e `successi_ora`.

Per calcolare gli indicatori di prestazione degli impiegati è necessario memorizzare la durata di ogni chiamata. Il posto per tale informazione è la classe `RisultatoChiamata`. A tale classe sono aggiunti gli attributi `tempo_inizio` e `tempo_fine`. Si assume inoltre che ogni risultato di chiamata sia legato a un impiegato attraverso un'associazione.

L'asserzione 4 produce attributi da includere nella classe `Sostenitore`. Gli attributi sono `id_sostenitore` (chiave primaria), `nome_sostenitore`, `telefono`, `indirizzo_postale`, `prima_data`, `ultima_data`, `contatore_campagne`, `ore_preferite`, `num_carta_credito`. Alcuni di questi attributi (`indirizzo_postale`, `ore_preferite`) sono sufficientemente complessi, più avanti nel processo di sviluppo potrebbero essere convertiti in ulteriori classi (per ora, comunque, si rappresentano come attributi).

L'asserzione 5 si riferisce alla classe `ChiamataProgrammata` a cui si aggiungono gli attributi `telefono`, `priorità`, e `numero_tentativi`. Difficile dire come rendere concreto il requisito secondo cui chiamate successive dovrebbero essere fatte in ore distinte della giornata. Ciò è ovviamente responsabilità dell'algoritmo di ordinamento ma sono comunque necessarie delle strutture dati. Fortunatamente, un chiarimento su questo problema è fornito dall'asserzione successiva.

L'asserzione 6 produce una nuova classe `TipoChiamata` con gli attributi `descr_tipo`, `limite_tentativi` e `ore_alternative`. L'ultimo è una struttura dati complessa, simile a `ore_preferite` in `Sostenitore` (potrebbe diventare in seguito una classe distinta).

L'ultima asserzione classifica i risultati di una chiamata. Si tratta di un suggerimento diretto per l'introduzione della classe `TipoRisultato`. I possibili tipi di risultato possono essere memorizzati nell'attributo `tipo_risultato`. Non è chiaro quali altri attributi possano essere inclusi in `TipoRisultato` ma questi emergeranno con l'analisi dei requisiti dettagliati. Uno di tali attributi potrebbe essere `azione_a_seguire` per gestire l'informazione sul passo successivo da eseguire per ciascun tipo di risultato.

La Figura 4.4 è un modello delle classi che riassume la precedente discussione. Le associazioni già presenti nel modello delle classi di business (Figura 3.6) sono state mantenute ma non ne sono state aggiunte altre.

4.2.2 Modellazione delle associazioni

Le associazioni collegano oggetti del sistema, facilitando la collaborazione tra essi. Senza associazioni gli oggetti possono solo essere messi in relazione a tempo d'esecuzione, se condividono attributi o sono a conoscenza (attraverso altri mezzi, come variabili globali) dell'identità di altri oggetti.

Le associazioni sono la più importante tipologia di relazioni nel modello, in particolare, nel modello degli "oggetti di business". Le associazioni offrono un supporto all'esecuzione di casi d'uso e, di conseguenza, legano insieme le specifiche dello stato e del comportamento.

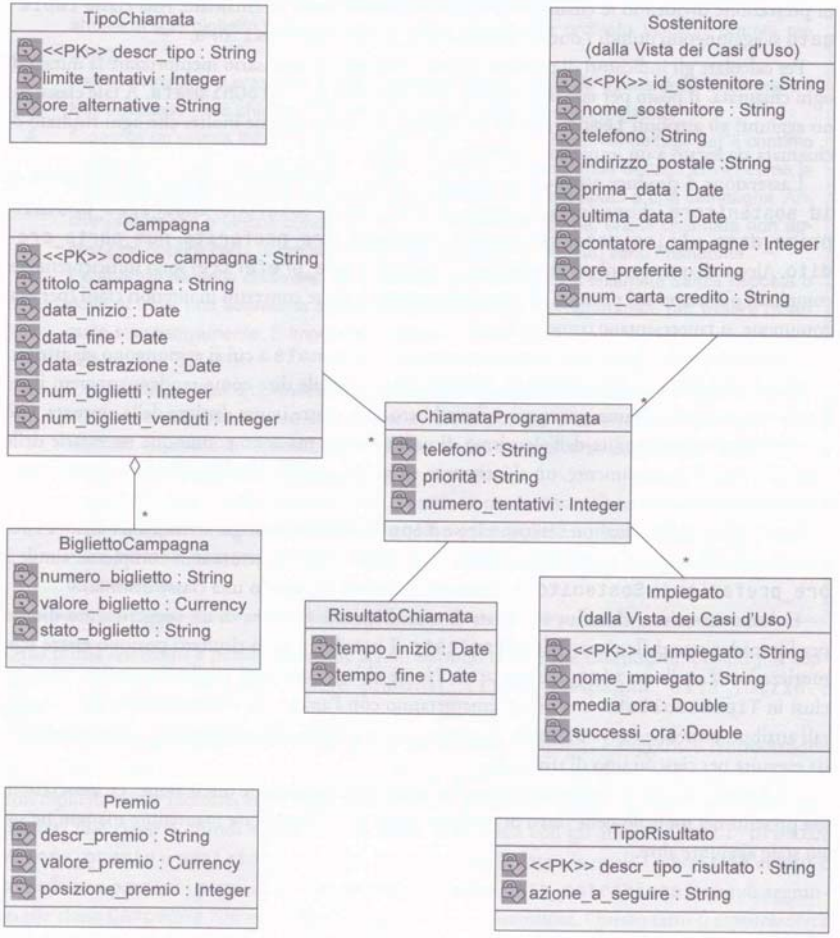


FIGURA 4.4 Specifiche delle classi (Telemarketing).

4.2.2.1 Determinare le associazioni

La ricerca delle associazioni principali è un "effetto collaterale" nella ricerca delle classi. Durante la definizione delle classi, l'analista decide in merito agli attributi delle classi e potrebbe osservare che alcuni di questi sono piuttosto associazioni verso altre classi. Gli attributi, infatti, possono aver un tipo dato primitivo oppure possono essere tipati con altre classi, stabilendo quindi delle relazioni. Essenzialmente, qualsiasi attributo con un *tipo dato non primitivo* dovrebbe essere rappresentato come un'associazione (o aggregazione) verso la classe rappresentante quel tipo di dato.

Immaginando di "eseguire" i casi d'uso è possibile trovare le restanti associazioni. Infatti, durante quest'esecuzione virtuale bisogna stabilire dei "canali di collaborazione" tra le classi che, generalmente, devono essere realizzati mediante associazioni.

Ogni *associazione ternaria* dovrebbe essere sostituita da un ciclo di associazioni binarie. Le associazioni ternarie introducono il rischio di interpretazioni semantiche non corrette.

Talvolta, un *ciclo di associazioni* non deve commutare per esprimere pienamente la sottostante semantica (Maciaszek, 1990). Un ciclo commuta se, partendo da un qualsiasi oggetto di una classe e seguendo le due vie possibili di collaborazione per arrivare a oggetti di un'altra classe del ciclo, si ottiene sempre lo stesso risultato. In tal caso almeno una delle associazioni nel ciclo può essere *derivata* dalle altre e dovrebbe essere eliminata (un buon modello semantico dovrebbe essere non ridondante). È possibile, o meglio probabile, che molte associazioni derivate, siano comunque incluse nel modello di progetto (ad esempio, per ragioni d'efficienza).

4.2.2.2 Specificare le associazioni

Specificare un'associazione implica:

1. assegnarle un nome;
2. assegnare i nomi dei ruoli;
3. determinarne la molteplicità (Sezione 2.1.3.2).

Le convenzioni di nome dovrebbero seguire quelle per i nomi degli attributi, lettere minuscole separate da "_" (Sezione 4.2.1.2.2).

Se due classi sono coinvolte in una sola associazione, la specifica del nome dell'associazione e dei nomi dei ruoli (Sezione 2.1.2.1.1) è opzionale (uno strumento CASE distingue internamente ogni associazione attraverso propri identificatori).

I *nomi dei ruoli* possono essere usati per chiarire meglio associazioni complesse, in particolare le *associazioni ricorsive* (che mettono in relazione oggetti delle stessa classe). Se indicati, i nomi dei ruoli in un'associazione dovrebbero essere scelti ricordando che, nei modelli di progetto, essi diventeranno attributi nella classe posta al capo opposto della linea che rappresenta l'associazione.

Le *molteplicità* dovrebbero essere specificate per entrambi i terminali (ruoli) dell'associazione. Se non evidenti, esse possono essere omesse a questo stadio.

4.2.2.3 Esempio di specifica di associazioni

Esempio 4.8 (CMS)

Riferendosi agli Esempi 4.3 e 4.6, determinare e specificare le associazioni tra classi del sistema CMS a partire dai requisiti descritti in tali esempi.

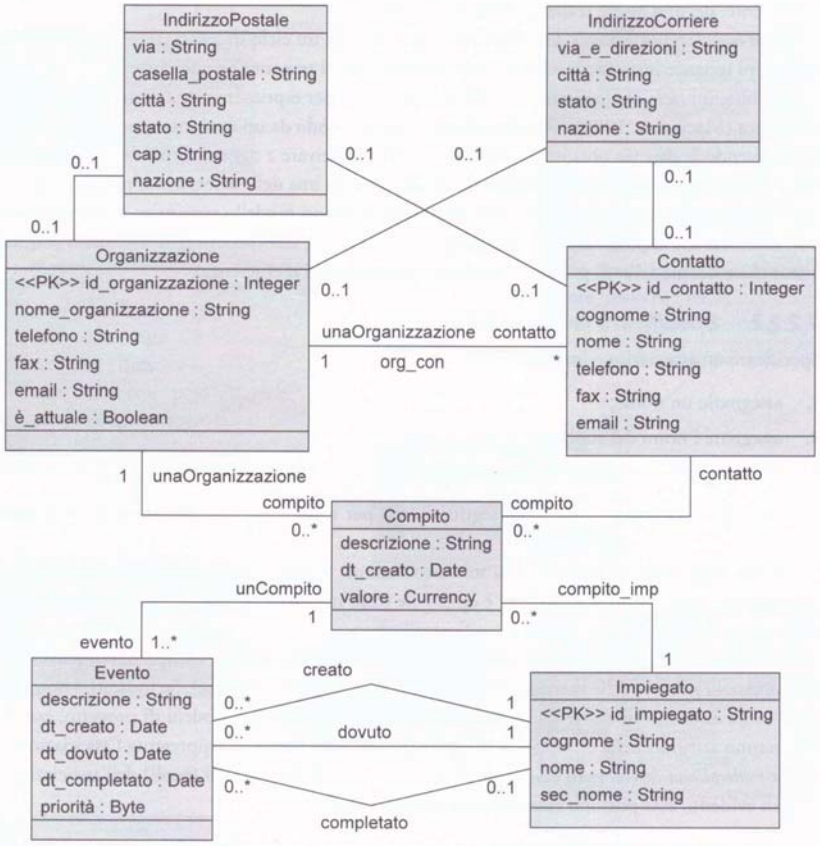


FIGURA 4.5 Specifica delle associazioni (CMS).

Il modello contenente le associazioni per il sistema CMS è presentato in Figura 4.5. Al fine di dimostrare la flessibilità possibile nella rappresentazione delle associazioni, l'uso dei nomi d'associazione e dei nomi di ruolo non è sistematico.

La molteplicità di tutte le associazioni tra `IndirizzoPostale` e `IndirizzoCorriere` da un lato e `Organizzazione` e `Contatto` dall'altro è "zero o uno". Ad esempio, un oggetto di `Organizzazione` è connesso al più a un oggetto di `IndirizzoPostale` ma solo se l'indirizzo postale dell'organizzazione è noto. Viceversa, un particolare oggetto di `IndirizzoPostale` è messo in relazione con un oggetto di `Organizzazione` *oppure* con un oggetto di `Contatto`. Le molteplicità devono quindi essere "zero o uno" ma questo non basta a esprimere la mutua esclusione (*oppure*) per cui il vincolo dovrebbe essere ulteriormente specificato usando il supporto alla rappresentazione di vincoli di UML (Sezione 5.1.2).

L'associazione tra `Organizzazione` e `Contatto` mostra l'uso sia dei nomi d'associazione sia dei nomi di ruolo. Questi ultimi saranno convertiti in nomi d'attributo nel modello d'implementazione, che conterrà i seguenti attributi: `Organizzazione.contatto` e `Contatto.unaOrganizzazione`. Il prefisso "una" indica esattamente la molteplicità "uno" del ruolo `unaOrganizzazione`. La molteplicità del ruolo `contatto` è "molti" (i limiti inferiore e superiore non sono determinati).

La molteplicità del ruolo `contatto`, tra `Compito` e `Contatto`, non è specificata. I requisiti non spiegano se un compito debba essere direttamente legato a un contatto. Se ciò accade, non è ancora chiaro se il compito possa essere legato a più di un contatto.

Infine, ci sono tre associazioni tra `Evento` e `Impiegato`. Tali associazioni determinano quale impiegato ha creato l'evento, chi lo eseguirà e chi, infine, lo completerà. Alla creazione dell'evento, l'impiegato che dovrà completarlo non è noto (per cui la molteplicità dell'associazione `completato`, dalla parte dell'impiegato, è "zero o uno").

4.2.3 Modellazione delle relazioni di aggregazione e composizione

L'*aggregazione* e la sua forma più "forte", la *composizione*, implicano la semantica "tutto-parte" tra la classe composta (superset) e una classe componente (subset) (Sezione 2.1.4). In UML, l'aggregazione è trattata come una forma vincolata di associazione. Questo può essere considerato come una grossolana sottostima del significato modellistico dell'aggregazione. È sufficiente osservare che l'aggregazione, con la generalizzazione, è la tecnica più importante per il riuso di funzionalità nei sistemi object-oriented.

La potenza modellistica di UML sarebbe grandemente agevolata se il linguaggio offrisse un supporto alle quattro possibili semantiche per l'aggregazione (Maciaszek *et al.*, 1996b):

1. aggregazione "possiede in modo esclusivo",
2. aggregazione "possiede",
3. aggregazione "ha",
4. aggregazione "raggruppa".

L'aggregazione *possiede in modo esclusivo* stabilisce che:

- c'è una *dipendenza esistenziale* tra le classi componenti e la classe composta (quindi, la cancellazione di un oggetto composto si propaga agli oggetti componenti nella relazione, obbligandone la cancellazione);
- l'aggregazione è *transitiva* (se l'oggetto C1 è parte di B1 e B1 è parte di A1, allora C1 è parte di A1);
- l'aggregazione è *asimmetrica* (se B1 è parte di A1, allora A1 non è parte di B1);
- l'aggregazione è *fissa* (se B1 è parte di A1, allora non può essere parte di un qualsiasi altro oggetto Ai ($i \neq 1$));

L'aggregazione *possiede* ha le tre prime proprietà dell'aggregazione *possiede in modo esclusivo* cioè:

- dipendenza esistenziale;
- transitività;
- asimmetricità.

L'aggregazione *ha* è semanticamente più debole dell'aggregazione *possiede*, avendo solo le due proprietà:

- transitività;
- asimmetricità.

L'aggregazione *raggruppa* specifica semplicemente un raggruppamento di oggetti indipendenti per un qualche scopo, raggruppamento che non fa assunzioni rispetto alle proprietà sopra indicate. Può essere considerata come un'astrazione in cui una collezione di oggetti è vista come un oggetto più astratto un *gruppo*. Un oggetto può essere parte di più gruppi (quindi, la *molteplicità* di un'aggregazione *raggruppa* può essere *molti-a-molti*).

Sebbene l'*aggregazione* sia stata riconosciuta come concetto modellistico fondamentale almeno quanto la *generalizzazione* (Smith e Smith, 1977), essa ha avuto un'attenzione marginale nell'analisi e nella progettazione orientata agli oggetti (a eccezione di domini applicativi come i sistemi multimediali). Fortunatamente, tale tendenza potrebbe in futuro essere invertita a causa dei risultati provenienti dal lavoro dei metodologi sulle *strutture comuni di progetto* (*design patterns*). Ciò è evidente, per esempio, nel trattamento dell'aggregazione (composizione) presentato nel fondamentale libro di Gamma *et al.* (1995).

4.2.3.1 Determinare aggregazioni e composizioni

La ricerca di aggregazioni e associazioni è generalmente fatta in parallelo. Se un'associazione possiede una o più delle quattro proprietà discusse in precedenza, allora può essere rappresentata come un'aggregazione.

Un test significativo per la ricerca di aggregazioni è usare le frasi "*ha*" e "*è parte di*" nella descrizione della relazione. In una descrizione "top-down", la frase è "ha" (ad esempio, `Libro` "ha" `Capitolo`). In una descrizione "bottom-up", la frase è "è parte di" (ad esempio, `Capitolo` "è parte di" `Libro`). Se la relazione interpretata con queste frasi non ha senso, allora *non* è un'aggregazione.

Dal punto di vista strutturale, l'aggregazione spesso mette in relazione parecchie classi mentre un'associazione è poco sensata oltre il grado binario (Sezione 2.1.3.1). Quando è necessario mettere in relazione più di due classi, un'aggregazione *raggruppa* potrebbe essere una buona scelta modellistica.

4.2.3.2 Specificare aggregazioni e composizioni

UML fornisce solo un limitato supporto per l'aggregazione. Una forma forte di aggregazione in UML è chiamata *composizione*. Nella composizione, l'oggetto composto potrebbe contenere "fisicamente" gli oggetti che sono sue parti (semantica "per valore"). Un oggetto componente può, in tal caso, provenire da (appartenere a) uno e un solo oggetto composto. La *composizione* UML corrisponde (più o meno) alle aggregazioni *possiede in modo esclusivo* e *possiede*.

La forma debole di aggregazione in UML è chiamata semplicemente *aggregazione*. Ha una semantica "per referenza", cioè l'oggetto composto non contiene "fisicamente" gli oggetti che sono sue parti ma è semplicemente legato a essi. Lo stesso oggetto componente può appartenere a più oggetti composti. L'*aggregazione* UML corrisponde, all'incirca, alle aggregazioni *ha* e *raggruppa*.

In UML, il *rombo pieno* rappresenta una composizione e il *rombo vuoto* rappresenta un'aggregazione. I restanti elementi di specifica dell'aggregazione sono analoghi a quelli usati per l'associazione.

4.2.3.3 Esempio di specifica di aggregazioni e composizioni

Esempio 4.9 (Immatricolazioni universitarie)

Riferendosi agli Esempi 4.1 e 4.5, considerare i seguenti requisiti aggiuntivi:

1. I dati accademici di uno studente (disponibili a richiesta) dovrebbero comprendere i voti ottenuti dallo studente in ciascun corso da questi frequentato (e da cui non si è ritirato nel limite delle prime tre settimane dall'inizio del semestre).
2. Ogni corso ha un docente incaricato ma anche altri docenti potrebbero insegnare nel corso. Ci potrebbero essere, per un certo corso, docenti incaricati differenti (e anche docenti "ausiliari" differenti) in differenti semestri.

La Figura 4.6 mostra un modello delle classi che enfatizza le relazioni d'aggregazione. Uno `Studente` "ha" come componenti diversi oggetti `InformazioneAccademica` per cui è una classe di composizione (semantica "per valore" in UML). Ogni oggetto di `InformazioneAccademica` include l'attributo `codice_corso`. Questo è necessario poiché l'associazione `sceglie` che sarà implementata in `Studente` con l'attributo `sceglie_crsoff` (di tipo *collezione*, cioè `Set[CorsoOfferto]`) è indipendente dagli oggetti di `InformazioneAccademica`.

`Corso` "ha" `CorsoOfferto` è un'aggregazione UML (semantica "per referenza"). Ogni oggetto di `CorsoOfferto` è solo logicamente contenuto in un oggetto `Corso`. `CorsoOfferto` può anche partecipare in altre aggregazioni e/o associazioni (ad esempio, con `Studente` e `DocenteIncaricato`).

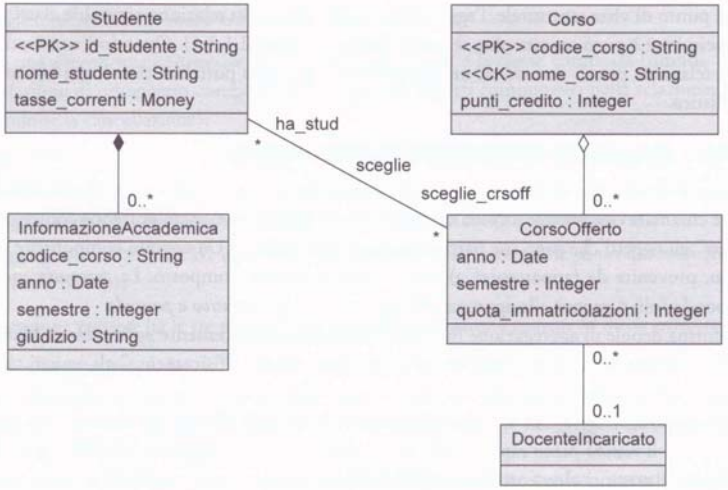


FIGURA 4.6 Specifica delle aggregazioni (Immatricolazioni universitarie).

4.2.4 Modellazione delle relazioni di generalizzazione

Le *caratteristiche* (attributi e operazioni) comuni di una o più classi possono essere astratte in una classe più generica. Ciò è noto come *generalizzazione*. La relazione di generalizzazione è una connessione tra una classe generica (*superclasse*) e classi più specifiche (*sottoclassi*). La generalizzazione permette l'*ereditarietà* (*riuso*) delle caratteristiche nella superclasse da parte delle sottoclassi. Generalmente, in un sistema object-oriented l'ereditarietà si applica alle classi, non agli oggetti (sono ereditati i tipi, non i valori).

Oltre all'*ereditarietà*, la generalizzazione ha due obiettivi (Rumbaugh *et al.*, 1999):

1. Sostituibilità.
2. Polimorfismo.

Secondo il principio di *sostituibilità* l'oggetto di una sottoclasse è un valore ammesso per una variabile sulla superclasse (ma non viceversa). Ad esempio, se una variabile è dichiarata per oggetti di classe `Frutto`, allora un oggetto di classe `Mela` è un valore ammesso per tale variabile ("una mela è un frutto" ma "un frutto può non essere una mela").

Secondo il principio del *polimorfismo* la stessa operazione può avere implementazioni differenti in classi distinte. Un oggetto chiamante può invocare un'operazione senza conoscere o far attenzione a quale implementazione sarà usata per l'esecuzione. L'oggetto chiamato conosce da quale classe proviene ed esegue l'implementazione appropriata.

Il polimorfismo è sfruttato al meglio quando è usato insieme all'ereditarietà. Spesso un'operazione polimorfa è solo dichiarata nella superclasse ma senza indicare alcun'implementazione; all'operazione è data una *segnatura* (il nome e la lista degli argomenti formali con i tipi relativi) ma l'implementazione è rimandata e deve essere fornita in ciascuna sottoclasse. Una tale *operazione* è detta *astratta*.

Il concetto di *operazione astratta* non dovrebbe essere confuso con quello di *classe astratta*. Quest'ultima è una classe che non possiede direttamente alcun oggetto istanza (ma le sue sottoclassi potrebbero avere oggetti istanza). Potrebbero non esserci istanze della classe `Vegetale`, essendo le istanze oggetti delle classi `Patata`, `Carota`, etc.

In realtà, una classe con un'operazione astratta è astratta. Un classe concreta, come `Mela`, non può possedere operazioni astratte (Hoffer *et al.*, 1999). Mentre le operazioni astratte sono rappresentate nelle *specifiche del comportamento*, le classi astratte appartengono alle *specifiche dello stato*.

4.2.4.1 Determinare le generalizzazioni

In genere, molte superclassi/sottoclassi sono introdotte già nella lista iniziale delle classi. Altre generalizzazioni sono determinate durante la definizione delle associazioni. Associazioni distinte (anche su una stessa classe) potrebbero coinvolgere classi a livelli differenti di generalizzazione/specializzazione. Ad esempio, la classe `Corso` potrebbe essere associata alla classe `Studente` (uno studente *sceglie* un corso) e alla classe `Assistente` (un assistente *insegna* in un corso); un'ulteriore analisi potrebbe mostrare che `Assistente` è una sottoclasse di `Studente`.

Un test significativo per la ricerca di generalizzazioni è usare le frasi "*può essere*" e "*è un tipo di*" nella descrizione della relazione tra due classi. In una descrizione "top-down", la frase è "può essere" (ad esempio, `Studente` "può essere" un `Assistente`). Nell'interpretazione "bottom up", la frase è "è un tipo di" (ad esempio, `Assistente` "è un tipo di" `Studente`). Si noti che se `Assistente` è considerato anche un tipo di `Insegnante` si ha un esempio di *ereditarietà multipla*.

4.2.4.2 Specificare le generalizzazioni

In una relazione di generalizzazione una classe eredita la struttura e il comportamento definiti in una o più altre classi. La generalizzazione è rappresentata in UML da un segmento continuo con una *freccia triangolare* orientata dalla parte della superclasse.

La specifica completa della generalizzazione comprende svariate opzioni. Ad esempio, è possibile definire l'*accesso*, identificando se una classe passa *diritti* a un'altra classe, decidendo cosa fare in situazione di *ereditarietà multipla*, etc. Questi problemi saranno discussi nel Capitolo 5.

4.2.4.3 Esempio di specifica di generalizzazioni

Esempio 4.10 (Video Store)

Con riferimento agli Esempi 4.2 e 4.5, le classi identificate in Figura 4.2 (Esempio 4.5) implicano una gerarchia di generalizzazione con radice la classe `SupportoVideo`. Esaminare le differenze di stato e di comportamento tra le diverse tipologie di `SupportoVideo` assumendo che il sistema Video Store abbia necessità di conoscere se una `CassettaVideo` è una cassetta originale oppure è una copia registrata (ciò è rappresentato dall'attributo `è_registrata`). Si assuma anche che la capacità di memorizzazione di un VideoDisco permetta di tenere più versioni dello stesso film, ciascuna in una lingua diversa o con finali differenti.

Si estenda quindi il modello in Figura 4.2 includendo le relazioni tra le classi e, in particolare, la specifica delle generalizzazioni.

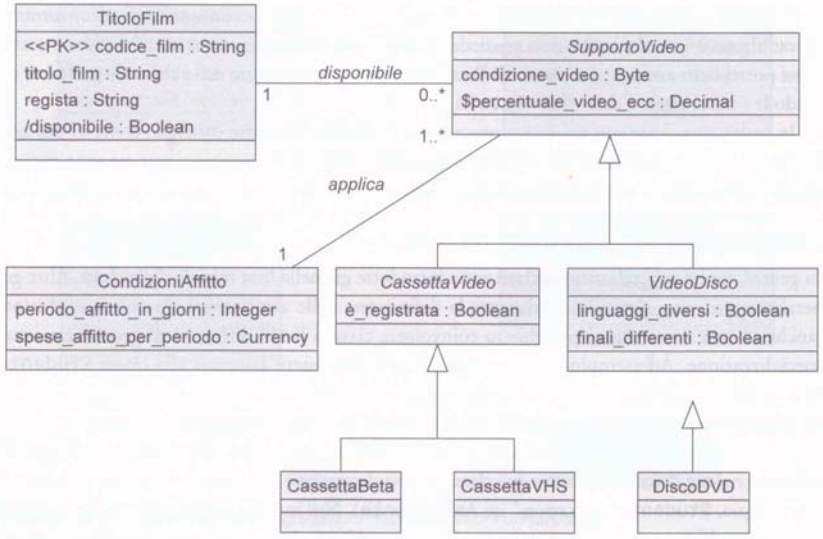


FIGURA 4.7 Specifica delle generalizzazioni (Video Store).

La Figura 4.7 è un modello esteso delle classi per il sistema Video Store. La gerarchia di generalizzazione indica che `SupportoVideo` "può essere" `CassettaVideo` oppure `VideoDisco`. `CassettaVideo` "può essere" `CassettaBeta` oppure `CassettaVHS`. `VideoDisco` "può essere" `DiscoDVD` (attualmente `VideoDisco` "deve essere" `DiscoDVD` ma s'ipotizza che nuove sottoclassi di `VideoDisco` siano in futuro possibili).

`SupportoVideo`, `CassettaVideo` e `VideoDisco` sono *classi astratte* (indicato dai nomi scritti in corsivo). Le classi astratte non permettono di istanziare oggetti. Gli oggetti di `SupportoVideo` sono anche istanze delle classi `CassettaBeta`, `CassettaVHS` e `DiscoDVD`. Quindi, ad esempio, l'associazione `disponibile` collega in realtà un oggetto di `TitoloFilm` a uno o più oggetti concreti della classe `SupportoVideo` (cioè a oggetti di `CassettaBeta`, `CassettaVHS` e/o `DiscoDVD`).

Il medesimo fatto è vero anche per l'associazione `applica`. Tuttavia, in questo caso, è possibile intravedere un'inefficienza nel modello. È noto dai requisiti del sistema Video Store che le condizioni d'affitto sono diverse per cassette video e dischi video dello stesso film. Ma tutte le cassette dello stesso film hanno le stesse condizioni d'affitto così come per tutti i video dischi dello stesso film. Ciò suggerisce che a un oggetto `CondizioniAffitto` sarebbe più corretto associare oggetti delle classi `CassettaVideo` e `VideoDisco`. Per ora, si rimanda ai capitoli successivi il trattamento di quest'inefficienza.

4.2.5 Modellazione degli oggetti

La modellazione s'interessa alle definizioni di sistemi. Un modello non è un sistema eseguibile e, di conseguenza, non mostra gli oggetti istanza. In genere, il numero di oggetti nel sistema è enorme e la loro rappresentazione grafica è impossibile. Tuttavia, durante la modellazione delle classi, frequentemente si fa riferimento a singoli oggetti e si discutono scenari complessi usando tali oggetti come esempi.

4.2.5.1 Specificare gli oggetti

UML definisce una rappresentazione grafica per un *oggetto* (Sezione 2.1.1.1). Si possono rappresentare diagrammi degli oggetti per illustrare complesse relazioni tra classi o per mostrare le modifiche agli oggetti nel tempo. I modelli degli oggetti possono essere usati per illustrare ed esaminare come gli oggetti collaborano durante un'esecuzione del sistema.

4.2.5.2 Esempio di specifica di oggetti

Esempio 4.11 (Immatricolazioni universitarie)

Lo scopo di quest'esempio è mostrare alcuni oggetti istanze di classi presenti nel modello delle classi di Figura 4.6 (Esempio 4.9).

La Figura 4.8 è un diagramma degli oggetti che corrisponde al modello delle classi di Figura 4.6. La figura mostra un oggetto `Studente` (Don Donaldson) che contiene fisicamente due oggetti `InformazioneAccademica` (relativa ai corsi COMP224 e COMP326). Don è attualmente immatricolato in due corsi: COMP225 e COMP325. Il corso COMP325 è stato un corso offerto nell'anno 2000 al secondo semestre. Rick Richards era il docente incaricato di quel corso.

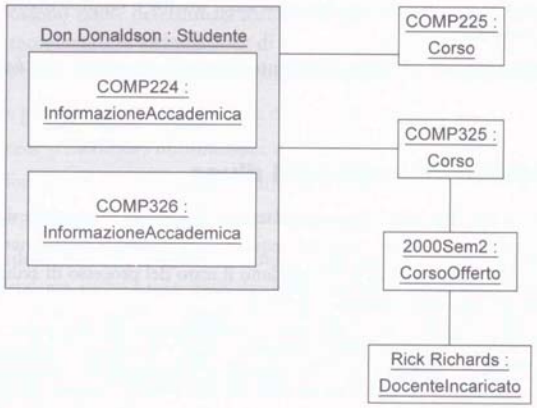


FIGURA 4.8 Diagramma degli oggetti per (Immatricolazioni universitarie).

4.3 Specifiche del comportamento

Il comportamento del sistema, come appare a un utente esterno, è descritto nei *casi d'uso* (Sezione 2.2.2). I modelli dei casi d'uso possono essere sviluppati a diversi livelli d'astrazione e si applicano al sistema in quanto tale al fine di specificare le sue principali funzionalità. Possono essere anche usati per rappresentare il comportamento di un *package* UML, parte di un package o anche di una *classe* all'interno del package.

Durante l'*analisi* i casi d'uso, rappresentando i requisiti del sistema, enfatizzano *cosa* il sistema fa o dovrebbe fare. Durante il *progetto*, le viste dei casi d'uso possono essere usate per specificare come deve essere implementato il comportamento del sistema.

Un caso d'uso richiede generalmente l'esecuzione di una computazione che avviene tramite interazioni tra oggetti. Le *computazioni* possono essere rappresentate con diagrammi d'attività. Le *interazioni* tra oggetti possono essere specificate con diagrammi di sequenza o diagrammi di collaborazione (i diagrammi di sequenza sono analizzati in questo capitolo mentre i diagrammi di collaborazione saranno usati nel capitolo sul progetto (Sezione 6.2)).

Le specifiche del comportamento forniscono una *vista operazionale* del sistema. Il compito principale è, a questo punto, definire i *casi d'uso* del dominio applicativo e determinare quali *classi* sono coinvolte nella loro esecuzione. S'identificano quindi le *operazioni* delle classi e lo *scambio di messaggi* tra gli oggetti. Sebbene le interazioni provochino delle modifiche nello stato degli oggetti, nelle specifiche del comportamento si definisce un *vista operazionale* su uno stato *bloccato* del sistema mentre le modifiche sullo stato degli oggetti sono esplicitamente rappresentate nelle *specifiche del cambiamento di stato*.

I modelli dei casi d'uso dovrebbero essere sviluppati iterativamente e concorrentemente ai modelli delle classi. Le classi determinate nelle specifiche dello stato sono ulteriormente elaborate e le operazioni più importanti sono identificate. Si rammenti, comunque, che le specifiche dello stato definiscono *classi entity* ("oggetti di business"). Con il procedere della modellazione del comportamento, s'incontrano altre due categorie di classi:

1. classi asservite a eventi del sistema (generati dagli utenti) e rappresentanti processi di business (*classi control*);
2. classi che rappresentano gli oggetti nell'interfaccia del sistema (*classi boundary*) (Sezione 5.2.4).

4.3.1 Modellazione dei casi d'uso

La modellazione dei casi d'uso è strettamente integrata con la determinazione dei requisiti (Capitolo 3). I requisiti testuali nel documento dei requisiti devono essere trasformati in casi d'uso nel documento delle specifiche. Se i casi d'uso guidano il resto del processo di sviluppo si dice che il processo è *guidato dal problema* (Sezione 4.2.1.1.3).

Analogamente alla modellazione delle classi, la modellazione dei casi d'uso è intrinsecamente *iterativa* e *incrementale*. Il diagramma iniziale dei casi d'uso può essere determinato dai requisiti di alto livello. Questo potrebbe essere un *modello dei casi d'uso di business* (Sezione 3.5.2). Ulteriori raffinamenti potrebbero essere guidati da requisiti più dettagliati. Se i requisiti utente variano durante il ciclo di vita dello sviluppo, tali variazioni sono prima apportate al documento dei requisiti e in seguito al modello dei casi d'uso. Le variazioni ai casi d'uso sono quindi riportate sugli altri modelli (Hoffer *et al.*, 1999; Rational, 2000).

4.3.1.1 Determinare i casi d'uso

Nella ricerca dei casi d'uso, l'analista deve assicurarsi che essi siano coerenti con le caratteristiche essenziali del concetto di caso d'uso. Un caso d'uso rappresenta (Hoffer *et al.*, 1999; Kruchten, 1999; Quatrani, 2000; Rumbaugh *et al.*, 1999):

1. una funzionalità *completa* (comprendendo il *flusso principale*, le sue deviazioni (*sottoflussi*), e le condizioni eccezionali (*flussi alternativi*));
2. una funzionalità *visibile esternamente* (non una funzione interna);
3. un comportamento *ortogonale* rispetto ai comportamenti espressi dagli altri casi d'uso (casi d'uso possono condividere oggetti durante l'esecuzione ma ciascun'esecuzione è indipendente dalle altre);
4. una funzionalità iniziata da un *attore* (ma, una volta iniziato, il caso d'uso può interagire con altri attori); tuttavia, è possibile che un attore riceva semplicemente i risultati di un caso d'uso iniziato da un altro attore;
5. una funzionalità che produce, al termine dell'esecuzione del singolo caso d'uso, un risultato significativo per un certo *attore*.

I *casi d'uso* si determinano con l'analisi di:

1. requisiti identificati nel documento dei requisiti;
2. attori e loro finalità nel sistema.

I problemi di gestione dei requisiti sono stati discussi nella Sezione 3.4. Si ricordi che i requisiti sono tipati e che, ai fini della ricerca dei casi d'uso, l'interesse è solo per i *requisiti funzionali*.

I casi d'uso possono essere determinati dall'analisi dei compiti a cui adempiono gli attori. Jacobson (1992) raccomanda la formulazione di un insieme di domande sugli attori. Le risposte a tali domande possono risultare nell'identificazione dei casi d'uso. Le domande sono:

- Quali sono le principali attività eseguite da ciascun attore?
- Un attore accede o modifica l'informazione nel sistema?
- Un attore informa il sistema su qualsiasi variazione in altri sistemi?
- Un attore dovrebbe essere informato su modifiche inattese nel sistema?

I casi d'uso riguardano non solo le necessità espresse dagli attori (in un certo senso, si può parlare di *casi d'uso d'attore*) ma anche i principali elementi funzionali di un sistema (*casi d'uso di sistema*). Sovente i casi d'uso di sistema estraggono ciò che è comune a diversi casi d'uso d'attore specificando servizi generici applicabili (anche attraverso l'ereditarietà) all'insieme dei casi. L'"attore" del caso d'uso di sistema è il progettista/programmatore, non l'utente. I casi d'uso del sistema sono identificati nella fase di progetto.

4.3.1.2 Specificare i casi d'uso

La specifica dei casi d'uso comprende la presentazione grafica di attori, casi d'uso, e di quattro tipologie di relazione (Fowler, 1999; Rumbaugh *et al.*, 1999):

1. Associazione.
2. Inclusione.
3. Estensione.
4. Generalizzazione.

La relazione d'*associazione* stabilisce una comunicazione tra un attore e un caso d'uso. La relazione di *generalizzazione* fra casi d'uso permette a un caso d'uso specializzato la modifica di un qualunque aspetto del caso d'uso di base.

Le relazioni *inclusione* ed *estensione* sono stereotipate dalle parole: `<<include>>` e `<<extend>>`. La relazione `<<include>>` specifica che un caso d'uso di base comprende sistematicamente il comportamento descritto dal caso d'uso incluso. Il caso d'uso incluso non è mai eseguito da solo ma solamente in quanto parte del caso d'uso di base. La relazione `<<extend>>` fornisce una forma controllata di estensione del comportamento di un caso d'uso di base per mezzo dell'attivazione di un altro caso d'uso in corrispondenza di specifici punti d'estensione. Il caso d'uso di base potrebbe operare in certi casi da solo e in altri (al verificarsi di certe condizioni) richiedere l'estensione (ma solo in specifici punti). La relazione `<<include>>` si differenzia dalla relazione `<<extend>>` poiché il caso d'uso "incluso" è sempre attivato mentre nell'estensione sono descritti comportamenti opzionali.

Nella pratica, la determinazione delle relazioni fra i casi d'uso richiede una certa cautela; dedicare troppo sforzo alla ricerca di quali relazioni si applichino a specifiche coppie di casi d'uso crea spesso confusione. Inoltre, i casi d'uso di alto livello tendono a intrecciarsi tanto che le loro relazioni possono dominare e oscurare il diagramma, spostando l'enfasi da una corretta identificazione dei casi d'uso alle relazioni fra gli stessi.

4.3.1.3 Esempi di specifica di casi d'uso

Esempio 4.12 (Immatricolazioni universitarie)

Riferendosi alla descrizione del sistema d'Immatricolazioni universitarie (Sezione 2.3.1) e ai requisiti definiti negli Esempi 4.1 e 4.4 (Sezione 4.2.1), determinare i casi d'uso dall'analisi dei requisiti funzionali.

La Figura 4.9 mostra un diagramma dei casi d'uso di alto livello per le Immatricolazioni universitarie. Il modello contiene quattro attori e quattro casi d'uso. Ogni caso d'uso è iniziato da un attore ed è una funzionalità completa, esternamente visibile e ortogonale alle altre. Tutti gli attori, ad eccezione di `Studente`, attivano qualche caso d'uso. `Studente` riceve i risultati d'esame e le istruzioni d'immatricolazione prima che il piano di studi per il prossimo semestre possa essere inserito e convalidato.

Il caso d'uso `Fornire Risultati Esame` *potrebbe* estendere il caso d'uso `Fornire Istruzioni Immatricolazione`. L'estensione non avviene sempre: ad esempio per i nuovi

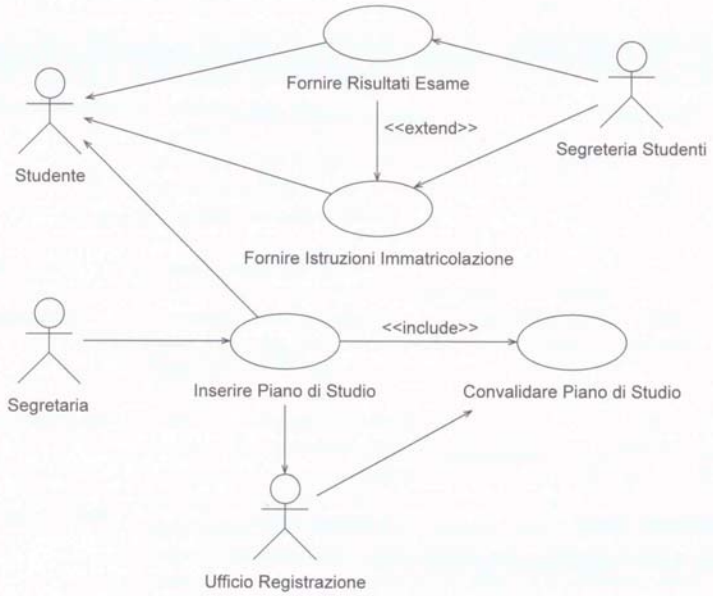


FIGURA 4.9 Diagramma dei casi d'uso (Immatricolazioni universitarie).

studenti i risultati d'esame non esistono. Ciò è coerente con l'uso della relazione `<<extend>>` piuttosto che con `<<include>>`.

Tra il caso d'uso `Inserire Piano di Studi` e il caso d'uso `Convalidare Piano di Studi` è stata introdotta una relazione `<<include>>` perché il primo caso d'uso include sempre il secondo: ogniqualvolta il piano di studi è inserito, deve essere convalidato rispetto a sovrapposizioni d'orario, approvazioni speciali, etc.

Esempio 4.13 (CMS)

Riferendosi alla descrizione del sistema CMS (Sezione 2.3.3) e ai requisiti definiti negli Esempi 4.3 e 4.6 (Sezione 4.2.1), determinare i casi d'uso dall'analisi dei requisiti funzionali.

Ci sono tre attori e cinque casi d'uso nel diagramma dei casi d'uso per il sistema CMS (Figura 4.10). Un aspetto interessante del modello è che tra gli attori esistono relazioni di generalizzazione. Il `Manager Servizi Cliente` "è un tipo di" `Impiegato Servizi Cliente` il quale, a sua volta, "è un tipo di" `Impiegato`. La gerarchia di generalizzazione migliora l'espressività del diagramma. Qualsiasi evento trattato da un `Impiegato` può anche essere trattato da un `Impiegato Servizi Cliente` oppure da un `Manager Servizi Cliente`. Di conseguenza, `Manager Servizi Cliente` è implicitamente associato con (e può attivare) ogni caso d'uso.

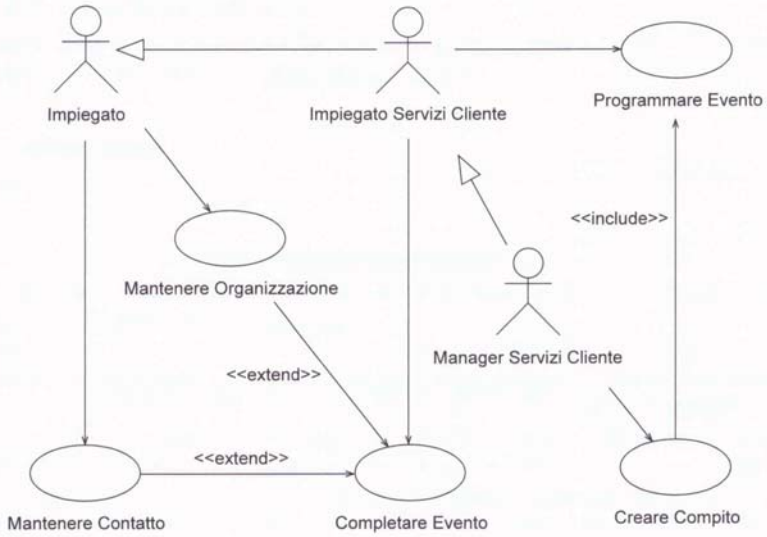


FIGURA 4.10 Diagramma dei casi d'uso (CMS).

Il caso d'uso `Creare Compito <<include>> Programmare Evento` a causa del requisito per cui un compito non può essere creato senza aver programmato il primo evento. Le relazioni `<<extend>>` indicano che il completamento di un evento può attivare modifiche ai dettagli dell'organizzazione o del contatto.

Esempio 4.14 (Video Store)

Riferendosi alla descrizione del sistema Video Store (Sezione 2.3.2) e ai requisiti definiti negli Esempi 4.2 e 4.5 (Sezione 4.2.1), determinare i casi d'uso dall'analisi dei requisiti funzionali.

Per uno dei casi d'uso, scrivere le specifiche mediante delle frasi associate ai seguenti titoli: sintesi, attori, precondizioni, descrizione, eccezioni e postcondizioni.

Ci sono sei casi d'uso e solo due attori nel diagramma dei casi d'uso per il sistema Video Store (Figura 4.11). Gli attori secondari, come `Cliente` o `Fornitore`, non sono mostrati (tali attori non attivano alcun caso d'uso). L'attore `Impiegato` attiva (direttamente o implicitamente) tutti i casi d'uso. La relazione tra gli attori `Dispositivo Lettura` e `Impiegato` è una *dipendenza*, stereotipata dall'analista con la frase `<<dipende da>>`.

Ogni caso d'uso nel diagramma deve essere ulteriormente documentato nel repository dello strumento CASE. In particolare, è necessaria una descrizione testuale. La Tabella 4.4 descrive "AffittareVideo" usando una comune struttura per la specifica testuale dei casi d'uso (Sezione 2.2.2.4).

TABELLA 4.4 Specifica testuale per il caso d'uso "Affittare Video" (Video Store).

Caso d'uso	Affittare Video
Breve descrizione	Un cliente richiede l'affitto di una cassetta video o disco preso dagli scaffali della Video Store o che è stato precedentemente riservato dal cliente stesso. Supponendo che il cliente sia affidabile, la cassetta è data in affitto una volta che il pagamento sia stato ricevuto. Se la cassetta non è restituita in tempo, un'informativa è inviata al cliente.
Attori	Impiegato, Dispositivo lettura.
Precondizioni	Una cassetta video o un disco è disponibile per essere affittato. Il cliente ha la tessera della Video Store. I dispositivi di lettura funzionano correttamente. L'impiegato sa come usare il sistema.
Flusso principale	Un cliente può chiedere all'impiegato la disponibilità di un video (incluso un video prenotato) o prendere dagli scaffali una cassetta o un disco. Il video e la tessera vengono letti e qualsiasi dettaglio di "inadempienza" o ritardo è evidenziato all'impiegato il quale deve richiedere al cliente chiarimenti. Se il cliente non è "inadempiente", allora può prendere in affitto un massimo di otto video. Tuttavia, se il cliente è "inaffidabile" allora questi deve versare un deposito pari ad un periodo d'affitto per ciascuna cassetta o disco. Una volta che la somma da pagare è ricevuta, la scorta è aggiornata e le cassette e i dischi sono dati al cliente insieme alla ricevuta dell'affitto. Il cliente paga in contanti, carta di credito o bonifico. Ogni record d'affitto memorizza (sotto il conto del cliente) le date di uscita e di restituzione, insieme all'identificazione dell'impiegato. Un record d'affitto distinto è creato per ogni video affittato. Il caso d'uso genera un'informativa al cliente se il video non è stato riportato entro due giorni dalla data di restituzione prevista, e una seconda informativa dopo altri due giorni (a quel punto, il cliente è annotato come "inadempiente").
Flussi alternativi	Un cliente non ha la tessera (il caso d'uso "Gestire Cliente" potrebbe essere attivato per fornire una nuova tessera). Un cliente cerca di affittare troppi film. Non può essere affittato il film richiesto al cliente poiché questi è " inadempiente". Il supporto video o la tessera cliente non possono essere letti a causa del loro danneggiamento. Il bonifico o la carta di credito è rifiutato.
Postcondizioni	Il film è dato in affitto e il database è aggiornato di conseguenza.

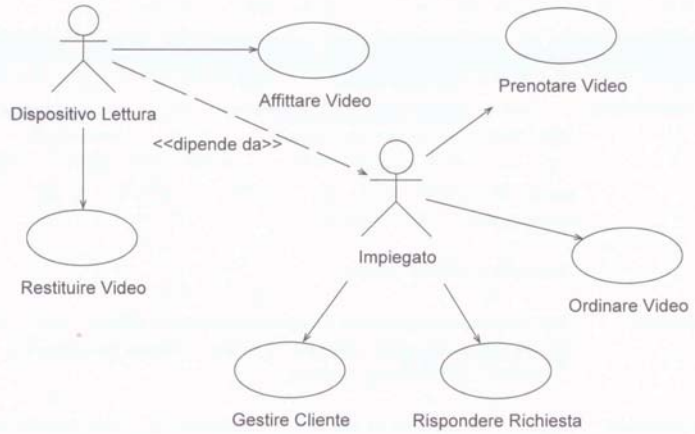


FIGURA 4.11 Diagramma dei casi d'uso (Video Store).

Esempio 4.15 (Telemarketing)

Riferendosi alla descrizione del sistema Telemarketing (Sezione 2.3.4) e ai requisiti definiti negli Esempi 3.1, 3.2, 3.3 e 4.7, determinare i casi d'uso dall'analisi dei requisiti funzionali.

La soluzione in Figura 4.12 comprende vari casi d'uso. Gli attori comunicano direttamente con tre casi d'uso: `Programmare` e `Fare Prossima Chiamata`, `Registrare Risultato Chiamata` e `Visualizzare Dettagli Chiamata`. Quest'ultimo può essere esteso a `Visualizzare Dettagli Campagna`, `Visualizzare Storia Sostenitore` e `Visualizzare Dettagli Premio`. Il caso d'uso `Aggiornare Sostenitore` estende `Visualizzare Storia Sostenitore`. `Registrare Vendita Biglietto` e `Programmare Richiamo` possono estendere `Registrare Risultato Chiamata`. Si assume che gli attori possano comunicare indirettamente con i casi d'uso d'estensione ma nessun'esplicita comunicazione è rappresentata nel diagramma.

4.3.2 Modellazione delle attività

I diagrammi (o grafi) d'attività sono stati introdotti recentemente in UML (Sezione 2.2.3). Come i tradizionali diagrammi di flusso (*flow chart*) e di struttura (*structure chart*) resi popolari dai metodi strutturati per il progetto di programmi procedurali, i diagrammi d'attività rappresentano il flusso logico nei programmi object-oriented (sebbene ad alto livello d'astrazione). La differenza consiste nella possibilità di rappresentare, nei diagrammi d'attività, il *controllo concorrente* accanto al *controllo sequenziale*.

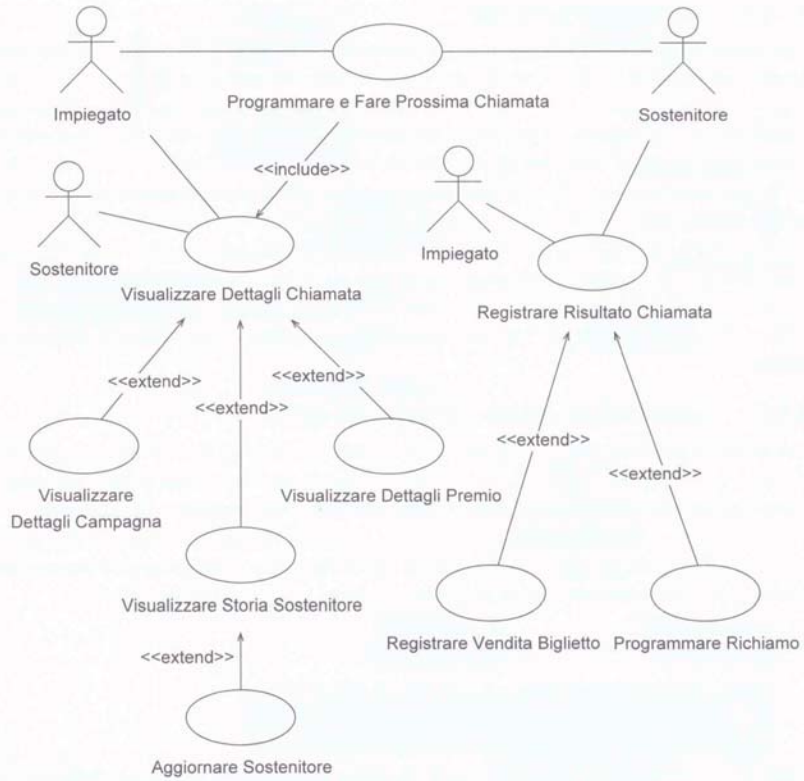


FIGURA 4.12 Diagramma dei casi d'uso (Telemarketing).

I modelli delle attività sono largamente usati nella *progettazione*. Tuttavia, permettono di esprimere computazioni o flussi di controllo ad alto livello d'astrazione durante l'*analisi*. I grafi di attività possono essere usati per mostrare vari livelli di dettaglio di una computazione.

I modelli delle attività possono essere particolarmente utili per definire il flusso delle *attività* nell'esecuzione di un caso d'uso. Poiché i modelli delle attività non mostrano gli *oggetti* che svolgono le attività, può essere costruito un grafo delle attività anche se il modello delle classi non è stato sviluppato oppure è in corso di sviluppo; ogni attività sarà definita infine da una o più operazioni in una o più classi che collaborano (il progetto dettagliato per tali collaborazioni sarà probabilmente rappresentato con un diagramma di collaborazione (Sezione 6.2.2)).

4.3.2.1 Determinare le attività

Ogni caso d'uso può essere rappresentato per mezzo di uno o più grafi di attività. L'evento prodotto da un attore che inizia un caso d'uso è lo stesso evento che attiva l'esecuzione del grafo delle attività. L'esecuzione procede da uno *stato d'attività* al successivo. Uno stato d'attività termina quando la sua computazione è terminata. Interruzioni basate su eventi esterni, che potrebbero causare la terminazione dello stato attività, sono permessi solo in situazioni eccezionali (se si pensa che esse siano frequenti, allora un diagramma statechart dovrebbe essere usato al posto di quello delle attività).

Le attività si determinano soprattutto con l'analisi delle frasi nelle *specifiche testuali dei casi d'uso* (Tabella 4.4). Qualsiasi frase verbale è un'attività candidata. L'analisi degli stati delle attività e la descrizione dei flussi di controllo può condurre alla scoperta di flussi alternativi o concorrenti che introducono separazioni o biforcazioni nel diagramma delle attività (come sarà illustrato in seguito).

4.3.2.2 Specificare le attività

Una volta noti gli stati d'attività, la specifica delle attività è un processo relativamente semplice che connette le attività stesse tramite *linee di transizione*. Processi concorrenti sono *attivati* (biforcati) e *riuniti* con *barre di sincronizzazione*. Processi alternativi sono *creati* (*separati*) e *fusi* con *rombi*.

Gli eventi esterni non fanno normalmente parte dei grafi d'attività. Comunque, se devono essere riportati sul grafo esistono tecniche grafiche adatte allo scopo. Analogamente, esiste una notazione grafica per descrivere il flusso di oggetti in ingresso per, o in uscita da, un'attività.

4.3.2.3 Esempio di specifica di attività

Esempio 4.16 (Video Store)

Riferendosi all'Esempio 4.14 e in particolare alla specifica testuale del caso d'uso "Affittare Video" (Tabella 4.4), rappresentare un diagramma delle attività per "Affittare Video".

La Figura 4.13 mostra un diagramma delle attività per il caso d'uso "Affittare Video". Non è sorprendente che il diagramma rifletta la specifica testuale del caso d'uso (Tabella 4.4). L'elaborazione inizia quando è letta la tessera cliente o il supporto video: le due attività sono tra loro indipendenti (questo è mostrato per mezzo della barra di biforcazione).

L'attività `Verificare Cliente` controlla la storia del cliente e la computazione procede dopo la valutazione delle condizioni decisionali. Se un cliente è "`inadempiente`" allora "Affittare Video" termina. Se un cliente è inaffidabile allora si richiederà un deposito prima di concludere la transazione (il caso d'uso sarà poi fatto ripartire). Altrimenti, il cliente è affidabile e l'attività `Iniziare Transazione Affitto` "scatta" (è eseguita).

La condizione `[più di 8 video]` assicura che non più di otto video siano affittati a un singolo cliente (bisognerà verificare che alcuni video siano restituiti prima di procedere). Dopo che il pagamento è stato accettato, un'altra biforcazione permette l'esecuzione concorrente delle attività `Aggiornare Scorta` e `Stampare Ricevuta`. Tali processi concorrenti sono riuniti prima che l'attività `Termina Transazione Affitto` scatti e l'intera elaborazione sia terminata.

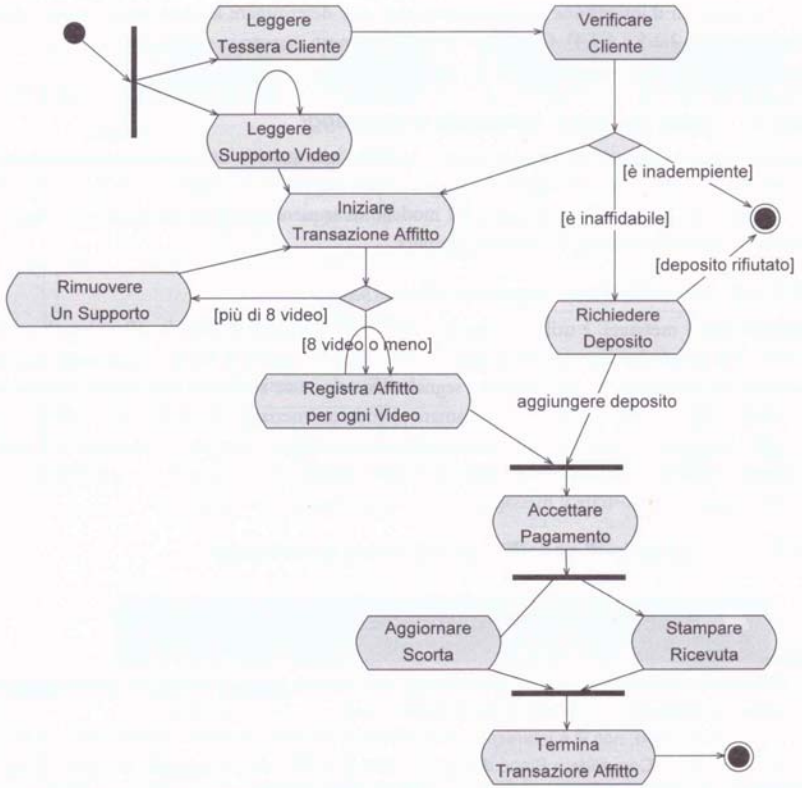


FIGURA 4.13 Diagramma delle attività per il caso d'uso "Affittare Video" (Video Store).

4.3.3 Modellazione delle interazioni

I diagrammi di sequenza e di collaborazione sono due tipologie di diagramma d'interazione. Mostrano strutture (pattern) d'*interazione* tra oggetti, necessarie per portare a compimento un caso d'uso, un'operazione o un'altra componente comportamentale.

I diagrammi di sequenza mostrano lo scambio di messaggi tra oggetti, organizzato in sequenza temporale. I diagrammi di collaborazione enfatizzano le relazioni (tra oggetti) lungo cui i messaggi sono scambiati. Si è già osservato come i diagrammi di sequenza siano più utili nell'analisi e i diagrammi di collaborazione nel progetto.

Poiché i modelli d'interazione fanno riferimento a oggetti, richiedono che almeno la prima iterazione nella modellazione dello stato sia stata completata e le classi principali siano state identificate. Sebbene le interazioni interessino gli stati degli oggetti, i diagrammi d'interazione non rappresentano esplicitamente le modifiche di stato negli oggetti. Questo, infatti, è compito delle specifiche del cambiamento di stato e dei diagrammi statechart (Sezioni 2.2.6 e 4.4).

I diagrammi d'interazione possono essere usati per determinare le *operazioni* (metodi) nelle classi (Sezione 2.2.5 e 4.3.4). Qualsiasi messaggio verso un oggetto in un diagramma d'interazione deve essere trattato da un qualche metodo nella classe di quell'oggetto.

4.3.3.1 Determinare le sequenze di messaggi

La ricerca delle sequenze di messaggi segue i modelli delle attività. Le *attività* in un diagramma d'attività sono associate a *messaggi* in un diagramma di sequenza. Se i livelli d'astrazione usati nella costruzione del modello d'attività e del modello di sequenza sono simili, allora la corrispondenza tra attività e messaggi è piuttosto semplice.

4.3.3.2 Specificare le sequenze di messaggi

Specificando i messaggi, è utile distinguere i messaggi in segnali e chiamate (Rumbaugh *et al.*, 1999). Un *segnale* denota una comunicazione asincrona tra oggetti; il mittente può continuare la propria esecuzione non appena inviato il segnale. Una *chiamata* indica un'invocazione sincrona di un'operazione; il mittente, emessa la chiamata attende un messaggio di ritorno per riprendere la propria esecuzione. Il messaggio di ritorno può contenere alcuni valori per il chiamante o, semplicemente, notificare che l'operazione invocata è stata completata con successo; in quest'ultimo caso, è possibile non mostrare il messaggio di ritorno nel diagramma di sequenza.

4.3.3.3 Esempio di specifica di sequenze di messaggi

Esempio 4.17 (Immatricolazioni universitarie)

Riferendosi agli Esempi 4.9 e 4.12 e al caso d'uso "Inserire Piano di Studio" in Figura 4.9, costruire un diagramma di sequenza per tale caso d'uso.

In questo caso, non vi è interesse per la verifica di un piano di studio inserito. Infatti, un altro caso d'uso ("Convalidare Piano di Studio") tratta la verifica dei prerequisiti, sovrapposizioni d'orario, approvazioni speciali, etc. È solo necessario controllare se il corso nel quale lo studente vuole essere registrato è offerto nel prossimo semestre e se è ancora aperto (ovvero ci sono ancora posti disponibili).

La Figura 4.14 mostra il diagramma di sequenza per il precedente scenario. L'attore `Segretaria` inizia il caso d'uso inviando un messaggio di richiesta a un oggetto della classe di presentazione `FinestraInserimentoPiano` per aggiungere uno studente (identificato dall'argomento `std`) a un corso (argomento `crs`) in un semestre (argomento `sem`).

L'oggetto della classe `FinestraInserimentoPiano` verifica con l'oggetto `unoStudente` se questo può essere immatricolato (per esempio, se ha pagato le tasse). L'argomento di output `s_check` ritorna il valore '`si`' o '`no`' a `:FinestraInserimentoPiano`. Se il risultato è '`no`', l'immatricolazione non può procedere e l'oggetto `:FinestraInserimentoPiano` invia un messaggio `distruggi` a se stesso (*self*) per distruggersi; la *condizione* `[s_check='no']` specifica l'opzionalità dell'operazione. L'uso di un messaggio *self* per indicare la distruzione dell'oggetto è una "scorciatoia"; in realtà, un oggetto istanza non può distruggere se stesso e il messaggio `distruggi` deve essere inviato all'oggetto classe. UML ha anche una notazione specifica per la

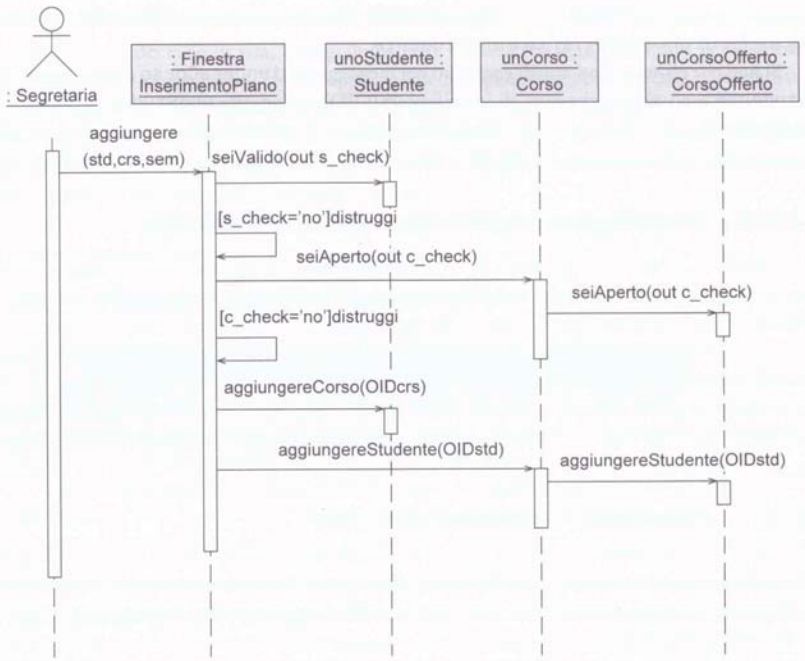


FIGURA 4.14 Diagramma di sequenza per il caso d'uso "Inserire Piano di Studio" (Immatricolazioni universitarie).

distruzione d'oggetto, una grande X posta sulla linea di vita dell'oggetto nel punto in cui l'oggetto è distrutto.

Se `unoStudente` può essere immatricolato allora è necessario verificare se `unCorso` è ancora aperto. Per rispondere a tale richiesta, l'oggetto `unCorso` deve interrogare il suo attuale oggetto `unCorsoOfferto` (identificato seguendo l'aggregazione da `Corso` a `CorsoOfferto` (Figura 4.6)). Se non ci sono posti disponibili nell'attuale oggetto `unCorsoOfferto`, l'immatricolazione non può procedere e l'oggetto `:FinestraInserimentoPiano` invia nuovamente un messaggio *self* a se stesso per distruggersi.

Se l'immatricolazione può andare avanti, l'oggetto `:FinestraInserimentoPiano` richiede due operazioni: a `unoStudente` per aggiungere a se stesso `unCorsoOfferto`, e a `unCorsoOfferto` per aggiungere a se stesso `unoStudente`. La sequenza delle due operazioni può essere invertita se il programma assicura l'integrità referenziale tra `unCorsoOfferto` e `unoStudente` (cioè se uno studente è immatricolato in un corso offerto allora il corso offerto deve avere lo studente nella sua lista di studenti).

Si noti che se la memorizzazione degli oggetti di `Studente` e `CorsoOfferto` avviene in un database allora l'oggetto `:FinestraInserimentoPiano` invierebbe solo uno dei due messaggi, ad esempio, `aggiungereCorso` a `unoStudente` e, una volta che questo è stato eseguito,

sarebbe compito del DBMS che gestisce il database mantenere l'integrità referenziale inserendo un legame da `unCorsoOfferto` a `unoStudente`.

Si noti che gli argomenti attuali contenuti nei messaggi `aggiungereCorso` e `aggiungereStudente` sono, rispettivamente, gli identificatori (OID) dell'oggetto `unoStudente` o `unCorsoOfferto`. Tali OID sono stati determinati quando `:FinestraInserimentoPiano` ha avuto accesso agli oggetti `unoStudente` e `unCorso` con i messaggi `seiValido` e `seiAperto`.

4.3.4 Modellazione delle interfacce pubbliche

Un'*interfaccia pubblica* di una classe è determinata dall'insieme di *operazioni* che la classe offre come servizi ad altre classi del sistema. Tali operazioni sono dichiarate con una visibilità pubblica. È solo attraverso le interfacce pubbliche che gli oggetti possono collaborare per eseguire i casi d'uso.

Le interfacce pubbliche sono inizialmente determinate alla fine della fase d'analisi, quando le specifiche dello stato e del comportamento sono largamente definite. Nell'analisi, si definisce solo la *segnatura* di ciascuna operazione pubblica (nome operazione, lista dei tipi degli argomenti formali e tipo di ritorno). La definizione del *metodo* che implementa l'operazione è invece compito del progetto.

4.3.4.1 Determinare le operazioni delle classi

Le operazioni delle classi si evincono soprattutto dai diagrammi di sequenza. Ciascun *messaggio* in un diagramma di sequenza, oltre che restituire un valore, deve essere trattato da un'*operazione* nell'oggetto destinazione (Sezione 2.2.5.2). Se i modelli di sequenza sono interamente sviluppati, la determinazione delle operazioni pubbliche è automatica.

Nella pratica, comunque, i diagrammi di sequenza, anche se sviluppati per tutti i casi d'uso, possono non fornire sufficienti dettagli per permettere di determinare le operazioni pubbliche che potrebbero entrare a far parte di una classe. Inoltre, i diagrammi di sequenza possono non essere disponibili per operazioni che interessano diversi casi d'uso (ad esempio, complesse transazioni di business).

Per tali ragioni, metodi supplementari per la ricerca di operazioni possono fornire un buon aiuto. Uno di tali metodi viene dall'osservazione che gli oggetti sono responsabili del proprio "destino" e, di conseguenza, devono offrire supporto alle quattro operazioni primitive:

1. create;
2. read;
3. update;
4. delete.

Queste operazioni (note con l'acronimo CRUD, Sezione 3.5.2.1) permettono a un oggetto di inviare messaggi a un altro oggetto per richiedere:

1. un nuovo oggetto istanza;
2. l'accesso allo stato dell'oggetto;
3. la modifica dello stato dell'oggetto;
4. che un oggetto distrugga se stesso.

4.3.4.2 Specificare le operazioni delle classi

A questo stadio del ciclo di vita, il diagramma delle classi deve essere modificato per inglobare tutte le *segnature di operazione*. Anche l'applicabilità dell'operazione potrebbe essere determinata. In mancanza di tale informazione, si assume l'applicabilità a *livello d'istanza* (l'operazione si applica a *oggetti istanza*). L'applicabilità a *livello di classe* (*statica*) deve essere esplicitamente dichiarata (con il simbolo '$' prima del nome dell'operazione). L'applicabilità a livello di classe stabilisce che l'operazione si applica all'*oggetto classe* (Sezione 2.1.6).

Altre proprietà delle operazioni, come concorrenza, comportamento polimorfo e specifica algoritmica, saranno trattate più avanti, cioè durante la progettazione.

4.3.4.3 Esempio di specifica di operazioni delle classi

Esempio 4.18 (Immatricolazioni universitarie)

Si considerino gli Esempi 4.9 e 4.17 e le classi `Corso` e `CorsoOfferto` in Figura 4.6 (oggetti di tali classi sono stati usati nel diagramma di sequenza della Figura 4.14).

Derivare le operazioni dal diagramma di sequenza e aggiungerle alle classi `Corso` e `CorsoOfferto`.

Le operazioni specificate nel modello esteso delle classi in Figura 4.15 sono ottenute direttamente dal diagramma di sequenza di Figura 4.14. Le icone davanti agli attributi indicano che gli attributi sono *privati*, quelle davanti alle operazioni indicano che tali operazioni sono *pubbliche*.

Le operazioni riportate sono `Corso.seiAperto`, `CorsoOfferto.seiAperto`, `Corso.aggiungereStudente` e `CorsoOfferto.aggiungereStudente`. Si assume che un oggetto di `Corso` ottenga la lista dei propri studenti accedendo ai suoi oggetti componenti, di classe `CorsoOfferto`.

Sebbene gli attributi di relazione non siano normalmente mostrati nei modelli d'analisi (sono impliciti nei legami di relazione), sono stati rappresentati tre attributi di relazione necessari alle operazioni: `Corso.crs_off`, `CorsoOfferto.std` e `CorsoOfferto.crs`. Si noti che `crs_off` e `std` hanno *tipo parametrico*, sono cioè *collezioni* (`set` e `list`) di oggetti.

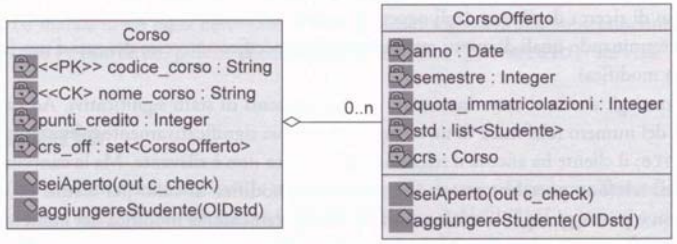


FIGURA 4.15 Specifica delle operazioni delle classi (Immatricolazioni universitarie).

4.4 Specifiche del cambiamento di stato

Lo stato di un oggetto in un certo istante di tempo è determinato dai valori dei suoi attributi, inclusi gli attributi di relazione. Mentre le specifiche dello stato definiscono (tra l'altro) gli attributi di una classe, le specifiche del comportamento definiscono le operazioni di una classe, alcune delle quali possono avere "*effetti collaterali*" (cioè alterano lo stato di un oggetto). Tuttavia, per capire come un oggetto può modificare il suo stato nel tempo, è necessario disporre di una vista più specifica del sistema. Tale vista è fornita dalle specifiche del cambiamento di stato.

L'importanza delle specifiche del cambiamento di stato varia a seconda dei domini applicativi. La rappresentazione delle modifiche di stato in applicazioni di business è, in genere, molto meno critica che in *applicazioni ingegneristiche* e *real-time*. Queste ultime, infatti, sono spesso interamente orientate alle variazioni di stato. Nella modellazione di tali sistemi, è necessario concentrarsi in partenza sui cambiamenti di stato: Cosa accade se la temperatura è troppo alta? Cosa accade se la valvola è chiusa? Cosa accade se il contenitore è pieno? Etc.

In questo libro l'interesse predominante è sulle *applicazioni di business* dove l'analisi delle modifiche di stato è meno importante, per cui, in genere, è tipicamente svolta verso la fine dell'analisi (e approfondita in fase di progetto). Molte delle specifiche del cambiamento di stato prendono in considerazione *condizioni eccezionali* nel sistema e quindi sono rappresentate dopo che il comportamento normale del sistema è stato specificato.

4.4.1 Modellazione dell'evoluzione dello stato degli oggetti

La rappresentazione dell'evoluzione dello stato degli oggetti è fatta con diagrammi statechart (Sezione 2.2.6), cioè mediante grafi bipartiti di *stati* e *transizioni* (queste ultime causate da *eventi*). Modelli di evoluzione degli stati sono costruiti per ogni classe che esibisca un interessante *comportamento dinamico*. Non tutte le classi nel diagramma delle classi cadono in tale categoria, in molti casi l'evoluzione degli oggetti delle classi è poco interessante.

I diagrammi statechart possono essere anche usati per descrivere il comportamento dinamico di altri elementi modellistici, ad esempio i casi d'uso, le collaborazioni o le operazioni. Ciò è comunque non frequente e alcuni strumenti CASE potrebbero non offrire, in questo caso, alcun supporto.

4.4.1.1 Determinare gli stati degli oggetti

Il processo di ricerca degli stati degli oggetti è basato sull'analisi degli attributi contenuti in una classe, determinando quali di questi attributi sono di specifico interesse dei casi d'uso (che li utilizzano in modifica).

Non tutti gli attributi sono interessati da cambiamenti di stato significativi. Ad esempio, la modifica del numero telefonico di un cliente non cambia significativamente lo stato di un oggetto `Cliente`: il cliente ha ancora il telefono e la modifica non è rilevante. Ma la cancellazione del numero di telefono potrebbe essere un'interessante modifica di stato per alcuni casi d'uso: il cliente non è più raggiungibile telefonicamente! Analogamente, la modifica del numero telefonico con modifica del prefisso potrebbe essere una variazione di stato interessante, indicante che il cliente si è spostato in un'altra zona; tale cambiamento potrebbe essere registrato e rappresentato in un diagramma statechart.

4.4.1.2 Specificare gli stati degli oggetti e la loro evoluzione

La notazione UML di base per le specifiche statechart è stata descritta nella Sezione 2.2.6. Per usare correttamente tale notazione, un analista deve comprendere come i diversi concetti sono interrelati, quali combinazioni di concetti sono vietate o non sono naturali, e quali scorciatoie di notazione sono possibili. È probabile, a tal proposito, che lo strumento CASE introduca alcune restrizioni ma potrebbe anche introdurre delle interessanti estensioni.

Una transizione di stato scatta quando un certo evento accade *oppure* una certa condizione è soddisfatta. Quest'ultimo fatto significa, ad esempio, che una linea di transizione non deve necessariamente essere etichettata con il *nome di un evento*. Una *condizione* (scritta tra parentesi quadre) permette lo scatto di una transizione in uscita da un'attività ogniqualvolta l'*attività* è completata e la condizione è vera.

Comunemente, una transizione è attivata da un evento di tipo *segnale* (comunicazione asincrona) o *chiamata* (comunicazione sincrona) (Sezione 4.3.3.2). Altre due tipologie di evento sono: *eventi modifica*, ed *eventi temporali*. In particolare, gli eventi temporali permettono lo scatto di transizioni in base a una nozione di tempo relativo o assoluto, e sono molto utili in alcuni modelli (ad esempio, alle 8 di mattina, ogni settimana, dopo due ore, etc.).

Un'altra considerazione nella modellazione con statechart riguarda la possibilità di specificare *azioni d'ingresso* dentro a un'icona di stato o su una transizione entrante. Analogamente, *azioni d'uscita* possono essere poste dentro a icone di stato o su transizioni uscenti; la scelta di quale tecnica usare può influenzare la leggibilità del modello (si veda Rumbaugh *et al.*, 1991).

4.4.1.3 Esempio di specifica di diagrammi statechart

Esempio 4.19 (Video Store)

Riferendosi alla classe `TitoloFilm` nell'Esempio 4.9 (Figura 4.7), specificarne l'evoluzione con un diagramma statechart.

Il diagramma statechart per `TitoloFilm` è illustrato in Figura 4.16. Il diagramma mostra i differenti modi in cui le transizioni possono essere specificate. Le transizioni tra gli stati `Disponibile` e `Non in Scorta` specificano nomi di eventi parametrizzati, insieme ai nomi delle azioni e alle condizioni di guardia. La transizione dallo stato `Prenotato` a `Non Prenotato` manca di un esplicito evento d'attivazione: essa scatterà al completamento dell'attività nello stato `Prenotato`, purché la condizione `[non più prenotato]` sia vera.

Sommario

In questo capitolo sono state discusse le tecniche integrate di rappresentazione fornite da UML, utilizzando i casi di studio introdotti nel Capitolo 2. Tutti i modelli UML più usati sono stati illustrati. Le discussioni pratiche suscitate dall'analisi dei casi di studio possono riassumersi in uno slogan: non ci sono soluzioni nere o bianche, vere o false nei modelli di analisi per costruire i sistemi informativi! Per ciascun problema possono esistere diverse potenziali soluzioni. L'abilità

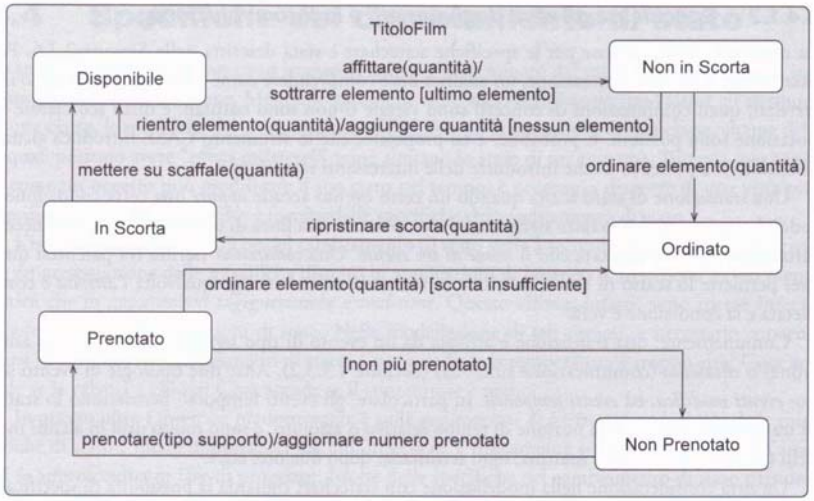


FIGURA 4.16 Diagramma statechart per la classe "TitoloFilm" (Video Store).

dell'analista risiede nella sua capacità di trovare una soluzione che soddisfi i requisiti del cliente e che funzioni correttamente.

- Le specifiche dello stato descrivono il dominio di analisi dal punto di vista *statico* delle classi, dei loro *attributi* e delle loro *relazioni*. Ci sono molti metodi per *determinare le classi* ma nessuna "ricetta" è la migliore. Un'integrazione di metodi adattata alla conoscenza e all'esperienza dell'analista è una risposta pratica a questo problema. Le classi sono specificate con *diagrammi delle classi* UML. I diagrammi visualizzano le classi e tre tipologie di relazione tra esse: *associazioni*, *aggregazioni* e *generalizzazioni*.
- Le specifiche del comportamento descrivono il dominio di analisi da una prospettiva *operazionale* (per non usare il termine abusato di prospettiva *funzionale*). Le "linee guida" per le specifiche di comportamento o meglio, per l'analisi dei requisiti e per il progetto di sistema, sono i *casi d'uso*. I *diagrammi dei casi d'uso* forniscono solo una semplice visualizzazione ma la reale forza dei casi d'uso è nelle loro *specifiche testuali*. Altri diagrammi di comportamento sono derivati dai modelli dei casi d'uso e comprendono i *diagrammi d'attività*, i *diagrammi d'interazione* e, infine, le *operazioni* delle classi.
- Le specifiche del cambiamento di stato descrivono il dominio di analisi da una prospettiva *dinamica*. Gli oggetti sono "tempestati" da eventi, alcuni dei quali causano modifiche allo *stato* degli oggetti stessi. I *diagrammi statechart* permettono la rappresentazione delle modifiche allo stato.

Domande

D1 Discutere come i requisiti informativi e funzionali identificati nella fase di determinazione dei requisiti sono in relazione con i modelli di stato, di comportamento e di cambiamento di stato nella fase di specifica dei requisiti.

D2 Discutere i pro e i contro nell'uso di strumenti CASE per la specifica dei requisiti.

D3 Spiegare le principali differenze tra i quattro approcci per la ricerca delle classi.

D4 Discutere e confrontare i processi associati rispettivamente con la ricerca e la specifica degli attributi e delle operazioni delle classi. Modellereste gli attributi e le operazioni in parallelo o separatamente? Perché?

D5 Riferendosi al modello delle classi in Figura 4.7 (Esempio 4.10), spiegare perché `TitoloFilm.disponibile` è rappresentato come attributo derivato mentre `SupportoVideo.percentuale_video_ecc` come attributo statico.

D6 Associazioni ternarie e derivate non sono desiderabili nei modelli di analisi. Perché? Fornire esempi.

D7 Si consideri il modello delle classi nella Figura 4.5 (Esempio 4.8) e, in particolare, la parte alta del modello che introduce le classi `IndirizzoPostale`, `IndirizzoCorriere`, `Organizzazione`, `Contatto`, e le relative associazioni. Si pensi a modi differenti di rappresentare i medesimi requisiti. Il modello può essere reso più flessibile per trattare potenziali modifiche ai requisiti? Quali sono i pro e i contro delle differenti soluzioni?

D8 Dare un esempio per ciascuna delle quattro tipologie di aggregazione: *possiede in modo esclusivo*, *possiede*, *ha* e *raggruppa*.

D9 Si consideri il modello delle classi nella Figura 4.6 (Esempio 4.9). Come cambierebbe la semantica del modello se la classe `InformazioneAccademica` fosse rappresentata con una classe di associazione connessa all'associazione `sceglie`?

D10 L'ereditarietà senza il polimorfismo è possibile ma poco utile. Perché? Fornire un esempio.

D11 Quali modelli UML sono utili per le specifiche del comportamento? Spiegarne le caratteristiche individuali e come sono interrelati tra loro per definire il comportamento del sistema.

D12 Spiegare come i casi d'uso si differenziano dalle funzioni o transazioni di business.

D13 Dare degli esempi di relazione <<include>> e <<extend>> tra i casi d'uso. Qual è la differenza principale?

D14 Dare un esempio di una classe con alcuni attributi. Discutere quali attributi possono attivare una transizione di stato e quali sono indifferenti alle modifiche di stato.

Esercizi

Requisiti Aggiuntivi (Immatricolazioni universitarie)

Si considerino i seguenti requisiti aggiuntivi per il sistema Immatricolazioni universitarie:

1. L'Università è organizzata in facoltà. Le facoltà si dividono in dipartimenti. Gli accademici lavorano per un singolo dipartimento.
2. La maggior parte dei curriculum sono gestiti da una facoltà ma alcuni sono gestiti congiuntamente da due o più facoltà.
3. I nuovi studenti ricevono l'accettazione e le istruzioni d'immatricolazione per posta. Gli studenti in anni successivi al primo che possono essere immatricolati all'anno successivo ricevono (per posta) le istruzioni d'immatricolazione insieme alla notifica dei loro risultati d'esame.
4. Le istruzioni d'immatricolazione includono l'orario dei corsi offerti.
5. Durante l'immatricolazione, gli studenti potrebbero consultare un consigliere accademico nella facoltà in cui sono registrati, per la formulazione del loro piano di studio.
6. Gli studenti non sono vincolati a seguire solo corsi offerti dalla facoltà in cui sono iscritti e potrebbero, in ogni momento, modificare la facoltà d'iscrizione (compilando un modulo di modifica piano di studio disponibile presso la segreteria studenti).
7. Per scegliere alcuni corsi, uno studente deve prima superare i corsi di prerequisito con un giudizio minimo (cioè superare tali corsi con la semplice sufficienza potrebbe non bastare). Uno studente che non ha superato in modo soddisfacente un corso potrebbe tentare di rifare ancora il medesimo corso. Per essere immatricolati in un corso per la terza volta oppure con requisiti inferiori a quelli richiesti è necessaria un'approvazione speciale da parte del delegato del Preside della facoltà.
8. Gli studenti sono classificati come a tempo parziale se, nell'anno in corso, sono immatricolati in corsi per un ammontare totale di crediti non superiore a 18 punti.
9. Per essere immatricolati in un piano di studi che preveda più di 14 crediti totali per un dato semestre si deve ottenere un permesso speciale.

Requisiti Aggiuntivi (Video Store)

Si considerino i seguenti requisiti aggiuntivi per il Video Store:

1. Cassette e dischi riportati in ritardo implicano un pagamento uguale a un periodo d'affitto supplementare. Ogni supporto di film ha un identificatore unico.
2. I film sono ordinati dai fornitori i quali sono generalmente capaci di fornire cassette e/o dischi in una settimana. Tipicamente, diversi film sono ordinati con un singolo ordine a un fornitore.
3. Si accettano prenotazioni per un film che è stato ordinato oppure le cui copie sono attualmente in affitto. Si accettano anche prenotazioni per film che non sono né disponibili né ordinati: in questo caso, si richiede al cliente un deposito pari a un periodo d'affitto.
4. I clienti possono fare molte prenotazioni ma richieste di prenotazione distinte sono preparate per i film prenotati. Una prenotazione potrebbe essere cancellata se il cliente non risponde entro una settimana dalla data in cui gli è stata comunicata la disponibilità del film richiesto. Se è stato versato un deposito, allora è accreditato sul conto del cliente.
5. Il database memorizza le normali informazioni su fornitori e clienti, come indirizzi, numeri di telefono, etc. Ogni ordine verso un fornitore identifica i film ordinati, i formati (cassette/dischi) richiesti, le quantità, la data di consegna prevista, il prezzo d'acquisto, gli sconti applicabili, etc.
6. Quando una cassetta è riportata da un cliente o consegnata da un fornitore, si soddisfano anzitutto le prenotazioni. Questo implica l'invio di una comunicazione al cliente che ha prenotato. Al fine di assicurare che le prenotazioni siano gestite correttamente, sia l'avviso al cliente sia il susseguente affitto sono messi in relazione alla prenotazione.
7. Un cliente può affittare diverse cassette e/o dischi ma ognuno di questi deve costituire una registrazione separata. Per ogni affitto, tale registrazione comprende i tempi e le date di uscita, restituzione prevista, restituzione effettiva. Alla restituzione, la registrazione di affitto è aggiornata per indicare che il video è stato riportato e il pagamento finale (o rimborso) è stato eseguito. Anche l'impiegato che autorizza l'affitto è registrato. I dettagli sugli affitti dei clienti sono conservati per un anno al fine di permettere valutazioni statistiche sui clienti stessi.
8. Tutte le transazioni sono eseguite in contanti, con bonifico o carte di credito. I clienti devono versare il pagamento quando cassette e/o dischi sono ritirati.
9. Quando una cassetta e/o disco è restituito in ritardo (o non può essere restituito per una qualsiasi ragione), si addebita una certa somma sul conto del cliente oppure il cliente corrisponde direttamente tale somma.
10. Se una cassetta o disco deve essere restituito da più di due giorni, un'Informativa è inviata al cliente. Una volta che due Informative su una singola cassetta/disco sono state inviate, il cliente è annotato come inadempiente e il prossimo affitto deve essere gestito di conseguenza.

E1 Immatricolazioni universitarie – riferendosi ai requisiti aggiuntivi precedenti e all'Esempio 4.1 (Sezione 4.2.1.1.7), quali nuove classi possono essere derivate dai requisiti estesi?

E2 Video Store – riferendosi ai requisiti aggiuntivi precedenti e all'Esempio 4.2 (Sezione 4.2.1.1.7), quali nuove classi possono essere derivate dai requisiti estesi?

E3 Immatricolazioni universitarie – riferendosi ai requisiti aggiuntivi precedenti, all'Esercizio E1 e all'Esempio 4.9 (Sezione 4.2.3.3), estendere il modello delle classi di Figura 4.6 (Esempio 4.9) per inglobare i requisiti estesi.

E4 Video Store – riferendosi ai requisiti aggiuntivi precedenti, all'Esercizio E2 e all'Esempio 4.10 (Sezione 4.2.4.3), estendere il modello delle classi di Figura 4.7 (Esempio 4.10) per inglobare i requisiti estesi.

E5 Immatricolazioni universitarie – riferendosi ai requisiti aggiuntivi precedenti e all'Esempio 4.12 (Sezione 4.3.1.3), estendere il modello dei casi d'uso di Figura 4.9 (Esempio 4.12) per inglobare i requisiti estesi.

E6 Video Store – riferendosi ai requisiti aggiuntivi precedenti e all'Esempio 4.14 (Sezione 4.3.1.3), studiare la specifica testuale per il caso d'uso `Affittare Video` nella Tabella 4.4 (Esempio 4.14). Sviluppare dei diagrammi distinti per rappresentare i casi d'uso alternativi, ignorando l'ultimo paragrafo nella sezione "Flusso principale" della tabella (questo paragrafo si riferisce piuttosto al caso d'uso Gestione Cliente).

E7 Immatricolazioni universitarie – riferendosi ai requisiti aggiuntivi precedenti e all'Esempio 4.17 (Sezione 4.3.3.3), estendere il diagramma di sequenza di Figura 4.14 (Esempio 4.17) per inglobare il controllo dei prerequisiti (uno studente sarà aggiunto a un corso se e solo se ha superato i corsi di prerequisito).

E8 Immatricolazioni universitarie – riferendosi all'Esempio 4.18 (Sezione 4.3.4.3) e alla soluzione dell'Esercizio E7, usare il diagramma di sequenza esteso per aggiungere operazioni alle classi rilevanti, includendo le due classi in Figura 4.15 (Esempio 4.18).

E9 CMS – riferendosi all'Esempio 4.8 (Sezione 4.2.2.3), sviluppare un diagramma statechart per la classe `Evento`.

E10 Telemarketing – riferendosi all'Esempio 4.7 (Sezione 4.2.1.2.3), aggiungere le classi d'associazione non riportate in Figura 4.4 e determinare le molteplicità di ciascuna associazione.

E11 CMS – riferendosi all'Esempio 4.8 (Sezione 4.2.2.3), considerare le classi `Organizzazione`, `Contatto`, `IndirizzoPostale`, `IndirizzoCorriere`. Come estensione al modello in Figura 4.5, approfondire la specifica introducendo una struttura gerarchica di generalizzazione per le organizzazioni.

Capitolo 5

Approfondimento dell'analisi

Il precedente capitolo ha dipinto un quadro roseo della modellazione visuale object-oriented con UML: gli esempi non hanno richiesto un grande sforzo, il linguaggio visuale di rappresentazione è semplice e attraente all'uso, le dipendenze tra modelli sono evidenti, le tecniche di modellazione sono state usate senza porsi problemi e riservando poca attenzione alle soluzioni alternative. Nella realtà, lo sviluppo del software è molto meno agevole e non ci sono soluzioni semplici a problemi complessi. Gli oggetti forniscono la tecnologia attuale per risolvere tali problemi ma devono essere usati al giusto livello di complessità.

Questo capitolo vuole fornire una valutazione critica della tecnologia a oggetti e della sua idoneità alla soluzione di problemi complessi. Saranno introdotti concetti più specializzati nella specifica delle classi quali: stratificazione delle classi, ereditarietà e delegazione. Si confronteranno soluzioni, si formuleranno opinioni e saranno suggerite soluzioni alternative. Dato il carattere sostanzialmente tecnico, molti argomenti discussi in questo capitolo si estendono direttamente al progetto di sistema. Ciò è coerente con l'intrinseca ubiquità di UML e con la natura iterativa e incrementale del processo di sviluppo object-oriented del software.

5.1 La modellazione delle classi

I concetti di rappresentazione fino a qui discussi sono sufficienti alla produzione di modelli d'analisi completi ma a un livello d'astrazione che non esaurisce tutti i possibili dettagli che possono essere evidenziati durante la fase di analisi. UML comprende notazioni per svariati concetti aggiuntivi che non sono ancora stati trattati o a cui, precedentemente, si è solo fatto cenno.

Questi concetti aggiuntivi includono: stereotipi, vincoli, informazione derivata, visibilità, associazioni qualificate, classi d'associazione, classi parametrizzate. Sono comunque concetti opzionali e molti modelli, pur non usandoli, potrebbero essere ugualmente accettabili; se usati, devono essere applicati con attenzione e precisione in modo che qualunque futuro lettore del modello possa interpretare senza ambiguità le intenzioni dell'analista.

5.1.1 Stereotipi

Uno *stereotipo* estende un elemento UML esistente: non è un nuovo elemento ma una modifica (e un arricchimento) alla semantica dell'elemento di base. Questa tecnica non cambia la struttura di UML, ma ne permette l'estensione e la personalizzazione in diversi modi.

Tipicamente, gli stereotipi sono *etichettati* nei modelli con un nome tra virgolette come `<<global>>`, `<<PK>>`, `<<include>>`; è altresì possibile introdurne una presentazione grafica attraverso *icone*.

Alcuni stereotipi (i più comuni) sono predefiniti in UML, e possono essere messi a disposizione da strumenti CASE sotto forma di icona (la maggior parte degli strumenti CASE fornisce all'analista la possibilità di creare nuove icone). La Figura 5.1 è un esempio di classe stereotipata con un'icona, con un'etichetta e non stereotipata.

La restrizione secondo cui uno stereotipo estende la semantica ma non la struttura di UML è poco rilevante. Alla base di ogni sistema object-oriented sta il fatto che *ogni cosa* al suo interno *è un oggetto*: una classe è un oggetto, un attributo è un oggetto, un metodo è un oggetto, etc.; "stereotipando" una classe si crea un nuovo elemento di rappresentazione che introduce una nuova categoria di oggetti. Ad esempio, nelle Sezioni 7.6.1 e 9.2.1 sarà mostrato come gli stereotipi possono essere usati per creare nuove tecniche di rappresentazione (come diagrammi di navigazione nelle finestre e nei programmi). Se usati in quest'ottica, gli stereotipi estendono UML per una particolare finalità che, spesso, ha a che vedere con la modellazione di progetto piuttosto che con quella d'analisi. I modelli di progetto devono adeguarsi alla piattaforma d'implementazione, quindi è ragionevole che trattino con elementi adatti a rappresentare correttamente la piattaforma stessa.

Un insieme di stereotipi aventi una certa finalità comune e riguardanti certi aspetti di progetto è chiamato *profilo*. I futuri standard UML potrebbero comprendere profili per comuni obiettivi di progetto come database o architetture CORBA.

5.1.2 Vincoli

Gli stereotipi sono spesso confusi con i vincoli, infatti la distinzione fra i due concetti è, a volte, non molto netta. Uno *stereotipo* può essere usato per introdurre un nuovo *vincolo* al modello, qualcosa di significativo per l'analista e per il quale UML non offre alcun supporto.

Qualsiasi elemento di rappresentazione può avere un vincolo associato o può essere stereotipato. Solo vincoli semplici sono mostrati in un diagramma, attraverso un testo tra parentesi graffe (o con una *nota*, si veda la Sezione 5.1.3) e sono semanticamente e graficamente legati a un elemento del modello. Vincoli più elaborati (troppo lunghi per essere mostrati nel modello visuale) sono memorizzati nel repository dello strumento CASE utilizzato, in genere sotto forma di documento testuale.

FIGURA 5.1 Icone ed etichette per gli stereotipi.

Esempio 5.1 (Telemarketing)

Riferendosi all'Esempio 4.7 (Sezione 4.2.1.2.3), si consideri la parte del Requisito 2 che recita: "Tutti i biglietti sono numerati. I numeri sono unici per i biglietti di una campagna".

Nella soluzione all'Esempio 4.7 (Figura 4.4) non era possibile rappresentare il precedente vincolo (a meno d'includere il `codice_campagna` (con il `numero_biglietto`) in `BigliettoCampagna` come parte di una chiave primaria composta).

L'obiettivo è rappresentare il vincolo nel diagramma delle classi.

La Figura 5.2 rappresenta una semplice estensione del modello delle classi che mostra tale vincolo *legato* alla classe `BigliettoCampagna`. In qualsiasi rapporto prodotto dallo strumento CASE, il vincolo sarà elencato con le altre proprietà della classe.

Esempio 5.2 (CMS)

Riferendosi all'Esempio 4.8 (Sezione 4.2.2.3), si ipotizzi che sia stato trovato un nuovo requisito, riguardante il fatto che gli impiegati non possono programmare eventi per se stessi (l'impiegato che ha creato un evento deve essere diverso da quello che lo eseguirà).

Estendere il modello delle classi di Figura 4.5 per includere il nuovo requisito.

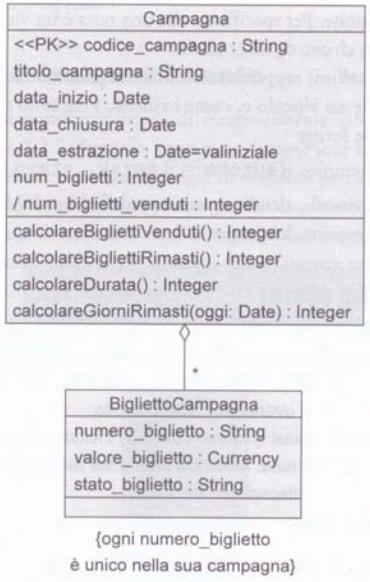


FIGURA 5.2 Vincolo per una classe (Telemarketing).

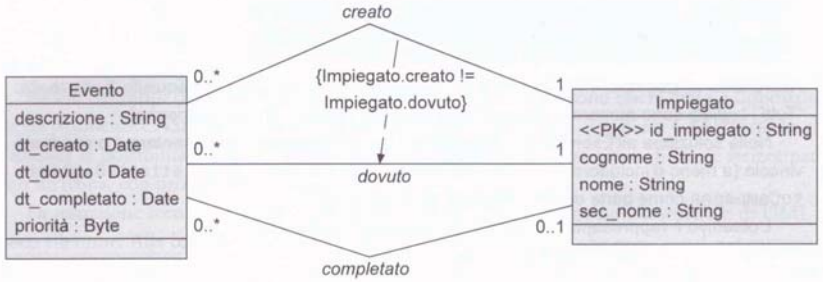


FIGURA 5.3 Vincolo sulle associazioni (CMS).

La soluzione a quest'esempio (Figura 5.3) è un vincolo sulle linee d'associazione. Ciò è mostrato come una freccia tratteggiata di dipendenza dall'associazione `creato` all'associazione `dovuto`. Il vincolo è *legato* alla freccia.

5.1.3 Note e tag

Quando il testo del vincolo è lungo o il vincolo deve essere espresso coinvolgendo tre o più simboli grafici, può essere usato il concetto UML di *nota*. Graficamente, "una nota è un rettangolo con l'angolo in alto a destra ripiegato" (Rumbaugh *et al.*, 1999, p. 359).

Il simbolo di nota può contenere del testo che esprime un vincolo ma, in generale, può contenere una qualsiasi informazione. Per specificare che una nota è un vincolo, questa dovrebbe essere stereotipata con la parola chiave `constraint`.

Come le note, i *tag* (cartellini) rappresentano dell'informazione testuale arbitraria. Tale informazione potrebbe esprimere un vincolo e, come i vincoli, i tag sono scritti all'interno di parentesi graffe e hanno la seguente forma

`tag = valore,` ad esempio: `{autore = Leszek, stato = 2ª iterazione}.`

Come per gli stereotipi e i vincoli, alcuni tag sono predefiniti in UML. Un uso tipico dei tag riguarda le informazioni di gestione del progetto.

Esempio 5.3 (CMS)

Con riferimento all'Esempio 4.8 (Sezione 4.2.2.3) e al precedente Esempio 5.2, si ipotizzi che sia stato trovato un nuovo requisito: l'impiegato che ha creato un compito deve anche creare, nella stessa transazione, il primo evento per quel compito.

Estendere il modello delle classi (Figure 4.5 e 5.3) includendo il nuovo requisito mediante un vincolo rappresentato da una nota. Mostrare anche sul diagramma che l'autore è Leszek e il diagramma è alla sua seconda iterazione.

La Figura 5.4 mostra il modello esteso. L'informazione progettuale è rappresentata da un tag posto nell'angolo in alto a sinistra. Una nota è stata usata per esprimere il vincolo su tre associazioni. La nota è stereotipata con la parola chiave `constraint` per specificare che la nota è un vincolo.

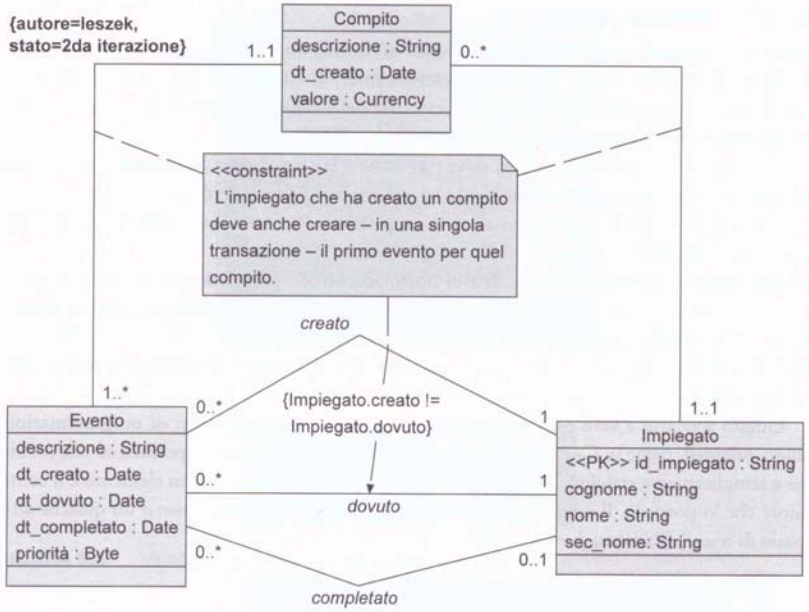


FIGURA 5.4 Nota vincolo e tag (CMS).

5.1.4 Visibilità e incapsulamento

Il concetto di *visibilità* e la nozione collegata di *incapsulamento* sono stati inizialmente trattati nelle Sezioni 2.1.2.1.2 e 2.1.2.2.2. Tuttavia, sono state distinte solo la visibilità pubblica e quella privata di attributi e operazioni. La terza tipologia di visibilità predefinita in UML, la visibilità *protetta*, non è stata discussa, così come non sono state discusse le sottili influenze tra visibilità ed *ereditarietà*. E neppure è stata discussa una tecnica comune e conveniente (ma alcuni sostengono indesiderabile) per "far breccia" nell'incapsulamento di un oggetto: un'operazione *amica*.

La notazione UML standard per la visibilità usa i segni: + (per pubblica), # (per protetta) e – (per privata), posti prima del nome di una proprietà. Spesso, gli strumenti CASE cambiano questa notazione piuttosto monotona. La Figura 5.5 è un esempio di notazione grafica più attraente.

FIGURA 5.5 Notazione della visibilità in uno strumento CASE.

5.1.4.1 Visibilità protetta

Avere delle *proprietà private* (attributi e operazioni) di una classe base accessibili esclusivamente da oggetti della classe stessa non è sempre conveniente. Ad esempio, ci sono molte situazioni in cui gli oggetti di una classe derivata (una sottoclasse della classe base) dovrebbero poter accedere alle proprietà private della classe base.

Si consideri una gerarchia di classi dove `Persona` è la classe base e `Impiegato` è una classe derivata. Se `Joe` è un oggetto di `Impiegato`, allora, per definizione di generalizzazione, `Joe` deve aver accesso alle (o almeno ad alcune) proprietà di `Persona` (ad esempio, all'attributo `data_di_nascita`).

Per permettere a una classe derivata di aver libero accesso alle proprietà della sua classe base tali proprietà devono essere definite nella classe base come *protette* (come discusso in Sezione 2.1.2.1.2, la visibilità si applica tra classi, all'interno della stessa classe è tutto visibile, per cui se `Betty` è un altro oggetto di `Impiegato`, questo può aver accesso a qualsiasi proprietà di `Joe`, sia essa pubblica, privata o protetta).

Questo scenario è vero per la maggioranza dei più comuni ambienti di programmazione object-oriented, come in C++ e Java (Fowler e Scott, 2000). UML non è specifico su tale problema e semplicemente stabilisce che: "la visibilità è parte della relazione tra un elemento e il contenitore che lo possiede. Il contenitore potrebbe essere un package, una classe o un qualche altro spazio di nomi" (Rumbaugh *et al.*, 1999, p. 497).

Esempio 5.4 (Telemarketing)

Riferendosi alla specifica del caso Telemarketing (Sezione 2.3.4) e all'Esempio 4.7 (Sezione 4.2.1.2.3), si consideri la seguente osservazione, contenuta nella specifica: "gli schemi prevedono speciali campagne con bonus per ricompensare i sostenitori per i loro acquisti, per attrarre nuovi sostenitori etc." (quest'osservazione non è stata ancora rappresentata).

Si supponga che delle campagne di bonus comprendano "blocchi di biglietti". Se un sostenitore compra l'intero blocco, gli è regalato un biglietto extra relativo a una campagna associata.

Il compito è:

- aggiornare il modello delle classi onde inglobare la classe `CampagnaBonus`;
- modificare la visibilità degli attributi di `Campagna` (Figura 5.2) in modo che possano essere visibili dalla classe `CampagnaBonus` ad eccezione di `data_inizio`. Rendere `codice_campagna` e `titolo_campagna` visibili alle altre classi nel modello;
- aggiungere le seguenti operazioni alla classe `Campagna`: `calcolareBigliettiVenduti`, calcolareBigliettiRimasti, calcolareDurata, calcolareGiorniRimasti.
- osservare che le altre classi, distinte da `Campagna`, non hanno interesse per `calcolaBigliettiVenduti` ma solo per l'operazione `calcolareBigliettiRimasti`. Anche `calcolareDurata` è usata solo dall'operazione `Campagna.calcolareGiorniRimasti`.
- rappresentare nella classe `CampagnaBonus` l'attributo `dimensione_blocco_biglietti` e fornire un'operazione di accesso a questo dato chiamandola `bloccoDimensione`.

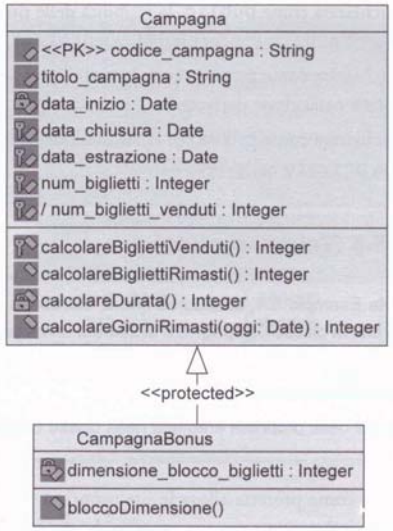


FIGURA 5.6 Visibilità delle proprietà di una classe (Telemarketing).

La Figura 5.6 mostra un modello delle classi che corrisponde alla precedente descrizione. Lo stereotipo `protected` sulla linea di generalizzazione sarà spiegato nella prossima sezione. In `Campagna`, l'operazione `calcolareDurata` è privata, l'operazione `calcolareBigliettiVenduti` è protetta. Le due rimanenti operazioni in `Campagna` sono pubbliche. L'operazione `bloccoDimensione` è specificata in `CampagnaBonus` ed è pubblica.

5.1.4.2 Visibilità delle proprietà ereditate nelle classi

Come riconosciuto da UML, la *visibilità* si applica a oggetti a vari livelli. La visibilità si riferisce anzitutto a *oggetti primitivi*, attributi e operazioni. Ma la visibilità può anche essere specificata rispetto ad altri "contenitori" e questo crea un groviglio di regole che si sovrappongono.

Si consideri ad esempio una comune situazione in cui la visibilità è definita in una gerarchia d'ereditarietà a livello della classe base *e anche* a livello di proprietà della classe base. Si ipotizzi che una classe B sia una sottoclasse della classe A. La classe A contiene un insieme di attributi e operazioni, alcuni pubblici, altri privati e altri protetti. La domanda è: "Qual è la visibilità delle proprietà ereditate nella classe B?".

Una risposta a tale domanda dipende dal livello di visibilità assegnato alla classe base A quando quest'ultima è dichiarata nella classe derivata B. La classe base potrebbe essere stata definita pubblica (`class B: public A`), o protetta (`class B: protected A`), o ancora privata (`class B: private A`).

Una soluzione tipica al precedente scenario è la seguente (Horton, 1997):

- le proprietà `private` (attributi e proprietà) della classe base A non sono visibili agli oggetti di classe B, indipendentemente da come la classe A sia stata dichiarata in B;

- se la classe base A è dichiarata come `public`, la visibilità delle proprietà ereditate non cambia nella classe derivata B (`public` è ancora pubblica e `protected` è ancora protetta);
- se la classe base A è dichiarata come `protected`, la visibilità delle proprietà `public` ereditate muta in `protected` nella classe derivata B;
- se la classe base A è dichiarata come `private`, la visibilità delle proprietà ereditate `public` e `protected` muta in `private` nella classe derivata B.

Esempio 5.5 (Telemarketing)

Riferendosi al precedente Esempio 5.4, si assuma che lo stereotipo `protected` sull'arco di generalizzazione indichi che la classe `Campagna` è dichiarata come protetta nella classe `CampagnaBonus`:

class CampagnaBonus: protected Campagna

Spiegare il livello di visibilità delle proprietà ereditata nella classe `CampagnaBonus`.

Se la classe base è dichiarata come protetta allora le sue proprietà pubbliche diventano protette nella classe derivata. Le proprietà protette restano protette. Le proprietà private non sono, ovviamente, accessibili alla classe derivata.

Le Figure 5.7 e 5.8 rappresentano le finestre d'interazione con il repository di uno strumento CASE che mostrano, rispettivamente, gli attributi e le operazioni di `CampagnaBonus` dopo che l'ereditarietà ha prodotto il suo effetto. Per essere più precisi, le classi in Figura 5.7 e 5.8 dovrebbero anche contenere l'attributo `data_inizio` e l'operazione `calcolareDurata()` ereditati da `Campagna` (si noti che le proprietà private della classe base rimangono private e restano inaccessibili alle operazioni della classe derivata; tuttavia, la definizione di una proprietà privata è ereditata nel modo consueto).

5.1.4.3 Classi "amiche"

Potrebbero presentarsi dei casi in cui una funzione (o anche un'intera classe) deve poter accedere direttamente alle proprietà di un'altra classe, indipendentemente dai livelli di visibilità di tali proprietà. Questa situazione sorge quando le due classi sono "intrecciate" e le operazioni in una classe dipendono dalle proprietà dell'altra. Un esempio tipico potrebbe essere quello delle due classi `Libro` e `Ripiano` con un'operazione in `Libro` chiamata `mettereLibroSulRipiano`. Una soluzione per questi problemi è dichiarare l'operazione `mettereLibroSulRipiano` un'*amica* (*friend*) all'interno della classe `Ripiano` con una dichiarazione del tipo:

```
friend void Libro:: mettereLibroSulRipiano ()
```

Un'amica può essere un'altra classe o un'operazione di un'altra classe. L'amicizia non è reciproca, anche se una classe rende amica un'altra classe, il viceversa potrebbe non essere vero.

Un'amica (operazione o classe) è dichiarata *internamente* alla classe che assegna l'amicizia. Tuttavia, un'operazione amica non è una proprietà della classe, perciò non vi si può applicare alcuna visibilità. Ciò significa anche che nella definizione di un'amica non è possibile far riferimento agli attributi della classe in cui è stata dichiarata usando solo i loro nomi: ciascun attributo deve essere qualificato dal nome della classe (come se l'amica fosse una normale operazione esterna).

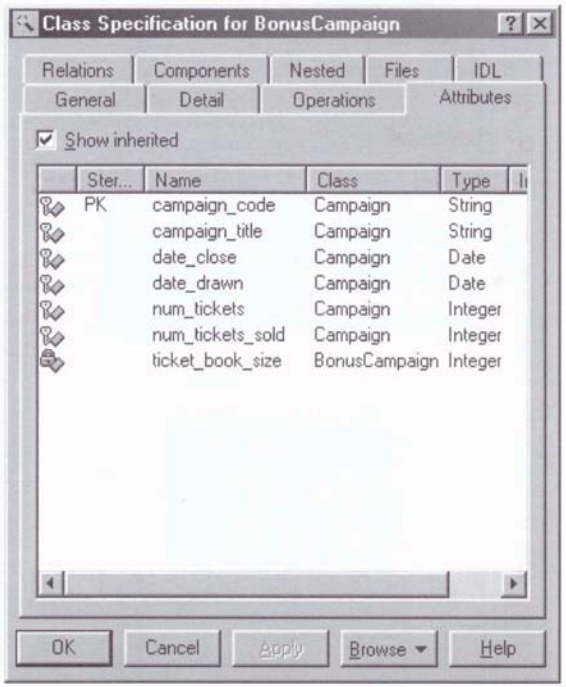


FIGURA 5.7 Attributi ereditati in `CampagnaBonus` (Telemarketing).

In UML, l'amicizia è mostrata come una *relazione di dipendenza*, linea tratteggiata da una classe o operazione amica alla classe che assegna l'amicizia. Lo stereotipo `friend` è legato all'arco di dipendenza. Per ammissione, la notazione UML né riconosce né supporta completamente la semantica dell'amicizia.

Esempio 5.6 (Telemarketing)

Riferendosi all'Esempio 4.7 (Sezione 4.2.1.2.3), si consideri la relazione tra le classi `Campagna` e `ChiamataProgrammata`.

Gli oggetti della classe `ChiamataProgrammata` sono molto attivi e necessitano di speciali privilegi per l'esecuzione delle loro operazioni. In particolare, contengono un'operazione chiamata `ottenereBigliettiRimasti` che stabilisce se c'è qualche biglietto rimanente per soddisfare l'ordine di un sostenitore. È importante che questa operazione abbia accesso diretto alle proprietà di `Campagna` (come `num_biglietti` e `num_biglietti_venduti`).

Dichiarare l'operazione `ottenereBigliettiRimasti` e renderla un'amica di `Campagna`.

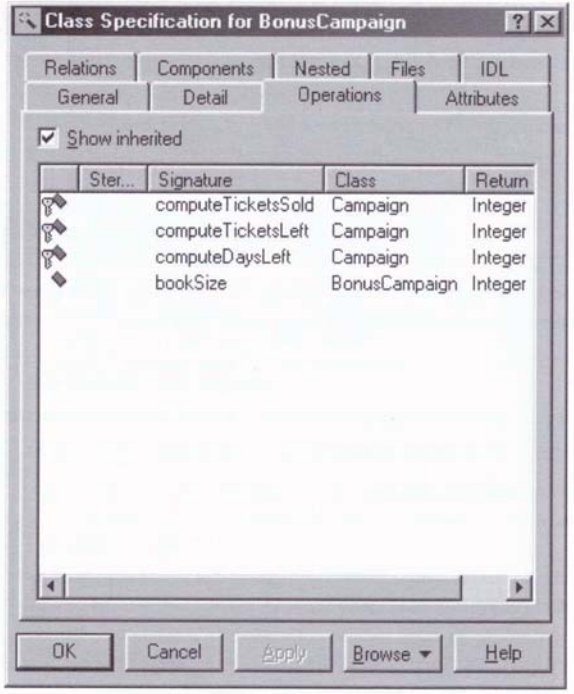


FIGURA 5.8 Operazioni ereditate in CampagnaBonus (Telemarketing).

La soluzione UML all'esempio è mostrato nella Figura 5.9. L'operazione `ottenereBigliettiRimasti` è mostrata all'interno di `ChiamataProgrammata`. Si ricordi che in un linguaggio di programmazione usato per l'implementazione, `ottenereBigliettiRimasti` sarà dichiarata amica all'interno di `Campagna` (poiché `Campagna` decide chi è sua amica), e quindi completamente definita in `ChiamataProgrammata`.

La *relazione di dipendenza* rappresenta il fatto che `Campagna` assegna qualche stato d'amicizia a `ChiamataProgrammata`. La notazione non specifica se l'amicizia sia per l'intera classe oppure per alcune delle sue operazioni (e per quali).

5.1.5 Informazione derivata

L'*informazione derivata* è una tipologia di vincolo che si applica (con maggiore frequenza) a un attributo o a un'associazione. Strettamente parlando, l'informazione derivata è ridondante, poiché può essere calcolata se necessario a partire dall'informazione esistente.

Sebbene l'informazione derivata non arricchisca la semantica di un *modello d'analisi*, lo rende più leggibile (a causa del fatto che ciò che può essere calcolato è dichiarato nel modello stesso). La

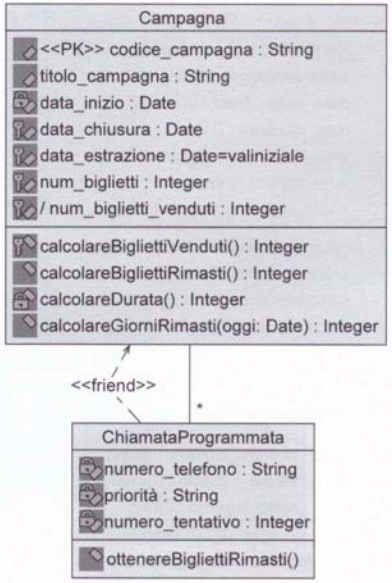


FIGURA 5.9 Amica (Telemarketing).

decisione di mostrare l'informazione derivata nel modello di analisi è piuttosto arbitraria, può essere ammessa purché sia adottata coerentemente per l'intero modello.

La conoscenza d'informazione derivata è più importante nei *modelli di progetto*, dove bisogna tenere conto dell'ottimizzazione dell'accesso all'informazione. Nei modelli di progetto è possibile decidere se l'informazione derivata debba essere memorizzata (dopo essere stata calcolata) o essere calcolata dinamicamente ogniqualvolta sia necessaria. Questa non è una nuova caratteristica: in uno storico modello per database, il modello reticolare, essa era nota come dati *attuali* (cioè memorizzati) e *virtuali*.

La notazione UML per l'informazione derivata è uno "slash" (/) posto prima del nome dell'attributo o associazione derivati.

5.1.5.1 Attributi derivati

Sebbene senza spiegazione, gli attributi derivati sono già stati usati in precedenza in diagrammi d'esempio. L'attributo `num_biglietti_venduti` nella Figura 5.2 è un attributo derivato il cui valore è calcolato per mezzo dell'operazione `calcolareBigliettiVenduti`. L'operazione, partendo da una campagna, segue l'aggregazione verso i biglietti e controlla ciascuno `stato_biglietto`. Se `stato_biglietto` è "venduto" allora è aggiunto al contatore di biglietti venduti. Quando tutti i biglietti sono stati trattati, il valore attuale di `num_biglietti_venduti` è determinato. Una versione estesa di Figura 5.2 è mostrata in Figura 5.10.

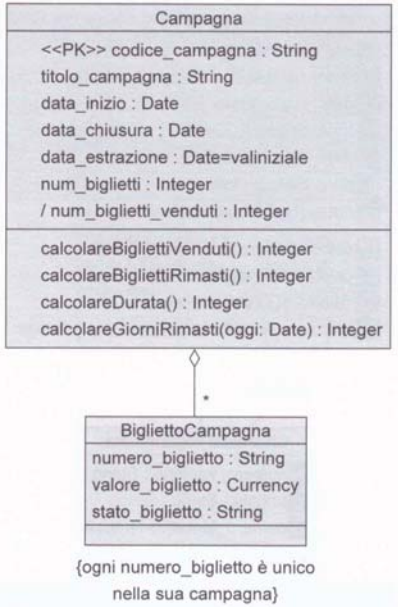


FIGURA 5.10 Attributo derivato (Telemarketing).

5.1.5.2 *Associazioni derivate*

Le associazioni derivate costituiscono un argomento più controverso. In una tipica situazione, un'associazione derivata si stabilisce fra tre classi connesse da tre associazioni in un ciclo. In tale situazione potrebbe darsi che una delle tre associazioni possa essere derivata dalle altre due (situazione conosciuta come *commutatività del ciclo*).

Esempio 5.7 (Acquisti OnLine)

Riferendosi al caso degli Acquisti OnLine presentato nel Capitolo 2 e al diagramma delle classi nella Figura 2.32 (Sezione 2.2.4.6), si consideri la parte del modello con le classi `Cliente`, `Ordine` e `Fattura`. È possibile introdurre un'associazione derivata nel modello? Se ciò è possibile, aggiungere una tale associazione al diagramma.

Effettivamente è possibile aggiungere un'associazione derivata tra le classi `Cliente` e `Fattura`, chiamata `/CliFatt` nella Figura 5.11 di seguito. L'associazione è derivata perché un `Cliente` può ottenere le sue fatture passando per i suoi ordini e, viceversa, è possibile ricavare il `Cliente` di una `Fattura` cercando il solo ordine corrispondente e quindi risalendo al singolo `Cliente` (coerentemente con le molteplicità delle associazioni tra `Cliente`, `Ordine` e `Fattura`).

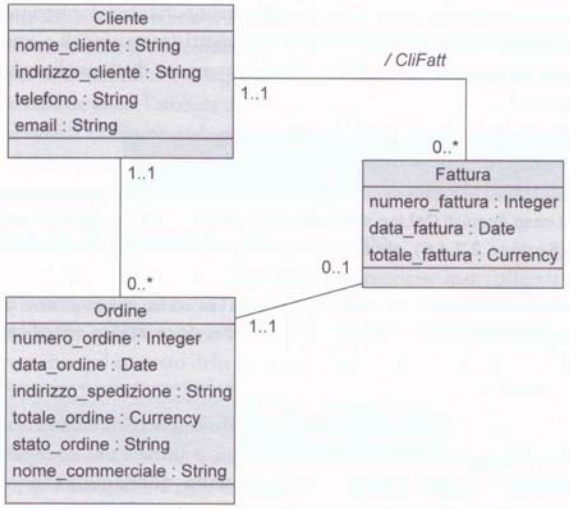


FIGURA 5.11 Associazione derivata (Telemarketing).

5.1.6 Associazioni qualificate

Il concetto di *associazione qualificata* è piuttosto controverso. Alcuni modellisti l'apprezzano, altri lo evitano poiché è possibile comunque costruire modelli delle classi completi e sufficientemente espressivi senza di esso.

Un'associazione qualificata è un'associazione binaria che su un lato ha un *qualificatore* (le associazioni possono essere qualificate su entrambi i lati ma ciò è raro). Il qualificatore contiene uno o più attributi che servono come chiave per attraversare l'associazione, dalla *classe qualificata* alla *classe obiettivo* (posta all'altro capo dell'associazione) usando il qualificatore stesso.

Ad esempio, un'associazione tra `Volo` e `Passeggero` è, in generale, molti-a-molti. Tuttavia, quando la classe `Volo` è qualificata dagli attributi `numero_posto` e `partenza`, fra tutti i passeggeri di un certo volo è selezionato quello che a una certa data di partenza occupa un certo posto; per effetto della chiave composta (`numero_volo + numero_posto + partenza`) introdotta dal qualificatore (Figura 5.12) la molteplicità è quindi ridotta, da (0..*) a (0..1). Si osservi che non necessariamente la qualificazione individua un solo oggetto.

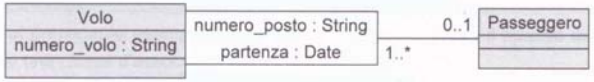


FIGURA 5.12 Associazione qualificata.

L'unicità introdotta da un qualificatore è spesso un'importante informazione semantica che non può essere rappresentata efficacemente con altri mezzi (come vincoli o attributi aggiuntivi nella classe obiettivo). Inoltre, non è generalmente conveniente duplicare l'attributo qualificante nella classe obiettivo.

Esempio 5.8 (Acquisti OnLine)

Riferendosi al caso Acquisti OnLine sviluppato nel Capitolo 2 e al diagramma delle classi nella Figura 2.32 (Sezione 2.2.4.6), si consideri una parte del modello con le classi `Ordine` e `Computer`.

Modificare l'associazione tra `Ordine` e `Computer` in un'associazione qualificata per rappresentare esplicitamente il vincolo "su un ordine deve essere specificato un singolo computer".

Si ricordi che un computer è ordinato dopo essere stato configurato da un cliente. Un numero univoco può identificare ciascuna di tali configurazioni, e anche a una configurazione standard può essere assegnato un numero univoco. Quindi, `numero_configurazione` può servire come qualificatore per `Ordine`. La chiave composta (`numero_ordine + numero_configurazione`) è legata a un singolo oggetto di `Computer` (Figura 5.13).

5.1.7 Classi d'associazione e classi materializzate

Nella Sezione 2.1.3.4 è stata spiegata ed esemplificata una *classe d'associazione*, un'associazione che è anche una classe. Una classe d'associazione è usata tipicamente se c'è un'associazione molti-a-molti tra due classi e ciascuna istanza dell'associazione (un *legame*) ha propri valori d'attributo che saranno memorizzati nella classe d'associazione stessa.

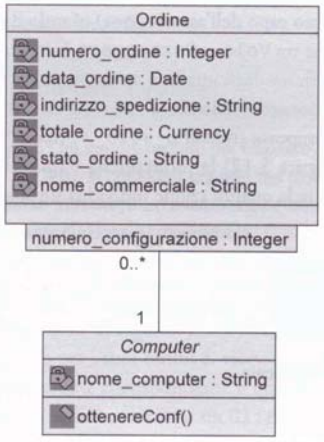


FIGURA 5.13 Associazione qualificata (Acquisti OnLine).

Semplice come sembra, il concetto d'associazione porta con sé un vincolo delicato. Si consideri una classe d'associazione *C* tra le classi *A* e *B*. Il vincolo è che esista una e una sola istanza di *C* per ogni coppia di istanze di *A* e *B* legate nell'associazione (cioè per ogni legame). Se un tale vincolo non è accettabile allora l'analista deve *materializzare* l'associazione sostituendo la classe *C* con una classe ordinaria *D* (Rumbaugh *et al.*, 1999). La *classe materializzata D* avrà due associazioni verso *A* e *B* rispettivamente. La classe *D* è indipendente dalle classi *A* e *B*, ogni sua istanza ha una propria identità, perciò può essere, se necessario, legata a diverse istanze della classe *A* o *B*.

5.1.7.1 Esempio di un database del personale

Per spiegare la differenza tra una classe d'associazione e una classe materializzata è necessario un esempio mirato. L'esempio è in relazione alla registrazione degli stipendi (attuali e precedenti) di impiegati (spesso, la distinzione tra una classe d'associazione e una classe materializzata appare con maggiore frequenza nel contesto della rappresentazione d'informazioni temporali).

Esempio (Database Personale)

Un unico `id_imp` è assegnato a ogni impiegato in un'organizzazione. Il nome dell'impiegato consiste del cognome, nome, e iniziale secondo nome.

Ogni impiegato ha un certo livello retributivo e a ciascun livello è associato un intervallo retributivo, cioè uno stipendio minimo e uno massimo. Gli intervalli retributivi per un livello non mutano mai, così come le date iniziale e finale di ciascun livello. Se è necessario modificare lo stipendio minimo o massimo, è creato un nuovo livello retributivo.

Sono registrati i precedenti stipendi di ciascun impiegato, con le date iniziale e finale di ogni livello. È anche registrata qualsiasi variazione dello stipendio di un impiegato all'interno dello stesso livello.

5.1.7.2 Modellazione con classi d'associazione

Gli oggetti di una classe d'associazione (come gli oggetti di una qualsiasi classe ordinaria) ricevono un OID all'atto dell'istanziazione (Sezione 2.1.1.3). Indipendentemente dagli OID generati dal sistema, gli oggetti possono essere anche identificati attraverso i valori di propri attributi. Nel caso di una classe d'associazione, un oggetto può ricevere una sua identità da attributi aventi come tipo le classi coinvolte nell'associazione (Sezione 2.1.2.1.1), e non sono necessari altri attributi per contribuire all'identificazione dell'oggetto.

Esempio 5.9 (Database Personale)

Riferendosi al precedente esempio (Database Personale) costruire il relativo modello delle classi usando una classe d'associazione.

Saranno ovviamente usate due classi per memorizzare i dettagli dell'impiegato (`Impiegato`) e le informazioni sui livelli retributivi (`LivelloStipendio`). Il problema sta nella rappresentazione delle attribuzioni (storiche e attuali) degli stipendi agli impiegati.

Inizialmente, potrebbe sembrare naturale l'uso di una classe d'associazione `StoriaStipendioCA`, come mostrato in Figura 5.14. La soluzione però è lacunosa. Infatti, gli oggetti di `StoriaStipendioCA` derivano la propria identità dalla chiave composta creata dai riferimenti alle chiavi primarie delle classi `Impiegato` e `LivelloStipendio` (cioè `id_imp` e `id_livello`) e questo significa che a ogni coppia costituita da un `Impiegato` e da un `LivelloStipendio` possiamo associare al più un oggetto di `StoriaStipendioCA`, in contrasto con il requisito secondo cui "qualsiasi variazione dello stipendio di un impiegato all'interno dello stesso livello deve essere registrata". La soluzione in Figura 5.14 non può essere accettata ed è necessario un modello "migliore".

5.1.7.3 Modellazione con classi materializzate

Una classe materializzata è indipendente dalle classi associate e non impone alcuna restrizione al numero di oggetti delle classi associate legati a un oggetto della classe stessa. Inoltre, la chiave primaria di una classe materializzata non usa riferimenti alle chiavi primarie delle classi associate.

Esempio 5.10 (Database Personale)

Riferendosi al precedente esempio (Database Personale) costruire il relativo modello delle classi usando una classe materializzata.

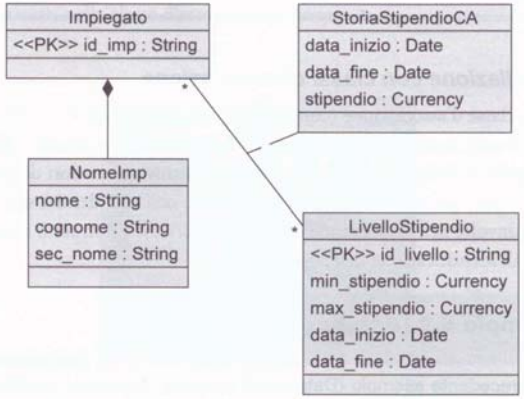


FIGURA 5.14 Uso inadeguato di una classe d'associazione (Database Personale).

La Figura 5.15 mostra un modello delle classi che usa la classe materializzata `StoriaStipendioCM`. La classe non ha un'esplicita chiave primaria, si può in ogni caso ritenere che una chiave primaria consista di `id_imp` e `num_seq`, dove l'attributo `num_seq` contiene il numero sequenziale delle variazioni di stipendio relative a un impiegato. Come specificato dalle cardinalità delle relazioni in cui è coinvolta la classe materializzata, a ogni coppia costituita da un `Impiegato` e da un `LivelloStipendio` possiamo associare un numero qualsiasi di oggetti di `StoriaStipendioCM`, perciò il modello permette di rappresentare effettivamente le variazioni di stipendio di un impiegato all'interno del medesimo livello retributivo.

Si noti che il modello nella Figura 5.15 dovrebbe probabilmente essere ulteriormente migliorato, in particolare perché l'assunzione "gli intervalli retributivi per un livello non mutano mai" potrebbe essere attenuata.

5.2 Stratificazione delle classi

Gli sviluppatori di software sanno fin troppo bene che la difficoltà di produrre un piccolo sistema non può essere neanche confrontata con quella del rilascio di una soluzione su larga scala. Un piccolo sistema a oggetti è spesso semplice da comprendere, implementare, e mettere in produzione. Una soluzione a oggetti di grandi dimensioni può essere dominata da complesse *reti* di oggetti che rispondono a eventi casuali e invocano una "ragnatela" di operazioni interrelate (metodi). Senza un chiaro progetto architetturale e processi di sviluppo rigorosi, grandi progetti software spesso sono votati al fallimento.

Un ben noto principio della psicologia cognitiva, *la regola del 7 ± 2*, stabilisce che la memoria a breve termine della mente umana può trattare *simultaneamente* fino a nove (7 + 2) "cose" (ele-

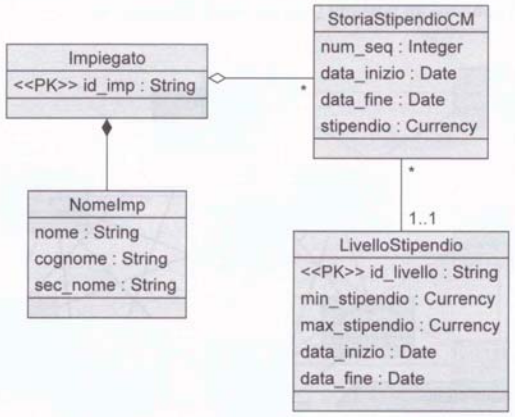


FIGURA 5.15 Migliore soluzione con una classe materializzata (Database Personale).

menti grafici, idee, concetti, etc.); il limite inferiore di cinque (7 – 2) indica che trattare meno di cinque cose costituisce un problema banale. Questo comunque non cambia il fatto che è necessario produrre grandi sistemi e che tali sistemi siano complessi.

Una componente importante di questa complessità risiede nel fatto che i modelli di rappresentazione permettono una comunicazione tra oggetti senza nessuna limitazione. Gli oggetti, nei sistemi complessi, sono legati fra loro attraverso scambi di messaggi (bidirezionali). In queste *reti*, il numero di canali di comunicazione tra gli oggetti può crescere esponenzialmente con l'introduzione di nuovi oggetti. Come osservato da Szyperski "...i riferimenti tra oggetti introducono legami che attraversano domini arbitrari d'astrazione. La corretta stratificazione delle architetture di sistema sta quindi da qualche parte tra lo stimolante e l'impossibile" (Szyperski, 1998, p. 57).

I sistemi di maggior successo sono stati organizzati in *gerarchie*, e attraverso questa tecnica le strutture a rete possono essere limitate e controllate con più attenzione. Generalmente, solo gli oggetti di strati adiacenti possono comunicare direttamente. Le gerarchie riducono la complessità da esponenziale a polinomiale, introducendo strati di oggetti e vincoli di intercomunicazione tra strati. La complessità è nascosta e suddivisa in strati distinti.

5.2.1 Complessità delle reti

Per discorrere sulla *complessità* dei sistemi a oggetti è necessario accordarsi su una sua misura. Come misurare la complessità? La complessità può avere diverse tipologie e forme. Una semplice ma significativa misura è il numero di connessioni tra le classi, perché è attraverso queste connessioni che un oggetto può inviare un messaggio a un altro oggetto. Si definisce *connessione* l'esistenza di un legame persistente o transitorio tra le classi (Sezione 2.1.1.3).

Ogni connessione tipicamente permette un'interazione bidirezionale tra le classi, cioè dalla classe *A* verso *B* e da *B* verso *A*. La Figura 5.16 illustra la complessità di una rete formata da sette classi. Ci sono `n(n-1)/2` connessioni possibili tra *n* classi, e tale formula, applicata alle sette classi, risulta in 21 connessioni possibili (con 42 canali d'interazione).

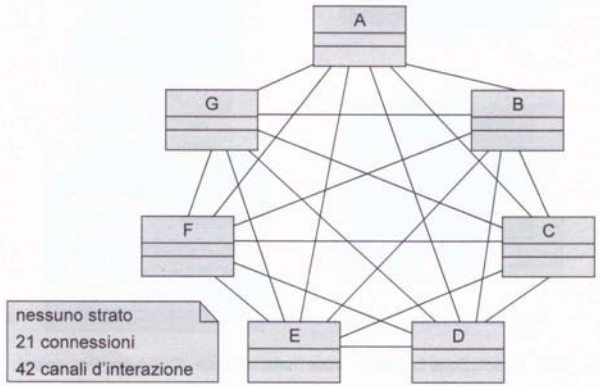


FIGURA 5.16 Complessità di reti.

Si noti che tale misura di complessità dipende dal numero di classi, non di oggetti. Nei programmi, gli oggetti, e non le classi, inviano messaggi ad altri oggetti di una stessa o di altra classe e questo introduce una difficoltà ulteriore per il programmatore responsabile del rilascio della logica applicativa, della gestione delle variabili del programma e delle altre strutture dati. La sfida principale, comunque, non è la complessità di un singolo programma ma la complessità dell'intero insieme di programmi sviluppati per gestire il sistema.

5.2.2 Complessità delle gerarchie

Un approccio al controllo della complessità consiste nella riduzione delle strutture di rete attraverso il raggruppamento delle classi in *gerarchie di classi*. In tal modo, le classi possono naturalmente essere organizzate in strati i quali enfatizzano una scomposizione gerarchica tra essi mentre, internamente, permettono interazioni a rete.

La stratificazione gerarchica riduce la complessità, raggruppando le classi in strati e permettendo solo interazioni dirette tra classi nello stesso strato e in strati adiacenti. Ad esempio, la Figura 5.17 mostra la complessità di una gerarchia di sette classi ammesso che sia possibile raggrupparle in quattro strati. A confronto della struttura a rete di Figura 5.16, la complessità è ridotta da 42 a 26 canali d'interazione possibili (l'esempio di Figura 5.17 è fatto in modo da corrispondere ai quattro livelli dell'approccio *BCDE* che sarà discusso nella Sezione 6.1.3.2).

In una struttura gerarchica con un alto numero di classi il numero di *connessioni interstrato* potrebbe ancora costituire un problema. In tal caso, per ridurre la complessità delle connessioni interstrato, le comunicazioni interstato potrebbero essere incanalate attraverso poche dedicate *classi mediatore*. Ciò ridurrebbe ulteriormente la complessità generale del sistema.

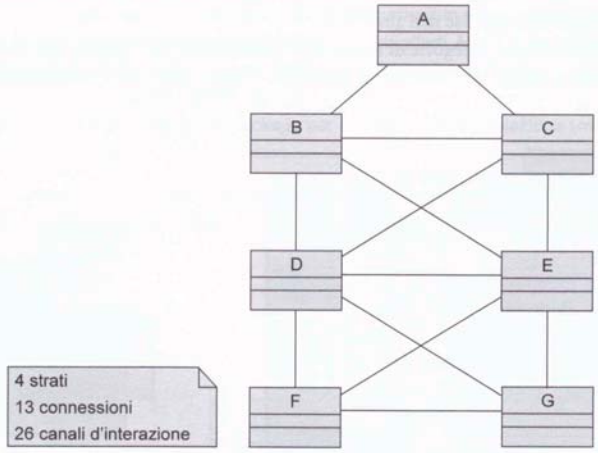


FIGURA 5.17 Complessità di gerarchie.

5.2.3 Package

UML fornisce una nozione di *package* (pacchetto) per rappresentare un gruppo di classi (o altri elementi di rappresentazione, ad esempio, casi d'uso) (Rumbaugh *et al.*, 1999). Un package serve per partizionare il modello logico di un programma applicativo raggruppando insiemi di classi strettamente relazionate tra loro e solo debolmente accoppiate ad altri gruppi (Lakos, 1996).

I package possono essere *annidati* e questo permette la decomposizione di grandi sistemi in sottosistemi, moduli, etc. Ad esempio, un package stereotipato come `system` potrebbe contenere package stereotipati come `subsystem`. Un package interno ha accesso agli elementi contenuti nel package esterno ma non viceversa. Le regole di visibilità si applicano per l'accesso agli elementi dei package interni poiché un package interno è incapsulato.

Una classe può essere posseduta solo da un package. Ciò non impedisce che la classe appaia in altri package o comunichi con classi in altri package. Dichiarando una classe all'interno di un package come pubblica, protetta o privata è possibile controllare la comunicazione e le dipendenze tra classi in package distinti (Sezione 5.1.4).

5.2.3.1 Notazione per i package

Un package è mostrato come un'icona a forma di raccoglitore (Figura 5.18). Un package annidato è disegnato internamente al proprio package esterno. Ciascun package deve avere un proprio diagramma delle classi che specifica le classi in esso contenute.

I package possono essere messi in relazione per mezzo di due tipologie di relazione: *generalizzazione* e *dipendenza*. La relazione nella Figura 5.18 è una dipendenza indicante che `PackageAnnidato` dipende da `Package2`. La natura esatta della dipendenza non è dettagliata, è specificato solo che le modifiche al `Package2` potrebbero richiedere modifiche in `PackageAnnidato`. La dipendenza più frequente riguarda il fatto che una classe di un package invia un messaggio a una classe dell'altro.

UML specifica diverse categorie di relazione di dipendenza (ad esempio, una dipendenza *usage*, una dipendenza *access*, una dipendenza *visibility*). È stato osservato che la determinazione delle categorie di dipendenza non è di particolare aiuto nei diagrammi d'analisi. La vera natura di ciascuna dipendenza dovrebbe essere inserita, come un vincolo descrittivo, nel repository del CASE per essere usata in seguito nei modelli di progetto.

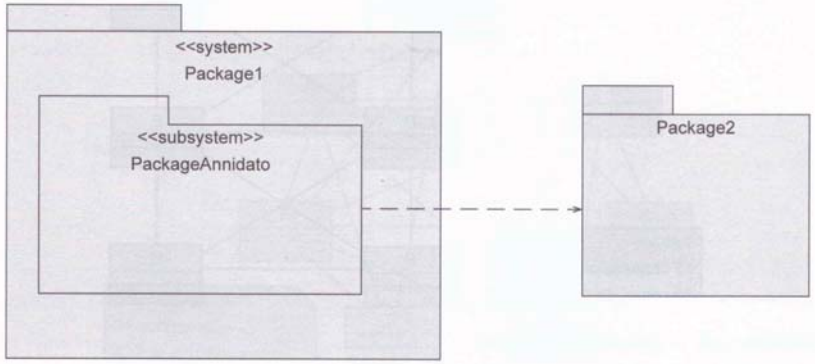


FIGURA 5.18 Package e dipendenze.

Si noti che una generalizzazione tra i package implica anche una dipendenza. La dipendenza implicata è da un package sottotipo al package supertipo. Le modifiche al package supertipo interessano il package sottotipo.

5.2.3.2 Diagramma dei package

Il concetto di diagramma dei package non esiste in UML ma è senza dubbio un utile strumento. I package sono correlati a un diagramma delle classi o in un diagramma dei casi d'uso. Nel primo caso, un diagramma dei package rappresenta una vista sullo stato del sistema mentre nel secondo caso, una vista sul comportamento del sistema. Una volta creato, l'assegnamento di classi (o di casi d'uso) a un package è descritta in un diagramma delle classi (o diagramma dei casi d'uso) distinto.

Senza dubbio, un diagramma dei package che esprime una vista sullo stato descrive la principale ossatura architetturale per il sistema. Invece, un diagramma dei package comportamentale serve solo a fini descrittivi per esprimere una struttura funzionale di alto livello (Conallen, 2000). Il diagramma dei package relativo allo stato aiuta nella gestione della dimensione e complessità del sistema. È anche fondamentale come meccanismo d'accesso e configurazione per rendere possibile la collaborazione tra sviluppatori.

Esempio 5.11 (Immatricolazioni universitarie)

Un'analisi più approfondita dell'applicazione Immatricolazioni universitarie rivela che il sistema deve essere "informato" dell'orario dei corsi offerti e dei curriculum scelti dagli studenti al fine di convalidare l'immatricolazione degli studenti stessi ai corsi offerti. Non è noto se "gestione orario" e "gestione curriculum" esistono come moduli distinti inseriti nel sistema d'immatricolazione. Se così non è, il sistema dovrà includere tali moduli.

Il compito è fornire un modello dei package per il sistema Immatricolazione Universitarie.

La Figura 5.19 mostra i tre package e le relazioni di dipendenza tra essi. `Immatricolazione` dipende da `Gestione Curriculum` e `Gestione Orario`.

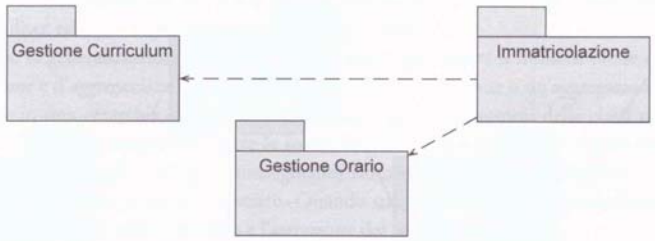


FIGURA 5.19 Package (Immatricolazioni universitarie).

5.2.4 L'approccio BCE

L'approccio BCE (*Boundary-Control-Entity*: frontiera, controllo, entità) è un approccio alla modellazione a oggetti basata su una ripartizione delle classi in tre categorie distinte, già introdotte nella Sezione 4.2 come classi di presentazione (boundary), di controllo (control) e di sistema (entity). Sarà usato l'acronimo BCE perché UML predefinisce le tre categorie (boundary, control ed entity) come possibili stereotipi su una classe (Rumbaugh *et al.*, 1999).

Una *classe boundary* descrive gli oggetti che rappresentano l'interfaccia tra un attore e il sistema. Essa rappresenta una parte dello stato del sistema che è presentata all'utente in forma visiva oppure sonora. Gli oggetti *boundary* spesso *persistono* al di là della singola esecuzione del programma.

Una *classe control* descrive oggetti che percepiscono gli eventi generati dall'utente e controllano l'esecuzione di un processo di business. Essa rappresenta un supporto per le azioni e le attività di un caso d'uso. Gli oggetti di controllo spesso non persistono oltre l'esecuzione del programma.

Una *classe entity* descrive gli oggetti che rappresentano la semantica delle entità nel dominio applicativo. Essa corrisponde a una struttura dati nel database del sistema. Gli oggetti *entity* sono sempre persistenti e possono partecipare a molti casi d'uso.

L'approccio BCE corrisponde approssimativamente al ben noto approccio *Model-View-Controller* (MVC) (Collins, 1995; Buschmann *et al.*, 1996; Olsen, 1998).

Come mostrato nella Figura 5.20, in una gerarchia di package ben progettata, un attore può interagire solo con gli oggetti nel `Package Boundary`, gli oggetti nel `Package Entity` possono interagire solo con gli oggetti nel `Package Control` e gli oggetti nel `Package Control` possono interagire con qualsiasi oggetto (si veda Conallen, 2000).

Le classi boundary corrispondono alle classi presenti nel progetto dell'interfaccia GUI. Gli oggetti boundary sono dipendenti dagli oggetti entity (tramite alcuni oggetti control). Un oggetto entity rende note le variazioni del suo stato in modo che l'oggetto della classe boundary possa aggiornare la visualizzazione. Una regione dello schermo che non è temporaneamente sincronizzata con l'entità cui si riferisce è detta *danneggiata*.

Gli oggetti control trattano le interazioni tra gli eventi generati dall'utente e gli oggetti entity e boundary interessati. In particolare, ciascun oggetto boundary che rende possibile un'interazione deve avere un oggetto control associato. A causa di ciò, alcune architetture (ad esempio, Microsoft Foundation Classes (MFC)) combinano le due funzionalità in una singola classe.

L'approccio BCE è un "modo di pensare" che riflette una buona pratica ingegneristica del software. Come tale, dovrebbe essere parte di qualsiasi metodo di sviluppo delle applicazioni, indipendentemente dalla sottostante piattaforma d'implementazione. Il principale vantaggio dell'approccio BCE è nel raggruppamento di classi in strati gerarchizzati. Ciò migliora la comprensibilità del modello e ne riduce la complessità. Riduce, inoltre, il rischio di creare classi che fanno troppo, *classi mostro* che hanno il controllo sull'intera funzionalità del sistema. In un buon progetto object-oriented, l'intelligenza del sistema è uniformemente distribuita tra molte classi.

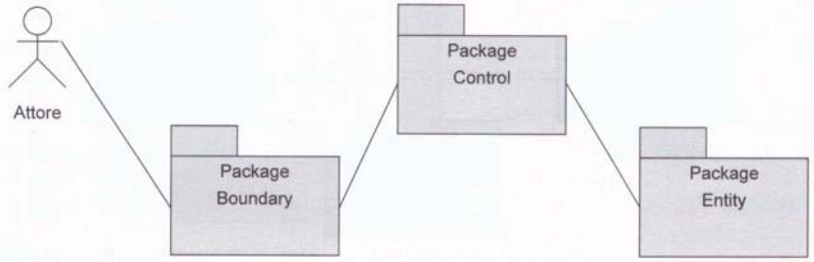


FIGURA 5.20 Connessioni d'oggetti nella gerarchia BCE di package.

Un ulteriore vantaggio dell'approccio BCE è la sua corrispondenza con le architetture moderne dei sistemi distribuiti in cui si distingue la gestione dei dati (entity) dalla presentazione (boundary), via un livello intermedio dedicato alla logica applicativa (control).

Bisogna comunque ricordare che `PackageEntity` descrive le classi entity nella memoria di programma dell'applicazione non nel database del server (quest'ultimo probabilmente non è neppure object-oriented ma relazionale, perciò gli oggetti persistenti del business sono memorizzati come "record" di tabelle). Per trattare la corrispondenza tra le classi entity e la memorizzazione relazionale, e per gestire altre rilevanti informazioni sulla struttura del database, è necessario uno strato distinto di classi chiamato `PackageDatabase` (si veda Sezione 6.1.3.2).

5.3 Generalizzazione ed ereditarietà

Ci sono tre principali tipologie di relazione tra classi: associazione, aggregazione e generalizzazione. Un attento lettore potrebbe aver notato che *non* è stata molto enfatizzata l'utilità della generalizzazione nei modelli d'analisi. La *generalizzazione* è un concetto utile e potente ma può anche creare molti problemi dovuti agli intricati meccanismi di *ereditarietà*, soprattutto in grandi progetti software.

I termini generalizzazione ed ereditarietà sono legati ma non sono equivalenti. È importante conoscere la differenza. Il prezzo dell'imprecisione è una mancanza di comprensione (come risulta evidente da un'analisi della letteratura, contenente spesso prolisse discussioni su vantaggi e svantaggi di generalizzazione ed ereditarietà).

La *generalizzazione* è una relazione semantica tra classi per cui l'*interfaccia* della sottoclasse deve comprendere tutte le proprietà (pubbliche e protette) della superclasse. L'*ereditarietà* è "il meccanismo per mezzo del quale elementi più specifici incorporano la struttura e il comportamento definito da elementi più generali" (Rumbaugh *et al.*, 1999).

5.3.1 Generalizzazione e sostituibilità

Dal punto di vista semantico, la *generalizzazione* introduce nuove classi, generiche o più specifiche, e stabilisce relazioni superclasse-sottoclasse nel modello.

Sebbene la generalizzazione introduca nuove classi, può ridurre il numero totale di relazioni d'*associazione* e d'*aggregazione* nel modello. Infatti, se un'associazione o un'aggregazione legano la superclasse in una gerarchia di generalizzazione (si vedano i diagrammi delle classi nelle Figure 2.31 e 4.7) il legame si estende a tutte le sottoclassi. In un buon modello, il compromesso tra profondità della generalizzazione e (conseguente) riduzione di relazioni d'associazione/aggregazione deve essere correttamente bilanciato. Quando usata con giudizio, la generalizzazione migliora l'espressività, la comprensibilità e l'astrazione dei modelli del sistema.

I benefici della generalizzazione provengono dal principio di *sostituibilità* secondo cui un oggetto della sottoclasse può essere usato al posto di un oggetto della superclasse in qualsiasi parte del codice ove vi sia un accesso all'oggetto della superclasse. Comunque, e sfortunatamente, il meccanismo d'ereditarietà potrebbe essere usato in modo da annullare i benefici del principio di sostituibilità.

5.3.2 Ereditarietà e incapsulamento

L'*incapsulamento* richiede che gli attributi di un oggetto siano accessibili solo attraverso le operazioni dichiarate nell'interfaccia dell'oggetto. Se applicato rigorosamente, l'incapsulamento conduce a un alto livello d'indipendenza dai dati in modo che future modifiche alle strutture dati incapsulate non richiedano anche modifiche ai programmi esistenti. Ma è questa la nozione di incapsulamento che ha senso nelle applicazioni?

L'*ereditarietà* compromette l'incapsulamento permettendo alle sottoclassi d'accedere direttamente ad *attributi protetti*. Le computazioni che spaziano su oggetti provenienti da classi distinte potrebbero richiedere che tali classi siano *amiche*, infrangendo quindi, ulteriormente, l'incapsulamento. Inoltre, gli *attributi statici*, globali per tutti gli oggetti della classe, potrebbero essere richiesti. In pratica, è impossibile dichiarare tutti i dati come privati. Infine, si osservi che l'incapsulamento si riferisce alla nozione di classe, non di oggetto, e nella maggior parte degli ambienti di programmazione object-oriented (ad eccezione di Smalltalk) un oggetto non può nascondere qualunque cosa a un altro oggetto della stessa classe.

Considerazioni pratiche possono essere fatte riguardo alle interrogazioni: gli utenti che accedono ai database per mezzo di SQL o simili *linguaggi d'interrogazione ad hoc*, vorrebbero riferirsi direttamente ad attributi nelle interrogazioni, piuttosto che essere obbligati a usare dei metodi di accesso ai dati che rendono più difficile ed esposta a errori la formulazione delle interrogazioni stesse. Tale requisito è particolarmente sentito in applicazioni di data warehouse con interrogazioni di tipo statistico OLAP (OnLine Analytical Processing).

Le applicazioni dovrebbero quindi essere progettate in modo da raggiungere un livello ragionevole d'incapsulamento, bilanciandolo con ereditarietà, interrogazioni ad hoc, e requisiti computazionali.

5.3.3 Ereditarietà dell'interfaccia

Quando la generalizzazione è usata con l'obiettivo della sostituibilità, allora è sinonimo della nozione di *ereditarietà dell'interfaccia* (*ereditarietà tipo-sottotipo*). Questa è una forma "tranquilla" di ereditarietà. Una sottoclasse eredita attributi (tipi) e operazioni (segnature): si dice che una sottoclasse supporta l'interfaccia della superclasse. L'implementazione delle operazioni ereditate può essere differita.

Le *interfacce* possono essere dichiarate attraverso un *classe astratta* in cui *tutte* le operazioni sono astratte (sono solo dichiarate, rimandandone la specifica alle sottoclassi). Si ricordi che una classe astratta potrebbe fornire implementazioni (anche solo parziali) per alcune operazioni mentre un'interfaccia differisce tutte le specifiche delle operazioni.

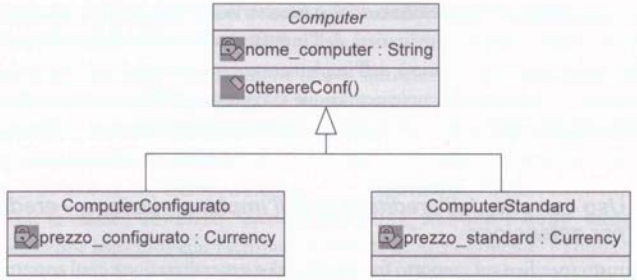


FIGURA 5.21 Ereditarietà dell'interfaccia.

La Figura 5.21 è un estratto dal modello delle classi relativo al caso Telemarketing che mostra l'ereditarietà dell'interfaccia. `Computer` è una classe astratta, nessun oggetto di essa può essere creato. L'operazione `ottenereConf()` avrà due implementazioni distinte, una in `ComputerConfigurato` e l'altra in `ComputerStandard`. Supponendo di lasciare in `Computer` solo la definizione delle operazioni di interfaccia, Figura 5.22 è un modo alternativo di rappresentare l'ereditarietà dell'interfaccia usando il simbolo "lollypop" per indicare l'interfaccia supportata. La relazione "lollypop" da una sottoclasse (classe concreta) a una classe astratta è formalmente chiamata *relazione di realizzazione*.

5.3.4 Ereditarietà dell'implementazione

Come osservato nella precedente sezione, la generalizzazione può essere usata per ottenere la *sostituibilità* ed è quindi realizzata attraverso l'*ereditarietà dell'interfaccia*. Tuttavia, la generalizzazione può anche essere usata (deliberatamente o meno) per ottenere il *riuso del codice* ed è quindi realizzata per mezzo dell'*ereditarietà dell'implementazione*. Questa interpretazione della generalizzazione è molto potente (talvolta "pericolosamente" potente) ed è anche l'interpretazione più comune.

L'*ereditarietà dell'implementazione* (chiamata anche *ereditarietà del codice* o *ereditarietà di classe*) implica l'eredità delle proprietà dalla superclasse nelle sottoclassi e permette la loro modifica o riscrittura (*overriding*) con nuove implementazioni, quando necessario. La riscrittura può significare l'inclusione (la chiamata) di un metodo della superclasse nel metodo della sottoclasse estendendo quest'ultimo con nuova funzionalità. Può anche significare una completa sostituzione del metodo della superclasse da parte di quello della sottoclasse. L'ereditarietà dell'implementazione permette la condivisione delle descrizioni delle proprietà, il *riuso del codice* e il *polimorfismo*.

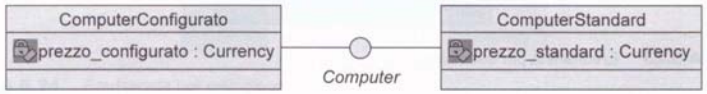


FIGURA 5.22 Uso della relazione di realizzazione per rappresentare l'ereditarietà dell'interfaccia.

Costruendo modelli con la generalizzazione è necessario aver ben chiaro quale tipologia d'ereditarietà sia implicata. L'uso dell'ereditarietà dell'interfaccia è sicuro poiché coinvolge solo la segnatura delle operazioni. L'ereditarietà dell'implementazione coinvolge l'ereditarietà del codice, cioè l'ereditarietà di frammenti d'implementazione (Szyperski, 1998; Harmon e Watson, 1998). Se non controllata con attenzione, l'ereditarietà dell'implementazione può creare dei problemi. Di seguito, sono discussi vantaggi e svantaggi dell'ereditarietà dell'implementazione.

5.3.4.1 Uso corretto dell'ereditarietà dell'implementazione – ereditarietà per estensione

UML è piuttosto specifico sul rapporto fra ereditarietà e generalizzazione e sul corretto uso dell'ereditarietà dell'implementazione (Rumbaugh *et al.*, 1999). Il solo uso corretto dell'ereditarietà è la definizione incrementale di una classe (*ereditarietà per estensione*). Una sottoclasse ha più proprietà (attributi e/o metodi) della sua superclasse, cioè una sottoclasse *è-un-tipo-di* superclasse.

L'esempio nella Figura 2.17 (qui ripetuto per convenienza come Figura 5.23) rappresenta un'ereditarietà per estensione. Qualsiasi oggetto di `Impiegato` *è-un-tipo-di* oggetto di `Persona`. Ciò non significa che un oggetto `Impiegato` sia contemporaneamente istanza di due classi: `Persona` e `Impiegato` (si veda la discussione sulla classificazione multipla alla Sezione 2.1.5.2.2). Un oggetto di `Impiegato` è istanza della classe `Impiegato` e ci sono oggetti di `Persona` che sono solo oggetti di tale classe (cioè non sono oggetti di `Impiegato`).

Nell'ereditarietà per estensione, la riscrittura di proprietà va usata con attenzione. Dovrebbe essere permesso solo rendere le proprietà più specifiche (introducendo vincoli sui valori che possono assumere gli attributi o definendo implementazioni più efficienti delle operazioni). Se la riscrittura cambiasse il significato di una proprietà, allora un oggetto della sottoclasse potrebbe non essere più sostituibile a un oggetto della superclasse ove fosse richiesto.

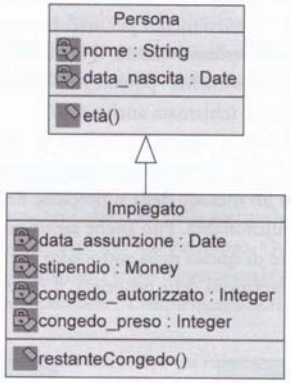


FIGURA 5.23 Ereditarietà per estensione.

5.3.4.2 Uso problematico dell'ereditarietà dell'implementazione - ereditarietà per restrizione

Nell'ereditarietà per estensione, la definizione di una sottoclasse è estesa con nuove proprietà. È comunque possibile usare l'ereditarietà come un meccanismo di restrizione se alcune delle proprietà ereditate sono eliminate (rimosse) nella sottoclasse. Tale ereditarietà è chiamata *ereditarietà per restrizione* (Rumbaugh *et al.*, 1991).

La Figura 5.24 mostra due esempi d'ereditarietà per restrizione. Poiché l'ereditarietà non può essere selettiva, la classe `Cerchio` erediterebbe `asse_minore` e `asse_maggiore` da `Ellisse` e dovrebbe sostituirli con l'attributo `diametro`. Analogamente, `Pinguino` erediterebbe la capacità di volare (operazione `volare`) da `Uccello` e dovrebbe sostituirla con l'operazione `nuotare` (forse volare a un'altitudine negativa potrebbe stare per nuotare).

L'ereditarietà per restrizione è problematica. Un oggetto della superclasse può essere anche sostituito da un oggetto della sottoclasse ammesso che chiunque stia usando l'oggetto sia consapevole che alcune proprietà sono soppresse.

Nell'ereditarietà per restrizione, le proprietà di una classe sono usate (attraverso l'ereditarietà) per implementare un'altra classe. Se la rimozione non è estesa, l'ereditarietà per restrizione può avere dei benefici ma, in generale, essa presenta problemi di manutenzione.

5.3.4.3 Uso improprio dell'ereditarietà dell'implementazione - ereditarietà per convenienza

L'ereditarietà che non è né d'estensione né di restrizione è una "triste novità" nella modellazione di sistema. Una tale ereditarietà può esistere tra due o più classi che hanno implementazioni simili ma senza alcuna relazione tassonomica tra i concetti rappresentati dalle classi. Una classe è arbitrariamente scelta come antenato delle altre classi. Questa è chiamata *ereditarietà di convenienza* (Maciaszek *et al.*, 1996a; Rumbaugh *et al.*, 1991).

La Figura 5.25 presenta due esempi di ereditarietà per convenienza. La classe `Segmento` è definita come sottoclasse della classe `Punto`. Chiaramente, un segmento non è un punto e conseguentemente la generalizzazione come definita in precedenza non può applicarvisi. Tuttavia,

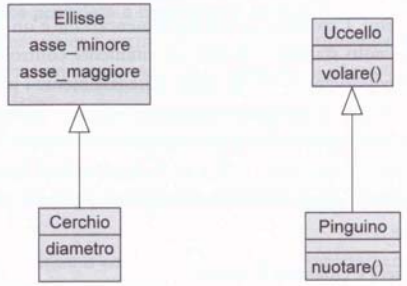


FIGURA 5.24 Ereditarietà per restrizione.

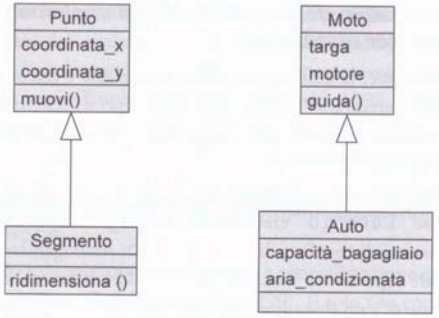


FIGURA 5.25 Ereditarietà per convenienza.

l'ereditarietà può essere usata. Di fatti, per la classe `Punto` è possibile definire attributi come `coordinata_x` e `coordinata_y` e un'operazione `Muovi`. La classe `Segmento` erediterebbe tali proprietà e potrebbe definire un'operazione aggiuntiva `Ridimensiona`. L'operazione `Muovi` deve essere rimossa. Analogamente, nel secondo esempio, `Auto` eredita le proprietà di `Moto` e ne aggiunge alcune nuove.

L'ereditarietà per convenienza è impropria ed è semanticamente scorretta. Inoltre, conduce a una rimozione estesa delle proprietà. Il principio di sostituibilità non è normalmente valido poiché gli oggetti non sono di tipo simile (`Segmento` non è un `Punto`, `Auto` non è una `Moto`); purtroppo, nella pratica, gli sviluppatori usano spesso l'ereditarietà per convenienza poiché molti ambienti di programmazione a oggetti incoraggiano indiscriminatamente l'uso dell'ereditarietà dell'implementazione. Molti linguaggi sono dotati di una miriade di strumenti per una "programmazione potente" con l'ereditarietà mentre il supporto per altre caratteristiche (specialmente l'aggregazione) degli oggetti è dimenticato.

5.3.4.4 *I rischi nell'ereditarietà dell'implementazione*

La precedente discussione non significa che sia sufficiente vietare l'ereditarietà per convenienza per eliminare tutti i problemi. L'ereditarietà dell'implementazione è un problema complesso e rischioso da affrontare per molti standard. Se non attentamente controllata e gestita, si può fare troppo uso e abuso dell'ereditarietà, reintroducendo potenzialmente i problemi che, in un primo tempo, si era pensato di risolvere. Ciò è particolarmente vero nello sviluppo di grandi sistemi con centinaia di classi e migliaia di oggetti, con modifiche dinamiche allo stato degli oggetti e con l'evoluzione delle strutture delle classi (come nelle tipiche applicazioni di business).

I principali fattori di rischio sono in relazione ai seguenti problemi (Szyperski, 1998):

- Classi di base "fragili".
- Modifica delle proprietà e chiamate di ritorno.
- Ereditarietà multipla dell'implementazione.

5.3.4.4.1 Classi di base fragili

Il problema delle *classi di base* (superclassi) *fragili* riguarda la definizione di sottoclassi ancora valide e operazionali quando la loro superclasse (o superclassi, in caso d'ereditarietà multipla) subisce un'evoluzione nell'*implementazione*: si tratta di un problema serio, in particolare quando si considera che le superclassi potrebbero essere ottenute da fonti esterne, senza alcun controllo da parte del gruppo di sviluppo del sistema.

Si supponga, ad esempio, che le superclassi dalle quali l'applicazione deve ereditare siano parte di un sistema operativo, un DBMS o un'interfaccia GUI. Se si acquista un OODBMS (Object Oriented DBMS) per lo sviluppo dell'applicazione, si acquista, in realtà, una libreria di classi che implementa le funzioni tipiche di un database, come la persistenza, la gestione delle transazioni, la concorrenza, il ripristino, etc. Se le classi dell'applicazione ereditano da quella libreria di classi, l'impatto delle nuove versioni del OODBMS sull'applicazione non è prevedibile (ed è certamente così se non si prendono precauzioni durante il progetto del modello d'ereditarietà per l'applicazione).

Il problema delle classi di base fragili è difficile da gestire, a meno di dichiarare immutabili tutte le interfacce pubbliche o, almeno, evitare che l'ereditarietà dell'implementazione dalla superclasse sia fuori controllo. Modifiche all'implementazione di una superclasse (di cui è anche possibile non avere il codice sorgente) hanno un effetto spesso imprevedibile sulle sottoclassi nel sistema applicativo. Ciò è vero anche se l'interfaccia delle superclassi non è modificata. La situazione può essere ulteriormente aggravata se le modifiche interessano anche le interfacce. Alcuni esempi sono (Szyperski, 1998):

- Modificare la segnatura di un metodo.
- Suddividere un metodo in due o più nuovi metodi.
- Riunire metodi esistenti in un metodo più ampio.

Il nucleo del problema riguarda il fatto che gli sviluppatori della superclasse dovrebbero conoscere in anticipo come altri sviluppatori ignoti useranno quella superclasse in futuro, cosa, ovviamente, impossibile senza una "sfera di cristallo". Nella sezione 5.4, si discuteranno alcuni metodi alternativi di sviluppo che, pur non basandosi sull'ereditarietà, producono ancora la funzionalità attesa di un oggetto.

5.3.4.4.2 Modifica delle proprietà e chiamate di ritorno

L'ereditarietà dell'implementazione permette di scegliere la tipologia di sostituzione del codice ereditato. Esistono cinque tecniche secondo cui un metodo di una sottoclasse può riusare il codice presente in una superclasse:

1. La sottoclasse può ereditare l'interfaccia e l'implementazione del metodo e non introdurre modifiche all'implementazione.
2. La sottoclasse può ereditare il codice e includerlo (chiamarlo) nel proprio metodo con la stessa segnatura.
3. La sottoclasse può ereditare il codice e modificarlo completamente con una nuova implementazione e con la stessa segnatura.
4. La sottoclasse può ereditare il codice vuoto (cioè la dichiarazione del metodo è vuota) e quindi fornire l'implementazione per quel metodo.
5. La sottoclasse può ereditare solo l'interfaccia del metodo (ereditarietà dell'interfaccia) e quindi fornire l'implementazione per quel metodo.

Delle cinque tecniche di riuso, le prime due sono le più "dolorose" quando la classe base evolve. La terza mostra "disprezzo" per l'ereditarietà. Le ultime due tecniche sono casi speciali: la quarta è banale, la quinta non coinvolge l'ereditarietà dell'implementazione.

Esempio 5.12 (Telemarketing)

Riferendosi all'applicazione di Telemarketing e in particolare all'Esempio 5.4 (Sezione 5.1.4.1), modificare la relazione di generalizzazione tra `Campagna` e `CampagnaBonus` per includere operazioni che esemplifichino le prime tre tecniche di riuso precedentemente elencate.

La Figura 5.26 mostra una versione della relazione di generalizzazione tra `Campagna` e `CampagnaBonus` adatta a mostrare le tre tecniche di riuso. Le operazioni coinvolte sono: `ottenereDataFine`, `stampareDettagliBiglietti` e `calcolareBigliettiRimasti`.

L'implementazione dell'operazione `ottenereDataFine()` è ereditata da `CampagnaBonus` e riusata così com'è (Tecnica 1). Le due restanti operazioni sono modificate in `CampagnaBonus` (sono stati duplicati i loro nomi nella sottoclasse).

Si assuma che l'operazione `stampareDettagliBiglietti` nella sottoclasse segua la Tecnica 2, cioè richiami l'operazione ereditata `Campagna.stampareDettagliBiglietti()` per stampare, diciamo, il numero totale di biglietti nella campagna, e quindi stampare informazioni specifiche sui biglietti bonus quale numero di blocchi di biglietti e la loro dimensione.

Si assuma anche che l'operazione `calcolareBigliettiRimasti()` segua la Tecnica 3 di riuso, cioè in `CampagnaBonus` il codice ereditato di `Campagna.calcolareBigliettiRimasti()` sia modificato completamente.

L'Esempio 5.12 mostra come la modifica contribuisca al problema della classe base fragile. L'ereditarietà dell'implementazione combinata con lo scambio di messaggi può creare ovunque canali di comunicazione: accanto a semplici *chiamate che seguono la direzione dell'ereditarietà*, possono essere effettuate *chiamate di ritorno* (Sezione 6.2.2.8). Tali chiamate sono chiamate all'oggetto mittente e possono ricreare una rete di canali di comunicazione che, come già detto, è insostenibile nel caso di grandi sistemi (Sezione 5.2).

Come osservato da Szyperski (1998, p. 104): "Insieme allo stato osservabile, ciò conduce a

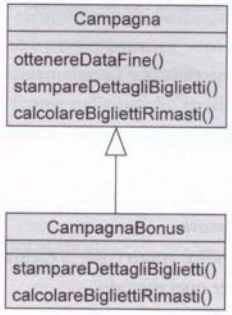


FIGURA 5.26 Ereditarietà e overriding (Telemarketing).

una semantica rientrante, simile a quella dei sistemi concorrenti. Grafi arbitrari di chiamata formati da reti interagenti d'oggetti annullano la classica stratificazione e rendono norma il codice rientrante".

Per rendere giustizia all'ereditarietà, le chiamate di ritorno sono possibili ogniqualvolta ci sia un riferimento tra oggetti. Per citare ancora Szyperski (1998, p. 57): "con il riferimento a oggetti... ogni invocazione di metodo è potenzialmente una chiamata verso l'alto, ogni metodo potenzialmente una chiamata di ritorno". L'ereditarietà aumenta solo la difficoltà; il problema è che l'aumento risulta non indifferente!

5.3.4.4.3 Ereditarietà multipla dell'implementazione

L'ereditarietà multipla è stata introdotta nella Sezione 2.1.5.2.1 ma senza fare distinzione tra l'*ereditarietà multipla dell'interfaccia* (sottotipo multiplo) e l'*ereditarietà multipla dell'implementazione* (sottoclasse multipla). L'ereditarietà multipla dell'interfaccia permette la fusione dei "contratti" d'interfaccia. L'ereditarietà multipla dell'implementazione permette la fusione di frammenti d'implementazione.

L'*ereditarietà multipla dell'implementazione* non aggiunge realmente un nuovo "problema" all'ereditarietà dell'implementazione. Piuttosto, esaspera i problemi causati da classi base fragili, modifiche e chiamate di ritorno. Indipendentemente dalla necessità di limitare l'ereditarietà di qualsiasi frammento di codice ripetuto (se due o più superclassi definiscono la stessa operazione), l'ereditarietà multipla potrebbe anche forzare la rinomina di operazioni ogniqualvolta i relativi nomi siano uguali (e in realtà corrispondano a operazioni distinte).

In questo contesto, vale la pena di ricordare l'inerente crescita della complessità dovuta all'ereditarietà multipla, crescita dovuta alla mancanza di supporto nei sistemi a oggetti per la *classificazione multipla* (Sezione 2.1.5.2.2). In queste condizioni, tutte le diramazioni d'ereditarietà ortogonali con radice in una comune superclasse devono essere trattate introducendo classi "unione" specialmente create per ogni possibile combinazione delle diramazioni (si veda ad esempio la Sezione 2.1.5.2.2).

5.4 Modellazione di aggregazione e delegazione

L'aggregazione è la più potente tecnica conosciuta per gestire la complessità di grandi sistemi attraverso l'allocazione di classi in strati organizzati in gerarchie d'astrazione. Sebbene l'aggregazione sia stata considerata un concetto fondamentale di rappresentazione almeno quanto la generalizzazione, solo un'attenzione marginale le è stata riservata nello sviluppo a oggetti delle applicazioni (ad eccezione di domini applicativi particolari come i sistemi multimedia).

L'*aggregazione* (e la sua variazione più forte, la composizione) è una relazione di *contenimento*. Una *classe composta* contiene una o più *classi componenti*. Le classi componenti sono elementi della classe composta (elementi che hanno un'esistenza propria).

Negli ambienti di programmazione (compresa la maggioranza dei database a oggetti), l'aggregazione è implementata allo stesso modo delle associazioni convenzionali, per mezzo di riferimenti tra gli oggetti composto e componente. Sebbene a tempo di compilazione la struttura dell'aggregazione sia la stessa dell'associazione, il comportamento è diverso. La semantica dell'aggregazione è più forte ed è (sfortunatamente) responsabilità del programmatore assicurare che il comportamento obbedisca a tale semantica.

5.4.1 Semantica dell'aggregazione

Mentre gli attuali ambienti di programmazione ignorano l'aggregazione, i metodi di sviluppo delle applicazioni a oggetti incorporano l'aggregazione ma le riservano minor enfasi. Inoltre (o come conseguenza della mancanza di supporto negli ambienti di programmazione), i metodi di sviluppo di applicazioni a oggetti non si sforzano di rafforzare un'interpretazione semantica rigorosa del costrutto d'aggregazione, trattandolo spesso come una speciale forma d'associazione.

Come discusso nella Sezione 4.2.3, quattro possibili semantiche per l'aggregazione possono essere distinte (Maciaszek *et al.*, 1996b):

1. aggregazione " possiede in modo esclusivo",
2. aggregazione "possiede",
3. aggregazione "ha",
4. aggregazione "raggruppa".

UML riconosce solo due semantiche dell'aggregazione, chiamate *aggregazione* (semantica per referenza) e *composizione* (semantica per valore) (Sezione 2.1.4). Nel seguito si mostra come la notazione UML esistente può essere usata per rappresentare le quattro tipologie distinte d'aggregazione precedentemente identificate.

5.4.1.1 L'aggregazione "possiede in modo esclusivo"

L'aggregazione "possiede in modo esclusivo" può essere rappresentata in UML per mezzo di una composizione (o un'aggregazione) stereotipata con `possesso esclusivo` e vincolata con la parola chiave *bloccato* (*frozen*) (Fowler e Scott, 2000). Il vincolo *bloccato* si applica a una classe componente e stabilisce che un oggetto di tale classe non può essere *riconnesso* (durante la sua esistenza) a un altro oggetto composto, cioè un oggetto componente può essere cancellato ma non può essere passato a un altro proprietario. L'uso della composizione o della semplice aggregazione dipende dalla semantica del caso in esame tenendo presente che una relazione è più specifica dell'altra (Sezione 2.1.4)

La Figura 5.27 mostra due esempi d'aggregazione *possesso esclusivo*. L'esempio di sinistra è rappresentato con la semantica per valore di UML (un rombo pieno), quello di destra con la semantica per referenza di UML (un rombo vuoto).

Un oggetto di `Capitolo` è parte al più di un `Libro`. Una volta incorporato (per valore) in

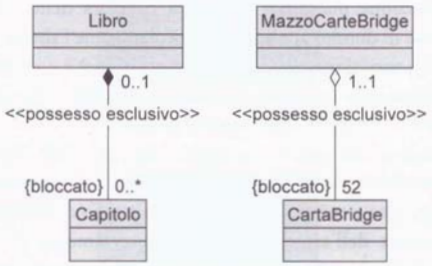


FIGURA 5.27 Aggregazione *possiede in modo esclusivo*.

un oggetto composto, non può essere riconnesso a un altro oggetto `Libro`, per cui la connessione è bloccata.

Un `MazzoCarteBridge` contiene esattamente cinquantadue carte. La semantica per referenza di UML è usata per rappresentare il possesso. Qualsiasi oggetto di `CartaBridge` proviene esattamente da un oggetto di `MazzoCarteBridge` e non può essere riconnesso a un altro mazzo di carte. A causa della dipendenza esistenziale, la distruzione di un oggetto composto implica la distruzione dei suoi componenti.

5.4.1.2 L'aggregazione "possiede"

Come l'aggregazione *possiede in modo esclusivo*, l'aggregazione *possiede* può essere espressa in UML con la semantica per valore della composizione (un rombo pieno) o la semantica per referenza dell'aggregazione (un rombo vuoto). In qualunque momento un oggetto componente può appartenere al più a un oggetto composto (cardinalità 0..1 o 1..1) ma può essere *riconnesso* a un altro oggetto composto. Quando un oggetto composto è cancellato, i suoi oggetti componenti sono anch'essi cancellati.

La Figura 5.28 mostra due esempi d'aggregazione *possiede*. Un oggetto `Amplificatore` può essere riconnesso a un altro oggetto di `ImpiantoStereo`. Analogamente un oggetto `Copertone` può essere spostato da un oggetto di `Bicicletta` a un altro. Anche in questo caso, la distruzione di un oggetto di `ImpiantoStereo` o `Bicicletta` si propaga verso gli oggetti componenti.

5.4.1.3 L'aggregazione "ha"

L'aggregazione *ha* normalmente si rappresenta in UML usando la semantica per referenza dell'aggregazione (un rombo vuoto) con molteplicità (0..1) o (1..1) sul ruolo dell'oggetto composto. Non c'è una dipendenza esistenziale nell'aggregazione *ha*: la cancellazione di un oggetto composto non si propaga automaticamente verso gli oggetti componenti. L'aggregazione *ha* è caratterizzata solo da transitività e asimmetria.

Un esempio di aggregazione *ha* è mostrato nella Figura 5.29. Se un oggetto di `Carrello` ha svariati oggetti di `CassaBirra` e un oggetto di `CassaBirra` contiene svariati oggetti di `BottigliaBirra`, allora quell'oggetto di `Carrello` ha quegli oggetti di `BottigliaBirra` (*transitività*). Se un oggetto di `Carrello` ha un oggetto di `CassaBirra` allora quest'ultimo non può avere quell'oggetto di `Carrello` (*asimmetria*).

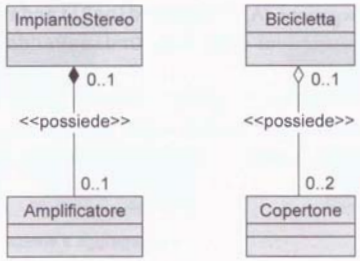


FIGURA 5.28 Aggregazione *possiede*.

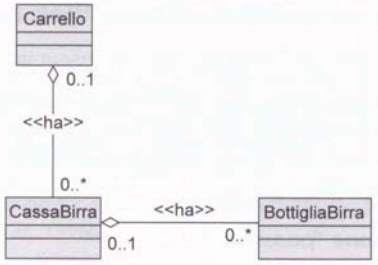


FIGURA 5.29 Aggregazione *ha*.

5.4.1.4 L'aggregazione "raggruppa"

L'aggregazione *raggruppa* permette la *molteplicità molti-a-molti* nella relazione. Nessuna speciale assunzione è fatta su dipendenza esistenziale, transitività, asimmetria o blocco. Se necessario, queste proprietà possono essere espresse introducendo dei vincoli UML opportuni. A causa della molteplicità molti-a-molti, l'aggregazione *raggruppa* può essere rappresentata in UML usando la semplice semantica per referenza dell'aggregazione (un rombo vuoto).

La Figura 5.30 mostra quattro relazioni d'aggregazione *raggruppa*. Un oggetto di `SessioneJAD` (Sezione 3.2.2.2) consiste di un moderatore e uno o più segretari, utenti e sviluppatori. Ogni oggetto elementare può "entrare" in più di un oggetto di `SessioneJAD`.

5.4.2 Aggregazione come alternativa alla generalizzazione

Una generalizzazione può essere rappresentata come un'aggregazione. Si consideri la Figura 5.31. Gli *ordini* del cliente che non sono evasi possono essere *pendenti* per una qualche ulteriore azione. L'ordine pendente può essere *ritardato* se per essere evaso si attende che la scorta sia sufficiente, oppure è un ordine *futuro* se deve essere evaso a una data specificata dal cliente.

Il modello a sinistra è una generalizzazione (G) per gli ordini cliente: la classe `OrdineG` può essere un `OrdinePendenteG` che a sua volta può essere `OrdineRitardatoG` oppure `OrdineFuturoG`. L'ereditarietà assicura la condivisione di attributi e operazioni tra tutte le classi dell'albero di generalizzazione. Nel modello di destra della Figura 5.31 è mostrata una semantica simile ottenuta con l'aggregazione (A): le classi `OrdineRitardatoA` e `OrdineFuturoA` includono gli attributi e le operazioni della classe `OrdinePendenteA` la quale a sua volta incorpora la classe `OrdineA`.

Sebbene i due modelli di Figura 5.31 abbiano semantica simile, ci sono delle differenze. La principale deriva dall'osservazione che il modello con la generalizzazione è basato sulla nozione di classe mentre il modello con l'aggregazione è centrato sulla nozione di oggetto.

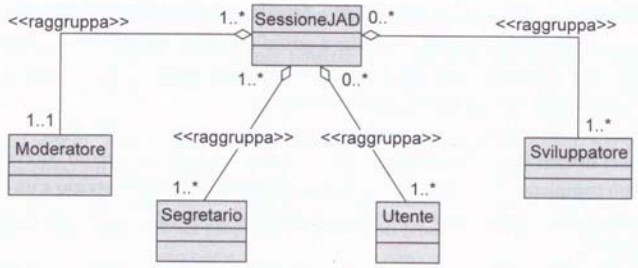


FIGURA 5.30 Aggregazione *raggruppa*.

Un particolare oggetto di `OrdineRitardatoG` è anche un oggetto di `OrdinePendenteG` e di `OrdineG`. C'è un solo *identificatore d'oggetto* (OID) per `OrdineRitardatoG`. Nel caso dell'aggregazione, un particolare oggetto `OrdineRitardatoA` è costruito su tre oggetti distinti, ciascuno con il proprio OID, l'oggetto `OrdineRitardatoA` stesso, l'oggetto in esso contenuto `OrdinePendenteA` e l'oggetto in quest'ultimo contenuto `OrdineA`.

La generalizzazione usa l'*ereditarietà* per implementare la propria semantica. L'aggregazione usa la *delegazione* per riusare l'implementazione degli oggetti componenti, come discusso nel seguito.

5.4.2.1 Delegazione e prototipi

Il modello computazionale dell'ereditarietà è basato sulla nozione di classe. Tuttavia è possibile basare il modello computazionale sulla nozione di oggetto. Un modello computazionale centrato su oggetti struttura gli oggetti in gerarchie d'aggregazione. Ogniqualvolta un oggetto composto (*oggetto esterno*) non riesce a completare un compito da solo, può chiamare i metodi in uno dei suoi oggetti componenti (*oggetti interni*): questo meccanismo è chiamato *delegazione*.

Negli approcci basati sulla delegazione la funzionalità del sistema è implementata includendo (*clonando*) la funzionalità di oggetti esistenti nella nuova funzionalità richiesta. L'idea è ricercare

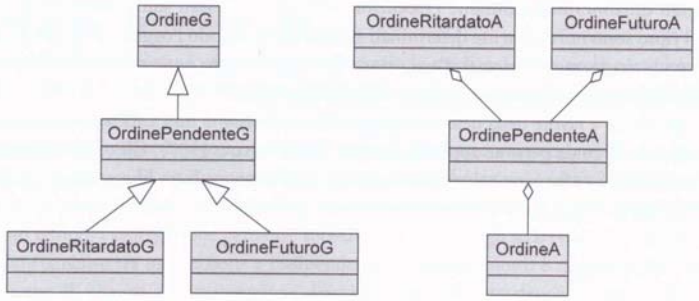


FIGURA 5.31 Generalizzazione e aggregazione.

funzionalità utili negli oggetti esistenti (*oggetti interni*) e quindi implementare la funzionalità desiderata in un *oggetto esterno* tramite chiamate a servizi forniti da oggetti interni. Gli oggetti esistenti sono trattati come prototipi per la creazione di nuovi oggetti, per cui i sistemi che si costruiscono in tal modo sono chiamati *sistemi prototipali*.

Un oggetto potrebbe avere una *relazione di delegazione* con qualsiasi altro oggetto identificabile e visibile nel sistema (Lee e Tepfenhart, 1997). Quando un oggetto "esterno" riceve un messaggio e non può completare il servizio da solo, delega l'esecuzione di quel servizio a un oggetto "interno". Quest'ultimo, se necessario, potrebbe inoltrare il messaggio a uno qualunque dei suoi oggetti "interni".

I quattro tipi d'aggregazione identificati nella Sezione 5.4.1 possono essere usati per controllare il livello di visibilità di oggetti interni. Ad esempio, un oggetto esterno potrebbe esporre le interfacce degli oggetti interni usando forme più deboli d'aggregazione (come nelle aggregazioni *ha* e *raggruppa*). Nelle forme forti d'aggregazione, un oggetto esterno potrebbe nascondere al mondo esterno le interfacce dei propri oggetti interni (introducendo quindi un livello d'*incapsulamento*).

5.4.2.2 Delegazione ed ereditarietà

Si può dimostrare che la delegazione permette di rappresentare l'ereditarietà e viceversa. Questo significa che la stessa funzionalità di sistema può essere ottenuta con l'ereditarietà *oppure* con la delegazione. Il consenso su tale problema fu raggiunto per la prima volta alla conferenza di Orlando, Florida, nel 1987 ed è noto come il Trattato di Orlando (Stein *et al.*, 1989).

Nelle precedenti sezioni, sono stati discussi i problemi connessi all'ereditarietà dell'implementazione, per cui ci si può chiedere se la delegazione permetta di evitare tali problemi. La risposta a questa domanda non è semplice (Szyperski, 1998).

Dal punto di vista del *riuso*, la delegazione è molto vicina all'ereditarietà. Un oggetto esterno *riusa* l'implementazione di un oggetto interno. La differenza è che, nel caso dell'ereditarietà, il controllo è sempre ritornato all'oggetto che riceve il messaggio originale (richiesta di servizio) dopo che il servizio è stato portato a termine. Nel caso della delegazione, una volta che il controllo è passato da un oggetto esterno a un oggetto interno, rimane in quest'ultimo. Qualsiasi ritorno all'oggetto che ha emesso il messaggio deve essere esplicitamente gestito nella delegazione.

L'altro potenziale vantaggio della delegazione è che la condivisione e il riuso possono essere determinati dinamicamente durante l'esecuzione. Nei sistemi basati sull'ereditarietà, la condivisione e il riuso sono normalmente determinati staticamente quando l'oggetto è creato. Il compromesso è tra la sicurezza e la velocità d'esecuzione della *condivisione anticipata* dell'ereditarietà, e la flessibilità della *condivisione non anticipata* della delegazione.

Un argomento a favore della delegazione è che la condivisione non anticipata è più naturale e vicina al modo in cui le persone apprendono (Lee e Tepfenhart, 1997). Gli oggetti sono naturalmente combinati per formare soluzioni più ampie e possono evolvere liberamente. La prossima sezione fornirà un altro punto di vista sul medesimo problema.

5.4.3 Aggregazione e oloni - alcune considerazioni

In Maciaszek *et al.* (1996a) e (1996b), per limitare la complessità dei modelli a oggetti è stato proposto un nuovo approccio per descrivere le architetture software, basato sull'interpretazione di Arthur Koestler della struttura dei sistemi naturali (Koestler, 1967; Koestler, 1978). Il concetto centrale è l'idea di "*oloni*" i quali possono essere interpretati sia come oggetti parte di altri più complessi, sia come un tutto costituito a sua volta da oggetti più semplici.

Questo concetto deriva dai *sistemi viventi* che, strutturalmente, sono *aggregazioni* di unità semiautonome le quali mostrano sia proprietà indipendenti (come se fossero un tutto) sia proprietà interdipendenti (come se fossero parte, insieme ad altre con cui interagiscono, di un tutto a livello gerarchico più alto). Come Arthur Koestler direbbe, sono aggregazioni di oloni, dal Greco *holos* che significa il *tutto*, con il suffisso *one* (al singolare) per suggerire una particella o una parte (come in protone o neutrone) (Koestler, 1967).

Parte e tutto in senso assoluto non esistono negli organismi viventi o ugualmente nei sistemi sociali. Gli oloni sono stratificati gerarchicamente secondo la complessità, ad esempio, in un organismo biologico è possibile distinguere una gerarchia di atomi, molecole, cellule, tessuti, organi e sistemi organici. Tali gerarchie di oloni sono chiamate *olocrazie*.

Ciascuno strato di un'olocrazia nasconde la propria complessità allo strato soprastante. *Guardando in basso*, un olone è qualcosa di completo e unico, un tutto. *Guardando in alto*, un olone è una componente elementare, una parte. Ogni strato dell'olocrazia contiene molti oloni, ad esempio atomi (idrogeno, carbonio, ossigeno, etc.), cellule (nervi, muscoli, globuli, etc.).

Guardando all'interno, un olone fornisce servizi ad altri oloni. *Guardando all'esterno*, un olone richiede servizi ad altri oloni. Le olocrazie sono sistemi aperti; non ci sono oloni "foglia" o "vertice" in assoluto, eccetto quelli identificati come tali per convenienza interpretativa. Con tale caratteristica, sistemi semplici possono evolvere in sistemi complessi.

Oloni singoli sono di conseguenza rappresentati da quattro caratteristiche:

1. La loro struttura interna di interazione (interazioni interne possono formare pattern caratteristici).
2. Un'aggregazione auto-assertiva di oloni subordinati.
3. Una tendenza a integrarsi rispetto agli oloni superiori.
4. Relazioni con i loro oloni di pari grado.

Molti sistemi complessi sono organizzati in olocrazie le quali nascondono la complessità negli strati più bassi, fornendo livelli maggiori d'astrazione negli strati più alti delle proprie strutture. Tale situazione è coerente con la semantica dell'aggregazione.

L'*aggregazione* fornisce la separazione degli interessi, permette a ciascuna classe di restare incapsulata e di focalizzarsi su un comportamento specifico (collaborazioni e servizi) in maniera che sia slegata dall'implementazione delle sue classi superiori (come invece non è nella generalizzazione). Allo stesso tempo, l'aggregazione permette spostamenti liberi tra gli strati durante l'esecuzione.

Il bilanciamento tra integrazione e indipendenza di oggetti (oloni) è raggiunto per mezzo del requisito che gli oggetti dell'aggregazione devono "rispettare l'interfaccia di ogni altro oggetto" (Gamma *et al.*, 1995). L'incapsulamento non è rotto poiché gli oggetti comunicano solo attraverso le proprie interfacce. L'evoluzione del sistema è agevolata poiché la comunicazione tra oggetti non è codificata nell'implementazione attraverso meccanismi simili all'ereditarietà.

Strutturalmente l'aggregazione può rappresentare grandi quantità d'oggetti raggruppandoli in vari insiemi e stabilendo tra questi delle relazioni tutto-parte. Funzionalmente, l'aggregazione permette agli oggetti (oloni) di guardare "in alto" e "in basso". Comunque, l'aggregazione non permette di rappresentare la necessaria interoperabilità tra oloni di pari rango: tale salto strutturale e funzionale può essere colmato con successo per mezzo delle relazioni di generalizzazione e associazione.

Nell'approccio suggerito in Maciaszek *et al.* (1996a) e (1996b), l'aggregazione fornisce una soluzione "verticale" e la generalizzazione una soluzione "orizzontale" allo sviluppo d'applicazioni a oggetti. L'aggregazione diviene un concetto di rappresentazione dominante che determina la struttura globale del sistema. Tale struttura potrebbe essere ulteriormente formalizzata fornendo un insieme di pattern di progetto (Gamma *et al.*, 1995) che supportino specificamente l'approccio a oloni e utilizzino le quattro tipologie d'aggregazione. Chiaramente tutto questo è un campo aperto di indagine.

Sommario

Questo capitolo completa la discussione sull'analisi dei requisiti. È stato analizzato il supporto della tecnologia a oggetti allo sviluppo di grandi sistemi. Il capitolo è tecnicamente complesso ma offre una visione approfondita della tecnologia a oggetti che non si trova facilmente in altri libri sull'analisi e il progetto dei sistemi, e che ha cercato di evidenziare importanti debolezze e lati oscuri della tecnologia stessa.

Gli *stereotipi* (da non confondersi con *vincoli*, *note* e *tag*) sono la principale tecnica UML a supporto dell'estensione. Permettono di modellare caratteristiche che vanno oltre quelle predefinite in UML. A causa della natura del capitolo, gli stereotipi sono stati largamente usati.

La *visibilità* pubblica e privata, introdotta nei precedenti capitoli, dà solo un supporto di base all'importante nozione d'*incapsulamento*. La *visibilità protetta* permette il controllo dell'incapsulamento all'interno di strutture d'ereditarietà. La *visibilità a livello di classe* (opposta alla visibilità di singoli attributi e operazioni) è un altro importante concetto legato all'ereditarietà. La nozione di classe o operazioni *amiche* permette di "rompere" l'incapsulamento per trattare speciali situazioni.

UML offre altri svariati concetti per migliorare l'espressività dei modelli delle classi, quali *attributi derivati*, *associazioni derivate* e *associazioni qualificate*. Uno degli aspetti più intriganti della modellazione delle classi è la scelta tra *classe d'associazione* e *classe materializzata*.

I moderni sistemi software sono molto complessi. È importante che i loro modelli rendano più semplice e riducano questa intrinseca complessità quando possibile. Forse il meccanismo più importante per trattare la complessità del software è la *stratificazione gerarchica* delle architetture di sistema. Una corretta strutturazione delle classi in *package*, organizzati secondo l'*approccio* BCE, è un importante obiettivo architetturale.

Il concetto di *generalizzazione ed ereditarietà* è un'arma a doppio taglio nella modellazione di sistema. Da un lato, agevola il riuso del software e migliora espressività, comprensibilità e astrazione dei modelli di sistema. Dall'altro, se non usato correttamente, ha un potenziale di autodistruzione per tutti questi benefici.

Il concetto di *aggregazione e delegazione* è un'importante alternativa di rappresentazione alla generalizzazione e all'ereditarietà. La *delegazione e i sistemi prototipo* hanno un beneficio aggiuntivo di sostegno alle strutture architetturali gerarchiche. L'astrazione *olone* fornisce un'interessante intuizione su come sistemi complessi potrebbero o dovrebbero essere costruiti.

Domande

D1 Che cos'è un profilo UML? Fornire un esempio.

D2 Talvolta, una classe può solo istanziare oggetti immutabili, cioè oggetti che non cambiano dopo la loro creazione. Come tale requisito può essere rappresentato in UML?

D3 Spiegare la differenza tra un vincolo e una nota.

D4 L'incapsulamento è equivalente alla visibilità? Spiegare.

D5 La visibilità delle proprietà ereditate in una classe derivata dipende dal livello di visibilità dato alla classe base nella dichiarazione della classe derivata. Cosa accade se la classe base è dichiarata come privata? Quali sono le conseguenze sul resto del modello? Fornire un esempio.

D6 Il concetto di amica si applica a una classe o a un'operazione. Spiegare la differenza e fornire un esempio (non presente nel testo) dove l'uso di un'amica potrebbe essere desiderabile.

D7 Quali sono i benefici a livello di modellazione dell'informazione derivata?

D8 Quando una classe materializzata dovrebbe sostituire una classe d'associazione? Fornire un esempio (diverso da quelli presenti nel testo).

D9 Qual è la complessità di una rete con nove classi (misurata come il numero di possibili connessioni tra queste classi)? Disegnare una gerarchia con quattro strati per queste nove classi e valutare la riduzione di complessità che può essere raggiunta.

D10 Si supponga che un modello delle classi per un'applicazione bancaria contenga una classe chiamata `CalcoloInteresse`. Da quale package BCE potrebbe provenire tale classe? Spiegare.

D11 Che cos'è il principio di sostituibilità? Spiegare.

D12 Spiegare la differenza tra l'ereditarietà dell'interfaccia e l'ereditarietà dell'implementazione.

D13 Che cos'è il problema della classe base fragile? Quali sono le principali ragioni che rendono "fragili" le classi base?

D14 Spiegare la differenza tra l'aggregazione "possiede in modo esclusivo" e "possiede". Quali sono i vantaggi modellistici che si traggono distinguendo tra queste due tipologie d'aggregazione?

D15 Confrontare l'ereditarietà e la delegazione. Quali sono le similitudini e quali le differenze?

Esercizi

E1 Riferendosi alla Figura 2.13 (Sezione 2.1.3.2), si supponga che un insegnante il quale gestisce un corso offerto debba anche insegnare nel corso stesso. Modificare il diagramma nella Figura 2.13 per modellare questo requisito.

E2 Riferendosi alle Figure 2.17 (Sezione 2.1.5) e 2.18 (Sezione 2.1.5.1), combinare le due figure in un singolo modello delle classi.

Progettare la visibilità nel modello delle classi. Spiegare.

Quale sarebbe la visibilità degli attributi ereditati nella classe `Impiegato` e nella classe `Manager`? Spiegare.

E3 Immatricolazioni universitarie – Con riferimento all'Esempio 4.9 (Sezione 4.2.3.3) valutare se è possibile introdurre dell'informazione derivata nel modello di Figura 4.6. Se ciò è possibile, modificare il diagramma.

E4 Riferendosi alla Figura 2.15 (Sezione 2.1.3.4), si supponga che il sistema debba monitorare le valutazioni degli studenti nei diversi corsi offerti dello stesso corso. Ciò a causa del vincolo che uno studente può fallire lo stesso corso solo tre volte (una quarta immatricolazione nel corso non è permessa).

Estendere il diagramma di Figura 2.15 per rappresentare tale vincolo, usando una classe materializzata. Rappresentare e/o spiegare qualunque ipotesi fatta.

E5 Con riferimento alla specifica del caso Valutazione delle Spese Pubblicitarie presentata negli Esercizi del Capitolo 3, progettare un diagramma dei package per tale sistema. Spiegare qualunque ipotesi fatta.

E6 Video Store - Riferendosi all'Esempio 4.10 (Sezione 4.2.4.3), ridisegnare il diagramma di Figura 4.7 usando relazioni di realizzazione.

E7 Video Store - Riferendosi all'Esempio 4.10 (Sezione 4.2.4.3), ridisegnare il diagramma di Figura 4.7 usando l'aggregazione al posto della generalizzazione. Spiegare vantaggi e svantaggi del nuovo modello.

Capitolo 6

Elementi di base del progetto di sistema

Questo capitolo è il seguito diretto delle idee e degli argomenti discussi nel capitolo precedente. La differenza è che la materia è ora discussa dal punto di vista del progetto. Argomenti meno rilevanti nell'analisi, come i dettagli tecnici delle collaborazioni tra oggetti, sono ora affrontati in profondità.

Il progetto di sistema comprende due principali elementi: il progetto architetturale e il progetto di dettaglio. Il progetto architetturale coinvolge l'organizzazione stratificata di classi e package, l'assegnazione dei processi alle risorse di calcolo, la gestione del riuso e dei componenti. Il progetto di dettaglio riguarda i modelli di collaborazione necessari per la realizzazione delle funzionalità di sistema specificate dai casi d'uso.

Seguendo il principio adottato nel Capitolo 2, gli elementi di base del progetto di sistema sono spiegati e allo stesso tempo illustrati tramite esempi, utilizzando l'applicazione Acquisti OnLine la cui analisi è stata sviluppata nel Capitolo 2.

6.1 Architettura software

Il progetto è un modello di basso livello di un'architettura di sistema e dei suoi comportamenti interni. Il *progetto* è costruito sulla piattaforma hardware/software d'implementazione del sistema. Nell'ottica di uno sviluppo iterativo e incrementale, i modelli d'analisi sono successivamente elaborati con l'aggiunta di dettagli tecnici e considerazioni hardware/software, attraverso cui il modello d'analisi diventa un modello di progetto.

La distinzione tra l'analisi e il progetto non è netta. È possibile discutere di argomenti molto tecnici senza riferirsi a specifiche soluzioni hardware/software. In tal senso, gran parte della discussione nel Capitolo 5 potrebbe essere classificata come progetto piuttosto che analisi.

La descrizione di un sistema in termini dei suoi moduli è chiamato *progetto architetturale*. Il progetto architetturale comprende decisioni sulle strategie di soluzione per i componenti client e server del sistema[1].

1. Saranno usati i termini *client* e *server* per indicare i componenti architetturali del sistema (da non confondere client con cliente che individua negli esempi la classe degli acquirenti, *N.d.T.*)

La descrizione dei comportamenti interni di ogni modulo (caso d'uso) è chiamato *progetto di dettaglio*. Il progetto di dettaglio sviluppa algoritmi e strutture dati per ciascun modulo. Tali algoritmi e strutture dati devono essere sviluppati tenendo conto dei vincoli imposti dalla sottostante piattaforma di implementazione.

6.1.1 Architettura distribuita

Il progetto architetturale riguarda sia la scelta di una *strategia di soluzione* sia la *modularizzazione* del sistema. La strategia di soluzione deve risolvere i problemi relativi al lato client e al lato server, come pure a qualunque "middleware" necessario per "legare" il client e il server. La decisione sui blocchi base (moduli) è solo parzialmente dipendente dalla strategia di soluzione scelta.

Client e server sono concetti logici (Bochenski, 1994). Un *client* è un processo che produce richieste verso il processo server. Il *server* è un processo che soddisfa le richieste dei client. Normalmente, i processi client e server sono eseguiti su computer distinti ma è perfettamente possibile implementare un sistema client/server su una singola macchina.

Tipicamente, il *processo client* è responsabile del controllo dell'informazione visualizzata sullo schermo dell'utente e della gestione degli eventi prodotti dall'utente. Il *processo server* è un processo che opera su un qualunque elaboratore gestendo, generalmente con un database, dati che potrebbero essere richiesti da un processo client.

L'architettura *client/server* (C/S) può essere estesa per rappresentare un sistema distribuito arbitrario. Qualunque computer con un database può essere un client in alcune transazioni e un server in altre transazioni. Connettere tali computer per mezzo di una rete di comunicazione produce un'*architettura distribuita d'elaborazione*, come mostrato in Figura 6.1.

In un sistema d'elaborazione distribuita il client può accedere a un qualunque numero di server. Tuttavia, se il client ha accesso a un solo server alla volta, in una singola richiesta potrebbe non essere possibile combinare dati di due o più database server. Se, viceversa, ciò è possibile allora l'architettura supporta un sistema di database distribuiti.

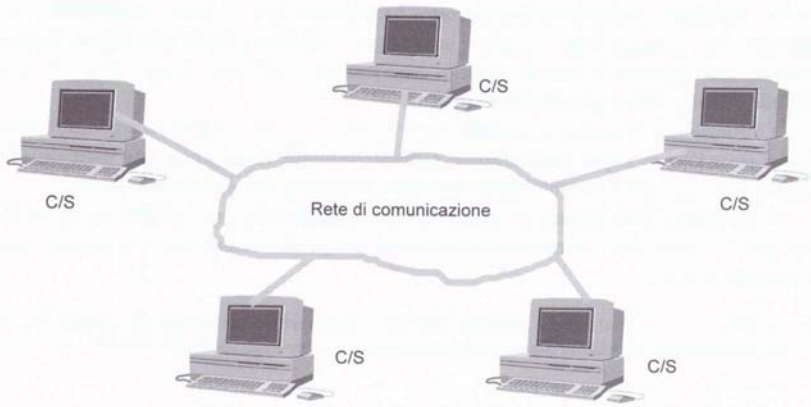


FIGURA 6.1 Architettura di un sistema d'elaborazione distribuita.

6.1.2 Architettura a tre strati

Nella Sezione 5.2.4 è stato descritto l'approccio BCE (boundary-control-entity) osservando che tale approccio è coerente con un'*architettura a tre strati* (talvolta chiamata *architettura a tre livelli*) in cui è introdotto uno strato intermedio distinto per la logica applicativa (control) tra il client (boundary) e il server (entity).

Come per i processi client e server, il *processo applicativo* è un concetto logico che non necessariamente deve essere supportato da un hardware separato. Il processo applicativo può egualmente essere eseguito su un nodo client o su un nodo server, cioè compilato nel processo client o server e implementato tramite tecniche come Dynamic Link Library (DLL), Application Programming Interface (API), Remote Procedure Calls (RPC), etc. (Next, 1996).

Quando la logica applicativa è compilata nel client si parla di *client "pesante"*, quando è compilata nel server si parla di *client "leggero"*. Architetture intermedie, in cui la logica applicativa è parzialmente compilata nel client e parzialmente nel server, sono tuttavia possibili.

La logica applicativa può essere anche portata su un nodo distinto di elaborazione, come mostrato in Figura 6.2. Questa è un'architettura a tre strati "pura". Le sue migliori caratteristiche sono buona flessibilità, possibilità di crescita, indipendenza dall'utente, disponibilità, e basso costo di miglioramento. Tuttavia, una tale architettura potrebbe avere un alto costo iniziale e potrebbe presentare problemi di efficienza (Umar, 1997).

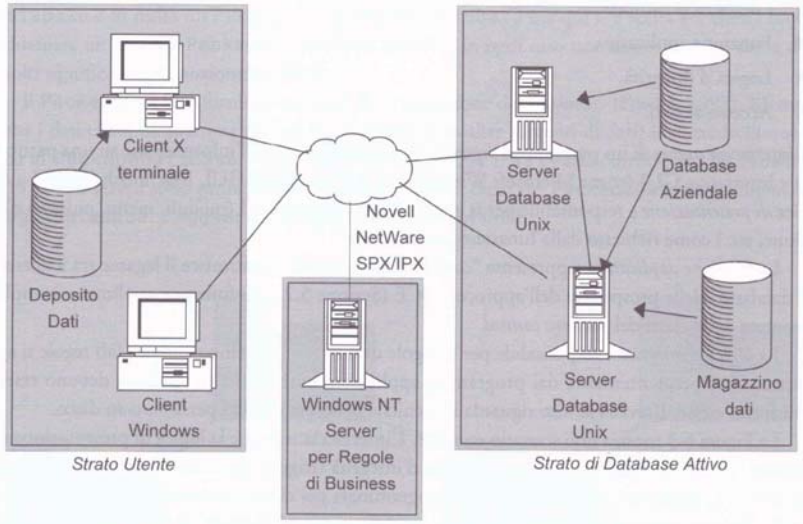


FIGURA 6.2 Architettura a tre strati.

Esercizi

E1 Riferendosi alla Figura 2.13 (Sezione 2.1.3.2), si supponga che un insegnante il quale gestisce un corso offerto debba anche insegnare nel corso stesso. Modificare il diagramma nella Figura 2.13 per modellare questo requisito.

E2 Riferendosi alle Figure 2.17 (Sezione 2.1.5) e 2.18 (Sezione 2.1.5.1), combinare le due figure in un singolo modello delle classi.

Progettare la visibilità nel modello delle classi. Spiegare.

Quale sarebbe la visibilità degli attributi ereditati nella classe `Impiegato` e nella classe `Manager`? Spiegare.

E3 Immatricolazioni universitarie – Con riferimento all'Esempio 4.9 (Sezione 4.2.3.3) valutare se è possibile introdurre dell'informazione derivata nel modello di Figura 4.6. Se ciò è possibile, modificare il diagramma.

E4 Riferendosi alla Figura 2.15 (Sezione 2.1.3.4), si supponga che il sistema debba monitorare le valutazioni degli studenti nei diversi corsi offerti dello stesso corso. Ciò a causa del vincolo che uno studente può fallire lo stesso corso solo tre volte (una quarta immatricolazione nel corso non è permessa).

Estendere il diagramma di Figura 2.15 per rappresentare tale vincolo, usando una classe materializzata. Rappresentare e/o spiegare qualunque ipotesi fatta.

E5 Con riferimento alla specifica del caso Valutazione delle Spese Pubblicitarie presentata negli Esercizi del Capitolo 3, progettare un diagramma dei package per tale sistema. Spiegare qualunque ipotesi fatta.

E6 Video Store - Riferendosi all'Esempio 4.10 (Sezione 4.2.4.3), ridisegnare il diagramma di Figura 4.7 usando relazioni di realizzazione.

E7 Video Store - Riferendosi all'Esempio 4.10 (Sezione 4.2.4.3), ridisegnare il diagramma di Figura 4.7 usando l'aggregazione al posto della generalizzazione. Spiegare vantaggi e svantaggi del nuovo modello.

La descrizione dei comportamenti interni di ogni modulo (caso d'uso) è chiamato *progetto di dettaglio*. Il progetto di dettaglio sviluppa algoritmi e strutture dati per ciascun modulo. Tali algoritmi e strutture dati devono essere sviluppati tenendo conto dei vincoli imposti dalla sottostante piattaforma di implementazione.

6.1.1 Architettura distribuita

Il progetto architetturale riguarda sia la scelta di una *strategia di soluzione* sia la *modularizzazione* del sistema. La strategia di soluzione deve risolvere i problemi relativi al lato client e al lato server, come pure a qualunque "middleware" necessario per "legare" il client e il server. La decisione sui blocchi base (moduli) è solo parzialmente dipendente dalla strategia di soluzione scelta.

Client e server sono concetti logici (Bochenski, 1994). Un *client* è un processo che produce richieste verso il processo server. Il *server* è un processo che soddisfa le richieste dei client. Normalmente, i processi client e server sono eseguiti su computer distinti ma è perfettamente possibile implementare un sistema client/server su una singola macchina.

Tipicamente, il *processo client* è responsabile del controllo dell'informazione visualizzata sullo schermo dell'utente e della gestione degli eventi prodotti dall'utente. Il *processo server* è un processo che opera su un qualunque elaboratore gestendo, generalmente con un database, dati che potrebbero essere richiesti da un processo client.

L'architettura *client/server* (C/S) può essere estesa per rappresentare un sistema distribuito arbitrario. Qualunque computer con un database può essere un client in alcune transazioni e un server in altre transazioni. Connettere tali computer per mezzo di una rete di comunicazione produce un'*architettura distribuita d'elaborazione*, come mostrato in Figura 6.1.

In un sistema d'elaborazione distribuita il client può accedere a un qualunque numero di server. Tuttavia, se il client ha accesso a un solo server alla volta, in una singola richiesta potrebbe non essere possibile combinare dati di due o più database server. Se, viceversa, ciò è possibile allora l'architettura supporta un sistema di database distribuiti.

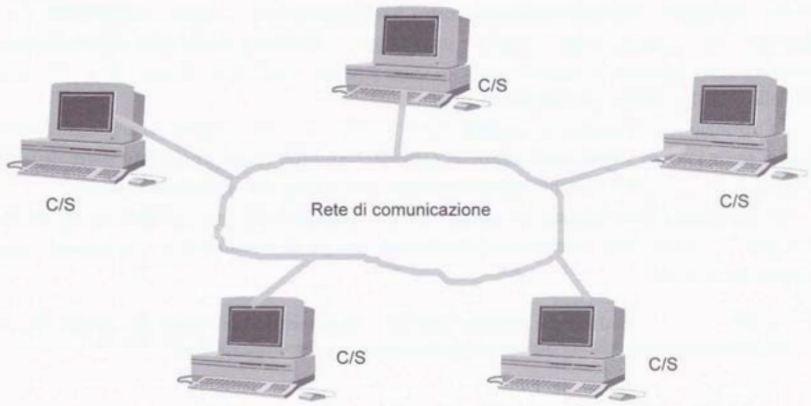


FIGURA 6.1 Architettura di un sistema d'elaborazione distribuita.

6.1.2 Architettura a tre strati

Nella Sezione 5.2.4 è stato descritto l'approccio BCE (boundary-control-entity) osservando che tale approccio è coerente con un'*architettura a tre strati* (talvolta chiamata *architettura a tre livelli*) in cui è introdotto uno strato intermedio distinto per la logica applicativa (control) tra il client (boundary) e il server (entity).

Come per i processi client e server, il *processo applicativo* è un concetto logico che non necessariamente deve essere supportato da un hardware separato. Il processo applicativo può egualmente essere eseguito su un nodo client o su un nodo server, cioè compilato nel processo client o server e implementato tramite tecniche come Dynamic Link Library (DLL), Application Programming Interface (API), Remote Procedure Calls (RPC), etc. (Next, 1996).

Quando la logica applicativa è compilata nel client si parla di *client "pesante"*, quando è compilata nel server si parla di *client "leggero"*. Architetture intermedie, in cui la logica applicativa è parzialmente compilata nel client e parzialmente nel server, sono tuttavia possibili.

La logica applicativa può essere anche portata su un nodo distinto di elaborazione, come mostrato in Figura 6.2. Questa è un'architettura a tre strati "pura". Le sue migliori caratteristiche sono buona flessibilità, possibilità di crescita, indipendenza dall'utente, disponibilità, e basso costo di miglioramento. Tuttavia, una tale architettura potrebbe avere un alto costo iniziale e potrebbe presentare problemi di efficienza (Umar, 1997).

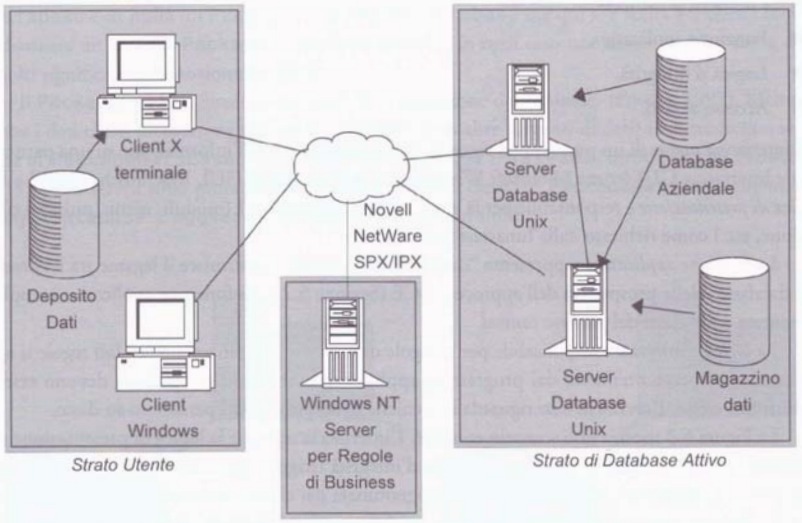


FIGURA 6.2 Architettura a tre strati.

6.1.3 Programmazione del database

Indipendentemente da dove la logica applicativa risiede, il programma (client) interagisce con il database (server) per ottenere informazioni da visualizzare e/o da manipolare ad opera dell'utente.

I database moderni possono essere programmati, cioè è possibile preparare delle procedure e inserirle in modo permanente nei database stessi (*procedure memorizzate*). A questo punto possono essere invocate da un programma client (o da un'altra procedura memorizzata) per mezzo di una normale istruzione di chiamata.

Una speciale tipologia di procedura memorizzata, chiamata *trigger*, non può essere richiamata esplicitamente. Un trigger è eseguito automaticamente in seguito a un tentativo di modifica del database e può essere usato per implementare regole di business, valide a livello aziendale e tali che debbano essere soddisfatte in modo indipendente da programmi client (o da procedure memorizzate). Un trigger controlla l'integrità e la consistenza del database, non permettendo alle singole applicazioni di violare le regole di business implementate a livello di database tramite il trigger stesso.

6.1.3.1 Interazione applicazione - database

Un aspetto importante nello sviluppo del sistema riguarda la decisione di quali parti programmare nel client e quali nel database. Le parti "programmabili" da considerare comprendono:

- Interfaccia utente.
- Logica di presentazione.
- Funzione applicativa.
- Logica d'integrità.
- Accesso ai dati.

L'*interfaccia utente* di un programma riguarda la visualizzazione dell'informazione su una particolare interfaccia GUI (come Microsoft Windows GUI, Unix Motif GUI, Macintosh GUI). La *logica di presentazione* è responsabile per la gestione degli oggetti GUI (moduli, menu, pulsanti d'azione, etc.) come richiesto dalla funzione applicativa.

La *funzione applicativa* rappresenta "cosa" fa l'applicazione e costituisce il legame tra il client e il database. Nella prospettiva dell'approccio BCE (Sezione 5.2.4) la funzione applicativa è implementata nelle classi del *package control*.

La *logica d'integrità* è responsabile per le regole di business a livello aziendale. Tali regole si applicano indipendentemente dai programmi applicativi, cioè tutti i programmi devono essere conformi a esse. L'*accesso ai dati* riguarda le tecniche di accesso ai dati persistenti su disco.

La Figura 6.3 mostra uno scenario comune. L'interfaccia utente e la logica di presentazione risiedono sul client. L'accesso ai dati e la logica d'integrità (trigger) sono sotto la responsabilità del database. La funzione applicativa è spesso programmata nel client (sotto forma di interrogazioni SQL) durante la fase di sviluppo del sistema, ma è spostata nel database (come procedure memorizzate) nella versione finale del prodotto.

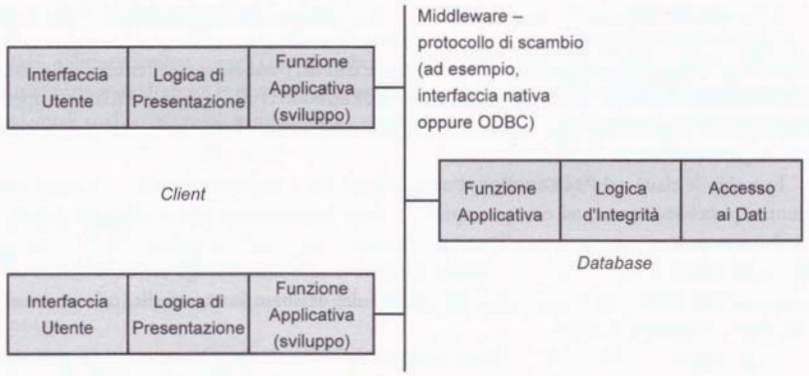


FIGURA 6.3 Interazione applicazione-database.

6.1.3.2 Approccio BCED

L'approccio *Boundary-Control-Entity-Database* (BCED) è un'estensione dell'approccio BCE (Sezione 5.2.4) che lascia le classi di sistema nel `PackageEntity` mentre le classi responsabili dell'estrazione dei dati dal database sono poste nel `PackageDatabase` (Figura 6.4). Il `PackageDatabase` è in realtà un `PackageInterfacciaDatabase` ma qui si è scelta l'etichetta breve (chiamare un package `PackageInterfaccia` non è in ogni caso una buona scelta a causa dei molti significati della nozione di interfaccia).

Il `PackageEntity` fornisce "un modello d'esecuzione del business" (Fowler, 1997). Memorizza i dati che il programma ottiene dal database (o da altre sorgenti di dati) invocando due servizi al `PackageDatabase`: `caricaMi(unOggetto)` e `salvaMi(unOggetto)` (Fowler, 1997). Tali servizi sono probabilmente implementati in classi del `PackageDatabase` chiamate `LeggiDatabase` e `AggiornaDatabase`, o con nomi simili.

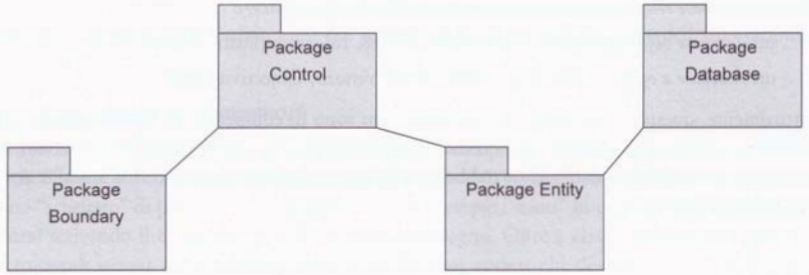


FIGURA 6.4 Strati BCED.

Il `PackageDatabase` fornisce un livello di indirezione tra l'applicazione e il database. La logica applicativa (`PackageControl`) è ora distinta dalle modifiche ai dati. Qualsiasi modifica alle strutture del database e ai protocolli per recuperare dati dal database sono effettuate nelle classi del `PackageDatabase`. Per quanto riguarda il `PackageControl`, restano invariati gli aspetti riguardanti la memorizzazione persistente e quella temporanea (bufferizzazione) dei dati nel `PackageEntity`.

In realtà, le classi nel `PackageDatabase` svolgono un insieme di altri servizi. I servizi aggiuntivi potrebbero essere, ad esempio, aprire e chiudere connessioni con il database, istruire il database quando una transazione effettua un'operazione di commit o rollback, determinare (in fase d'esecuzione) dei parametri per il database, gestire le autorizzazioni agli utenti, estrarre e memorizzare l'informazione di meta-dati sugli oggetti del database (come tabelle, colonne, viste, procedure memorizzate, indici, etc.).

La distinzione d'interessi introdotta dall'approccio BCED può essere "imposta" ai progettisti di sistema e ai programmatori attraverso la semplice tecnica di porre prefissi ai nomi delle classi: poiché ogni classe può provenire da un solo package, questa può essere nominata con una singola lettera di prefisso (B, C, D o E). Ad esempio, la classe etichettata `B_VistaScorrimentoAlbero` appartiene al `PackageBoundary` e la classe `D_AggiornaDatabase` al `PackageDatabase`.

6.1.3.3 Sistema software

Una strategia di soluzione per un client object-oriented si svilupperà tipicamente attorno ad ambienti software forniti di strumenti di gestione dell'interfaccia GUI; perciò una decisione iniziale riguarda la scelta di quali tecniche usare per la connessione al database, come:

- un'*interfaccia nativa del database*, fornita da molti linguaggi di 4° generazione (4GL, ad esempio PowerBuilder, Developer 2000, Delphi); oppure,
- un driver *ODBC* o *JDBC* verso il database (ad esempio, in Visual C++, Power J++).

Poiché "i dati restano mentre le applicazioni vanno e vengono" una strategia di soluzione per il server probabilmente influenzerà significativamente la strategia del client. Spesso, sono scelti gli ambienti di sviluppo del client messi a disposizione dai fornitori di database.

Le strategie per i server comprendono:

- un *database relazionale* (ad esempio Sybase, Oracle, Informix); o
- un *database object-relational* (ad esempio Oracle8, Informix/Illustra, UniSQL); o
- un *database a oggetti* (ad esempio ObjectStore, Versant, Objectivity/DB).

Attualmente, queste tre tecnologie per database non sono in competizione; sono eccellenti per problemi diversi e rispondono alle necessità di differenti domini applicativi. Come risultato, la decisione sul modello di database dovrebbe prendere in considerazione le necessità attuali e attese dell'applicazione.

6.1.4 Strategie di riuso

UML definisce il riuso come "l'uso di elaborati preesistenti" (Rumbaugh *et al.*, 1999). In precedenza sono state discusse le *tecniche per il riuso del software* nell'ambito del modello object-oriented, quali ereditarietà e delegazione. In questa sezione, l'interesse è per *strategie di riuso del software* implicanti anche il livello *granularità* a cui il riuso è fatto, granularità che può essere:

- la classe;
- il componente; o
- la strategia di risoluzione.

Associate alla granularità, vi sono tre strategie per il riuso (Coad *et al.*, 1995; Gamma *et al.*, 1995):

1. Librerie di classi (toolkit).
2. Framework (strutture configurabili).
3. Pattern di analisi e pattern di progetto.

6.1.4.1 Riuso di librerie di classi

Una *libreria di classi* enfatizza il *riuso del codice* a livello di classe. In questa tipologia di riuso il programmatore "colma i buchi" nel programma per mezzo di chiamate a classi *concrete* presenti in una libreria di classi. Il corpo principale del programma non è riusato ma è "scritto" dal programmatore.

Esistono due tipologie (livelli) di librerie (Page-Jones, 2000):

1. Librerie fondamentali.
2. Librerie architetturali.

Le *classi fondamentali* sono generalmente messe a disposizione dagli ambienti di programmazione a oggetti. Comprendono classi per l'implementazione di tipi dato primitivi (come `String`), tipi dato strutturati (come `Date`) e collezioni (come `Set`, `List` o `Index`).

Le *classi architetturali* sono normalmente disponibili come parti di un sistema software (il sistema operativo, il software di database, il software di interfaccia GUI, etc.). Ad esempio, quando si acquista un DBMS object-oriented quello che realmente si ottiene è una libreria architetturale che implementa le funzionalità attese dal sistema come persistenza, gestione delle transazioni e concorrenza.

6.1.4.2 Riuso di framework

Un *framework* enfatizza il *riuso progettuale* a livello di componenti (i componenti saranno discusse in Sezione 6.1.5). Mentre le librerie rendono disponibili classi di base, un framework fornisce uno "scheletro" di programma. Il programmatore "riempie i vuoti" in questo scheletro (lo configura) scrivendo il codice di cui il framework ha bisogno. Oltre a classi concrete (per gestire il framework stesso), un framework fornisce molte classi astratte che devono essere implementate (configurate) dal programmatore.

Un framework è una struttura (un software applicativo) configurabile. I migliori esempi di framework sono i sistemi ERPS quali SAP, PeopleSoft, Baan o J.D. Edwards (tuttavia, il riuso in tali sistemi non è generalmente basato su tecniche object-oriented).

Framework object-oriented per lo sviluppo dei sistemi informativi sono proposti all'interno di tecnologie a componenti distribuite quali CORBA, DCOM e EJB (Sezione 1.1.1). Sono noti come "*oggetti di business*", prodotti "distribuibili" per venire incontro a specifiche necessità applicative o di business. Ad esempio, un oggetto di business potrebbe essere un framework di contabilità con classi configurabili quali `Fattura` o `Cliente`.

Benché il framework sia un'attraente proposta per il riuso, può soffrire di alcune carenze. Forse la più significativa è che tali generiche soluzioni del tipo "minimo comun denominatore" sono spesso sub-ottimali o anche obsolete. Di conseguenza, il framework non dà un vantaggio competitivo a chi lo adotta e può creare un carico di manutenzione supplementare quando si cercano soluzioni innovative.

6.1.4.3 Riuso di pattern

I *pattern* enfatizzano il riuso fornendo idee ed esempi (strategie di risoluzione) per *collaborazioni* tra oggetti già note come modelli di buone pratiche di sviluppo in grado di condurre a soluzioni comprensibili e scalabili (le collaborazioni sono discusse nella Sezione 6.2). Il pattern può applicarsi alla fase di analisi o di progetto del ciclo di sviluppo (quindi, *pattern di analisi* e *pattern di progetto*).

Un pattern è una soluzione documentata di cui è stato dimostrato il buon comportamento in un certo numero di situazioni. Tali situazioni sono identificate e possono essere usate come indice per la ricerca di una soluzione durante lo sviluppo. Qualsiasi svantaggio noto o effetto collaterale di un pattern dovrebbe essere elencato per permettere allo sviluppatore di prendere una decisione ragionata.

Il riuso di pattern è sostanzialmente a livello di concetti, sebbene molti pattern di progetto contengano esempi di codice che il programmatore può riusare. La finalità di un *pattern di progetto* (ad esempio, Gamma *et al.*, 1995) è quella della collaborazione (Sezione 6.2), tipicamente più ampia di una classe ma più ristretta di un componente. La finalità di un *pattern di analisi* (ad esempio, Fowler, 1997) dipende dal livello d'astrazione al quale il pattern si applica.

6.1.5 Componenti

Un *componente* è una parte fisica del sistema (una parte dell'implementazione), cioè un programma software (Booch *et al.*, 1999; Lakos, 1996; Rumbaugh *et al.*, 1999; Szyperski, 1998). I componenti sono percepiti generalmente come parti eseguibili (EXE) del sistema. Ma un componente può anche essere una parte del sistema non direttamente eseguibile (ad esempio, un file di codice sorgente, un file di dati, DLL - Dynamic Link Library, o una procedura memorizzata di database).

Attualmente UML definisce cinque stereotipi standard per i componenti (Booch *et al.*, 1999):

1. *Eseguibile* (cioè il modulo è in forma binaria direttamente eseguibile).
2. *Libreria* (cioè una libreria, statica o dinamica, di oggetti).
3. *Tabella* (cioè una tabella di dati in un database).
4. *File* (cioè un codice sorgente o un insieme di dati).
5. *Documento* (cioè un testo leggibile).

Le caratteristiche di un componente sono (Rumbaugh *et al.*, 1999; Szyperski, 1998):

- è un'unità di allocazione, cioè di assegnazione del componente a un particolare nodo del sistema distribuito (in altri termini, un componente non può essere rilasciato parzialmente);
- è un'unità di composizione (cioè è documentato, definito e autosufficiente per essere inserito in altri componenti);
- non possiede uno stato persistente (e quindi non conserva memoria fra un'attivazione e la successiva);
- è una parte sostituibile di un sistema, cioè può essere rimpiazzato da altro componente purché abbia la medesima interfaccia;
- svolge un compito ben determinato;
- può essere inserito in altri componenti.

6.1.5.1 Notazione per i componenti

Un componente è rappresentato graficamente come un rettangolo sul cui lato sinistro vi sono due rettangoli più piccoli. Il nome identificatore del componente è scritto all'interno del rettangolo più grande. Una relazione di dipendenza può essere specificata tra due componenti qualunque.

La Figura 6.5 mostra il componente `FatturazioneEXE`, stereotipato come eseguibile.

6.1.5.2 Diagramma dei componenti

Un diagramma di componenti mostra dei componenti e le loro relazioni. I componenti possono essere legati per mezzo di *relazioni di dipendenza*. Un componente dipendente richiede i servizi del componente da cui dipende. I componenti possono essere legati anche con *relazioni di composizione*, cioè un componente può contenere un altro componente.

La Figura 6.6 mostra tre componenti. Il componente `FatturazioneEXE` dipende dagli altri due componenti. La natura di tali dipendenze non è specificata poiché le interfacce dei componenti non sono definite.

UML permette la rappresentazione delle interfacce di componenti usando la notazione col cerchio (*lollypop*, si veda Sezione 5.3.3). Se una dipendenza tra i componenti passa attraverso le interfacce, allora un componente con una data interfaccia può sostituire un qualunque altro componente con la stessa interfaccia (ammesso che ciò abbia senso).

6.1.5.3 Componenti e package

Il *package* è una parte logica del sistema (Sezione 5.2.3). A livello logico, ogni classe proviene da un solo package. A livello fisico, ogni classe è implementata da almeno un componente ed è pos-

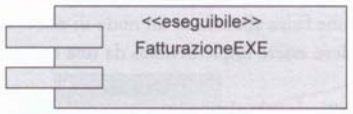


FIGURA 6.5 Notazione per i componenti.

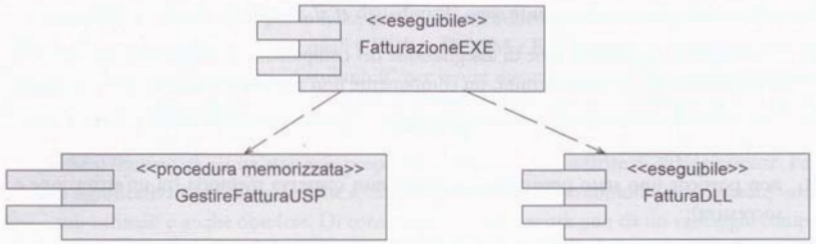


FIGURA 6.6 Diagramma dei componenti.

sibile che un componente implementi una sola classe. Le classi astratte che definiscono interfacce sono spesso implementate da uno o più componenti.

Il *package* è generalmente un'unità architetturale più ampia del componente. Tende a raggruppare classi in un modo *orizzontale*, per mezzo della "prossimità statica" delle classi nel dominio applicativo. Le *componenti* sono gruppi *verticali* di classi con "prossimità comportamentale" e potrebbero provenire da domini distinti ma contribuiscono a un singolo pezzo di un'attività di business, ad esempio, un caso d'uso.

Questa ortogonalità architetturale tra package e componenti rende difficile stabilire dipendenze tra essi. Una situazione frequente è che un package logico è basato su poche componenti fisiche.

Esempio 6.1 (Immatricolazioni universitarie)

Con riferimento ai package identificati nell'Esempio 5.11 (Sezione 5.2.3.2), considerare il package `Orario`. Si assuma che tale package sia implementato come un programma C++ contenente la logica dell'allocazione delle aule ai corsi. Il programma accede a un database per le informazioni sui corsi e sulle aule. Due procedure memorizzate sono implementate per fornire tali servizi al programma.

Disegnare un diagramma dei componenti che mostri le dipendenze tra il package e i componenti necessari.

La Figura 6.7 mostra il modello dei componenti. Sono identificati tre componenti: `AllocAulaEXE`, `AulaUSP` e `CorsoUSP`. Il package `Orario` dipende da questi componenti.

6.1.5.4 Componenti, classi e interfacce

Come le classi, i componenti possono realizzare interfacce. La distinzione è di due tipi. Primo, un componente è un'astrazione fisica allocata su un nodo di elaborazione. Una classe rappresenta un elemento logico che deve essere implementato da una componente per agire come astrazione fisica.

Secondo, un componente rivela solo alcune interfacce delle classi che contiene. Molte altre interfacce sono incapsulate dal componente, sono usate solo internamente dalle classi che collaborano e non sono visibili agli altri componenti esterni.

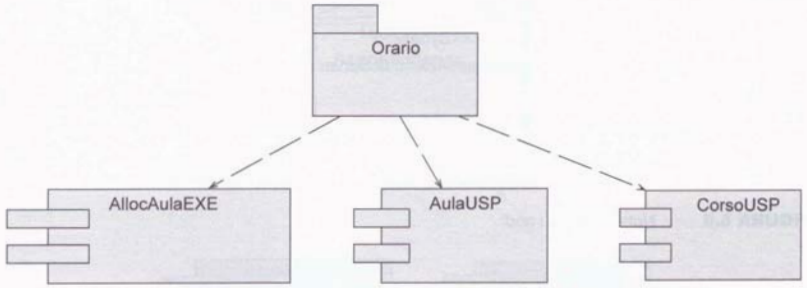


FIGURA 6.7 Package e componenti (Immatricolazioni universitarie).

L'interfaccia che un componente realizza potrebbe essere implementata in una classe distinta. Tale classe è chiamata *classe dominante* (Rumbaugh *et al.*, 1999). Poiché una classe dominante rappresenta l'interfaccia del componente, qualunque oggetto interno al componente è raggiungibile dalla classe dominante via legami di composizione. "Una classe dominante sussume l'interfaccia della componente" (Rumbaugh *et al.*, 1999).

Esempio 6.2 (Immatricolazioni universitarie)

Con riferimento ai tre componenti nell'Esempio 6.1 (Sezione 6.1.1.3.3), si assuma che il componente `AllocAulaEXE` inizi l'allocazione delle aule ai corsi fornendo al componente CorsoUSP l'identificazione del corso. A tal fine, corsoUSP realizza l'interfaccia Allocare.

Il componente CorsoUSP svolge il resto del lavoro richiedendo i dettagli dell'aula al componente AulaUSP. Per fornire un tale servizio, AulaUSP implementa l'interfaccia Prenotare.

La Figura 6.8 mostra il diagramma dei componenti corrispondente ai precedenti requisiti.

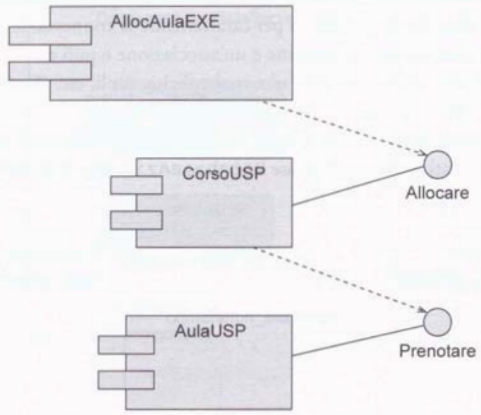


FIGURA 6.8 Mostrare le interfacce sul diagramma dei componenti (Immatricolazioni universitarie).

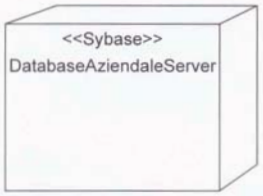


FIGURA 6.9 Notazione per i nodi.

6.1.6 Rilascio (deployment)

In UML, un'architettura a tre strati (come nella Figura 6.2) o qualunque altra architettura di sistema è rappresentata come un *diagramma di deployment*; ad esempio, il diagramma nella Figura 6.2 è un corretto diagramma di deployment UML nel quale le risorse d'elaborazione sono rappresentate con speciali icone.

Una singola risorsa d'elaborazione (un oggetto fisico a tempo d'esecuzione) è chiamata un *nodo*. Il nodo ha almeno una memoria e delle capacità d'elaborazione (ad esempio, il terminale X client nella Figura 6.2). Il nodo può anche essere un server database, in questo caso è un server attivo (cioè programmabile).

6.1.6.1 Notazione per il nodo

In UML, un *nodo* è rappresentato graficamente come un *cubo* e può essere stereotipato. A ogni nodo è assegnato un nome univoco.

La Figura 6.9 mostra il nodo `DatabaseAziendaleServer`. Lo stereotipo `«Sybase»` informa che il nodo è posto su un DBMS Sybase.

6.1.6.2 Diagramma di deployment

Un diagramma di deployment mostra i nodi e le loro relazioni. I nodi possono essere legati per mezzo di *relazioni di connessione*. Una relazione di connessione può essere etichettata per indicare il protocollo di rete usato (se applicabile) o per caratterizzare la connessione in qualche altro modo. In generale, una relazione di connessione è un'associazione e può essere rappresentata con le proprietà tipiche dell'associazione, quali grado, molteplicità, ruoli, etc. (Sezione 2.1.3).

La Figura 6.10 mostra due nodi in un diagramma di deployment. La relazione di connessione `scarica_nottetempo` specifica che il nodo `MagazzinoDatiServer` (su software Sybase) preleva ogni notte dei dati dal nodo database `DatabaseAziendaleServer`.

FIGURA 6.10 Diagramma di deployment.

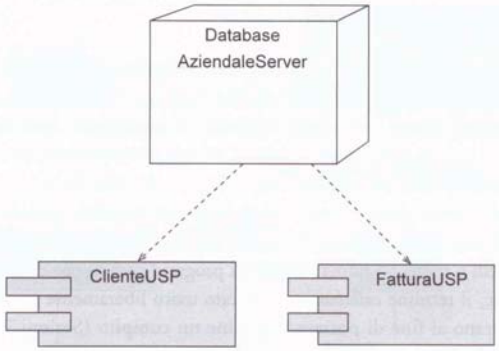


FIGURA 6.11 Un nodo e i suoi componenti.

6.1.6.3 Nodi e componenti

I nodi sono locazioni della rete in cui risiedono e sono eseguiti i componenti. Un nodo insieme ai propri componenti è talvolta chiamato *unità di distribuzione* (Booch *et al.*, 1999).

La Figura 6.11 mostra un nodo chiamato `DatabaseAziendaleServer`. Sul nodo si eseguono due procedure memorizzate rappresentate come componenti: `ClienteUSP` e `FatturaUSP`.

La Figura 6.12 è un modo alternativo di rappresentare il "contenimento" dei componenti in un nodo. Tale notazione può essere estesa in modo che l'intero diagramma dei componenti possa essere sovrapposto a un diagramma di deployment (Fowler e Scott, 2000).

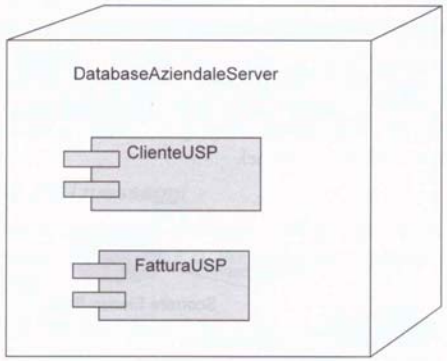


FIGURA 6.12 Un nodo e i suoi componenti.

6.2 Collaborazioni

Il progetto architetturale ha un impatto sul progetto dettagliato nel senso che determina la piattaforma hardware/software cui il progetto dettagliato deve essere conforme. A parte ciò, il progetto dettagliato è il seguito diretto dell'analisi. L'obiettivo è trasformare i modelli di analisi in documenti di dettaglio su cui i programmatori possono basarsi per implementare il sistema.

Nell'analisi, i modelli sono semplificati nascondendo i dettagli che interferiscono con la presentazione di un particolare punto di vista del sistema. Nel progetto, si tende a fare esattamente l'opposto. Si prende una parte dell'architettura del sistema alla volta e si aggiungono dettagli tecnici ai relativi modelli o si creano nuovi modelli di progetto a un livello d'astrazione inferiore.

Precedentemente, il termine *collaborazione* è stato usato liberamente per riferirsi a insiemi di oggetti che collaborano al fine di portare a termine un compito (Sezioni 2.1.1.2 e 2.1.2.2.1). UML usa il termine collaborazione con lo stesso intento e associa a esso specifici metodi di modellazione. In particolare, le collaborazioni sono usate per specificare la *realizzazione* di casi d'uso e operazioni (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999).

6.2.1 Notazione per la collaborazione

La notazione per la *collaborazione* è simile a quella per il caso d'uso: una collaborazione è rappresentata da un'ellisse con bordo tratteggiato.

La Figura 6.13 mostra due collaborazioni: `Scorrere Elenco Studenti` e `Aggiungere Studente a Corso Offerto` che realizzano (tramite *relazioni di realizzazione* stereotipate) il caso d'uso `Inserire Piano di Studi`.

Si noti che degli strumenti CASE potrebbero sostituire la rappresentazione esplicita delle relazioni di realizzazione, come nella Figura 6.13, con hyperlink tra il caso d'uso e i relativi modelli di collaborazione (Booch *et al.*, 1999); in tal caso, il progettista può navigare tra caso d'uso e modelli di collaborazione senza usare alcuna relazione di realizzazione.

6.2.2 Diagramma di collaborazione

Il diagramma di collaborazione è un tipo di modello d'interazione UML (Sezione 2.2.5). L'altro tipo è il diagramma di sequenza (Sezione 2.2.5.3). È preferibile usare i diagrammi di sequenza nell'analisi e i diagrammi di collaborazione nel progetto (vedere la successiva Sezione 6.2.3 per un confronto tra questi modelli d'interazione).

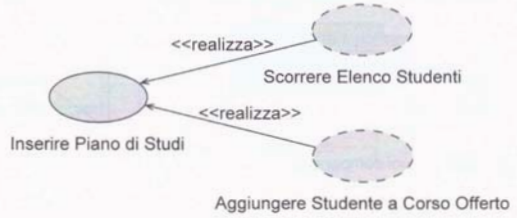


FIGURA 6.13 Notazione per le collaborazioni.

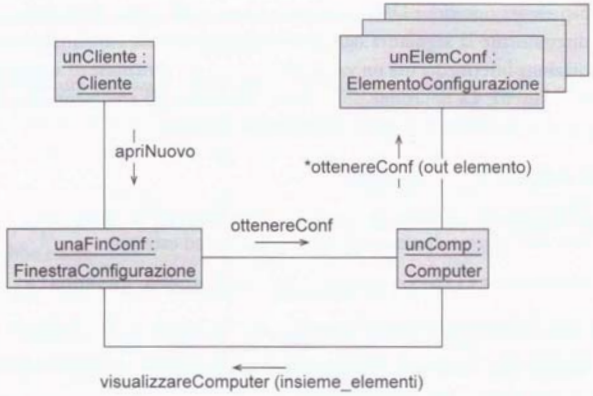


FIGURA 6.14 Diagramma delle collaborazioni per "Visualizzare Configurazione Attuale" (Acquisti OnLine).

Il diagramma di collaborazione nella Figura 6.14 corrisponde al diagramma di sequenza nella Figura 2.33 (Sezione 2.2.5.1). I numeri di sequenza, opzionali (e infatti in Figura 6.14 non sono riportati), rappresentano la sequenza temporale dei messaggi. In algoritmi complessi è difficile assegnare una significativa sequenza temporale ai messaggi e modelli aggiuntivi (quali diagrammi d'attività o pseudocodice) potrebbero essere necessari per esprimere le sequenze temporali.

Si noti che i nomi d'oggetti in un diagramma di collaborazione sono in realtà i *ruoli* che gli oggetti rivestono in una collaborazione. La notazione UML è:

```
nomeruolo:nomeclasse
```

Per essere precisi, un ruolo non è un oggetto. In UML, un ruolo è un "classificatore" rappresentante qualunque oggetto che potrebbe apparire in istanze distinte della collaborazione. Altresì, lo stesso oggetto potrebbe rivestire ruoli distinti in collaborazioni diverse. In generale, il nome ruolo potrebbe essere omesso ma il simbolo ":" prima del nome della classe deve essere indicato per evidenziare che una generica istanza della classe riveste un ruolo nella collaborazione (Rumbaugh *et al.*, 1999).

6.2.2.1 Notazione per i messaggi

La struttura di un messaggio corrisponde alla *segnatura* del corrispondente metodo dell'oggetto a cui il messaggio è indirizzato (Sezione 2.1.2.2). Per inviare un messaggio, l'oggetto mittente deve fornire (Page-Jones, 2000):

- L'*OID* dell'oggetto destinatario, che può essere contenuto in uno degli attributi del mittente (in una *variabile*, dal punto di vista della programmazione).
- Il nome dell'*operazione* (*metodo*) nell'oggetto destinatario.
- Eventualmente, gli *argomenti* attuali di ingresso (`in`) e di ritorno (`out`) per sostituire i corrispondenti argomenti formali nel metodo.

La Figura 6.15 presenta notazioni UML alternative per la definizione di messaggi. La notazione più in alto è direttamente la segnatura del metodo (per brevità, la parola chiave `in` può essere omessa). La notazione intermedia usa un valore di ritorno e un'istruzione di assegnazione al posto di un argomento d'uscita. La notazione più in basso sostituisce gli argomenti di ingresso e uscita con l'indicazione grafica (frecce con cerchietti) dei dati scambiati.

6.2.2.2 Struttura del messaggio

Nei messaggi di Figura 6.15 la variabile contente l'OID dell'oggetto destinatario potrebbe corrispondere al nome dell'oggetto destinatario sul diagramma, ad esempio:

```
unaBanca1.dammiPrestito(in somma_ric, out somma_assegnata)
```

Si noti la struttura del messaggio, prima l'oggetto e poi l'operazione, che enfatizza:

- il punto di vista object-oriented, interessato alla collaborazione tra oggetti piuttosto che alle chiamate di procedura (Page-Jones, 2000);
- la possibile natura polimorfa del messaggio, in cui l'operazione (`dammiPrestito`) potrebbe avere diverse implementazioni in classi diverse (classi a cui appartengono gli oggetti riferiti dalla variabile `unaBanca1`).

Gli *argomenti dei messaggi* nella Figura 6.15 sono dati. Ciò è accettabile nei comuni ambienti object-oriented ma è in contrasto con una visione rigorosa del paradigma object-oriented secondo la quale gli argomenti dei messaggi dovrebbero essere variabili contenenti OID (Page-Jones, 2000). Si supponga infatti, per un istante, che gli argomenti nel messaggio `dammiPrestito` siano oggetti della classe `Moneta`. In tal caso, i valori delle variabili `somma_ric` e `somma_assegnata` sarebbero OID (puntatori) alle celle di memoria contenenti l'ammontare, in dollari, dei due argomenti.

6.2.2.3 Tipi di messaggio

Un messaggio può essere inviato a un oggetto classe (Sezione 2.1.6) oppure a un oggetto istanza. Tipici messaggi inviati a un oggetto classe sono *costruttori* (per creare nuovi oggetti istanza) e *distruttori* (per distruggere oggetti istanza esistenti). Altri tipi di messaggi (sia per oggetti istanza sia

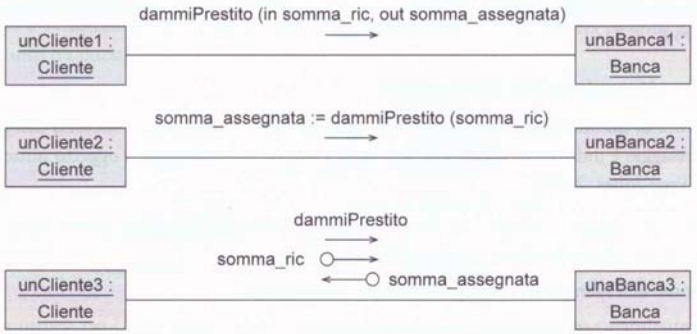


FIGURA 6.15 Notazioni alternative per i messaggi.

per oggetti classe) possono essere classificati in tre gruppi (i nomi alternativi tra parentesi si riferiscono alla nomenclatura proposta in Page-Jones (2000)):

- Messaggi di lettura (messaggi interrogativi, orientati al presente).
- Messaggi d'aggiornamento (messaggi informativi, orientati al passato).
- Messaggi di collaborazione (messaggi imperativi, orientati al futuro).

Un *messaggio di lettura* richiede che l'oggetto destinatario fornisca alcune informazioni, ad esempio i valori di propri attributi. È "orientato al presente" poiché tratta l'informazione attuale. Un esempio potrebbe essere:

```
unaBanca1.aperturaOre (in giornosettimana, out ore)
```

Un *messaggio d'aggiornamento* chiede che un oggetto destinatario aggiorni se stesso sulla base dell'informazione fornita nel messaggio. È "orientato al passato" poiché aggiorna un'informazione dell'oggetto già nota al mittente. Un esempio potrebbe essere:

```
unCliente1.nuovoPunteggioCredito (in punteggio_credito, data_effettiva)
```

Un *messaggio di collaborazione* chiede all'oggetto destinatario l'esecuzione di un'azione di supporto alla collaborazione. È "orientato al futuro" poiché è richiesto all'oggetto di svolgere un'azione al posto del mittente come parte di un più ampio compito da completare. Il messaggio `dammiPrestito` è un esempio di messaggio di collaborazione.

```
unaBanca1.dammiPrestito (in somma_ric, out somma_assegnata)
```

Per esaudire la richiesta, `unaBanca1` potrebbe, a sua volta, richiedere ad altri oggetti ulteriore collaborazione.

6.2.2.4 Riscrittura e sovraccarico

Progettando i modelli d'interazione è necessario tener presente che un metodo può essere riscritto o sovraccaricato. Questi due termini non sono equivalenti.

La riscrittura (*overriding*) di un metodo è stata spiegata nella Sezione 5.3.4.4.2 e costituisce la base del polimorfismo, specificando l'esistenza di più metodi con lo stesso nome in classi distinte (generalmente in sottoclassi della classe in cui il metodo è stato introdotto). Una differente implementazione dello stesso metodo potrebbe essere eseguita in funzione della classe dell'oggetto destinatario.

Il sovraccarico (*overloading*) di un metodo significa che ci possono essere più metodi con lo stesso nome ma nella stessa classe. Ad esempio, a parte il metodo `dammiPrestito` precedentemente definito, potrebbe esserci un altro metodo `dammiPrestito` nella classe `Banca` che potrebbe comprendere un argomento aggiuntivo per specificare il minimo prestito che un cliente si attende, come mostrato di seguito:

```
unaBanca1.dammiPrestito (in somma_ric, somma_minima, out somma_assegnata)
```

Il metodo `dammiPrestito` è ora sovraccaricato e l'implementazione che verrà eseguita dipende dalla segnatura del messaggio. Ne segue che la determinazione delle segnature dei messaggi è importante durante la fase di progetto.

FIGURA 6.16 Relazione d'aggregazione con ruoli.

6.2.2.5 Messaggi iterativi e tipi parametrici

Un *messaggio iterativo* è inviato ripetutamente a più oggetti della classe, cioè itera su più oggetti. Per indicare l'iterazione, UML usa una *marca d'iterazione*, un asterisco, davanti all'etichetta del messaggio (Figura 2.33 nella Sezione 2.2.5.1).

Gli oggetti sui quali un messaggio iterativo opera sono noti come *collezioni*. La collezione può essere un insieme, una lista (insieme ordinato) o un vettore di oggetti. In un diagramma di collaborazione, una collezione è mostrata come una pila di oggetti sovrapposti (Figura 6.14).

In senso object-oriented, una collezione (insieme, lista, etc.) è essa stessa una classe. Infatti, molti ambienti di programmazione object-oriented forniscono classi `Set`, `List`, `FixedArray`, `VarryingArray`, `BtreeIndex`, etc. Una classe collezione può allora essere usata per contenere oggetti di una classe definita dall'utente. Un tipico uso delle classi collezione è nell'implementazione delle relazioni di associazione o aggregazione con molteplicità "molti" (Sezione 2.1.3.2).

Si consideri la Figura 6.14 e il messaggio iterativo `*ottenereConf (out elemento)` da `unComp` a `unElemConf`. Il messaggio itera su una collezione di oggetti di `ElementoConfigurazione`. Tuttavia, il diagramma di collaborazione non rivela come la collezione è implementata. Questa informazione può essere derivata dal diagramma delle classi.

La Figura 6.16 è il diagramma delle classi che mostra la relazione d'aggregazione tra le classi `Computer` e `ElementoConfigurazione`. La Figura 6.17 rappresenta un diagramma delle classi a livello implementativo con i nomi dei ruoli convertiti in nomi d'attributi.

Come si vede dalla Figura 6.17, il tipo dell'attributo `ha_elem_conf` è `List<ElementoConfigurazione>`. La classe `ElementoConfigurazione` è il parametro per la classe collezione `List`. In generale, `List` può sostituire il proprio parametro formale con altre classi. Si dice che il tipo di `ha_elem_conf` è un *tipo parametrico*.

La definizione del tipo parametrico è essenziale per la comprensione dell'applicabilità di un messaggio iterativo. Il messaggio `*ottenereConf(out elemento)` itera su `List` di oggetti di `ElementoConfigurazione`. La lista costituisce il valore attuale dell'attributo `ha_elem_conf`. Per essere precisi, la lista contiene i valori OID di tutti gli oggetti di `ElementoConfigurazione` inclusi in un particolare oggetto `Computer`.

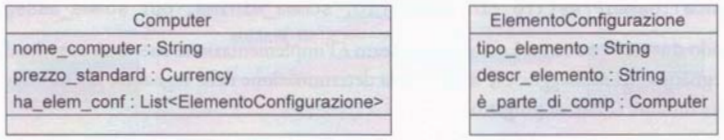


FIGURA 6.17 Relazione d'aggregazione implementata con attributi.

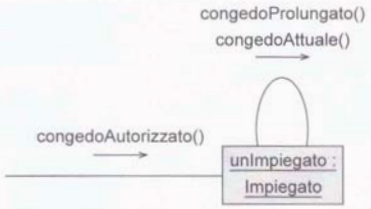


FIGURA 6.18 Messaggi self.

6.2.2.6 Messaggi "self"

Un oggetto può inviare un messaggio a se stesso. Un messaggio *self* specifica un'invocazione locale, cioè un metodo chiama un altro metodo nello stesso oggetto (il nome "self" è preso a prestito dall'ambiente Smalltalk, il corrispondente termine C++ e Java è *this*).

Un messaggio self potrebbe occorrere in una sequenza di messaggi di collaborazione (Sezione 6.2.2.3) che agiscono sulla medesima classe prima di passare il controllo a un'altra classe, oppure se il controllo ritorna al mittente e il messaggio successivo invoca ancora un metodo sul mittente.

La Figura 6.18 fornisce un esempio di due messaggi self (`congedoAttuale` e `congedoProlungato`) attivati come risultato della richiesta di calcolare `congedoAutorizzato` dell'impiegato.

In alcune situazioni può essere utile inviare, in un messaggio, l'OID del mittente al destinatario; in questo caso, self può considerarsi un oggetto costante (contenente l'OID del mittente) e può essere passato come argomento di un messaggio, cioè:

```
nomemessaggio (self)
```

Inviare l'OID del mittente all'oggetto destinatario non è normalmente necessario in un *messaggio sincrono* (durante il quale il mittente attende il ritorno del controllo dal destinatario prima di continuare l'esecuzione). Il ritorno del controllo dell'esecuzione è automatico, almeno in tutti i principali ambienti di programmazione object-oriented.

6.2.2.7 Messaggi asincroni

In un *messaggio asincrono*, un oggetto mittente non deve attendere che il destinatario finisca il proprio lavoro prima di continuare l'esecuzione. Lo scambio asincrono di messaggi presuppone l'esecuzione concorrente di diversi cammini di computazione nel programma.

UML usa la notazione a mezza freccia per indicare un messaggio asincrono. L'oggetto `io` nella Figura 6.19 genera due computazioni indipendenti inviando due messaggi asincroni agli oggetti `miaCaffettiera` e `miaRadio`.

Lo scambio asincrono di messaggi è frequente nelle applicazioni ingegneristiche e in tempo-reale. Le tipiche applicazioni di business si basano principalmente sullo scambio sincrono di messaggi.

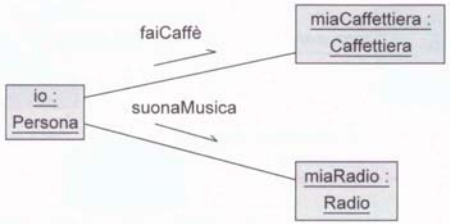


FIGURA 6.19 Scambio asincrono di messaggi.

6.2.2.8 Chiamate di ritorno

Le chiamate di ritorno sono state precedentemente menzionate nel contesto dell'ereditarietà (Sezione 5.3.4.4.2). Una chiamata di *ritorno* è una chiamata al mittente del messaggio. Un tipico uso delle chiamate di ritorno è nello *scambio asincrono di messaggi* dove il mittente richiede al destinatario di essere informato sul completamento di un'azione: il destinatario deve inviare indietro un messaggio asincrono al mittente.

La Figura 6.20 estende il diagramma di collaborazione di Figura 6.19 con una chiamata di ritorno da `miaCaffettiera` a `io`. Dopo tutto `io` sto aspettando il caffè e non intendo essere interrotto una volta che il caffè è pronto.

La chiamata di ritorno è una tecnica utile ma intrinsecamente difficile da usare nella programmazione. Affinché le chiamate di ritorno svolgano correttamente il lavoro, il destinatario deve in qualche modo osservare lo stato del mittente (e le sue variazioni) prima che la chiamata di ritorno sia inviata (inviare il messaggio asincrono `caffèPronto` a `io` quando sono già uscito non è molto utile!).

6.2.3 Diagrammi di sequenza e diagrammi di collaborazione

I diagrammi di sequenza e di collaborazione, che enfatizzano aspetti differenti dell'interazione tra oggetti, sono equivalenti nel senso che possono essere automaticamente convertiti uno nell'altro.

I diagrammi di sequenza mettono l'enfasi sulla sequenza temporale dei messaggi tra oggetti ma sono poco utili nella rappresentazione di cammini alternativi di messaggi (per cui sono più adatti i diagrammi d'attività). Sono anche poco pratici nel rappresentare collaborazioni con molti oggetti (sebbene un'attenta disposizione delle linee di vita degli oggetti possa frequentemente migliorare la leggibilità del diagramma stesso).

Un diagramma di collaborazione mostra esplicitamente le relazioni statiche tra oggetti lungo cui i messaggi possono propagarsi. Come risultato, i diagrammi di collaborazione sono più espressivi per visualizzare, ad esempio, messaggi polimorfi. La forma dei diagrammi di collaborazione permette di mostrare più oggetti nella stessa area grafica. I messaggi possono essere completamente specificati e annotati.

Nella fase di analisi è preferibile usare i diagrammi di sequenza, riservando i diagrammi di collaborazione per i modelli di progetto. Entrambi i diagrammi possono essere completati con diagrammi d'attività, specificati al livello di astrazione corrispondente a quello del diagramma di sequenza o di collaborazione cui si riferiscono.

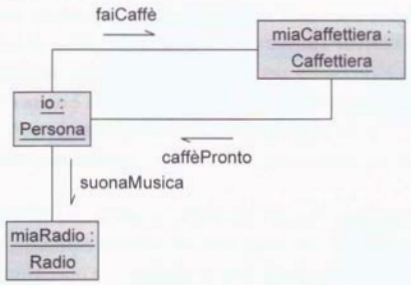


FIGURA 6.20 Chiamate di ritorno nello scambio asincrono di messaggi.

6.2.4 Realizzazione dei casi d'uso

Le *collaborazioni* stanno al progetto come i casi d'uso stanno all'analisi (i casi d'uso guidano l'analisi così come le collaborazioni guidano il progetto). I casi d'uso sono *realizzati* per mezzo di collaborazioni.

La collaborazione ha una parte strutturale e una comportamentale. La *parte strutturale* rappresenta l'aspetto statico della collaborazione. È rappresentata da un sottoinsieme del diagramma delle classi comprendente le classi che devono collaborare. Tale diagramma è poi *elaborato* (rispetto alla versione ottenuta dall'analisi) con dettagli implementativi; in particolare, sono specificate le segnature delle operazioni delle classi.

La *parte comportamentale* rappresenta la dinamica, cioè *come* gli elementi statici collaborano tramite *interazioni* (Sezione 2.2.5). Oltre ai diagrammi di sequenza (Sezione 2.2.5.3), i diagrammi di collaborazione sono spesso usati (durante il progetto) per la specifica comportamentale delle collaborazioni. In pratica, gli aspetti strutturali e comportamentali delle collaborazioni sono sviluppati concorrentemente.

6.2.4.1 Aspetti strutturali della collaborazione

Esempio 6.3 (Immatricolazioni universitarie)

Con riferimento all'Esempio 4.9 (Sezione 4.2.3.3.) e all'Esempio 4.17 (Sezione 4.3.3.3), si consideri il caso d'uso "Inserire Piano di Studio". Il caso d'uso gestisce l'immatricolazione di studenti nei corsi offerti.

In questo esempio, si assuma che il caso d'uso controlli (prima che uno studente possa essere immatricolato) se lo studente ha pagato le tasse e se soddisfa i prerequisiti. Estendere il diagramma delle classi nella Figura 4.6 (Esempio 4.9) per rappresentare l'aspetto strutturale della collaborazione nello scenario descritto.

Per risolvere il problema bisogna rielaborare il diagramma delle classi prodotto durante l'analisi introducendo le nuove classi (con relativi attributi e operazioni) necessarie per realizzare la collaborazione. Il diagramma di sequenza per il caso d'uso sviluppato durante l'analisi fornisce un'utile guida (Figura 4.14, Sezione 4.3.3.3).

Come mostrato in Figura 6.21, sono state aggiunte poche classi al diagramma. La classe *boundary* `FinestraInserimentoPiano` è ottenuta dal diagramma di sequenza nella Figura 4.14. La classe `PrereqCorso` è necessaria per stabilire una relazione prerequisito molti-a-molti tra i corsi. La classe `Giudizio` è aggiunta per completezza ma non è usata nel caso d'uso considerato. Precedenti giudizi sugli studenti sono mantenuti in `InformazioneAccademica`. `Giudizio` contiene i voti (punteggio) e i giudizi che uno studente raggiunge nei corsi offerti attualmente seguiti.

Gli attributi nelle classi precedentemente definite nell'analisi non sono stati modificati. La nuove classi contengono attributi che dovrebbero essere auto-esplicativi. Le principali aggiunte al diagramma delle classi sono le definizioni delle operazioni. Le operazioni sono direttamente derivate dal diagramma di collaborazione relativo all'aspetto comportamentale della collaborazione (si veda la sezione successiva).

6.2.4.2 *Aspetti comportamentali della collaborazione*

Esempio 6.4 (Immatricolazioni universitarie)

Con riferimento all'Esempio 4.9 (Sezione 4.2.3.3) e all'Esempio 4.17 (Sezione 4.3.3.3), si crei un diagramma di collaborazione per l'aspetto comportamentale della collaborazione specificata nell'Esempio 4.9. Il diagramma di collaborazione dovrebbe essere un'elaborazione del diagramma di sequenza in Figura 4.14 (Sezione 4.17).

La Figura 6.22 fornisce una soluzione dell'esempio. In confronto al diagramma di sequenza (Figura 4.14), sono stati aggiunti due oggetti: `unaInformazioneAccademica` e `unPrereqCorso`. Questi due oggetti sono necessari per gestire il controllo delle tasse e dei prerequisiti dello studente prima che questi possa essere registrato in un `CorsoOfferto`.

6.2.5 Realizzazione di operazioni

Le *collaborazioni* possono anche essere usate per rappresentare la realizzazione di operazioni complesse (Booch *et al.*, 1999), cioè operazioni che richiedono la collaborazione di diversi oggetti. Operazioni più semplici, limitate a una o due classi, sono meglio rappresentate come diagrammi d'attività, in particolare se comprendono algoritmi elaborati. Lo stesso principio si applica a operazioni semplici che siano componenti di un'operazione complessa rappresentata come collaborazione.

Infine, le operazioni più semplici sono rappresentate con *pseudocodice* o possono essere direttamente implementate nel codice. Il codice che realizza tali operazioni può anche essere incluso nel modello UML come una nota o un documento allegato.

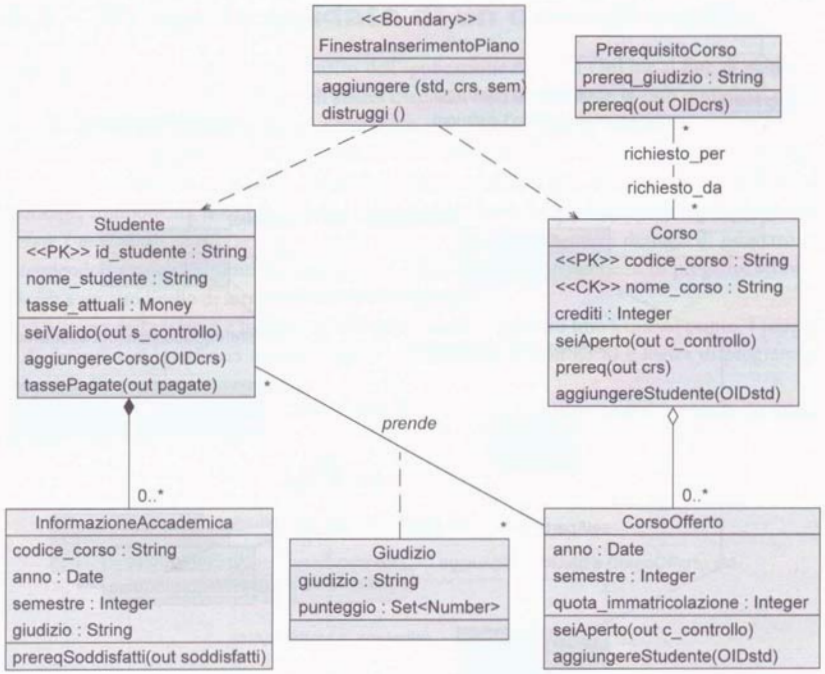


FIGURA 6.21 Aspetto strutturale della collaborazione (Immatricolazioni universitarie).

Esempio 6.5 (Immatricolazioni universitarie)

Con riferimento agli Esempi 4.18 (Sezione 4.3.4.3), 6.3 (Sezione 6.2.4.1) e 6.4 (Sezione 6.2.4.2), si consideri il messaggio `aggiungereStudente(OIDstd)` nella Figura 6.22 (Esempio 6.4). Il compito è progettare un diagramma d'attività per realizzare l'operazione `CorsoOfferto.aggiungereStudente`. L'operazione dovrebbe aggiungere uno studente all'attributo `CorsoOfferto.std`. L'attributo ha un tipo parametrico `list<Studente>` (Figura 4.15, Esempio 4.18).

Prima che lo studente sia aggiunto al corso offerto, il sistema controlla se il corso offerto è ancora aperto. Ciò è necessario, sebbene lo stesso controllo sia stato fatto appena prima per generare l'evento `aggiungereStudente(OIDstd)` (Figura 6.22). In un DBMS con molti utenti, lo stato del corso offerto (aperto o chiuso) potrebbe essere stato modificato nel frattempo.

Una volta che lo studente sia stato aggiunto a `:CorsoOfferto`, l'algoritmo deve creare il legame opposto in `:Studente`. L'oggetto `:Studente` deve essere aggiornato per puntare all'oggetto `:CorsoOfferto`.

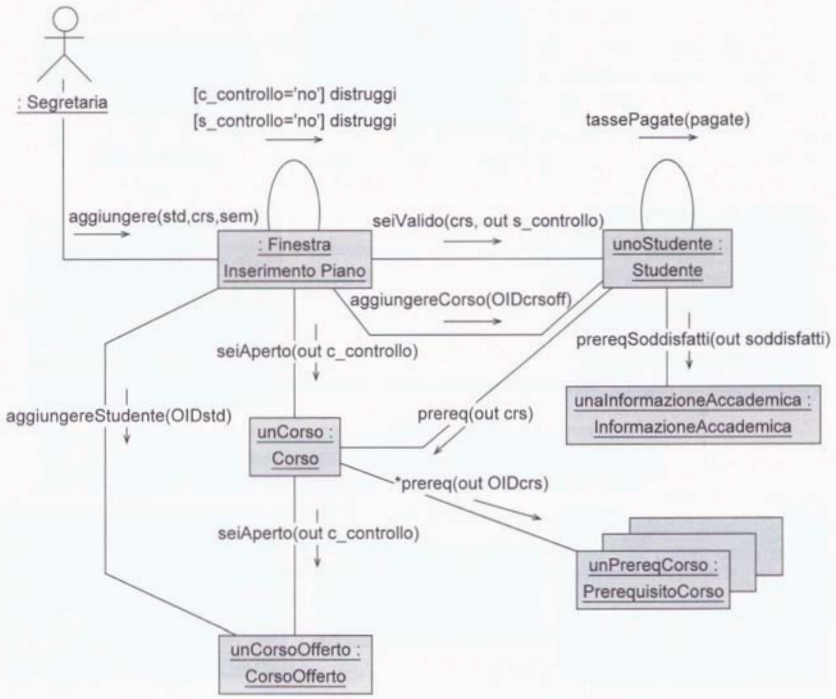


FIGURA 6.22 Aspetto comportamentale della collaborazione (Immatricolazioni universitarie).

La Figura 6.23 fornisce una soluzione all'esempio. Il diagramma d'attività è riferito alla classe `CorsoOfferto`, tranne per la necessità di comunicare con la classe `Studente` onde mantenere l'integrità referenziale dei legami.

L'*evento* (messaggio) `aggiungereStudente` "attiva" l'*attività* (metodo) `aggiungereStudente`. Il messaggio interno `seiAperto` attiva il metodo `seiAperto`. L'attività `seiAperto` controlla la quota di studenti che si possono immatricolare rispetto al numero di studenti già immatricolati (i controlli sono fatti per mezzo di *azioni* interne: *do/*...).

Se la decisione è negativa, lo *stato* dell'oggetto `CorsoOfferto` diventa `Chiuso`, altrimenti è `Aperto`. L'esecuzione del diagramma d'attività termina se lo stato è `Chiuso`.

Se lo stato è `Aperto`, l'attività `aggiungereStudente` continua. L'*azione* `/aggiungere OIDstd a CorsoOfferto.std` risulta nel nuovo stato `Studente Aggiunto`. In tale stato, un'altra azione `/aggiungere OIDcrsoff a Studente.crsoff` invoca l'attività `Studente.impostareCrsOff`.

L'ultima attività identifica un nuovo metodo da includere nella classe `Studente` (Figura 6.21); specifica anche la necessità di inserire in `Studente` l'attributo chiamato `crsoff` di tipo `list<Corso>`.

6.3 Progetto guidato di un caso di studio

Nel Capitolo 2, è stata presentata l'analisi dell'applicazione Acquisti OnLine al fine di guidare il lettore attraverso i concetti di base dell'analisi usando il caso di studio. In questo capitolo si estende lo stesso esempio guidato facendo riferimento ai concetti di *base del progetto*.

L'obiettivo di questo esempio è duplice. Primo, *rafforzare i nuovi concetti* (introdotti nei Capitoli 5 e 6) usandoli in un singolo dominio applicativo, analizzando sia i problemi architetturali (package, componenti, nodi) sia i problemi di progetto dettagliato (collaborazioni). Secondo, *elaborare i modelli di analisi* sviluppati nella Sezione 2.2, aggiungendo i dettagli di progetto ed estendendo i modelli esistenti per trasformarli effettivamente in documenti di progetto, sebbene ancora a un alto livello di astrazione.

Lo scopo dell'esempio è limitato ai concetti introdotti nel libro fino a questo punto. I progetti di dettaglio di interfaccia utente (Capitolo 7), database (Capitolo 8) e logica di programma (Capitolo 9) non sono qui considerati.

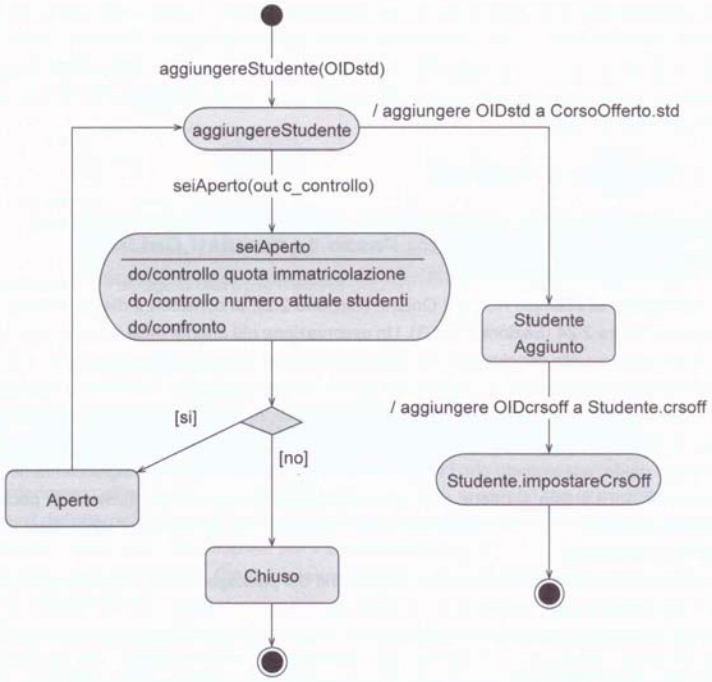


FIGURA 6.23 Diagramma d'attività per rappresentare la realizzazione di un'operazione (Immatricolazioni universitarie).

6.3.1 Progetto dei package

Nella Sezione 5.2.3 si è osservato che i package potrebbero raggruppare sia classi sia altri elementi di rappresentazione: più comunemente questo avviene per i casi d'uso. Quando usati nell'*analisi*, i package sono generalmente applicati per identificare i principali gruppi di casi d'uso. Nel *progetto*, i package sono usati nel modo più tradizionale, cioè per raggruppare classi. Conseguentemente, si può distinguere tra:

- Package di casi d'uso.
- Package di classi.

L'uso di package è giustificato nella strutturazione di *grandi sistemi*. Sistemi ridotti potrebbero essere compresi e gestiti senza package poiché i casi d'uso possono già fornire un ragionevole livello di modularità. Occasionalmente, i package possono essere considerati "elementi architetturali" per trattare una probabile estensione del sistema oppure per tener conto che l'insieme iniziale di casi d'uso e classi potrebbe crescere con l'avanzamento dello sviluppo e del progetto dettagliato.

Un *package di classi*, in particolare, evolve durante il progetto incorporando le classi *boundary*, *control* e *database* oltre alle classi *entity* che costituiscono l'obiettivo della fase di analisi (le altre tipologie di classi non sono in realtà di interesse se non in fase di progetto). Coerentemente, il modello dei *package di classi* ha un ruolo dominante nel progetto. Il modello dei *package di casi d'uso* è generalmente usato solo nei passi iniziali del progetto, per essere sostituito ai fini pratici dal modello dei package di classi.

6.3.1.1 Package di casi d'uso

Esempio di progetto: Passo 1 (Acquisti OnLine)

Con riferimento all'esempio Acquisti OnLine (Sezione 2.2), si consideri il diagramma dei casi d'uso nella Figura 2.24 (Sezione 2.2.2.3). Un'osservazione più attenta delle specifiche di Acquisti OnLine conduce alla conclusione che il modello dei casi d'uso non è completo.

Studiando e documentando i casi d'uso già identificati, potremmo certamente scoprire più casi d'uso. I nuovi casi d'uso potrebbero essere estensioni o inclusioni di casi d'uso esistenti (Sezione 4.3.1.2) o potrebbero essere casi d'uso trascurati nell'analisi iniziale dei requisiti.

In ogni modo, assumendo che i nuovi casi d'uso possano aumentare significativamente la complessità del sistema si ritiene utile la strutturazione dei casi d'uso esistenti in package. I package potrebbero essere considerati come "contenitori funzionali" per l'allocazione di casi d'uso nuovi o esistenti.

In questo esempio, si costruisce un diagramma dei package di casi d'uso per contenere i casi d'uso esistenti e per prendere in considerazione la futura crescita.

La soluzione all'esempio è presentata nella Figura 6.24 la quale mostra come i casi d'uso di Figura 2.24 siano stati assegnati a cinque package. In Figura 6.24 sono anche mostrate le relazioni di dipendenza tra i package; si osservi che le relazioni riflettono l'ordine in cui un cliente che vuole acquistare un computer usa le pagine e i moduli web (online).

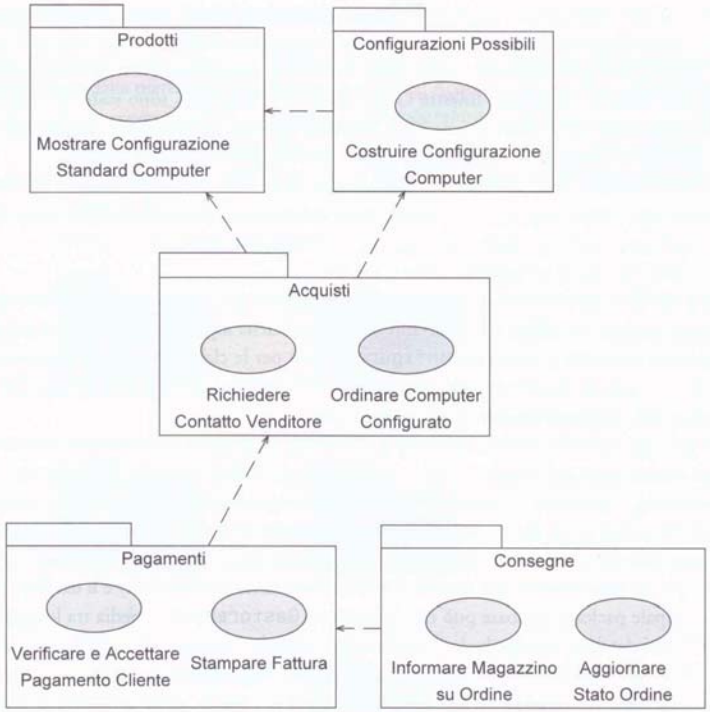


FIGURA 6.24 Package di casi d'uso (Acquisti OnLine).

6.3.1.2 Package di classi

Esempio di progetto: Passo 2 (Acquisti OnLine)

Con riferimento all'esempio Acquisti OnLine (Sezione 2.2), si osservi che la maggior parte delle classi definite nella Sezione 2.2 rappresenta gli oggetti persistenti del database ("oggetti di business"). Un modello più completo per il sistema richiederebbe che le classi del programma applicativo siano identificate. Ciò può essere fatto durante il progetto delle collaborazioni, che sarà trattato più avanti.

Sebbene non siano ancora note le classi del programma applicativo, è possibile ragionare sui package che raggrupperebbero le classi in unità coerenti con l'approccio BCED (Sezione 6.1.3.2). Il compito in questo esempio è di ragionare sui package possibili in Acquisti OnLine e le loro principali dipendenze.

Il modo migliore di analizzare l'esempio è di immaginare quali siano i compiti da eseguire per accettare l'ordine di un cliente per un certo computer configurato. Un'ovvia osservazione è separare le due distinte funzioni del sistema: la configurazione del computer e l'inserimento dell'ordine. Queste due funzioni richiedono finestre GUI distinte. Nell'esempio, sono state identificate due sole *classi boundary* ma è certo che ciascuna funzione richiede un insieme di oggetti GUI, rendendo ragionevole introdurre due package: `ConfigurazioneGUI` e `OrdineGUI`.

Dal "lato business" dello spettro delle classi, è stato identificato l'insieme di classi riportato nel diagramma delle classi (Figura 2.31). Queste *classi persistenti del database* possono essere naturalmente raggruppate in tre package *entity*: `Clienti`, `Computer` e `Ordini` (quest'ultimo include anche le classi `Fattura` e `Pagamento`).

Nel modello mancano ancora i package che raggruppano le classi *control*, cioè classi che legano le classi *boundary* ed *entity*, e supportano l'esecuzione della logica applicativa. È necessario un package (sarà chiamato `ProcessoConfigurazione`) per le classi utili alla configurazione dei computer e al calcolo dei prezzi delle configurazioni. È anche necessario un package, `InserimentoOrdini`, responsabile della gestione degli ordini.

I tre package *entity* (`Clienti`, `Computer` e `Ordini`) rappresentano la struttura della memoria a tempo d'esecuzione per le classi persistenti del database. Essi forniscono una rappresentazione object-oriented e transitoria di classi memorizzate in strutture di database generalmente non object-oriented (ad esempio, tabelle relazionali) e, in un qualunque istante di esecuzione del programma, contengono solo una frazione del database. Conseguentemente, c'è la necessità di classi, contenute in uno o più *package database*, che stabiliscano un'interfaccia fra le classi entity e il database stesso.

Il principale package database può essere chiamato `GestoreDati` e media tra le classi entity e le tabelle del database ogniqualvolta l'applicazione necessiti l'accesso o la modifica al contenuto del database. Esso dipende da due altri package database chiamati `Connessione` e `Schema`. Le classi nel package `Connessione` sono responsabili per la gestione delle connessioni, autorizzazioni e transazioni. Il package `Schema` contiene le informazioni attuali sullo schema degli oggetti del database (tabelle, colonne, procedure memorizzate, trigger, etc.). L'applicazione può inizialmente istanziare gli oggetti di `Schema` in modo da verificare l'esistenza di tali strutture nel database prima di tentare l'accesso al database stesso (in tal modo, sarebbe possibile per l'applicazione verificare, ad esempio, la presenza di una procedura memorizzata *prima* di invocarla).

La Figura 6.25 mostra i package identificati e le principali dipendenze tra essi. Si noti che senza conoscere tutte le classi e i loro legami di comunicazione, l'insieme di dipendenze tra i package è ancora provvisorio. Il principio di base è che i package boundary dipendono dai package control, i quali, a loro volta, dipendono dai package entity. Infine, i package entity dipendono dai package database.

Si noti inoltre che le dipendenze tra i package *non sono transitive* (Fowler e Scott, 2000). Ad esempio, nella Figura 6.25 una modifica all'interno del package `Clienti` potrebbe richiedere modifiche in `ProcessoConfigurazione` e `InserimentoOrdini` ma non necessariamente in `ConfigurazioneGUI` e `OrdineGUI`. La non-transitività delle dipendenze è (ovviamente) un elemento essenziale nella riduzione di complessità connessa alla strutturazione gerarchica delle classi e dei package (Sezione 5.2.2).

I *cicli* nella struttura delle dipendenze sono un fattore di complessità. Come principio, i cicli dovrebbero essere evitati (Lakos, 1996) ma, in pratica, essi sono talvolta difficili da eliminare, in particolare tra package allo stesso strato gerarchico (Fowler e Scott, 2000).

6.3.2 Progetto dei componenti

I *componenti* sono le parti fisiche del sistema. Il progetto dei componenti, di conseguenza, non può prescindere dalla piattaforma implementativa scelta: Acquisti OnLine deve essere un'applicazione web con un server database (altri aspetti della piattaforma implementativa non sono stati ancora considerati). Poiché in questo libro non si intende consigliare soluzioni specifiche o suggerire fornitori di software, si ragionerà su una possibile piattaforma implementativa a un livello ragionevole di generalità.

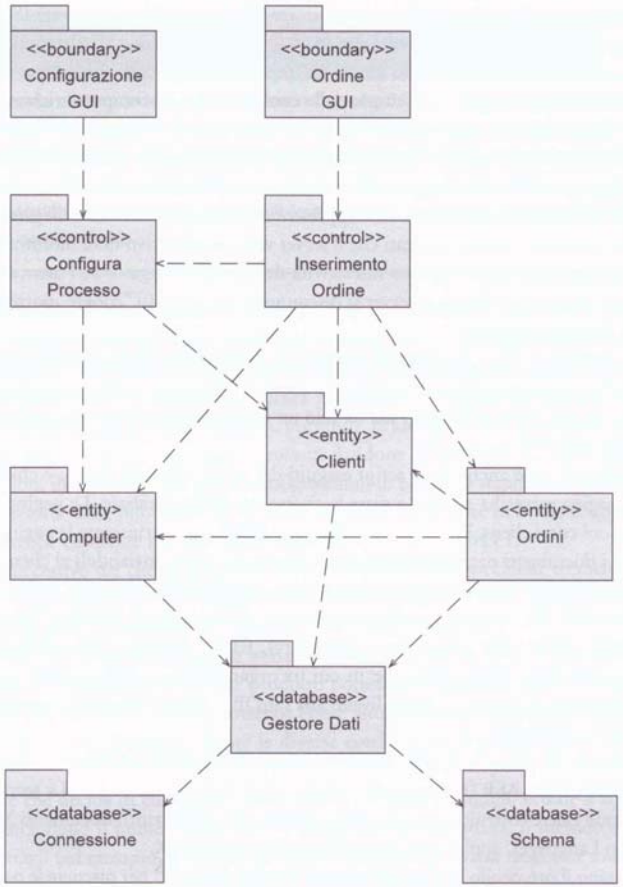


FIGURA 6.25 Package di classi (Acquisti OnLine).

6.3.2.1 Implementazione di applicazioni web

"Un'applicazione web è un sistema web che permette ai propri utenti di eseguire una logica di business con un browser web" (Conallen, 2000). La *logica di business* può risiedere sul server e/o sul client. Un'*applicazione web* è di conseguenza un tipo di sistema Client/Server (Sezione 6.1.1) con un sito web.

Un *browser* Internet presenta, sullo schermo del computer, le *pagine web* fornite dal *server web*. Le pagine web possono essere *statiche* (non modificabili) oppure *dinamiche*. Una pagina web può anche essere un *modulo* che l'utente riempie. È possibile suddividere lo schermo in aree (*frames*) in modo che l'utente possa vedere più pagine web allo stesso tempo.

Un'applicazione web può includere un *server applicativo* (application server) (Sezione 6.1.2) per gestire la logica applicativa e monitorare lo stato dell'applicazione. Nelle applicazioni come Acquisti OnLine il monitoraggio dello stato è un'importante attività di sistema per tenere traccia delle azioni dei clienti collegati, ad esempio delle configurazioni di computer richieste. Un cliente potrebbe decidere di acquistare una particolare configurazione in qualunque momento durante una *sessione in linea* e il sistema deve legare l'ordine di acquisto alla configurazione.

Una tecnica tipica di monitoraggio dello stato è memorizzare un indicatore (*cookie*) nel browser, cioè una breve stringa di caratteri che rappresenta lo stato di un utente collegato. A causa del possibile alto numero di utenti collegati che il server web o applicativo deve monitorare, un *timeout di sessione* potrebbe essere imposto sull'attività dell'utente collegato. Se l'utente non è attivo per 15 minuti (il timeout tipico), il server si disconnette dal client (il "cookie" potrebbe essere rimosso dalla macchina client).

Script e applet sono usati per rendere dinamica la *pagina client*. Uno *script* (ad esempio scritto in JavaScript) è un programma interpretato dal browser. Un'*applet* è una componente compilata che si esegue nel contesto del browser ma ha solo un limitato accesso alle altre risorse del computer client (per ragioni di sicurezza).

Una pagina web può anche avere script eseguiti dal server. Una tale pagina è chiamata *pagina server*. Una pagina server ha accesso a tutte le risorse del server database. Le pagine server gestiscono le sessioni con i client, impostano "cookie" nel browser e costruiscono le pagine client (cioè costruiscono i documenti pagina dagli oggetti business del server, inviandoli al client).

Librerie standard di accesso ai dati sono usate per permettere a script nelle pagine server di accedere al database. Le tipiche tecnologie impiegate comprendono ODBC (Open Database Connectivity), JDBC (Java Database Connectivity), RDO (Remote Data Objects) e ActiveX Data Objects (ADO). Nella situazione in cui un'organizzazione standardizza un particolare DBMS, le chiamate a funzioni di basso livello alla LibDB (Libreria Database) permettono un accesso più diretto al database.

La tecnologia del server web può basarsi su pagine HTML (HyperText Markup Language) contenenti script quali ASP (Active Server Pages) o JSP (Java Server Pages). La tecnologia per le pagine web può essere costruita su script client (JavaScript o VBScript), documenti XML (eXtensible Markup Language), applet Java, controlli JavaBean o ActiveX.

I client usano il protocollo HTTP (Hypertext Transfer Protocol) per ottenere le pagine web dal server web. La pagina può contenere script o moduli DLL (Dynamic Link Library) compilati e direttamente eseguibili, ad esempio ISAPI (Internet Server Application Programming Interface), NSAPI (Netscape Server Application Programming Interface), CGI (Common Gateway Interface) o Java servlet (Conallen, 2000).

Il *cookie* è un meccanismo primitivo per mantenere la connessione tra il client e il server in un sistema Internet che, altrimenti, sarebbe intrinsecamente *non-connesso*. Meccanismi più sofisticati di connessione trasformano Internet in un *sistema a oggetti distribuiti* in cui gli oggetti sono univocamente identificati con OID (Sezione 2.1.1.3) e comunicano tramite OID con altri oggetti. I meccanismi principali sono implementati nelle proposte CORBA, DCOM e EJB (Sezione 1.1.1): in tali tecnologie, gli oggetti possono comunicare senza usare HTTP o passando attraverso un server web (Conallen, 2000).

6.3.2.2 Diagramma dei componenti

Esempio di progetto: Passo 3 (Acquisti OnLine)

Con riferimento alla precedente discussione in Sezione 6.3.2.1, si proponga un diagramma dei componenti per Acquisti OnLine. Osservare che un componente è un'unità funzionale "compatta", con interfacce ben specificate in modo da poter essere trattato come una parte sostituibile del sistema. Poiché la piattaforma d'implementazione per Acquisti OnLine non è specificata, l'identificazione di componenti più ridotti (quali librerie, procedure memorizzate, etc.) non è possibile a questo stadio.

Un modo di condurre questo passo è considerare la sequenza tipica di accesso alle pagine web da parte di un cliente in linea che vuole acquistare un computer. Le linee guida possono essere ottenute dall'analisi dei casi d'uso (Sezione 2.2.1) e dai package di casi d'uso (Sezione 6.3.1.1).

La prima pagina web che un cliente in linea visiterebbe è la pagina del fornitore che elenca le categorie di prodotto (quali server, desktop, portatili), evidenzia le ultime offerte e sconti, fornisce i link alle pagine web che riportano i prodotti dandone di ciascuno una breve descrizione. Le brevi descrizioni includono prezzi per le configurazioni di prodotto standard. Questa parte del sistema è interessata dalla pubblicità dei prodotti verso un acquirente in linea. È un'unità funzionale che potrebbe costituire il componente chiamato `ListaProdotti`.

Il passo successivo del cliente potrebbe essere la richiesta delle specifiche tecniche per un prodotto scelto. Ciò include una visualizzazione del prodotto da differenti angolature mediante una pagina web a sé stante (componente `VisualizzazioneProdotti`).

Assumendo che le precedenti pagine web abbiano attirato l'attenzione di un cliente su un prodotto, differenti configurazioni per tale prodotto potrebbero essere richieste onde soddisfare necessità specifiche del cliente. Questo potrebbe essere fatto attraverso pagine web dinamiche in cui costruire e visualizzare interattivamente le diverse configurazioni (componente `Configurazione`).

Al cliente che decide di comprare il prodotto è presentato il modulo ordine d'acquisto. I dati da inserire includono il nome, l'indirizzo per la spedizione e la fattura, il metodo di pagamento scelto e i dettagli (ad esempio, il numero di carta di credito) da inviare mediante un protocollo di trasferimento sicuro. Questo potrebbe essere il quarto componente `Acquisto`.

L'ultimo componente identificabile in questo esempio ha a che vedere con l'evasione dell'ordine e la sua "tracciatura", cioè la possibilità di vedere lo stato dell'ordine su una pagina web (dopo che il numero del cliente e il numero d'ordine sono stati inseriti). Questo componente può essere chiamato `TracciaturaOrdini`.

I cinque componenti identificati sono mostrati nella Figura 6.26. È interessante notare la somiglianza di questo diagramma con quello dei package di casi d'uso di Figura 6.24. Questo non è sorprendente poiché sia i package di casi d'uso sia i componenti sono unità funzionali. I package di casi d'uso sono unità "logiche" di funzionalità (non hanno un impatto diretto sul modo in cui il sistema sarà costruito e non ha senso introdurli se il sistema è semplice). I componenti sono unità "fisiche" indipendenti di implementazione e devono essere progettate con cura anche per sistemi ridotti.

6.3.3 Progetto di rilascio

La natura "non connessa" di Internet (Sezione 6.3.2.1) rende il rilascio di un'applicazione web significativamente più difficile di quello di un'applicazione C/S.

Per iniziare, il server web deve costituire il punto di instradamento tra tutti i browser client e il database. Se la gestione della sessione non può essere affrontata in modo soddisfacente con la tecnologia *cookie* allora sono necessari *oggetti distribuiti*. Il rilascio di oggetti distribuiti richiederebbe un elemento architetturale distinto, un server applicativo, da collocare tra il server web e il server database.

Il progetto del rilascio deve affrontare anche problemi di sicurezza. I protocolli di trasferimento sicuro e cifratura creano richieste aggiuntive al progetto di rilascio. È anche necessaria un'attenta pianificazione rispetto ai carichi di rete, connessioni Internet, backup, etc.

6.3.3.1 *Rilascio di applicazioni web*

Un'architettura di rilascio capace di supportare applicazioni web sofisticate include quattro strati di nodi elaborativi:

1. Client con browser.
2. Server web.
3. Server applicativo.
4. Server database.

Il browser del *nodo client* può essere usato per visualizzare pagine statiche o dinamiche. Pagine con script e applet possono essere scaricate ed eseguite all'interno del browser. Funzionalità aggiuntive possono essere fornite al browser client per mezzo di oggetti quali controlli ActiveX o JavaBeans. Codice applicativo in esecuzione sul nodo client (esternamente al browser) potrebbe soddisfare altri requisiti dell'interfaccia.

Il *server web* gestisce le richieste di pagina dal browser e dinamicamente genera pagine e codice per l'esecuzione e la visualizzazione sul client. Il server web tratta anche la configurazione e parametrizzazione della sessione con l'utente.

Il *server applicativo* gestisce la logica di business ed è indispensabile quando oggetti distribuiti sono coinvolti nell'implementazione. I componenti di business rendono pubbliche le proprie interfacce verso altri nodi tramite tecnologie quali CORBA, DCOM o EJB. I componenti di business possono incapsulare dati persistenti memorizzati in un database, probabilmente un database relazionale. Questi comunicano con il *server database* via protocolli di connessione quali JDBC o ODBC. Il nodo database gestisce la memorizzazione dati e l'accesso multi-utente.

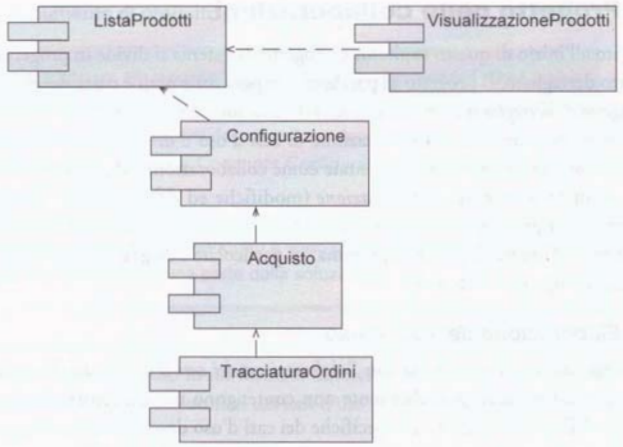


FIGURA 6.26 Diagramma dei componenti (Acquisti OnLine).

6.3.3.2 *Diagramma di deployment*

Esempio di progetto: Passo 4 (Acquisti OnLine)

Con riferimento alla discussione nella Sezione 6.3.3.1 si proponga un diagramma di deployment per Acquisti OnLine. In particolare valutare la necessità in Acquisti OnLine di un server applicativo.

Come mostrato nella Figura 6.27, l'applicazione Acquisti OnLine può essere messa in produzione senza un server applicativo distinto. Il server web esegue il codice nelle pagine server. Il vantaggio potenziale di un server applicativo è che i componenti applicativi possono essere riusati da altre applicazioni web per invocare la medesima logica di business. Tuttavia, Acquisti OnLine è un sistema a sé stante (stand-alone) e non vi sono altre applicazioni web che possono trarre vantaggio da tale logica di business.

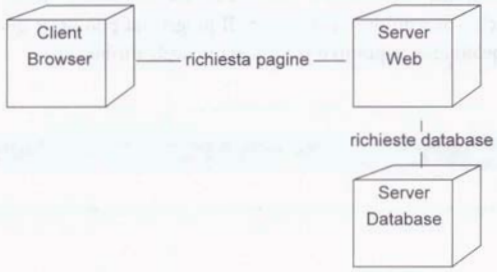


FIGURA 6.27 Diagramma di deployment (Acquisti OnLine).

6.3.4 Progetto delle collaborazioni

Come spiegato all'inizio di questo capitolo, il progetto di sistema si divide in progetto architetturale e progetto dettagliato. Il progetto di package, componenti e nodi è parte del *progetto architetturale*. Il *progetto di dettaglio* si concentra sulle collaborazioni (Sezione 6.2).

Le *collaborazioni* definiscono la realizzazione di casi d'uso e di operazioni complesse (operazioni semplici non devono essere rappresentate come collaborazioni). Il progetto delle collaborazioni conduce inevitabilmente all'*elaborazione* (modifiche ed estensioni) dei diagrammi delle classi esistenti e alla produzione di nuovi diagrammi di collaborazione (o elaborazione di diagrammi esistenti). Altre tipologie di diagramma, in particolare i diagrammi statechart, potrebbero dover essere sviluppati o elaborati.

6.3.4.1 Elaborazione dei casi d'uso

Il progetto delle collaborazioni inizia con l'*elaborazione dei casi d'uso*. I casi d'uso documentati durante l'analisi dei requisiti probabilmente non contengono un sufficiente livello di dettaglio per il progetto delle collaborazioni. Le specifiche dei casi d'uso devono essere sviluppate in documenti di progetto attraverso richieste agli utenti riguardanti il livello di sistema.

Se la gestione dei requisiti (Sezione 3.4) ha mostrato delle carenze, ora si verifica l'ultima opportunità di essere più formali e disciplinati. Per assicurare una corretta gestione del cambiamento e della tracciabilità, i requisiti dovrebbero essere classificati con attenzione e memorizzati nel repository di uno strumento CASE (l'importanza di una gestione accurata dei requisiti va molto oltre l'analisi dei requisiti e le fasi iniziali di progetto; questo sarà discusso nel Capitolo 10).

6.3.4.1.1 Numerazione e strutturazione dei requisiti

I requisiti devono essere numerati (Sezione 3.4.1) e strutturati (Sezione 3.4.2). Entrambe le attività possono essere condotte efficacemente con l'ausilio di uno strumento CASE. I tentativi di tracciare manualmente le modifiche ai requisiti sono destinati al fallimento. Con uno strumento CASE, la rinumerazione e ristrutturazione dei requisiti sono semplici.

La Figura 6.28 mostra una parte del documento di specifica di un caso d'uso con i requisiti numerati usando il sistema decimale con il prefisso CU (Caso d'Uso). Il prefisso è d'aiuto quando il documento del caso d'uso contiene più di un tipo di requisito. Nella figura i requisiti sono racchiusi tra parentesi quadre e sottolineati; anche il colore può essere di aiuto per accrescere la comprensibilità del documento.

Una volta inseriti in un repository, i requisiti possono essere visti e modificati per mezzo di altri strumenti supportati dallo strumento CASE. La Figura 6.29 mostra una di tali visualizzazioni nella quale la gerarchia di requisiti è enfatizzata. Il progettista può usare questa visualizzazione per modificare un qualunque requisito o per associarvi degli attributi.

6.3.4.1.2 Documento di progetto per un caso d'uso

Esempio di progetto: Passo 5 (Acquisti OnLine)

Si consideri il documento d'analisi del caso d'uso riportato nella sezione 2.2.2.4 (Tabella 2.2) e relativo al caso d'uso "Ordinare Computer Configurato". Il documento non è sufficiente per il progetto di una collaborazione.

Obiettivo di questo passo è l'elaborazione del documento in una specifica del caso d'uso a livello di progetto. Il documento elaborato è stato organizzato come nella Figura 6.28 (infatti, le Figure 6.28 e 6.29 mostrano una parte della soluzione).

Il testo del documento di progetto per il caso d'uso è mostrato alla pagina seguente. Si noti che il documento può essere stampato in un formato differente da quello qui presentato. Ad esempio, la visualizzazione e la stampa dei numeri dei casi d'uso può essere soppressa.

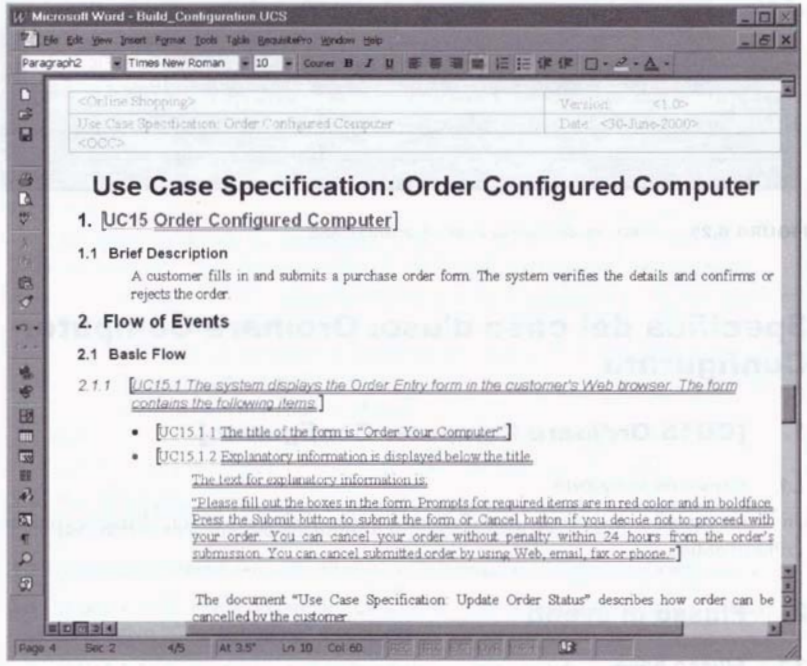


FIGURA 6.28 Estratto di un documento di caso d'uso gestito da uno strumento CASE.

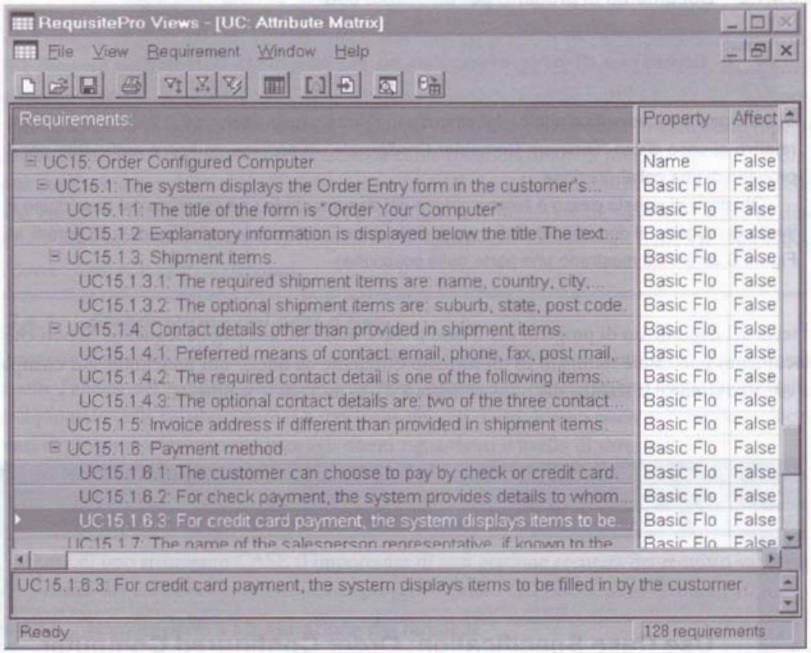


FIGURA 6.29 Gestione dei requisiti in uno strumento CASE.

Specifica del caso d'uso: Ordinare Computer Configurato

1. [CU15 Ordinare Computer Configurato]

1.1 Breve descrizione

Un cliente compila e invia un modulo d'ordine acquisto. Il sistema verifica i dettagli e conferma o rifiuta l'ordine.

2. Flusso di eventi

2.1 Flusso base

2.1.1 [CU15.1 Il sistema visualizza il modulo Inserimento Ordine nel browser web del cliente. Il modulo contiene i seguenti elementi.]

- [CU15.1.1 Il titolo del modulo è "Ordina il tuo Computer".]

- [CU15.1.2 Informazioni esplicative sono visualizzate sotto il titolo.
  Il testo dell'informazione esplicativa è:
  "Si prega di riempire i campi del modulo. Gli elementi obbligatori sono in rosso e in grassetto. Si prema il pulsante `Invia` per dare l'ordine oppure il pulsante `Cancella` se si decide di non procedere con l'ordine. È possibile cancellare il proprio ordine senza penalità fino a 24 ore dall'invio dell'ordine. È possibile cancellare l'ordine inviato attraverso web, email, fax, o telefono".]
  Il documento "Specifica del Caso d'Uso: Aggiorna Stato Ordine" descrive come un ordine può essere cancellato dal cliente.
- [CU15.1.3 Elementi per la consegna.]
  [CU15.1.3.1 Gli elementi richiesti per la spedizione sono: nome, nazione, città, via, indicazioni per il corriere.]
  [CU15.1.3.2 Gli elementi opzionali per la spedizione sono: quartiere, stato, codice postale.]
- [CU15.1.4 Dettagli di comunicazione oltre a quelli forniti negli elementi per la spedizione.]
  [CU15.1.4.1 Mezzo di comunicazione preferito: email, telefono, fax, posta, corriere.]
  [CU15.1.4.2 É richiesto uno dei seguenti dettagli: email, telefono, fax.]
  [CU15.1.4.3 I dettagli di comunicazione opzionali sono: due dei tre elementi elencati in precedenza, e l'indirizzo postale (se diverso da quello fornito negli elementi per la spedizione).]
- [CU15.1.5 L'indirizzo di fatturazione se distinto da quello fornito negli elementi per la spedizione.]
- [CU15.1.6 Metodo di pagamento.]
  [CU15.1.6.1 Il cliente può scegliere di pagare con assegno o carta di credito.]
  [CU15.1.6.2 Per pagamento con assegno, il sistema fornisce i dettagli per compilare l'assegno stesso.]
  [CU15.1.6.3 Per pagamento con carta di credito, il sistema presenta la lista selezionabile delle carte di credito accettate e visualizza gli elementi da compilare (numero di carta di credito, scadenza).]
- [CU15.1.7 Il nome del venditore, se già conosciuto dal cliente.]
- [CU15.1.8 I due pulsanti d'azione: `Invia` e `Cancella`.]

2.1.2 [CU15.2 Il sistema visualizza al cliente i campi per inserire i dettagli dell'ordine posizionando il cursore sul primo campo scrivibile (l'elemento Nome).]

2.1.3 [CU15.3 Il sistema permette d'inserire l'informazione in qualsiasi sequenza.]

2.1.4 [CU15.4 Se il cliente non invia o cancella il modulo prima di 15 minuti, il flusso alternativo "ClienteInattivo" è eseguito.]

2.1.5 [CU15.5 Se il cliente preme il pulsante Invia e tutte le informazioni richieste sono state fornite, il modulo d'ordine è inviato al server web. Il server web comunica con il server database per memorizzare l'ordine nel database.]

2.1.6 [CU15.6 Il server database assegna un unico numero d'ordine e un numero conto cliente all'ordine di acquisto.]

2.1.7 [CU15.7 Se il server database non riesce a creare e memorizzare l'ordine, il flusso alternativo "EccezioneDatabase " è eseguito.]

2.1.8 [CU15.8 Se il cliente invia un modulo d'ordine con informazioni incomplete, il flusso alternativo "InformazioneIncompleta" è eseguito.]

2.1.9 [CU15.9 Se il cliente ha fornito l'indirizzo email come mezzo preferito di comunicazione, il sistema invia un messaggio email al cliente con i numeri cliente e ordine, e i dettagli dell'ordine, come conferma della ricezione. Il caso d'uso termina.]

2.1.10 [CU15.10 Altrimenti, i dettagli dell'ordine saranno spediti al cliente e il caso d'uso termina.]

2.1.11 [CU15.11 Se il cliente preme il pulsante Cancella, il flusso alternativo "Cancellare" è eseguito.]

2.2 *Flussi alternativi*

2.2.1 ClienteInattivo

[CU15.4.1 Se il cliente è inattivo per più di 15 minuti, il sistema chiude la connessione con il browser. Il caso d'uso termina.]

2.2.2 EccezioneDatabase

[CU15.7.1 Se il database segnala un'eccezione, il sistema la interpreta e informa il cliente sulla natura dell'errore. Se il cliente si è disconnesso, il sistema invia come email il messaggio d'errore al cliente e al venditore. Il caso d'uso termina (se il cliente non è raggiungibile attraverso Internet o email, il venditore deve contattare il cliente con altri mezzi).]

2.2.3 InformazioneIncompleta

[CU15.8.1 Se il cliente non ha riempito tutti gli elementi obbligatori, il sistema invita il cliente a fornire l'informazione mancante. L'elenco dell'informazione mancante è visualizzato. Il caso d'uso continua.]

2.2.4 Cancellare

[CU15.11.1 Se il cliente preme il pulsante `Cancella`, i campi già compilati del modulo sono annullati. Il caso d'uso continua.]

3. Pre-Condizioni

3.1 Il cliente ottiene la pagina web del sistema tramite il proprio browser Internet. La pagina visualizza i dettagli del computer configurato con il prezzo. Il cliente preme il pulsante `Acquista`.

3.2 Il cliente preme il pulsante `Acquista` prima che siano trascorsi 15 minuti dall'ultima richiesta di configurazione computer da costruire e visualizzare nella pagina del browser.

4. Post-Condizioni

4.1 Se l'ordine del cliente ottiene risposta positiva, l'ordine d'acquisto è registrato nel database del sistema. Altrimenti, lo stato del sistema non è modificato.

6.3.4.2 *Struttura delle collaborazioni*

Il diagramma delle classi sviluppato durante l'analisi identifica le *classi persistenti del database*. Le *classi del programma applicativo* non possono in realtà essere determinate prima che la fase di progetto abbia inizio. Le classi per le collaborazioni devono essere progettate coerentemente con la tecnologia scelta per l'applicazione (Sezione 6.3.2.1).

Una *pagina web* può contenere script eseguiti sul server e/o interpretati dal browser sul client. Il progetto delle collaborazioni per le pagine web è una sfida: la tecnologia potrebbe non supportare propriamente la visione object-oriented del mondo. In alcuni casi, l'intera applicazione potrebbe essere guidata da una singola pagina web caricata con oggetti e applet.

In applicazioni web più complesse, dovrebbe essere possibile distinguere le pagine client dalle pagine server. Nella prospettiva dell'approccio BCED (Sezioni 6.1.3.2 e 6.3.1.2), le *pagine client* in formato HTML corrispondono alle *classi boundary*. Le *pagine server* possono avere associazioni con le pagine client da un lato e con altre risorse del server web dall'altro. Le pagine server corrispondono alle *classi control* nella gerarchia BCED (Conallen, 2000).

Esempio di progetto: Passo 6 (Acquisti OnLine)

Si consideri il documento di progetto del caso d'uso "Ordinare Computer Configurato" nella Sezione 6.3.4.1.2. Basandosi sul flusso di eventi descritto nel documento, progettare l'aspetto strutturale della collaborazione che realizza il caso d'uso.

Anche per questo semplice caso d'uso, il progetto dettagliato della collaborazione è oltre lo scopo di questo esempio, per cui saranno fatte alcune semplificazioni. Non c'è bisogno di gestire tutti i dettagli strutturali e di forma del modulo ordine (si veda Capitolo 7). Analogamente, non c'è bisogno di trattare le interfacce tra l'applicazione e il database (cioè non c'è interesse per il recupero e la memorizzazione dell'informazione nel database) (si veda Capitolo 8).

Applicando l'approccio BCED (Sezione 6.3.1.2), tutte le classi proposte dovrebbero aver assegnato un prefisso `b`, `c`, `d` oppure `e` per indicare la propria associazione all'appropriato package BCED. Le classi dovrebbero anche essere stereotipate con uno dei seguenti stereotipi: `«pagina client»`, `«modulo»`, `«pagina server»`, `«entity»`, `«interfacciadb»`.

Qualsiasi relazione di associazione e aggregazione tra classi deve essere identificata. Altresì, le relazioni d'istanziazione (creazione di oggetti) devono essere mostrate per indicare il flusso di messaggi per istanziare gli oggetti (le relazioni di istanziazione sono indicate in UML con frecce punteggiate dal mittente di un messaggio alla classe cui si richiede di istanziare il nuovo oggetto). Se un oggetto è istanziato come risultato di un evento utente (ad esempio un utente che preme un pulsante d'azione), l'evento può essere mostrato sulla linea d'istanziazione.

La Figura 6.30 è il diagramma delle classi che rappresenta la collaborazione strutturale per il caso d'uso "Ordinare Computer Configurato". Poiché il caso d'uso è solo parte dell'applicazione, per rappresentare le pre e post-condizioni del caso d'uso, sono state introdotte due classi da un altro caso d'uso ("Costruire Configurazione Computer"): `b_PaginaClientConfigurazione` e `e_Configurazione`.

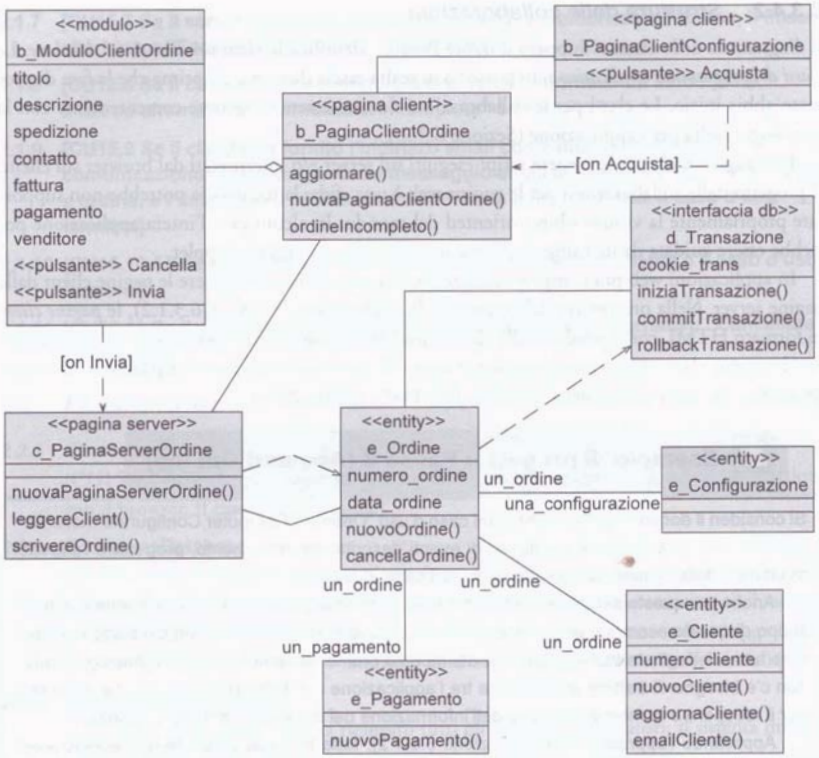


FIGURA 6.30 Collaborazione strutturale per il caso d'uso "Ordinare Computer Configurato" (Acquisti OnLine).

Si noti che l'evento `[on Acquista]` attivato da `b_PaginaClientConfigurazione` stabilisce il contesto per il caso d'uso. La classe `b_PaginaClientOrdine` è responsabile per il trattamento del contenuto di `b_ModuloClientOrdine`. Quest'ultima rappresenta un modulo ovvero una collezione di campi d'ingresso sullo schermo i quali accettano dati dall'utente. Una volta che il modulo è riempito, è sottomesso a una pagina server (`c_PaginaServerOrdine`) per l'elaborazione.

I quattro oggetti «entity» (`e_Ordine`, `e_Configurazione`, `e_Pagamento`, `e_Cliente`) memorizzano l'informazione attuale sull'ordine indipendentemente dalle pagine web. Si ricordi, tuttavia, che questi sono solo oggetti in memoria, la memorizzazione persistente di un ordine nel database non è un obiettivo di questo esempio.

Associato con la collaborazione vi è un oggetto di `d_Transazione`, istanziato da `c_PaginaServerOrdine` una volta che l'ordine è stato inviato. Sebbene non mostrato esplicitamente sul diagramma, la classe `d_Transazione` dovrebbe comunicare con il database per assicurarsi che la transazione ordine sia stata correttamente eseguita. Questa classe è anche il meccanismo applicativo per la gestione dello stato del client (ad esempio, impostando e rimuovendo "cookies" nel client).

6.3.4.3 *Comportamento delle collaborazioni*

Esempio di progetto: Passo 7 (Acquisti OnLine)

Si considerino il documento di progetto del caso d'uso "Ordinare Computer Configurato" nella Sezione 6.3.4.1.2 e il precedente passo dell'esempio (Sezione 6.3.4.2). Basandosi sul flusso di eventi nel documento di caso d'uso e sulle classi di Figura 6.30, progettare l'aspetto comportamentale della collaborazione per il caso d'uso.

La Figura 6.31 è il diagramma di collaborazione che rappresenta il comportamento del caso d'uso "Ordinare Computer Configurato". Il diagramma esprime i requisiti funzionali del caso d'uso. La collaborazione inizia quando la prima pre-condizione del caso d'uso è soddisfatta e il cliente preme il pulsante `Acquista`. Ciò è rappresentato sul diagramma con il messaggio `[on Acquista]nuovaPaginaClientOrdine` che genera un oggetto della classe `b_PaginaClientOrdine`. Quest'oggetto è una pagina client in formato HTML che contiene un oggetto `"modulo"` della classe `b_ModuloClientOrdine` (Figura 6.30). Gli attributi di quest'ultima classe rappresentano i campi di ingresso del modulo, compresi due pulsanti d'azione: `Cancella` e `Invia`.

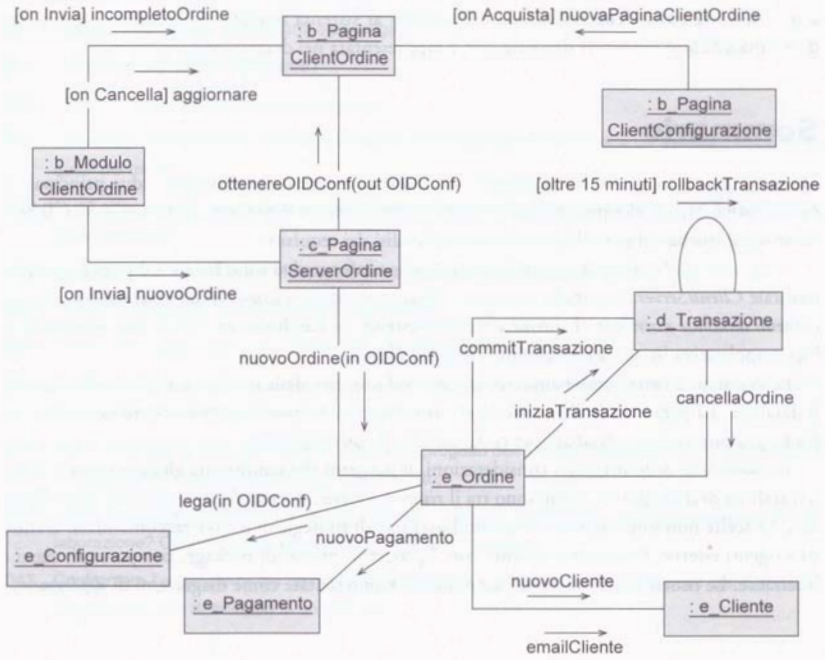


FIGURA 6.31 Collaborazione comportamentale per il caso d'uso "Ordinare Computer Configurato" (Acquisti OnLine).

Una volta che la pagina client visualizza il modulo nel browser (CU15.1), il cliente può inserire le informazioni relative all'ordine (CU15.2 e CU15.3). L'azione `Cancella` (CU15.11.1) è servita da `b_PaginaClientOrdine` (il metodo `aggiornare`). L'azione `Invia` quando il client non ha riempito tutti i campi obbligatori (CU15.8.1) è anch'essa servita da `b_PaginaClientOrdine` (il metodo `ordineIncompleto`).

Quando il cliente preme il pulsante `Invia` e tutte le informazioni richieste sono state fornite (CU15.5), la classe `c_PaginaServerOrdine` prende il controllo, creando gli oggetti `e_Ordine`, `e_Cliente`, `e_Pagamento`, `e_Configurazione`. I primi tre oggetti memorizzano gli elementi del modulo d'ordine inseriti dal cliente.

L'oggetto `e_Configurazione` è stato istanziato in precedenza da un altro caso d'uso ("Costruire Configurazione Computer"). Ora è legato a `e_Ordine` dopo che `c_PaginaServerOrdine` ne ha ottenuto l'OID dall'attributo di legame in `b_PaginaClientOrdine` (esiste un'associazione definita tra `b_PaginaClientOrdine` e `b_PaginaClientConfigurazione`).

Una volta che `e_Ordine` è istanziato, la transazione di business inizia. A tal fine, un nuovo oggetto `d_Transazione` è creato: esso gestisce lo stato del client (compresi i timeout guidati da cookie). Se la transazione ha successo, `e_Ordine` richiede a `e_Client` di confermare la ricezione dell'ordine al cliente per mezzo di un messaggio email (CU15.9).

Se il cookie raggiunge il timeout (CU15.4) oppure il database restituisce un errore (CU15.7.1), il metodo `rollbackTransazione` richiede a `e_Ordine` che cancelli se stesso e a `d_Transazione` che restituisca il controllo al sistema operativo (la comunicazione tra `d_Transazione` e il server database non è rappresentata nel diagramma).

Sommario

Questo capitolo ha aperto la discussione sul progetto di sistema presentando i due principali (e caratteristici) aspetti del progetto, il progetto architetturale e il progetto di dettaglio. La discussione segue quella del precedente capitolo sull'analisi dei requisiti.

Le tipiche applicazioni di sviluppo di un sistema informativo sono basate sul principio architetturale *Client/Server*. Specifiche soluzioni C/S comprendono i *sistemi d'elaborazione distribuita* e i *sistemi database distribuiti*. I *sistemi a tre-strati* estendono l'architettura C/S di base separando la logica applicativa in uno strato distinto (fisico e/o logico).

Le decisioni architetturali hanno un impatto sul progetto delle interfacce tra un'applicazione e il database. La gerarchia BCE di package (introdotta nel Capitolo 5) deve essere estesa con un package d'interfaccia al database per costituire una *gerarchia BCED*.

Il *riuso* è una delle maggiori considerazioni di progetto che condiziona gli elementi sia architetturali sia di dettaglio. Le scelte sono tra il *riuso di librerie*, il *riuso di framework* e il *riuso di pattern*. Le scelte non sono esclusive, una combinazione di strategie di riuso è raccomandata. Il riuso da sorgenti esterne deve essere coerente con il progetto interno di package, componenti, classi e interfacce. Le risorse computazionali sono quindi rappresentate come diagrammi di *deployment*.

Il progetto di dettaglio si concentra sulle *collaborazioni*. Una collaborazione specifica la realizzazione di un caso d'uso o di un'operazione e rappresenta lo scambio di messaggi tra oggetti. I problemi da considerare comprendono riscrittura, sovraccarico e iterazione dei metodi, tipi parametrici, messaggi "self", messaggi asincroni e chiamate di ritorno. Gli aspetti *strutturali* della collaborazione sono rappresentati in diagrammi delle classi; gli aspetti *comportamentali* nei diagrammi di collaborazione.

Il capitolo si conclude con un *esempio guidato* di progetto, il seguito dell'esempio d'analisi introdotto nel Capitolo 2 (usando l'applicazione Acquisti OnLine). L'esempio ha illustrato l'uso delle tecniche UML di progetto: il progetto di package, componenti, rilascio e collaborazioni.

Domande

D1 Spiegare la differenza tra un sistema d'elaborazione distribuita e un sistema di database distribuiti.

D2 Che cos'è un'architettura a tre strati? Se ne discutano vantaggi e svantaggi.

D3 Cosa significa che i database moderni possono essere programmati?

D4 Qual è lo scopo del package database nell'approccio BCED?

D5 Confrontare il riuso di librerie di classi e di framework.

D6 Come si relazionano componenti e package?

D7 Che cos'è una classe dominante?

D8 Che cos'è una relazione di connessione?

D9 Che cos'è una relazione di realizzazione in una collaborazione? Fornire un esempio (diverso da quelli del testo).

D10 Quali sono i modi alternativi di rappresentare gli argomenti di un messaggio in un diagramma di collaborazione?

D11 Quali sono i principali tipi di messaggio (a parte i costruttori e distruttori)?

D12 Qual è la differenza tra riscrittura e sovraccarico?

D13 Come sono legati i messaggi d'iterazione alle collezioni?

D14 Il mittente di un messaggio potrebbe inviare il proprio OID all'oggetto destinatario. Questa asserzione è valida per i messaggi asincroni? Spiegare.

D15 Quale diagramma UML può essere usato per il progetto dell'aspetto strutturale di una collaborazione? Spiegarne l'adeguatezza per questo compito.

D16 Quali diagrammi UML possono essere usati per il progetto dell'aspetto comportamentale della collaborazione? Confrontarne l'adeguatezza per questo compito.

D17 Confrontare l'applicabilità dei package dei caso d'uso e dei package di classi nelle fasi d'analisi e progetto del ciclo di sviluppo di un sistema.

Esercizi

Requisiti aggiuntivi (Video Store)

Si considerino i seguenti requisiti aggiuntivi per il Video Store (qui ripetuti per convenienza dalla fine del Capitolo 4):

1. Cassette e dischi riportati in ritardo implicano un pagamento uguale a un periodo d'affitto supplementare. Ogni supporto di film ha un identificatore unico.
2. I film sono ordinati da fornitori i quali sono generalmente capaci di fornire cassette e/o dischi in una settimana. Generalmente, diversi film sono ordinati in un singolo ordine.
3. Si accettano prenotazioni per un film che è stato ordinato oppure le cui copie sono attualmente in affitto. Si accettano anche prenotazioni per film che non sono né disponibili né ordinati: in questo caso, si richiede al cliente un deposito pari a un periodo d'affitto.
4. I clienti possono fare molte prenotazioni ma richieste di prenotazione distinte sono preparate per ogni film prenotato. Una prenotazione potrebbe essere cancellata se il cliente non risponde entro una settimana dalla data in cui gli è stata comunicata la disponibilità del film richiesto. Se è stato versato un deposito, allora è accreditato sul conto del cliente.
5. Il database memorizza le normali informazioni su fornitori e clienti, come indirizzi, numeri di telefono, etc. Ogni ordine verso un fornitore identifica i film ordinati, i formati (cassetta/disco) richiesti, le quantità, la data di consegna prevista, il prezzo d'acquisto, gli sconti applicabili, etc.
6. Quando una cassetta è riportata da un cliente o consegnata da un fornitore, si soddisfano anzitutto le prenotazioni. Questo implica l'invio di una comunicazione al cliente che ha prenotato. Al fine di assicurare che le prenotazioni siano gestite correttamente, sia l'avviso al cliente sia il susseguente affitto sono messi in relazione alla prenotazione.
7. In una singola transazione un cliente può affittare diverse cassette e/o dischi ma ognuno di questi deve costituire una registrazione separata. Per ogni affitto, tale registrazione comprende i tempi e le date di uscita, restituzione prevista, restituzione effettiva. Alla restituzione, la registrazione di affitto è aggiornata per indicare che il video è stato riportato e il pagamento finale (o rimborso) è stato eseguito. Anche l'impiegato che autorizza l'affitto è registrato. I dettagli sugli affitti dei clienti sono conservati per un anno al fine di permettere valutazioni statistiche sui clienti stessi.
8. Tutte le transazioni sono eseguite in contanti, con bonifico o carta di credito. I clienti devono versare il pagamento quando cassette e/o dischi sono ritirati.
9. Quando una cassetta (disco) è restituita in ritardo (o non può essere restituita per una qualsiasi ragione), si addebita una certa somma sul conto del cliente oppure il cliente corrisponde direttamente tale somma.
10. Se una cassetta (disco) deve essere restituita da più di due giorni, un'Informativa è inviata al cliente. Una volta che due Informative su una singola cassetta (disco) sono state inviate, il cliente è annotato come inadempiente e il successivo affitto è soggetto alla rimozione dello stato di inadempienza.

Si considerino i requisiti aggiuntivi sopra definiti e le soluzioni agli esercizi sul Video Store nel Capitolo 4.

E1 Video Store – Progettare un diagramma di collaborazione strutturale per la realizzazione del caso d'uso "Prenotare Video".

E2 Video Store – Progettare un diagramma di collaborazione comportamentale per la realizzazione del caso d'uso "Prenotare Video".

E3 Video Store – Progettare un diagramma di collaborazione strutturale per la realizzazione del caso d'uso "Restituire Video".

E4 Video Store – Progettare un diagramma di collaborazione comportamentale per la realizzazione del caso d'uso "Restituire Video".

E5 Video Store – Progettare un diagramma di collaborazione strutturale per la realizzazione del caso d'uso "Ordinare Video".

E6 Video Store – Progettare un diagramma di collaborazione comportamentale per la realizzazione del caso d'uso "Ordinare Video".

E7 Video Store – Progettare un diagramma di collaborazione strutturale per la realizzazione del caso d'uso "Gestire Cliente".

E8 Video Store – Progettare un diagramma di collaborazione comportamentale per la realizzazione del caso d'uso "Gestire Cliente".

E9 Video Store – Riferendosi all'attività `Aggiornare Scorta` in Figura 4.13, disegnare un diagramma delle attività per la realizzazione dell'operazione stessa.

Capitolo 7

Progetto dell'interfaccia utente

Il tempo in cui gli schermi di computer erano monocromatici e la tastiera il solo dispositivo d'ingresso è passato. Oggi gli schermi sono intelligenti e colorati, e per controllare l'esecuzione di un programma l'utente è dotato di mouse (per non dire di voce e contatto con lo schermo). Ovviamente, il programma può ancora essere progettato per impedire eventi illegali o non autorizzati, ma lo spostamento del controllo dall'algoritmo all'utente ha modificato il modo in cui i sistemi e le loro interfacce sono progettati e implementati.

Lo sviluppo delle interfacce utente inizia con le prime bozze di finestre GUI nella fase d'analisi dei requisiti. Tali bozze sono usate per la raccolta dei requisiti, nelle interviste con i clienti, per costruire prototipi, per essere incluse nei documenti di caso d'uso. Durante il progetto, le finestre GUI per l'applicazione sono sviluppate in modo conforme al sottostante software di presentazione GUI e alle caratteristiche e ai vincoli dell'ambiente di programmazione scelto. Questi argomenti costituiscono il tema del capitolo.

7.1 Progetto dell'interfaccia come attività multidisciplinare

Il progetto GUI (Graphical User Interface) è un'*attività pluridisciplinare* che necessita di diverse capacità, augurabilmente tutte presenti nel *gruppo di progetto*; una singola persona non ha, probabilmente, le capacità combinate di un artista grafico, dell'analista dei requisiti, del progettista di sistema, del programmatore, dell'esperto della tecnologia, dello psicologo del comportamento e altre ancora dipendenti dalla natura del sistema.

Un tipico progetto GUI per lo sviluppo di un sistema informativo inizia con i *casi d'uso*. L'*analista* che descrive il flusso d'eventi per un caso d'uso ha, in mente, una "visione" dell'interfaccia da usare per la gestione dell'interazione. In alcuni casi, l'analista potrebbe scegliere di inserire rappresentazioni grafiche per le interfacce utente nel documento dei casi d'uso poiché il solo testo può non descrivere adeguatamente tali complesse interazioni. Occasionalmente, i processi di raccolta e negoziazione dei requisiti del cliente necessitano la produzione di bozze dell'interfaccia.

Il *progettista* coinvolto nella specifica delle collaborazioni per la realizzazione dei casi d'uso deve avere una chiara immagine visuale delle "schermate" GUI: se queste non sono state impostate precedentemente dall'analista, il progettista deve necessariamente produrle. Le rappresentazioni

del progettista devono essere conformi alla sottostante tecnologia GUI (librerie per la produzione di finestre, browser Internet, etc.). Anche a tal fine, potrebbe essere utile consultare un *esperto della tecnologia* sottostante per usare con successo le caratteristiche tecnologiche.

Prima che i progetti delle collaborazioni siano passati ai programmatori per l'implementazione, deve essere costruito un prototipo userfriendly (amichevole) delle schermate GUI. Questo compito dovrebbe coinvolgere *artisti grafici* e *psicologi* per offrire un'interfaccia attraente e usabile.

Il compito del *programmatore* non è solo quello di implementare le schermate ma anche quello di suggerire modifiche motivate dall'ambiente di programmazione. In alcuni casi, le modifiche possono determinare dei miglioramenti, in altri, esse potrebbero penalizzare il progetto a causa delle restrizioni imposte dagli ambienti di programmazione.

Questa breve discussione suggerisce che il progetto dell'interfaccia è un compito complesso, e diversi testi sono stati scritti per enfatizzarne i vari aspetti (ad esempio, Constantine e Lockwood, 1999; Galitz, 1996; Fowler, 1998; Olsen, 1998; Ruble, 1997; Windows, 2000). Questo capitolo è centrato su ciò che il *progettista* di sistema deve conoscere per progettare, in *cooperazione* con gli altri esperti coinvolti, un'interfaccia di successo.

7.2 Dal prototipo all'implementazione dell'interfaccia

Il principio fondamentale nel progetto dell'interfaccia è che "*l'utente ha il controllo*" (con l'ipotesi che il sistema, e non l'utente, controlli integrità, correttezza e sicurezza dei dati). Un moderno programma object-oriented è *guidato da eventi* (messaggi) ai quali gli oggetti reagiscono. La comunicazione interna tra gli oggetti è attivata a partire da eventi esterni generati dall'utente.

L'interfaccia è anche lo "strumento" attraverso cui è venduto un certo prodotto/servizio al cliente: da qui deriva l'importanza di realizzare un qualcosa di "gradevole" che catturi l'attenzione "visiva" dell'utente stesso. Un prototipo GUI può servire al duplice scopo di valutare l'attrattiva della schermata che sarà sottoposta all'utente e dare l'idea delle sue funzioni; l'aspetto definitivo sarà sviluppato nella fase d'implementazione.

Esempio 7.1 (CMS)

Si considerino il caso di studio CMS - Gestione Contatti con i Clienti (Sezione 2.3.3) e i relativi esempi nel Capitolo 4. In particolare, si consideri il progetto della classe `Organizzazione` nell'Esempio 4.8 (Sezione 4.2.2.3).

Lo scopo di quest'esempio è mostrare come può modificarsi una schermata GUI passando dal prototipo iniziale all'implementazione finale. Si assuma che la sottostante tecnologia GUI sia Microsoft Windows.

Un prototipo GUI per la classe Organizzazione è presentato nella Figura 7.1: lo scopo principale è visualizzare dati e oggetti di controllo nella finestra. I commenti degli utenti e le idee del gruppo di sviluppo dell'interfaccia hanno causato diverse modifiche alla finestra: la Figura 7.2 mostra una possibile implementazione definitiva dell'interfaccia come una finestra di dialogo. Come si può notare,

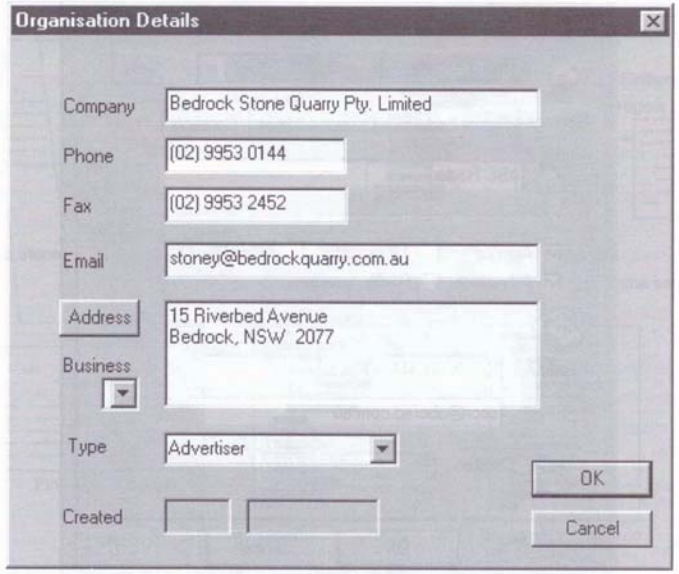


FIGURA 7.1 Prototipo di finestra per la classe Organizzazione (CMS) (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

il programmatore ha optato per una lista a tendina per visualizzare le organizzazioni che sono quindi dettagliate tramite la cartella a schede (`General`, `PostalAddress`, `CourierAddress`, `Contacts`). Le modifiche apportate rendono il dialogo con il cliente più agevole e sono conformi ai principi di progetto GUI di Microsoft Windows.

7.3 Linee guida per il progetto d'interfaccia centrato sull'utente

Il progetto GUI deve essere *centrato* sull'utente. A quest'osservazione si associa un insieme di linee guida per gli sviluppatori software, pubblicate dai fornitori di ambienti per la produzione di interfacce GUI (ad esempio, Windows, 2000) e discusse anche in diversi testi (ad esempio, Galitz, 1996; Ruble, 1997).

Le *linee guida* costituiscono i fondamenti sui quali lo sviluppatore costruisce. Per questo, dovrebbero fare da sfondo alle considerazioni dello sviluppatore su tutte le decisioni di progetto GUI. Alcune di tali linee guida sono basate sul buonsenso, altre sono motivate dalla moderna tecnologia GUI.

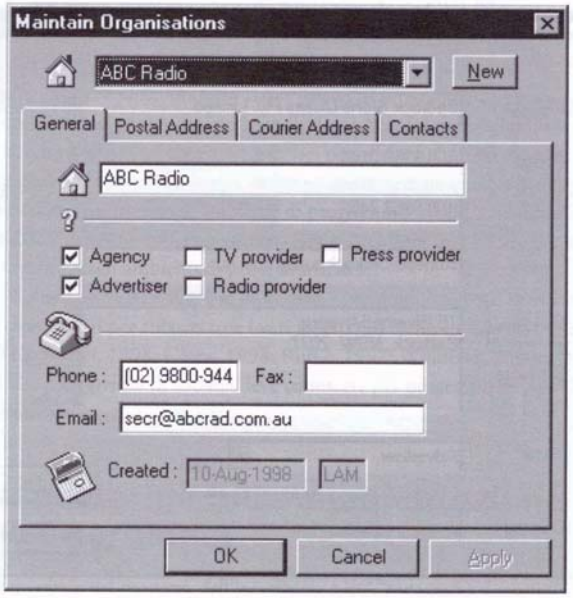


FIGURA 7.2 Finestra implementata per la classe Organizzazione (CMS) (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.3.1 L'utente ha il controllo

La principale linea guida GUI, "*l'utente ha il controllo*", potrebbe essere chiamata più correttamente la "*percezione del controllo da parte dell'utente*". Qualcuno chiama tale principio "*senza cura materna*" (*no mothering*): "Il programma non dovrebbe agire come vostra madre ovvero facendo cose per voi" (Treisman, 1994). Il significato implicito è: l'utente inizia le azioni e se, come risultato, il programma prende il controllo, l'utente ne è informato nel modo più appropriato (una clessidra, un indicatore d'attesa, o l'equivalente).

La Figura 7.3 mostra un flusso di controllo tipico in un'interazione uomo-computer. Un evento utente (un menu azione, una pressione del mouse, il movimento del cursore sullo schermo, etc.) può aprire una finestra GUI o invocare un programma, generalmente un programma scritto in 4GL-SQL. Quando il programma prende temporaneamente il controllo, è eseguito e può restituire il controllo alla medesima finestra oppure a un'altra. In alternativa, può chiamare un altro modulo in 4GL-SQL o può invocare una routine esterna. Il programma, infatti, farà in genere qualcosa per l'utente e gli restituirà dei risultati.

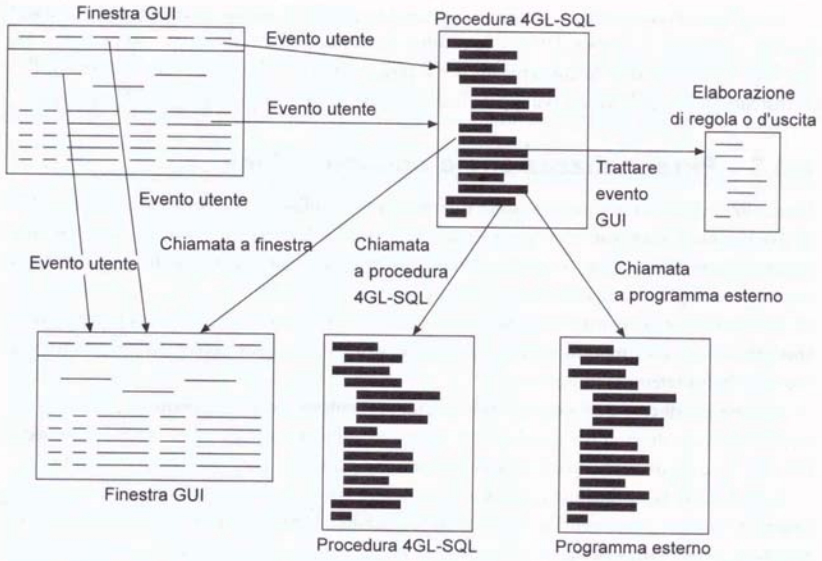


FIGURA 7.3 Flusso di controllo in un programma GUI.

7.3.2 Consistenza

La *consistenza*, nell'ambito del progetto dell'interfaccia, indica l'aderenza a standard e al modo usuale di "fare le cose". Ci sono almeno due dimensioni della consistenza:

- conformità agli standard del fornitore della tecnologia GUI;
- conformità alla nomenclatura, alle tecniche di codifica e ad altre consuetudini sviluppate internamente all'organizzazione.

Entrambe le dimensioni sono importanti e la seconda (su cui gli sviluppatori hanno influenza) non deve contraddire la prima. Se un'applicazione è sviluppata per Windows allora l'aspetto e le convenzioni Windows dovrebbero essere seguite. Su un Macintosh, sostituire il celebrato "menu mela" per mezzo di un "menu canguro", come una volta ha tentato di fare l'autore di questo libro, non è una buona idea!

Uno sviluppatore GUI non deve quindi essere troppo creativo e innovativo nel progetto d'interfaccia: gli utenti si aspettano un ambiente familiare e prevedibile, cioè conforme alle convenzioni adottate nelle precedenti applicazioni (ad esempio, menu e pulsanti d'azione messi all'incirca nello stesso posto). Come ha osservato Treisman (1994), s'immagini cosa un progettista d'automobili farebbe alla comunità dei conducenti se le nuove automobili fossero prodotte con i pedali d'acceleratore e del freno scambiati!

La conformità con gli standard interni non può, e non deve, essere sottostimata. Tali standard possono riguardare la nomenclatura e la codifica di menu, pulsanti d'azione, campi dello schermo, etc., e comprendere le convenzioni per il posizionamento di oggetti sullo schermo e l'uso consistente (nelle applicazioni sviluppate internamente) di altri elementi tipici dell'interfaccia.

7.3.3 Personalizzazione e configurazione

Personalizzazione e configurazione sono due linee guida collegate. La personalizzazione GUI è semplicemente la configurazione per uso personale mentre la configurazione, secondo l'interpretazione data in questo libro, è un compito amministrativo il cui obiettivo è di "ritagliare" un sistema software per gruppi d'utenti di rilevante interesse aziendale.

Un esempio di personalizzazione è un utente che riordina e ridimensiona colonne in un browser e salva queste modifiche come preferenze personali. Successivamente, il programma si attiva usando le preferenze impostate.

Un esempio di configurazione è fornito da un programma che può operare diversamente (fornendo interfacce diverse) per utenti novizi ed esperti; ad esempio, agli utenti novizi potrebbe essere offerto un aiuto esplicito e messaggi d'attenzione per eventi potenzialmente "pericolosi".

In molti casi, la distinzione tra personalizzazione e configurazione potrebbe risultare poco rilevante. Cambiare gli elementi di menu, creare nuovi menu, etc. ne è un esempio: se fatto per uso personale, si tratta di personalizzazione, se fatto dall'amministratore di sistema per una comunità di utenti, si tratta di configurazione.

7.3.4 Tolleranza

Una buona interfaccia dovrebbe tollerare che gli utenti possano provare, commettere errori, o prendere strade errate e, se necessario, tornare indietro al punto iniziale. La tolleranza implica quindi mettere a disposizione dell'utente un'operazione di *annullamento* delle modifiche a più livelli (e la simmetrica operazione di *ripristino*).

Tutto questo è facile da dire ma difficile da implementare. L'implementazione della tolleranza nell'interfaccia è una sfida particolare in applicazioni per database multi-utente. Ad esempio, l'utente che ritira (e spende) denaro dal conto bancario non può annullare questa operazione! Egli può solo depositare del denaro sul conto bancario con un'altra transazione. È materia di discussione il fatto che un'interfaccia tollerante debba mettere in guardia l'utente sulle conseguenze delle sue azioni.

7.3.5 Retroazione

La *retroazione* è una conseguenza diretta della prima linea guida (l'utente ha il controllo): quando il controllo è temporaneamente in mano al programma l'utente deve sapere cosa sta accadendo. Allo scopo, il sistema dovrebbe fornire all'utente informazioni audiovisive di ritorno, relative a qualsiasi evento che possa interessare l'utente stesso.

Nella maggior parte dei casi, un indicatore d'attesa è una sufficiente indicazione per mostrare che il programma sta lavorando. Per quelle parti dell'applicazione che potrebbero subire occasionali problemi, ad esempio in termini di prestazioni, una forma più incisiva di reazione potrebbe essere necessaria (quale la visualizzazione di un messaggio di spiegazione).

7.3.6 Estetica e usabilità

L'*estetica* riguarda l'attrazione visiva del sistema. L'*usabilità* riguarda la semplicità, l'efficienza, l'affidabilità e la produttività nell'uso dell'interfaccia. In ultima analisi, entrambe riguardano la *soddisfazione dell'utente* nell'uso dell'interfaccia È a questo punto che lo sviluppatore GUI necessita dell'assistenza di un grafico e di uno psicologo comportamentale.

Ci sono diverse regole auree per un progetto usabile ed esteticamente accettabile (Galitz, 1996; Constantine e Lockwood, 1999). I problemi da considerare comprendono il movimento e il puntamento dell'occhio umano, l'uso dei colori, il senso di bilanciamento e di simmetria, l'allineamento e la spaziatura degli elementi, il senso delle proporzioni, il raggruppamento di elementi in relazione, etc.

Le linee guida di estetica e di usabilità trasformano lo sviluppatore GUI in un artista. È bene ricordare in questo contesto che "semplice è bello". In applicazioni complesse, la semplicità è raggiunta al meglio per mezzo di un approccio "divide et impera": la presentazione progressiva dell'informazione in modo che questa sia mostrata solo quando necessaria e, possibilmente, in finestre distinte.

7.4 Finestre d'interfaccia

Ci sono due aspetti principali del progetto GUI, il progetto delle finestre e il progetto dei controlli d'ingresso e d'uscita nelle finestre. Entrambi dipendono dal sottostante ambiente GUI. Nella discussione che segue Microsoft Windows è assunto come ambiente sottostante (Windows, 2000).

Una tipica applicazione Windows si costruisce a partire da una singola finestra applicativa, la *finestra primaria*. La finestra primaria è completata da un insieme di *finestre secondarie* che offrono supporto alle attività dell'utente, create e distrutte nella finestra primaria durante l'interazione; tali attività spesso corrispondono alle operazioni fondamentali (create, read, update e delete) sul database.

7.4.1 Finestra primaria

Una *finestra primaria* possiede una cornice contenente una barra titolo (barra d'intestazione) per la finestra, una barra menu, barre degli strumenti, una barra di stato e un contenuto visibile e modificabile. Barre verticali e orizzontali di scorrimento sono usate per navigare attraverso il contenuto, se richiesto.

Il contenuto della finestra può essere organizzato in pannelli. I pannelli permettono di vedere e manipolare del contenuto informativo differente ma logicamente collegato a quello della finestra principale. La Figura 7.4 mostra una finestra primaria visualizzata dopo un accesso (login) riuscito all'applicazione. Il pannello sulla sinistra contiene una mappa dell'applicazione in stile Windows Explorer (il pulsante di chiusura nell'angolo in alto a destra informa l'utente che la finestra stessa può essere rimossa, se desiderato). Il commento spiega la ben nota terminologia Windows.

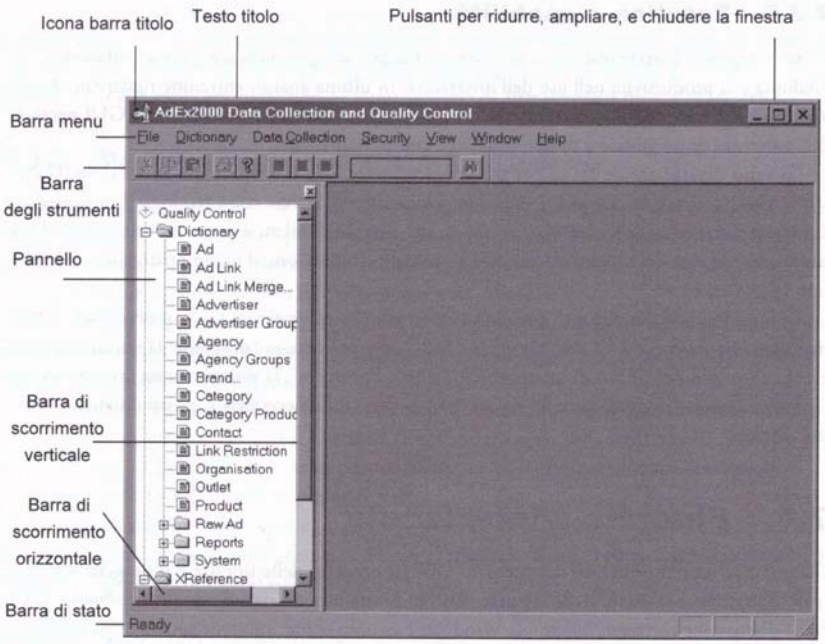


FIGURA 7.4 Finestra primaria (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

Una caratteristica distintiva di una finestra primaria è l'esistenza delle barre menu e strumenti. La barra degli strumenti contiene icone d'azione per gli elementi del menu più frequentemente usati. Le icone forniscono un modo veloce per eseguire azioni frequenti.

Esempio 7.2 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti (Sezione 2.3.3) e i relativi esempi nel Capitolo 4. Gli utenti richiedono che l'applicazione CMS sia modellata sulla funzionalità della finestra Calendario in Microsoft Outlook (Figura 7.5).

La finestra primaria dovrebbe visualizzare le attività previste per la giornata relative all'impiegato che sta usando il sistema. Il controllo Calendario può visualizzare attività passate e future. Gli eventi previsti per una particolare ora del giorno (eventi temporizzati) devono essere visualizzati nel pannello sinistro di Microsoft Outlook. Tuttavia, c'è anche il requisito di mostrare e trattare eventi fissi (non temporizzati), sospesi (dovevano essere trattati nel passato) e completati.

Si tratta di progettare la finestra primaria CMS che sia conforme ai requisiti qui espressi.

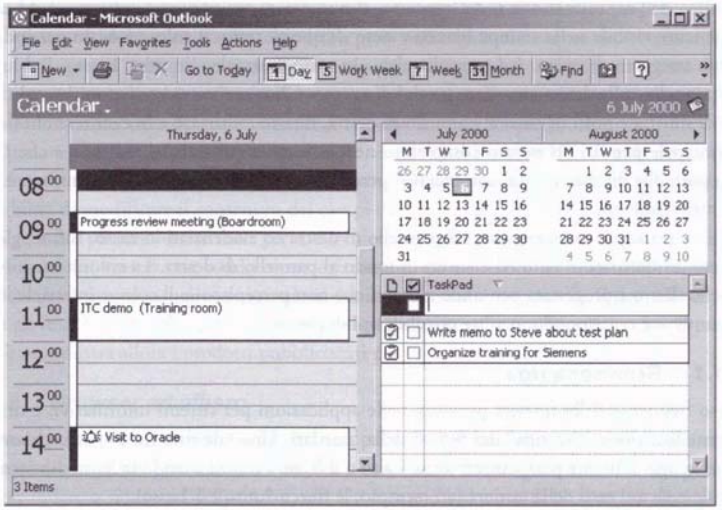


FIGURA 7.5 Microsoft Outlook - Finestra Calendario.

La Figura 7.6 mostra la finestra primaria CMS. Il controllo Calendario è progettato come una finestra riposizionabile che, se necessario, può essere chiusa. Per ciascun evento, nel pannello a sinistra, sono mostrati una breve descrizione dell'evento e i nomi dell'organizzazione o del contatto. Per alcuni eventi potrebbero essere mostrate informazioni aggiuntive, ad esempio il numero

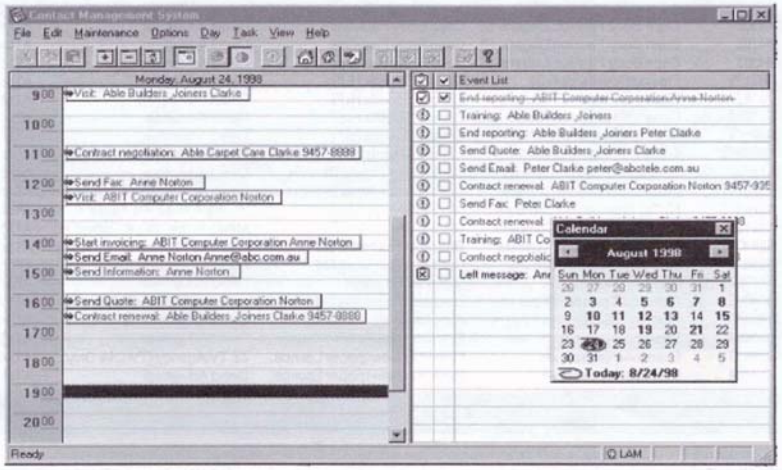


FIGURA 7.6 Finestra primaria per il caso CMS (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

telefonico dell'organizzazione o del contatto, il numero di fax oppure l'indirizzo. Sebbene non chiaramente visibile nella stampa bianco e nero di questo testo, il colore è usato per indicare la priorità assegnata all'evento (ad esempio, alta – in rosso, normale – in nero, bassa – in blu).

Il pannello sulla destra visualizza tre tipi di evento. Gli eventi *completati* sono rimossi dal pannello di sinistra e posti in cima al pannello di destra, il testo è sbarrato e in carattere blu (la ragione principale per cui un evento completato non è rimosso dalla visualizzazione è che l'evento stesso potrebbe diventare "non completato" perché ci si può accorgere che il lavoro non è effettivamente terminato).

Gli eventi *sospesi* sono elencati nel pannello di destra ed evidenziati in rosso. Infine, gli eventi *fissi* sono evidenziati in nero ed elencati in fondo al pannello di destra. La colonna a sinistra del pannello destro è progettata pensando a utenti che non percepiscono il colore: infatti, le icone in essa contenute corrispondono ai tre possibili tipi di evento.

7.4.1.1 Browser a riga

Un uso frequente della finestra primaria nelle applicazioni per sistemi informativi, è di fornire una visualizzazione "per riga" dei record della basedati. Una tale finestra è spesso chiamata un *browser a riga*. L'utente può scorrere verso l'alto o il basso i record usando la barra di scorrimento verticale o alcuni tasti della tastiera (ad esempio, le frecce ↑ alto e ↓ basso).

La Figura 7.7 è un esempio di browser a riga. Il documento (etichettato Ad Link) interno alla

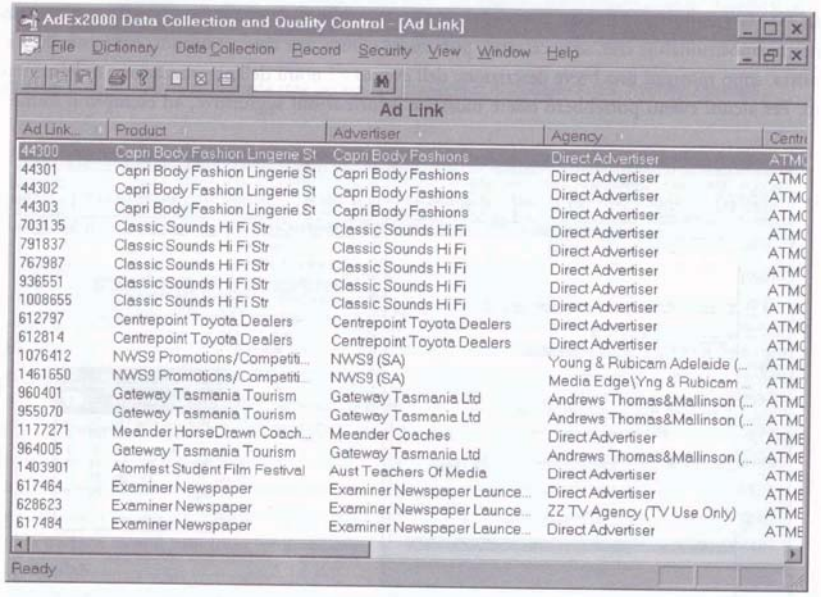


FIGURA 7.7 Finestra con browser a riga (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

finestra primaria è una *finestra figlia* (spiegata più avanti nel capitolo) con un proprio insieme di *pulsanti finestra* (ridurre, ripristinare, chiudere), posti nell'angolo alto destro della barra menu. Le colonne della griglia del browser sono *ridimensionabili* e le loro posizioni possono essere *scambiate*. Le colonne, inoltre, sono *ordinabili*: agendo con il mouse sul nome di una colonna si ordinano i record in modo ascendente o discendente rispetto ai valori di quella colonna.

In qualunque momento, una sola riga (record) è attiva nel browser. Una doppia pressione con il mouse su quella riga, normalmente visualizza una *finestra di modifica* con i dettagli del record e la possibilità di modificare il contenuto dei campi del record stesso.

Dei *pannelli* possono essere usati per suddividere la finestra verticalmente e/o orizzontalmente. La Figura 7.8 mostra una suddivisione orizzontale. Come indica il titolo della finestra, i tre pannelli sono usati per visualizzare i prodotti per inserzionista e per agenzia. Il pannello di mezzo mostra gli inserzionisti dell'agenzia di pubblicità selezionata nel pannello più in alto. Il pannello più in basso mostra allora i prodotti pubblicizzati per l'inserzionista selezionato.

7.4.1.2 Browser ad albero

Un altro tipo comune di browser in una finestra primaria è il *browser ad albero* che visualizza i record secondo un indice gerarchico. I nodi nell'albero indice contengono controlli che permettono di espandere o ridurre i livelli sottostanti. Un esempio ben noto di browser ad albero è la visualizzazione delle cartelle del computer in Windows Explorer.

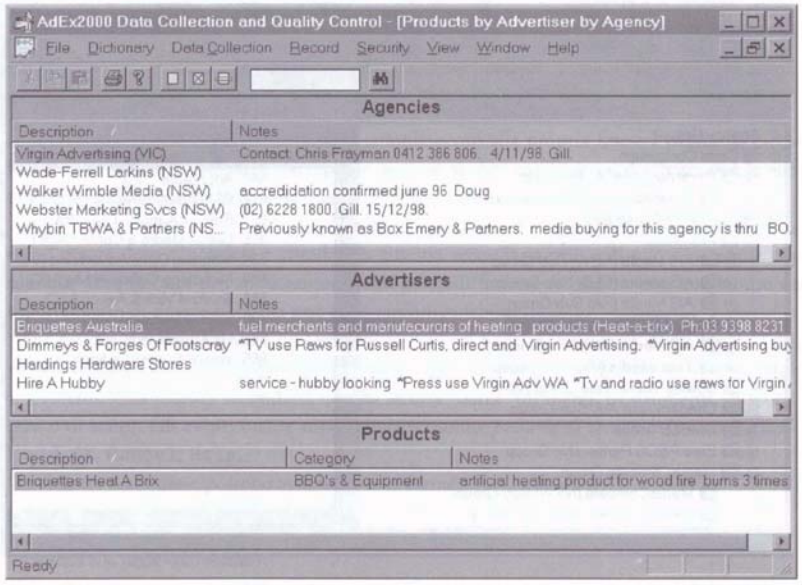


FIGURA 7.8 Finestra multi-pannello con browser a riga (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

Diversamente da un browser a riga, un browser ad albero dovrebbe permettere modifiche sul posto, cioè modifiche del contenuto della finestra senza attivare una finestra di modifica. Le modifiche in un browser ad albero sono fatte in genere attraverso operazioni di drag and drop (trasporto e rilascio).

La Figura 7.9 mostra un browser ad albero nel pannello di sinistra della finestra. Il pannello di destra è un browser a riga. Selezionando nel browser ad albero un nodo, cioè un record che rappresenta un gruppo agenzie, le agenzie che fanno parte di quel gruppo sono visualizzate nel browser a riga.

7.4.1.3 Pagina web

Se usata come punto d'ingresso a un'applicazione, una *pagina web* può anche essere trattata come una speciale tipologia di finestra primaria. Diversamente dalle applicazioni convenzionali, la barra menu e la barra degli strumenti della pagina web non sono usati per compiti applicativi ma per generiche attività di navigazione nel web. Gli eventi utente nelle applicazioni web sono normalmente programmati con pulsanti d'azione e legami (hyperlink) attivi.

La Figura 7.10 mostra la pagina web che è il punto d'ingresso al sito web della Biblioteca della Macquarie University. La barra menu e la barra degli strumenti non si applicano alla pagina web. I legami sono usati per la ricerca nel database della biblioteca e per chiedere in prestito sia libri sia altri oggetti presenti in biblioteca.

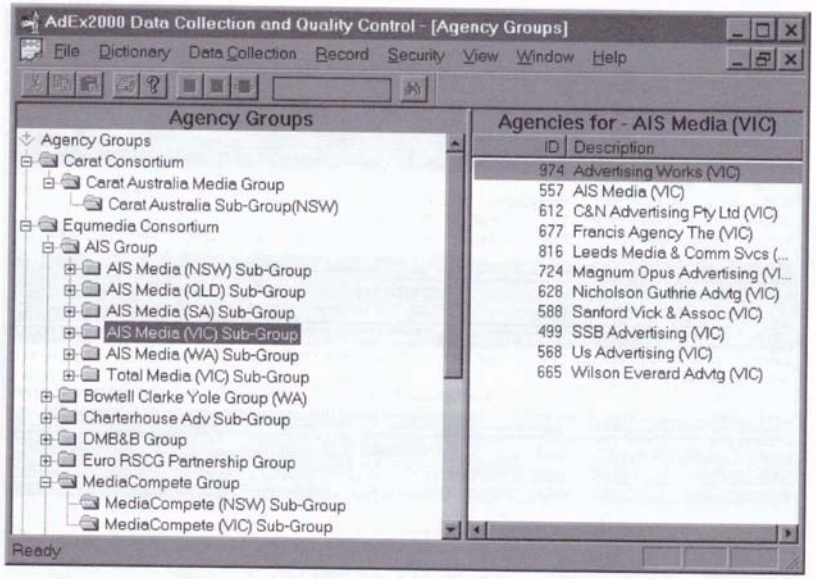


FIGURA 7.9 Finestra a pannelli con browser ad albero e a riga (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

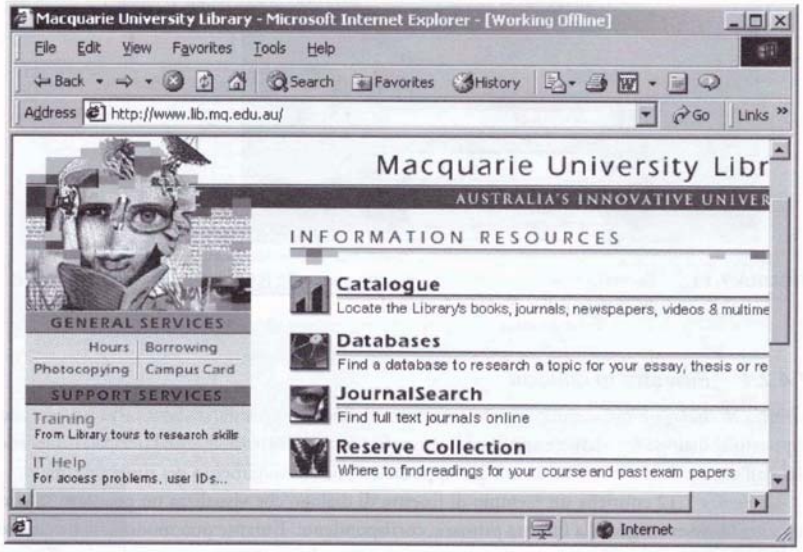


FIGURA 7.10 Finestra pagina web (per gentile concessione di Macquarie University, Sydney, Australia).

7.4.2 Finestre secondarie

Una *finestra secondaria* è, generalmente, di complemento alla propria finestra primaria, estendendone le funzionalità (in particolare, per le operazioni che modificano il database, cioè le operazioni di inserimento, cancellazione e aggiornamento).

Una finestra secondaria ha un comportamento tipicamente *modale* rispetto alla finestra primaria: l'utente deve rispondere e chiudere la finestra secondaria prima di poter interagire con qualunque altra finestra dell'applicazione (sono possibili finestre secondarie che non hanno un comportamento di questo tipo, ma non sono raccomandabili!).

Una *finestra d'accesso* (logon) è un semplice esempio di finestra secondaria. Nella Figura 7.11, le principali differenze riscontrabili visivamente tra una finestra primaria e una secondaria risultano evidenti. Una finestra secondaria non ha alcuna "barra" (di menu, di strumenti, di scorrimento o di stato). Gli eventi utente sono acquisiti tramite *pulsanti di comando* (*pulsanti d'azione*), quali `OK`, `Cancel`, `Help`.

Le finestre secondarie possono assumere varie forme:

- finestra di dialogo (dialog box);
- cartella a schede (tab folder);
- lista a tendina (drop-down list);
- finestra di messaggio (message box).

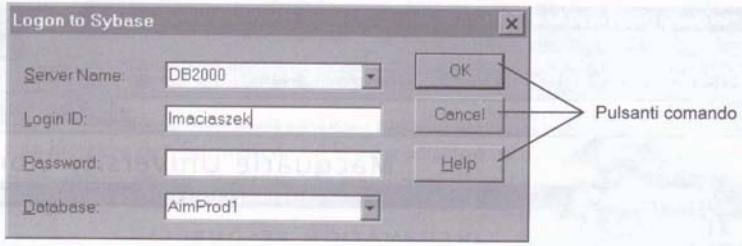


FIGURA 7.11 Semplice finestra secondaria – finestra d'accesso (logon) (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.4.2.1 Finestra di dialogo

Finestra di dialogo è quasi sinonimo del concetto di finestra secondaria. Essa offre un semplice supporto al dialogo tra utente e applicazione, implicando, in particolare, che l'informazione inserita dall'utente debba essere utilizzata dall'applicazione nello svolgimento dei suoi compiti.

La Figura 7.12 contiene un esempio di finestra di dialogo che visualizza un prodotto selezionato nel browser a riga della finestra primaria corrispondente. L'utente può modificare il valore di qualunque *campo modificabile* e quindi confermare la modifica (`OK`) o decidere di non effettuare modifiche (`Cancel`).

Esempio 7.3 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti (Sezione 2.3.3), i relativi esempi nel Capitolo 4, e gli Esempi 7.1 e 7.2 in questo Capitolo.

La finestra primaria per il caso CMS (Figura 7.6) non permette delle manipolazioni sugli eventi, per cui l'inserimento di un nuovo evento oppure l'aggiornamento di un evento esistente deve essere realizzato tramite una finestra di dialogo.

Una doppia pressione con il mouse su un evento nella finestra primaria dovrebbe far apparire la finestra di dialogo con tutti i dettagli di quell'evento. La finestra di dialogo non deve visualizzare solo le informazioni sull'evento ma anche l'organizzazione e il contatto cui l'evento si riferisce e i dati sul suo compito.

I dettagli dell'evento che possono essere visualizzati ed eventualmente modificati comprendono il tipo dell'evento, una sua descrizione, data, ora e impiegato relativi alla creazione, e i tempi previsti di trattamento, di consegna e di completamento.

Si progetti una finestra di dialogo per la manipolazione degli eventi conforme ai requisiti sopra descritti.

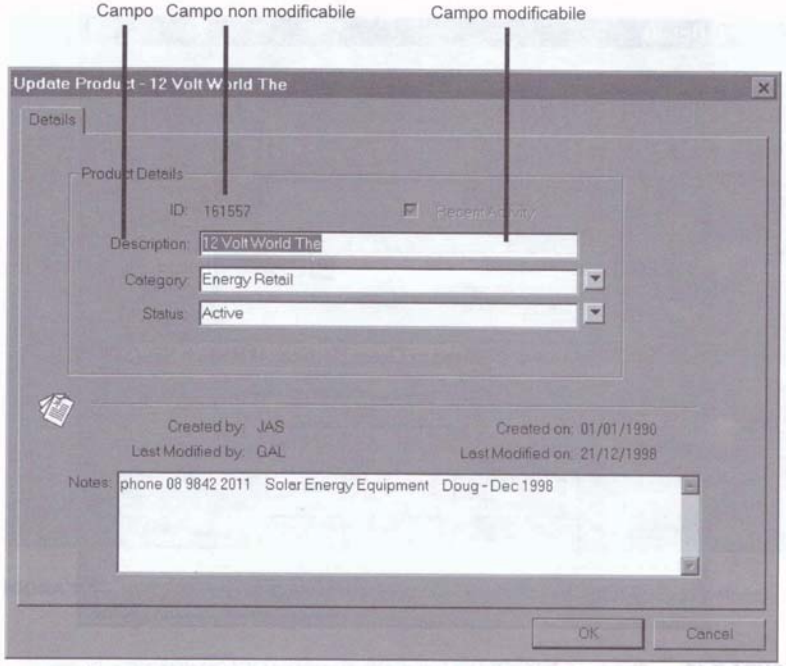


FIGURA 7.12 Finestra di dialogo (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

La Figura 7.13 è la soluzione proposta all'esempio. I dettagli dell'evento comprendono i campi `Organisation` e `Contact` che non sono modificabili (l'obiettivo dell'evento non può essere modificato). Analogamente, non sono modificabili i campi che seguono l'etichetta `Created`.

I campi valore adiacenti all'etichetta `Completed` non sono modificabili nel senso che l'utente non può scrivere al loro interno; tuttavia, premendo il pulsante `Complete` vi saranno inseriti automaticamente i valori di data, ora e codice impiegato validi in quel momento. Dopo il completamento dell'evento, il pulsante `Complete` è ridenominato `Uncomplete`, per cui l'impiegato ha ora la possibilità di renderlo "non completato" se ne ravvisa la necessità.

A questo punto l'impiegato ha la possibilità di salvare le modifiche nel database e tornare alla finestra primaria premendo il pulsante `OK`. In alternativa, può premere `Cancel` per cancellare le modifiche e rimanere nella finestra di dialogo. Infine, l'impiegato può premere il pulsante `New Event` che salva le modifiche (dopo conferma dell'utente), annulla tutti i campi nella finestra di dialogo e permette all'utente di creare un nuovo evento (senza tornare alla finestra primaria).

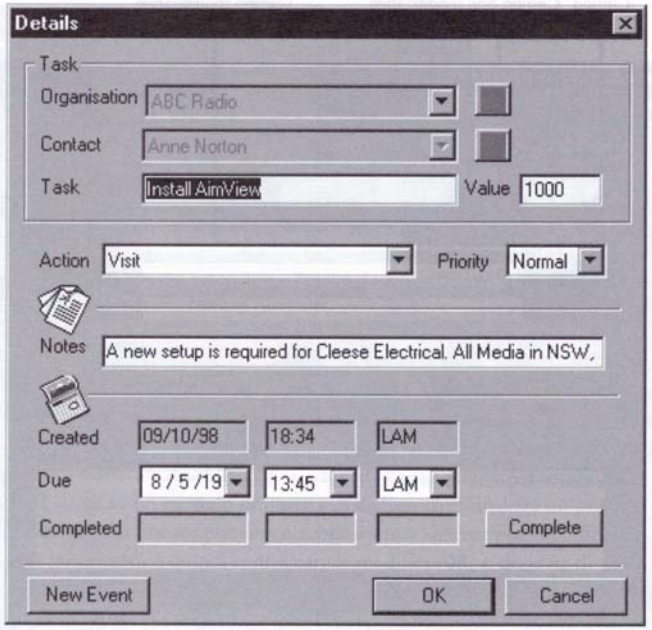


FIGURA 7.13 Finestra di dialogo (CMS) (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.4.2.2 *Cartella a schede*

Una *cartella a schede* (tab folder) è utile quando la quantità d'informazione che deve essere visualizzata in una finestra secondaria eccede l'area della finestra stessa e l'informazione può essere logicamente suddivisa in sezioni. In qualunque momento, l'informazione di una sezione è visibile in cima alla pila delle schede componenti (la nomenclatura Microsoft Windows per una cartella a schede è *tabbed property sheet* e ciascuna scheda è chiamata *property page*).

La Figura 7.14 mostra una cartella a schede per l'inserimento dei dati di una nuova organizzazione pubblicitaria, strutturati su quattro schede (`Organisation`, `Postal Address`, `Courier Address`, `Advertiser`). I pulsanti di comando in basso (`OK` e `Cancel`) agiscono sull'intera finestra, non solo sulla scheda attualmente visibile.

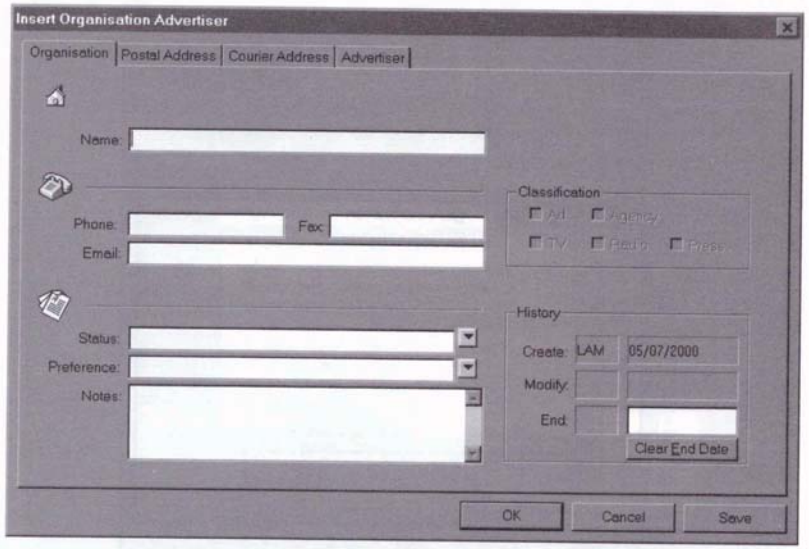


FIGURA 7.14 Cartella a schede (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

Esempio 7.4 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti (Sezione 2.3.3), i relativi esempi nel Capitolo 4, e gli Esempi 7.1, 7.2 e 7.3 in questo capitolo.

Si consideri, nella cartella a schede `Maintain Organisations` di Figura 7.2, la scheda `Contacts`: il suo scopo è permettere l'accesso e la modifica dei dati della classe `Contatto` (altrimenti, l'utente dovrebbe sempre tornare alla finestra primaria e attivare un'altra finestra secondaria).

Si progetti il contenuto della scheda `Contacts`.

Come mostrato in Figura 7.15, la scheda `Contacts` visualizza solo i nomi dei contatti nell'organizzazione. Tuttavia, la scheda ha il proprio insieme di pulsanti comando per aggiungere (`Add`), modificare (`Edit`), o cancellare (`Delete`) il contatto attualmente evidenziato. Un'azione per aggiungere (`Add`) o modificare (`Edit`) un contatto ha come risultato l'apertura e la sovrapposizione alla finestra `Maintain Organisations` di una finestra secondaria `Maintain Contacts` attraverso cui aggiornare i dati.

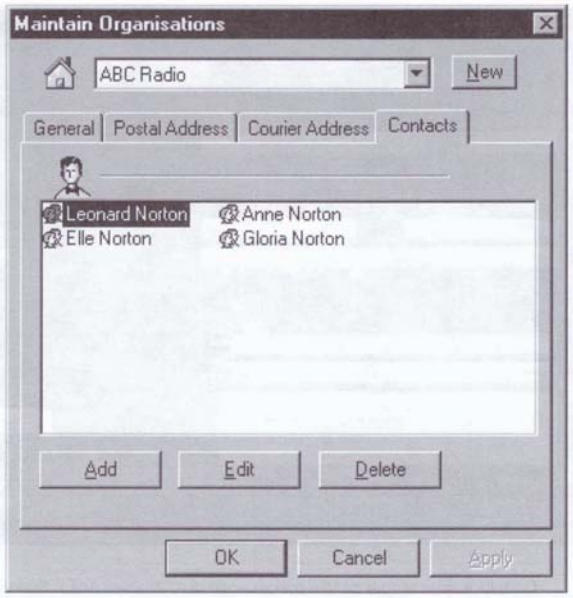


FIGURA 7.15 Cartella a schede (CMS) (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.4.2.3 Lista a tendina

In alcuni casi, una *lista a tendina* (o un insieme di liste a tendina) è un conveniente sostituto di una cartella a schede. Una tale lista fornisce un elenco di valori selezionabili da cui l'utente può scegliere quello da usare. Nelle operazioni di inserimento l'utente può aggiungere semplicemente un valore alla lista a tendina, valore che sarà visualizzato alla successiva apertura della lista.

Come mostrato nella Figura 7.16, una lista a tendina non deve necessariamente essere una semplice lista di valori, ma può anche essere un albero di valori.

7.4.2.4 Finestra di messaggio

Una *finestra di messaggio* è una finestra secondaria che visualizza un messaggio all'utente. Il messaggio può avere significati diversi, come richiamare l'attenzione dell'utente, fornire una spiegazione, segnalare una condizione eccezionale, etc. I pulsanti comando nella finestra messaggio offrono all'utente una o più scelte di replica.

La Figura 7.17 mostra un messaggio che richiede una conferma dall'utente (acknowledgment) tramite la pressione del pulsante OK.

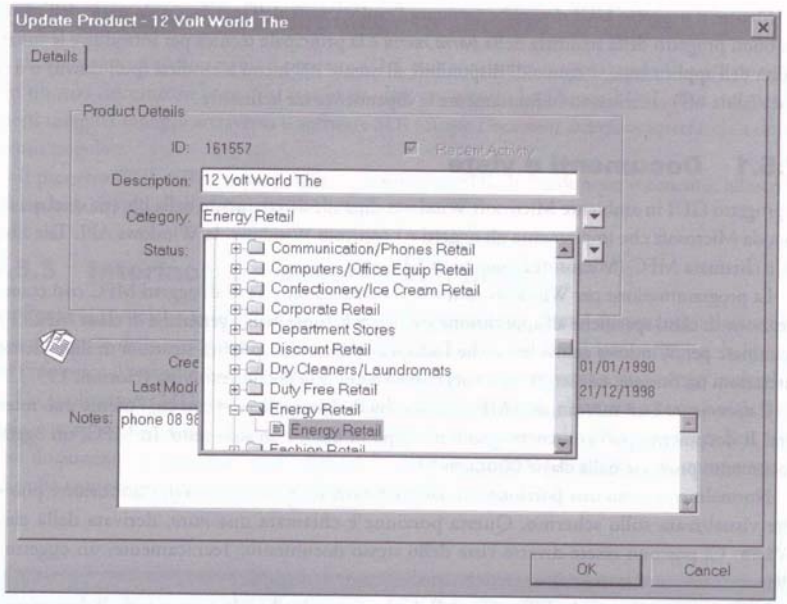


FIGURA 7.16 Lista a tendina (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.5 Dipendenze tra finestre

L'applicazione appare all'utente come un insieme di finestre in collaborazione. È compito del progettista GUI organizzare le dipendenze tra finestre in una struttura coerente e facile da comprendere. L'utente non dovrebbe mai sentirsi perduto tra le finestre aperte.

Idealmente, il legame tra la finestra primaria e la finestra secondaria correntemente attiva dovrebbe essere un cammino, non una gerarchia: ciò può essere ottenuto attraverso finestre secondarie di tipo *modale* rispetto alle finestre precedenti. Il trattamento che sarà discusso nella Sezione 7.5.3 (MDI - Multiple Document Interface) dovrebbe permettere di affrontare una qualunque situazione complessa senza la necessità di dover introdurre finestre non modali.

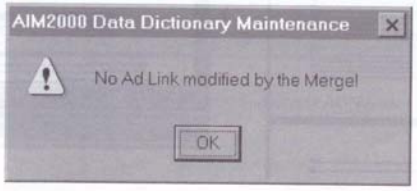


FIGURA 7.17 Finestra messaggio (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

Mentre il progetto GUI dovrebbe facilitare l'esplorazione dell'interfaccia da parte dell'utente, un buon progetto della struttura della *barra menu* è la principale tecnica per introdurre le funzionalità dell'applicazione. I comandi disponibili all'utente nei menu a tendina (pull-down) o a cascata (slide-off) chiariscono indirettamente le dipendenze tra le finestre.

7.5.1 Documenti e viste

Il progetto GUI in ambiente Microsoft Windows dipende direttamente dalla libreria di classi fornita da Microsoft che implementa gli oggetti e i controlli Windows, la Windows API. Tale libreria è chiamata MFC (Microsoft Foundation Classes).

La programmazione per Windows coinvolge l'istanziazione e l'uso di oggetti MFC così come la creazione di classi specifiche all'applicazione ereditanti le funzionalità generiche di classi MFC. Programmare per Windows coinvolge anche l'adeguamento a una specifica struttura di dipendenze e interazioni tra finestre: tale struttura è nota come l'approccio documento/vista (Horton, 1997).

Il *documento* è un meccanismo MFC di raccolta di dati applicativi con cui l'utente può interagire. Il documento può contenere qualunque tipo di dato, non solo testo. In MFC, un oggetto documento proviene dalla classe `CDocument`.

Normalmente, solo una porzione dei dati memorizzati in un oggetto di `CDocument` può essere visualizzata sullo schermo. Questa porzione è chiamata una *vista*, derivata dalla classe `CView`. Ci possono essere diverse viste dello stesso documento. Tecnicamente, un oggetto di `CView` e la finestra in cui tale oggetto è visualizzato sono distinti.

La Figura 7.18 mostra la differenza tra il documento e la vista. In questo caso, il documento è un documento testuale ma in generale potrebbe essere qualunque archivio di informazione estratta dal database.

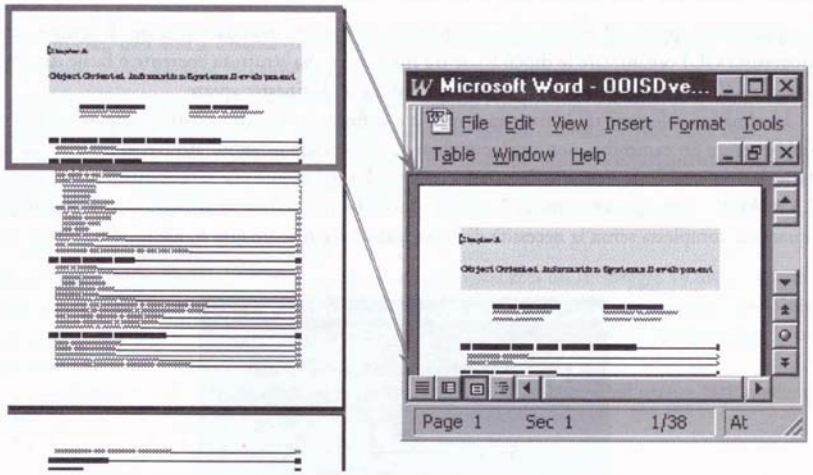


FIGURA 7.18 Il documento e una sua vista.

7.5.2 Interfaccia a documento singolo

In alcune semplici applicazioni, il progetto GUI può consistere di una singola finestra primaria con un solo documento aperto in essa in un dato momento. La libreria MFC offre supporto a questi semplici sviluppi attraverso il software SDI (*Single Document Interface*, interfaccia a documento singolo).

Il progetto GUI dell'esempio CMS usa il software SDI. In qualunque momento, all'utente sono presentati gli eventi della giornata (Figura 7.19).

7.5.3 Interfaccia a documenti multipli

Applicazioni complesse richiedono l'apertura contemporanea di diversi documenti (i documenti possono essere dello stesso tipo, ma spesso sono di tipi diversi). Per il loro sviluppo, la libreria MFC fornisce il software MDI (*Multiple Document Interface*, interfaccia a documenti multipli).

Un'applicazione MDI usa ancora *una singola finestra primaria*, chiamata *finestra radice*. Tuttavia, è possibile l'apertura di molti documenti all'interno della cornice della finestra radice. Ciascun documento è chiamato *finestra figlia* e, logicamente, ognuna di queste può comportarsi a sua volta come una finestra radice rispetto a sue eventuali sottofinestre.

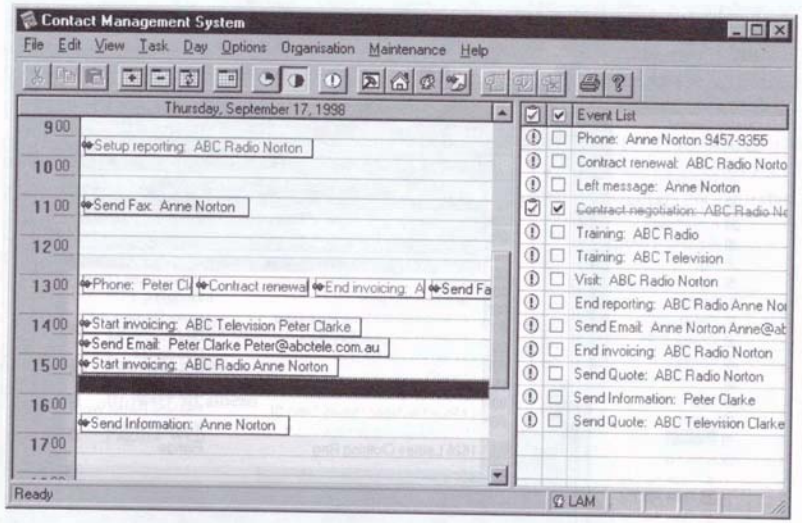


FIGURA 7.19 Un'applicazione SDI (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

Il fatto che il framework MDI abbia solo una singola finestra primaria al suo interno è testimoniato dalla presenza di *una sola barra menu* condivisa da tutte le finestre figlia. Analogamente, le finestre figlia condividono le barre degli strumenti e di stato della radice.

La Figura 7.20 mostra un'applicazione MDI in cui sono aperti quattro documenti (browser a riga) all'interno della cornice della finestra primaria; il quinto documento, sulla sinistra, è un browser ad albero.

7.6 Navigazione nelle finestre

La rappresentazione grafica (statica) delle finestre GUI non dà informazione su come l'utente possa *navigare* tra le finestre. È quindi necessario progettare un *sistema di navigazione nelle finestre*. Un diagramma di navigazione nelle finestre dovrebbe visualizzare le finestre dell'applicazione e gli oggetti di controllo che permettono all'utente di passare da una finestra all'altra.

Sfortunatamente, UML non ha un modello grafico per rappresentare questo tipo di navigazione. È quindi necessario sviluppare un proprio modello, oppure usare gli stereotipi UML e configurare uno dei diagrammi UML per rappresentare la navigazione nelle finestre.

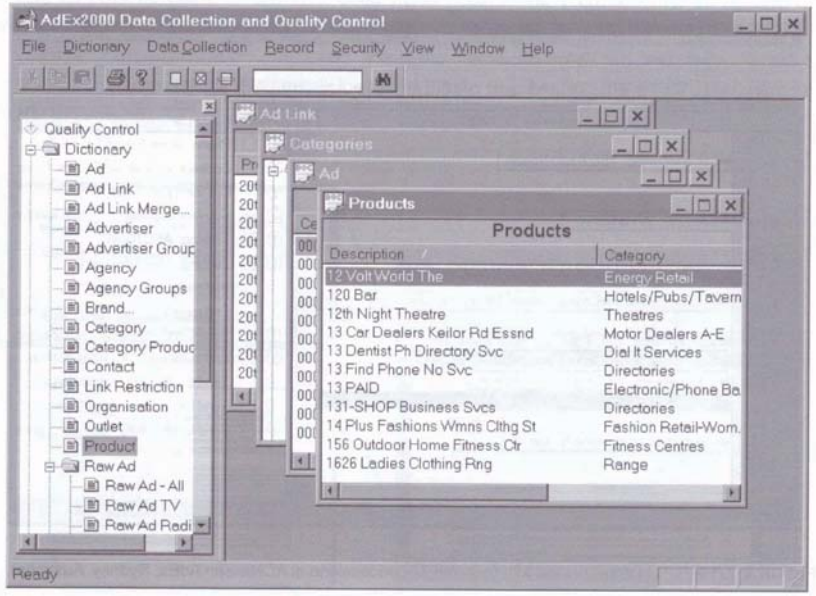


FIGURA 7.20 Applicazione MDI (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.6.1 Stereotipare il diagramma d'attività per la navigazione nelle finestre

Il diagramma d'attività di UML è un buon candidato da stereotipare al fine di rappresentare la navigazione nelle finestre. Il diagramma d'attività mostra le transizioni tra *attività* (Sezione 2.2.3.2); tuttavia, essendo comunque un tipo di macchina a stati, può anche rappresentare *stati* d'oggetti. Questa dualità dei diagrammi d'attività può essere usata per rappresentare graficamente l'analoga dualità richiesta dagli oggetti GUI: una finestra GUI è simile a uno *stato* in attesa di eventi (*attività*).

Gli stati e le attività nel diagramma d'attività possono essere stereotipati. Gli *stereotipi di stato* possono identificare diverse tipologie di finestra e altri oggetti GUI che persistono (hanno una durata) tra un evento e l'altro. La Figura 7.21 mostra uno stato, stereotipato come finestra primaria, che rappresenta un *browser di prodotti* visualizzati nella finestra primaria dell'applicazione. Nel modello è indicato anche lo *stato iniziale*.

Gli *stereotipi d'attività* possono identificare differenti tipi di controllo GUI che possono essere usati per attivare eventi sugli stati (ora stereotipati come oggetti finestra GUI). All'opposto degli stati, le attività sono considerate di durata trascurabile rispetto alla scala temporale di riferimento.

Le attività (ovali) possono essere rappresentate internamente allo stato cui si applicano. La Figura 7.22 mostra tre attività interne allo stato (rettangolo a bordi smussati) `Browser Prodotti`, stereotipato come finestra primaria; la finestra può essere usata per attivare eventi di inserimento, cancellazione o aggiornamento di un prodotto. Tutti e tre gli eventi possono essere attivati da un pulsante d'azione di uno strumento di libreria (toolbox button) o da un elemento di menu (l'evento d'aggiornamento può essere attivato anche tramite una doppia pressione con il mouse su una riga prodotto).

L'elenco completo di stereotipi d'attività e di stato per il supporto al progetto di un sistema navigazione finestre dipende dall'interfaccia GUI selezionata. Gli sviluppatori potrebbero anche usare stereotipi icona. Di seguito è riportata una lista (incompleta) di stereotipi per l'interfaccia GUI di Microsoft Windows:

- *Stati (finestre)*
  - Finestra primaria
    - Pannello nella finestra primaria
    - Browser a riga
    - Browser ad albero
    - Pagina Web

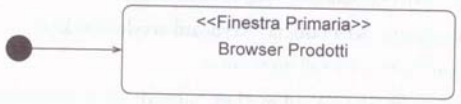


FIGURA 7.21 Stato stereotipato come oggetto GUI.

Il fatto che il framework MDI abbia solo una singola finestra primaria al suo interno è testimoniato dalla presenza di *una sola barra menu* condivisa da tutte le finestre figlia. Analogamente, le finestre figlia condividono le barre degli strumenti e di stato della radice.

La Figura 7.20 mostra un'applicazione MDI in cui sono aperti quattro documenti (browser a riga) all'interno della cornice della finestra primaria; il quinto documento, sulla sinistra, è un browser ad albero.

7.6 Navigazione nelle finestre

La rappresentazione grafica (statica) delle finestre GUI non dà informazione su come l'utente possa *navigare* tra le finestre. È quindi necessario progettare un *sistema di navigazione nelle finestre*. Un diagramma di navigazione nelle finestre dovrebbe visualizzare le finestre dell'applicazione e gli oggetti di controllo che permettono all'utente di passare da una finestra all'altra.

Sfortunatamente, UML non ha un modello grafico per rappresentare questo tipo di navigazione. È quindi necessario sviluppare un proprio modello, oppure usare gli stereotipi UML e configurare uno dei diagrammi UML per rappresentare la navigazione nelle finestre.

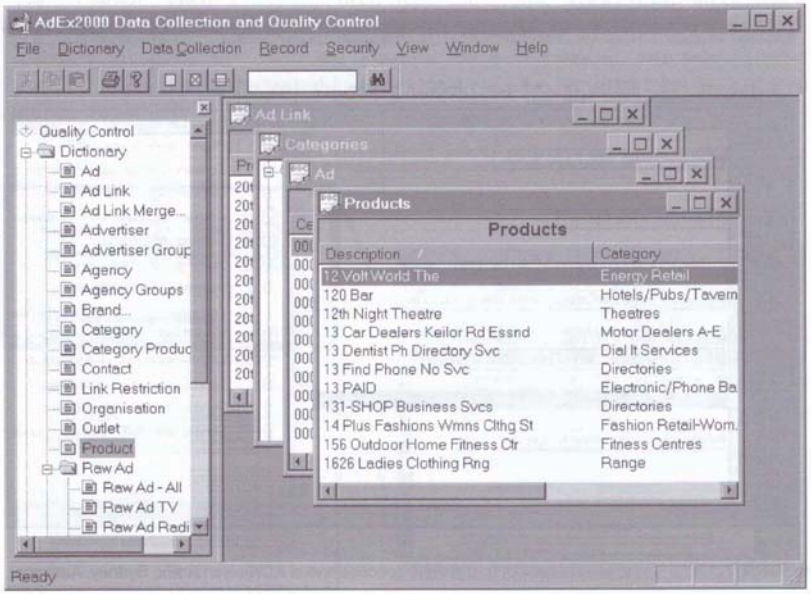


FIGURA 7.20 Applicazione MDI (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

7.6.1 Stereotipare il diagramma d'attività per la navigazione nelle finestre

Il diagramma d'attività di UML è un buon candidato da stereotipare al fine di rappresentare la navigazione nelle finestre. Il diagramma d'attività mostra le transizioni tra *attività* (Sezione 2.2.3.2); tuttavia, essendo comunque un tipo di macchina a stati, può anche rappresentare *stati* d'oggetti. Questa dualità dei diagrammi d'attività può essere usata per rappresentare graficamente l'analoga dualità richiesta dagli oggetti GUI: una finestra GUI è simile a uno *stato* in attesa di eventi (*attività*).

Gli stati e le attività nel diagramma d'attività possono essere stereotipati. Gli *stereotipi di stato* possono identificare diverse tipologie di finestra e altri oggetti GUI che persistono (hanno una durata) tra un evento e l'altro. La Figura 7.21 mostra uno stato, stereotipato come finestra primaria, che rappresenta un *browser di prodotti* visualizzati nella finestra primaria dell'applicazione. Nel modello è indicato anche lo *stato iniziale*.

Gli *stereotipi d'attività* possono identificare differenti tipi di controllo GUI che possono essere usati per attivare eventi sugli stati (ora stereotipati come oggetti finestra GUI). All'opposto degli stati, le attività sono considerate di durata trascurabile rispetto alla scala temporale di riferimento.

Le attività (ovali) possono essere rappresentate internamente allo stato cui si applicano. La Figura 7.22 mostra tre attività interne allo stato (rettangolo a bordi smussati) `Browser Prodotti`, stereotipato come finestra primaria; la finestra può essere usata per attivare eventi di inserimento, cancellazione o aggiornamento di un prodotto. Tutti e tre gli eventi possono essere attivati da un pulsante d'azione di uno strumento di libreria (toolbox button) o da un elemento di menu (l'evento d'aggiornamento può essere attivato anche tramite una doppia pressione con il mouse su una riga prodotto).

L'elenco completo di stereotipi d'attività e di stato per il supporto al progetto di un sistema navigazione finestre dipende dall'interfaccia GUI selezionata. Gli sviluppatori potrebbero anche usare stereotipi icona. Di seguito è riportata una lista (incompleta) di stereotipi per l'interfaccia GUI di Microsoft Windows:

- *Stati* (*finestre*)
  - Finestra primaria
    - Pannello nella finestra primaria
    - Browser a riga
    - Browser ad albero
    - Pagina Web

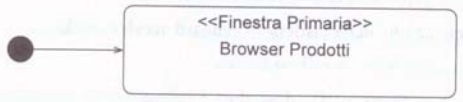


FIGURA 7.21 Stato stereotipato come oggetto GUI.

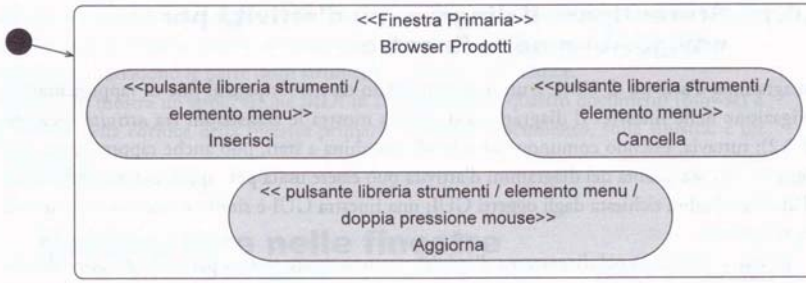


FIGURA 7.22 Attività stereotipate come oggetti di controllo GUI.

- Finestra secondaria
  - Finestra di dialogo
  - Finestra di messaggio
  - Cartella a schede
- Dati Window
  - Controllo di testo (Text box)
  - Controllo composto (Combo box)
  - Pulsanti di variazione (Spin box)
  - Colonna
  - Riga
  - Gruppo di campi
- *Attività (controlli finestra)*
  - Elemento di menu a tendina (Drop-down menu item)
  - Elemento di menu ad apparizione (Pop-up menu item)
  - Pulsante barra strumenti (Toolbar button)
  - Pulsante comando (Command button)
  - Doppia pressione mouse (Double click)
  - Selezione da lista (Picklist selection)
  - Tasto della tastiera (Keyboard key)
  - Tasto funzione della tastiera (Keyboard function key)
  - Tasto di accelerazione della tastiera (Keyboard accelerator key)
  - Pulsante di scorrimento (Scrolling button)
  - Pulsante chiusura finestra (Window close button)

7.6.2 Diagramma di navigazione nelle finestre

Una volta determinati gli stereotipi di stato e d'attività, è possibile usare le *transizioni* per connettere attività e stati. Il risultato è un diagramma di navigazione nelle finestre in cui le attività fanno scattare le transizioni di stato.

La Figura 7.23 estende il precedente esempio e mostra gli stati risultanti dall'esecuzione delle tre attività disponibili nella finestra `Browser Prodotti`. Lo stato raggiunto con l'evento `Aggiorna` è espanso nella finestra `Aggiorna Prodotto` (una `<<finestra di dialogo>>`) che a sua volta contiene quattro attività (stereotipate come `<<pulsante comando>>`). Premendo il pulsante `OK` oppure `Cancella` si causa una transizione di ritorno alla finestra `Browser Prodotti`. Premendo il pulsante `Salva` oppure `Annulla` non si modifica la finestra attiva (se si volesse rappresentare la modifica di stato all'interno della finestra attiva, sarebbe necessario estendere il modello con ulteriori stereotipi).

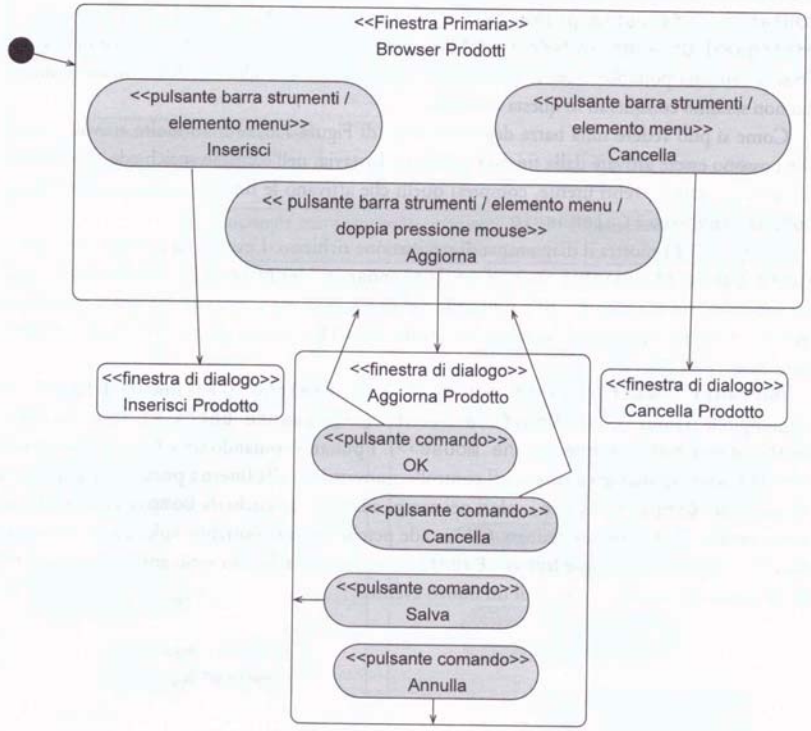


FIGURA 7.23 Diagramma di navigazione nelle finestre.

Il modello nella Figura 7.23 è limitato agli oggetti GUI. Nel Capitolo 9 sarà mostrato come i diagrammi d'attività possono essere ulteriormente stereotipati per visualizzare l'intera logica applicativa, compresi gli accessi ai database.

Esempio 7.5 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti (Sezione 2.3.3), i relativi esempi nel Capitolo 4, e gli Esempi 7.1, 7.2 e 7.3 in questo capitolo.

Sviluppare un diagramma di navigazione nelle finestre per le finestre in Figura 7.6 (Esempio 7.2) e in Figura 7.13 (Esempio 7.3). Il diagramma dovrebbe identificare e modellare i principali eventi utente sulla finestra primaria (Figura 7.6) e sulla finestra di dialogo (Figura 7.13). Dovrebbe anche modellare l'uso di un Calendario (un controllo composto).

Il diagramma di navigazione richiesto dall'esempio dovrebbe considerare le tre finestre: `Gestione Contatti` (`<<finestra primaria>>`), `Dettagli Compito/Evento` (`<<finestra di dialogo>>`), `Calendario` (`<<controllo composto>>`). I pannelli di sinistra e destra della finestra primaria potrebbero essere considerati come possibili sottofinestre della finestra primaria, ma non saranno considerati in questa soluzione.

Come si può vedere sulla barra degli strumenti di Figura 7.6, ci sono molte attività (eventi) che possono essere attivate dalla finestra primaria. Tuttavia, nell'esempio si richiede solo di identificare i principali eventi utente, compresi quelli che attivano le finestre secondarie `Dettagli Compito/Evento` e `Calendario`.

La Figura 7.24 mostra il diagramma di navigazione richiesto. L'evento `Calendario` (`<<pulsante barra strumenti>>`) visualizza il `Calendario` (`<<controllo composto>>`); questo controllo (si veda Figura 7.6) è riducibile e si può rimuovere premendo il relativo pulsante di chiusura finestra. Selezionare un mese (`<<scorrimento>>`) o un giorno (`<<selezione>>`) non rimuove il controllo.

`Dettagli Compito/Evento` (`<<finestra di dialogo>>`) è la finestra di Figura 7.13 raggiungibile tramite l'evento `Modifica Evento` (`<<pulsante barra strumenti/elemento menu/doppia pressione mouse>>`). I pulsanti comando `OK` e `Cancella` rimuovono la finestra di dialogo e passano il controllo nuovamente alla finestra primaria. L'attivazione del pulsante `Completo` scrive i valori nei campi adiacenti all'etichetta `Completed` ma il controllo rimane alla finestra di dialogo. Ciò accade perché l'utente potrebbe voler creare un nuovo evento. Premendo il pulsante `Nuovo Evento` i campi sullo schermo sono annullati permettendo all'utente di inserire i dettagli di un nuovo evento.

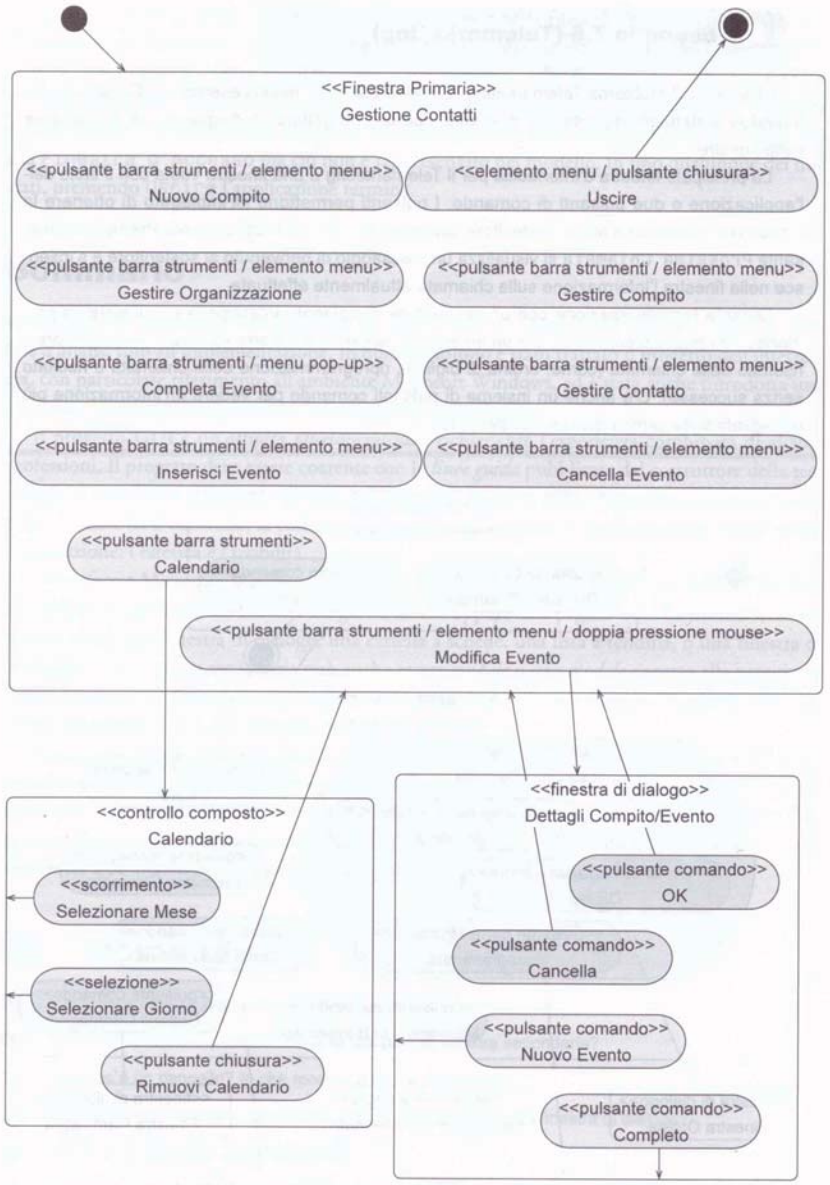


FIGURA 7.24 Diagramma di navigazione nelle finestre (CMS).

Esempio 7.6 (Telemarketing)

Si considerino il problema Telemarketing (Sezione 2.3.4) e i relativi esempi nel Capitolo 4. Riferendosi ai dettagli aggiuntivi qui di seguito riportati, progettare un diagramma di navigazione nelle finestre.

La principale finestra d'interfaccia per il Telemarketing è una finestra vuota con il titolo dell'applicazione e due pulsanti di comando. I pulsanti permettono all'impiegato di ottenere la `Prossima Chiamata` ad un sostenitore oppure di `Uscire` dall'applicazione. Premendo il pulsante `Prossima Chiamata` si visualizza un messaggio di benvenuto al sostenitore e s'inserisce nella finestra l'informazione sulla chiamata attualmente effettuata.

Durante la conversazione con un sostenitore, l'impiegato interagisce con il sistema premendo vari pulsanti comando. C'è un insieme di pulsanti nella barra strumenti per registrare il risultato della chiamata (come: ordine di biglietti, riprogrammazione della chiamata o risultato senza successo). C'è anche un insieme di pulsanti comando per vedere un'informazione più dettagliata sulla campagna o sul sostenitore.

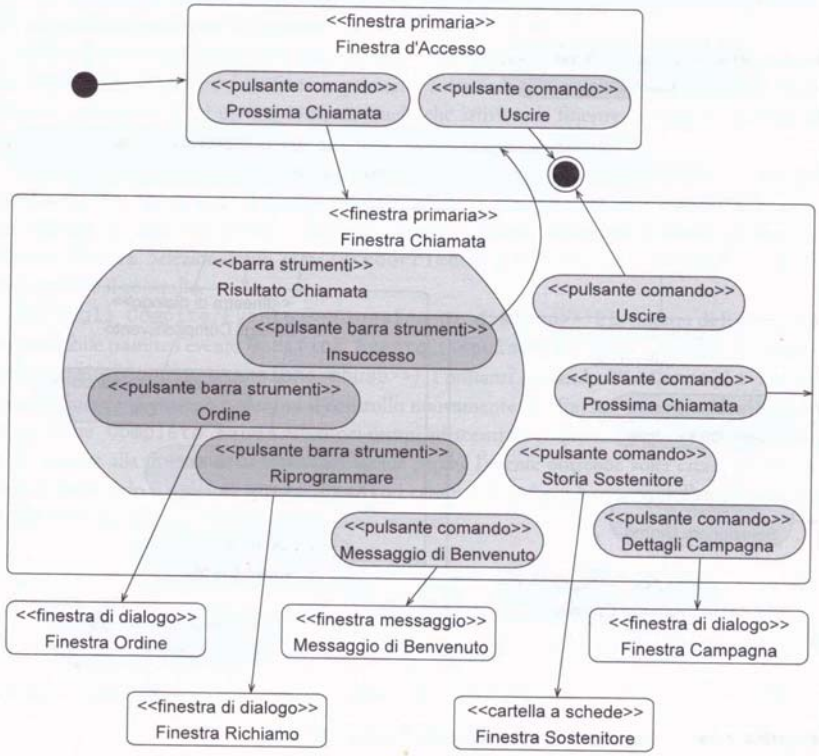


FIGURA 7.25 Diagramma di navigazione nelle finestre (Telemarketing).

La Figura 7.25 presenta il diagramma di navigazione nelle finestre dell'esempio. Sebbene la `Finestra d'Accesso` e la `Finestra Chiamata` siano la stessa `<<finestra primaria>>`, sono esplicitamente rappresentate come due stati della finestra. Lo stato `Finestra d'Accesso` è attivo dopo il lancio iniziale del programma e quando il risultato di una chiamata è `Insuccesso`. Probabilmente gli eventi nelle finestre secondarie possono restituire il controllo alla `Finestra d'Accesso` ma ciò non è rappresentato nel modello. In uno qualunque dei due stati, premendo `Uscire` l'applicazione termina.

Sommario

Lo sviluppo di interfacce GUI è presente in tutto il ciclo di sviluppo del software, a partire dalla fase d'analisi fino all'implementazione. In questo capitolo è stato trattato il progetto dell'interfaccia, con particolare riferimento all'ambiente Microsoft Windows, ed è stata anche introdotta una notazione grafica per rappresentare la navigazione nelle finestre.

Il progetto GUI è un'*attività pluridisciplinare* richiedente l'esperienza combinata di alcune professioni. Il progetto deve essere coerente con le *linee guida* pubblicate dal costruttore della tecnologia d'interfaccia a finestre adottata nel progetto. Le linee guida riguardano alcuni principi quali: l'utente ha il controllo, la consistenza, la personalizzazione, la configurazione, la tolleranza, la retroazione, l'estetica e l'usabilità.

L'interfaccia Microsoft Windows distingue tra finestra primaria e finestra secondaria. La *finestra primaria* può essere un browser a riga, ad albero oppure una pagina web. La *finestra secondaria* può essere una finestra di dialogo, una cartella a schede, una lista a tendina, o una finestra di messaggio. Una finestra secondaria può anche essere *modale* o *non modale* rispetto alla propria finestra primaria. Le dipendenze tra finestre sono programmabili, in ambiente Windows, per mezzo dell'*interfaccia SDI* o dell'*interfaccia MDI*.

Il progetto visuale di singole finestre è solo un aspetto dello sviluppo di interfacce GUI. Il secondo aspetto è in relazione alla *navigazione nelle finestre* che rappresenta i possibili cammini di navigazione attraverso le finestre dell'applicazione. In questo capitolo, sono stati introdotti i diagrammi di navigazione nelle finestre che estendono i diagrammi d'attività UML. In effetti, corrispondono a un nuovo *profilo UML* per la rappresentazione della navigazione finestre.

Domande

D1 Elencare e definire brevemente le linee guida di un progetto GUI.

D2 In cosa una finestra primaria si differenzia da una finestra secondaria?

D3 Che cos'è un pannello? In che modo è utile nel progetto GUI?

D4 Che cos'è una cartella a schede? In cosa si differenzia da una finestra di dialogo?

D5 Che cos'è l'approccio documento/vista?

D6 L'interfaccia MDI permette più d'una finestra primaria? Spiegare.

D7 Come sono rappresentati finestre e controlli finestra nei diagrammi di navigazione nelle finestre? Tale rappresentazione è consistente con l'obiettivo dei diagrammi di attività UML? Spiegare.

Esercizi

Requisiti aggiuntivi (Telemarketing)

Si considerino i seguenti requisiti aggiuntivi per il Telemarketing:

1. La finestra Controllo Telemarketing è l'interfaccia di controllo primaria per l'applicazione Telemarketing. La finestra visualizza all'impiegato la lista delle chiamate attualmente in coda. Quando l'impiegato richiede una chiamata, il sistema stabilisce la connessione e l'impiegato è in grado di trattare la chiamata. Sullo schermo è visualizzata una finestra Sintesi Chiamata che mostra il tempo d'inizio, il tempo di fine e la durata della chiamata in corso.
2. Una volta stabilita una connessione, la finestra Controllo Telemarketing visualizza le informazioni sulla chiamata in corso: chi è stato chiamato, per quale campagna, quale tipo di chiamata si sta facendo.
3. In qualunque momento della conversazione, l'impiegato può vedere la storia del sostenitore (finestra Storia Sostenitore) rispetto alle campagne precedenti. Analogamente, possono essere visti i dettagli sulla campagna cui la chiamata in corso si riferisce (finestra Campagna).
4. L'interfaccia GUI permette una registrazione veloce dei possibili risultati della chiamata: piazzamento (sono stati ordinati dei biglietti), richiamare, insuccesso, nessuna risposta, impegnato, segreteria, fax, errore (cioè numero errato), e disconnesso.
5. La finestra Campagna visualizza i dettagli della campagna, i dettagli dei biglietti, e i dettagli dei premi per la campagna. I dettagli della campagna comprendono il numero d'identificazione, il titolo, le date d'inizio, di chiusura e di estrazione dei premi. I dettagli dei biglietti comprendono il numero di biglietti nella campagna, quanti sono stati venduti e quanti sono ancora disponibili. I dettagli dei premi comprendono la descrizione, il valore e la posizione dei premi (primo, secondo o terzo).
6. La finestra Storia Sostenitore mostra la storia delle chiamate passate e la storia delle campagne a cui il sostenitore ha partecipato. La storia della chiamate elenca le chiamate recenti, i tipi di queste chiamate, i risultati, e l'identificazione delle chiamate e degli impiegati. La storia delle campagne informa sulla vendita biglietti e sulle vincite relative al sostenitore.
7. Selezionando l'azione Piazza, si attiva la finestra Piazzamento che permette all'impiegato l'assegnazione di biglietti al sostenitore e la registrazione del pagamento.
8. Selezionando una delle azioni: Nessuna Risposta, Impegnato, Segreteria, Fax, Disconnesso o Errore, il risultato "nessuna risposta", "impegnato", "segreteria", "fax", "disconnesso" o "errore" è, rispettivamente, registrato nel sistema per la chiamata in corso. Le chiamate non andate a buon fine (primi tre casi) sono riprogrammate dal sistema per il giorno successivo (purché non si sia superato il limite di tentativi definito per quel tipo di chiamata), mentre negli ultimi tre casi, si aggiorna il telefono del sostenitore attuale a "numero errato".
9. Selezionando l'azione Richiamare, il risultato "richiamare" è registrato nel sistema e la finestra Programmazione Chiamate è invocata al fine di ottenere la data e l'ora per il richiamo da inserire. Le chiamate sono quindi riprogrammate dal sistema (con le nuove priorità).
10. Uscendo dalla finestra Piazzamento, se tutti i biglietti restanti nella campagna sono stati venduti, le chiamate residue ai sostenitori per quella campagna sono rimosse dalla coda chiamate.

E1 Telemarketing – si considerino i requisiti sopra descritti e le soluzioni al caso di studio Telemarketing. Si consideri anche il diagramma delle classi definito nell'Esempio 4.7 (Sezione 4.2.1.2.3).

Modificare ed estendere il diagramma delle classi di Figura 4.4, integrando i requisiti aggiuntivi sopra specificati.

E2 Telemarketing – si considerino i requisiti sopra descritti e le soluzioni al caso di studio Telemarketing.

Progettare e abbozzare la finestra primaria per il caso Telemarketing, cioè la finestra Controllo Telemarketing. Spiegare come questa finestra soddisfa i requisiti applicativi.

E3 Telemarketing – si considerino i requisiti sopra descritti e le soluzioni al caso di studio Telemarketing.

Progettare e abbozzare la finestra Storia Sostenitore. Spiegare come questa finestra soddisfa i requisiti applicativi.

E4 Telemarketing – si considerino i requisiti sopra descritti e le soluzioni al caso di studio Telemarketing.

Costruire il diagramma di navigazione nelle finestre per l'applicazione Telemarketing (non farsi influenzare eccessivamente dalla soluzione all'Esempio 7.6 - Sezione 7.6.2). Se richiesto dalla complessità della soluzione, suddividere il diagramma in diagrammi più semplici. Spiegare come la vostra soluzione soddisfa i requisiti applicativi.

E5 Acquisti OnLine – si consideri la soluzione del caso di studio Acquisti OnLine.

Progettare e abbozzare la pagina web che visualizza lo stato degli ordini correnti al cliente. Spiegare le eventuali complicazioni.

E6 Acquisiti OnLine - si consideri la soluzione del caso di studio Acquisti OnLine.

Rappresentare il diagramma di navigazione nelle finestre per l'applicazione Acquisti OnLine. Se richiesto dalla complessità della soluzione suddividere il diagramma in diagrammi ridotti. Spiegare come la vostra soluzione soddisfa i requisiti applicativi.

Capitolo 8

Progetto del database

I sistemi informativi sono, per la maggior parte, multi-utente. Questa caratteristica, da sola, richiede un database cui diversi utenti e programmi applicativi possano accedere concorrentemente. I programmi applicativi dipendono dal database mentre il contrario non è vero. La conseguenza è ovvia: un buon progetto di database, che possa favorire e supportare *tutti* i programmi applicativi, è condizione necessaria affinché un sistema informativo fornisca le funzionalità volute.

In UML, i diagrammi delle classi definiscono le strutture dati richieste da un'applicazione. Le strutture dati persistenti (rappresentate nel progetto UML come classi entity e come relazioni tra queste classi) saranno memorizzate nel database di sistema come strutture diverse in funzione del modello sottostante al database, che può essere object-oriented, object-relational o relazionale.

Questo capitolo tratta l'implementazione degli oggetti nei database. In particolare, si analizza la conversione di classi entity, associazioni, aggregazioni e generalizzazioni nelle strutture dati disponibili nei tre modelli di dati sopra citati. Non essendo l'obiettivo di questo testo, non saranno trattati i problemi relativi all'integrazione di schemi, cioè l'integrazione di diagrammi di classi con dati in comune, derivati dalle esigenze di applicazioni diverse sullo stesso database.

8.1 Il database

In tutto il testo, lo sviluppo di un'applicazione client è stato separato dal progetto del database server. Dunque, i modelli delle classi e i package BCED (Sezione 6.1.3.2) riflettono le strutture applicative di dati e non le strutture di memorizzazione proprie del database.

Le classi del package *entity* rappresentano nell'applicazione gli oggetti del database persistenti ma *non* sono le strutture persistenti del database. Le classi nel package *Database* incapsulano la comunicazione tra l'applicazione e il database ma nessuna di loro è una struttura persistente. Perciò, è necessario a questo punto progettare lo *strato persistente* di memorizzazione, cioè lo schema del database.

Il database può essere gestito da sistemi basati sul modello relazionale (ad esempio, Sybase, DB2, Oracle8), object-relational (ad esempio, UniSQL, Oracle8) o a oggetti (ad esempio, ObjectStore, Versant). È improbabile che il modello di memorizzazione per un nuovo sistema possa essere uno qualunque dei modelli più datati, quali il gerarchico (ad esempio, IMS), il reticolare (ad esempio, IDMS), a indici invertiti o simili (ad esempio, Total, Adabas). In alcuni casi, ma in pratica non nelle moderne applicazioni dei sistemi informativi, la persistenza può essere implementata in semplici file.

8.1.1 Modelli dei dati

La ricerca nel campo dei database ha storicamente sviluppato la propria visione della realtà attraverso *modelli di dati* (cioè, secondo UML, modelli dello stato). Gli attuali DBMS, hanno esteso le capacità dei database includendo *modelli di comportamento*, con la possibilità di *memorizzare ed eseguire programmi* (trigger e procedure memorizzate); la rappresentazione dei dati resta comunque l'aspetto più caratteristico nello sviluppo di un database.

Lo *schema del database* è un'astrazione che presenta le strutture del database nei termini del modello di dati scelto. L'architettura comune di riferimento per i DBMS prevede tre livelli (esterno, logico e fisico) e ciascun livello prevede un modello di dati adeguato:

1. modello dei dati esterno (concettuale);
2. modello dei dati logico;
3. modello dei dati fisico.

Lo *schema esterno* rappresenta, nel *modello concettuale* ad alto livello, i dati richiesti da una singola applicazione. Un database normalmente offre supporto a molte applicazioni: quindi sono costruiti anzitutto i corrispondenti schemi esterni di applicazione che, in seguito, sono integrati in un unico schema concettuale dei dati. La tecnica più comune per la rappresentazione concettuale dei dati è il diagramma ER (Entity-Relationship) (ad esempio, Maciaszek, 1990).

Lo *schema logico* (talvolta chiamato anche schema concettuale *globale*) fornisce una rappresentazione integrata dei dati memorizzati nel database, di supporto a qualunque applicazione esistente o prevista.

Lo *schema fisico* è specifico di un particolare DBMS. Definisce come i dati sono, di fatto, memorizzati sui dispositivi di memorizzazione persistente (tipicamente, dischi). Lo schema fisico definisce aspetti quali l'uso di indici e il clustering dei dati per un'elaborazione efficiente.

Gli strumenti CASE di supporto al progetto e all'implementazione del sistema forniscono, in genere, una singola tecnica di rappresentazione dei dati per gli schemi logico e fisico. Tali strumenti tendono a chiamare tale modello combinato come modello dei dati fisico.

La Figura 8.1 mostra come i modelli UML di un'applicazione si relazionino con i modelli del database. Le classi del package *entity* rappresentano gli "oggetti di business" di un'applicazione, il relativo diagramma delle classi UML può sostituire un diagramma ER come strumento per la rappresentazione concettuale del database.

Il package *Database* non "guida" la rappresentazione del database: piuttosto, è guidato da quest'ultima. Il package *Database*, stabilendo la relazione tra gli oggetti e il database, "disaccoppia" le classi entity dallo schema del database: quindi, può essere progettato in parallelo o successivamente alla definizione stessa del database.

8.1.2 Corrispondenza tra oggetti e database

La relazione tra l'applicazione e il database (di cui è responsabile il package *Database*) potrebbe costituire un problema delicato (Sezione 6.1.3), e ciò per due ragioni fondamentali. Primo, le strutture di memorizzazione del database potrebbero aver poco a che vedere con il paradigma object-oriented. Secondo, il database non è quasi mai progettato per una singola applicazione.

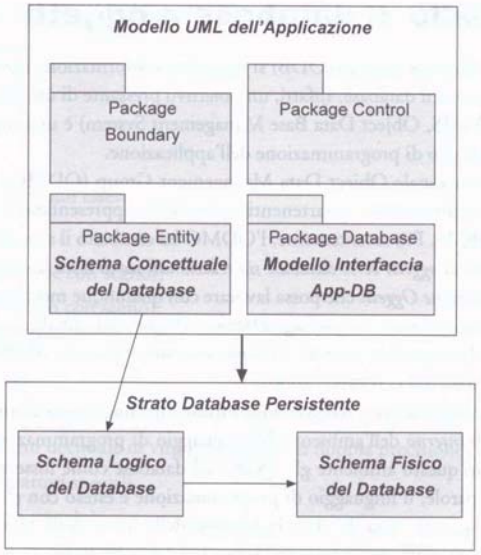


FIGURA 8.1 Modelli UML e persistenza.

La prima ragione riguarda la conversione delle classi del package *entity* (Sezioni 5.2.4 e 6.1.3.2) in strutture *non object-oriented* (generalmente, in tabelle relazionali). Anche se il database scelto per l'implementazione è un database a oggetti, gli aspetti specifici del database necessitano di un'attenta considerazione.

La seconda ragione richiede che il database sia *progettato per tutte le applicazioni* in modo ottimale, non per l'unica applicazione considerata. Le applicazioni dovrebbero aver associata una priorità in base alla loro "importanza" per l'azienda in esame: in questo modo, le strutture del database possono essere adattate alle applicazioni più importanti. Allo stesso tempo, il progettista del database dovrebbe "guardare al futuro", anticipando possibili richieste di dati necessari per le prossime applicazioni e cercando di tenerne conto nel progetto.

Si può ritenere che, attualmente, il database persistente sia un *database relazionale* poiché tale tecnologia domina il mercato. Per i grandi database aziendali, il passaggio alla tecnologia dei *database a oggetti* è una possibile evoluzione, attraversando lo stadio intermedio (se non definitivo) della tecnologia *object-relational*.

Nel seguito, sarà presentato dapprima uno scenario ideale (quantunque improbabile), dove la memorizzazione persistente è realizzata con un puro *database a oggetti*. Quindi, si discuterà la trasformazione dagli oggetti a un *database object-relational*. Infine, sarà considerato il modello *relazionale*, il più restrittivo e, di conseguenza, il più complesso su cui basare la trasformazione degli oggetti.

8.2 Modello di database a oggetti

Con un modello di *database a oggetti* (ODB) si realizza la trasformazione meno "traumatica" tra il programma applicativo e il database. Infatti, un obiettivo pressante di un *sistema di gestione di database a oggetti* (ODBMS, Object Data Base Management System) è un'integrazione agevole del database con il linguaggio di programmazione dell'applicazione.

Il comitato internazionale Object Data Management Group (ODMG) ha standardizzato il modello ODB. Le organizzazioni appartenenti all'ODMG rappresentano tutti i maggiori fornitori di software ODBMS. Più recentemente, l'ODMG ha cambiato il suo statuto per concentrarsi sulla *trasformazione di oggetti verso database sia relazionali sia di altro tipo*, rendendo lo standard un *API di Memorizzazione Oggetti* che possa lavorare con qualunque modello di dati. In effetti, lo standard può essere usato come un package *Database* (Figura 8.1) finalizzato alla trasformazione dall'applicazione al database (e viceversa). L'ultimo standard (Gennaio 2000) è noto come Object Data Standard: ODMG 3.0 (ODMG, 2000).

Lo standard specifica che un ODBMS non fornisce un linguaggio distinto (quale SQL) per manipolare i dati all'*interno* dell'ambiente del linguaggio di programmazione dell'applicazione. Invece, fa apparire in questo ambiente gli oggetti del database come fossero normali oggetti del linguaggio. In altre parole, il linguaggio di programmazione è esteso con gli oggetti del database che implementano la persistenza dei dati, la gestione delle transazioni, le interrogazioni navigazionali (cioè le interrogazioni che "navigano" lungo le relazioni), etc.

È compreso comunque un linguaggio distinto d'interrogazione affinché sia possibile aver accesso al database dall'*esterno* dell'ambiente del linguaggio di programmazione. Tale linguaggio è tipicamente chiamato Object SQL (OSQL). L'OSQL estende le capacità d'interrogazione di SQL relazionale con interrogazioni navigazionali e la capacità di elaborare tipi di dati più complessi, come, ad esempio, i tipi parametrici (Sezione 6.2.2.5).

8.2.1 Primitive di rappresentazione per ODB

Le primitive di rappresentazione di un modello di database a oggetti sono l'*oggetto* e il *letterale* (ODMG, 2000; Eaglestone e Ridley, 1998). Ciascun *oggetto* ha un OID (Sezione 2.1.1.3), mentre un *letterale* non ha un OID: il suo valore è come se fosse un identificatore.

Un ODB distingue tra *classe* (*implementazione*) e *tipo* (*specifica*). Un tipo può avere più classi. Ad esempio, il tipo Impiegato può essere implementato in una classe Smalltalk e/o in una classe Java. La semantica del tipo permette di separarne la specifica dalle varie implementazioni. La specifica del comportamento astratto di un tipo è un'*interfaccia* che non può essere direttamente istanziata.

Una classe ODB ha *proprietà* e *operazioni*. Una proprietà può essere un *attributo* o una *relazione* (cioè un attributo che lega un oggetto a uno o più altri oggetti).

8.2.1.1 Tipo letterale e tipo oggetto

Uno dei principali vantaggi di un ODB è il supporto predefinito per il tipo letterale e il tipo oggetto che rende un ODB una piattaforma d'implementazione naturale per lo sviluppo object-oriented di un sistema informativo. Se solo un ristretto numero di sviluppatori si converte ai database a oggetti, la causa è da ricercarsi sia in alcune debolezze di tali database (come un limitato supporto multi-utente) sia nel potere commerciale posseduto dai fornitori più influenti di database.

Un *tipo letterale* può essere (ODMG, 2000):

- atomico (semplice);
- strutturato;
- collezione;
- nullo.

Un tipo letterale *atomico* può essere:

- numerico
  - short (intero breve con segno);
  - long (intero lungo con segno);
  - unsigned short;
  - unsigned long;
  - float (numero decimale in virgola mobile e in singola precisione);
  - double (numero decimale in virgola mobile e in doppia precisione);
- alfanumerico e caratteri speciali
  - char (carattere singolo);
  - string (stringa di caratteri);
  - boolean (cioè i valori logici "vero" o "falso");
  - octet (stringa di bit per rappresentare dati "grezzi");
  - enum (lista enumerativa di valori permessi).

Un tipo letterale *strutturato* è una struttura dati predefinita che consiste di più d'un letterale atomico; esso può essere:

- date (ad esempio `9 Luglio 2000`);
- time (ad esempio `11:14`);
- timestamp (ad esempio `9 Luglio 2000 11:14:56`);
- interval (ad esempio `11:14, 11:19`).

Un tipo letterale *collezione* è un tipo parametrico (Sezione 6.2.2.5) dove i valori del parametro sono tipi letterali (ma possono anche essere tipi oggetto). Un tipo collezione può essere:

- `set<t>` (cioè un insieme dove tutti gli elementi sono dello stesso tipo letterale o tipo oggetto t; ad esempio `set<nome_dip>`, dove `nome_dip` è una stringa);
- `bag<t>` (è un multi-insieme, cioè un insieme che permette elementi replicati);
- `list<t>` (cioè un insieme ordinato);
- `array<t>` (cioè una collezione ordinata di elementi, di dimensioni dinamiche, dove ogni elemento può essere trovato attraverso la sua posizione);
- `dictionary<t,v>` (cioè una sequenza non ordinata, un indice, di coppie chiave-valore senza chiavi duplicate).

Un tipo letterale *nullo* può essere specificato per qualsiasi altro tipo letterale (ad esempio `string` o `list<>`). Il letterale nullo significa un valore nullo. Come nei database relazionali, il valore nullo rappresenta una delle due seguenti possibilità: "valore al momento sconosciuto" (ad esempio, una data di nascita non nota) oppure "valore non applicabile" (ad esempio, un uomo non può avere un cognome da nubile). Il valore nullo non è né zero né il carattere "spazio": è una particolare sequenza di bit adibita esclusivamente a questo scopo.

Un *tipo oggetto* può essere (ODMG, 2000):

- oggetto atomico;
- oggetto strutturato;
- collezione (lo stesso insieme di possibilità come per il letterale collezione ma il parametro è una classe; ad esempio, in `set <Studente>`, dove `Studente` è una classe, le istanze sono insiemi di studenti).

La Figura 8.2 è un esempio di dichiarazioni di tipo. La classe `Impiegato` ha sei proprietà. Una di queste proprietà (`nome_imp`) è un oggetto strutturato il cui valore è l'OID di un'istanza di `NomePersona`. La classe `NomePersona` non è mostrata ma consiste presumibilmente di attributi quali `cognome`, `nome`, `iniziale_secondo_nome`.

8.2.1.2 Relazioni e relazioni inverse

Il modello ODB si costruisce sulle primitive di rappresentazione discusse nelle sezioni precedenti. Dei tre tipi di relazione (cioè associazione, aggregazione, generalizzazione), il modello ODB offre un supporto diretto ad associazione e generalizzazione. L'*aggregazione* è semplicemente rappresentata come una restrizione dell'associazione.

Le *associazioni* sono implementate con tipi collezioni di oggetti, in particolare `set<>` e `list<>`. In pratica, le associazioni sono responsabili del ritorno all'accesso navigazionale nei database, in contrasto con l'accesso in stile relazionale basato su valori (si tratta di un ritorno poiché già negli storici database reticolari la navigazione costituiva il modo consueto di operare).

La Figura 8.3 mostra come l'associazione molti-a-molti tra le classi `Studente` e `CorsoOfferto` sia rappresentata in un ODB. Il modello grafico distingue tra attributi e relazioni, ma

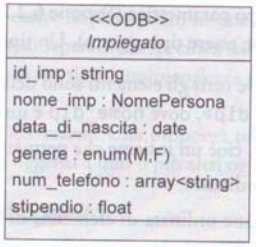
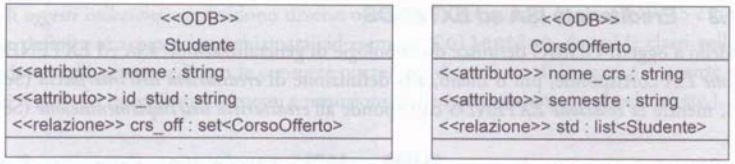


FIGURA 8.2 Dichiarazioni di tipo in ODB.




```
class Studente
{
    attribute      string              nome;
    attribute      string              id_stud;
    relationship   set<CorsoOfferto>   crs_off;
                   inverse CorsoOfferto::std;
};
class CorsoOfferto
{
    attribute      string              nome_crs;
    attribute      string              semestre;
    relationship   list<Studente>      std;
                   inverse Studente::crs_off;
};
```

FIGURA 8.3 Associazione in ODB.

nessuna linea d'associazione è tracciata tra le classi poiché le proprietà di relazione (`crs_off` e `std`) già implementano l'associazione stessa.

La parola chiave `inverse`, presente nella definizione dello schema, forza il vincolo di *integrità referenziale* sull'associazione ed elimina la possibilità di *puntatori non corretti* (dangling pointer), dichiarando esplicitamente una relazione come l'inversa di un'altra. Ad esempio, per aggiungere uno `Studente` a un `CorsoOfferto`, il programmatore può aggiungere l'oggetto di `CorsoOfferto` (il suo OID per essere precisi) all'attributo `crs_off` di tipo `set<CorsoOfferto>` oppure aggiungere l'oggetto di `Studente` a `std` di tipo `list<Studente>`. Una volta che il programmatore modifica un lato dell'associazione, il lato opposto (l'inverso) è automaticamente modificato dal sottostante ODBMS.

Il modello ODB, sebbene non mostrato nelle precedenti definizioni di classe, permette la specifica di *chiavi* cioè valori che identificano univocamente gli oggetti della classe. Diversamente dai database relazionali, la chiave non è né il solo né il principale metodo di identificazione degli oggetti (i valori OID hanno questo scopo). Tuttavia, a volte la ricerca in un ODB per mezzo di valori chiave può essere utile.

La chiave può essere *semplice* o, quando consiste di più di un attributo, *composta*. Una classe può avere diverse chiavi che possono essere distinte attraverso numeri in sequenza (o altre tecniche visuali).

8.2.1.3 Ereditarietà ISA ed EXTENDS

Il modello a oggetti ODMG definisce due tipologie di generalizzazione: ISA ed EXTENDS. La *relazione ISA* corrisponde, più o meno, alla definizione di *ereditarietà dell'interfaccia* (Sezione 5.3.3), mentre la *relazione EXTENDS* corrisponde all'*ereditarietà dell'implementazione* (Sezione 5.3.4).

La figura 8.4 mostra un diagramma UML esteso con relazioni ISA e EXTENDS. Secondo l'ultima versione dello standard, `Impiegato` eredita solo il comportamento da `INPersona` e non le sue proprietà, per cui una copia di `ddn` (data di nascita) deve essere esplicitamente inclusa in `Impiegato`. `Manager` eredita da `Impiegato` sia lo stato (dichiarazioni di proprietà) sia il comportamento (codice delle operazioni).

Lo standard ODMG permette l'*ereditarietà ISA multipla* delle interfacce (cioè l'ereditarietà multipla del comportamento). Sia le interfacce sia le classi possono ereditare da interfacce ma le interfacce non possono ereditare da classi. Inoltre, l'ereditarietà ISA non è permessa tra classi. Infine, lo standard permette solo l'*ereditarietà EXTENDS singola* dello stato e del comportamento.

8.2.1.4 Operazioni predefinite

Un ODBMS è dotato di *operazioni predefinite* a supporto dei tipi oggetto e letterale. Il progettista del sistema deve aver familiarità con le interfacce dello specifico ODBMS a disposizione per trarne vantaggio durante il progetto.

Le operazioni più interessanti sono quelle che trattano tipi strutturati e collezioni. Ad esempio, per il tipo *Date* operazioni predefinite permettono di verificare se una data cade in un certo intervallo temporale.

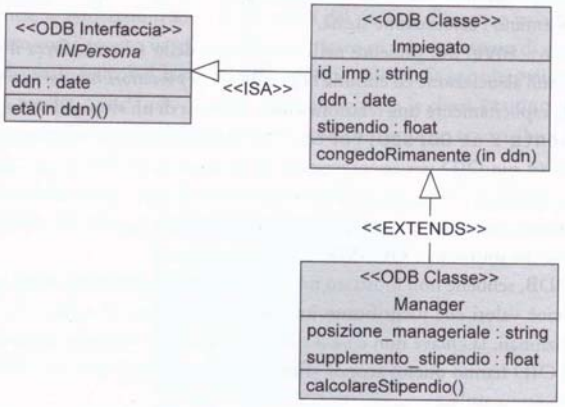


FIGURA 8.4 Ereditarietà in ODB.

Gli *oggetti collezione* condividono diverse operazioni. Per questo, in un ODBMS è normalmente definita una superclasse chiamata, ad esempio, `Collection`, da cui le classi collezione introdotte dall'utente ereditano le consuete operazioni di gestione di insiemi o sequenze di elementi (come, ad esempio, inserimenti e rimozioni di elementi, unione, intersezione, etc.).

8.2.2 Trasformazione UML - ODB

La trasformazione dei modelli UML verso un ODB è relativamente semplice: il compito di un ODB è proprio quello di fornire l'implementazione object-oriented di un modello di dati object-oriented. Il progetto di un ODB, infatti, si può rappresentare con diagrammi di classi UML, stereotipate in modo da poter permettere l'espressione di caratteristiche e vincoli propri del ODB stesso.

Il processo di trasformazione deve essere svolto per quelle parti dei modelli di stato e di comportamento connesse agli oggetti persistenti, cioè alle classi del package *entity*.

Il processo è centrato essenzialmente sulla trasformazione dello stato, mentre la trasformazione del comportamento è parte del progetto architetturale (Sezione 6.1), del progetto delle collaborazioni (Sezione 6.2) e del progetto client/server (Capitolo 9). In particolare, è responsabilità del progetto architetturale prendere la decisione iniziale su dove i vari processi dovrebbero essere eseguiti, se sul client o sul server. Questa decisione condiziona il progetto dettagliato delle collaborazioni e il progetto client/server.

8.2.2.1 Trasformazione delle classi entity

Un attento lettore avrà notato che, nelle rappresentazioni UML fin qui considerate, gli attributi delle classi sono stati definiti su *tipi dato atomici* e su pochi *tipi strutturati predefiniti* (`Date`, `Currency`).

Ma cosa può dirsi su semplici questioni come: "Cosa accade se un impiegato ha diversi numeri telefonici? Come si dovrebbe rappresentare questa situazione durante l'analisi? È veramente necessaria una classe distinta di numeri telefonici?" Oppure: "È possibile rappresentare il nome di un impiegato come un singolo attributo ma con una struttura interna (consistente di nome di famiglia, primo nome, e iniziale del secondo nome)? È realmente necessario avere una classe distinta per i nomi dell'impiegato?". Inoltre, i ruoli delle associazioni implicano una traduzione "naturale" in termini di *collezioni*.

Teoricamente, UML non pone limitazioni su estensioni al sistema di tipi, permettendo di definire nuove classi (nell'analisi) e usando tipi parametrici (nel progetto). In pratica, per evitare la proliferazione di classi minori, si evita di seguire questa strada fino a quando non sia noto il supporto fornito dalla piattaforma d'implementazione al sistema di tipi estensibili e ai tipi strutturati e collezione predefiniti. Il risultato è che questioni simili alle precedenti sono frequentemente accantonate fino al progetto di dettaglio.

I precedenti problemi sono ora analizzati e spiegati attraverso la trasformazione di alcuni dei modelli di analisi, già presentati nel testo, in schemi delle classi per un ODB.

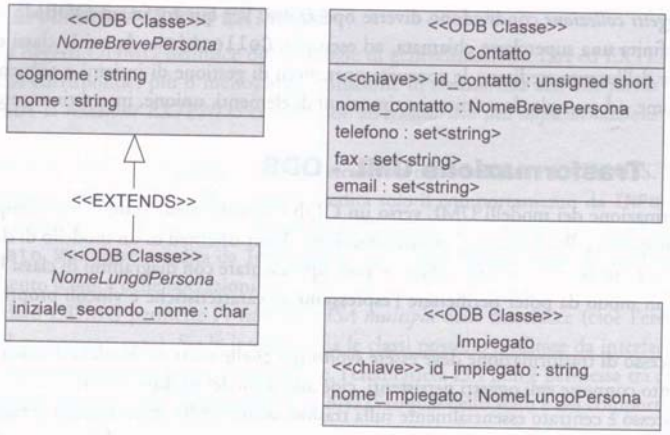


FIGURA 8.5 Trasformazione delle classi entity in schema ODB (CMS).

Esempio 8.1 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti, le specifiche delle classi nell'Esempio 4.6 (Figura 4.3 - Sezione 4.2.1.2.3) e le classi `Contatto` e `Impiegato` (si noti che le questioni precedentemente formulate si applicano a queste due classi).

`Contatto` ha gli attributi `cognome` e `nome` ma non possiede il concetto di nome_contatto. Analogamente, `Impiegato` contiene `cognome` e `nome` ma anche `sec_nome` e non è possibile interrogare il database su nome_impiegato poiché tale concetto non esiste.

`Contatto` ha anche gli attributi `telefono`, `fax`, `email`. L'attuale modello non permette che un contatto abbia più di un `telefono`, `fax` o `email`, sebbene questo sia poco realistico.

Il compito è trasformare le classi entity `Contatto` e `Impiegato` nel progetto per un ODB. La trasformazione dovrebbe analizzare e trattare i problemi precedentemente identificati.

La Figura 8.5 mostra la trasformazione. Sono state introdotte due classi astratte ODB per rappresentare i tipi `NomeBrevePersona` e `NomeLungoPersona`. La prima classe definisce il tipo di dato per `nome_contatto` nella classe `Contatto`. La seconda classe, `NomeLungoPersona`, eredita i due attributi da `NomeBrevePersona` attraverso la relazione EXTENDS e, insieme al proprio attributo (`iniziale_secondo_nome`, contenente l'iniziale del secondo nome), fornisce il tipo di dato per `nome_impiegato` nella classe `Impiegato`.

La possibilità per un oggetto di `Contatto` di avere diversi telefoni, fax e email è consentita con il tipo parametrico `set`, uno dei tipi collezione supportati dai sistemi per ODB.

8.2.2.2 Trasformazione delle associazioni

Nei modelli UML le associazioni tra classi permettono la navigazione tra gli oggetti delle classi stesse, esattamente una delle caratteristiche che contraddistinguono i database a oggetti.

La trasformazione delle associazioni verso un ODB è, di conseguenza, particolarmente semplice, come già mostrato per l'associazione tra `Studente` e `CorsoOfferto` nella Sezione 8.2.1.2 (Figura 8.3). Una proprietà di relazione in una classe del ODB è tipata con il nome della classe (o una collezione sulla classe) con cui esiste la relazione.

Ciò detto, si potrebbe ottimizzare il progetto durante la trasformazione rappresentando alcuni attributi UML (o classi UML) come interfacce del ODB, usando poi tali interfacce come tipi per le proprietà nelle classi del ODB (si veda Figura 8.5).

Esempio 8.2 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti, le specifiche delle associazioni nell'Esempio 4.8 (Figura 4.5 - Sezione 4.2.2.3) e il precedente esempio di trasformazione delle classi entity (Figura 8.5).

Il compito è trasformare il modello di Figura 4.5 in un progetto per un ODB. Durante la trasformazione è necessario valutare quali attributi o classi UML siano buoni candidati per diventare interfacce del ODB da usare (riusare) nelle classi del ODB stesso.

La Figura 8.6 è una delle possibili soluzioni. Oltre alle precedenti *classi ODB* (`NomeBrevePersona` e `NomeLungoPersona`), è stata introdotta una *gerarchia d'ereditarietà* EXTENDS con radice la classe ODB `Indirizzo`, le cui sottoclassi sono utilizzate nelle classi `Organizzazione` e `Contatto` del ODB per gli attributi `indirizzo_postale` e `indirizzo_corriere`. Nella figura, non è tracciato alcun legame d'associazione poiché le relazioni sono rappresentate come proprietà delle classi.

8.2.2.3 Trasformazione delle aggregazioni

Come spiegato in precedenza (Sezioni 2.1.4, 4.2.3 e 5.4), UML riconosce due sole semantiche dell'aggregazione: l'*aggregazione* con la semantica per referenza e la *composizione* con la semantica per valore. Pur limitata, la nozione di aggregazione UML probabilmente non è direttamente supportata dalle più comuni implementazioni di ODB (e neppure da qualunque altra tipologia di database).

Nei database, le aggregazioni sono rappresentate come associazioni oppure come attributi annidati. Se è necessario verificare che la semantica dell'aggregazione sia rispettata rigorosamente, si deve ricorrere a procedure opportune (nei programmi applicativi).

Il principio di trasformazione delle aggregazioni è semplice. Un'*aggregazione UML* è trattata come se fosse un'associazione. Una *composizione UML* diviene una classe composta contenente un attributo annidato che rappresenta la classe componente. Poiché la classe componente ha una struttura interna (cioè non è atomica), deve essere dapprima definita un'interfaccia ODB come un tipo oggetto strutturato. L'attributo annidato prende i valori di quel tipo oggetto. Va detto, infine, che sono possibili alcune variazioni alle precedenti strategie di trasformazione ai fini di maggiore efficienza d'elaborazione, riuso, scalabilità, manutenibilità, etc.

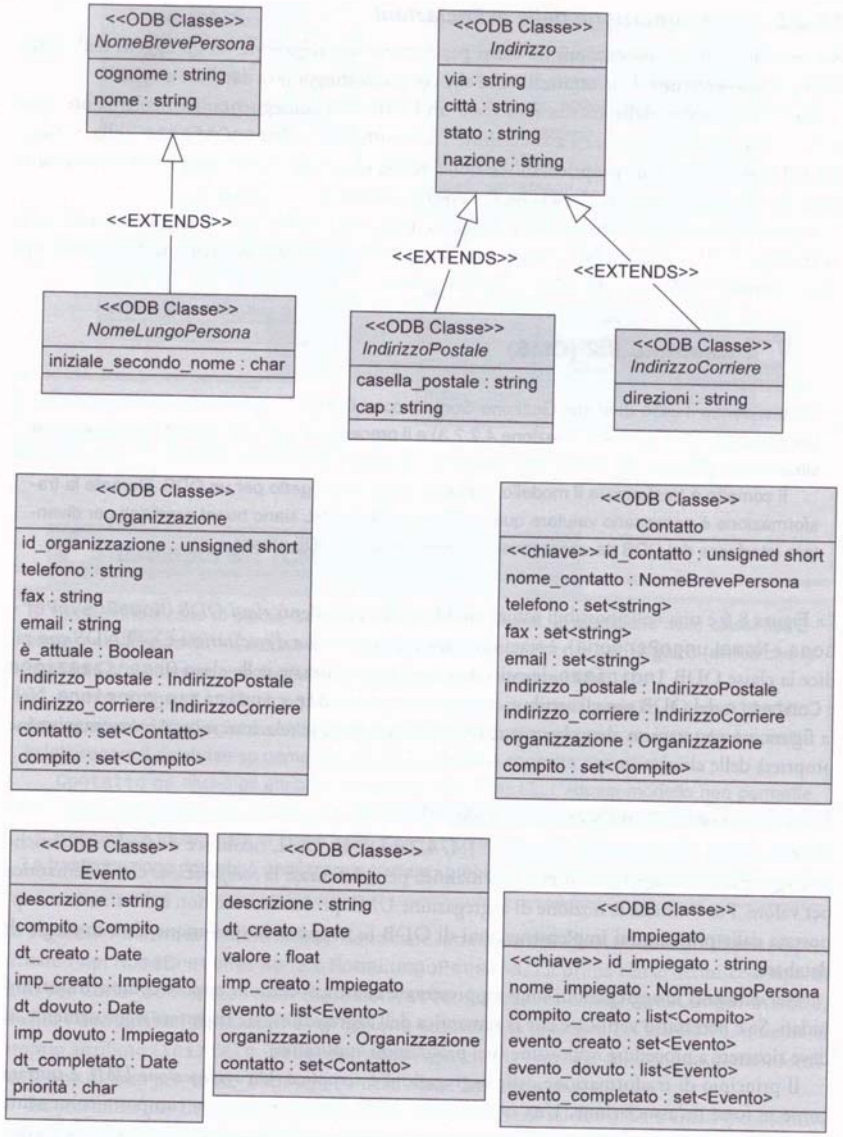


FIGURA 8.6 Trasformazione delle associazioni in schema ODB (CMS).

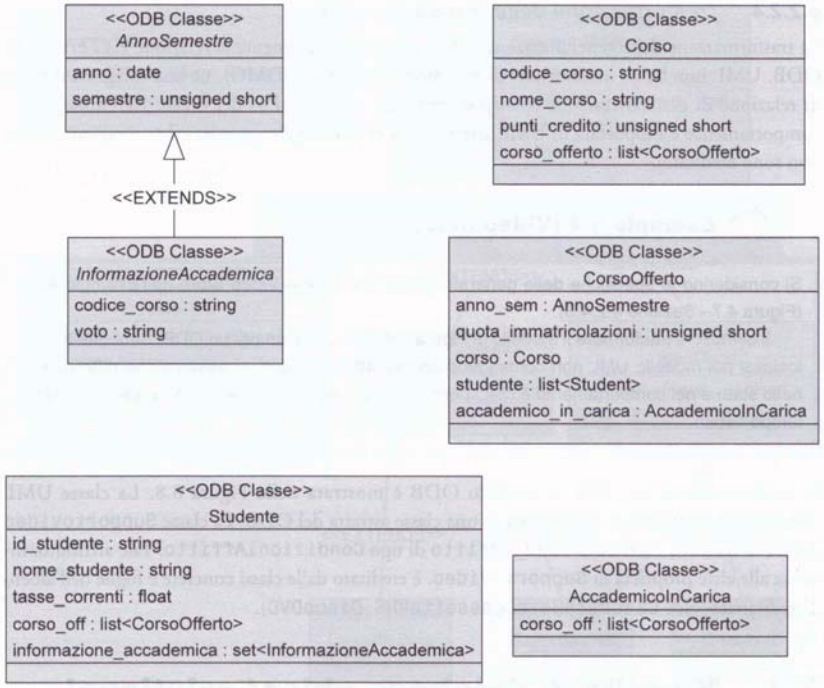


FIGURA 8.7 Trasformazione delle aggregazioni in schema ODB (Immatricolazioni universitarie).

Esempio 8.3 (Immatricolazioni universitarie)

Si considerino le specifiche d'aggregazione per le Immatricolazioni universitarie nell'Esempio 4.9 (Figura 4.6 - Sezione 4.2.3.3).

Il compito è trasformare il modello UML di Figura 4.6 in un progetto per ODB. Per trattare gli attributi annidati e per semplificare il modello, alcune interfacce verso il ODB potrebbero dover essere definite e usate come tipi per gli attributi delle classi.

La Figura 8.7 fornisce una possibile soluzione. Sono state definite due classi astratte nel ODB: `AnnoSemestre` e `InformazioneAccademica`. La prima non è particolarmente importante e il modello potrebbe farne a meno; la seconda è necessaria per definire la composizione UML attraverso l'attributo `informazione_accademica` annidato nella classe `Studente`.

L'aggregazione UML tra `Corso` e `CorsoOfferto` è rappresentata come una normale associazione. La semantica dell'aggregazione dovrebbe essere verificata proceduralmente. Le proprietà di relazione definiscono le relazioni di aggregazione e associazione presenti nella Figura 4.6.

8.2.2.4 Trasformazione delle generalizzazioni

La trasformazione delle generalizzazioni UML sfrutta essenzialmente la relazione *EXTENDS* del ODB. UML non ha né una nozione di *interfaccia* (nel senso ODMG), né una categoria distinta di relazione di generalizzazione corrispondente alla relazione *ISA* del ODB. L'ereditarietà del comportamento è supportata in UML attraverso la visibilità delle proprietà (le operazioni private non sono ereditate).

Esempio 8.4 (Video Store)

Si considerino le specifiche delle generalizzazioni per il caso Video Store nell'Esempio 4.10 (Figura 4.7 - Sezione 4.2.4.3).

Il compito è trasformare il modello di Figura 4.7 in un progetto per un ODB. Sebbene le sottoclassi nel modello UML non contengano ancora attributi propri, si assuma che differiscano nello stato e nel comportamento e che specifiche proprietà saranno aggiunte a queste classi a tempo debito.

La trasformazione da UML al modello ODB è mostrata nella Figura 8.8. La classe UML `CondizioniAffitto` è trasformata in una classe astratta del ODB. La classe `SupportoVideo` del ODB contiene l'attributo `cond_affitto` di tipo `CondizioniAffitto`. Tale attributo, insieme alle altre proprietà di `SupportoVideo`, è ereditato dalle classi concrete e foglie dell'albero d'ereditarietà (cioè `CassettaBeta`, `CassettaVHS`, `DiscoDVD`).

8.3 Modello di database object-relational

La prossima "grande ondata" nella tecnologia dei database sarà il modello object-relational (Stonebraker e Moore, 1996). Come dice il nome, un database object-relational (ORDB) combina il modello relazionale con il modello a oggetti. Un solo sistema di gestione di database object-relational (ORDBMS) è in grado di elaborare le strutture dati relazionali (*tabelle relazionali*) e le strutture dati a oggetti (*tabelle di oggetti*).

Lo standard per il modello ORDB è stato ratificato nel 1999 con il nome SQL:1999 (Eisenberg e Melton, 1999) (noto in precedenza con il nome informale di SQL3), dopo più di sei anni di lavoro. Lo standard, sviluppato a cura di American National Standards Institute (ANSI) e di International Organization for Standardization (ISO), lascia comunque non risolti molti problemi legati al modello ORDB e si ipotizza che sarà rivisto, più o meno, ogni tre anni.

Il modello ORDB è compatibile con SQL92, l'ultimo standard per database relazionali. Il modello estende il meccanismo convenzionale della tabella relazionale con un nuovo meccanismo per memorizzare oggetti nelle tabelle SQL. Estende anche il limitato supporto relazionale per i tipi definiti dall'utente con *tipi strutturati* arbitrariamente complessi (per incapsulare attributi e operazioni).

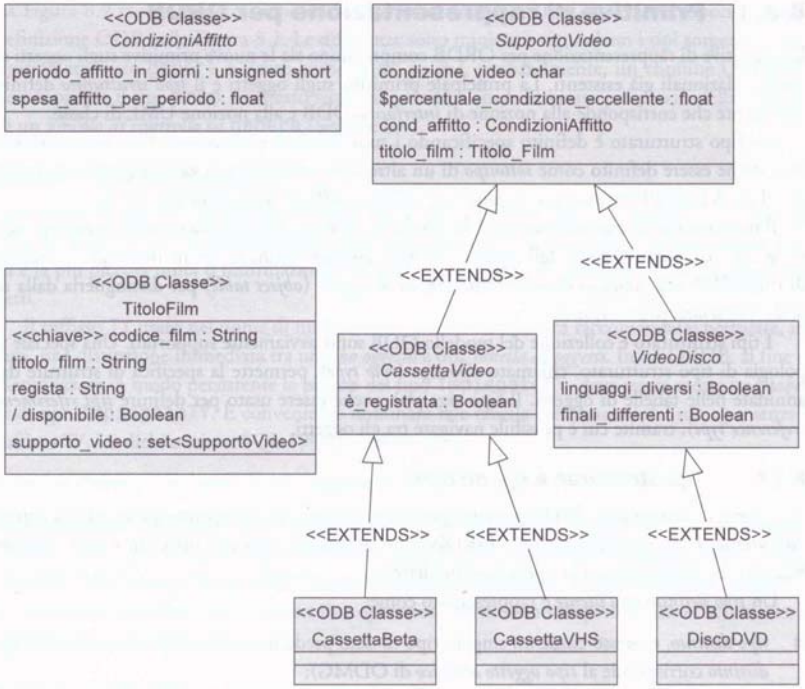


FIGURA 8.8 Trasformazione delle generalizzazioni in schema ODB (Video Store).

Mentre lo standard continua a evolvere, i principali fornitori di database relazionali (Oracle, IBM, Informix) hanno avuto come obiettivo la fornitura di prodotti ORDBMS comprendenti almeno un supporto parziale del modello ORDB. In questo senso, uno dei maggiori problemi per i fornitori di ORDBMS è stato (e ancora è) come integrare le preesistenti caratteristiche relazionali con le nuove caratteristiche relative agli oggetti al fine di permettere una graduale migrazione dai sistemi relazionali a soluzioni ORDB. Questo problema non è sostanzialmente affrontato dallo standard SQL:1999; comunque, nel seguito della sezione si farà riferimento alle specifiche SQL:1999 (saranno possibili occasionali eccezioni che fanno riferimento a prodotti ORDBMS reali, in particolare Oracle8, allo scopo di evidenziare le differenze tra teoria e pratica).

8.3.1 Primitive di rappresentazione per ORDB

Le primitive di rappresentazione per ORDB comprendono sia le nuove primitive sugli oggetti sia quelle relazionali già esistenti. La principale primitiva sugli oggetti è il *tipo strutturato* definito dall'utente che corrisponde alla nozione di *interfaccia* ODB e alla nozione UML di classe.

Un tipo strutturato è definito specificando i suoi *attributi* e *operazioni*. Un tipo strutturato può anche essere definito come *sottotipo* di un altro tipo strutturato. A tale proposito, va notato che il modello ORDB supporta l'*ereditarietà multipla dell'interfaccia* (Sezione 5.3.3).

Il meccanismo di memorizzazione è la *tabella*. Le *colonne* della tabella possono assumere valori dei tipi strutturati definiti dall'utente. Una tale tabella è chiamata, in alcune implementazioni di ORDBMS (ad esempio Oracle8) una *tabella di oggetti* (*object table*) per distinguerla dalla *tabella relazionale* convenzionale.

I tipi strutturato e collezione del modello ODB sono ovviamente supportati. Una speciale tipologia di tipo strutturato, chiamato *tipo riga* (*row type*), permette la specifica di strutture dati annidate nelle tabelle di oggetti. Il tipo riga può anche essere usato per definire *tipi riferimento* (*reference type*), tramite cui è possibile navigare tra gli oggetti.

8.3.1.1 Tipi strutturati e tipi distinti

Le *colonne* di una tabella ORDB possono prendere valori di *tipi predefiniti* oppure di tipi *definiti dall'utente*. I *tipi predefiniti* di un ORDB sono analoghi a ciò che offre un ODB (Sezione 8.2.1.1). Lo stesso vale per le operazioni predefinite.

Un *tipo definito dall'utente* è rappresentato come:

- *tipo distinto*, espresso come un singolo tipo di dato predefinito chiamato *tipo sorgente* (il *tipo distinto* corrisponde al *tipo oggetto atomico* di ODMG);
- *tipo strutturato*, espresso come una lista di definizioni d'attributo e d'operazione (il *tipo strutturato* corrisponde al *tipo oggetto strutturato* di ODMG).

Un *tipo strutturato* permette agli utenti di definire i propri nomi di tipo. Una definizione di tipo strutturato consiste delle seguenti dichiarazioni:

- *attributi* che rappresentano lo stato degli oggetti del tipo strutturato;
- *operazioni* che definiscono il comportamento degli oggetti del tipo strutturato;
- operazioni che definiscono l'*uguaglianza / disuguaglianza* degli oggetti del tipo strutturato (necessarie per confrontare due oggetti del tipo strutturato, oppure per *ordinarli* in qualche modo, oppure per *convertire* il tipo strutturato di un oggetto in un altro tipo).

Un *attributo* del tipo strutturato può essere tipato con un *tipo sorgente*, un *tipo distinto*, un *tipo collezione* o un altro tipo strutturato. SQL:1999 definisce svariati tipi collezione (i prodotti ORDB possono definire ulteriori tipi collezione):

- set;
- list;
- multiset (lo stesso del tipo bag per le ODB);
- array.

La Figura 8.9 mostra la definizione di un tipo strutturato per `Impiegato`, corrispondente alla definizione ODB nella Figura 8.2. Le differenze sono minime e riguardano i tipi sorgente predefiniti. L'attributo `genere` è tipato come `char` poiché, probabilmente, un comune ORDBMS non supporta il tipo `enum`. Lo stesso risultato potrebbe essere ottenuto per mezzo della specifica di un *vincolo di controllo* su `genere` (Sezione 8.4).

8.3.1.2 Tabelle di oggetti

Una *tabella di oggetti* è un insieme di righe in cui ogni *riga* è un oggetto (istanza) di un *tipo riga* (si veda Sezione 8.3.1.3) univocamente identificato per mezzo di un OID. Coerentemente, la riga è la più piccola unità d'informazione che può essere inserita o cancellata in una tabella di oggetti.

Il suffisso `TY` usato nel nome di un tipo strutturato è una pratica raccomandata: permette, infatti, una distinzione immediata tra un *tipo oggetto* e una *tabella di oggetti*. In un ORDB, al fine di memorizzare in modo persistente le istanze del tipo `ImpiegatoTY`, deve essere creata una *tabella* di tipo `ImpiegatoTY`. È conveniente nominare tale tabella con il nome del suo tipo senza il suffisso `TY`. La dichiarazione SQL:1999 potrebbe essere:

```
create table Impiegato of ImpiegatoTY;
```

L'*incapsulamento* di attributi di un tipo strutturato per mezzo di operazioni (si veda, ad esempio, Sezioni 2.1.2.1.2 e 5.1.4) non è stato definito in SQL:1999 (Eisenberg e Melton, 1999). Tuttavia, SQL:1999 assume che un ORDB generi, per ciascun attributo, un'operazione *osserva* (`get`) e un'operazione *modifica* (`set`). Queste due operazioni permettono, rispettivamente, la lettura e la modifica di un attributo.

8.3.1.3 Tipi riga

Un *tipo riga* permette di costruire una tabella con una struttura interna relativamente complessa senza ulteriore uso di tipi strutturati o collezioni. Un tipo riga è una sequenza di *campi* (coppie <nome campo> <tipo dato>). In effetti, un tipo riga permette di avere una tabella all'interno di un'altra tabella poiché una colonna in una tabella può contenere valori riga.

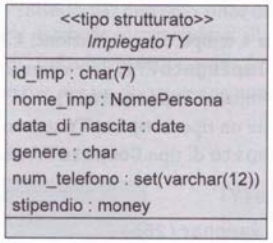


FIGURA 8.9 Dichiarazioni di tipo in ORDB.

L'esempio che segue spiega come i tipi riga possono essere usati per definire una tabella con una struttura interna complessa. L'esempio s'ispira ad alcune classi del caso CMS (si veda Figura 8.6).

```
create table Contatto
    (id_contatto            integer,
     nome_contatto          row
        (cognome            varchar(30),
         nome               varchar(20)),
    indirizzo_postale       row
        (casella_postale    varchar(10),
         codice_postale     varchar(10),
    indirizzo               row
        (via                varchar(30),
         città              varchar(20),
         stato              varchar(20),
         nazione            varchar(25))));
```

Dal punto di vista della programmazione del database, i tipi riga permettono la memorizzazione di intere righe in variabili, passando poi tali variabili come argomenti d'ingresso e/o d'uscita alle operazioni.

8.3.1.4 Tipi riferimento

Un tipo strutturato può essere usato per definire un *tipo riferimento*. La parola chiave `ref` è usata per definire riferimenti. Ad esempio `imp ref(ImpiegatoTY)` è un riferimento, contenuto in una tabella di oggetti, a un tipo strutturato. Un valore della colonna `imp` è un OID che identifica un oggetto (una riga) in una tabella di tipo `ImpiegatoTY`. Si dice che il valore è un riferimento a una riga in una *tabella referenziabile* (*referenceable table*). Una tabella referenziabile deve essere una *tabella tipata* (*typed table*) cioè una tabella con associato un tipo strutturato.

In SQL:1999 i tipi riferimento sono *risolvibili staticamente (statically scoped)*: ciò significa che la tabella cui si riferiscono è nota a tempo di compilazione. Ciò significa anche che il valore di `imp`, con tipo riferimento `ref(ImpiegatoTY)`, deve riferirsi a un'unica tabella (invece di una riga in una qualunque tabella definita su `ImpiegatoTY`).

Ad esempio, è possibile definire un tipo `CompitoTY` con un campo imp di tipo riferimento a `ImpiegatoTY` e una tabella `Compito` di tipo `CompitoTY`, nel modo seguente:

```
create type CompitoTY
    (descrizione varchar(255),
     imp ref(ImpiegatoTY) scope Impiegato);
create table Compito of CompitoTY;
```


Una soluzione SQL:1999 alternativa potrebbe omettere la dichiarazione `scope Impiegato` nella definizione di tipo ed estendere la definizione di `create table` nel modo seguente:

```
create table Compito of CompitoTY
    (imp with options scope Impiegato);
```

In pratica, restringendo, in SQL:1999, il tipo riferimento a una singola tabella, si rendono i riferimenti molti simili alle chiavi esterne nei RDB, eccezion fatta per i vantaggi derivanti dall'uso di OID come valori per i riferimenti. Infatti, questo rende i riferimenti univoci all'interno del database, non modificabili, ma riusabili da altri oggetti (righe), e migliora le prestazioni nella navigazione tra oggetti.

I tipi riferimento possono essere usati in un ORDB per implementare le *associazioni uno-a-uno*. Per implementare le *associazioni molti-a-molti*, potrebbero essere usate le *collezioni* (Sezione 8.3.1.1) *di riferimenti*, se disponibili naturalmente. La Figura 8.10 sfrutta l'esempio presentato nella Figura 8.3 per mostrare come l'associazione molti-a-molti tra `Studente` e `CorsoOfferto` può essere rappresentata in un ORDB. In questo e nei seguenti esempi si usa, per definire i riferimenti, una forma breve con la tabella al posto del tipo (ad esempio, `imp_dovuto: ref(Impiegato)`).

8.3.1.5 Colonne, campi e attributi

SQL:1999 introduce un'attenta distinzione tra le nozioni di colonna, campo e attributo (SQL, 2000). Ciò costituisce un importante chiarimento terminologico:

- *colonna* è una componente strutturale di una *tabella*;
- *campo* è una componente strutturale di un *tipo riga*;
- *attributo* è una componente strutturale di un *tipo strutturato*.

Una colonna può essere *a valori nulli*, in altre parole può prendere NULL come valore. Una colonna può essere anche una colonna *identità* (prende OID come valori). Un tipo di dati per una colonna, campo o attributo può essere un *tipo riferimento*.

8.3.1.6 Ereditarietà OF e UNDER

SQL:1999 permette la specializzazione di tipi esistenti. Attualmente, è permessa solo l'*ereditarietà singola*. È possibile creare gerarchie di tabelle corrispondenti alle gerarchie di tipi: una "supertabella" deve essere dichiarata come "di" (`of`) un supertipo e una "sottotabella" come "di" (`of`) un sottotipo. La Figura 8.11 mostra tuttavia che un tipo può non essere indicato nella gerarchia delle tabelle.

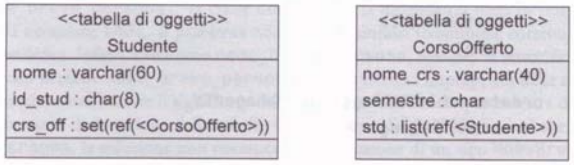
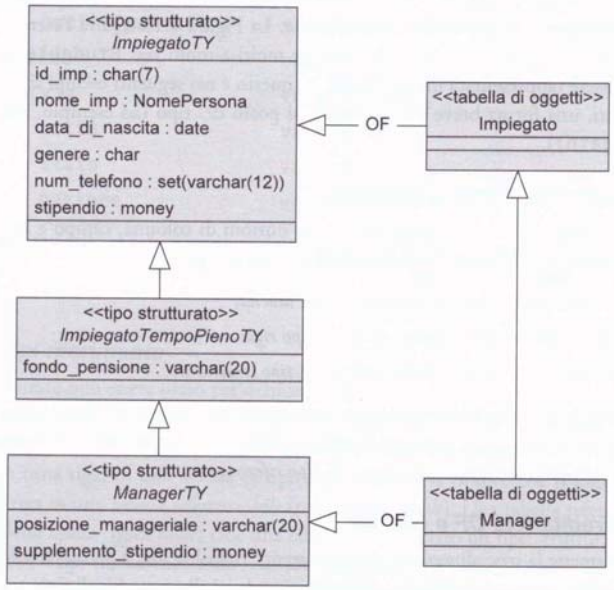


FIGURA 8.10 Associazione in ORDB.

Il codice SQL:1999, sottostante il diagramma di Figura 8.11, si riferisce al tipo più specializzato (`ManagerTY`) e alla tabella (`Manager`). La parola chiave `under` è usata in SQL:1999 per definire sia una gerarchia di tipi sia una gerarchia di tabelle. La parola chiave `of` permette di definire il tipo strutturato di una tabella. Nel diagramma `OF` è usata anche come nome delle relazioni di generalizzazione tra le tabelle e i tipi corrispondenti.

Il tipo `ManagerTY` è stato dichiarato essere `instantiable`, essendo possibile creare oggetti di quel tipo. Lo stesso tipo è stato dichiarato come tipo specializzato `final`, non essendo permessi ulteriori suoi sottotipi.

Si noti che SQL:1999 non ha alcuna esplicita nozione di *interfaccia*. L'*incapsulamento* (privato, protetto, pubblico) non è definito, e l'ereditarietà adottata in SQL:1999 è l'*ereditarietà singola dell'implementazione*.




```
create type ManagerTY
    under ImpiegatoTempoPienoTY
    as (posizione_manageriale varchar(20),
        supplemento_stipendio money)
    instantiable
    final ;

create table Manager of ManagerTY
    under Impiegato;
```

FIGURA 8.11 Ereditarietà in ORDB.

8.3.2 Trasformazione UML - ORDB

Come nel caso del modello ODB, la trasformazione delle classi UML contenute nel package entity in un progetto per ORDB può essere definita usando nuovamente UML. Stereotipi e altri meccanismi di estensione UML dovrebbero essere sufficienti per esprimere i concetti di un ORDB.

Si noti, tuttavia, che in pratica la trasformazione non è fatta sulla base dello standard SQL:1999 "astratto" ma su un reale prodotto ORDB. Il prodotto reale, infatti, potrebbe non supportare alcune caratteristiche di SQL:1999 mentre, a sua volta, potrebbe possedere caratteristiche non menzionate in SQL:1999.

Ad esempio, l'attuale versione di Oracle8 (Oracle8i Versione 8.1.5.0.0) non supporta né l'ereditarietà né il tipo riga. Possiede la collezione vettore (array collection) ma nessun'altra tipologia di collezione. Possiede la nozione di tabella annidata (e, conseguentemente, di tabella annidata di riferimenti) di vista oggetti (per "trasformare" le tabelle relazionali in oggetti) etc.

8.3.2.1 Trasformazione delle classi entity

Esempio 8.5 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti, le specifiche delle classi nell'Esempio 4.6 (Figura 4.3 - Sezione 4.2.1.2.3) e anche l'Esempio 8.1 (Sezione 8.2.2.1).

Il compito in quest'esempio è sviluppare un modello di progetto per ORDB semanticamente corrispondente al progetto per ODB nella Figura 8.5.

Una soluzione a quest'esempio richiede una scelta su come rappresentare le classi astratte `NomeBrevePersona`, `NomeLungoPersona` (Figura 8.5) nel modello di progetto per il ORDB. Le classi astratte non sono definite in SQL:1999 ma è ancora necessario rappresentare in qualche modo `NomeBrevePersona` e `NomeLungoPersona`.

La Figura 8.12 dimostra due possibili soluzioni. Nel caso di `NomeLungoPersona`, è stato creato un tipo strutturato `NomeLungoPersonaTY`. Quest'ultimo può essere usato come tipo per un attributo in `ImpiegatoTY`. Nel caso di `NomeBrevePersona`, è stata creata una classe[1] (`nome_breve_persona`) con lo scopo di "simulare" il tipo riga (il tipo riga non è un concetto object-oriented!). La tabella `Contatto` è di tipo `ContattoTY` ma ha una colonna aggiuntiva `nome_breve_persona` del tipo riga. È usata una relazione di dipendenza alla classe `nome_breve_persona` al fine di mostrare la struttura del tipo riga.

1. La classe `nome_breve_persona`, la classe `Contatto` e la dipendenza sono in realtà una forzatura semantica della notazione UML. Il problema non sta nella sintassi (comunque corretta) ma nel significato della dipendenza. Infatti, l'attributo `nome_breve_persona`, secondo la semantica UML, non ha alcun legame con la classe `nome_breve_persona`. Da ciò deriva l'improprietà della dipendenza tracciata. Nell'intenzione dell'Autore il significato del diagramma UML nella parte destra della figura è fornito dalla sottostante dicharazione della tabella `Contatto` in SQL:1999. Si noti che, a differenza di `NomeLungoPersona`, la soluzione non necessita la dichiarazione di un tipo `NomeBrevePersonaTY`. *(N.d.T.)*

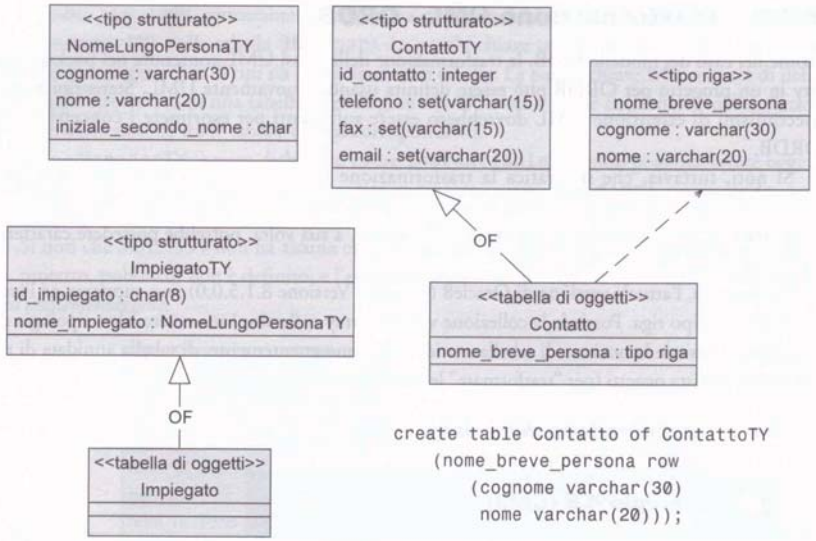


FIGURA 8.12 Trasformazione delle associazioni in schema ORDB (CMS).

8.3.2.2 *Trasformazione delle associazioni*

Esempio 8.6 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti, le specifiche delle associazioni nell'Esempio 4.8 (Figura 4.5 - Sezione 4.2.2.3) e i precedenti esempi di trasformazione (Sezioni 8.2.2.2. e 8.3.2.1).

Il compito in quest'esempio è sviluppare un modello di progetto per ORDB semanticamente corrispondente al progetto di ODB nella Figura 8.6 (Sezione 8.2.2.2). Per semplificare la soluzione e concentrarsi sulle associazioni, si assuma che tutte le interfacce ODB nella Figura 8.6 siano convertite in tipi riga nel progetto di ORDB (non è necessario mostrare le definizioni dei tipi riga nella soluzione).

La Figura 8.13 presenta una soluzione dell'esempio. Le tabelle di oggetti contengono colonne tipate con tipi riga e colonne tipate con tipi riferimento. I tipi strutturati contengono attributi tipati sia con tipi sorgente atomici sia con collezioni di tipi sorgente.

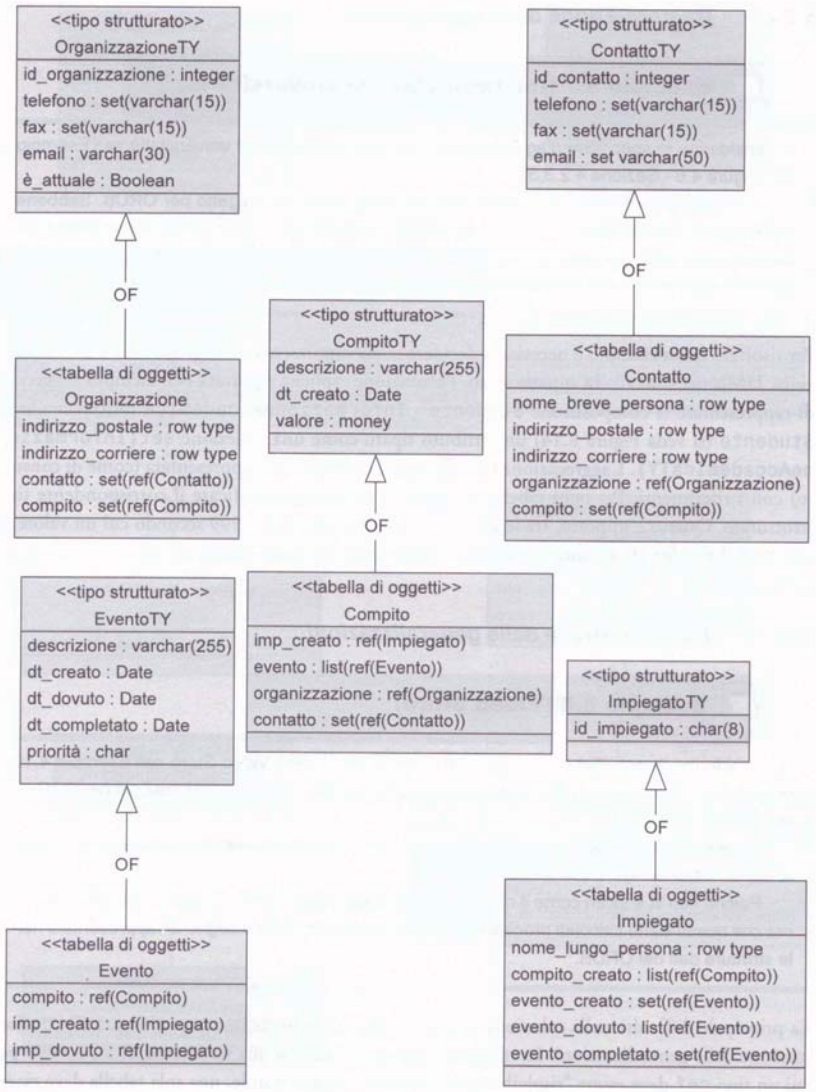


FIGURA 8.13 Trasformazione delle associazioni in schema ORDB (CMS).

8.3.2.3 Trasformazione delle aggregazioni

Esempio 8.7 (Immatricolazioni universitarie)

Si considerino le specifiche d'aggregazione per le Immatricolazioni universitarie nell'Esempio 4.9 (Figura 4.6 - Sezione 4.2.3.3).

Il compito è trasformare il modello UML di Figura 4.6 in un progetto per ORDB. Sebbene nell'Esempio 8.3 sia stato sviluppato un analogo progetto per ODB, non si vuole essere influenzati dalla precedente soluzione. Si assuma che sia possibile usare collezioni di tipi strutturati (non solo collezioni di tipi sorgente primitivi) come tipi di attributi nei tipi strutturati.

Per risolvere quest'esempio è necessario decidere come rappresentare l'aggregazione e la composizione UML nel ORDB. In questo senso, l'assunzione esplicita riportata nell'esempio suggerisce di rappresentare la composizione `Studente - InformazioneAccademica` introducendo in `Studente` (si veda Figura 8.14) un attributo tipato come una collezione `set(InformazioneAccademicaTY)`. L'aggregazione `Corso - CorsoOfferto` è rappresentata (come di consueto) con i riferimenti. Per ogni tabella di oggetti è necessario specificare il corrispondente tipo strutturato. Questo è imposto, tra le altre cose, dal requisito SQL:1999 secondo cui un valore di tipo `ref` deve identificare una riga in una tabella tipata (cioè una tabella con un tipo strutturato specificato).

8.3.2.4 Trasformazione delle generalizzazioni

Esempio 8.8 (Video Store)

Si considerino le specifiche delle generalizzazioni per il caso Video Store nell'Esempio 4.10 (Figura 4.7 - Sezione 4.2.4.3) e l'esempio di trasformazione delle generalizzazioni per ODB visto in Sezione 8.2.2.4.

Il compito è di sviluppare un progetto per un ORDB semanticamente equivalente al progetto per un ODB di Figura 8.8 (Sezione 8.2.2.4).

Poiché non si è sicuri come il modello ORDB supporta gli attributi statici e derivati, si assuma che questi siano calcolati proceduralmente e quindi non vi sia bisogno di rappresentarli nelle strutture dati del ORDB.

La principale difficoltà nella soluzione di quest'esempio è solo indirettamente legata alla trasformazione delle generalizzazioni. La maggiore sfida è in relazione alla restrizione di SQL:1999 per cui un tipo `ref` deve essere "risolvibile staticamente": in altre parole, una sola tabella deve essere associata staticamente (a tempo di compilazione) al tipo `ref`.

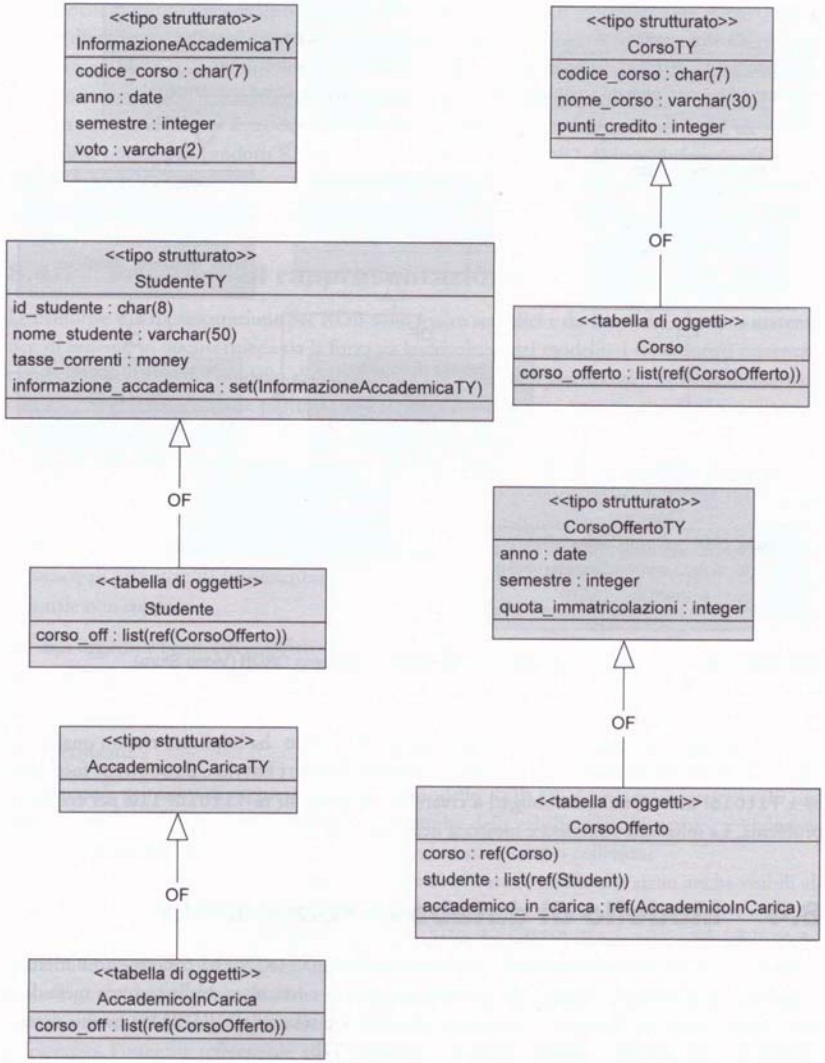


FIGURA 8.14 Trasformazione delle aggregazioni in schema ORDB (Immatricolazioni universitarie).

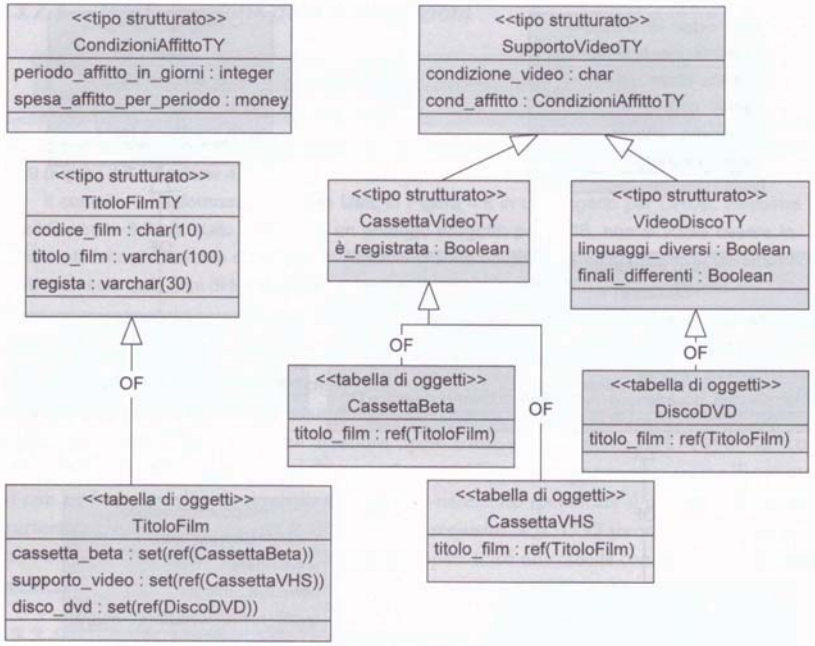


FIGURA 8.15 Trasformazione delle generalizzazioni in schema ORDB (Video Store).

In precedenza, associare `TitoloFilm` e `SupportoVideo` ha implicato che un qualunque oggetto di `SupportoVideo` (cioè di `CassettaBeta`, `CassettaVHS` e `DiscoDVD`) fosse legato a `TitoloFilm`. Ora, si è obbligati a creare tre associazioni in `TitoloFilm` per trattare il problema. La soluzione completa è mostrata nella Figura 8.15.

8.4 Modello di database relazionale

Durante gli ultimi 20 anni, il modello per database relazionali (RDB) ha conquistato il mercato del software per database, sostituendo i modelli gerarchico e reticolare. Nella seconda metà degli anni 1990, i fornitori di sistemi di gestione per database relazionali (RDBMS) hanno dovuto confrontarsi con prodotti ODBMS basati sul modello ODB.

La conseguenza è stata la nascita di prodotti ORDBMS, destinati in futuro a giocare un ruolo dominante. I fornitori di RDBMS tradizionali, quali Oracle, IBM o Informix, offrono oggi i più affermati prodotti ORDBMS. Nel frattempo, i prodotti ODBMS non hanno incrementato la propria quota di mercato, e, quindi, hanno modificato il loro obiettivo diventando interfacce applicative per la memorizzazione d'oggetti a supporto dell'interoperabilità tra applicazioni client e qualunque server dati (in particolare database relazionali).

Sebbene il futuro non sarà più nel modello RDB, l'inerzia del mercato è tale che passeranno ancora diversi anni prima che i grandi sistemi migrino alla tecnologia ORDB oppure ODB. Inoltre molti problemi non hanno bisogno di sofisticate soluzioni a oggetti, difficili da gestire, per cui saranno ancora molte le nuove applicazioni sviluppate nella tecnologia RDB.

L'attuale standard per il modello RDB è noto come SQL92, ratificato nel 1992 da ANSI e ISO. Tutti i principali prodotti RDBMS sul mercato (Oracle, DB2, Sybase, Informix, SQL Server, etc.) sono conformi a tale standard, sebbene con estensioni proprietarie, come, ad esempio i *trigger*, riconosciuti solo nello standard SQL:1999.

8.4.1 Primitive di rappresentazione per RDB

Le primitive di rappresentazione per RDB sono molto semplici e derivano dal concetto matematico di *insieme*; in questo risiede sia la forza sia la debolezza del modello. I fondamenti matematici rendono la natura del modello *dichiarativa* (in contrasto con *procedurale*): l'utente dichiara *che cosa* ha bisogno dal database piuttosto che istruire il sistema su *come* cercare l'informazione (il RDBMS provvederà a cercare i dati nel database).

Ma ciò che a prima vista appare semplice diviene piuttosto complicato quando il problema da risolvere è complicato: non ci sono soluzioni semplici per problemi complessi! Per risolvere un problema complesso sono necessarie primitive raffinate di rappresentazione.

Forse, il miglior modo di caratterizzare il modello RDB è di stabilire cosa non supporta. Delle principali primitive di rappresentazione disponibili nei modelli ODB e ORDB, il modello relazionale non supporta:

- tipi oggetto e i concetti associati (quali ereditarietà e metodi);
- tipi strutturati;
- collezioni;
- riferimenti.

La principale primitiva di rappresentazione nel modello RDB è la *tabella relazionale* ovvero una matrice consistente di righe e di *colonne*. Le colonne della tabella possono assumere esclusivamente *valori atomici* mentre non sono permessi valori strutturati o collezioni.

Nel modello RDB non esistono *legami navigazionali* tra le tabelle che siano anche visibili all'utente: le relazioni tra tabelle, infatti, sono stabilite confrontando valori nelle rispettive colonne, per cui non esistono legami persistenti. La caratteristica del ORDB di mantenere relazioni predefinite tra le tabelle, è chiamata, nel modello relazionale, *integrità referenziale*.

La Figura 8.16 mostra le primitive di rappresentazione per i RDB e le loro dipendenze. Tutti i concetti hanno nomi singolari ma alcune dipendenze si applicano a più di un'istanza di concetto: ad esempio, l'integrità referenziale è definita su una o più tabelle. La maggior parte dei concetti nella Figura 8.16 sono discussi nel seguito di questo capitolo, altri sono trattati, più compiutamente, nel Capitolo 9.

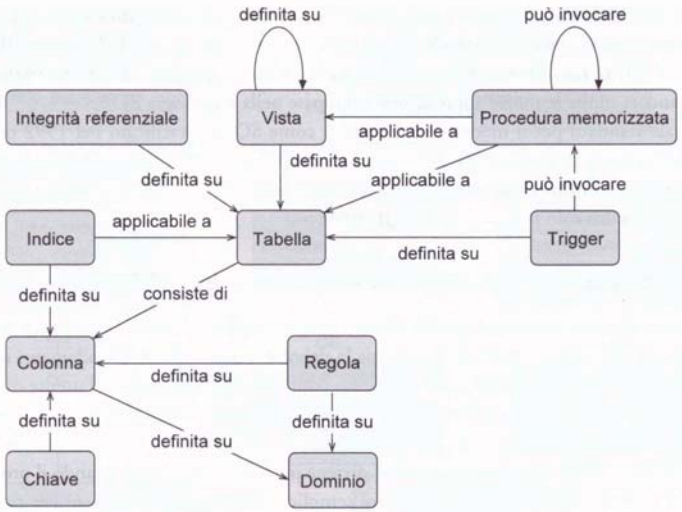


FIGURA 8.16 Dipendenze tra primitive di rappresentazione per RDB.

8.4.1.1 Colonne, domini e regole

I database relazionali definiscono i dati attraverso tabelle composte da righe e colonne. Il valore del dato memorizzato all'intersezione di una qualunque riga e colonna deve essere un unico valore semplice (indivisibile): in altre parole, si può affermare che le *colonne* hanno *domini* (tipi di dato) *atomici*.

Un *dominio* definisce l'insieme possibile di valori che una colonna può assumere. Il dominio può essere anonimo (ad esempio: `genere char(1)`) oppure dotato di nome (ad esempio: `genere Genere`). In quest'ultimo caso, il dominio `Genere` è stato definito precedentemente e quindi usato nella definizione della colonna. Una sintassi per la definizione del dominio potrebbe essere:

```
create domain Genere char(1);
```

Un *dominio con nome* può essere usato nella definizione di colonne in più tabelle. Questo forza la consistenza tra le definizioni: le modifiche alla definizione del dominio si riflettono automaticamente nelle definizioni delle colonne. Sebbene, in prima analisi, questa sia un'interessante possibilità, il suo uso è impedito una volta che il database sia stata *popolato*, cioè caricato con i dati.

Le colonne e i domini possono avere limiti imposti da *regole di business*; ad esempio, una regola di business può introdurre:

- un *valore predefinito* (se non è stato fornito alcun valore per città, si assume "`Sydney`");
- un *intervallo di valori* (l'età permessa è nell'intervallo dei numeri da `18` a `80`);
- una *lista di valori* (il colore permesso è "`verde`", "`giallo`" o "`rosso`");
- vincoli sul *formato del valore* (ad esempio, il valore deve iniziare con la lettera "K").

Per mezzo del precedente meccanismo possono essere definite solo regole molto semplici riguardanti singole colonne o domini. Regole più complesse, coinvolgenti più tabelle, possono essere definite come vincoli di *integrità referenziale*. Un ulteriore meccanismo per definire regole di business è il *trigger*.

8.4.1.2 Tabelle relazionali

Una *tabella relazionale* è definita da un insieme fisso di colonne aventi un tipo (cioè un dominio) predefinito o definito dall'utente. Le tabelle possono avere un numero qualunque di *righe* (record). Poiché una tabella è un *insieme* in senso matematico, non vi sono righe ripetute nella tabella.

Un valore colonna in una particolare riga potrebbe essere `NULL`. Il valore `NULL` può avere uno dei due seguenti significati: "valore al momento sconosciuto" oppure "valore non applicabile" (Sezione 8.2.1.1).

Una conseguenza del fatto che nelle tabelle non vi siano righe ripetute è che ogni tabella ha una *chiave primaria* (*primary key*). Una *chiave* è un insieme *minimo* di colonne (eventualmente una sola colonna) tale che i valori di dette colonne permettano di identificare *univocamente* una singola riga nella tabella. Una tabella può avere diverse chiavi, dette chiavi *candidate*: una di queste è arbitrariamente scelta come la più importante per l'utente e diventa la *chiave primaria* (le altre chiavi sono chiamate *alternative*).

In pratica, una tabella di un RDBMS può *non avere* una chiave. Questo significa che una tabella (senza chiave) potrebbe avere righe ripetute (caratteristica poco utile per un database relazionale poiché tali righe non sono distinguibili). Il discorso è diverso per i sistemi ODB e ORDB dove la presenza dell'OID fornisce sempre una distinzione (due oggetti potrebbero quindi essere equivalenti ma non identici, come, ad esempio, due copie di questo testo).

Sebbene UML possa essere stereotipato per la rappresentazione logica di database relazionali, è più conveniente usare una notazione specifica di rappresentazione. La Figura 8.17 mostra una tale notazione per il sistema DB2.

La tabella `Impiegato` consiste di otto colonne. Le ultime tre colonne accettano valore `NULL`. La colonna `id_imp` è la chiave primaria. Le colonne {`cognome`, `data_di_nascita`} definiscono la chiave candidata (alternativa). La colonna `genere` è definita sul dominio `Genere`.

Impiegato			
id_imp	CHAR(7)	<pk>	not null
cognome	VARCHAR(30)	<ak>	not null
nome	VARCHAR(20)		not null
data_di_nascita	DATE	<ak>	not null
genere	Genere		not null
num_telefono1	VARCHAR(12)		null
num_telefono2	VARCHAR(12)		null
stipendio	DEC(8,2)		null

FIGURA 8.17 Definizione di tabella in RDB.

A causa della limitazione nei RDB secondo cui una colonna può assumere solo valori singoli e atomici, qualche difficoltà è sorta nella rappresentazione del nome dell'impiegato e dei suoi numeri telefonici. Nel primo caso, sono state usate due colonne: `cognome` e `nome` (nello schema, le due colonne non sono né raggruppate né messe in relazione in qualche modo). Nel secondo caso, si è adottata la soluzione con due colonne {`num_telefono1`, `num_telefono2`} permettendo un massimo di due numeri telefonici per impiegato.

Una volta che la tabella è stata definita in uno strumento CASE, il codice per creare la tabella può essere automaticamente generato dallo strumento, come mostrato qui di seguito. Tale codice comprende la definizione del dominio `Genere` e la definizione di una regola sul dominio.

```
-- ================================================================
-- Domain: Genere
-- ================================================================
create distinct type Genere as CHAR(1) %WITHCOMPAR%;
-- ================================================================
-- Table: Impiegato
-- ================================================================
create table Impiegato (
    id_imp                  CHAR(7)                 not null,
    cognome                 VARCHAR(30)             not null,
    nome                    VARCHAR(20)             not null,
    data_di_nascita         DATE                    not null,
    genere                  Genere                  not null
        constraint C_genere check (genere in ("F","M","f","m")),
    num_telefono1           VARCHAR(12),
    num_telefono2           VARCHAR(12),
    stipendio               DEC(8,2),
primary key (id_imp),
unique (data_di_nascita, cognome)
);
```

8.4.1.3 Integrità referenziale

Il modello RDB mantiene relazioni tra tabelle per mezzo di vincoli d'*integrità referenziale*. Le relazioni non sono connessioni riga-a-riga fisse. Piuttosto, un RDB "scopre" le connessioni riga-a-riga ogniqualvolta l'utente richiede al sistema di cercare una relazione. Questa "scoperta" è fatta confrontando i valori della *chiave primaria* in una tabella con i valori della *chiave esterna* (*foreign key*) nella medesima o in altra tabella.

Una *chiave esterna* è definita come un insieme di colonne in una tabella i cui valori sono o `NULL` oppure sono gli stessi della chiave primaria nella medesima o in altra tabella. Tale corrispondenza tra le chiavi primaria ed esterna è chiamata *integrità referenziale*. Le chiavi primaria ed esterna in un'integrità referenziale devono essere definite sullo stesso dominio ma non devono necessariamente avere gli stessi nomi.

La Figura 8.18 mostra una rappresentazione grafica per l'integrità referenziale, ottenuta aggiungendo la chiave esterna `id_dip` alla tabella `Impiegato` e tracciando la relazione tra questa tabella e la tabella `Dipartimento`. Per ciascuna riga di `Impiegato`, il valore della chiave esterna può essere o `NULL` oppure deve assumere uno dei valori di `id_dip` in `Dipartimento` (altrimenti un impiegato lavorerebbe per un dipartimento inesistente). Sulla linea di relazione si definisce anche il comportamento associato all'integrità referenziale. Ci sono quattro possibili *vincoli dichiarativi d'integrità referenziale* associati con le operazioni di *cancellazione* e *aggiornamento* sulle tabelle. La questione riguarda cosa fare con le righe di `Impiegato` se una riga di `Dipartimento` è cancellata o aggiornata (cioè quando `id_dip` è aggiornato). Vi sono quattro possibilità:

1. `Upd(R); Del(R)` – (*restrict*) le operazioni di cancellazione e aggiornamento sono bloccate se ci sono ancora righe di `Impiegato` legate a un `Dipartimento`;
2. `Upd(C); Del(C)` – (*cascade*) si cancellano tutte le righe di `Impiegato` legate al `Dipartimento` cancellato;
3. `Upd(N); Del(N)` – (*null*) nelle righe di `Impiegato` legate al `Dipartimento` cancellato si assegna a `NULL` il valore di `id_dip`;
4. `Upd(D); Del(D)` – (*default*) nelle righe di `Impiegato` legate al `Dipartimento` cancellato, `id_dip` viene aggiornato con un valore *predefinito*.

La rappresentazione dell'integrità referenziale diventa complicata quando la relazione tra le tabelle è molti-a-molti, come tra `Studente` e `CorsoOfferto` (Figura 8.3 - Sezione 8.2.1.2). Per gestire il problema è necessario introdurre una *tabella ausiliaria*, quale `StdACrsOff` nella Figura 8.19. Scopo della tabella è rappresentare la relazione molti-a-molti e permettere la specifica dei vincoli d'integrità referenziale (tramite chiavi esterne corrispondenti alle chiavi primarie delle tabelle `Studente` e `CorsoOfferto`).

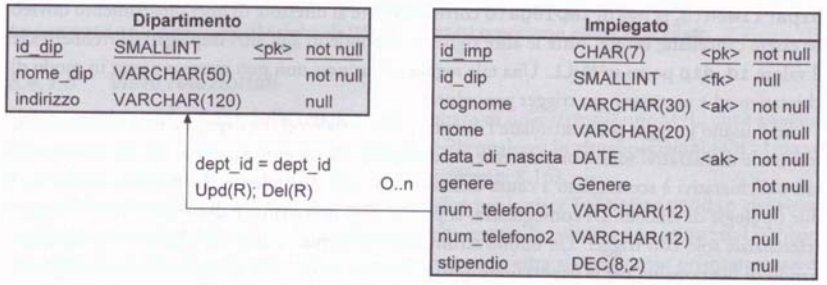


FIGURA 8.18 Integrità referenziale.

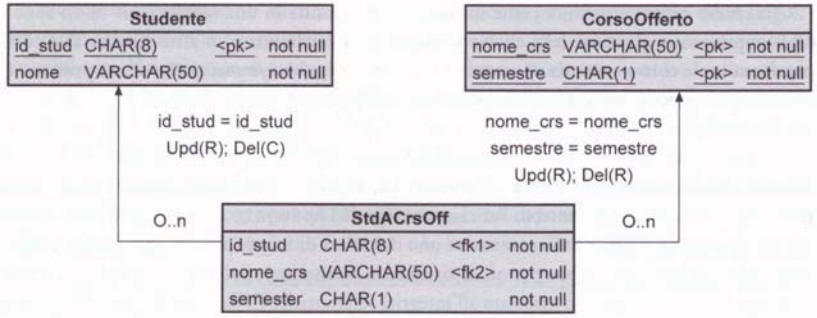


FIGURA 8.19 Integrità referenziale per relazione molti-a-molti.

8.4.1.4 Trigger

Le *regole* e i *vincoli dichiarativi d'integrità referenziale* permettono di definire semplici vincoli (regole di business) sul database. Non sono tuttavia sufficienti per definire regole più complesse o per definire possibili eccezioni alle regole. Una soluzione a questo problema nei RDB (standardizzata in SQL:1999) è il trigger.

Un *trigger* è un programma (generalmente di dimensioni ridotte), scritto usando un'estensione di SQL, eseguito automaticamente in seguito a un'operazione di modifica sulla tabella cui il trigger è associato. Un'operazione di modifica può essere una qualunque istruzione di modifica SQL: `insert`, `update` o `delete`.

Un trigger può essere usato per implementare *regole di business* significativamente più complesse di quelle discusse nella Sezione 8.4.1.1. Ad esempio, la regola di business che impedisce di effettuare modifiche alla tabella `Impiegato` durante il fine settimana, può essere programmata in un trigger che scattando a fronte di un qualsiasi tentativo di modifica sulla tabella esegue una verifica sulla data provocando, durante il fine settimana, il rifiuto del database di eseguire l'operazione.

Un trigger può anche essere usato per forzare vincoli d'integrità referenziale più elaborati. Ad esempio, una regola potrebbe stabilire che alla cancellazione di una riga (un dipartimento) in `Dipartimento`, la riga di `Impiegato` corrispondente al direttore di quel dipartimento dovrebbe essere cancellata; invece, tutte le altre righe corrispondenti agli altri impiegati dovrebbero aver il valore `id_dip` posto su `NULL`. Una tale regola di business non può essere espressa in modo dichiarativo ed è necessario un trigger procedurale.

Se si usano i trigger per controllare l'integrità referenziale nel database, i vincoli d'integrità referenziale dichiarativi sono normalmente abbandonati. L'uso contemporaneo di vincoli procedurali e dichiarativi è sconsigliato a causa delle intricate interdipendenze che possono crearsi tra le due tipologie di vincolo. Di conseguenza, la pratica oggi dominante è di programmare l'integrità referenziale solo con trigger. Un buono strumento CASE può aiutare in questo compito, disponendo di funzionalità per la generazione automatica del codice. Un esempio di codice trigger generato da uno strumento CASE, per il RDBMS Sybase, è mostrato qui di seguito. Il trigger implementa il vincolo dichiarativo `Del(R)`, cioè non permette la cancellazione di una riga di `Dipartimento` se a essa sono legate righe di `Impiegato`.


```
create trigger manteneredip
      on Dipartimento
      for delete
      as
      if @@rowcount = 0
            return /* evita di eseguire il trigger se non ci sono
            righe interessate */
      if exists
            (select * from Impiegato, deleted
                  where Impiegato.id_dip = deleted.id_dip)
      begin
            print "Test CANCELLAZIONE RISTRETTA : nessuna
            cancellazione"
            rollback transaction
            return
      end
      return
go
```

La prima istruzione `if` controlla se l'operazione SQL `delete` (che ha mandato in esecuzione il trigger) potrebbe cancellare delle righe. Se ciò non accade, il trigger non procede: infatti, nessun danno può essere causato. Se alcune righe di `Dipartimento` possono essere cancellate, il sistema Sybase le memorizza in una tabella interna chiamata `deleted`. Il trigger esegue quindi una ricerca (operazione di *join* su `id_dip` tra le tabelle `Impiegato` e `deleted`) per scoprire se esiste qualche impiegato che lavora per il(i) dipartimento(i) da cancellare. Se ne esistono, il trigger rifiuta l'azione `delete`, visualizza un messaggio e annulla la transazione (istruzione `rollback transaction`). Altrimenti, le righe di `Dipartimento` possono essere cancellate.

8.4.1.5 Viste relazionali

Una *vista relazionale* è una tabella temporanea, risultato di un'interrogazione SQL, cui è assegnato un nome. La vista può essere usata come una tabella qualsiasi in altre operazioni SQL. Una vista può essere derivata da una o più tabelle e/o viste (Figura 8.16).

La Figura 8.20 mostra una rappresentazione grafica per la vista `ImpNoStipendio` che visualizza tutte le informazioni della tabella `Impiegato` eccetto la colonna `stipendio`. L'istruzione `create view` sottostante la figura mostra come una vista sia realmente un'interrogazione con nome.

FIGURA 8.20 Vista relazionale.

```
=================================================================
— View: ImpNoStipendio
=================================================================
create view ImpNoStipendio () as
      select Impiegato.id_imp, Impiegato.id_dip,
             Impiegato.cognome, Impiegato.nome,
             Impiegato.data_di_nascita, Impiegato.genere,
             Impiegato.num_telefono1, Impiegato.num_telefono2
      from   Impiegato;
```

Teoricamente, la vista è un potente meccanismo che può avere molti usi. Può essere usato per supportare la *sicurezza del database*, non permettendo agli utenti di vedere i dati di una tabella (ad esempio, lo stipendio degli impiegati nell'esempio precedente). Può presentare i dati agli utenti sotto *diverse prospettive*. Può *isolare l'applicazione dalle modifiche* alle definizioni di tabelle, se, evidentemente, le definizioni modificate non incidono sulla vista stessa. Permette una più semplice espressione delle *interrogazioni complesse*: l'interrogazione può essere costruita usando livelli multipli di viste.

Nella pratica, l'uso del concetto di vista nel modello RDB è severamente limitato dall'incapacità (e spesso dall'impossibilità teorica) di permettere gli aggiornamenti a partire dalle viste stesse. Un *aggiornamento di vista* è la possibilità di effettuare un'operazione di modifica (`insert`, `update`, `delete` in SQL) sulla vista e di cambiare, come risultato, le sottostanti tabelle di base. Il supporto SQL92 per l'aggiornamento di viste è molto limitato, in sostanza inesistente.

8.4.1.6 Forme normali

Uno dei concetti più importanti ma allo stesso tempo meno compresi nel progetto di un RDB è la *normalizzazione*. Una tabella relazionale deve essere in una qualche *forma normale* (NF). Esistono sei forme normali, elencate di seguito nel loro ordine naturale, dalle inferiori alle superiori:

- 1 NF
- 2 NF
- 3 NF
- BCNF (NF Boyce-Codd)
- 4 NF
- 5 NF

Una tabella che è in una forma normale superiore è anche in tutte le forme normali inferiori. Una tabella le cui colonne non sono né strutturate né multi-valore è nella prima forma normale (1 NF) (è il requisito fondamentale del modello relazionale).

Una tabella in una "bassa" forma normale può esibire delle anomalie, chiamate *anomalie di aggiornamento*, come effetto collaterale conseguente a un'operazione di modifica (`insert`, `update`, `delete`) sulla tabella stessa. Ad esempio, se la stessa informazione è ripetuta più volte nella stessa colonna di una tabella, allora un suo aggiornamento deve essere riportato in tutte le righe contenenti l'informazione ripetuta (altrimenti il database è lasciato in uno stato scorretto). Si può dimostrare che le anomalie di aggiornamento si riducono passando a forme normali superiori.

Il passaggio di una tabella a una forma normale superiore avviene scomponendola opportunamente in due o più tabelle ridotte. La tabella originale può tuttavia essere sempre ricostruita unendo con un'operazione SQL di `join` le tabelle ridotte.

Lo scopo di questo testo non riguarda la teoria della normalizzazione (il lettore interessato può far riferimento a numerosi testi come: Date, 2000; Maciaszek, 1990; Silberschatz *et al.*, 1997; Ramakrishnan e Gehrke, 2000); qui è sufficiente osservare che un buon progetto di RDB corrisponde generalmente a un buon livello di normalizzazione.

Un *buon progetto* dipende dalla possibilità di valutare come l'intero insieme di operazioni d'aggiornamento e recupero dell'informazione usa il RDB. Se il database è molto dinamico cioè soggetto a frequenti operazioni di aggiornamento, allora conviene creare tabelle più ridotte (più *normalizzate*) per meglio localizzare e facilitare tali aggiornamenti. Se, viceversa, il database è relativamente statico, cioè prevalgono le operazioni di interrogazione e gli aggiornamenti sono sporadici, allora è più indicato un *progetto denormalizzato* (la ricerca in una singola "grande" tabella è molto più efficiente della medesima ricerca su diverse tabelle normalizzate la cui esecuzione richiede comunque la ricomposizione preliminare, con il `join`, delle tabelle coinvolte).

8.4.2 Trasformazione UML - RDB

La trasformazione di un modello delle classi UML in uno schema di RDB deve considerare le limitazioni del modello RDB. Il problema è trasformare la *semantica dichiarativa* dei diagrammi delle classi in schemi logici relazionali di progetto: questo potrebbe non essere possibile. In tal caso, la semantica delle classi deve essere trattata proceduralmente nei programmi del database, cioè con procedure memorizzate (Sezione 9.1.2.1).

La trasformazione in modelli per RDB è stata ampiamente studiata nel contesto della rappresentazione ER e sue estensioni (ad esempio Maciaszek, 1990; Elmasri e Navathe, 2000). I principi sono gli stessi e tutti i maggiori problemi sono stati identificati negli studi condotti. Come nel caso dei modelli ORDB e ODB, la trasformazione non deve semplicemente essere conforme a standard (SQL92 nel caso RDB) ma deve essere in relazione al RDBMS scelto per l'implementazione.

8.4.2.1 Trasformazione delle classi entity

La trasformazione delle classi entity in tabelle relazionali deve produrre tabelle in 1 NF (questo implica che le colonne debbano essere atomiche). Nei modelli UML che derivano dall'analisi dei requisiti, come osservato nella Sezione 8.2.2.1, gli attributi delle classi (contrariamente alle relazioni tra classi) sono generalmente atomici: questo fatto, ovviamente, semplifica la trasformazione.

Esempio 8.9 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti, le specifiche delle classi nell'Esempio 4.6 (Figura 4.3 - Sezione 4.2.1.2.3) e la discussione concernente gli attributi non atomici nell'Esempio 8.1.

Trasformare le classi `Contatto` e `Impiegato` in un progetto RDB utilizzando strategie alternative di trasformazione.

La soluzione dell'esempio, che presuppone l'uso del RDBMS Oracle per l'implementazione, è mostrata nella Figura 8.21. L'attributo `nome_contatto` è stato rappresentato con un tipo dato atomico nella tabella `Contatto`. Ogni oggetto di `Contatto` può aver solo un `fax` e un `email`. Tuttavia, è possibile avere più numeri di `telefono`. La tabella `TelefonoContatto` serve a questo proposito (si è supposto che i numeri di telefono siano tutti distinti).

Nella tabella `Impiegato` si mantengono tre attributi distinti per `cognome`, `nome` e `iniziale_secondo_nome`. Si osservi che il database non ha alcuna conoscenza di `nome_impiegato` quale combinazione dei tre attributi.

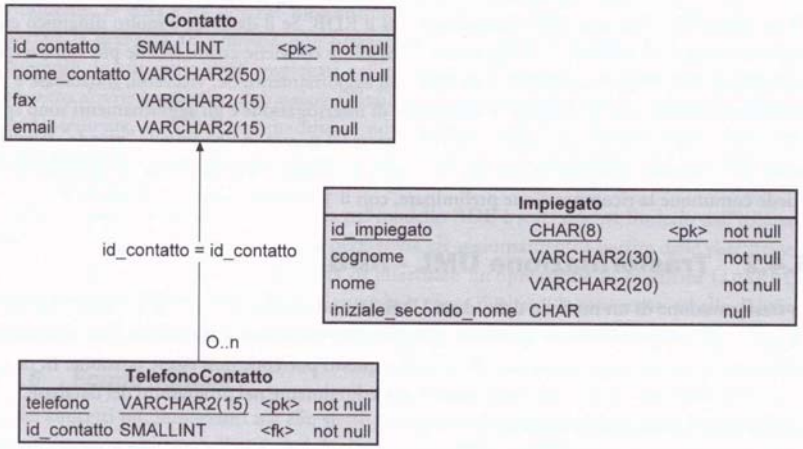


FIGURA 8.21 Trasformazione delle classi entity in schema RDB (CMS).

8.4.2.2 Trasformazione delle associazioni

La trasformazione delle associazioni verso un RDB coinvolge l'uso dei vincoli d'*integrità referenziale* tra le tabelle. Qualunque associazione uno-a-uno oppure uno-a-molti può essere direttamente espressa inserendo una *chiave esterna* in una tabella corrispondente alla chiave primaria dell'altra tabella.

Nel caso di *associazione uno-a-uno,* la chiave esterna può essere aggiunta a una qualunque delle due tabelle (la scelta dipende dalla direzione in cui viene prevalentemente usata). Sempre nel caso dell'associazione uno-a-uno, potrebbe essere utile combinare le due classi entity in una singola tabella (in funzione del livello di normalizzazione desiderato).

Per le associazioni uno-a-uno e uno-a-molti *ricorsive*, la chiave esterna e la chiave primaria sono nella stessa tabella. Ogni *associazione molti-a-molti* (ricorsiva o no) richiede una tabella ausiliaria come mostrato in Figura 8.19.

Esempio 8.10 (CMS)

Si considerino il caso di studio Gestione Contatti con i Clienti e le specifiche di associazione nell'Esempio 4.8 (Figura 4.5 - Sezione 4.2.2.3).

Trasformare il diagramma di Figura 4.5 in un progetto RDB.

Quest'esempio è piuttosto semplice a causa della mancanza di associazioni molti-a-molti nelle specifiche UML. Il diagramma RDB (per il RDBMS DB2) è mostrato nella Figura 8.22. Coerentemente con i principi RDB, sono state create diverse nuove colonne come chiavi primarie (per risparmiare spazio, è stata soppressa l'indicazione di annullabilità e di chiave riguardanti le colonne).

I vincoli d'integrità referenziale tra `IndirizzoPostale` e `IndirizzoCorriere` da un lato e `Organizzazione` e `Contatto` dall'altro, sono stati rappresentati con chiavi esterne nelle tabelle contenenti gli indirizzi. Ciò è arbitrario e i vincoli avrebbero potuto essere rappresentati nel modo opposto (cioè con le chiavi esterne in `Organizzazione` e `Contatto`).

8.4.2.3 Trasformazione delle aggregazioni

Un RDB non distingue tra associazione e aggregazione eccetto quando siano implementate proceduralmente con trigger e procedure memorizzate (in questo caso la semantica addizionale dell'aggregazione può essere specificata in modo esplicito). I principi fondamentali per la trasformazione di associazioni (Sezione 8.4.2.2) sono validi per la trasformazione di aggregazioni. Solo quando un'associazione può essere convertita in diverse soluzioni relazionali (ad esempio, per associazioni uno-a-uno) la semantica dell'aggregazione (come forma speciale di associazione) potrebbe influenzare la scelta.

Nel caso della *composizione*, dovrebbe essere fatto un primo tentativo di combinare le classi componente e composta in una singola tabella. Ciò è certamente possibile per aggregazioni uno-a-uno. Per le *aggregazioni uno-a-molti*, la classe componente (nelle forme debole e forte di aggregazione) deve essere rappresentata come una tabella distinta (con una chiave esterna che la lega alla propria tabella composta).

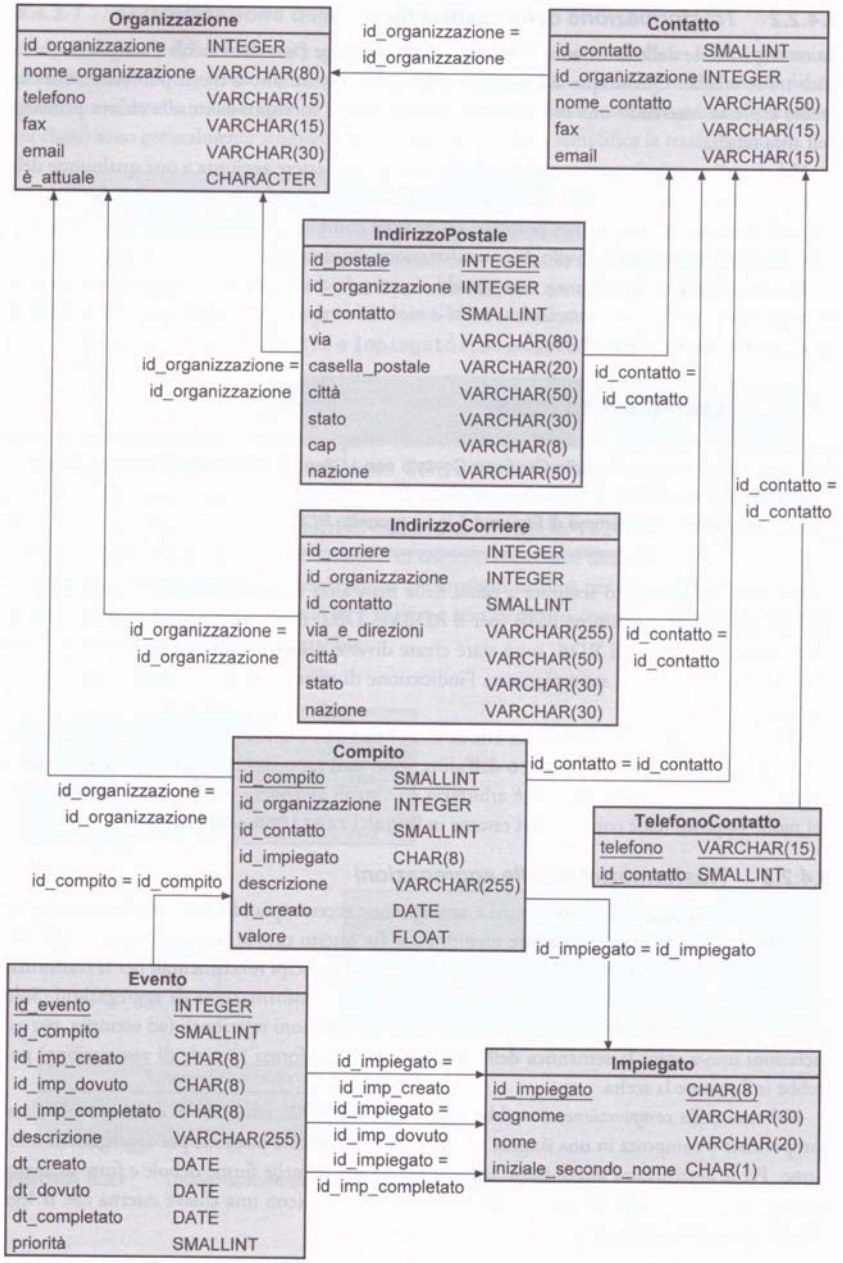


FIGURA 8.22 Trasformazione delle associazioni in schema RDB (CMS).

Esempio 8.11 (Immatricolazioni universitarie)

Si considerino le specifiche d'aggregazione per le Immatricolazioni universitarie nell'Esempio 4.9 (Figura 4.6 - Sezione 4.2.3.3).

Trasformare il diagramma di Figura 4.6 in un progetto per RDB.

L'esempio comprende una composizione da `Studente` a `InformazioneAccademica` e un'aggregazione da `Corso` a `CorsoOfferto`. Entrambe sono aggregazioni uno-a-molti e richiedono tabelle "componente" distinte.

Nel modello UML di Figura 4.6, esiste un legame indiretto di navigazione da `InformazioneAccademica` a `Corso`; inoltre, `InformazioneAccademica` ha un attributo `codice_corso` come parte della propria chiave primaria. Tale attributo potrebbe essere usato per stabilire un'integrità referenziale diretta tra le tabelle `InformazioneAccademica` e `Corso`. Lo stesso attributo è una chiave esterna nella tabella `CorsoOfferto`, come mostrato nella Figura 8.23 (per il RDBMS Informix).

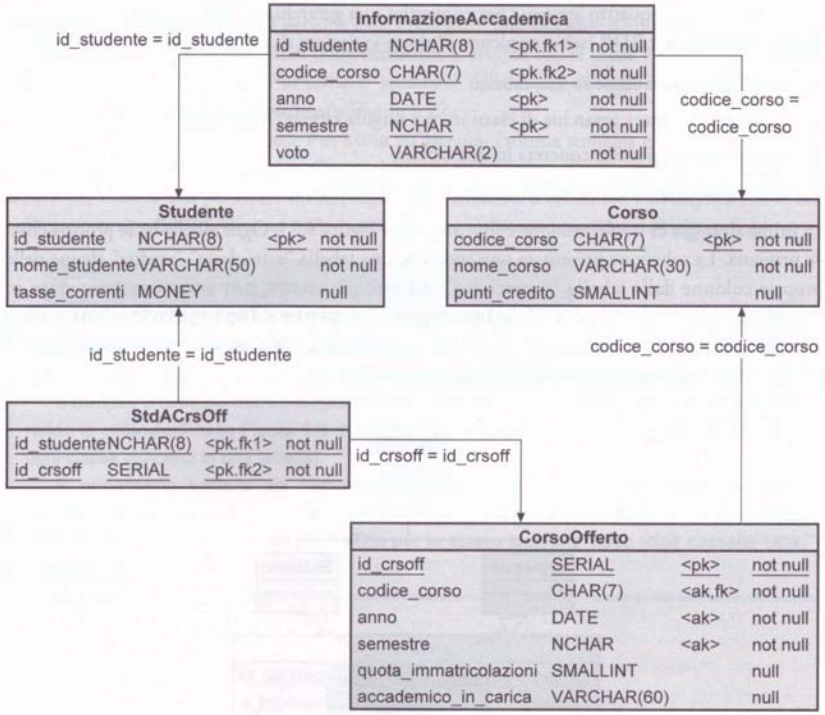


FIGURA 8.23 Trasformazione delle aggregazioni in schema RDB (Immatricolazioni universitarie).

L'*associazione molti-a-molti* tra le classi `Studente` e `CorsoOfferto` conduce a un'altra interessante osservazione, benché non in relazione con la trasformazione delle aggregazioni. L'associazione è trasformata in una tabella ausiliaria `StdACrsOff` con chiave primaria formata dalle chiavi primarie delle due tabelle principali.

La chiave primaria per `CorsoOfferto` potrebbe essere {`codice_corso,anno,semestre`}. Tuttavia, una tale chiave risulterebbe poco pratica, anche per `StdACrsOff`. Si è quindi preferito usare in `CorsoOfferto` una chiave primaria generata dal sistema e chiamata `id_crsoff` con tipo `SERIAL` (in Informix, il tipo per generare identificatori unici è chiamato `SERIAL`; lo stesso tipo potrebbe essere chiamato in qualche altro modo in altri RDBMS: in Sybase è chiamato `IDENTITY`, `UNIQUEIDENTIFIER` in Microsoft SQL Server, e `SEQUENCE` in Oracle).

8.4.2.4 Trasformazione delle generalizzazioni

La trasformazione delle relazioni di generalizzazione verso un RDB può essere fatta in diversi modi; tuttavia, i principi sottostanti sono meno intricati di quanto ci si potrebbe aspettare. Si deve osservare, tuttavia, che nell'esprimere una generalizzazione in una struttura dati per RDB s'ignorano gli aspetti caratteristici della generalizzazione: l'ereditarietà, il polimorfismo, il riuso del codice, etc.

Per illustrare le strategie di trasformazione delle generalizzazioni si consideri l'esempio in Figura 8.24. Esistono quattro strategie per convertire una gerarchia di generalizzazione in un modello di progetto per RDB (sebbene alcune ulteriori variazioni di tali strategie siano possibili):

1. trasformare ogni classe in una tabella;
2. trasformare l'intera gerarchia di classi in una singola tabella "superclasse";
3. trasformare ogni classe concreta in una tabella;
4. trasformare ogni classe concreta disgiunta in una tabella.

La prima strategia di trasformazione è illustrata nella Figura 8.25. Ogni tabella ha la propria chiave primaria. La soluzione presentata non indica se una tabella "sottoclasse" "eredita" alcune delle proprie colonne dalla tabella "superclasse". Ad esempio, `nome_persona` è memorizzato in `Persona` e può essere "ereditato" da `Impiegato`, `Studente` e `ImpiegatoStudente` attraverso operazioni di join: per questo, le prestazioni risultano penalizzate e potrebbe essere utile replicare `nome_persona` in tutte le tabelle della gerarchia.

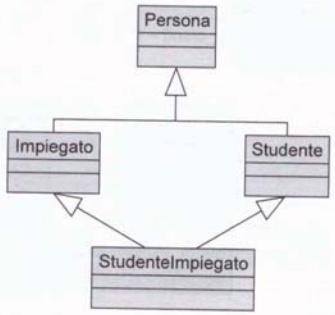


FIGURA 8.24 Gerarchia di generalizzazione per esemplificare la trasformazione in RDB.

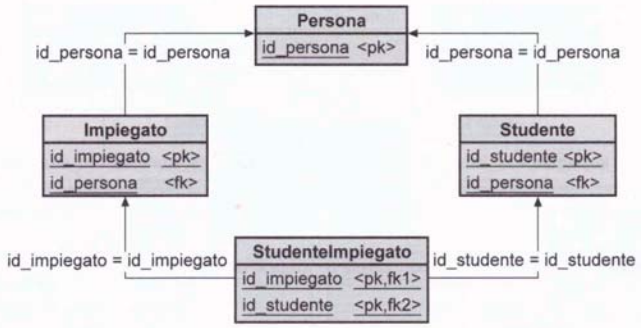


FIGURA 8.25 Trasformazione di ogni classe in tabella.

La seconda strategia di trasformazione è illustrata nella Figura 8.26 (per il RDBMS Microsoft SQL Server). La tabella `Persona` contiene tutti gli attributi di tutte le classi nella gerarchia di generalizzazione. Contiene anche due colonne (`è_impiegato` ed `è_studente`), con tipo `BOOLEAN`, per registrare se una persona è un impiegato, uno studente o entrambi.

Per illustrare la terza strategia di trasformazione, si assuma che la classe `Persona` sia astratta. Qualsiasi attributo della classe `Persona` è "ereditato" dalle tabelle corrispondenti alle classi concrete. Il risultato è quello mostrato nella Figura 8.27.

Assumendo ancora che la classe `Persona` sia astratta, l'ultima strategia è illustrata nella Figura 8.28 (per il RDBMS Informix). Contrariamente al modello nella Figura 8.25, si assuma che sia sempre noto se una persona è un impiegato, uno studente o entrambi. Ciò implica che le due colonne con tipo `BOOLEAN` sono `not null` (cioè non possono assumere il valore nullo).

Esempio 8.12 (Video Store)

Si considerino le specifiche delle generalizzazioni per il caso Video Store nell'Esempio 4.10 (Figura 4.7 - Sezione 4.2.4.3) e le estensioni introdotte nell'Esempio 8.4 (Sezione 8.2.2.4).

Il compito è di trasformare il diagramma in Figura 4.7 in un modello per un RDB, considerando le estensioni nella Figura 8.8 (Esempio 8.4), e usando la terza strategia (trasformare ogni classe concreta in una tabella).

È necessario trattare sia l'attributo derivato `disponibile` sia l'attributo statico `percentuale_video_ecc`. Come nei precedenti modelli per il Video Store, si ignori la possibilità di avere più d'una condizione d'affitto per lo stesso supporto video (cioè cassetta beta, cassetta vhs e disco dvd).

Persona			
id_persona	uniqueidentifier	<pk>	not null
è_impiegato	char(1)		null
è_studente	char(1)		null

FIGURA 8.26 Trasformazione della gerarchia di classi in tabella.

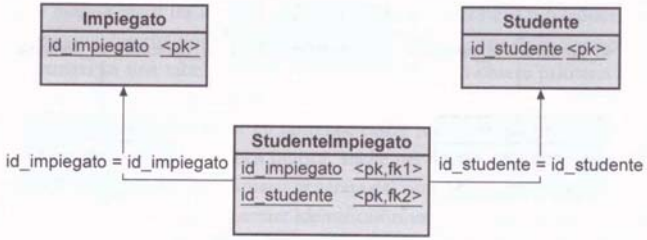


FIGURA 8.27 Trasformazione di ogni classe concreta in tabella.

Impiegato			
id_impiegato	NCHAR(8)	<pk>	not null
è_studente	BOOLEAN		not null

Studente			
id_studente	NCHAR(10)	<pk>	not null
è_impiegato	BOOLEAN		not null

FIGURA 8.28 Trasformazione di ogni classe concreta disgiunta in tabella.

Il progetto RDB (per il RDBMS Sybase) nella Figura 8.29 contiene una tabella per ciascuna delle tre classi concrete (`CassettaBeta`, `CassettaVHS` e `DiscoDVD`). Nelle tabelle è replicata l'informazione su `condizione_video`, `periodo_affitto_in_giorni` e `spesa_affitto_per_periodo`; solo un `periodo_affitto_in_giorni` e un `spesa_affitto_per_periodo` possono essere registrati per cassetta o disco.

Le colonne `è_registrata`, `linguaggi_diversi`, `finali_differenti` e `disponibile` sono tipati come `bit`. Per definizione, il tipo `bit` non permette valori nulli (il `bit` è zero o uno).

La colonna `disponibile` è messa a vero (uno) se nella scorta c'è almeno una cassetta o disco con un particolare film. Questa è un'informazione non molto utile se un cliente è interessato a un particolare tipo di supporti video. Una soluzione migliore potrebbe introdurre tre colonne `disponibile` di tipo `bit` oppure non memorizzare l'informazione e derivarla ogniqualvolta un cliente richiede un disco o una cassetta.

L'attributo statico `percentuale_video_ecc.` non è memorizzato in alcuna tabella del progetto. La sola tabella ove potrebbe essere memorizzato è `TitoloFilm`. Se memorizzato, varrebbero le medesime considerazioni fatte per l'attributo `disponibile`.

Sommario

Questo capitolo ha discusso e analizzato l'importanza preminente dei database nello sviluppo del software. Un cattivo progetto del database non può essere compensato con altri mezzi, incluso un buon progetto delle applicazioni. Sono stati trattati i tre principali modelli di database, il modello a oggetti, quello object-relational e il relazionale.

Esistono tre livelli di modelli per database, esterno, logico e fisico, ma questo capitolo ha trattato il livello del *modello logico*. Il *progetto del database* è inteso come la trasformazione di un modello delle classi UML in un modello logico di database.

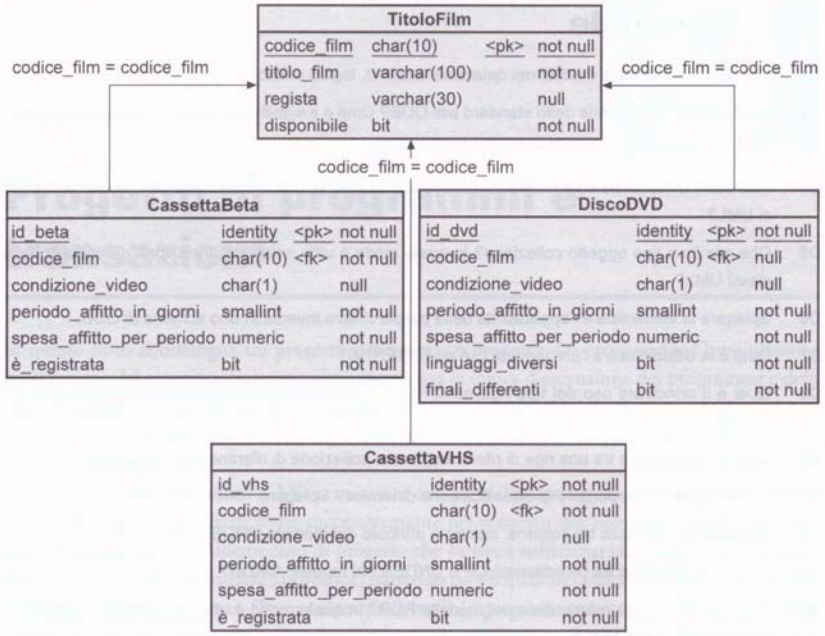


FIGURA 8.29 Trasformazione delle generalizzazioni in schema RDB (Video Store).

La trasformazione verso il *modello logico ODB* è la più semplice. Il modello ODB supporta direttamente i *tipi oggetto*, i *letterali strutturati* e le *collezioni*. Le relazioni sono rappresentate con gli *inversi*. L'ereditarietà *ISA* ed *EXTENDS* può essere usata per trasformare le generalizzazioni UML. Le aggregazioni UML possono essere trasformate usando *collezioni*.

Il *modello logico ORDB* è più complesso del modello ODB. Il modello ORDB fa una forte distinzione tra il *tipo strutturato* e la *tabella* come unico meccanismo di memorizzazione. Un *tipo riga* permette la specifica di strutture dati annidate, una possibilità che non è direttamente disponibile in UML. *Riferimenti* e *collezioni di riferimenti* sono anche supportati. L'ereditarietà *OF* e *UNDER* può essere usata per trasformare le generalizzazioni UML.

La trasformazione verso il *modello logico RDB* è la più elaborata. Il modello RDB non supporta i tipi oggetto, l'ereditarietà, i tipi strutturati, collezioni o riferimenti. I dati sono memorizzati nelle *tabelle*, tra cui esistono relazioni rappresentate con vincoli d'*integrità referenziale*. I *trigger* e le procedure memorizzate possono essere usati per programmare la semantica delle classi UML non esprimibile direttamente nelle tabelle (aggregazione e generalizzazione). La *normalizzazione* può ulteriormente influenzare la trasformazione.

Domande

D1 Discutere i tre livelli di modelli nei database: esterno, logico e fisico.

D2 Qual è stata l'evoluzione dello standard per ODB? Qual è il significato pratico di quest'evoluzione?

D3 Qual è la differenza tra il tipo e la classe nella ODB?

D4 Che cos'è un letterale collezione? In quale modo è utile nella trasformazione dei modelli delle classi UML?

D5 Che cos'è un tipo oggetto collezione? In quale modo è utile nella trasformazione dei modelli delle classi UML?

D6 Spiegare la semantica e l'applicabilità della parola chiave inverse in uno schema di ODB.

D7 Qual è la differenza tra l'ereditarietà ISA ed EXTENDS?

D8 Qual è il principale uso del tipo riga nel ORDB? Può essere usato per definire tipi riferimento? Spiegare.

D9 Qual è la differenza tra una riga di riferimenti e una collezione di riferimenti in un ORDB?

D10 I sistemi ORDB supportano la classificazione dinamica? Spiegare.

D11 Spiegare la differenza tra colonna, campo e attributo del modello ORDB.

D12 Qual è la differenza tra l'ereditarietà OF e UNDER nel modello ORDB?

D13 Che cos'è l'integrità referenziale nel modello RDB? In quale modo è utile nella trasformazione dei modelli delle classi UML?

D14 Spiegare i quattro vincoli dichiarativi d'integrità referenziale del modello RDB.

D15 Che cos'è un trigger? Qual è la sua relazione con l'integrità referenziale?

D16 Che cos'è un buon livello di normalizzazione in un RDB? Spiegare.

Esercizi

E1 Telemarketing - si consideri il caso di studio Telemarketing e la soluzione dell'Esercizio E1 nel Capitolo 7.

Trasformare il diagramma delle classi in uno schema ODB. Spiegare la trasformazione.

E2 Telemarketing - si consideri il caso di studio Telemarketing e la soluzione dell'Esercizio E1 nel Capitolo 7.

Trasformare il diagramma delle classi in uno schema ORDB. Spiegare la trasformazione.

E3 Telemarketing - si consideri il caso di studio Telemarketing e la soluzione dell'Esercizio E1 nel Capitolo 7.

Trasformare il diagramma delle classi in uno schema RDB. Spiegare la trasformazione.

Capitolo 9

Progetto di programmi e transazioni

In questo libro si distingue tra progetto del sistema e progetto dei programmi. Quest'ultimo è quell'aspetto del progetto di sistema che rappresenta la logica d'esecuzione dei programmi e definisce l'ambiente per la collaborazione tra client e server. I programmi del sistema informativo eseguono delle transazioni di business (una transazione è un'unità di consistenza del database, verificata dal DBMS).

Il progetto dei programmi e delle transazioni completa il progetto del sistema: produce la specifica della logica applicativa, usata successivamente nel progetto dell'interfaccia GUI e del database. Tale specifica è un documento di progetto che fornisce sufficienti istruzioni al programmatore per permettergli di iniziare la codifica, partendo da un qualche codice iniziale che, in genere, è automaticamente generato ("forward-engineering") impiegando moderni strumenti CASE. Le estensioni fatte dal programmatore al codice iniziale possono essere ribaltate ("reverse-engineering") verso i modelli UML di progetto, sempre attraverso lo strumento CASE, chiudendo il ciclo detto di "round-trip engineering" (tecnica circolare di sviluppo).

9.1 Progettare i programmi

Il *progetto dei programmi* è parte intrinseca del progetto del sistema (Capitoli 6, 7, 8). Il *progetto architetturale* dei package delle classi e dei componenti stabilisce l'ambiente generico d'esecuzione. Il *progetto dettagliato* dell'interfaccia GUI e del database specifica il "front-end" e il "back-end" di quell'ambiente. Il progetto dei programmi completa la specifica generica dell'ambiente d'esecuzione, trasformandola in un documento di progetto che può essere passato a un programmatore per l'implementazione finale.

Il progetto dei programmi prende in considerazione un programma applicativo alla volta. In tal senso, il progetto dei programmi è un'estensione diretta del *progetto dell'interfaccia utente* discussa nel Capitolo 7. Il progetto dei programmi usa un sottoschema dello *schema del database* (Capitolo 8) e definisce gli aspetti procedurali del database, procedure memorizzate e trigger specifici per ciascun programma considerato.

La logica d'esecuzione del programma si suddivide tra i processi client e server. Il *processo client* comprende la maggior parte delle collaborazioni tra oggetti (Sezione 6.2): un corretto bilanciamento tra la coesione e l'accoppiamento tra oggetti (si veda in seguito la Sezione 9.1.1) permette di dominare la complessità di quelle collaborazioni. Il *processo server* si prende cura, tra le altre cose, di eseguire le transazioni di business iniziate dal *processo client*.

9.1.1 Coesione e accoppiamento di classi

In precedenza (Capitoli 5 e 6), sono stati identificati i principi fondamentali per un buon progetto dei programmi, visti nel contesto più generale di un buon progetto del sistema. La *stratificazione delle classi* è il fondamento per la scrittura di programmi comprensibili, manutenibili, e scalabili. Il corretto uso di *ereditarietà* e *delegazione* è necessario per evitare di produrre programmi object-oriented che diventino sistemi incomprensibili il giorno dopo essere stati rilasciati ai clienti.

Un buon progetto dei programmi dipende da un buon bilanciamento tra la *coesione* e l'*accoppiamento* delle classi. Tali termini, introdotti con i metodi di progettazione strutturata, hanno nella progettazione object-oriented connotazione e importanza analoghe (Page-Jones, 2000; Schach, 1996).

La *coesione delle classi* misura il grado di indipendenza o autodeterminazione interna della classe. Una classe molto coesa è in grado di eseguire un'azione complessa oppure di raggiungere un singolo obiettivo: per questo, maggiore è la coesione, meglio è.

L'*accoppiamento tra le classi* misura il grado di interdipendenza o connessione tra le classi: quindi, minore è l'accoppiamento, meglio è (tuttavia, le classi devono essere "accoppiate" per collaborare!).

La coesione e l'accoppiamento sono, di conseguenza, in contrasto tra loro. Una maggiore coesione induce un minor accoppiamento e viceversa. Il compito del progettista è di raggiungere il miglior equilibrio tra i due aspetti. Riel (1996) ha proposto diverse euristiche per trattare questo problema:

- Due classi dovrebbero essere indipendenti tra loro oppure una classe dovrebbe dipendere dall'altra solo tramite l'interfaccia pubblica di quest'ultima.
- Gli attributi e i metodi associati dovrebbero essere raccolti in una classe (questa euristica è violata con frequenza da classi con molti metodi d'*accesso* (metodi `set` e `get`) riportati nelle loro interfacce pubbliche).
- Una classe dovrebbe corrispondere a una e una sola astrazione; quando sottoinsiemi disgiunti di metodi operano su sottoinsiemi disgiunti di attributi, è bene separare le funzionalità in classi distinte.
- L'intelligenza del sistema dovrebbe essere distribuita il più possibile uniformemente (cosicché le classi condividano uniformemente il lavoro).

9.1.1.1 Tipologie di accoppiamento tra classi

Affinché due classi possano comunicare, è necessario che siano "accoppiate". L'*accoppiamento* tra la classe X e la classe Y esiste se la classe X può *direttamente* riferirsi alla classe Y. Page-Jones (2000) elenca otto tipologie di accoppiamento tra classi (che chiama insieme di riferimento diretto tra classi):

1. X eredita da Y.
2. X ha un attributo di classe Y.
3. X ha un attributo con parametro di classe Y.
4. X ha un metodo con un argomento in ingresso di classe Y.
5. X ha un metodo con un argomento in uscita di classe Y.
6. X può accedere a una variabile globale di classe Y.
7. X può accedere a un metodo contenente una variabile locale di classe Y.
8. X è un'amica di Y.

9.1.1.2 La Legge di Demeter

L'accoppiamento tra classi è necessario alla comunicazione tra oggetti ma, come detto nella Sezione 5.2, dovrebbe essere confinato quanto possibile, all'*interno* degli strati delle classi (cioè all'*accoppiamento intra-strato*). L'*accoppiamento inter-strato* dovrebbe essere ridotto e incanalato con attenzione. Una guida aggiuntiva per limitare la comunicazione arbitraria tra classi è offerta dalla Legge di Demeter (Lieberherr e Holland, 1989).

La Legge di Demeter specifica quali destinatari sono permessi per i messaggi generati all'interno dei metodi di una classe, stabilendo che il destinatario di un messaggio può essere solo uno dei seguenti oggetti (Page-Jones, 2000):

1. il mittente stesso del metodo (cioè `this` in C++ e Java, `self` e `super` in Smalltalk),
2. un oggetto che è anche un argomento nella segnatura del metodo,
3. un oggetto cui fa riferimento un attributo dell'oggetto mittente (anche se l'attributo dell'oggetto mittente fa parte di una collezione),
4. un oggetto creato dal metodo,
5. un oggetto cui fa riferimento una variabile globale.

Per restringere l'accoppiamento indotto dall'ereditarietà, la terza regola può essere limitata agli attributi definiti nella classe stessa. Un attributo ereditato dalla classe non può essere usato all'interno di un metodo per identificare un oggetto destinatario di un messaggio. Questo vincolo è noto come *Legge Forte di Demeter* (Page-Jones, 2000).

9.1.1.3 Metodi di accesso e dipendenza di classi

Come detto nella Sezione 9.1.1, gli attributi e i relativi metodi dovrebbero essere raccolti in una classe (Riel, 1996). Una classe dovrebbe decidere del proprio destino; ad esempio, una classe può non permettere ad altre classi di accedere al proprio stato, limitando i metodi di accesso nella propria interfaccia pubblica.

I *metodi di accesso* riguardano le operazioni *osserva* (`get`) e *modifica* (`set`) (Sezione 8.3.1.2) le quali "aprono" una classe (i suoi oggetti) alla manipolazione da parte di altre classi (di oggetti di altre classi). Una disponibilità eccessiva di metodi d'accesso potrebbe quindi condurre a una "strumentalizzazione" della classe da parte di altre classi le quali decidono cos'è bene per essa!

Ciò detto, esistono situazioni in cui una classe *deve* aprirsi ad altre classi. Questo accade ogniqualvolta sia necessario implementare una *politica tra due o più classi* (Riel, 1996). Si supponga, ad esempio, di avere due classi `INTEGER` e `REAL` e si voglia implementare una "politica" per la conversione tra numeri interi e reali. In quale delle due classi la politica deve essere implementata? È necessaria una classe `CONVERTITORE` per implementare la politica? In ogni modo, almeno una di queste due classi deve permettere metodi di accesso, diventando *dipendente* verso tale politica.

Esempio 9.1 (Immatricolazioni universitarie)

Nel Capitolo 6, sono state progettate le collaborazioni per una parte del caso Immatricolazioni universitarie (Esempi 6.3 e 6.4 nelle Sezioni 6.2.4.1 e 6.2.4.2). Il modello delle collaborazioni ha fornito una distribuzione relativamente uniforme delle interazioni fra le classi ma non sono state considerate soluzioni alternative.

In questo esempio, si assuma che sia necessario inserire uno studente in un corso offerto. Per fare questo, bisogna cercare i corsi prerequisiti del corso offerto, poi controllare l'informazione accademica dello studente per stabilire se lo studente stesso soddisfi i prerequisiti individuati: a questo punto è possibile decidere sull'inserimento dello studente nel corso offerto.

Supponendo che un messaggio `inserisci()` sia inviato da un oggetto boundary `:FinestraImmatricolazioni` e che tre classi, `CorsoOfferto`, `Corso` e `Studente`, collaborino per portare a termine il compito, si progetti un insieme di possibili diagrammi di collaborazione per risolvere il problema, discutendo i pro e i contro delle differenti soluzioni.

La Figura 9.1 illustra la prima soluzione. L'oggetto boundary `:FinestraImmatricolazioni` inizia la transazione inviando il messaggio `inserisci()` all'oggetto `unCorso` il quale chiede a `unoStudente` l'informazione accademica, la controlla rispetto ai propri prerequisiti e decide se `unoStudente` possa essere inserito; se l'inserimento è possibile, richiede a `unCorsoOfferto` di inserire `unoStudente` nella propria lista di studenti.

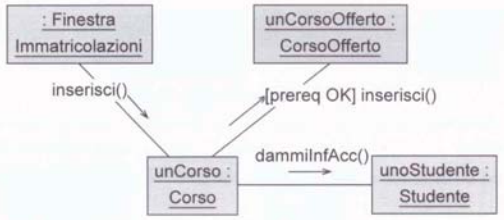


FIGURA 9.1 unCorso come gestore della politica (Immatricolazioni universitarie).

Lo scenario nella Figura 9.1 assegna il completo controllo all'oggetto `unCorso`, la classe `Studente` è dipendente. La soluzione è sbilanciata ma non c'è una via d'uscita elegante.

Si potrebbe spostare l'enfasi da `unCorso` a `unoStudente` ottenendo la soluzione presentata in Figura 9.2. Ora, `:FinestraImmatricolazioni` chiede a `unoStudente` di fare il lavoro principale. `unoStudente` richiede i prerequisiti a `unCorso` con il metodo d'accesso (rinominato) `dammiPrereq()` e decide se l'inserimento sia possibile, richiedendo in tal caso a `unCorsoOfferto` di inserire lo studente.

La Figura 9.3 illustra una soluzione in cui è `unCorsoOfferto` ad agire come un "programma principale" (una classe *predominante* (Riel, 1996)). La soluzione è imparziale rispetto a `unCorso` e `unoStudente` ma rende questi due oggetti dipendenti.

La soluzione può essere migliorata introducendo un oggetto control per gestire l'esecuzione (si veda l'approccio BCE, Sezione 5.2.4). L'*oggetto control* `:PoliticaImmatricolazioni` nella Figura 9.4 disaccoppia le tre classi entity dalla politica d'immatricolazione. Questo è un beneficio poiché ogni modifica alla politica d'immatricolazione è limitata a una singola classe control. Tuttavia, esiste il rischio che la classe `PoliticaImmatricolazioni` possa crescere e diventare una classe *predominante*.

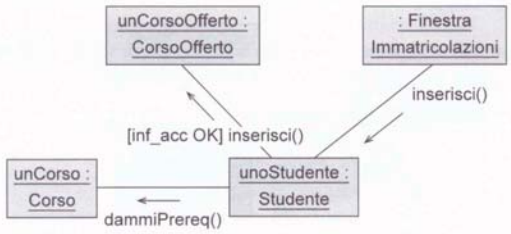


FIGURA 9.2 unoStudente come gestore della politica (Immatricolazioni universitarie).

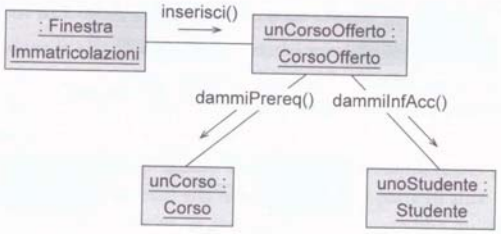


FIGURA 9.3 unCorsoOfferto come gestore della politica (Immatricolazioni universitarie).

9.1.1.4 Classificazione dinamica e classi disomogenee

Nella Sezione 2.1.5.2.3 è stato citato il problema della *classificazione dinamica* e osservato che i comuni ambienti di programmazione object-oriented non offrono, in genere, alcun supporto a questo problema. Tale carenza ha come conseguenza la progettazione di *classi disomogenee* (*mixed-instance cohesion class*).

Page-Jones (2000) stabilisce che "una classe disomogenea ha alcune caratteristiche indefinite per alcuni oggetti della classe". Alcuni metodi e/o alcuni attributi della classe si applicano solo a un sottoinsieme di oggetti della classe stessa. Ad esempio, la classe `Impiegato` potrebbe rappresentare oggetti che sono "normali" impiegati e altri che sono manager. A un manager è pagata una gratifica. Inviare un messaggio `pagareGratifica` a un oggetto di `Impiegato` non ha senso se quell'oggetto di `Impiegato` non è un manager.

Per eliminare la disomogeneità, è necessario estendere la gerarchia di generalizzazione per identificare sottoclassi di `Impiegato` quali `ImpiegatoNormale` e `Manager`. Tuttavia, un oggetto di `Impiegato` potrebbe essere un `ImpiegatoNormale` un giorno e `Manager` un altro giorno o viceversa. Per eliminare la disomogeneità sarebbe necessario permettere agli oggetti di cambiare dinamicamente classe, a tempo d'esecuzione.

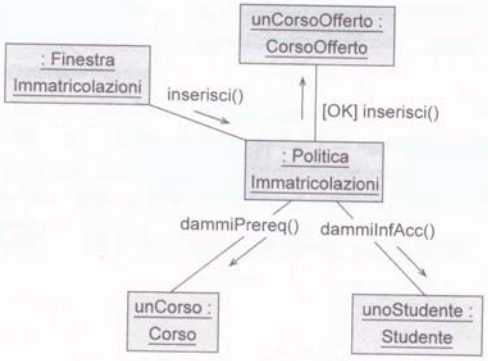


FIGURA 9.4 :PoliticaImmatricolazioni come gestore della politica (Immatricolazioni universitarie).

Esempio 9.2 (Immatricolazioni universitarie)

Si considerino le seguenti variazioni all'Esempio 9.1:

- I corsi offerti serali sono disponibili solo per studenti a tempo parziale.
- Gli studenti a tempo pieno possono essere inseriti solo nei corsi offerti durante il giorno.
- Lo studente a tempo parziale deve versare una minima tassa extra per essere inserito in un corso offerto serale.
- Uno studente a tempo parziale è automaticamente considerato a tempo pieno quando è iscritto in corsi per un totale di più di sei crediti in un dato semestre (e viceversa).

Il compito è proporre un modello delle collaborazioni tra classi omogenee. Il modello dovrebbe essere valutato con spirito critico e una soluzione alternativa, che eviti il problema della classificazione dinamica, dovrebbe essere suggerita e discussa.

Per eliminare la disomogeneità è necessario specializzare `Studente` in due sottoclassi `StudenteTempoParziale` e `StudenteTempoPieno` (Figura 9.5) (se uno studente può solo essere o a tempo parziale o a tempo pieno, la classe `Studente` può essere considerata astratta). Il messaggio `pagareTassaExtra(crs_off)` non sarà mai inviato a un oggetto di classe `StudenteTempoPieno` poiché la classe non ha tale metodo.

C'è ancora un problema. Uno studente a tempo parziale potrebbe preferire alcuni corsi offerti di giorno (cioè `preferenza_serale = "Falso"`) e quindi nessuna tassa extra deve essere pagata. In altre parole, permane ancora una disomogeneità in `StudenteTempoParziale`: inviare il messaggio `pagareTassaExtra(crs_off)` a un oggetto di `StudenteTempoParziale` non ha senso se lo studente sceglie un corso offerto di giorno.

La Figura 9.6 estende il progetto per eliminare questa seconda disomogeneità. La classe `StudenteTempoParzialeGiorno` non ha il metodo `pagareTassaExtra(crs_off)`. Ma cosa accade se un oggetto di `StudenteTempoParzialeGiorno` è obbligato a seguire un corso offerto serale poiché non ci sono più posti disponibili nei corsi offerti di giorno? Si dovrebbe specializzare ulteriormente per derivare una classe `StudenteTempoParzialeGiornoSfortunato`?

A meno di non voler cadere in situazioni ridicole, è meglio scegliere di non continuare a oltran-

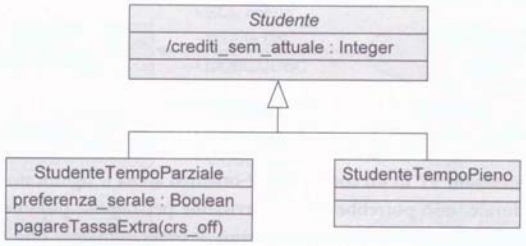


FIGURA 9.5 Collaborazione strutturale per eliminare la disomogeneità (Immatricolazioni universitarie).

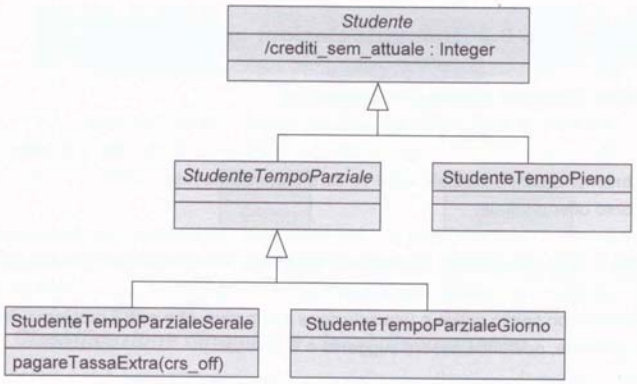


FIGURA 9.6 Collaborazione strutturale per eliminare un altro aspetto della disomogeneità (Immatricolazioni universitarie).

za l'eliminazione della disomogeneità. In realtà, il valore attuale dell'attributo `crediti_sem_attuale` determina se uno studente è a tempo parziale oppure a tempo pieno (più di sei crediti).

In assenza di un supporto per la *classificazione dinamica* offerto dall'ambiente di programmazione, sarebbe responsabilità del programmatore permettere a un oggetto di cambiare classe a tempo d'esecuzione; questo è molto complesso da realizzare nel caso di oggetti persistenti con OID legati in qualche modo agli identificatori di classi.

L'alternativa è di limitare la profondità della gerarchia d'ereditarietà, di eliminare la necessità della classificazione dinamica, mantenendo un certo ammontare di disomogeneità. Ad esempio, è possibile, riferendosi alla collaborazione strutturale in Figura 9.5, risolvere il problema della preferenza serale permettendo a un oggetto di rispondere al messaggio `pagareTassaExtra(crs_off)` in modo diverso secondo il valore dell'attributo `preferenza_serale`. Ciò può essere programmato con un'istruzione `if` come mostra il seguente pseudocodice:

```
method pagareTassaExtra(crs_off) for StudenteTempoParziale
        if preferenza_serale = "Falso"
                return
        else
                "eseguire richiesta pagamento"
end method
```

Sebbene l'uso di istruzioni `if` in un codice object-oriented abbia il significato di un "ritorno" al paradigma procedurale, esso potrebbe essere inevitabile per ragioni puramente pragmatiche. Piuttosto che sforzarsi con la *classificazione dinamica*, il programmatore può introdurre una *semantica dinamica* in un classe: un oggetto risponde in modo diverso allo stesso messaggio, secondo il proprio stato locale attuale. Un diagramma statechart potrebbe essere usato per specificare la dinamica complessa di una tale classe.

9.1.2 Progettare la collaborazione client/server

I programmi del sistema informativo interagiscono con i database usando un linguaggio, tipicamente SQL, per accedere e modificare il database stesso. Per capire come un programma client comunica con il database server, è necessario ricordarsi che SQL ha diversi dialetti e può essere usato in un programma a diversi livelli di astrazione.

La Figura 9.7 distingue cinque livelli di interfaccia SQL. A *Livello 1*, SQL è usato come un linguaggio di definizione dei dati (DDL) per definire le strutture del database (schema del database). Progettisti e amministratori (DBA) del database sono i principali utenti di SQL a Livello 1.

A *Livello 2*, SQL è usato come linguaggio di interrogazione e di manipolazione dei dati (DML). A tale livello, SQL serve per recuperare dati e per modificarli (con le operazioni insert, update, delete).

Un alto numero di utenti usa SQL a Livello 2, dal principiante al DBA esperto. SQL a questo livello è *interattivo*: un utente può formulare un'interrogazione all'esterno di un qualsiasi ambiente di programmazione, eseguendola immediatamente sul database. SQL a Livello 2 è il punto d'ingresso per apprendere un SQL più elaborato ai livelli successivi.

I programmatori usano SQL a livelli superiori al Livello 2. A tali livelli, SQL permette l'elaborazione di *un record alla volta* in aggiunta all'elaborazione di *un insieme alla volta* disponibile (come unica opzione) a Livello 2. L'elaborazione di *un insieme alla volta* prende una o più tabelle (insiemi di record) in ingresso a un'interrogazione e restituisce una tabella in uscita. Sebbene sia una potente funzionalità, può essere difficile da usare per interrogazioni complesse.

La funzionalità di elaborazione di *un record alla volta* fornisce al programmatore la possibilità di scorrere uno per uno i record restituiti dall'interrogazione e decidere cosa fare con ciascuno di tali record; questa funzionalità è chiamata un *cursore* ed è disponibile in SQL a livelli superiori al Livello 2.

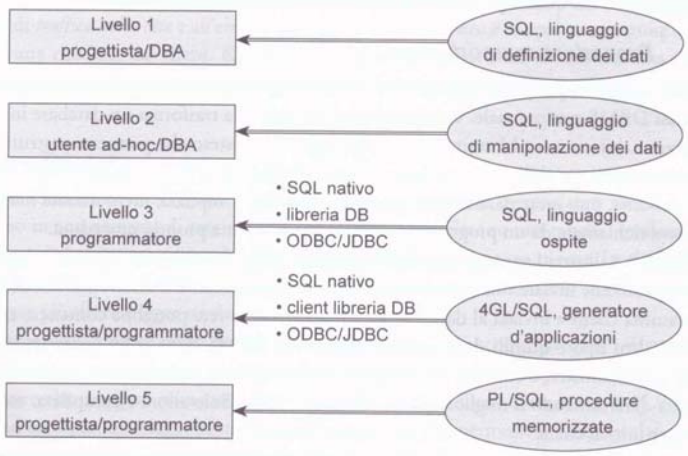


FIGURA 9.7 Interfacce SQL.

SQL a *Livello 3* è ospitato (*embedded*) in un linguaggio di programmazione convenzionale, quale C o COBOL. Poiché un compilatore del linguaggio di programmazione non comprende SQL, è necessario un pre-compilatore (pre-processore) per tradurre le istruzioni SQL in chiamate a funzioni nella libreria DB fornita dal produttore del DBMS (se il programmatore decide di usare direttamente le funzioni della libreria DB, il pre-compilatore non è necessario).

Un modo comune per far comunicare un programma client con il database è attraverso librerie standard quali *Open Database Connectivity* (ODBC) e *Java Database Connectivity* (JDBC). Per programmare in tal modo, è necessario un "driver" software ODBC o JDBC per il particolare DBMS in uso. ODBC e JDBC forniscono un linguaggio standard per database, tradotto dal driver in SQL nativo del DBMS.

ODBC/JDBC hanno il vantaggio di disaccoppiare il programma dal DBMS. Se il programma dovesse, in futuro, migrare su un diverso DBMS, allora una semplice sostituzione del driver dovrebbe essere sufficiente. Più importante è il fatto che l'uso di ODBC/JDBC permette a una singola applicazione di inoltrare interrogazioni su più DBMS distinti.

Lo svantaggio di ODBC/JDBC è che si tratta di un "minimo comune denominatore" per SQL. Un'applicazione client non può trarre vantaggio da alcuna caratteristica o estensione speciale di SQL offerta da un particolare DBMS.

SQL a *Livello 4* usa le stesse strategie di SQL a Livello 3 per inglobare SQL nei programmi client. SQL a Livello 4 fornisce, tuttavia, un ambiente di programmazione più potente quale un generatore d'applicazioni o un linguaggio di quarta generazione (4GL – 4 Generation Language). Un 4GL è generalmente dotato di funzionalità per la costruzione e la gestione di interfacce GUI. Poiché le applicazioni del sistema informativo richiedono interfacce sofisticate, un 4GL/SQL è una scelta frequente per la costruzione di tali applicazioni.

SQL a *Livello 5* complementa i Livelli 3 e 4 fornendo la possibilità di spostare alcune istruzioni SQL dal programma client a un database server programmabile (procedure memorizzate). SQL è usato come un linguaggio di programmazione (PL/SQL) con cui redigere programmi client dall'interno dei quali è possibile richiamare le procedure memorizzate.

9.1.2.1 Procedure memorizzate

Il RDBMS Sybase ha per primo introdotto il concetto di *procedura memorizzata*, ora presente in un qualsiasi DBMS commerciale. Una procedura memorizzata trasforma un database in un sistema programmabile attivo ed è scritta in un linguaggio SQL esteso che permette costrutti di programmazione quali variabili, cicli, selezione e istruzioni d'assegnazione. Una procedura memorizzata ha un nome, può avere parametri in ingresso e uscita, è compilata, memorizzata nel database e può essere richiamata da un programma client come fosse una propria procedura.

La Figura 9.8 illustra i vantaggi per un programma client che richiama una procedura memorizzata piuttosto che inviare una completa interrogazione al server. Un'interrogazione costruita in un programma client e inviata al database server attraverso la rete potrebbe contenere errori sintattici o di altro tipo e quindi deve essere verificata dal DBMS. Una volta verificata, il DBMS controlla se il chiamante è autorizzato a eseguire l'interrogazione, nel qual caso l'interrogazione è ottimizzata determinando il migliore piano d'accesso ai dati. Solo allora è compilata, eseguita e i risultati restituiti al client.

Se, invece, un'interrogazione (o un insieme d'interrogazioni) è scritta come procedura memorizzata, allora è ottimizzata e compilata in modo definitivo, e memorizzata nel database server.

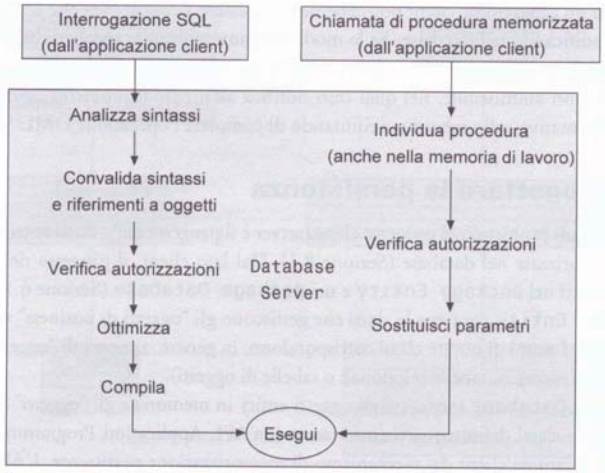


FIGURA 9.8 Confronto tra richiamo da un client SQL e richiamo di procedura memorizzata.

Un programma client, invia una breve chiamata con il nome della procedura e una lista di parametri attuali. Se è fortunato, la procedura potrebbe risiedere nella memoria di lavoro del DBMS, altrimenti, sarà portata in memoria dal database. L'autorizzazione dell'utente è controllata come nel caso di un'interrogazione SQL, i parametri attuali sostituiscono i corrispondenti parametri formali, la procedura memorizzata è eseguita e, infine, i risultati sono restituiti al chiamante.

Come visto nello scenario precedente, una procedura memorizzata fornisce modi molto efficienti di accesso al database da un programma client. I vantaggi prestazionali sono dovuti al risparmio di *traffico nella rete* e all'eliminazione di *analisi sintattica e compilazione* ogniqualvolta sia ricevuta una richiesta dal client. Riveste anche maggiore importanza il fatto che una procedura memorizzata è *mantenuta in un unico posto* e può, quindi, essere chiamata da molti programmi client.

9.1.2.2 Trigger

Un *trigger* (Sezione 8.4.1.4) è una speciale tipologia di procedura memorizzata eseguita in modo autonomo in conseguenza di eventi come insert, update o delete su una tabella. In alcuni sistemi (ad esempio Sybase) ciò implica che ogni tabella possa avere fino a tre trigger. In altri sistemi (ad esempio Oracle), sono identificate varianti aggiuntive di eventi possibili: ciò conduce alla possibilità di avere più di tre trigger su ciascuna tabella.

Il trigger, oltre a essere usato per la verifica procedurale di vincoli d'integrità referenziale (vedi Sezione 8.4.1.4), può essere programmato per verificare una *qualsiasi regola di business* valida per il database e che non può essere violata da alcun programma client o istruzione SQL/DML interattiva (ad esempio, un trigger potrebbe essere scritto per non permettere l'accesso al database durante alcuni periodi o giorni dell'anno).

L'utente di un programma client potrebbe neppure essere a conoscenza che il trigger "osserva" cosa si sta modificando nel database. Se le modifiche non violano le regole di business, i trigger non sono visibili ai programmi. Un trigger si manifesta all'utente quando questi emette un comando DML non ammissibile, nel qual caso notifica all'utente il problema, visualizzando un messaggio informativo sullo schermo e rifiutando di compiere l'operazione DML.

9.1.3 Progettare la persistenza

Uno dei principali problemi nel progetto client/server è il progetto delle classi le cui istanze devono essere memorizzate nel database (Sezione 8.1). Dal lato client, il progetto della persistenza coinvolge le classi nel `package Entity` e nel `package Database` (Sezione 6.1.3.2).

Il `package Entity` contiene le classi che gestiscono gli "oggetti di business" nella memoria del programma. I nomi di queste classi corrispondono, in genere, ai nomi di "oggetti" persistenti nel database (ad esempio, tabelle relazionali o tabelle di oggetti).

Il `package Database` media tra gli oggetti entity in memoria e gli "oggetti" persistenti nel database. Le sue classi definiscono un'interfaccia (un'API, Application Programming Interface) che isola il programma client dal meccanismo di memorizzazione persistente. L'API può fornire un accesso indipendente dal database agli "oggetti" persistenti, incapsulando il linguaggio di programmazione del database, ad esempio SQL. Essa è una libreria di classi che può essere acquistata, ad esempio, dai fornitori di database a oggetti (vedi Sezione 8.2).

Il progetto della persistenza richiede un insieme di classi nel `package Database` che permettano di: stabilire la connessione al database, leggere (selezionare) record dal database, modificare (inserire, cancellare, aggiornare) record nel database. Tali classi possono essere indicate rispettivamente come: `D_Connetti`, `D_Leggi`, `D_Aggiorna`.

Ogni oggetto entity `:E_ClasseEntity` avrà i corrispondenti oggetti `:D_Leggi` e `:D_Aggiorna` per interagire con il database. Avrà inoltre, un corrispondente oggetto `:C_Visualizza` per visualizzare il proprio stato in una finestra GUI.

9.1.3.1 Collaborazione strutturale per la persistenza

Il progetto della persistenza deve comprendere i modelli di collaborazione strutturale e comportamentale (Sezione 6.2.4). La Figura 9.9 mostra un modello di collaborazione strutturale per la persistenza con un modello relazionale.

Il modello mostra classi generiche, ma, nella realtà, saranno usate versioni specializzate della maggior parte di queste classi. Inoltre, il modello è incompleto: rappresenta solo le classi che collaborano per selezionare dati da una tabella e visualizzare i risultati in una finestra browser a riga.

Il modello usa frecce tratteggiate per indicare le *relazioni d'istanziazione*. Tali relazioni implementano il *paradigma Produttore/Prodotto* per creare oggetti (SourcePro, 2001). Il paradigma usa gli oggetti di una classe (`Produttore`) per istanziare oggetti di un'altra classe (`Prodotto`). Invece di invocare costruttori pubblici, i produttori usano costruttori privati. Quindi, le relazioni d'istanziazione implementano i *legami d'associazione* a tempo d'esecuzione tra produttori e prodotti.

Il programma client deve stabilire una connessione al database prima di poter accedere a un qualsiasi oggetto persistente (la durata della connessione è normalmente pari alla durata dell'esecuzione del programma). Tale connessione è ottenuta dal `D_DriverMediatore`, usando l'indirizzo del database (URL), il nome e la parola chiave dell'utente. `D_DriverMediatore` stabilisce la connessione caricando un driver appropriato dall'insieme di driver ODBC/JDBC registrati (oppure nativi del database) e quindi istanzia `D_Sessione` e `D_ListaTabelle`.

`C_Richiesta` rappresenta un evento client che richiede un'interazione con un oggetto del database. L'associazione con `D_Sessione` assicura che tutte le richieste al database usino la medesima connessione (minimizzando il sovraccarico associato all'apertura e chiusura di molte connessioni).

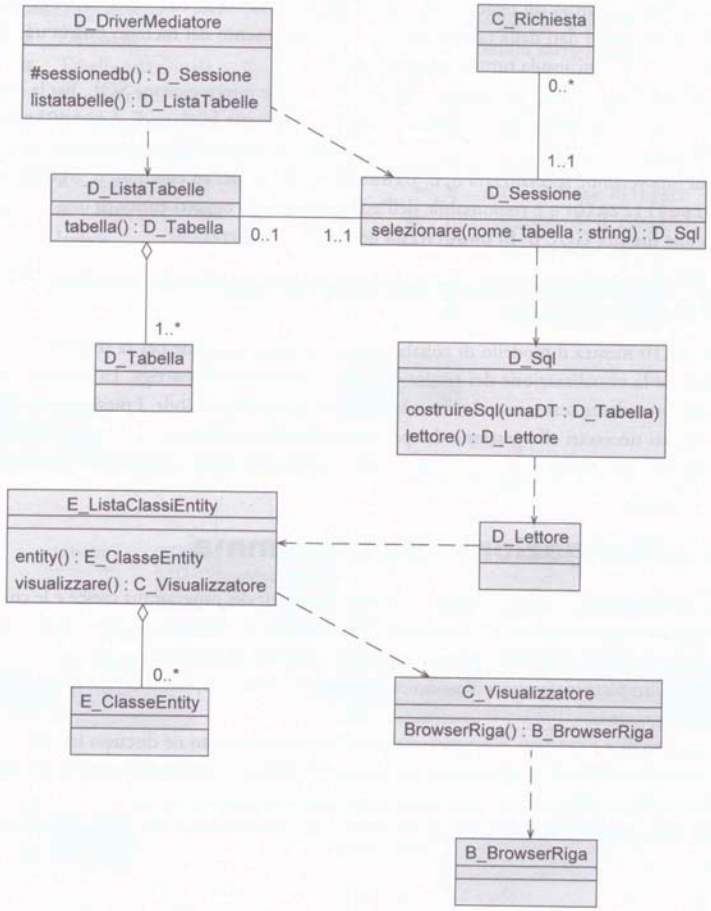


FIGURA 9.9 Collaborazione strutturale per la persistenza.

`D_Sessione` istanzia un oggetto `:D_Sql` il quale, a sua volta, costruisce un'interrogazione SQL per selezionare i dati dalla tabella (fornita come argomento del metodo `costruireSql`). In generale, `D_Sql` incapsula tutte le istruzioni SQL.

`D_Lettore` legge l'insieme risultante da una qualunque interrogazione SQL. Per la selezione di righe da un'unica tabella del database, `D_Lettore` istanzia la classe `E_ListaClassiEntity`. Si tratta di una classe parametrizzata che contiene le istanze di `E_ClasseEntity` (cioè le istanze di una versione specializzata di `E_ClasseEntity` quale, ad esempio, `E_Studente`).

`C_Visualizzatore` è responsabile dell'assegnazione agli oggetti entity di una veste grafica adatta a una finestra GUI. `B_BrowserRiga` gestisce la visualizzazione.

9.1.3.2 Collaborazione comportamentale per stabilire la sessione con il database

La Figura 9.10 mostra il modello di collaborazione comportamentale per la selezione di dati da una tabella e la visualizzazione dei risultati in una finestra browser a riga. La spiegazione delle classi nella precedente sezione rende il modello facilmente comprensibile. I messaggi `new()` creano gli oggetti necessari alla gestione del paradigma Produttore/Prodotto. L'oggetto etichettato `main` che crea la richiesta può essere considerato l'equivalente di un "programma principale".

9.2 Navigazione di programma

In presenza di database programmabili, il numero di oggetti del programma cresce e le collaborazioni tra essi diventano sempre più complesse. I diagrammi di navigazione nelle finestre (Sezione 7.6.2) non sono più sufficienti come documenti di progetto, da cui un programmatore possa intraprendere l'implementazione, e devono essere estesi nei più completi diagrammi per la navigazione di programma.

Il concetto di *navigazione di programma* non è né standardizzato né discusso in UML. Nonostante ciò, si tratta di una fondamentale astrazione richiesta per colmare il vuoto tra il progetto e l'implementazione del sistema. L'alternativa è una cattiva ingegneria del software, in cui l'*architettura* e la *logica* del programma non sono documentate e sono lasciate alle decisioni del programmatore.

9.2.1 Stereotipare il diagramma d'attività per la navigazione di programma

Per trasformare i diagrammi di navigazione nelle finestre in diagrammi per la navigazione di programma, è necessario aggiungere stereotipi lato server ai diagrammi d'attività UML. Sia gli *stati* (rettangoli arrotondati) sia le *attività* (ovali) dovrebbero essere stereotipati. Gli stereotipi devono prendere in considerazione le caratteristiche del modello del DBMS o anche di un particolare DBMS (questa è, ovviamente, la principale ragione, perché i diagrammi di navigazione non sono previsti da UML).

Il seguente elenco di stereotipi può essere usato per progettare navigazioni con un RDBMS. In funzione del livello d'astrazione al quale il diagramma per la navigazione di programma è costruito, l'elenco può contrarsi o estendersi con oggetti aggiuntivi:

- *Stati* (*oggetti dato*):
  - Database
  - Tabella relazionale
    - Colonna
    - Riga (record)
  - Vista relazionale
    - Colonna
    - Riga (record)
  - Indice
  - Cluster
- *Attività* (*oggetti programma*):
  - Procedura memorizzata
  - Trigger
    - On Insert
    - On Update
    - On Delete
    - Altre tipologie di trigger
  - Interrogazione SQL client
    - Interrogazione nativa
    - Interrogazione libreria DB
    - Interrogazione ODBC/JDBC

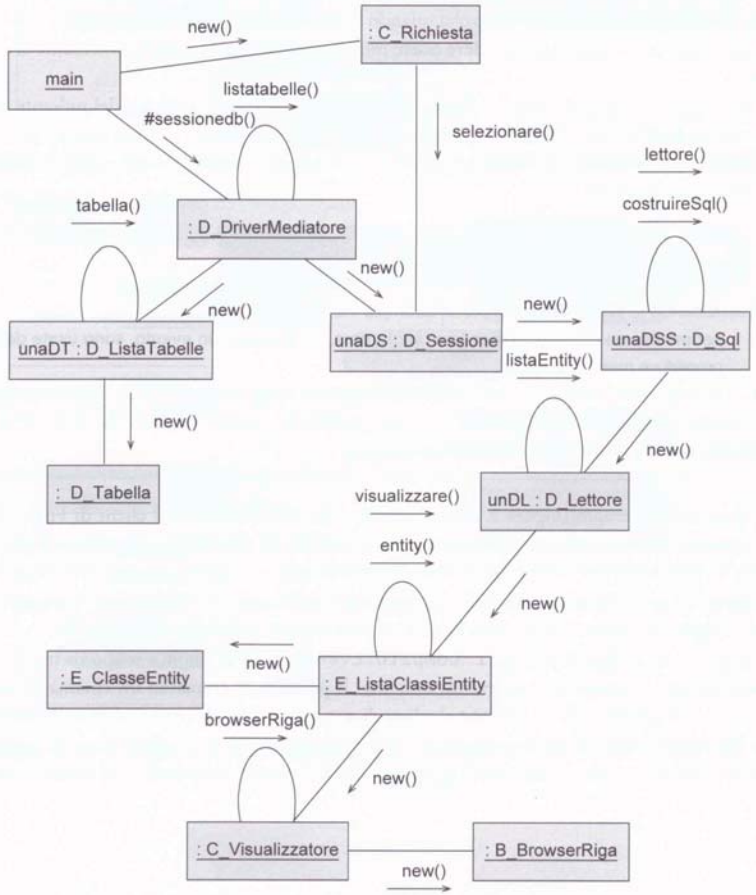


FIGURA 9.10 Collaborazione comportamentale per la persistenza.

9.2.2 Diagramma per la navigazione di programma

Per illustrare i diagrammi per la navigazione di programma saranno estesi gli esempi dei diagrammi di navigazione nelle finestre presentati nella Sezione 7.6.2. La Figura 9.11 è il diagramma per la navigazione di programma che estende (ristrutturandolo leggermente) il diagramma di navigazione nelle finestre in Figura 7.23.

La Figura 9.11 introduce due *stati server*: `Controllo Magazzino` (`<<database>>`) e `Prodotto` (`<<tabella>>`), e tre *attività server*: `Scegliere Prodotto` (`<<procedura memorizzata>>`), `Aggiornare Prodotto` (`<<procedura memorizzata>>`), `In Aggiornamento` (`<<trigger>>`); è aggiunta inoltre l'attività client `Scorri` (`<<barra scorrimento/tastiera>>`). Con tali elementi aggiuntivi, il diagramma per la navigazione di programma in Figura 9.11 indica al programmatore che per visualizzare prodotti nel `Browser Prodotti`, è necessario chiamare una procedura memorizzata (`Scegliere Prodotto`). La stessa procedura memorizzata è coinvolta quando l'utente scorre verso il basso o verso l'alto la finestra nel browser, il cui contenuto deve essere modificato attingendo conseguentemente al database.

Nella parte inferiore del diagramma, è possibile osservare che la pressione del pulsante `OK` o `Salva` in `Aggiorna Prodotto` (`<<finestra di dialogo>>`) invoca `Aggiornare Prodotto` (`<<procedura memorizzata>>`). Un trigger controlla il risultato della modifica (`In Aggiornamento`).

Esempio 9.3 (CMS)

Si considerino le seguenti estensioni all'Esempio 7.5 (Sezione 7.6.2):

- per modificare (inserire, cancellare, aggiornare, completare) un evento, sono usate delle procedure memorizzate,
- sono presenti dei trigger per monitorare inserimenti e aggiornamenti sulla tabella Evento.

Il compito è progettare un diagramma per la navigazione di programma in questa parte dell'applicazione CMS onde trattare le modifiche di eventi.

La soluzione all'esempio (Figura 9.12) comprende solo stati e attività del client di Figura 7.24 che risultano rilevanti per la modifica di eventi. Si noti che la procedura memorizzata `Completare Evento` può essere invocata sia dalla finestra primaria sia da una finestra di dialogo. Poiché `Completare Evento` aggiorna la tabella `Evento`, il trigger `In Modifica` è eseguito. Lo stesso trigger può essere attivato dalla procedura memorizzata `Salvare Evento`.

La casella di dialogo `Dettagli Compito/Evento` serve al duplice scopo di inserire un nuovo evento e di aggiornare un evento esistente. Il pulsante di comando `OK` chiama la procedura memorizzata `Salvare Evento`. La lista di parametri nella chiamata informa la procedura se `OK` significa inserire oppure aggiornare. Corrispondentemente, l'esecuzione della procedura manda in esecuzione uno dei due trigger: `In Inserimento Evento` e `In Modifica`.

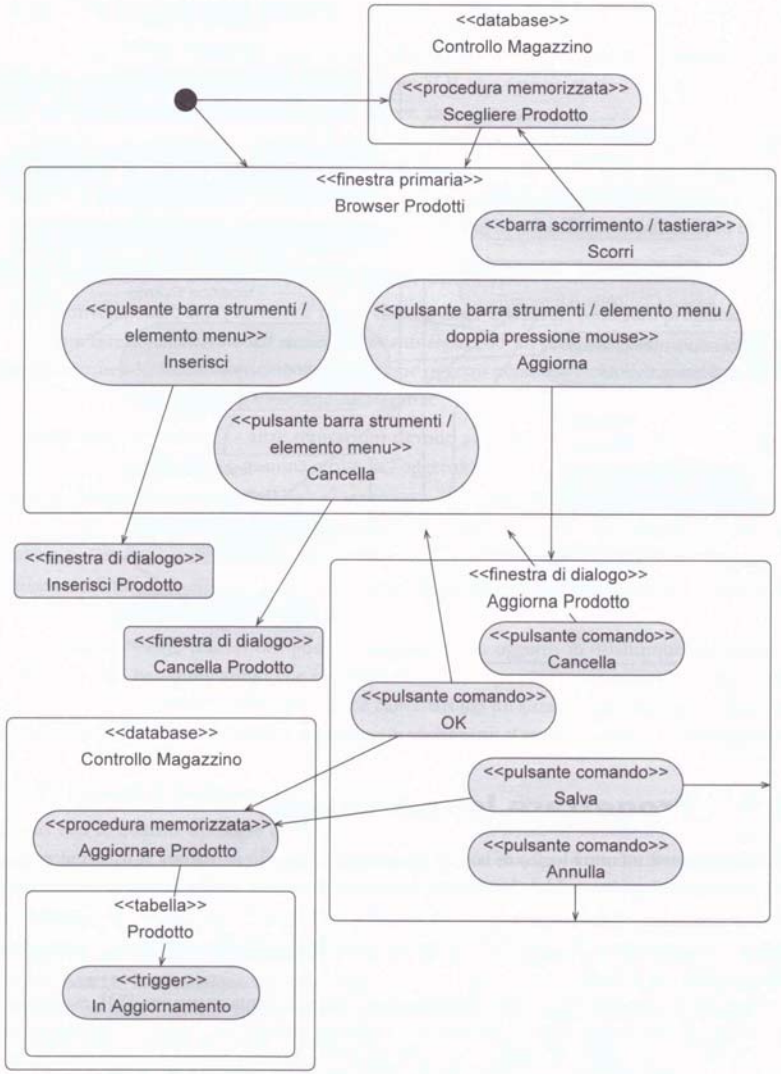


FIGURA 9.11 Diagramma per la navigazione di programma.

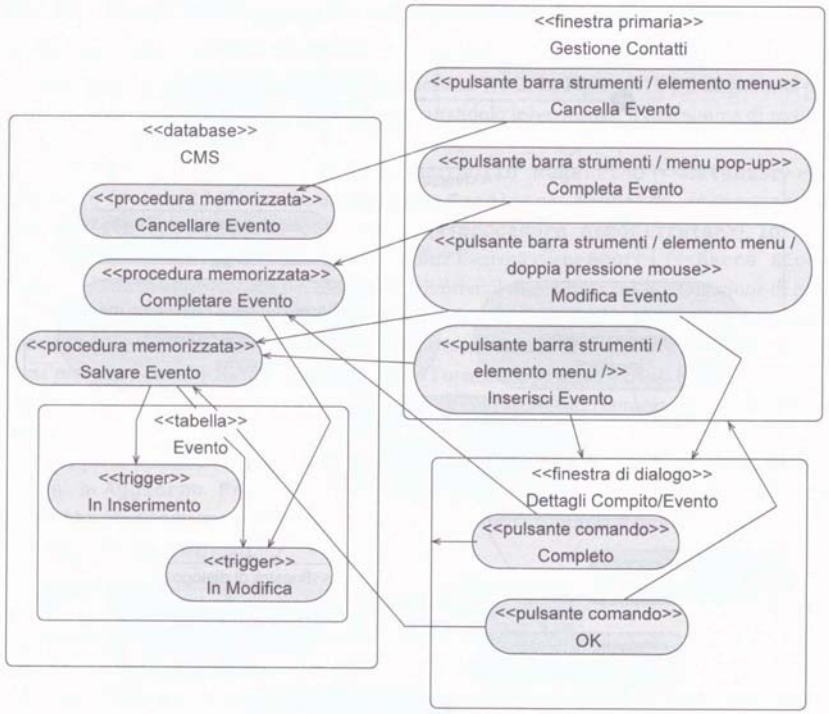


FIGURA 9.12 Diagramma per la navigazione di programma (CMS).

9.3 Progettare le transazioni

Una *transazione* è un'unità logica di lavoro, contenente una o più istruzioni SQL, la cui esecuzione è richiesta da un utente. Una transazione è un'unità di *consistenza del database*: lo stato del database è consistente dopo che la transazione è stata completata. Per assicurare tale consistenza, un gestore delle transazioni di un DBMS deve assicurare due servizi fondamentali: il *ripristino* e il *controllo della concorrenza*.

Seguendo lo standard SQL, una transazione inizia con la prima istruzione SQL eseguibile (in alcuni sistemi, può essere richiesta un'esplicita istruzione, ad esempio, `begin transaction`) e termina con un'istruzione di `commit` o `rollback`. L'istruzione `commit` memorizza in modo persistente le modifiche nel database. L'istruzione `rollback` annulla qualsiasi modifica fatta dalla transazione.

La transazione deve essere *atomica*: i risultati di tutte le istruzioni SQL nella transazione sono accettati interamente oppure altrettanto interamente rifiutati (con `rollback`). A seconda delle necessità del dominio applicativo e dello stile d'interazione uomo-macchina scelto, una transazione può essere breve come un'istruzione singola SQL oppure può coinvolgere una serie di istruzioni SQL.

9.3.1 Transazione breve

Le applicazioni più convenzionali in un sistema informativo richiedono spesso *transazioni brevi*. Una transazione breve contiene una o più istruzioni SQL che devono essere completate il più velocemente possibile, in modo da non interferire con altre transazioni.

Si consideri, ad esempio, un sistema di prenotazioni aeree con il quale molti agenti di viaggio fanno prenotazioni voli per viaggiatori in tutto il mondo. È essenziale che ogni transazione di prenotazione sia eseguita velocemente dal DBMS, in modo che la disponibilità dei posti di un volo sia aggiornata e il database sia pronto per elaborare la successiva transazione che attende in coda.

9.3.1.1 Controllo pessimistico della concorrenza

I DBMS convenzionali, con l'eccezione degna di nota degli ODBMS, sono stati progettati avendo in mente transazioni brevi. Tali sistemi lavorano seguendo un *controllo pessimistico della concorrenza*, acquisendo *blocchi* (lock) su un qualunque oggetto persistente che la transazione elabora. Esistono quattro tipologie di blocco su un oggetto:

1. *blocco esclusivo* (*scrittura*) - altre transazioni devono attendere fino a quando la transazione che detiene un tale blocco termina e rilascia l'oggetto;
2. *blocco in aggiornamento* (*intenzione di scrittura*) - altre transazioni possono leggere l'oggetto ma la transazione che detiene il blocco può farlo diventare un blocco esclusivo, quando ne abbia necessità;
3. *blocco in lettura* (*condiviso*) - altre transazioni possono leggere ed eventualmente ottenere un blocco di aggiornamento sull'oggetto;
4. *nessun blocco* - altre transazioni possono aggiornare un oggetto in qualunque momento; è possibile solo per applicazioni che permettono "letture sporche", cioè una transazione legge dati che possono essere modificati o anche cancellati (da un'altra transazione) prima della sua conclusione.

9.3.1.2 Livelli di isolamento

Associati con le quattro tipologie di blocco, vi sono quattro *livelli di isolamento* tra transazioni concorrentemente in esecuzione. È responsabilità del progettista di sistema decidere quale livello di isolamento sia appropriato per l'insieme di transazioni sul database. I quattro livelli sono (Khoshafian *et al.*, 1992):

1. *Lettura sporca possibile* - la transazione `t1` ha modificato un oggetto ma non ha ancora eseguito il `commit`; la transazione `t2` legge l'oggetto; se `t1` esegue un `rollback`, la transazione `t2` ha letto un'informazione "sporca" che, in un certo senso, non è mai esistita nel database.
2. *Lettura non ripetibile possibile* - `t1` legge un oggetto; `t2` aggiorna l'oggetto; `t1` legge ancora lo stesso oggetto ma a questo punto ottiene un diverso valore per lo stesso oggetto.
3. *Lettura fantasma possibile* - `t1` legge un certo insieme di oggetti (ad esempio, tutti quelli associati a un oggetto iniziale); `t2` inserisce un nuovo oggetto nell'insieme; `t1` ripete l'operazione di lettura e vede un oggetto "fantasma" (apparso fra le due letture e che potrebbe scomparire se `t2` esegue un `rollback`).
4. *Lettura ripetibile* - `t1` e `t2` sono ancora eseguite concorrentemente ma la loro esecuzione interlacciata produce i medesimi risultati come se le due transazioni fossero state eseguite una per volta (questa è chiamata un'*esecuzione serializzabile*).

Le tipiche applicazioni in un sistema informativo (interattive e basate su interfacce GUI) richiedono transazioni brevi. Il livello di isolamento, tuttavia, potrebbe essere diverso per transazioni distinte pur nella medesima applicazione. L'istruzione SQL `set transaction` può essere usata a tale proposito. Il compromesso è ovvio, un incremento del livello d'isolamento riduce la concorrenza globale del sistema.

Una decisione cruciale di progetto è, tuttavia, indipendente dalle precedenti considerazioni: l'inizio di una transazione deve essere sempre ritardato fino all'ultimo istante. È infatti inaccettabile far partire una transazione da una finestra client e, poi, metterla in attesa fino a quando ottiene qualche informazione ulteriore dall'utente per completare il suo lavoro (l'utente potrebbe essere molto lento nel fornire l'informazione o potrebbe decidere di spegnere il computer mentre la transazione è in attesa). Il *timeout della transazione* alla fine eseguirà un `rollback` della transazione stessa ma il danno alla prestazione globale di sistema è stato fatto.

9.3.1.3 Ripristino automatico

La Legge di Murphy recita, più o meno, che se una cosa può andare male allora andrà male certamente. I programmi possono contenere errori, i processi in esecuzione possono eseguire operazioni non permesse o essere interrotti, la corrente può mancare, una testina di lettura del disco può rompersi, etc. Fortunatamente, ogni DBMS fornisce una funzione di *ripristino* (*recovery*) automatico per la maggior parte di tali situazioni, partendo dalle copie dei dati prima e dopo gli aggiornamenti, copie che sono conservate nel *giornale* (*log*) di sistema; solo nel caso di una perdita fisica dei dati su disco è necessario l'intervento del DBA, al fine di indicare al DBMS come recuperare il database dall'ultima sua *copia di sicurezza* (*backup*).

Il ripristino è automatico ma un DBA può controllarlo definendo la frequenza dei *punti di allineamento* (*checkpoint*). Un checkpoint impone al DBMS l'arresto temporaneo di tutte le transazioni e la scrittura nel database di tutte le modifiche acquisite fino a quel momento (a partire dall'ultimo checkpoint).

La Figura 9.13 illustra i problemi coinvolti nel ripristino automatico dai guasti (Kirkwood, 1992). La transazione `t1` ha eseguito il `commit` dopo il checkpoint ma prima del guasto di sistema. Poiché, generalmente, la scrittura fisica avviene quando sono necessarie le aree di lavoro (buffer) in memoria centrale, il DBMS non sa se tutte le modifiche dopo il checkpoint siano state fisicamente scritte nel database, per cui eseguirà la ri-scrittura (*redo*) dei dati aggiornati dalla transazione `t1` partendo dalle copie presenti nel giornale.

Sulla transazione `t2` è stato eseguito un `rollback` tra il checkpoint e il guasto. Come nel caso della transazione `t1`, il DBMS non sa se le modifiche imposte dal `rollback` siano state effettivamente eseguite sul disco: perciò, il DBMS esegue ancora il `rollback`.

Le altre transazioni hanno iniziato la propria esecuzione dopo il checkpoint. Per la transazione `t3` si esegue una ri-scrittura per garantire che le modifiche a essa relative diventino permanenti nel database. Analogamente, per la transazione `t4` sarà riconfermato il `rollback`.

La transazione `t5` potrebbe non richiedere alcuna azione di ripristino da parte del DBMS poiché essa era in esecuzione al momento del guasto. Qualunque modifica fatta da `t5` prima del guasto potrebbe infatti non essere stata scritta nel database (ma solo nel giornale). L'utente è informato che la transazione era in esecuzione al momento del guasto e sarà fatta ripartire.

9.3.1.4 Ripristino programmabile

Mentre i guasti inattesi sono automaticamente recuperati dal DBMS, i progettisti e i programmatori dovrebbero controllare qualunque problema sorga durante l'esecuzione delle transazioni. Un DBMS fornisce uno spettro di opzioni per il `rollback`, da applicarsi al programma, affinché sia sempre possibile un ripristino controllato (eseguito sempre in modo trasparente all'utente).

Per iniziare, le linee guida GUI, come la "tolleranza" o "l'utente ha il controllo" (Sezione 7.3), richiedono che un programma permetta all'utente di commettere errori e di recuperarli. Un `rollback` controllato dal programmatore, applicato nel punto giusto del programma, può riportare il database allo stato iniziale, assumendo, naturalmente, che la transazione non abbia eseguito il `commit`.

Se la transazione ha eseguito il `commit`, il programmatore ha ancora la possibilità di scrivere una *transazione di compensazione*: l'utente può richiedere l'esecuzione della transazione di compensazione per annullare le modifiche al database. Le transazioni di compensazione sono progettate specificamente per permettere il ripristino programmabile e dovrebbero essere rappresentate nei casi d'uso.

9.3.1.4.1 Punto di salvataggio

Un *punto di salvataggio* (savepoint) è un'istruzione che suddivide in parti distinte una transazione "lunga". I punti di salvataggio sono inseriti in punti strategici del programma e il programmatore ha la possibilità di far eseguire un `rollback` del lavoro fino ad un certo punto di salvataggio piuttosto che dall'inizio della transazione.

Ad esempio, un programmatore potrebbe inserire un punto di salvataggio appena prima di un'operazione d'aggiornamento `update`. Se l'`update` fallisce, il programma esegue un `rollback` a quel punto di salvataggio e può tentare di eseguire ancora l'aggiornamento o, in alternativa, una qualsiasi altra azione per evitare di interrompere l'intera transazione.

In grandi programmi, i punti di salvataggio possono essere posti prima di ogni sottoprocedura. Se una sottoprocedura fallisce, sarebbe possibile eseguire un `rollback` all'inizio di quella sottoprocedura e rieseguirla con i parametri rivisti o, se necessario, eseguire una sottoprocedura di ripristino.

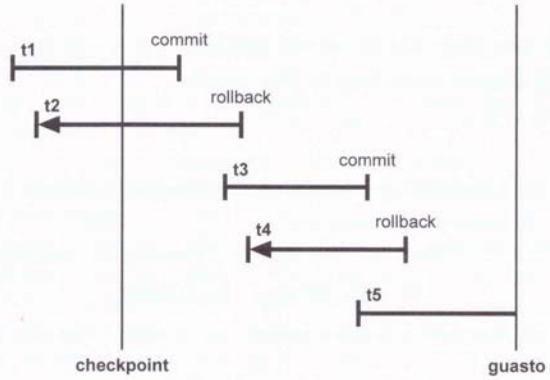


FIGURA 9.13 Ripristino automatico.

9.3.1.4.2 Trigger di rollback

Un *trigger di rollback* è una speciale tipologia di punto di salvataggio. Come spiegato nella Sezione 9.1.2.2, un trigger può essere usato per programmare una regola di business comunque complessa. Talvolta può essere indesiderabile eseguire il `rollback` dell'intera transazione quando il trigger rifiuta la modifica di una tabella (in seguito al tentativo della transazione di violare la regola di business); invece, è preferibile eseguire un'azione di recupero fornendo la possibilità di programmare il trigger in modo da procedere al `rollback` dell'intera transazione oppure del solo trigger. Nel secondo caso, il programma (eventualmente una procedura memorizzata) può analizzare il problema e decidere sulla prossima azione (anche se il `rollback` è eseguito per l'intera transazione, il programma potrebbe analizzare la causa dell'errore e visualizzare un messaggio più completo per l'utente).

9.3.1.5 Progettare procedure memorizzate e trigger

Un diagramma per la navigazione di programma (Sezione 9.2.2) potrebbe essere ulteriormente esteso per rappresentare anche *stati* della transazione. Questo potrebbe provocare problemi troppo complessi da un punto di vista grafico: una soluzione alternativa consisterebbe nell'introdurre diversi diagrammi di navigazione, ciascuno orientato alla presentazione di una diversa prospettiva di navigazione.

In ogni caso, i diagrammi per la navigazione di programma devono identificare trigger e procedure memorizzate, e quindi fornire, per ciascuno di questi oggetti, l'obiettivo, la specifica e il progetto dettagliato. In particolare, una notazione simile a uno pseudocodice potrebbe essere usata per definire gli algoritmi.

Ad esempio, nel seguito è presentato un algoritmo per la procedura memorizzata `Cancellare Evento` nell'applicazione CMS (si veda Figura 9.12, Sezione 9.2.2). La procedura controlla se l'utente (impiegato) che tenta la cancellazione è diverso da quello che ha creato l'evento: in tal caso, l'operazione di cancellazione è rifiutata, altrimenti la cancellazione è accettata e la procedura controlla se l'evento è l'ultimo rimanente per il compito, cancellando, se tale condizione è soddisfatta, il compito stesso.

```
BEGIN
INPUT PARAMETERS (@id_evento, @id_utente)
Select Evento (where id_evento = @id_evento)
IF @id_utente = Evento.id_imp_creato
      THEN
            delete Evento (where id_evento = @id_evento)
            IF non più eventi per
                  Compito.id_compito = Evento.id_compito AND
                  Evento.id_evento = @id_evento
```



```
            THEN
                  eliminare il Compito
            ENDIF
      ELSE
            comunica errore ("Solo il creatore dell'evento può elimi-
narlo")
ENDIF
END
```

La procedura memorizzata `Cancellare Evento` contiene le istruzioni per cancellare i record opportuni dalle tabelle `Evento` e `Compito`. Tali istruzioni potrebbero mandare in esecuzione i trigger definiti su queste tabelle. Se gli algoritmi di questi trigger, oltre al normale controllo dell'integrità referenziale, prevedono operazioni più complesse, il progettista dovrebbe fornire anche per esse la specifica in pseudocodice (compresa, ad esempio, la decisione sulla strategia di `rollback`).

9.3.2 Transazione lunga

Alcune nuove classi di applicazioni incoraggiano la cooperazione tra gli utenti. Tali applicazioni sono conosciute come applicazioni di supporto al *lavoro cooperativo* (CSCW, Computer-Supported Cooperative Work). I possibili esempi includono molte applicazioni d'ufficio, la redazione collettiva di documenti, la progettazione assistita, gli strumenti CASE, etc.

Sotto molti punti di vista, le applicazioni CSCW hanno requisiti per il database "ortogonali" a quelli che caratterizzano le applicazioni tradizionali (basate su transazioni brevi che isolano un utente dall'altro). Le applicazioni CSCW richiedono transazioni lunghe, gestione delle versioni, controllo della concorrenza durante la collaborazione, etc.

Il modello ODB fornisce un'ossatura per il CSCW e molti prodotti ODBMS sono sviluppati per questo dominio applicativo. Gli utenti in un'applicazione CSCW condividono informazioni e sono informati che il loro lavoro tratta dati condivisi. Quindi lavorano in un proprio *spazio di lavoro* (workspace) usando database locali i quali contengono dati *copiati* (con l'operazione *check-out*) dal database comune del gruppo; il lavoro si svolge per *transazioni lunghe* che possono estendersi su diverse sessioni al computer (gli utenti possono fare delle pause e quindi continuare al loro ritorno nella medesima transazione).

L'aspetto dominante di una transazione lunga è che non è permessa l'esecuzione automatica di `rollback` senza la tracciatura della causa del guasto (per apprezzare tale requisito, s'immagini il disappunto dell'autore se questo libro fosse cancellato a causa del guasto del computer). Il `rollback` di una transazione lunga è controllato dall'utente attraverso *punti di salvataggio* che memorizzano persistentemente (con l'operazione *check-in*) gli oggetti nel database privato dell'utente.

La nozione di *transazione breve* non è totalmente eliminata in un'applicazione CSCW. Le transazioni brevi sono ancora necessarie per garantire atomicità e isolamento *durante* le operazioni di check-out e check-in tra il database del gruppo e i database privati. *Blocchi brevi* sono in seguito rilasciati e *blocchi persistenti lunghi* sono imposti dal database del gruppo su tutti gli oggetti su cui è stato eseguito un check-out.

Gli obiettivi di un modello per transazioni lunghe comprendono (Hawryszkiewycz *et al.*, 1994; Maciaszek, 1998):

- lo scambio d'informazioni (anche se temporaneamente inconsistente) tra gli utenti cooperanti;
- la scoperta delle inconsistenze fra i dati e la mediazione delle possibili soluzioni;
- l'utilizzo delle versioni degli oggetti per fornire una condivisione controllata, senza perdita di lavoro in caso di guasti del sistema.

9.4 Round-trip engineering

Il processo iterativo e incrementale (Sezione 1.1.3.1) nella moderna produzione del software richiede un forte supporto per lo sviluppo ciclico tra il progetto e l'implementazione (*round-trip engineering*). La round-trip engineering è definita come l'applicazione ciclica di generazione del codice a partire dai modelli di progetto (*forward-engineering*) e di *reverse-engineering* dal codice ai modelli; si ottiene in tal modo "...la possibilità di lavorare su viste sia grafiche sia testuali, mantenendo consistenti le due viste". (Booch *et al.*, 1999)

Nelle applicazioni client/server, la round-trip engineering è applicata separatamente ai programmi applicativi *client* e ai programmi database *server*. È possibile che gli strumenti CASE usati per il progetto del client e il progetto del server siano differenti, giacché devono essere strettamente integrati con gli specifici ambienti di programmazione client e server.

9.4.1 Round-trip engineering per i programmi client

I principi di *round-trip engineering per applicazioni client* sono relativamente semplici. La Figura 9.14 mostra un diagramma d'attività per un tipico ciclo di round-trip engineering con un linguaggio di programmazione a oggetti (Paradigm, 1997; Rational, 1998). I modelli UML sono memorizzati nel repository dello strumento CASE (come accennato in precedenza, il repository è un database, spesso a oggetti).

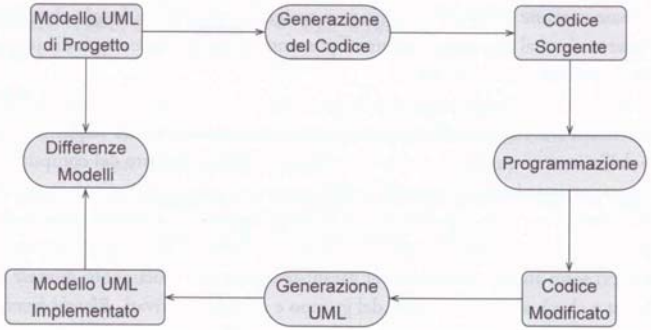


FIGURA 9.14 Round-trip engineering con un programma client.

Il `Modello UML di Progetto` (uno stato UML) è codificato nel linguaggio di programmazione voluto da `Generazione del Codice`, un'attività UML, il cui risultato è il `Codice Sorgente` (un file con una intestazione e l'implementazione).

Il `Codice Sorgente` può subire modifiche ed estensioni durante l'attività di `Programmazione` (messa in atto dal programmatore), e il risultante `Codice Modificato`, ad opera di un'attività (dello strumento CASE) chiamata `Generazione UML`, può essere riconvertito in un `Modello UML Implementato`. Lo strumento può quindi generare un rapporto `Differenze Modelli` per evidenziare le modifiche al progetto, confrontando l'attuale `Modello UML Implementato` e l'ultimo `Modello UML di Progetto`: le modifiche accettate sono integrate in un nuovo `Modello UML di Progetto` e può iniziare la prossima iterazione di round-trip engineering.

Nella pratica, possono nascere problemi dovuti alla non perfetta integrazione fra il linguaggio di programmazione (e il relativo compilatore) e lo strumento CASE. Ad esempio, lo strumento CASE potrebbe non risolvere riferimenti a dichiarazioni contenute in file sorgente specifici del compilatore o di librerie ausiliarie che non siano state esplicitamente incluse nel programma. Sebbene possano esistere modi per aggirare molti di questi problemi, ciò conduce sempre a soluzioni imperfette che richiedono modifiche manuali.

Detto questo, le imperfezioni non devono frenare il processo di round-trip engineering sui programmi client. Le imperfezioni possono essere corrette in modo manuale e i modelli UML generati permettono di documentare in modo consistente il programma (l'alternativa è la perdita di controllo sullo sviluppo dell'applicazione).

9.4.2 Round-trip engineering nei database

La round-trip engineering nei database (Maciaszek, 2000) coinvolge uno *schema fisico dei dati* (PDM, Physical Data Model), al termine del progetto, e il *database* (DB), al termine dell'implementazione. Il modello PDM, il cui sviluppo non è supportato da UML (vedi Sezione 8.4), deve essere orientato a un particolare DBMS.

La Figura 9.15 mostra un diagramma d'attività per la round-trip engineering con un database relazionale. Dopo la costruzione del modello PDM per un database (stato `PDM Iniziale`), esso può essere archiviato (attività `Archiviare`).

La trasformazione da `PDM Iniziale` a `DB Iniziale` è svolta dall'attività `Creare DB` (che dovrebbe generare lo schema del DB, comprendente anche i trigger). Svolgendo l'attività ricorsiva `Caricare`, il `DB Iniziale` è quindi popolato di dati. Se si rendono necessarie delle modifiche, l'attività `Modificare DB` introduce modifiche allo schema e determina la transizione del database allo stato `DB Attuale` (in generale, anche il modello PDM dovrà subire delle modifiche).

A questo punto si presentano due possibilità: le modifiche al `PDM Attuale` possono essere sincronizzate con il `DB Attuale` (attività `Sincronizzare PDM con DB`) oppure le modifiche al `DB Attuale` possono essere sincronizzate con il `PDM Attuale` (attività `Sincronizzare DB con PDM`). Come risultato di queste attività, il `PDM Attuale` dovrebbe essere archiviato e il `DB Attuale` potrebbe dover essere ricaricato.

La Figura 9.16 mostra lo schermo risultante da una sessione di reverse-engineering (attività `Sincronizzare PDM con DB`) da `DB Attuale` (DB Sybase) a `PDM Attuale` (PDM PowerDesigner).

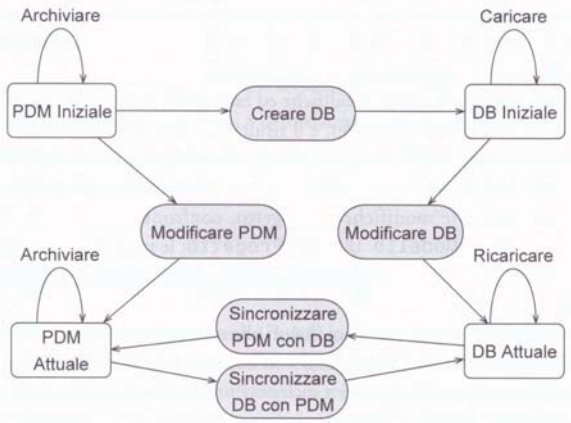


FIGURA 9.15 Round-trip engineering per un database.

Il processo di round-trip engineering nei database dovrebbe considerare che (Maciaszek, 2000):

- Il modello PDM può essere archiviato con un numero di versione per mezzo dello strumento CASE; tuttavia, un tipico RDBMS o ORDBMS non ha la capacità di gestire diverse versioni del DB (se non creando un nuovo DB).
- Dopo la generazione iniziale dello schema del DB, il DB stesso è caricato con i dati e ha inizio la programmazione del server. I programmatori dovrebbero avere la possibilità di modificare lo schema del DB (o chiederne la modifica) fino a quando le modifiche sono sincronizzate con l'attuale PDM. L'attività `Sincronizzare PDM con DB` dovrebbe essere fatta in blocco in momenti specifici di sincronizzazione.
- Qualsiasi modifica successiva al PDM archiviato deve essere sincronizzato con il DB, e quindi determinare una nuova istanza del DB (se la gestione di versioni diverse del DB non è disponibile).

9.4.3 Reingegnerizzare un database relazionale in un object-relational

Come discusso nel Capitolo 8, la prossima generazione della tecnologia per database adotterà probabilmente il modello ORDB. Vi sarà, di conseguenza, una necessità crescente di migrazione dai database relazionali esistenti alle piattaforme object-relational. La *reingegnerizzazione* (reengineering) è il processo che analizza e modifica un *sistema preesistente* (legacy) onde recuperarne il suo progetto e reimplementarlo in una nuova forma.

La reingegnerizzazione e la round-trip engineering condividono le tecnologie di trasformazione di modelli "in avanti" e "all'indietro". La distinzione è che la round-trip engineering s'interessa allo sviluppo evolutivo di un *nuovo sistema* mentre la reingegnerizzazione (che dovrebbe precedere la round-trip engineering) s'interessa alla ricostruzione del modello PDM iniziale dallo studio del *codice legacy* (codice preesistente).

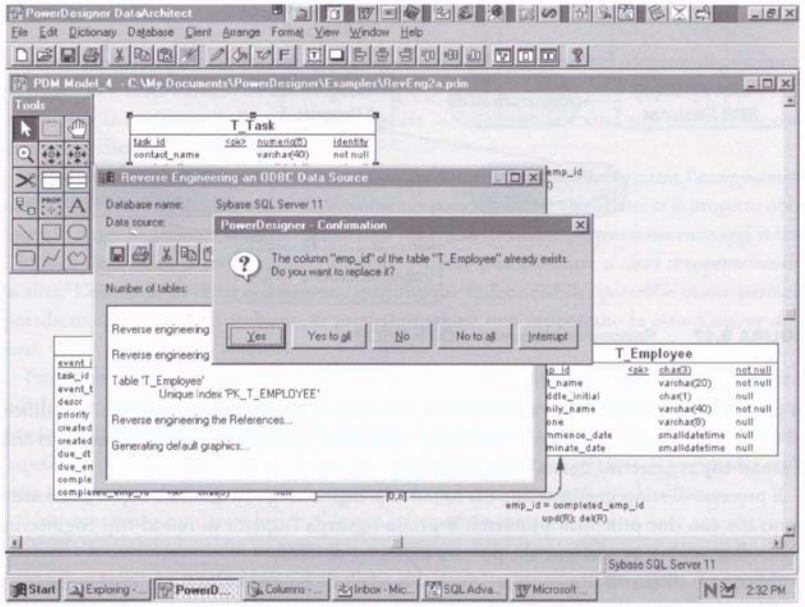


FIGURA 9.16 Sessione di reverse engineering da DB Sybase a PDM PowerDesigner.

Oggi, la maggior parte dei progetti di reingegnerizzazione riguarda la migrazione dei vecchi sistemi (sviluppati in COBOL) verso la tecnologia dei database relazionali. Nel futuro, i database relazionali diverranno sistemi "vecchi" da reingegnerizzare in soluzioni ORDB o ODB. Tuttavia, non è prevista una veloce crescita del mercato dei ODB paragonabile a quella del mercato dei ORDB: in parte a causa dell'inerzia di conversione alle nuove tecnologie e, in parte perché gli ODBMS riguardano attualmente soprattutto mercati di nicchia, come sistemi multimedia e per l'elaborazione cooperativa.

La Figura 9.17 riporta un diagramma d'attività che rappresenta la reingegnerizzazione di un RDB in un sistema ORDB. Il diagramma rappresenta anche il passo successivo di round-trip engineering.

Una `Implementazione RDB Esistente` è analizzata (attività `Analizzare DB`) al fine di determinare le strutture del database, le regole di business e la logica dei programmi applicativi. Quest'analisi conduce alla costruzione di un `Modello UML di Progetto` che riflette lo stato dell'implementazione RDB esistente ed è quindi usato per generare la `Implementazione ORDB` (con il relativo schema ORDB, i trigger e le procedure); il ORDB è poi caricato con i dati

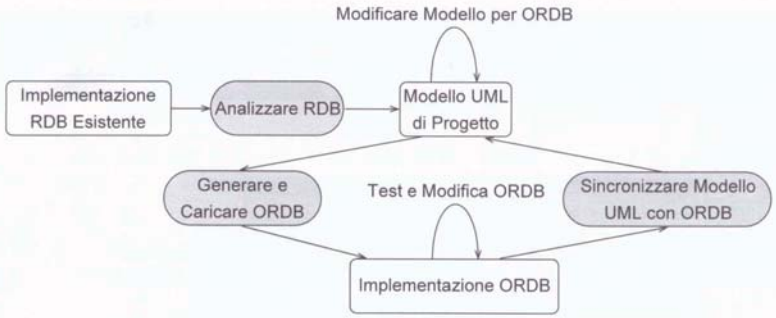


FIGURA 9.17 Reingegnerizzazione da RDB in ORDB.

e i programmi applicativi sono scritti e testati. Questo, in genere, conduce sempre a modifiche del ORDB che devono riflettersi nel `Modello UML di Progetto`: la prima iterazione del ciclo di round-trip engineering è stata completata.

Il processo di reingegnerizzazione e la round-trip engineering con database object-relational si scontrano con due principali problemi: il primo riguarda l'aspetto di round-trip engineering mentre il secondo è in relazione alla potenza espressiva di UML.

Il processo di round-trip engineering tra i modelli di progetto UML e le implementazioni ORDB è simile a quello per database relazionali, ma presenta problemi tecnici notevolmente più complessi. Infatti, il modello ORDB è maggiormente complesso e semanticamente più ricco del modello RDB (Capitolo 8).

Per iniziare, gli strumenti CASE devono adeguarsi al nuovo standard SQL:1999 e comprenderne le variazioni esistenti negli ORDBMS commerciali. Come nel caso del modello RDB, è improbabile che UML permetta di rappresentare correttamente il PDM per i sistemi ORDB. Infatti, UML è principalmente un linguaggio per l'analisi di sistemi. Per rappresentare i sistemi ORDB sono disponibili dei *profili* UML (ad esempio, Maciaszek e Wong, 2000): sfortunatamente, però, essi non trattano il dettaglio della gestione fisica dei dati e non sono in grado di distinguere tutte le sfumature di un ORDBMS, comprese quelle presenti nei diversi rilasci del medesimo ORDBMS.

Ad esempio, Oracle8 usa le viste oggetti per semplificare la migrazione da un database relazionale a uno object-relational. Analogamente ad altri tipi di oggetti ORDB, una vista oggetti può essere rappresentata in UML come una classe stereotipata, eventualmente con una propria icona grafica. Tuttavia, a una classe stereotipata deve essere data una precisa semantica in modo che lo strumento CASE possa comprenderne tutte le caratteristiche (quali tipi di relazione e altri legami sono possibili con qualunque altra classe nel modello). Quindi uno strumento CASE specifico (prodotto in questo esempio dalla Oracle) ha la possibilità di trattare tali problemi in modo più efficiente rispetto a UML.

Sommario

In questo capitolo sono stati trattati alcuni dei concetti più complessi nel progetto di programmi e transazioni e nella transizione dal progetto all'implementazione. Si sono quindi proposti i diagrammi per la navigazione di programma. Inoltre, si è discusso la round-trip engineering con i programmi client e con i database server.

Un programma ben progettato massimizza la *coesione nelle classi* e minimizza l'*accoppiamento tra le classi*. I principi di coesione e accoppiamento possono essere soddisfatti se il progetto obbedisce alla *Legge di Demeter* la quale specifica gli oggetti destinatario permessi nei messaggi interni ai metodi delle classi. L'uso eccessivo di *metodi d'accesso* può condurre a *classi strumentalizzabili* da altre. L'esistenza di *classi disomogenee*, quantunque indesiderabile, potrebbe essere permessa sporadicamente poiché gli ambienti di programmazione non supportano la *classificazione dinamica*.

Progettando le collaborazioni client/server, devono essere tenuti in considerazione i cinque livelli di *interfacce SQL*. *SQL a Livello 5* è di particolare interesse poiché permette all'utente di programmare direttamente il database. *Trigger* e *procedure memorizzate* influenzano pesantemente l'aspetto server del progetto del programma. I *diagrammi per la navigazione di programma* estendono i diagrammi di navigazione nelle finestre per inglobare gli aspetti del database.

Una *transazione* è un'unità logica di lavoro sul database che parte da uno stato consistente del database e assicura, una volta terminata, la consistenza dello stato risultante. Le transazioni assicurano la *concorrenza nel database* e il *ripristino del database* a seguito di un guasto. Le applicazioni convenzionali su database richiedono *transazioni brevi*. Nuove applicazioni su database lavorano con *transazioni lunghe*.

La *round-trip engineering* è il processo per mezzo del quale i modelli di progetto e i programmi sono sincronizzati pur potendo evolvere individualmente. La round-trip engineering combina l'*ingegneria in avanti* (*forward engineering*) dai modelli di progetto ai programmi e l'*ingegneria all'indietro* (*reverse engineering*) dai programmi ai modelli di progetto. La round-trip engineering è normalmente applicata separatamente ai *programmi client* e ai *programmi server*.

Domande

D1 In quale modo il progetto è influenzato dai principi di coesione e accoppiamento delle classi?

D2 Quali oggetti sono permessi come destinatari di un messaggio in conseguenza della Legge di Demeter?

D3 Che cos'è una "classe strumentalizzabile"?

D4 Spiegare le correlazioni tra la classificazione dinamica e le classi disomogenee.

D5 Descrivere brevemente i cinque livelli delle interfacce SQL.

D6 Quali sono i vantaggi di una chiamata di una procedura memorizzata rispetto a un'interrogazione inviata dal programma client al database? Esistono casi che possono suggerire di usare un'interrogazione SQL invece di una chiamata a una procedura memorizzata?

D7 Elencare i principali oggetti server rappresentati nei diagrammi per la navigazione di programma come stati e attività.

D8 Descrivere brevemente i blocchi nel controllo pessimistico della concorrenza.

D9 Descrivere brevemente i livelli d'isolamento di una transazione.

D10 Progettista e DBA possono controllare il ripristino di un database? Spiegare.

D11 Che cos'è una transazione di compensazione? In quale modo può essere usata nel progetto dei programmi?

D12 Che cos'è un punto di salvataggio? In quale modo può essere usata nel progetto dei programmi?

D13 Qual è la funzione del rapporto Differenze Modelli nella round-trip engineering con un programma client?

D14 I database convenzionali non hanno capacità predefinite di gestire diverse versioni di un DB. In che modo questo ne condiziona la round-trip engineering?

Esercizi

E1 Telemarketing – ci si riferisca al caso di studio Telemarketing e alla soluzione dell'Esercizio E4 nel Capitolo 7.

Estendere il diagramma di navigazione nelle finestre per l'applicazione Telemarketing in un diagramma per la navigazione di programma. Se la complessità della soluzione lo suggerisce, suddividere il diagramma in diagrammi piò ridotti. Spiegare in quale modo la vostra soluzione soddisfa i requisiti dell'applicazione.

E2 CMS - ci si riferisca alle soluzioni del caso di studio CMS, Gestione Contatti con i Clienti. In particolare, si consideri il diagramma di navigazione nelle finestre dell'Esempio 7.5 (Sezione 7.6.2).

Specificare procedure memorizzate, trigger e altri oggetti del database (non identificati nelle soluzioni presentate fino a questo punto) necessari al sistema CMS.

Estendere il diagramma di navigazione nelle finestre per il CMS (Figura 7.24) in un diagramma per la navigazione di programma. Se la complessità della soluzione lo suggerisce, suddividere il diagramma in diagrammi più semplici. Spiegare in che modo la vostra soluzione soddisfa i requisiti applicativi.

Capitolo 10

Testing e gestione del cambiamento

Il testing e la gestione del cambiamento non sono fasi distinte del ciclo di vita ma fanno parte integrante di ogni sua fase. Il testing non consiste solo nella ricerca degli errori (debugging) nei programmi: tutto ciò che è stato sviluppato, in una qualsiasi fase del ciclo di vita, deve essere verificato. Analogamente, la gestione del cambiamento non si applica solo alle richieste di estensioni e miglioramenti provenienti dagli stakeholder o ai difetti scoperti durante il testing. La gestione del cambiamento è un aspetto fondamentale nella gestione complessiva del progetto: una richiesta di cambiamento deve essere documentata, l'impatto di ogni modifica sugli elaborati dello sviluppo deve essere tracciato e ricontrollato dopo che la modifica è stata effettivamente realizzata.

La "tracciabilità" è alla base del testing e della gestione del cambiamento. La tracciabilità collega tutti gli elaborati dello sviluppo, compresi i requisiti, al fine di rendere possibile la generazione di una documentazione di sistema completa, di cui sia garantita la correttezza e la consistenza attraverso i vari documenti e modelli, dai requisiti alla documentazione tecnica e a quella dedicata all'utente.

Gli elementi tracciabili possono essere asserzioni testuali o modelli grafici. La tracciabilità stabilisce legami espliciti, diretti o indiretti, tra gli elementi tracciabili. I legami permettono un'analisi d'impatto, normalmente intrapresa se vi è l'intenzione di modificare un qualsiasi elemento coinvolto in una relazione di tracciabilità.

All'inizio del testo si è distinto tra i servizi e i vincoli del sistema. La tracciabilità e il testing sono spesso associati con i servizi del sistema manifestati nei requisiti dei casi d'uso. Tuttavia, non si deve dimenticare che anche i vincoli del sistema devono essere verificati e gestiti.

10.1 Testing dei servizi del sistema

Schach (1996) distingue tra testing metodico e testing informale dei servizi del sistema. Qualsiasi sviluppatore opera un *test informale* mentre modella o implementa un servizio del sistema. Per sua stessa natura, il testing informale è lontano dall'essere perfetto; inoltre la persona che ha sviluppato il servizio è probabilmente la meno adatta per ricercare gli errori nel servizio stesso.

Un testing informale è solo di marginale interesse e deve essere associato a un testing metodico. Vi sono due principali tipi di *testing metodico* (Schach, 1996):

1. Non basato sull'esecuzione (revisione formale)
   - Incontri di rassegna (walkthrough).
   - Ispezioni (inspection).
2. Basato sull'esecuzione
   - Testing delle specifiche.
   - Testing del codice.

10.1.1 Incontri di rassegna

Un *incontro di rassegna* è un tipo di revisione, basata su una riflessione di gruppo, che può essere condotta durante una qualsiasi fase dello sviluppo. È un incontro amichevole di sviluppatori, attentamente pianificato, con obiettivi precisi, un'agenda, una durata e dei partecipanti. Molti gruppi di sviluppo conducono incontri di rassegna su base settimanale.

Pochi giorni *prima dell'incontro di rassegna*, ai partecipanti sono consegnati i materiali (modelli, documenti, codice programmi, etc.) che devono essere rivisti durante l'incontro. I materiali sono raccolti e distribuiti ai partecipanti dal moderatore. I partecipanti studiano i materiali e provvedono, prima dell'incontro, a inviare al moderatore i propri commenti.

L'incontro è relativamente breve (due o tre ore al massimo). *Durante l'incontro* il moderatore presenta i commenti e, su ciascuno, apre la discussione. Il proposito dell'incontro è di determinare esattamente il problema, non di tormentare lo sviluppatore! Lo sviluppatore dietro il problema non è importante e potrebbe anche restare anonimo (sebbene normalmente questo non sia il caso). L'unico obiettivo è confermare l'esistenza di un problema mentre una sua soluzione non deve neppure essere tentata.

È opinione comune che il lavoro di rassegna funzioni molto bene: introduce rigore e professionalità nel processo di sviluppo, contribuisce alla produttività e al rispetto dei tempi, produce informazioni molto importanti e migliora la qualità del software.

10.1.2 Ispezioni

Analogamente alla rassegna, l'*ispezione* è un incontro amichevole; si svolge sotto la diretta supervisione della funzione aziendale che gestisce il progetto, con lo scopo di identificare e verificare i difetti, registrarli e programmare quando e da chi devono essere eliminati. Diversamente dagli incontri di rassegna, le ispezioni sono più formali e rigorose, sono condotte con minor frequenza e indirizzate solo verso determinati problemi critici.

Un'ispezione inizia con uno *stadio di pianificazione* che identifica i partecipanti e l'area d'interesse dell'ispezione stessa. Prima della sessione d'ispezione, è possibile avere un breve *incontro informativo*, durante il quale lo sviluppatore del prodotto da ispezionare introduce il tema. Il materiale per l'ispezione è consegnato ai partecipanti, nel periodo antecedente o direttamente durante l'incontro informativo.

L'*incontro informativo* si svolge, normalmente, una settimana prima dell'incontro vero e proprio per dare al gruppo di ispezione il tempo di studiare il materiale. Durante l'incontro di ispezione, i difetti sono identificati, registrati e numerati. Immediatamente dopo l'incontro di ispezione, il moderatore prepara un *archivio dei difetti*, gestito possibilmente da uno *strumento di gestione del cambiamento* associato al progetto.

Allo sviluppatore è normalmente richiesto di risolvere rapidamente i difetti e di registrare le loro soluzioni nello strumento di gestione. Il moderatore dovrebbe verificare la risoluzione dei difetti e decidere se sia necessaria una *re-ispezione*; una volta soddisfatto dalle risoluzioni apportate, d'accordo con il gestore del progetto, sottopone il prodotto al gruppo di *Software Quality Assurance* (SQA) dell'organizzazione (se tale gruppo esiste). Tale gruppo non deve essere coinvolto nel progetto se non per la verifica della qualità finale del prodotto, di cui il gruppo stesso (e non gli sviluppatori) è responsabile.

10.1.3 Testing delle specifiche

Il *testing delle specifiche* è un tipo di test basato sull'esecuzione e, quindi, si applica a prodotti software eseguibili (non a documenti o a modelli). Si ritrova anche sotto una varietà di nomi differenti, quali *testing a scatola nera* (black box), testing funzionale, testing guidato dall'input/output, etc.

Il principio su cui si basa il testing delle specifiche è il trattamento del modulo visto come una scatola nera di cui si ignora il contenuto (logica del programma e/o algoritmi computazionali), la quale riceve alcuni input e produce alcuni output.

Nel testing delle specifiche, i *requisiti di test* possono essere derivati dai *casi d'uso*; essi devono essere documentati in un *documento dei casi di test* e la loro verifica deve essere pianificata. Il piano dei test e il documento dei casi di test costituiscono uno scenario che deve essere memorizzato in un opportuno *strumento di gestione dei test*, per essere usato più volte durante lo sviluppo incrementale dei moduli (*testing di regressione*, Sezione 1.3.10).

Il testing delle specifiche evidenzia difetti che sono normalmente difficili da scoprire con altri mezzi. In particolare, il testing delle specifiche scopre *funzionalità dimenticate*, cioè funzionalità completamente trascurate oppure documentate come requisiti nei casi d'uso (e, conseguentemente, come requisiti di test) cui non ha fatto seguito l'implementazione.

10.1.4 Testing del codice

Il secondo tipo di test basato sull'esecuzione, il *testing del codice*, è anche conosciuto con altri nomi: *testing a scatola bianca* (white box) o *trasparente* (glass box), testing guidato dalla logica, testing orientato ai cammini.

Il testing del codice inizia con un'attenta analisi degli algoritmi del programma. I casi di test sono costruiti per *sollecitare il codice*, in altre parole per garantire che tutti i possibili cammini d'esecuzione nel programma siano verificati. Il testing del codice può essere supportato da strumenti di gestione dei test, e usato per il testing di regressione; tuttavia, per sua natura, richiede, in genere, un ampio coinvolgimento del programmatore nella messa a punto dei test e nell'utilizzo degli strumenti.

Il testing del codice, analogamente a tutte le altre forme di testing basate sull'esecuzione, non può essere esaustivo a causa dell'esplosione combinatoria del numero dei possibili casi di test anche per una modesta crescita della complessità del programma: a tal proposito si osservi che anche se fosse possibile fare un test per ogni possibile cammino d'esecuzione, non si potrebbe comunque garantire di aver trovato ogni difetto. L'antico adagio del testing recita: "Il testing può solo eliminare alcuni errori ma non può provare che il programma è corretto!".

10.2 Testing dei vincoli del sistema

Il *testing dei vincoli del sistema* è principalmente *basato sull'esecuzione*. Il suo scopo è determinare se i vincoli del sistema siano stati implementati come elencati nei requisiti e nei documenti di test. Il testing dei vincoli del sistema comprende elementi come:

- testing dell'interfaccia utente;
- testing del database;
- testing delle autorizzazioni;
- testing delle prestazioni;
- testing di stress;
- testing di comportamento a fronte di guasti;
- testing di configurazione;
- testing d'installazione.

I primi due tipi di test, *testing dell'interfaccia utente* e *testing del database*, sono strettamente associati al testing dei servizi del sistema e normalmente condotti in parallelo a esso: perciò sono anche inclusi nel documento dei casi di test (Sezione 10.3) prodotto per il testing dei servizi del sistema.

10.2.1 Testing dell'interfaccia utente

Il testing dell'interfaccia GUI s'intreccia con l'intero processo di sviluppo del software. Inizia nella fase di determinazione dei requisiti, insieme ad attività quali l'inclusione dei progetti delle finestre nei documenti dei casi d'uso e costruzione di prototipi GUI. Questi test GUI iniziali sono orientati alla verifica dei requisiti funzionali e all'usabilità.

Più avanti, quando il sistema è stato implementato, è richiesto un *testing post-implementazione*. I test sono dapprima condotti dagli sviluppatori, poi dal personale preposto al test e, infine (prima del rilascio del software), dai clienti (*test pilota*). Il seguente elenco è un esempio di domande, in un documento di test, progettato per il test post-implementazione dell'interfaccia (Bourne, 1997):

- Il nome della finestra corrisponde alla sua funzione?
- La finestra è modale o no? Come dovrebbe essere?
- Esiste una distinzione visiva tra campi opzionali e obbligatori?
- La finestra può essere ridimensionata, spostata, chiusa e riattivata? Dovrebbe essere possibile?
- È stato dimenticato qualche campo?
- Titoli, etichette, abbreviazioni, etc. sono scritti correttamente?
- I pulsanti di comando (`OK`, `Cancella`, `Salva`, etc.) sono usati consistentemente in tutte le finestre di dialogo?
- È sempre possibile annullare l'operazione corrente (compresa l'operazione di cancellazione)?
- I campi statici sono protetti dalle modifiche degli utenti? Se l'applicazione può modificare il testo statico, questo è fatto correttamente?

- I tipi e le dimensioni dei caratteri applicati ai campi di testo statico sono consistenti? Sono scritti correttamente?
- Le dimensioni dei campi di modifica corrispondono agli insiemi di valori che assumono?
- All'apertura della finestra, i campi di modifica sono inizializzati con i valori corretti?
- Tutti i valori immessi nei campi di modifica sono verificati dal programma client?
- Tutti i valori nelle liste a tendina sono correttamente popolati dal database?
- Le maschere per l'acquisizione dei dati sono consistenti con le specifiche?
- I messaggi d'errore sono leggibili e semplici da comprendere?

10.2.2 Testing del database

Come il testing dell'interfaccia, il testing del database è intrinseco in molte altre tipologie di test. La maggior parte del testing delle specifiche è basata su dati in input/output dal database. Tuttavia, un testing separato sul database è ancora necessario.

Il *testing del database* dopo la sua implementazione prevede, in genere, un test a scatola bianca (del codice) diretto alle *transazioni* (altri aspetti riguardanti il database, come: prestazioni, concorrenza, autorizzazioni, etc., possono essere trattati separatamente).

Come per il test dell'interfaccia, il test del database deve essere condotto per ciascuna funzione applicativa. I problemi su cui orientarsi nei test del database dovrebbero essere descritti in un documento generico. Tale documento dovrebbe essere allegato a tutti i test funzionali (servizi del sistema). Il seguente elenco fornisce un esempio dei problemi verso cui dovrebbero orientarsi i test (Bourne, 1997):

- Verificare che, con input corretti, la transazione si comporti come previsto. La reazione del sistema verso l'interfaccia è corretta? Il contenuto del database è corretto dopo la transazione?
- Verificare che, con input scorretti, la transazione si comporti come previsto. La reazione del sistema verso l'interfaccia è corretta? Il contenuto del database è ancora corretto dopo la transazione?
- Interrompere la transazione prima che finisca. La reazione del sistema verso l'interfaccia è corretta? Il contenuto del database è ancora corretto dopo la transazione?
- Eseguire la transazione concorrentemente con diversi processi. Rendere una transazione proprietaria di un blocco su una risorsa dati richiesta da altre transazioni. Gli utenti ottengono spiegazioni comprensibili dal sistema? Il contenuto del database è corretto dopo che la transazione è terminata?
- Estrarre qualsiasi istruzione SQL del client ed eseguirla interattivamente sul database. I risultati sono quelli previsti e uguali a quelli prodotti quando l'istruzione SQL è eseguita dal programma?
- Eseguire interattivamente un testing a scatola bianca delle più complesse interrogazioni SQL (da un programma client o da una procedura memorizzata), coinvolgendo tutti gli operatori relazionali di base (join, unione, valori nulli, funzioni di aggregazione, etc.).

10.2.3 Testing delle autorizzazioni

Il *testing delle autorizzazioni* può essere trattato come un'estensione dei primi due tipi di test. Gli oggetti sia nel client (interfaccia utente) sia nel server (database) dovrebbero essere protetti da un uso non autorizzato, quindi il testing dovrebbe verificare che i meccanismi di sicurezza definiti nel client e nel server proteggano effettivamente il sistema dall'intrusione non autorizzata.

La protezione inizia dal client; l'*interfaccia utente* del sistema dovrebbe essere capace di configurarsi dinamicamente per adeguarsi al livello d'*autorizzazione* dell'utente attuale (*autenticato* tramite identificatore e parola chiave propri): alcuni elementi dei menu, pulsanti di comando o anche intere finestre, dovrebbero essere resi inaccessibili all'utente che non ha la corretta autorizzazione.

Non tutte le possibili violazioni alla sicurezza possono essere trattate dal client. Il supporto per le autorizzazioni è un elemento rilevante di qualsiasi DBMS. A un utente può essere concesso il *privilegio* di:

- accedere a *oggetti server* individuali (tabelle, viste, colonne, procedure memorizzate, etc.);
- eseguire istruzioni SQL (select, update, insert, delete, etc.).

I permessi di un utente possono provenire da entrambi i tipi di privilegio ed essere assegnati a *livello utente* direttamente oppure a *livello gruppo*. I gruppi permettono all'amministratore della sicurezza di assegnare permessi a un gruppo di utenti con una singola operazione. Un utente può essere parte di nessuno o di diversi gruppi.

Per permettere maggiore flessibilità nella gestione delle autorizzazioni, i DBMS introducono, generalmente, un ulteriore livello d'autorizzazione, il livello *ruolo*. Il ruolo permette all'amministratore della sicurezza di assegnare gli stessi permessi a tutti gli utenti che ricoprono un determinato ruolo nell'organizzazione. I ruoli possono essere annidati e i permessi possono sovrapporsi.

Nei grandi sistemi informativi, il *progetto delle autorizzazioni* è un'attività complessa. Spesso, è prodotto un *database delle autorizzazioni*, accanto al database dell'applicazione, per memorizzare e manipolare i permessi client e server. Il programma applicativo consulta tale database dopo un accesso dell'utente, per identificare il suo livello d'autorizzazione e configurarsi di conseguenza.

Qualsiasi modifica ai permessi concernenti il database dell'applicazione è guidata dal database delle autorizzazioni: nessuno, neppure l'amministratore della sicurezza, può modificare direttamente i permessi al database dell'applicazione senza prima aggiornare il database delle autorizzazioni. La Figura 10.1 mostra un possibile progetto di database delle autorizzazioni.

10.2.4 Testing di altri vincoli

Il testing dei vincoli del sistema comprende anche:

- testing sulle prestazioni;
- testing di stress;
- testing di comportamento a fronte di guasti;
- testing di configurazione;
- testing d'installazione.

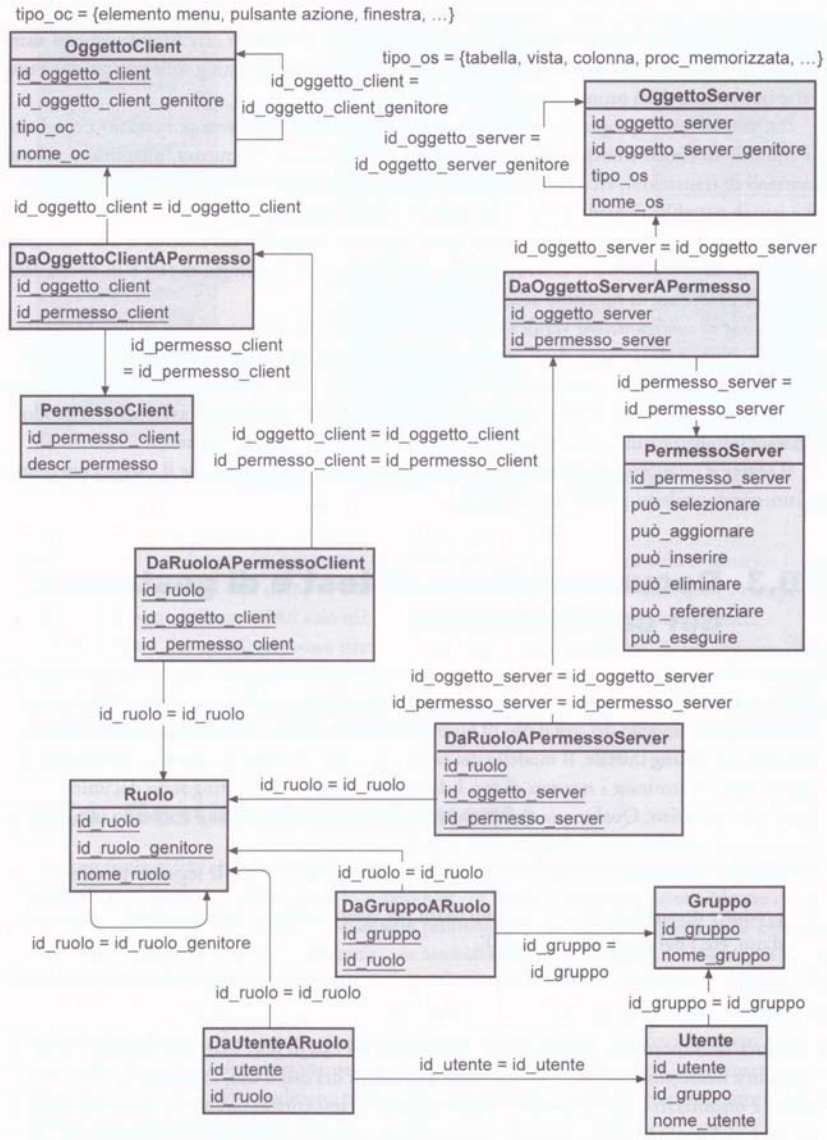


FIGURA 10.1 Progetto di un database delle autorizzazioni.

Il *testing sulle prestazioni* misura dei parametri di prestazione che si riferiscono, in genere, alla *velocità* e al *throughput* delle transazioni. I test sono condotti per diversi carichi di lavoro del sistema, comprendendo eventuali *picchi di carico* prevedibili a priori. Il testing sulle prestazioni è una parte importante della ottimizzazione (tuning) del sistema.

Il *testing di stress* è progettato per provocare il collasso del sistema sottoponendolo a condizioni anomale di carico (risorse scarse e allocate in modo scorretto, frequenza, quantità, o volumi anormali di transazioni, etc.). Il testing di stress è spesso accoppiato con il testing sulle prestazioni e perciò potrebbe richiedere un simile ambiente hardware e software.

Il *testing di comportamento a fronte di guasti* riguarda la risposta del sistema a una varietà di malfunzionamenti hardware, della rete, o del software. Questa tipologia di test è in stretta relazione alle procedure di ripristino attuate dal DBMS.

Il *testing di configurazione* verifica come il sistema opera su diverse configurazioni hardware e software. Nella maggioranza degli ambienti di produzione, si presuppone che il sistema debba comportarsi correttamente su varie stazioni di lavoro client che si connettono al database, usando una varietà di protocolli di rete. Le stazioni di lavoro client potrebbero avere diversi prodotti software installati e quindi potrebbero crearsi dei conflitti con l'applicazione.

Il *testing d'installazione* estende il testing di configurazione e verifica che il sistema operi propriamente su qualsiasi piattaforma installata.

10.3 Documentazione di test e di gestione del cambiamento

La *documentazione di test e di gestione del cambiamento* è parte integrante di altri documenti di sistema, a cominciare dai documenti dei casi d'uso (Figura 10.2). Le *caratteristiche* del sistema, identificate nel modello dei casi d'uso di business (Sezione 3.5.2), possono essere usate per scrivere il piano di testing iniziale. Il modello dei casi d'uso è quindi usato per scrivere i documenti dei casi di test e determinare i *requisiti di test*. I *difetti* trovati durante il testing sono documentati nel *documento dei difetti*. Qualunque *requisito dei casi d'uso* non implementato è inserito in un elenco contenuto in un documento delle estensioni dell'applicazione.

Quando è impiegato uno strumento CASE, gli sviluppatori hanno le seguenti possibilità:

- produrre documenti testuali e quindi usarli per creare requisiti (requisiti di test, requisiti casi d'uso, etc.) nel repository del CASE;
- usare lo strumento CASE per inserire i requisiti nel repository e quindi generare la documentazione.

La Figura 10.3 mostra un estratto di un documento dei casi di test, usato per prendere nota, nel repository, dei *requisiti di test*. Analogamente ai requisiti dei casi d'uso, i requisiti di test sono numerati e organizzati gerarchicamente. Molti requisiti di test corrispondono direttamente a requisiti dei casi d'uso, quindi la principale sezione nel documento è chiamata "Conformità alle Specifiche dei Casi d'Uso".

Altre sezioni del documento dei casi di test potrebbero identificare requisiti per il testing dell'interfaccia, del database e di componenti generici riusabili. Tali tipologie di test devono essere incluse, nel documento, come parte del testing delle unità funzionali per due ragioni. Anzitutto, per

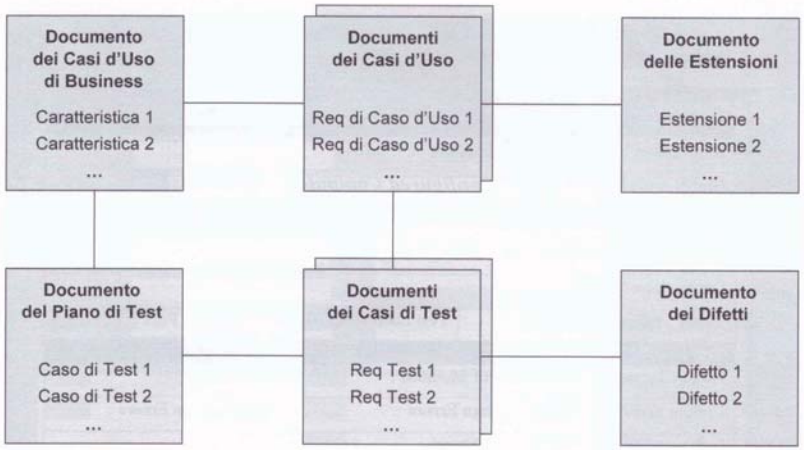


FIGURA 10.2 Documenti rilevanti per il testing e la gestione del cambiamento.

fare il test dell'interfaccia, del database o dei componenti generici, è necessario il contesto funzionale che dà un senso ai dati di input e di output. Inoltre, l'interfaccia, il database e i componenti generici possono mostrare difetti solo nel contesto di alcuni test funzionali, non di tutti.

I documenti dei casi di test sono usati per registrare i risultati dei test: le tre colonne che seguono ogni requisito di test nella Figura 10.3 servono a questo scopo. Un requisito di test può, alternativamente, fallire il test, passarlo sotto certe condizioni (nel qual caso è richiesta una spiegazione), passarlo incondizionatamente.

10.4 Gestire il cambiamento

Il testing rivela *difetti* che devono poi essere eliminati. A tal fine, i difetti sono formulati come *richieste di cambiamento* (change request), e assegnate, in seguito, agli sviluppatori. Alcune richieste di cambiamento possono riguardare *estensioni* piuttosto che difetti. Sia i difetti sia le estensioni implicano cambiamenti di stato, possono aver associata una priorità, avere dei responsabili, debbono poter essere tracciati fino al loro punto di origine nei documenti di test e dei casi d'uso, etc.

La gestione del cambiamento è un compito pesante per qualunque progetto in cui sono coinvolti più sviluppatori. Si consideri uno scenario dove due diversi difetti sono assegnati a due diversi sviluppatori per l'eliminazione; supposto che per correggere tali difetti, apparentemente indipendenti, siano richieste modifiche nello stesso componente, entrambi gli sviluppatori possono lavorare simultaneamente sulle correzioni e, alla fine, le modifiche più recenti potrebbero vanificare quelle meno recenti!

Per gestire correttamente il cambiamento, è necessario uno *strumento di gestione delle richieste di cambiamento* (parte dello strumento CASE). Lo strumento dovrebbe permettere di gestire "on-line" il cambiamento e assicurare che tutti gli sviluppatori lavorino sui documenti più recenti.

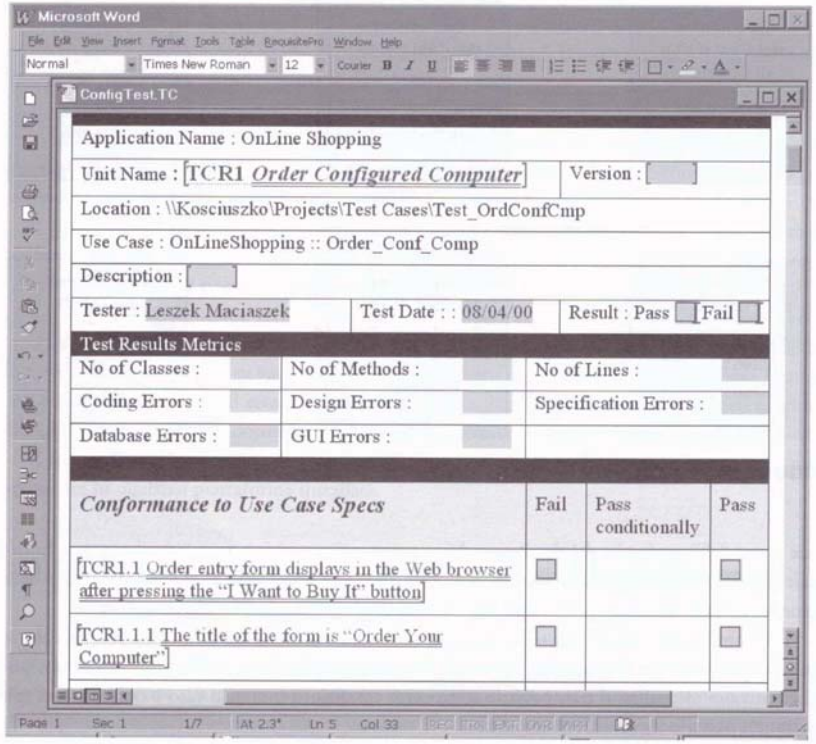


FIGURA 10.3 Documento dei casi di test.

Le modifiche ai documenti, introdotte da un membro del progetto, devono essere rese immediatamente disponibili agli altri sviluppatori. I potenziali conflitti sono risolti attraverso meccanismi di *blocco* e *controllo delle versioni*. Nel primo caso, un documento bloccato è temporaneamente indisponibile ad altri sviluppatori. Nel secondo caso, possono essere create versioni distinte dello stesso documento e qualsiasi conflitto tra le versioni è risolto, più avanti, attraverso una negoziazione.

10.4.1 Sottoporre una richiesta di cambiamento

Generalmente, una *richiesta di cambiamento*, relativa a un difetto o a un'estensione, è immessa nel repository del progetto. Una volta immessa, gli sviluppatori possono monitorarne i progressi, osservarne lo *stato* ed agire su essa. Le *azioni* che possono essere eseguite su una richiesta di cambiamento dipendono dallo stato corrente della richiesta stessa.

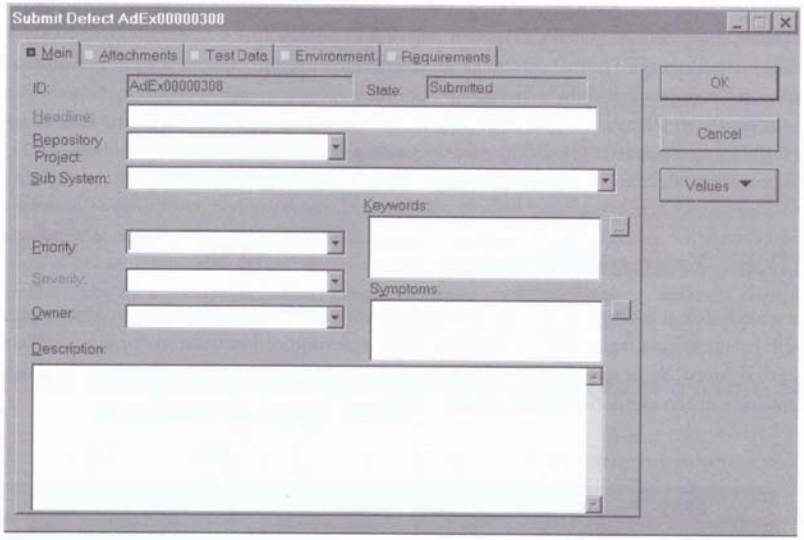


FIGURA 10.4 Finestra per inserire un difetto.

La Figura 10.4 mostra la scheda principale di una finestra di dialogo atta a immettere difetti che sono numerati e descritti in dettaglio (Rational, 2000). Le informazioni su priorità, gravità, progetto e proprietario sono immesse per mezzo di liste a tendina delle scelte possibili, liste che possono essere modificate secondo le necessità del progetto. Altri campi permettono di inserire informazioni descrittive, inclusa la possibilità di allegare una documentazione opportuna, ad esempio, dei frammenti di codice.

L'azione di *sottoporre* una richiesta di cambiamento può avere come risultato l'automatica *notifica via e-mail* ai membri del gruppo. La richiesta di cambiamento si porta quindi nello stato *Sottoposta*. Il gestore del progetto potrebbe configurare lo strumento per permettere azioni predefinite in ciascun stato. Ad esempio, nello stato Sottoposta, le azioni potrebbero essere: `Assegna` (a un membro del gruppo), `Modifica` (alcuni dettagli sulla richiesta), `Ritarda`, `Elimina` (richiesta senza soluzione), `Chiudi` (probabilmente, a seguito della risoluzione della richiesta).

10.4.2 Tenere traccia delle richieste di cambiamento

Ogni richiesta di cambiamento è assegnata a un membro del gruppo. Questi può `aprire` la richiesta e, portandola nello stato *Aperta*, ne rende impossibile una modifica del suo stato da parte di qualunque altro membro del gruppo.

Quando la richiesta di cambiamento è risolta, i dettagli della risoluzione sono memorizzati e una notifica via e-mail può essere inviata ai manager del progetto e al personale di test. Quest'ultimo potrebbe dover eseguire l'azione `Verifica` sulla richiesta di cambiamento risolta.

In ogni momento, lo strumento di gestione del cambiamento deve tener traccia delle richieste e rendere disponibili sia grafici sia rapporti (*metriche di progetto*) per monitorare il processo di risoluzione (ad esempio, fornire statistiche sui difetti non assegnati, assegnati, e risolti, valutare il carico di lavoro per ciascun membro del gruppo, etc.).

La Figura 10.5 mostra un grafico dei difetti attivi ordinati per priorità: sei difetti devono essere risolti immediatamente (1), a cinquantacinque dovrebbe essere data grande attenzione (2), sessantasette sono nella coda normale (3), e sessantotto hanno una bassa priorità (4).

10.5 Tracciabilità

La tracciabilità, il testing e la gestione del cambiamento non sono obiettivi in sé e devono essere svolti in na misura ragionevole: gli sviluppatori non devono concentrarsi su di essi ma sull'attività di sviluppo! Sorge quindi un problema di bilanciamento fra il costo associato a queste attività ausiliarie e quello che deriva dal *non* tenerne conto.

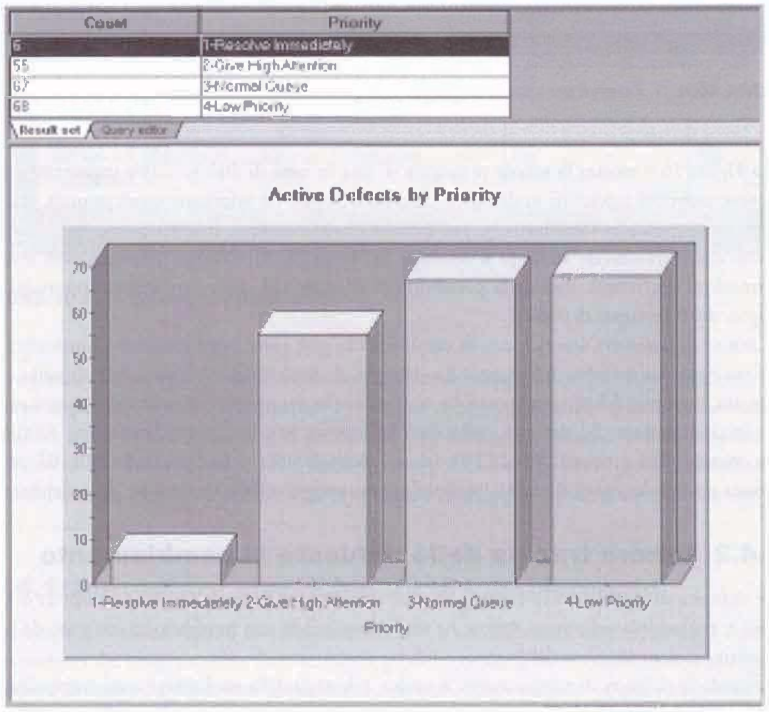


FIGURA 10.5 Grafico delle metriche.

Poiché la tracciabilità sta alla base del testing e della gestione del cambiamento, l'*analisi costi-benefici* dovrebbe determinare gli obiettivi e il livello di tracciabilità nel progetto. Come minimo, la tracciabilità dovrebbe essere conservata tra i requisiti dei casi d'uso e i difetti. In un modello più elaborato, potrebbero essere coinvolti, nelle relazioni di tracciabilità, le caratteristiche di sistema, i casi di test, le estensioni, i punti di verifica dei test, e altri elaborati di sviluppo del software.

Nel seguito, si prenderà in considerazione un modello di tracciabilità basato sui legami tra i documenti di sistema mostrati nella Figura 10.2. Il documento dei casi d'uso di business elenca le *caratteristiche del sistema*. Il documento del piano di test identifica i *casi di test*. Le *caratteristiche* sono legate ai *casi di test* e ai *requisiti dei casi d'uso* contenuti nel documento dei casi d'uso. Dai *requisiti di test* nei documenti dei casi di test si può risalire ai *casi di test*, ai *requisiti dei casi d'uso*, ai *difetti* e anche alle *estensioni* attraverso i *requisiti dei casi d'uso*. La tracciabilità diretta dai *difetti* alle *estensioni* non è necessaria.

10.5.1 Dalle caratteristiche del sistema ai casi d'uso e ai requisiti dei casi d'uso

Una *caratteristica del sistema* è una generica funzionalità da implementare nel sistema, e corrisponde, generalmente, a un caso d'uso di business in un modello dei casi d'uso di business (Sezione 3.5.2). Se il *modello dei casi d'uso di business* non è stato sviluppato, allora le caratteristiche del sistema sono identificate nel documento che esprime la *visione* aziendale a livello strategico.

Ogni caratteristica del sistema è realizzata da un insieme di *requisiti dei casi d'uso* associati a uno o più *casi d'uso*. Tracciare all'indietro i casi d'uso verso le necessità espresse dagli stakeholder (in termini di caratteristiche del sistema), aiuta a convalidare la correttezza del modello dei casi d'uso. Tale strategia è utile durante la fase di raccolta dei requisiti (semplificando il completamento della fase) e può anche aiutare nello sviluppo incrementale del prodotto.

Con questa strategia e se i requisiti dei casi d'uso propri di (ovvero interni a) ciascun caso d'uso sono solo indirettamente legati alle caratteristiche, potrebbe sorgere un problema. Infatti, sarebbe possibile arrivare a situazioni dove c'è una traccia tra una caratteristica e un caso d'uso, e una parte dei requisiti di quel caso d'uso non riguarda quella caratteristica bisogna decidere, quindi, se una tale traccia sia ancora valida. Al fine di evitare tali problemi di scalabilità e di lungo termine associati a tale strategia, la matrice di tracciabilità dovrebbe tracciare le caratteristiche non solo verso i casi d'uso ma anche direttamente verso i requisiti dei casi d'uso. Questo è possibile se ciascun caso d'uso è trattato come una gerarchia contenente al suo interno i requisiti che hanno contribuito a determinarlo.

Tale situazione è mostrata nella Figura 10.6. Le colonne contengono i casi d'uso e i requisiti dei casi d'uso interni ai casi d'uso. La visualizzazione gerarchica dei casi d'uso può essere espansa (fino ad arrivare ai requisiti) e ritratta (tramite i pulsanti + e -). Le frecce rappresentano le tracce dalle caratteristiche ai casi d'uso e ai requisiti dei casi d'uso. Alcune frecce sbarrate (\) evidenziano *tracce sospette*. Una traccia diviene sospetta quando viene modificato uno dei requisiti coinvolti. Lo sviluppatore deve esaminare i legami sospetti per definirli con precisione.

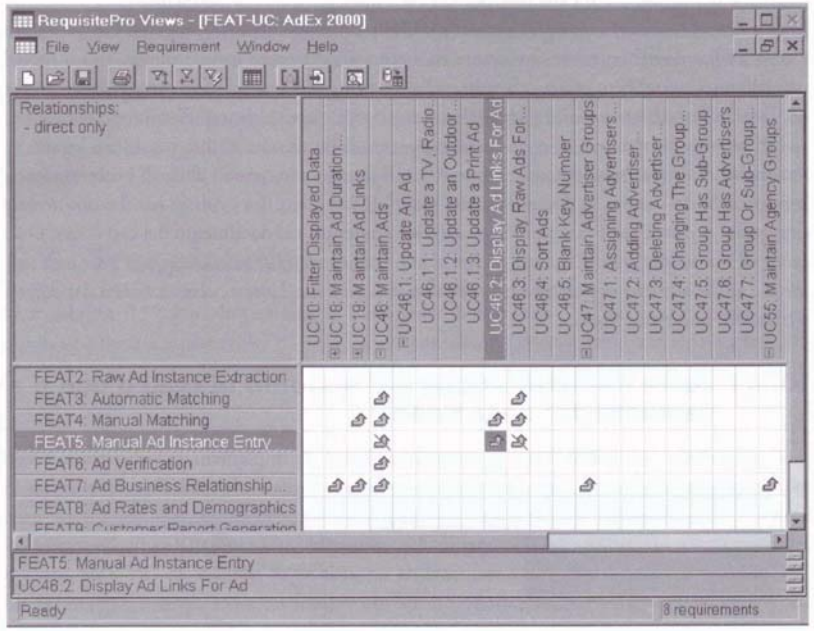


FIGURA 10.6 Tracciabilità dalle caratteristiche verso casi d'uso e requisiti dei casi d'uso (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

10.5.2 Dai piani di test ai casi di test e ai requisiti di test

Il documento del *piano di test* è per i casi di test quello che il documento dei casi d'uso di business è per i casi d'uso. Il piano di test identifica l'informazione progettuale ad alto livello e i componenti software (*casi di test*) su cui si dovrebbe condurre il test. Il piano di test descrive anche la strategia di testing per il progetto, le risorse richieste per il test, lo sforzo e il costo.

Ogni caso di test identificato nel piano di test deve essere descritto in un documento del caso di test. Correlare i requisiti di test ai casi di test e ai piani di test, porta con sé benefici simili alla tracciabilità tra le caratteristiche, i casi d'uso e i requisiti dei casi d'uso.

La Figura 10.7 mostra la matrice di tracciabilità dal piano di test ai casi di test e ai requisiti di test (interni ai casi di test). La visualizzazione gerarchica dei requisiti del test può essere ritratta ed espansa.

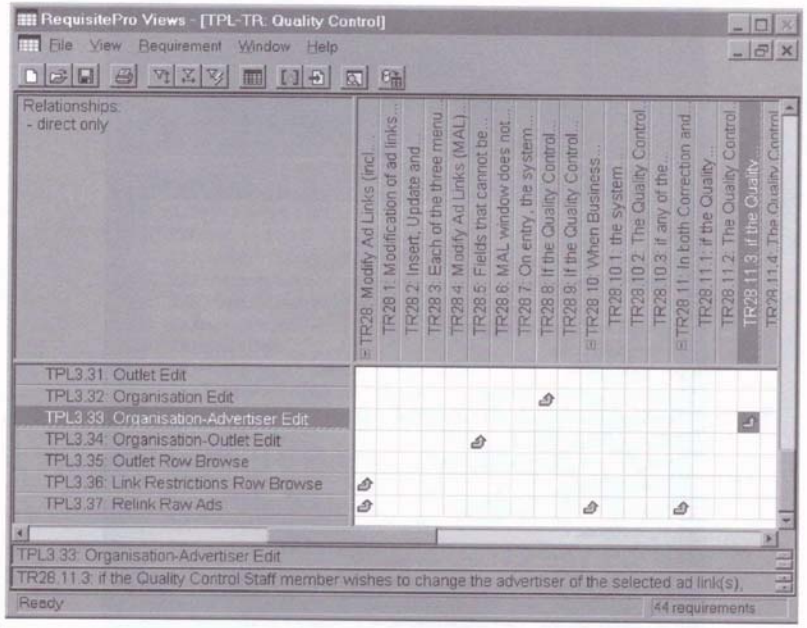


FIGURA 10.7 Tracciabilità dal piano di test verso casi di test e requisiti di test (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

10.5.3 Dai diagrammi UML ai documenti e ai requisiti

La tracciabilità e la gestione del cambiamento non si applicano solo ai documenti e ai requisiti testuali memorizzati nel repository del prodotto CASE. Il repository contiene anche modelli UML i cui oggetti grafici possono essere legati a documenti e requisiti.

La tracciabilità tra gli elaborati dei diagrammi UML e ogni altro elemento del repository (in particolare documenti e requisiti) è importante soprattutto per i casi d'uso. La Figura 10.8 mostra la finestra di dialogo atta a collegare un caso d'uso (`Maintain Ads`) a uno qualunque dei documenti contenuti nel repository (compresi quelli mostrati in Figura 10.2).

La Figura 10.9 mostra una finestra di dialogo per legare un caso d'uso (nuovamente `Maintain Ads`) a uno qualsiasi dei requisiti di caso d'uso.

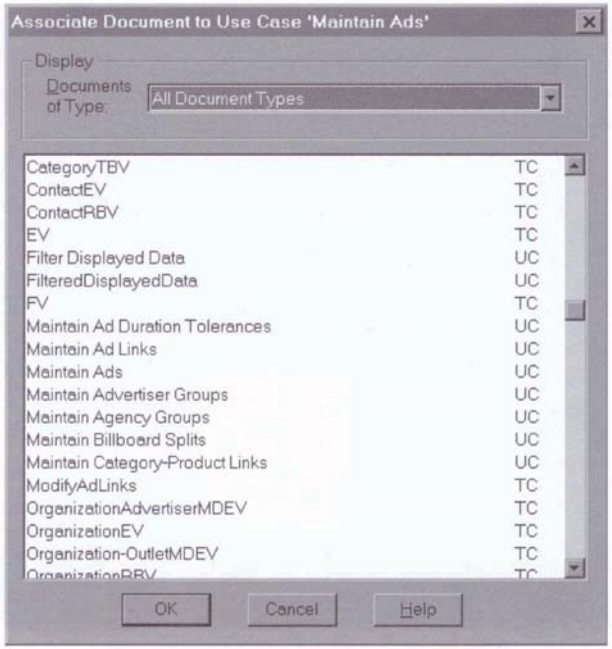


FIGURA 10.8 Collegare un documento a un caso d'uso (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

10.5.4 Dai requisiti dei casi d'uso ai requisiti di test

La tracciabilità tra i requisiti dei casi d'uso e i requisiti di test è critica per valutare se l'applicazione soddisfi i requisiti di business. I legami tra questi due tipi di requisito permettono all'utente di tracciare, all'indietro, i difetti, prima verso i requisiti di test, quindi verso i requisiti dei casi d'uso, risalendo alle caratteristiche del sistema interessate dai difetti stessi (si veda Figura 10.2).

La Figura 10.10 mostra la matrice di tracciabilità con le tracce tra i requisiti dei casi d'uso e i requisiti di test. Si noti che, sia i requisiti dei casi d'uso sia i requisiti di test sono organizzati gerarchicamente (i livelli gerarchici ai quali le tracce sono definite possono essere predeterminati).

10.5.5 Dai requisiti di test ai difetti

I documenti dei casi di test sono, in genere, specificati sotto forma di script che contengono dei *punti di verifica* definiti a partire dai requisiti di test. Gli script sono usati in test *manuali* ma molti di questi possono essere *automatizzati* per essere usati in strumenti di testing.

Un punto di verifica è usato (nel *testing di regressione*) per verificare lo stato di un oggetto di test in tutte le diverse versioni dell'applicazione sotto analisi. Vi sono vari tipi di punti di verifica

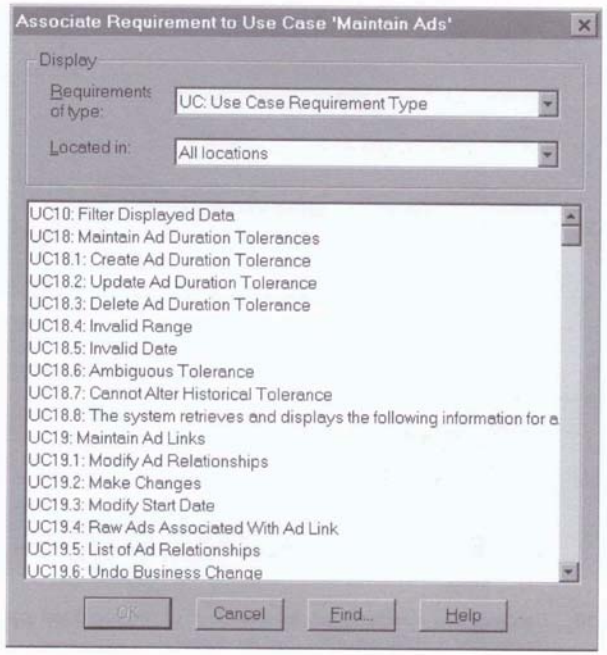


FIGURA 10.9 Collegare un requisito a un caso d'uso (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

(Rational, 2000); un punto di verifica può essere pensato per controllare che: un testo non sia stato modificato, certi valori numerici siano accurati, due file siano ancora uguali, gli elementi di un menu non siano stati cambiati, i risultati di una computazione siano quelli previsti, etc.

Il testing automatizzato richiede di lavorare con due file di dati: un file di dati di base e un file di dati correnti. Durante il rilevamento dei dati, in corrispondenza al punto di verifica si registra l'informazione sugli oggetti di test nel *file dati di base*. Questa informazione è la base rispetto alla quale sono confrontati i dati ottenuti dai test successivi (sulle nuove versioni dell'applicazione), memorizzando poi i risultati dei confronti nel *file dati correnti*. Ogni *punto di verifica fallito* deve essere ulteriormente analizzato e, se necessario, inserito nello strumento di gestione del cambiamento come un *difetto*.

Al termine, tutti i difetti scoperti con i test (sia manuali sia automatizzati), devono essere legati ai relativi requisiti di test. La Figura 10.11 mostra uno strumento che visualizza tutti i difetti in un browser a riga posto nella parte superiore della finestra; il difetto selezionato può essere associato a uno o più requisiti di test. Nell'esempio, due requisiti di test (mostrati nella parte inferiore della finestra) sono stati associati al difetto evidenziato.

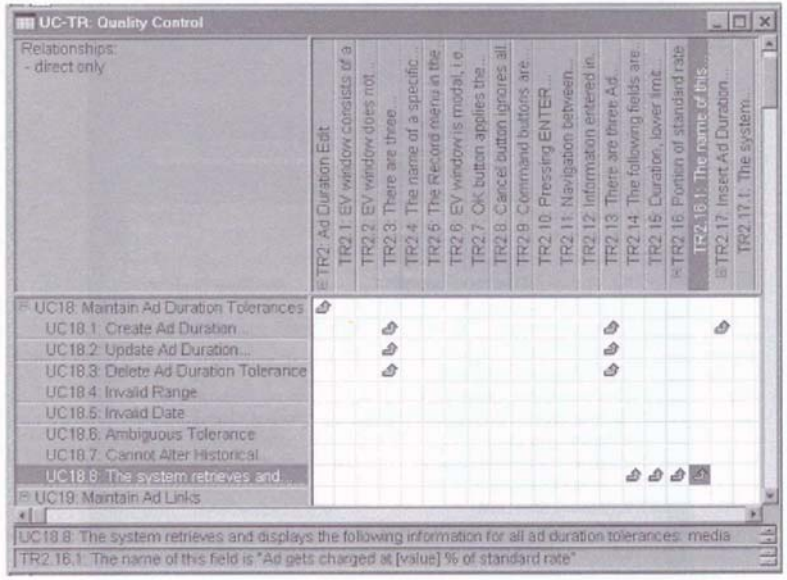


FIGURA 10.10 Tracciabilità da un requisito dei casi d'uso verso un requisito di test (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

10.5.6 Dai requisiti dei casi d'uso alle estensioni

Mentre i difetti sono direttamente tracciati verso i requisiti di test, le estensioni (da implementare in rilasci futuri del prodotto) devono trovare la propria spiegazione nei requisiti dei casi d'uso. In alcune situazioni, nelle quali un difetto è stato trasformato in un'estensione, i legami di tracciabilità tra i requisiti dei casi d'uso e i requisiti di test devono permettere all'utente di risalire dal difetto all'estensione.

La Figura 10.12 dimostra come il medesimo strumento (si veda Figura 10.11) possa essere usato per gestire estensioni e difetti (o qualunque altra richiesta di cambiamento).

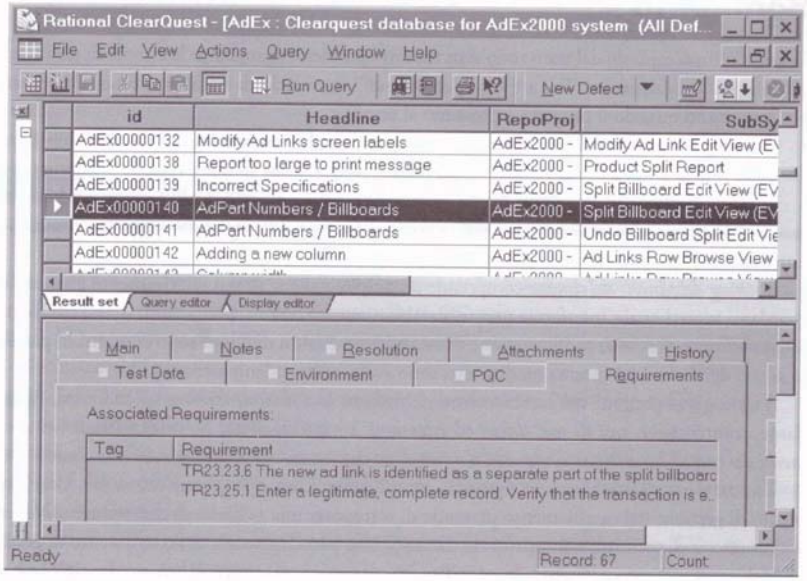


FIGURA 10.11 Tracciabilità da un requisito di test verso un difetto (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

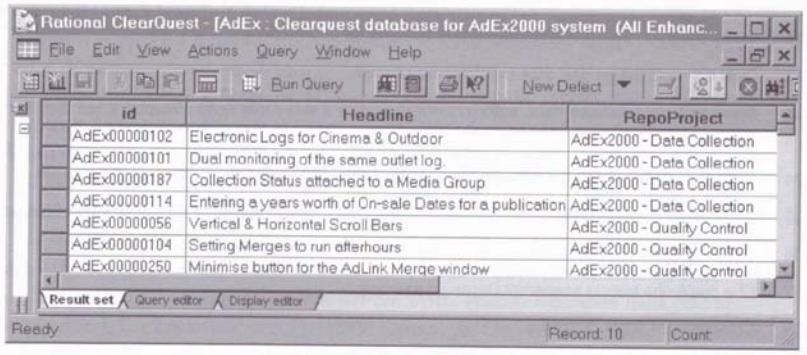


FIGURA 10.12 Estensioni (per gentile concessione di ACNielsen AdEx, Sydney, Australia).

Sommario

In quest'ultimo capitolo del testo sono stati trattati i problemi riguardanti le attività di testing e la gestione del cambiamento. Tali attività spaziano sull'intero ciclo di sviluppo, tuttavia diventano più importanti quando il progetto è prossimo al termine.

Il testing e la gestione del cambiamento suppongono che esistano legami di tracciabilità tra i documenti di sistema e che la validità di tali legami sia assicurata durante tutto lo sviluppo.

Il testing si divide in *testing dei servizi del sistema* e *testing dei vincoli del sistema*. Il testing dei servizi del sistema può essere basato sull'esecuzione (*testing delle specifiche* e *testing del codice*) o non basato sull'esecuzione (*incontri di rassegna* e *ispezioni*).

Il *testing dei vincoli del sistema* comprende un ampio insieme di test relativamente disparati riguardanti aspetti quali: interfaccia utente, database, autorizzazioni, prestazioni, stress, comportamento a fronte di guasti, configurazione e installazione. Alcuni di questi sono condotti in parallelo ai test dei servizi del sistema mentre altri sono svolti indipendentemente.

Il testing e la gestione del cambiamento richiedono una *documentazione* specializzata, riguardante *caratteristiche, casi di test, difetti* ed *estensioni*. I *requisiti di test* sono identificati nei *documenti dei casi di test* e legati ai *requisiti dei casi d'uso* contenuti nei *documenti dei casi d'uso*.

Una *richiesta di cambiamento* riguarda, comunemente, un *difetto* o un'*estensione*. Uno strumento di gestione del cambiamento permette di sottoporre una richiesta di cambiamento, tenendo poi traccia di come gli sviluppatori la trattano. Una parte essenziale dello strumento di gestione del cambiamento riguarda la specifica di cammini di *tracciabilità* tra le richieste di cambiamento e altri documenti di sistema, riguardanti specialmente i requisiti di test e i requisiti dei casi d'uso.

Domande

D1 In che modo un incontro di rassegna si differenzia da un'ispezione?

D2 Qual è il ruolo del gruppo SQA nell'organizzazione?

D3 Che cos'è un database delle autorizzazioni? Qual è il suo ruolo nello sviluppo e nel testing del sistema?

D4 Quali testing dei vincoli del sistema sono strettamente correlati al testing di stress? Spiegare.

D5 Quali testing dei vincoli del sistema sono strettamente correlati al testing d'installazione? Spiegare.

D6 Quali azioni sono possibili a seguito di una richiesta di cambiamento?

D7 Che cos'è una traccia sospetta? Darne un esempio.

D8 Che cos'è un punto di verifica?

D9 Spiegare la differenza tra il file dati di base e il file dati correnti.

Bibliografia

Allen, P. e Frost, S. (1998) *Component-Based Development for Enterprise Systems. Applying the SELECT Perspective*[a], Cambridge University Press, 462 pp.

Angell, I.O. e Smithson, S. (1991) *Information Systems Management. Opportunities and Risks*, Macmillan, 248 pp.

Arthur, L.J. (1992) *Rapid Evolutionary Development. Requirements, Prototyping and Software Creation*, John Wiley & Sons, 222 pp.

Bahrami, A. (1999) *Object Oriented Systems Development*, Irwin McGraw-Hill, 412 pp.

Bennett, S. McRobb, S. e Farmer, R. (1999) Object-Oriented Systems Analysis and Design Using UML, McGraw-Hill, 516 pp.

Bochenski, B. (1994) *Implementing Production-Quality Client/Server Systems*, John Wiley & Sons, 442 pp.

Bollinger, T.B. e McGowan, C. (1991) A Critical Look at Software Capability Evaluations, *IEEE Software*, 4, pp. 25-41.

Booch, G. Rumbaugh, J. e Jacobson, I. (1999) *The Unified Modeling Language. User Guide*, Addison-Wesley, 482 pp.

Bourne, K.C. (1997) *Testing Client/Server Systems*, McGraw-Hill, 572 pp.

Brooks, F.P. (1987) No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Software*, 4, pp. 10-19; ristampato in: *Software Project Management. Readings and Cases* (1997), ed. C.F. Kemerer, Irwin, pp. 2-14.

Buschmann, F. Meunier, R. Rohnert, H. Sommerlad, P. e Stal, M. (1996) *Pattern-Oriented Software Architecture. A System of Patterns*, John Wiley & Sons, 458 pp.

CMM (1995) *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 442 pp.

Coad, P. with North, D. e Mayfield, M. (1995) *Object Models. Strategies, Patterns, and Applications*, Yourdon Press, 506 pp.

Collins, D. (1995) *Designing Object-Oriented User Interfaces*, Benjamin/Cummings Publ., 590 pp.

Conallen, J. (2002) *Costruire applicazioni Web con UML*, Pearson Education Italia, 332 pp.

Constantine, L.L. e Lockwood, L.A.D. (1999) *Software for Use. A Practical Guide to the Models and Methods of Usage-Centered Design*, Addison-Wesley, 579 pp.

Date, C.J. (2000) *An Introduction to Database Systems*, Addison-Wesley, 7ª ed., 938 pp.

Davenport, T.H. (1993) *Process Innovation: Reengineering Work through Information Technology*, Harvard Business School Press, 338 pp.

Davenport, T.H. e Short, J. (1990) The New Industrial Engineering. Information Technology and Business Process Redesign. *Sloan Management Review*, Cambridge, Summer, pp. 11, 17.

Eaglestone, B. e Ridley, M. (1998) *Object Databases*, McGraw-Hill, 380 pp.

Eisenberg, A. e Melton, J. (1999) SQL:1999, Formerly Known as SQL3, *ACM SIGMOD Rec.*, 1, pp. 131-138.

Elmasri, R. e Navathe, S.B. (2000) *Fundamentals of Database Systems*, 3ª ed., Addison-Wesley, 956 pp.

Fowler, M. (1997) *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 358 pp.

Fowler, M. (1999) *UML - Current Status for Version 1.3*, http://ourworld.compuserve.com/homepages/Martin_Fowler/umlst.htm, 2 pp.

Fowler, M. e Scott, K. (2000) *UML Distilled. Guida rapida allo Standard Object Modeling Language*, 1ª ed. it., Addison-Wesley Longman Italia, 177 pp.

Fowler, S. (1998) *GUI Design Handbook*, McGraw-Hill, 318 pp.

Galitz, W.O. (1996) *The Essential Guide to User Interface Design. An Introduction to GUI Design Principles and Techniques*, John Wiley & Sons, 626 pp.

Gamma, E. Helm, R. Johnson, R. e Vlissides, J. (1995) *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 396 pp.

Gray, N.A.B. (1994) *Programming with Class*, John Wiley & Sons, 624 pp.

Hammer, M. (1990) Reengineering Work: Don't Automate, Obliterate, *Harvard Business Review*, Boston, Lug/Ago, p. 104.

Hammer, M. e Champy, J. (1993) *Reengineering the Corporation: A Manifesto for Business Revolution*, Allen & Unwin, 224 pp.

Hammer, M. e Champy, J. (1993) The Promise of Reengineering, *Fortune*, 9, p. 94.

Hammer, M. e Stanton, S. (1993) How Process Enterprises Really Work, *Harvard Business Review*, Boston, Nov/Dic, pp. 108-118.

Harmon, P. e Watson, M. (1998) *Understanding UML: The Developer's Guide. With a Web-Based Application in Java*, Morgan Kaufmann, 368 pp.

Hawryszkiewycz, I. Karagiannis, D. Maciaszek, L. e Teufel, B. (1994) RESPONSE - Requirements Specific Object Model for Workgroup Computing, *Int. J. of Intelligent & Cooperative Information Systems*, 3, pp. 293-318.

Henderson-Sellers, B. (1996) *Object-Oriented Metrics. Measures of Complexity*, Prentice Hall, 230 pp.

Hoffer, J.A. George, J.F. e Valacich, J.S. (1999) *Modern Systems Analysis and Design*, 2ª ed., Addison-Wesley, 854 pp.

Horton, I. (1997) *Beginning Visual C++ 5*, Wrox Press, 1054 pp.

Jacobson, I. (1992) *Object-Oriented Software Engineering. A Use Case Driven Approach*, Addison-Wesley, 524 pp.

Jordan, E.W. e Machesky, J.J. (1990) *Systems Development. Requirements, Evaluation, Design, and Implementation*, PWS-KENT Publ. Company, 648 pp.

Khoshafian, S. Chan, A. Wong, A. e Wong, H.K.T. (1992) *A Guide to Developing Client/Server SQL Applications*, Morgan Kaufmann Publ., 634 pp.

Kimball, R. (1996) *The Data Warehouse Toolkit. Practical Techniques for Building Dimensional Data Warehouses*, John Wiley & Sons, 388 pp.

Kirkwood, J. (1992) *High Performance Relational Database Design*, Ellis Horwood, 266 pp.

Koestler, A. (1967) *The Ghost in the Machine*, Hutchinson, 384 pp.

Koestler, A. (1978) *Janus. A Summing Up*, Hutchinson, 354 pp.

Kotonya, G. e Sommerville, I. (1998) *Requirements Engineering. Processes and Techniques*, John Wiley & Sons, 282 pp.

Kruchten, P. (1999) *The Rational Unified Process*, Addison-Wesley, 256 pp.

Lakos, J. (1996) *Large-Scale C++ Software Design*, Addison-Wesley, 846 pp.

Lee, R.C. e Tepfenhart, W.M. (1997) *UML and C++. A Practical Guide to Object-Oriented Development*, Prentice-Hall, 446 pp.

Lieberherr, K.J. e Holland, I.M. (1989) Assuring Good Style for Object-Oriented Programs, *IEEE Soft.*, 9, pp. 38-48.

Maciaszek, L.A. (1990) *Database Design and Implementation*, Prentice-Hall, 384 pp.

Maciaszek, L.A. (1998) Object Oriented Development of Business Information Systems - Approaches and Misconceptions, *Proc. 2nd Int. Conf. on Business Information Systems BIS'98*, Poznan, Poland, pp. 95-111.

Maciaszek, L.A. (2000) Process Model for Round-Trip Engineering with Relational Database, in: *Challenges of Information Technology Management in the 21st Century, 2000 Information Resources Management Association International Conference*, Anchorage, Alaska, USA, Idea Group Publishing, pp. 468-472.

Maciaszek, L.A. De Troyer, O.M.F Getta J.R. e Bosdriesz, J. (1996a) Generalization versus Aggregation in Object Application Development - the "AD-HOC" Approach, *Proc. 7th Australasian Conf. on Information Systems ACIS'96*, Vol. 2, Hobart, Tasmania, Australia, pp. 431-442.

Maciaszek, L.A. Getta, J.R. e Bosdriesz, J. (1996b) Restraining Complexity in Object System Development - the "AD-HOC" Approach, *Proc. 5th Int. Conf. on Information Systems Development ISD'96*, Gdansk, Poland, pp. 425-435.

Maciaszek, L.A. e Wong, K.S. (2000) UML Dialect for Designing Object-Relational Databases, in: *Challenges of Information Technology Management in the 21st Century, 2000 Information Resources Management Association International Conference*, Anchorage, Alaska, USA, Idea Group Publishing, pp. 473-477.

NEXT (1996) *Next Generation Computing. Distributed Objects for Business*, ed. P. Fingar, D. Read and J. Stikeleather, SIGS Books & Multimedia, 306 pp.

ODMG (2000) *The Object Data Standard: ODMG 3.0*, ed. R.G.G. Cattell, D.K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda e F. Velez, Morgan Kaufmann, 300 pp.

Olsen, D.R. (1998) *Developing User Interfaces*, Morgan Kaufmann Publ., 414 pp.

Page-Jones, M. (2000) *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, 458 pp.

Paradigm (1997) *Paradigm Plus. Round-trip Engineering with Microsoft Visual C++*, Platinum Technology, 36 pp.

Pfleeger, S.L. (1998) *Software Engineering. Theory and Practice*, Prentice-Hall, 576 pp.

Porter, M. (1985) *Competitive Advantage: Creating and Sustaining Superior Performance*, Free Press, 558 pp.

Porter, M.E. e Millar, V.E. (1985) *How Information Gives You Competitive Advantage*, Harvard Business Review, Lug/Ago, pp. 149-161.

Pressman, R.S. (1997) *Software Engineering. A Practitioner's Approach*, 4ª ed., McGraw-Hill, 852 pp.

Quatrani, T. (2000) *Visual Modeling with Rational Rose 2000 and UML*, Addison-Wesley, 256 pp.

Ramakrishnan, R. e Gehrke, J. (2000) *Database Management Systems*, McGraw-Hill, 906 pp.

Rational (1998) *Rational Rose 98. Roundtrip Engineering with C++*, Rational Software Corp., 454 pp.

Rational (2000) *Rational Solutions for Windows*, OnLine Documentation April 2000 Edition, Rational Software Corp.

Riel, A.J. (1996) *Object-Oriented Design Heuristics*, Addison-Wesley, 380 pp.

Robertson, J. e Robertson, S. (2000) *Volere Requirements Specifications Template*, Edizione 6.1, Atlantic Systems Guild, http://www.atlsysguild.com, 52 pp.

Robson, W. (1994) *Strategic Management and Information Systems. An Integrated Approach*, Pitman Publ., 570 pp.

Ruble, D.A. (1997) *Practical Analysis and Design for Client/Server and GUI Systems*, Yourdon Press, 516 pp.

Rumbaugh, J. Blaha, M. Premerlani, W. Eddy, F. e Lorensen, W. (1991) *Object-Oriented Modeling and Design*, Prentice-Hall, 500 pp.

Rumbaugh, J. (1994) Getting Started. Using Use Cases to Capture Requirements, *J. Object-Oriented Prog.*, Sett., pp. 8-10, 12, 23.

Rumbaugh, J. Jacobson, I. e Booch, G. (1999) *The Unified Modeling Language Reference Manual*, Addison-Wesley, 550 pp.

Schach, S. (1996) *Classical and Object-Oriented Software Engineering*, 3ª ed., Irwin, 604 pp.

Schmauch, C.H. (1994) *ISO 9000 for Software Deevelopers*, ASQC Quality Press, 156 pp.

Silberschatz, A. Korth, H.F. e Sudershan, S. (1997) *Database System Concepts*, 3ª ed., McGraw-Hill, 864 pp.

Smith, J.M. e Smith, D.C.P. (1977) Database Abstractions: Aggregation and Generalization, *ACM Trans. Database Syst.*, 2, pp. 105-133.

Sommerville, I. e Sawyer, P. (1997) *Requirements Engineering. A Good Practice Guide*, John Wiley & Sons, 392 pp.

Sowa, J.F. e Zachman, J.A. (1992) Extending and Formalizing the Framework for Information Systems Architecture, *IBM Syst. J.*, 3, pp. 590-616.

SourceProDB (2001): *SourcePro DB. Offering Power and Productivity for C++ Database Applications*, White Paper, Rogue Wave Software (disponibile su www.roguewave.com dal 20 maggio 2001).

SQL (2000) *Database Language SQL - Part 2: Foundation (SQL/Foundation)*, ISO-ANSI Working Draft, Marzo 2000, 1170 pp.

Stein, L.A. Lieberman, H. Ungar, D. (1989) A Shared View of Sharing: The Treaty of Orlando, in: *Object-Oriented Concepts, Databases, and Applications*, ed. W. Kim and F.H. Lochovsky, Addison-Wesley, pp. 31-48.

Stevens, P. e Pooley, R. (2000) *Using UML Software Engineering with Objects and Components*, Addison-Wesley, 256 pp.

Stodder, D. (1997) The Database Dozen, *Database Prog. and Design*, Industry in Focus Issue, Dic., pp. 9-24.

Stonebraker, M. e Moore, D. (1996) *Object-Relational DBMSs. The Next Great Wave*, Morgan Kaufmann Publ., 216 pp.

Szyperski, C. (1998) *Component Software. Beyond Object-Oriented Programming*, Addison-Wesley, 412 pp.

Treisman, H. (1994) How to Design a Good Interface Design, *Software Magazine*, Australia, Agosto, pp. 32-36.

Umar, A. (1997) *Object-Oriented Client/Server Internet Environments*, Prentice Hall, 528 pp.

Whitten, J.L. e Bentley, L.D. (1998) *Systems Analysis and Design Methods*, 4ª ed., Irwin McGraw-Hill, 724 pp.

Windows (2000) *The Windows Interface Guidelines for Software Design*, MSDN Library, CDROM collection, Microsoft.

Wirfs-Brock, R. e Wilkerson, B. (1989) Object-Oriented Design: A Responsibility-Driven Approach, in: *OOPSLA'89 Proceedings, SIGPLAN Notices*, No. 10, ACM, pp. 71-75.

Wirfs-Brock, R. Wilkerson, B. e Wiener, L. (1990) *Designing Object-Oriented Software*, Prentice-Hall, 342 pp.

Wood, J. e Silver, D. (1995) *Joint Application Development*, 2ª ed., John Wiley & Sons, 402 pp.

Yourdon, E. (1994) *Object-Oriented Systems Design. An Integrated Approach*, Yourdon Press, 400 pp.

Zachman, J.A. (1987) A Framework for Information Systems Architecture, *IBM Syst. J.*, 3, pp. 276-292.

Zachman, J.A. (1999) A Framework for Information Systems Architecture, *IBM Syst. J.*, 2/3, pp. 454-470.

Indice analitico

A

accoppiamento
- inter-strato, 315
- intra-strato, 315
- tra classi, 314-320

ActiveX Data Objects, 220
aggiornamento di vista, 302
aggregazione
- di classi, 40, 60, 123-125
- e generalizzazione, 184-186
- e oloni, 187-188
- Ha, 123-124, 183
- modellazione, 181-188
- molti-a-molti, 305
- Possesso Esclusivo, 123-124, 182-183
- Possiede, 123-124, 183
- Raggruppa, 123-124, 184
- semantica, 182-184
- trasformazione
  - *nel modello ODB, 279*
  - *nel modello ORDB, 292*
  - *nel modello RDB, 305-308*
- uno-a-molti, 305

API di Memorizzazione Oggetti, 272
applet Java, 220
Application Programming Interface (API), 193, 256
applicazioni web
- implementazione, 220-221
- rilascio, 222

approccio
- Boundary-Control-Entity (BCE), 172-173
- Boundary-Control-Entity-Database (BCED), 195-196
- guidato dai casi d'uso, 107
- Information System Architecture (ISA), 13
- object-oriented, 8, 19, 23, 24-25
- strutturato, 23-24
- SWOT, 10-11
- Value Chain Model (VCM), 10, 11, 74

architettura client "leggera", 193
architettura client "pesante", 193
architettura software, 191-203
- a strati, 17, 172, 193
- deployment, 202-203
- distribuita, 192
- programmazione del database, 194-196
- strategie di riuso, 197-198

asimmetricità, 41
asserzioni in linguaggio naturale, 88
asserzioni sui servizi, 16, 79, 96
associazione
- attori - casi d'uso, 131
- binaria, 38
- ciclo di, 121
- classi di, 40, 164-167
- derivata, 121, 162
- grado di, 38
- molteplicità, 38
- partecipazione obbligatoria di un oggetto, 39
- partecipazione opzionale di un oggetto, 39
- qualificata, 163-164

tra classi, 37-41
trasformazione
nel modello ODB, 279
nel modello ORDB, 290-292
nel modello RDB, 305-308
unaria (singolare), 38
attività
diagramma, 54-56, 259
modellazione, 52-56
stereotipi, 259
attori, 48, 94
attributi, 35, 58-59
a livello di classe, 45-46
derivati, 161
protetti, 174
statici, 174
visibilità, 35-36

B

blocchi, 331
browser a riga, 246-247
browser ad albero, 247-248
Business Process Improvement (BPI), 12
Business Process Reengineering (BPR), 12

C

cammini di computazione, 55
capability maturity model (CMM), 5-6
cartella a schede, 252-253
casi d'uso
attori, 48
di business, 93-94
diagramma, 49, 98
documento di progetto, 225-226
documento, 350
elaborazione, 224-225
modellazione, 47-52, 130-136
modello, 80, 130
realizzazione, 211-212
specifica, 225-228
tracciabilità, 355-356
caso di studio Acquisti OnLine, 47-72, 215-232
aggregazione, 61
associazione qualificata, 163-164
associazioni derivate, 162
attributi, 58-59
modellazione con statechart, 68-72
modellazione dei casi d'uso, 47-52
modellazione dell'interazione, 64-68
modellazione delle attività, 52-56
modellazione delle classi, 56-63
package dei casi d'uso, 216
progetto dei casi d'uso, 225
progetto dei package, 216-218
progetto delle collaborazioni, 224-232
progetto dei componenti, 219-222
progetto di rilascio, 222-223
caso di studio Database del Personale, 165-167
caso di studio Gestione Contatti con i Clienti (CMS)
cartella a schede, 252-254
diagramma di navigazione nelle finestre, 261-264
finestra di dialogo, 250-251
finestra di messaggio, 254
finestra primaria, 244-246
lista a tendina, 254
navigazione di programma, 328
ricerca degli attributi, 116-117
ricerca delle classi, 111-112
specifica del caso, 74-75
specifica delle associazioni, 121-123
specifica di caso d'uso, 133-134
trasformazione associazioni, 279, 290, 305
trasformazione classi entity, 279, 289, 304
vincoli, 154
caso di studio Immatricolazioni universitarie
aggregazione e composizione, 122-123
associazioni in un ORDB, 287

collaborazione, 211-212
componenti e package, 199-200
descrizione del caso di studio, 73
generalizzazione, 282
realizzazione di operazioni, 213-214
relazione di dipendenza, 171
ricerca degli attributi, 114
ricerca delle classi, 109-110
specifica dei casi d'uso, 132-133
specifica di sequenze, 140-142
specifiche di oggetti, 129
specifiche di operazioni, 143
caso di studio Telemarketing
classi amiche, 159-160, 161, 163
descrizione del caso di studio, 75-76
diagramma dei casi d'uso di business, 94
diagramma di navigazione nelle finestre, 264
diagrammi di contesto, 92-93
ereditarietà attributi, 175
modellazione dei casi di business, 95-96
ricerca attributi, 118-119
riscrittura, 180-181
specifica dei casi d'uso, 136
specifica delle classi, 120
vincoli, 153
visibilità, 156-157
caso di studio Video Store
descrizione del caso di studio, 74
ricerca degli attributi, 115-116
ricerca delle classi, 110-111
specifica dei casi d'uso, 134-135
specifica delle attività, 138
specifica delle generalizzazioni, 127-128
specifica statechart, 145
trasformazione delle generalizzazioni, 282, 292-294, 309-310
chiamate di ritorno, 179-181, 210
chiave esterna in un RDB, 298-299
chiave primaria in un RDB, 298-299
classi
"amiche", 158-160, 174
applicative, 107
architetturali, 197
astratte, 44-45, 67, 174
attributi delle, 35-36
boundary, 56-57, 104, 218
componenti, 181
composte, 181
concrete, 197
control, 56-57, 104, 172
di business, 107
dipendenti, 315-317
dominanti, 201
entity, 56-57, 172
fondamentali, 197
"fragili", 179
in un ODB, 272
incerte, 106
irrilevanti, 106
materializzate, 165-167
mediatore, 169
modellazione delle, 105-119
"mostro", 173
operazioni, 36-37
persistenti, 218
rilevanti, 106
classificazione dinamica, 44, 318-320
clonare, 185
coesione di classi, 314-320
Legge di Demeter, 315
tipologie, 315
Collaborazioni, 204-214
comportamento delle, 212, 231-232
diagramma di, 204-210
messaggi asincroni, 209
messaggi iterativi e tipi parametrici, 208
messaggi "self", 209
notazione e struttura dei messaggi, 205-206
tipi di messaggio, 206-207
e diagramma di sequenza, 210
e operazioni sulle classi, 36-37
progetto delle, 204-214, 224-225
progetto di programmi client/server, 321-324
realizzazione di operazioni, 212-214
struttura delle, 211-212, 231-232
tra oggetti, 31-32, 33
Common Gateway Interface (CGI), 220

Common Object Request Broker Architecture (CORBA), 2
componenti, 198-201
classi e interfacce, 200-201
diagramma, 199
e package, 199-200
progetto, 219-222
composizione
di classi, 40-41, 61, 123-125
modellazione, 181
relazioni di, 123-125
Computer-Assisted Software Engineering (CASE), 7, 9, 85, 224
Computer-Supported Cooperative Work (CSCW), 335
concetto di classe, 106
concetto "l'utente ha il controllo", 18, 23, 238, 240
condivisione anticipata (e non) dell'ereditarietà, 186
consistenza, 241-242
Controlli ActiveX, 220
controllo pessimistico della concorrenza, 331
cookie, 221, 222
costrutti concettuali, 2

D

database
consistenza, 330
package, 218
progetto, 269-312
programmati, 194
ripristino, 330
round-trip engineering, 337-338
schema esterno, 270
schema fisico, 270
schema logico, 270
sicurezza, 302
strato persistente, 269-271
testing, 347
database a oggetti (ODB), 196, 271, 272-274
primitive, 272-277
ereditarietà ISA ed EXTENDS, 276
operazioni predefinite, 276-277
relazioni e relazioni inverse, 274-275
tipo letterale e tipo oggetto, 272-274
trasformare, 277-282
aggregazioni, 279
associazioni, 279
classi entity, 277-278
generalizzazioni, 282
database object-relational (ORDB), 196, 282-294
primitive, 284-288
colonne, campi e attributi, 287
ereditarietà UNDER e OF, 287-288
tabelle di oggetti, 285
tipi riferimento, 286-287
tipi riga, 285-286
tipi strutturati e tipi distinti, 284-285
reingegnerizzazione, 338-340
round-trip engineering, 338-340
trasformare, 289-294
aggregazioni, 290-292
associazioni, 290
classi entity, 289
generalizzazioni, 292-294
database relazionali (RDB), 196, 271
primitive, 295-303
colonne, domini e regole, 296-297
forme normali, 302-303
integrità referenziale, 298-299
tabelle relazionali, 297-298
trigger, 300-301
viste relazionali, 301-302
progetto, 294-310
reingegnerizzazione, 338-340
round-trip engineering, 338-340
trasformare, 303-310
aggregazioni, 305-308
associazioni, 305
classi entity, 304
generalizzazioni, 308-310
delegazione, 181-188
deployment
diagramma, 202
nodi e componenti, 203
notazione, 202

diagramma
dei casi d'uso di business, 94
dei casi d'uso, 49, 98
dei componenti, 199
dei package, 170-171
di attività, 54-56
di collaborazione, 204-210
di contesto, 86, 98
di deployment, 202-203, 222-223
di flusso dei dati (DFD), 23, 86
di navigazione nelle finestre, 261-264
di sequenza, 64-65, 67-68, 210
di workflow, 12
Entity Relationship (ERD), 23
statechart, 71
UML, 357
difetti, 350, 351, 358-359
dimensione del progetto, 4
dipendenza esistenziale, 41, 124
dismissione, 19
Distributed Component Object Model (DCOM), 2
documenti
dei casi di test, 350, 351
del piano di test, 350, 356
delle specifiche, 16, 103
di caso d'uso, 52
di testing, 350-351
organizzativi, 83
studio dei, 83
visione, 93
domande
a scelta multipla, 82
con opinioni, 82
prevenute, 82
domini atomici, 296
dominio, 296
Dynamic Link Library (DLL), 193, 220
elaborazione
di "un insieme alla volta", 321
di "un record alla volta", 321

E

Enterprise JavaBeans (EJB), 2
ereditarietà
del codice, 175
dell'implementazione, 175-181
dell'interfaccia, 174-175
e generalizzazione, 126
e incapsulamento, 174
EXTENDS e ISA in un ODB, 276
modellazione, 173-181
multipla dell'interfaccia, 181, 284
multipla, 43, 181
OF e UNDER in un ORDB, 287-288
per convenienza, 177-178
per estensione, 176
per restrizione, 177
tra classi, 43-44
visibilità delle proprietà ereditate, 157-159
estensioni, 350, 360
estetica, 243
eXtensible Markup Language (XML), 220

F

fallimento dei progetti software, 3-4
fasi del ciclo di vita del software, 15-23
determinazione dei requisiti, 16
implementazione, 18
integrazione, 18-19
manutenzione, 19-20
metriche, 21
pianificazione di progetto, 20-21
progetto architetturale, 17
progetto di dettaglio, 17-18
specifica dei requisiti, 16-17
testing, 22-23
fattibilità, 20
economica, 20
operazionale, 20
tecnica, 20
temporale, 20

file dati di base, 359
finestre, 243-254
di dialogo, 250-251
di messaggio, 254
dipendenze tra, 255-258
figlia, 247, 257
primaria, 243-248
radice, 257
secondarie, 249-254
forme normali in un RDB, 302-303
funzione applicativa, 194
funzioni esterne al contesto applicativo, 49

G

garanzia di qualità del software, 22, 345
generalizzazione, 126-128
di casi d'uso, 131
di classi, 41-44
e incapsulamento, 174
e sostituibilità, 173-174
trasformazione
verso ODB, 282
verso ORDB, 294-295
verso RDB, 308-310
gerarchie
complessità delle, 168, 169
di requisiti, 89
gestione, 19
dei requisiti, 80
del cambiamento, 90, 351-354
di progetto nell'approccio a oggetti, 24-25
glossario, necessità del, 100
granularità nel riuso, 197

I

identificatore oggetto (OID), 32, 185, 205
identificatore unico, 89
identificazione dei requisiti, 88-89
impatto del cambiamento, 90
incapsulamento
ed ereditarietà, 174
nell'aggregazione, 187
nella delegazione, 186
nella modellazione delle classi, 155-160
incontri di rassegna, 22, 344
informazione derivata, 160-162
integrazione incrementale, 18-19
integrità referenziale, 295, 298-299
interazione
applicazione - database, 194
modellazione, 139-142
interfaccia utente, 18
a documenti multipli, 257-258
a documento singolo, 257
configurazione, 242
consistenza, 241-242
ereditarietà, 174-175
estetica, 243
"l'utente ha il controllo", 18, 23, 240
linee guida, 239-243
nativa, 196
natura multidisciplinare, 237-238
navigazione nelle finestre, 258-264
personalizzazione, 242
progetto, 237-267, 313
prototipo e implementazione, 238-239
requisiti, 99
retroazione, 242
Structured Query Language (SQL), 321
testing, 346-347, 348
tolleranza, 242
usabilità, 243
utente grafica (GUI), 18, 23
Internet Server Application Programming Interface (ISAPI), 220
interviste, 81-82
con domande a risposte aperte, 81
con domande a risposte chiuse, 81
non strutturate, 81
strutturate, 81
ISO 9000, 7
ispezioni, 22, 344-345

J

Java Database Connectivity (JDBC), 220, 322
JavaBean, 220
JavaScript, 220
joint application development (JAD), 84-85

L

lavoro cooperativo, 24, 335
legami
persistenti, 32-33
transitori, 33-34
Legge di Demeter nel progetto di programmi, 315
letterale nullo, 274
letterale strutturato, 273
Libreria Database (LibDB), 220
librerie di accesso ai dati, 220
linguaggi di modellazione, 7-9
liste a tendina, 254
livello
di isolamento nelle transazioni, 331-332
operativo, 14
strategico, 14
tattico, 14
logica d'integrità, 194

M

manutenzione
adattativa, 19
perfettiva, 19
matrice di dipendenza dei requisiti, 87
messaggi, 31, 33, 64
asincroni, 209
costruttori, 206
di aggiornamento, 207
di lettura, 207
distruttori, 206
iterativi, 208
notazione, 205-206
"self", 209
sincroni, 209
struttura, 205-206
tipi, 206
metodi d'accesso, 316-317
metriche, 21, 354
Microsoft Foundation Classes, 256
middleware, 17
missione aziendale, 10
modellazione
aggregazione, 61
associazione qualificata, 163-164
associazioni, 60
attributi, 58-59
classe d'associazione, 165-167
classe materializzata, 165-167
con statechart, 68-72
diagramma, 71-72
stati e transizioni, 70-71
delle classi, 56-63
delle interfacce pubbliche, 142-143
dello scopo del sistema, 80, 92-93
generalizzazioni, 62
informazione derivata, 160-162
note e tag, 154
stereotipi, 151-152
vincoli, 152-154
visibilità e incapsulamento, 155-160
modelli
del cambiamento di stato, 8, 46, 103-104
del comportamento, 8, 46, 103-104
delle classi di business, 80
dello scopo del sistema, 91-93, 98
dello stato, 8, 46, 103-104
di business, 80, 91-96
di specifica, 16
moduli e architettura di sistema, 5
moduli e rapporti di sistema, 83

N

navigazione
di programma, 326-329
nelle finestre, 258-264
negoziazione, 86-88
Netscape Server Application Programming Interface (NSAPI), 220
nomi dei ruoli, 35, 39
nominare le classi, 113
notazione per i nodi, 202
note, 154

O

Object Data Management Group (ODMG), 272
oggetti
attore, 68
classe, 30, 45-46, 68
collaborazione, 31-32, 33
collezione nel modello ODB, 274
del business, 81, 104
distribuiti, 222
istanza, 30-34, 45-46
modellazione, 129
notazione, 30-31
persistenti, 32
transitori, 32
olocrazie, 187
oloni e aggregazione, 187-188
On Line Analytical Processing (OLAP), 174
Open Database Connectivity (ODBC), 220, 322
operazione
a livello di classe, 45-46
astratta, 126
CRUD, 142
di creazione, 67
predefinite nel modello ODB, 276-277
sulle classi, 36-37
osservazione
attiva, 83
passiva, 83

P

package, 170-171
annidati, 170
control, 194
dei casi d'uso, 216
di classi, 216, 217-218
diagramma, 170-171
entity, 218
esempio di progetto, 216-218
nel progetto dei componenti, 199-200, 216-219
notazione, 170
pagina web come finestra primaria, 248
personalizzazione nel progetto dell'interfaccia utente, 242
pianificazione di sistema, 9-14
approccio BPR, 12
approccio ISA, 13
approccio SWOT, 10-11
approccio VCM, 11
livelli di gestione, 14
polimorfismo, 42, 126, 175
principio di sostituibilità, 126, 173-174
priorità dei requisiti, 87
problemi aperti, 99-100
procedure memorizzate, 322-323, 334-335
process redesign, 12

processi
incrementali, 5, 24, 47
iterativi, 5, 24, 47
progetto
architetturale, 17, 191, 313
dei componenti, 198-201
del processo client, 314
del processo server, 314
del sistema, 191-235
della persistenza, 324
delle transazioni, 330-335
atomiche, 330
brevi, 331-335
lunghe, 335
programmazione del database, 194-196
programmi client, round-trip engineering, 336-337
programmi guidati da eventi, 23, 24, 238
proprietà private, 156
prototipo
evolutivo, 84, 85
nel progetto dell'interfaccia utente, 238-239
nella determinazione dei requisiti, 84
"usa e getta", 84
punti
di allineamento nel ripristino automatico, 332
di salvataggio nel ripristino programmabile, 333, 335
di verifica, 358

Q

quesiti
a punteggio, 82
con ordinamento, 82
questionari, 82

R

rapid prototyping, 16
Rational Unified Process, 4, 5, 8
realizzazione
dei casi d'uso, 211-212
di operazioni, 212-214
regole di business, 16, 296, 300, 323
reingegnerizzare un RDB in un ORDB, 338-340
relazioni
di composizione, 199
di dipendenza, 159-160
nei componenti, 199
nei package, 170
di realizzazione, 204
Remote Data Objects, 220
Remote Procedure Calls (RPC), 193
requisiti
dei casi d'uso, 83
di manutenibilità, 99
di sicurezza, 99
di usabilità, 99
documenti, studio dei, 83
documento dei requisiti, 79, 96-100
funzionali, 79, 96, 98, 103
fuori dal contesto, 86-87
gerarchie, 89
gestione del cambiamento, 88-90
identificazione e classificazione, 88-89
in conflitto, 87
informativi, 79, 96, 98, 103
interviste, 81-82
JAD, 84-85
legali, 99
matrice di dipendenza, 87
negoziazione e validazione, 86-88
operativi, 99
osservazioni, 83
politici, 99
premesse, 98
principi, 79-80
problemi aperti, 99

prototipo, 84
questionari, 82
raccolta dei, 80-86
RAD, 85
rischi, 81, 87-88
schemi, 97
servizi del sistema, 98
sovrapposti, 87
testing, 355-356
tracciabilità, 90, 357-358
vincoli del sistema, 99
reti, complessità delle, 168-169
retroazione, 242
revisione dei requisiti, 79, 86
revisioni formali, 22
ricerca
degli attributi, 113
delle associazioni, 121
delle classi, 105-113
approccio basato sulle strutture comuni, 106-107
approccio CRC, 108
approccio frase nominale, 106
approccio guidato dai casi d'uso, 107
approccio misto, 108
richieste di cambiamento, 90, 352-354
rinegoziazione, 79
ripristino
automatico, 332
programmabile, 333-334
rischi, 81
nell'ereditarietà dell'implementazione, 178-181
nella raccolta dei requisiti, 87-88
riscrittura
delle operazioni, 42, 175, 179-180
e sovraccarico, 207
riuso
del codice, 175, 197
del software, 186, 197-198
di framework, 197-198
di librerie di classi, 197
di pattern, 198
progettuale, 197
rollback
di trigger, 333-334
nel ripristino automatico, 332
round-trip engineering, 336-340
da RDB a ORDB, 338-340
nei database, 337-338
per i programmi client, 336-337

S

scambio dei puntatori, 33
segnature, 45, 126, 205
semantica dichiarativa, 7-8
servizi del sistema, 98
servlet Java, 220
sistema di database distribuiti, 192
sistemi
a comportamento naturale, 30
complessi, 30
Enterprise Resource Planning (ERP), 2
multimedia, 24
pre-esistenti, 2
sociali, 3
software, studio dei, 83
slittamento dello scopo, 91
software commercial-off-the-shelf (COTS), 2
sottoclasse, 41, 126, 173-181
sovraccarico, 207
specialisti con strumenti avanzati di sviluppo (SWAT), 85
specifica
dei requisiti, 16-17, 103-104
del cambiamento di stato, 144-145
del comportamento, 127, 130-143
attività, 138
casi d'uso, 130-136
interazioni, 139-142
interfacce pubbliche, 142-143
delle associazioni, 121
delle classi, 112-120
dello stato, 104-129

associazioni, 120-123
classi, 105-119
generalizzazioni, 127-128
oggetti, 129
relazioni di aggregazione e composizione, 123-125
stakeholder, 3-4
stati annidati, 71
stato composto, 71
stereotipi
nella modellazione delle classi, 151-152
nella navigazione di programma, 326-327
per la navigazione nelle finestre, 259-261
strategia di soluzione, 17
stratificazione delle classi, 167-173, 314
strato persistente del database, 269-271
strumenti
di modellazione, 7-9
per i test di regressione, 22-23, 358-359
strutture non object-oriented, 271
superclasse, 41, 126, 173-181
sviluppo del software, 1-9
approccio object-oriented, 24-25
approccio strutturato, 23-24
linguaggi e strumenti di modellazione, 7-9
processo, 4-7
stakeholder, 3-4
sviluppo rapido delle applicazioni (RAD), 85

T

tabelle di oggetti, 282-285
tabelle relazionali, 295, 297-298
tag, 154
tecnologia a oggetti, 29-46
aggregazione e composizione, 40-41
associazione, 37-41
classe, 34-37
generalizzazione, 41-45
oggetto classe, 45-46
oggetto istanza, 30-34
tecnologia a componenti, 2
test
informali, 343
metodici, 343
pilota, 346
testing, 343-362
basato sull'esecuzione, 22
documenti, 350-351
gestione del cambiamento, 351-354
servizi del sistema, 343-345
tracciabilità, 354-361
vincoli del sistema, 346-350
autorizzazioni, 348
comportamento a fronte di guasti, 348-350
configurazione, 348-350
database, 347
installazione, 348-350
interfaccia utente, 346-347
prestazioni, 348-350
stress, 348-350
timeboxing, 85
tipi
distinti in un ORDB, 284-285
letterale in un ODB, 272-273
oggetto in un ODB, 272-273
oggetto strutturato, 274
parametrici, 40
riferimento in un ORDB, 286-287
riga in un ORDB, 285-286
strutturati in un ODB, 272-282
strutturati in un ORDB, 284-285
tolleranza, 242
tracce sospette, 355
tracciabilità, 90, 354-361
diagrammi UML, 357
difetti, 358-359
estensioni, 360
piani di test, 356
requisiti dei casi d'uso, 355-358
requisiti di test, 355-358

transazione atomica, 330
transitività, 41
trigger
  in un RDB, 300-301
  nei database, 194
  nella collaborazione client/server, 323
  rollback, 334

U

Unified Modeling Language (UML), 8
  diagrammi, 357
  documenti di test, 357
usabilità nel progetto dell'interfaccia utente, 243

V

validazione, 86-88
VBScript, 220
vincoli
  asserzioni, 16, 79, 96
  nella modellazione delle classi, 152-154
visibilità
  degli attributi, 35-36
  delle operazioni, 37
  delle proprietà ereditate nelle classi, 157
  negli strumenti CASE, 155
  nella modellazione delle classi, 155-160
  privata, 36
  protetta, 156-157
  pubblica, 35-36
viste relazionali, 301-302

Leszek A. Maciaszek

Sviluppo di sistemi informativi con UML

Analisi dei requisiti e progetto di sistema

Lo sviluppo di un sistema informativo comprende tre fasi iterative ed incrementali: analisi, progetto e implementazione. Questo testo pone particolare enfasi sui metodi e le tecniche usati nelle fasi di analisi e progetto, considerando i problemi d'implementazione solo per quel che può essere rilevante nella fase di progetto. Il testo è orientato allo sviluppo object-oriented del software, con l'uso di UML (Unified Modeling Language) per specificare i risultati della modellazione. L'approccio didattico è basato sul principio "insegnare attraverso gli esempi", e i concetti più importanti sono introdotti tramite casi di studio.

Obiettivo del testo è lo sviluppo object-oriented di "grandi" sistemi informativi. L'architettura software è basata su una soluzione client/server, dove il client è una stazione di lavoro dotata di un'interfaccia grafica e il server contiene un database, che può essere relazionale, object-relational, o object-oriented puro.

Vengono analizzate le tecniche per:

- integrare i modelli di analisi e di progetto
- dominare la complessità di grandi sistemi
- migliorare le architetture software
- promuovere la strutturazione stratificata degli oggetti
- comprendere le conseguenze di scelte inaccurate di modellazione
- condurre i test di progetto e gestire i cambiamenti
- costruire sistemi scalabili e di facile manutenzione

Il testo è orientato a corsi di studio introduttivi di informatica o di sistemi informativi, e potrebbe essere d'interesse sia per discipline come analisi e progetto di sistemi, ingegneria del software, basi di dati e tecnologie ad oggetti, sia nello sviluppo di applicazioni reali. I contenuti, infatti, sono stati organizzati pensando anche a professionisti impegnati nello sviluppo di sistemi informativi aziendali.
Al volume è associato un sito web con un set completo di materiale aggiuntivo disponibile a docenti, studenti e professionisti.

■ *Leszek A. Maciaszek insegna alla Macquarie University a Sydney. Nato in Polonia, il professor Maciaszek ha lavorato come ricercatore e consulente in molti Paesi, tra cui Stati Uniti, Germania, Francia, Italia, Olanda, Macao, Malesia, Thailandia e Kuwait. Dal 1985 si è trasferito in Australia dove ha maturato un'ampia esperienza come consulente e formatore in molte aziende multinazionali. Le sue ricerche e le sue pubblicazioni (quasi cento) riguardano il settore dei database, la tecnologia object-oriented, l'ingegneria del software e lo sviluppo di sistemi informativi.*

Pearson Education Italia

ISBN 88-7192-138-0

9 788871 921389

€ 35,12
Lire 68.000