Capitolo 8

Le classi nei processi

La struttura esterna di un sistema ne determina l'affermazione iniziale,
mentre l'organizzazione interna ne sancisce il successo nel tempo
GRADY BOOCH

Il modo migliore per determinare il fallimento di un progetto è codificare a partire
direttamente dai casi d'uso… Con ciò, ovviamente,
non si intende dire che i Casi d'Uso non vadano prodotti.
LVT

Introduzione

Il presente capitolo è focalizzato sull'utilizzo del formalismo dei diagrammi delle classi nel contesto dei processi di sviluppo del software (modelli a oggetti del dominio del business, di analisi e di disegno). A tal fine sono presi in considerazione i processi più formali: potrebbe non avere molto senso riferirsi a processi in cui la modellazione è considerata l'ultimo degli espedienti.

Coerentemente all'impostazione del libro, la presentazione dei vari modelli avviene sia da un punto di vista teorico, sia pratico attraverso l'utilizzo di un esempio concreto: un ipotetico sistema di gestione della sicurezza di un sistema Internet/Intranet.

Uno dei punti focali del processo di costruzione dei modelli a oggetti consiste nell'identificazione delle classi — e quindi degli oggetti — che costituiscono il sistema. Si tratta di una delle attività più complicate in assoluto della modellazione OO, che implica, essenzialmente, due capacità distinte: scoperta e invenzione. La prima si utilizza maggiormente durante la produzione dei modelli OO appartenenti alle prime fasi dei processi di sviluppo del software (modelli a oggetti del dominio e di business). In questi modelli è necessa-

rio focalizzare l'attenzione sulle entità realmente esistenti nell'area oggetto di studio, al fine di realizzare una versione OO dell'area business e quindi del vocabolario del sistema. L'attività di "invenzione" invece diventa più frequente durante le fasi di analisi e disegno, dove agli oggetti propri del domino è necessario aggiungere quelli di "infrastruttura" atti a realizzare i meccanismi che permettono al sistema di funzionare. Spesso non si tratta di una vera e propria attività di invenzione, bensì di riconoscimento di pattern, sia derivanti da soluzioni applicate nella realizzazione di precedenti sistemi funzionanti — in ultima analisi uno dei vantaggi dell'OO dovrebbe essere relativo alla riutilizzabilità del codice — sia dalla classificazione standard. Inoltre, il ricorso ad architetture "standard" come EJB e .net semplifica la realizzazione del modello di disegno: i pattern da applicare per realizzare l'infrastruttura sono ben definiti in partenza.

Per quanto concerne strettamente l'attività di scoperta, viene presentata una tecnica dimostratasi efficace nel processo di produzione del modello del dominio. Si tratta tuttavia di un'attività molto complessa legata alle capacità dei singoli, che solo l'esperienza e la conoscenza dell'area oggetto di studio possono rendere meno difficile.

Un aspetto al quale si conferisce particolare importanza è relativo all'influenza esercitata dall'architettura del sistema sui vari modelli da produrre. Già durante la fase di analisi dei requisiti, l'architettura stabilita per il sistema recita un ruolo molto importante: permette di stabilire la fattibilità dei requisiti, fornisce suggerimenti su come modellare le richieste dell'utente in modo da farle risultare più confacenti all'architettura del sistema e così via. Anche per questo motivo è opportuno iniziare a redigere il *Software Architecture Document* (documento sull'architettura software) fin dalle prime fasi del processo di sviluppo del software. Avanzando nelle varie fasi di cui è composto il processo di sviluppo del software, l'influenza dell'architettura diviene legittimamente più pressante: fin dalla fase di analisi è importante che il relativo modello ne rifletta l'organizzazione.

Il capitolo si chiude tentando di rispodere all'interrogativo "quando un modello di disegno può considerarsi ben progettato?": una delle domande più classiche e spinose dell'intero processo di sviluppo del software.

Perché disegnare

Sebbene il formalismo dei class diagram si presti a svariati utilizzi, la sua notorietà — anche per ragioni storiche: inizialmente non erano previsti molti altri utilizzi — è indubbiamente legata alla vista di disegno: l'associazione mentale tra i due concetti è piuttosto immediata per quasi tutti i tecnici. In tale ambito i diagrammi delle classi sono utilizzati per rappresentare l'organizzazione interna del sistema (struttura statica del codice) in termini di classi. Dunque si tratta di una vista a forte connotazione tecnica, nella quale l'attenzione viene definitivamente spostata dal "cosa" costruire al "come": ci si concentra sulla progettazione delle soluzioni necessarie per realizzare i requisiti funzionali, descritti dal modello dei casi d'uso, e quelli non funzionali catturati, inizialmente, nel relativo documento.

Al momento in cui viene scritto il presente libro, il modello di disegno è oggetto di un forte contenzioso tra due contrastanti correnti di pensiero… *pardon*, di implementazione. Sebbene il contenzioso sia più che altro relativo ai processi di sviluppo (XP, RUP, EUP, ecc.), lo UML finisce inevitabilmente per esserne investito direttamente in quanto strumento universalmente adottato per modellare gran parte dei manufatti richiesti dai processi a maggior grado di formalità. In particolare, un elemento di forte contrasto è proprio relativo alla necessità o meno di produrre un modello di disegno di cui il formalismo dei diagrammi delle classi rappresenta la parte più rilevante. Agli estremi dell'infuocato dibattito si collocano, rispettivamente, tecnici che non scriverebbero assolutamente una riga di codice senza aver prima realizzato un modello completo e dettagliato, e quelli che invece considerano la modellazione una perdita di tempo prezioso sottratto alla fase di codifica.

L'autore del libro considera la modellazione un'attività di fondamentale importanza nella produzione di un qualsiasi sistema software, sia nel caso in cui il processo utilizzato sia molto accademico, sia qualora si utilizzi un processo più "leggero". Ciò però non significa che sia necessario disegnare ogni dettaglio prima di avviare la fase implementativa.

Lavorando in diverse organizzazioni, dislocate non unicamente sul suolo italiano, è facilmente riscontrabile come l'attività di modellazione di sistemi software sia territorio colonizzato da miti, false credenze e guerre di religione.

Non è certo il male peggiore del mondo utilizzare saltuariamente l'attività di codifica come un valido strumento di investigazione, magari per aumentare il livello di comprensione di una nuova tecnologia, o per verificare se uno specifico strumento (per esempio una libreria fornita da un'azienda esterna) sia in grado di garantire i requisiti non funzionali richiesti. Tale approccio diviene veramente efficace qualora i riscontri ottenuti non restino circoscritti a una porzione di codice. È invece opportuno riesaminarli nella cornice di un modello globale che tenga conto anche degli aspetti più generali dell'intero sistema e delle relative evoluzioni.

L'affermazione che la modellazione costituisce unicamente uno spreco di tempo sottratto alla codifica spesso rappresenta il vano tentativo da parte di alcuni presunti tecnici di celare eventuali inadeguatezze. È sempre molto facile e ordinario denunciare ciò che non si comprende appieno come inutile… Ahimè, tra la possibilità di investire tempo e fatica per comprendere nuove tecnologie emergenti e quella di dichiararle inutili, la seconda alternativa tende a prevalere. La scusa ricorrente è che "modellare equivale a produrre documentazione superflua". Chiaramente un modello realizzato con strumenti formali costituisce già di per sé un'ottima documentazione, ma è un po' difficile considerare documentazione adeguata un diagramma disegnato su una lavagna, sebbene sia a tutti gli effetti un valido modello, se non altro per la difficoltà di inserire la lavagna in un apposito raccoglitore…

L'errata convinzione che la modellazione di un sistema sia una perdita di tempo è tipicamente diffusa tra i programmatori junior. Ciò costituisce un vero peccato sebbene comprensibile. In effetti, poiché la loro esperienza è focalizzata nella scrittura del codice, è abbastanza naturale attendersi tali conclusioni. Quando ciò si verifica, chiaramente le responsabilità maggiori sono ascrivibili a coloro che avrebbero dovuto favorire la crescita di professionalità del personale apprendista… (A questo punto la vocina presente nell'autore sta ridendo smodatamente… e si sta ricorrendo a un eufemismo). Quando poi si tratta di una convinzione radicata nella mente del personale tecnico "esperto", architetti e/o dirigenti, allora il problema diventa veramente serio. In generale, frasi del tipo "il sistema descritto nel libro o studiato all'Università è unicamente appartenente al mondo delle idee", oppure "il sistema prodotto da tizio è un esercizio di laboratorio che non ha nulla a che vedere con la realtà" dovrebbero incutere molto timore proprio nel personale meno esperto.

Indubbiamente, nel modo di produrre software c'è qualche cosa che non funziona propriamente… Come mai la maggior parte dei progetti fallisce, o viene concluso con un tempo e una spesa molto superiori a quanto stimato? Perché si assiste a progetti più volte sospesi, analizzati e poi riavviati o addirittura terminati prematuramente? Le stime parlano chiaro: negli USA, al momento in cui viene redatto il presente capitolo, solo il 15% dei progetti vengono consegnati e con una tolleranza accettabile delle stime iniziali. (Nel Belpaese, probabilmente le statistiche produrrebbero esiti opposti: se un progetto richiede molto più tempo di quello inizialmente pianificato, è sufficiente cambiare la pianificazione iniziale. Come dire: se il risultato non soddisfa, poco male, è sufficiente cambiare i dati di ingresso).

La sindrome "da morte prematura dei progetti" potrebbe essere imputata a responsabilità magari legate al sistema operativo che non prevede tutti i servizi occorrenti, oppure al compilatore che non compila abbastanza efficacemente e che non individua errori logici, o ancora alla libreria acquistata che non fornisce correttamente tutti i servizi promessi, o addirittura al linguaggio di programmazione non adeguato. Cosa dire poi del personale? Forse in tutti questi pretesti potrebbe albergare un fondo di verità, ma chiunque è in grado di capire che, tra tutti gli elementi citati, non esiste quello determinante: la prova schiacciante (*the body of evidence*). Probabilmente il motivo principale risiede nel frequente modo caotico di produrre software, dominato dalle sole attività di codifica e test. Si codifica scrivendo codice alla velocità della luce e poi si comincia a correggerlo a colpi di *debugger* (chiaramente, in questa seconda attività, il termine di paragone è quello dei tempi dei poveri bradipi).

Tipicamente è più difficile non sentirsi a proprio agio nell'attività di codifica. Non si tratta assolutamente di un'attività banale, ma lì, perlomeno, i problemi sono circoscritti a quelli evidenziati dal compilatore oppure ad alcuni a carattere logico che, con il debugger alla mano, possono essere risolti. Poco importa però se poi il sistema nel suo complesso tende a vacillare perché basato su una miriade di decisioni a corto respiro. Come asserisce

Martin Fowler, "le decisioni a corto raggio (*short-term decisions*) funzionano piuttosto bene finché il sistema è minimo, ma, come inizia a crescere, diventa difficile aggiungere nuove funzionalità, e i temutissimi bug cominciano a proliferare come funghi, sia in numero, sia in complessità. Caratteristica tipica di sistemi basati su decisioni a corto-respiro, è il lungo periodo di tempo richiesto dalla fase di test prima di dichiarare il sistema completo".

Ciò però non significa che sia necessario sospendere l'attività di codifica fino a che la lunga ed estenuante attività di modellazione non sia giunta a evidenziare anche i dettagli minimi dell'intero sistema. Come al solito è necessario trovare il giusto mezzo in funzione di diversi parametri, non ultimi il proprio team di sviluppatori e la tipologia del progetto. Probabilmente non è necessario disegnare ogni singola classe e ogni metodo (a patto di non lavorare con i soliti programmatori O*bject Disoriented*), così come non è indispensabile realizzare un diagramma di interazione per ogni scenario; è sufficiente concentrarsi sugli elementi più importanti, conferendo una certa libertà al team di sviluppo. Confinare elementi del proprio team al mero ruolo di "riempitori" di classi non è un'attività salutare per nessuno.

I modelli sono poi di importanza vitale durante la progettazione in cui una delle parole chiave deve essere *comunicare* e quindi anche solo questo motivo dovrebbe giustificare l'utilizzo dello UML.

Anche la documentazione, per quanto possa risultare odiosa, è molto importante e andrebbe realizzata prima, durante e dopo lo sviluppo di ciascuna porzione del sistema. Lo stesso modello di disegno evolve da versioni iniziali dette di specifica a quelli finali in cui rappresenta esattamente il codice e pertanto viene detto di implementazione. Chiaramente non è sufficiente produrre la documentazione, ma è necessario mantenerla aggiornata, tenendo presente il monito che alla fine "solo il codice è sincronizzato con sé stesso" (Scott Ambler).

Realizzare opportuni modelli prima di concentrarsi nell'attività di codifica, magari semplicemente discutendoli con l'ausilio di una lavagna, può aumentare enormemente la produttività dei gruppi: è possibile esplorare le diverse possibili soluzioni prima di codificare, non solo nell'ottica del singolo problema, ma anche nel contesto degli aspetti generali, attuali e futuri, del sistema. Durante l'attività di disegno è possibile scoprire che determinate idee o approcci sono errati o non completamente idonei. Chiaramente tali investigazioni, eseguiti al tempo di disegno, possono costare qualche giorno di lavoro, mentre cambiare direzione in fase di codifica potrebbe costare mesi o addirittura pregiudicare l'intero sistema. In questi casi, accorgendosi di aver sbagliato soluzione, spesso, invece di fermare il tutto ed effettuare una salutare operazione di refactoring, si assiste alla programmazione per "toppe successive", mentre il destino del "povero" sistema comincia a vacillare e ogni nuova funzionalità appressa la luttuosa fatalità.

Generalmente, si tenta di attribuire tutta la colpa agli utenti e ai temutissimi *change requirements* (cambiamenti dei requisiti). Molti vorrebbero risolvere il problema sempli-

cemente non ponendoselo proprio, ossia pretendendo di "congelare" i requisiti utente prima di procedere alle fasi successive: si tratta dell'ennesima "dolce" illusione. Umanamente non è possibile pensare tutto a priori e in modo assolutamente corretto. Proprio queste considerazioni dovrebbero far comprendere quanto modellare, progettare tenendo conto che le cose possano cambiare — non tutte, ovviamente — sia una sfida alla quale non è possibile sottrarsi. L'eterna variazione dei requisiti utente non può e non deve essere motivo per non analizzare e modellare con la scusa che "tanto le cose comunque cambieranno", bensì deve essere un terreno di cimento per realizzare sistemi migliori e più flessibili, in cui le modifiche diventino facilmente pianificabili. In questi contesti la mancata modellazione può portare alla realizzazione di sistemi in cui ogni modifica è fonte di estrema paura da parte del team, e allora cosa fare? Opporsi, opporsi con tutte le forze alle nuove richieste anche a costo di realizzare un sistema non a dimensione utente. Qualora poi le nuove specifiche non siano assolutamente inevitabili, non resta che ricorrere a tutti gli scongiuri possibili e immaginabili e rassegnarsi a modificare, con la speranza che gli aggiornamenti apportati a una parte del sistema non si ripercuotano negativamente proprio dove uno non se lo aspetterebbe.

La modellazione è la chiave di volta: si tratta di un ottimo strumento di investigazione e di comunicazione sia tra i membri del team, sia tra diversi team (requisiti, architettura, sviluppo, ecc.). Chiaramente è un'attività difficile, ad alto grado di creatività, legata all'esperienza dei singoli che poco spazio concede all'improvvisazione: tutto vive nel regno delle regole formali dell'OO. Le difficoltà che si incontrano, quindi, non sono tanto dovute all'UML, quanto alla complessità intrinseca del processo di disegno e di costruzione di sistemi che funzionano. È necessario essere consapevoli che per disegnare proficuamente un sistema OO o component-based, è assolutamente indispensabile possedere una buona padronanza della materia e tanta, tanta esperienza. Esperienza che si ottiene con anni dedicati allo studio e allo sviluppo di sistemi OO funzionanti. Purtroppo, esperienze maturate in settori diversi non sono di grande aiuto. Quando si intraprende il viaggio verso un "nuovo" mondo — un "nuovo" modo di pensare le astrazioni — è necessario modificare i propri schemi mentali iniziando dalla base: la classe.

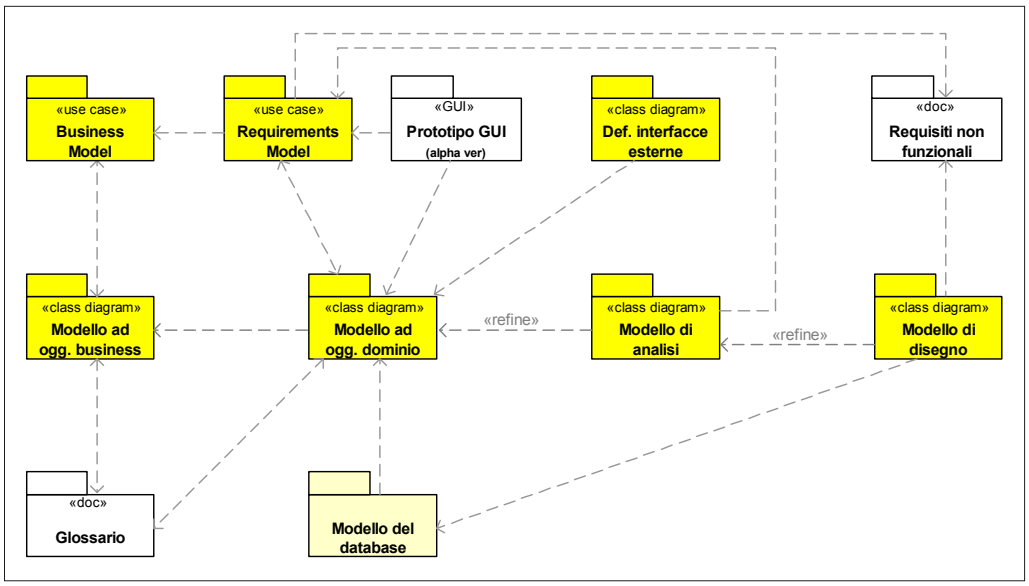
Tutto ciò è proprio così… Se vi pare.

Processi di sviluppo e modelli a oggetti

Il formalismo del diagramma delle classi si presta a essere utilizzato per produrre tutta una serie di manufatti (*artifacts*) coinvolti nel processo di sviluppo del software: dal modello a oggetti del dominio a quello del business, dal modello a oggetti di analisi a quello di disegno e così via (fig. 8.1).

La descrizione dettagliata dei vari manufatti è oggetto di studio dell'intero capitolo. Come già menzionato più volte, è opportuno ribadire che tutti i modelli basati sui diagrammi delle classi sono comunemente designati come modelli a oggetti (*object model*).

Figura 8.1 — *Frammento dei manufatti richiesti da un processo di sviluppo del software focalizzando l'attenzione a quelli realizzati per mezzo del formalismo dei diagrammi delle classi. La notazione cromatica indica la misura in cui il relativo artifact si presti a essere realizzato in UML. In particolare il giallo intenso indica che il manufatto si presta a essere prodotto interamente attraverso i diagrammi previsti dal linguaggio, mentre il grigio ne esclude quasi completamente la presenza.*



Ciò potrebbe lasciare perplesse molte persone, in quanto essendo gli oggetti istanze delle classi si corre il rischio di mettere insieme astrazioni con i relativi corpi. In ogni modo questa è la convenzione generalmente adottata.

Modello a oggetti del dominio

Un manufatto di estrema importanza nei processi di sviluppo del software, prodotto durante la fase di analisi dei requisiti utente, è costituito dal modello a oggetti del dominio (*Domain Object Model*), indicato più semplicemente con il nome "modello del dominio".

> L'autore preferisce ricorrere alla prima versione (eventualmente utilizzando l'acronimo DOM) al fine di evitare possibili confusioni in quanto la denominazione di modello del dominio include diversi manufatti, quali il modello a oggetti, gli use case, le business rule, ecc.

Il modello a oggetti del dominio, come suggerito dal nome, rappresenta un modello che

ha a che fare con entità "realmente esistenti" nell'area oggetto di studio che il sistema software, in qualche misura, dovrà automatizzare. Uno degli scopi primari del modello è contribuire alla piena comprensione del contesto (spazio del problema) del sistema focalizzando l'attenzione sulla relativa struttura statica.

Non è infrequente riferirsi allo spazio del dominio del problema con l'espressione "mondo concettuale" in quanto, generalmente, sono presenti "oggetti" che, pur non esistendo apertamente nel mondo reale, da esso "traspirano" oppure possono essere visti come derivati da specifici oggetti attraverso lo studio delle strutture e dei comportamenti.

Il modello a oggetti del domino è un modello statico (in quanto proiezione indipendente dal fattore tempo) dello spazio del problema che ne rappresenta un'astrazione in termini delle varie entità presenti corredate dalle relative interconnessioni. Le entità individuate sono ottime candidate di classi persistenti, ossia di classi i cui oggetti necessitano di essere memorizzati in forma permanente: in effetti sono le astrazioni dei dati trattati dal sistema. Chiaramente le classi entità individuate in questa fase sono in uno stato embrionale, quindi necessitano di ulteriori elaborazioni per assumere la forma finale utilizzata nel modello di disegno e pronta per essere implementata. Secondo le conclusioni dei Tres Amigos riportate nel libro *The Unified Software Development Process* ([BIB08]), le entità presenti nel modello del dominio dovrebbero appartenere essenzialmente alle seguenti categorie:

- entità "business", ossia oggetti rappresentanti entità (più o meno tangibili) manipolati nell'area business oggetto di studio. In un sistema per il commercio elettronico, queste entità rappresentano oggetti come: articoli, carrelli della spesa, cataloghi, ecc.;

- oggetti e concetti appartenenti al mondo concettuale che il sistema necessita di manipolare. Sempre nel caso del sistema per il commercio elettronico: regole di sconto, segnalazioni del verificarsi di condizioni richieste dall'utente (prodotto in sconto, prezzo dimezzato, ecc.);

- eventi che possono verificarsi, come l'aggiunta di un prodotto al carrello della spesa, la notifica a un utente di una promozione relativa a uno specifico articolo, ecc.

Nel modello del sistema teatrale presentato nel Capitolo 2, esempi di classi appartenenti al dominio del problema sono: `Spettacolo`, `Rappresentazione`, `Biglietto`, `Posto`, `Listino prezzi` ecc. (In sostanza tutte le classi riportate sono dei validi esempi, considerato che si tratta di un modello a oggetti del dominio.) In un sistema di prenotazioni di biglietti di voli aerei, esempi di classi appartenenti al modello del dominio sono: `Aeroporti`, `Aeromobili`, `Biglietti`, `Tratte`, `Vettori`, `Classi di servizio`, ecc.

Una caratteristica tipica dei modelli a oggetti del dominio consiste nello specificare poche o nessuna operazione nelle varie classi. Spesso si investe molto tempo nel tentativo di dettagliare le proprietà comportamentali delle classi (definizione dei metodi) e non sempre è il caso di investire del tempo in questa attività.

Probabilmente è opportuno attendere fasi più mature del processo di sviluppo nelle quali si ha una maggiore comprensione del dominio del problema. In questo stadio si è interessati essenzialmente all'organizzazione dei dati che, al livello di classi, si traduce nella relativa struttura: attributi e relazioni con le altre classi. Invece di specificare gli attributi, si potrebbe definire tutta una serie di metodi `get/set` (come per i componenti entità nell'architettura J2EE). Limitatamente a questo contesto, si tratterebbe unicamente di una perdita di tempo che tra l'altro renderebbe il modello decisamente meno leggibile, specie per coloro che dovranno rivedere il modello. Questi, tipicamente, sono esperti della area di business senza troppa conoscenza del mondo OO.

I lettori con esperienza della metodologia dell'analisi strutturata potranno constatare molte analogie tra il modello concettuale entity-relationship e il modello del dominio del problema di cui quest'ultimo rappresenta l'evoluzione OO.

Il modello del dominio deve essere prodotto in fasi iniziali del processo di sviluppo sia perché permette di bilanciare i casi d'uso rispetto la struttura dei dati, sia perché fornisce il punto di partenza per la realizzazione di tutta una serie di manufatti, non ultimi il modello dei componenti (come si vedrà, in genere ognuno di essi agisce su un opportuno sottoinsieme), il prototipo dell'interfaccia utente, ecc.

Considerando l'utilizzo che viene fatto del modello a oggetti del dominio all'interno del ciclo di vita del sistema e tutta la serie di manufatti che lo utilizzano, si può facilmente comprendere come esso possa essere considerato una colonna portante dell'intero sistema.

Il modello a oggetti del dominio, una volta terminato, rappresenta un'asserzione dello spazio del problema o, se si desidera, il compendio delle regole di business relative all'organizzazione dei dati manipolati. Per esempio, quando si afferma che un "`Trade` genera diversi messaggi" (ossia che le istanze della classe `Trade` sono associate a diversi oggetti `Message`) o che ogni "`Trade` è assegnato a un solo `Book` in funzione della valuta principale", non si fa altro che modellare le regole del business. Il DOM quindi è sicuramente un manufatto assolutamente necessario allo sviluppo del sistema, ma costituisce (come gran parte dei modelli) anche un prodotto finale, un "delivery" del processo. Si presta ad essere utilizzato per eventuali future reingegnerizzazioni (tipicamente la tecnologia è mol-

to variabile, il business no), per la progettazione di sottosistemi aggiuntivi, per l'illustrazione del business magari a nuovi dipendenti, ecc.

La produzione del modello del dominio è un'attività molto importante e sensibile per il buon avanzamento dell'intero processo che, teoricamente, dovrebbe essere realizzata dai business analyst. Considerata l'importanza del manufatto, probabilmente è opportuno che il modello venga, per così dire, elaborato con la continua presenza di personale tecnico con esperienza di modellazione. Una buona idea è ricorrere alla collaborazione di un architetto, sia per elevare il grado di formalità del modello stesso, sia per coinvolgere nella definizione dello spazio del problema persone che poi dovranno realizzare il sistema basato sul modello stesso.

Nei paragrafi seguenti viene presentata una tecnica pratica rivelatasi utile per la produzione del modello a oggetti del dominio del problema, corredata da tutta una serie di linee guida a carattere generale frutto dell'esperienza dell'autore. Tutti coloro con esperienza nella produzione di modelli entità-relazioni (*Entity-Relationship Diagram*) noteranno che è necessario utilizzare le stesse capacità: in ultima analisi si tratta di eseguire la medesima attività con un formalismo diverso.

Presentazione dello spazio del problema

Premessa

In questo paragrafo è descritto un dominio utilizzato come caso di studio per i vari esempi mostrati nel corso dell'intero capitolo. La difficoltà consiste nell'individuare un'area di studio che sia in qualche modo comune al maggior numero possibile di lettori e pertanto possa configurarsi come una piattaforma costruttiva di "monologo dell'autore". Il problema è che raramente accade di lavorare in sistemi atti ad automatizzare ambiti di pubblico dominio. In ogni modo si è deciso di presentare un esempio tratto dal mondo della sicurezza con la speranza che la maggior parte dei lettori vi sia entrata in contatto (in effetti si tratta di un'infrastruttura che dovrebbe essere presente in tutti i sistemi). In particolare l'attenzione è focalizzata sulla descrizione di un sottosistema che da un lato automatizzi il *workflow* necessario per fornire agli utenti del sistema le credenziali per accedervi e dall'altro garantisca il rispetto delle politiche di sicurezza (autenticazione, autorizzazione, auditing degli utenti, incidenti di sicurezza, ecc.). Il contesto in cui viene analizzata l'infrastruttura di sicurezza è relativo a un sistema in tecnologia web di un'azienda multinaziona-

le con sedi sparse in tutto il mondo che utilizzi la risorsa Internet come rete di connessione globale condivisa.

> La presentazione riportata successivamente viene arricchita durante l'illustrazione dell'intero processo. Si è deciso di non aggiornarla direttamente al fine di simulare il più possibile un caso reale in cui, volenti o nolenti, l'aumento della comprensione del dominio avviene procedendo nelle fasi del processo di sviluppo del software.
> Considerati i fini del libro, l'attenzione è circoscritta al sistema software, tralasciando considerazioni relative alla parte, pur molto importante, dell'infrastruttura hardware (proxy, firewall, ecc).
> La descrizione è stata riportata con l'obiettivo di disporre di un'area di studio da utilizzare per mostrare le diverse tipologie di modelli a oggetti presenti nelle varie fasi di un tipico processo di sviluppo del software e non per fornire un "white paper" sulla sicurezza. Quindi l'obiettivo è fissare un dominio di riferimento, indipendentemente dalla validità delle business rule riportate. In ogni modo eventuali suggerimenti provenienti da esperti della particolare area saranno ben accolti.

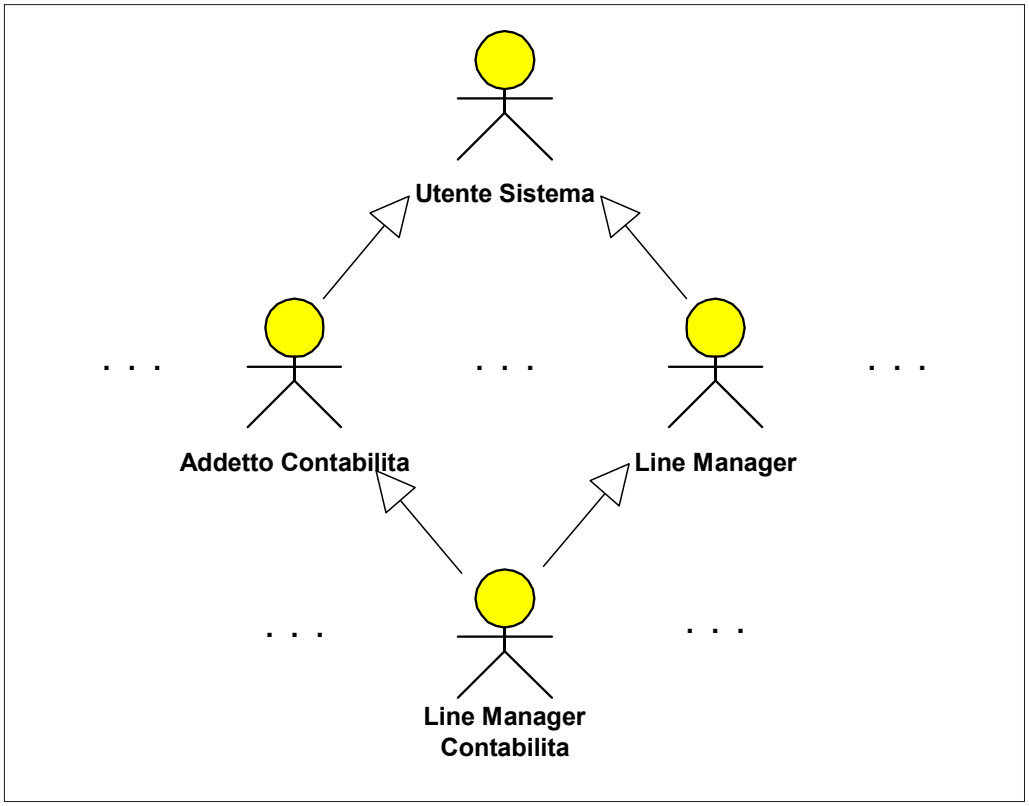
Descrizione

La comunità degli utenti del sistema da realizzare è classificata in gruppi di utenti in funzione delle mansioni svolte.

Di questi ruoli, quelli strettamente di interesse nel dominio di studio (in quanto veri e propri attori e non meri individui oggetto di controllo) sono i *Line Manager*, gli addetti alla sicurezza e i supervisori della sicurezza. I primi, oltre ad avere mansioni legate alla propria area di controllo (marketing, accounting, ecc.), hanno tutta una serie di responsabilità legate al loro status di manager. In altre parole, sono classificabili attraverso una doppia ereditarietà: ereditano dall'utente della propria area (devono essere in grado di poter eseguire tutti i compiti dei propri subalterni) e dal manager generico (devono svolgere mansioni manageriali).

> In sistemi come quello oggetto di studio, la corretta analisi e strutturazione della gerarchia degli utenti si ripercuote sull'organizzazione dell'infrastruttura di sicurezza e in particolare sui diritti di accesso del sistema. La relazione tra gli utenti e i relativi gruppi è chiaramente del tipo (n,n): un utente può appartenere a diversi gruppi e ciascuno di essi raggruppa svariati utenti. Inserire un utente ad un gruppo implica l'abilitazione dello stesso ad usufruire di un insieme ben definito di servizi (quelli necessari per espletare i compiti propri del ruolo) e l'esclusione dei restanti. In questo contesto dovrebbe essere evidente come gli attori dei casi d'uso rappresentino i gruppi in cui è possibile ripartire gli utenti del sistema.

Figura 8.2 — *Esempio di classificazione degli attori. Esempio relativo agli addetti del reparto di contabilità.*



Per quanto attiene all'infrastruttura di sicurezza oggetto di studio, i Line Manager hanno la responsabilità di richiedere:

- le credenziali per l'accesso al sistema per i dipendenti a loro assegnati specificando il profilo del relativo lavoro (dipartimenti accessibili, funzioni eseguibili, …),
- la variazione del profilo qualora un loro subalterno cambi la tipologia di lavoro (riceva una promozione, cambi di mansioni/dipartimento, …);
- l'eventuale sospensione dell'accesso ogni qualvolta il personale effettui un'assenza prolungata (malattia, vacanze, …);

- l'immediato lock (blocco), qualora i dipendenti si dimettano o siano licenziati, ecc.

Da tener presente che i `Line Manager` possono richiedere per i propri subalterni allocazioni, variazioni, cancellazione dei profili, ma la responsabilità finale spetta agli addetti della sicurezza. Le varie richieste avvengono attraverso il sistema e in particolare compilando un'apposita scheda denominata `Modulo di Accesso`. Queste schede, una volta compilate, sono sottoposte al sistema, il quale si occupa di convogliarle ad un apposito addetto alla sicurezza. Quest'ultimo ne effettua i controlli iniziali che possono generare un esito positivo e quindi il modulo è validato, oppure negativo e quindi il modulo viene respinto e posto all'attenzione del `Line Manager` richiedente. Una volta che un modulo è accettato questo viene automaticamente presentato all'attenzione del supervisore della sicurezza (si tenga a mente che si sta facendo riferimento a un'organizzazione internazionale con personale dislocato in diversi uffici sparsi in diverse nazioni), il quale può approvare definitivamente il modulo rendendolo operativo oppure rifiutarlo. I moduli di accesso respinti, in qualsiasi fase, tornano all'attenzione dei `Line Manager` richiedenti (corredati dalla motivazione per la sospensione), i quali possono decidere di apportare le necessarie variazioni, oppure, in casi estremi, di cancellare la pratica (per esempio una richiesta di blocco utente potrebbe essere rifiutata poiché l'utente in questione è già bloccato). Chiaramente è necessario che il sistema tenga traccia delle varie fasi attraverso le quali un modulo di accesso transita, memorizzando i vari utenti che ne originano le transizioni nelle varie fasi, i timestamp, eventuali commenti, e così via.

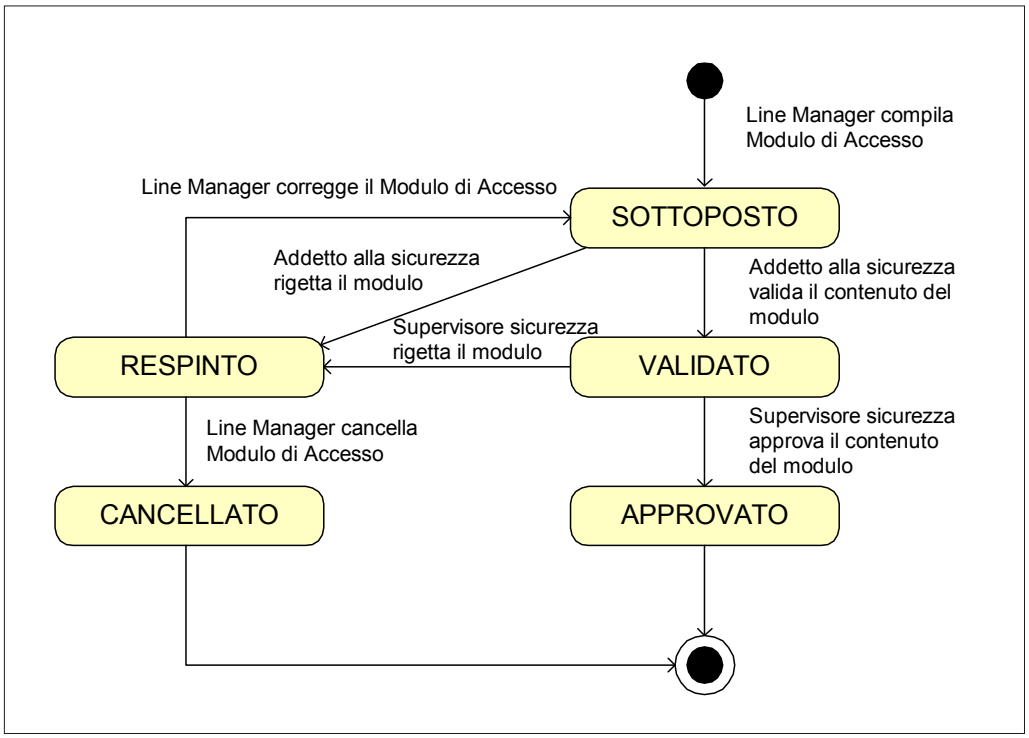
Questa descrizione si presta a essere efficacemente sintetizzata e chiarita attraverso un diagramma degli stati riportato in fig. 8.3. Come si può notare la relativa comprensione non richiede approfondite conoscenze di nozioni tecniche e quindi il diagramma si presta ad essere utilizzato come valida piattaforma di colloquio con gli utenti.

> L'esperienza insegna che diagrammi degli stati non eccessivamente complessi permettono quasi sempre, e in maniera indolore, di chiarire i requisiti utente, renderli più immediati, evidenziare eventuali lacune nell'analisi delle transizione tra gli stati, e così via. Quindi, qualora la parte del dominio oggetto di studio si presti ad essere rappresentata per mezzo della transizione di un oggetto (o di un gruppo opportunamente strutturato) attraverso un insieme finito di stati, è sempre consigliato rappresentare il tutto tramite un opportuno diagramma degli stati.

Si tenga presente che i moduli di accesso non sono relativi esclusivamente alla richiesta di un nuovo profilo per un utente, bensì permettono di richiedere la variazione di un profilo esistente, di sospendere, di bloccare e sbloccare un utente, e così via.

Una volta che un `Modulo di Accesso` è approvato, non significa che automaticamente la richiesta contenuta abbia effetto immediato. Il `Line Manager` deve indicare il

Figura 8.3 — *Diagramma degli stati del ciclo di vita di un modulo di accesso.*



periodo di validità degli effetti e quindi il sistema si farà carico di assicurare che questi abbiano luogo il giorno e l'ora indicati nel `Modulo di Accesso`. Ciò equivale a dire che deve essere previsto un meccanismo in grado di "prenotare" dei lavori da eseguire (`PendingTask`, come per esempio rimozione di una sospensione) e un servizio temporizzato che si occupi di eseguirli.

In questa fase, qualora il `Modulo di Accesso` si riferisca al primo profilo da assegnare all'utente, il sistema ha la responsabilità di generare la prima password temporanea e di notificarla al relativo utente tramite un'e-mail opportunamente crittografata. Da tener presente che il login dovrebbe costituire un identificativo univoco dell'utente da generare all'atto dell'inserimento dei relativi nel sistema di anagrafica dell'organizzazione stessa.

Per ciò che attiene alle funzioni relative al `Modulo di Accesso`, è possibile evidenziare una serie di business rules relative a controlli a carico del sistema: per esempio è necessario inibire ogni tentativo di

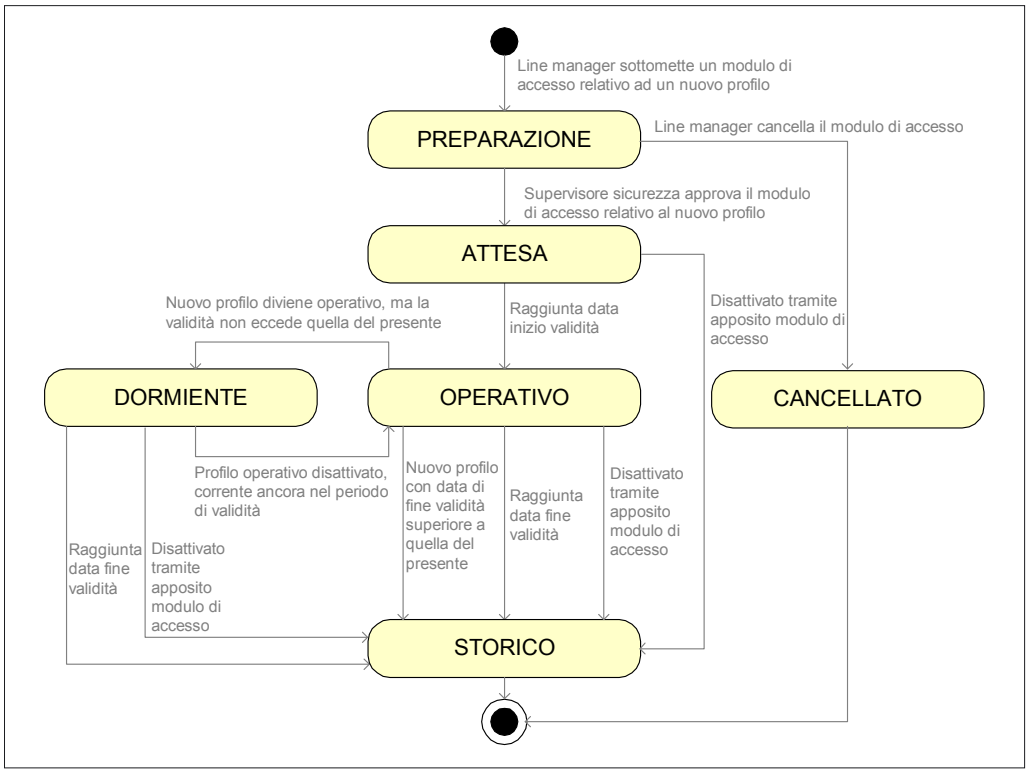
- compilare moduli di accesso riferiti a sé stesso da parte di un Line Manager;

- validare moduli di accesso riferiti a sé stesso da parte di un addetto alla sicurezza;
- approvare un modulo riferito a sé stesso da parte di un supervisore alla sicurezza, di approvarne altri da lui stesso validati, ecc.

> Il diagramma di fig. 8.4 si presta a essere ottimizzato racchiudendo gli stati di *ATTESA*, *DORMIENTE* e *OPERATIVO* in un macrostato (*VALIDO* per esempio). Ciò permetterebbe di evidenziare i flussi comuni, una sola volta, al livello del nuovo macrostato, mentre i restanti resterebbero inalterati. In questo contesto si è deciso di non operare in tal senso al fine di non complicare il diagramma.

Molto importante è memorizzare in appositi sistemi di log tutte le attività eseguite dagli utenti, nonché i tentativi di violare le politiche di sicurezza. Questi permettono di eseguire

Figura 8.4 — *Ciclo di vita di un profilo.*



appositi report e di rilevare tempestivamente tentativi di intrusione, meglio noti come incidenti di sicurezza.

Considerata l'eventuale complessità e distribuzione mondiale del sistema, è necessario introdurre il concetto di "privatezza dei dati". Nell'infrastruttura di sicurezza presa in considerazione, questo principio comporta l'introduzione del concetto di dipartimento e di filiale (da notare che *dipartimento* e *filiale* non sono utilizzati come sinonimi bensì con il loro naturale significato: una filiale è composta da diversi dipartimenti). Fino a questo punto si è considerato un classico livello di sicurezza operante al livello di servizi (considerando un sistema web, i servizi sono rappresentati da opportuni indirizzi URL delle pagine che li rendono fruibili agli utenti). Se per esempio un utente appartiene al gruppo degli addetti alla sicurezza, questo lo abiliterà a eseguire tutto un insieme di servizi (come per esempio validazione dei moduli di accesso) e ne escluderà l'esecuzione di tutto un altro insieme (come per esempio approvazione dei moduli di accesso). Nel sistema preso in esame si tratta della classica condizione necessaria ma non sufficiente. In effetti, se un utente dispone di un ruolo X ciò non lo abilita automaticamente a usufruirne per tutti i dipartimenti e/o filiali (per esempio, il direttore di una filiale di banca non può automaticamente esercitare lo stesso ruolo in tutte le altre filiali). Pertanto, ogni utente deve possedere nel profilo opportune sezioni (contesti di sicurezza) nelle quali è sancito un gruppo di appartenenza e i dipartimenti/filiali in cui l'utente è autorizzato a usufruire dei relativi servizi. Un addetto alla sicurezza, per esempio, sarà in grado di validare o rigettare i `Moduli di Accesso` solo relativi ai dipartimenti a cui è stato abilitato. Immaginando che l'azienda disponga di filiali a Roma, Milano, Parigi, Londra, … e un utente sia autorizzato a usufruire del proprio ruolo limitatamente per le sedi di Roma e Milano, ciò comporta che egli non sarà mai in grado di vedere e tanto meno manipolare dati relativi ad altre filiali. In alcune aree particolari, l'accesso ai dati di un utente deve essere ristretto anche a un livello inferiore rispetto a quello delle filiali, ecco dunque che spesso è necessario riferirsi al livello di dipartimento. Se per esempio si prende in considerazione un sistema bancario, è possibile notare come un utente con il ruolo di `Broker` dovrebbe aver accesso ai soli dati relativi ai contratti (più precisamente `Trade`) da lui stipulati più altri di carattere generare, e quindi il livello di accesso dovrebbe essere addirittura inferiore a quello di dipartimento, mentre un addetto alla risoluzione di eventuali anomalie presenti nelle trascrizione dei contratti (come per esempio regole di pagamento errate) potrebbe operare a livello di un'intera filiale o addirittura di più filiali.

Affinché questo meccanismo possa aver luogo, è molto importante che il sistema sia in grado di conoscere l'appartenenza dei vari dati, o gruppi, al fine di poterli confrontare con quelle assegnate all'utente. Ciò si ottiene associando ai dati anche i dipartimenti e le filiali di appartenenza. Se per esempio si utilizzasse un DBMS relazionale, sarebbe necessario indicare i dipartimenti e/o filiali di appartenenza dei vari record di dati. Chiaramente non è necessario restringere l'accesso per tutti i dati bensì solo per quelli ritenuti sensibili. Se per esempio il sistema manipolasse dati di carattere generale (per esempio dati statici rela-

tivi a valute, nazioni, filiali, e così via) non sarebbe ovviamente necessario restringere l'accesso a questa tipologia di dati.

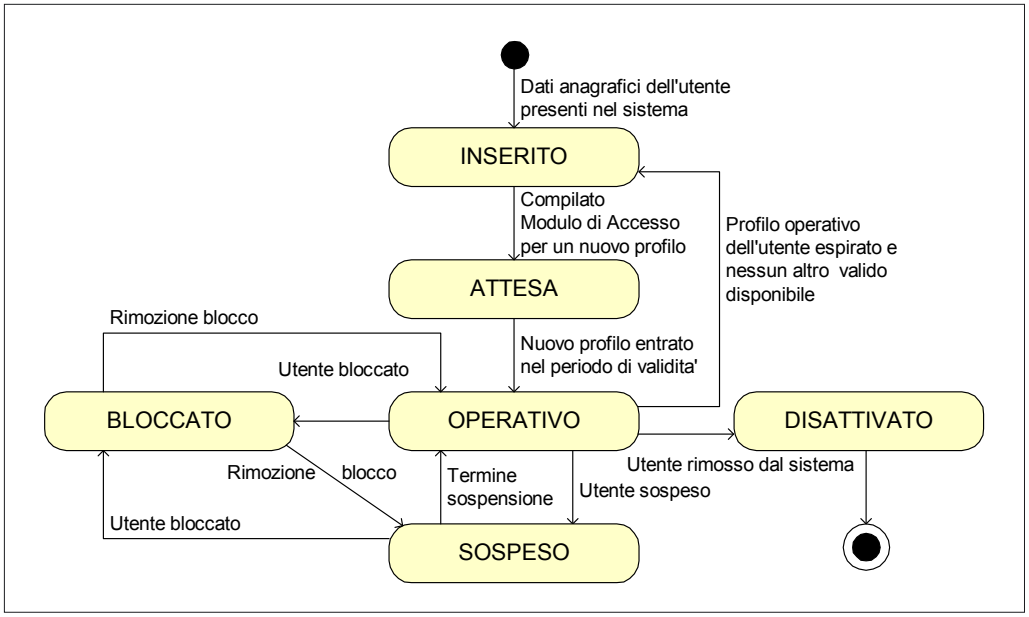
Riassumendo il tutto, quando un utente richiede di eseguire un servizio (inserisce un indirizzo URL nel browser o effettua un click su un link) il sistema reperisce l'elenco dei gruppi di appartenenza dell'utente e verifica se ad almeno uno di questi sia associata l'abilitazione all'esecuzione del servizio richiesto. In caso affermativo, l'utente è autorizzato ad eseguire il servizio, altrimenti ne viene negato l'accesso (livello di sicurezza relativo ai servizi). Una volta ottenuta l'autorizzazione ad usufruire del servizio (per esempio compilazione `Modulo di Accesso`), l'utente, per poterne fruire, deve tipicamente reperire determinati dati (reperimento del subalterno a cui fa riferimento la richiesta) ed eventualmente aggiornarli (memorizzazione di un nuovo `Modulo di Accesso`). Nel primo caso i dati reperiti vengono filtrati al fine di far transitare solo quelli relativi ai dipartimenti e filiali per le quali l'utente può esercitare il ruolo mentre, nel caso di inserimenti e/o aggiornamenti, sono bloccati eventuali tentativi di impostare dati appartenenti a dipartimenti e filiali per le quali non è autorizzato.

Introdotto il concetto di sicurezza o privatezza dei dati è possibile presentare il concetto di incompatibilità necessario per comprendere le ragioni per cui un addetto alla sicurezza potrebbe respingere un `Modulo di Accesso`. Una volta strutturata l'organizzazione degli utenti in gruppi è possibile notare che alcuni di essi sono incompatibili tra loro, in altre parole il sistema deve prevenire che i profili assegnati agli utenti possano contenere mansioni, dal punto delle policy aziendali, inconciliabili (per esempio il ruolo di una persona che stipula contratti con i clienti dovrebbe essere incompatibile con quello relativo al controllo dei contratti stipulati).

Si sta facendo riferimento al concetto noto con il nome di separazione delle mansioni (*separation of duties*). In sostanza, si vuole assicurare che uno stesso utente non possa avere il controllo completo di transazioni business dall'inizio alla fine. Il controllo delle transazioni deve essere ripartito tra più persone. Per esempio, se la transazione presa in esame è l'assegnazione di nuovo profilo, le responsabilità relative alla richiesta, prima validazione e approvazione finale, devono essere ripartite tra altrettanti utenti. Si vuole ovviamente evitare che una stessa persona, disponendo del completo controllo di alcune transazioni, possa usufruirne a scopi personali o peggio ancora fraudolenti (come ripeteva il grande Totò "l'occasione fa l'uomo ladro"). Spesso, oltre alla politica della separation of duties, si abbina quella della rotazione delle mansioni (*rotation of duties*) che comporta la periodica variazione dell'assegnazione dei lavori. Ciò chiaramente serve ad evitare che si instauri un circolo vizioso e fraudolento all'interno dell'organizzazione.

In alcuni casi le incompatibilità potrebbero non essere limitate alle mansioni, ma potrebbero includere anche concetti come filiali e dipartimenti (per esempio agli utenti potrebbe essere vietato di poter accedere contemporaneamente ai dati della filiale di Milano e di Roma). Le incompatibilità potrebbero poi essere classificate in *warning* (il sistema si limita ad evidenziarle demandando all'addetto della sicurezza la decisione finale) ed *error*

Figura 8.5 — *Diagramma degli stati relativi al ciclo di vita di un utente. Si tratta di un diagramma non completamente corretto dal punto di vista dello UML in quanto lo stato di* `BLOCCATO` *dovrebbe memorizzare lo stato da cui vi si arriva (in cui avviene il bloccaggio) per potervici ritornare qualora il blocco venga rimosso. Ciò si ottiene inserendo un particolare sottostato denominato* `History` *e indicato con la lettera* `H`*. In questo contesto si è deciso di non introdurlo al fine di non appesantire ulteriormente la trattazione.*



(il sistema blocca il `Modulo di Accesso`). Il controllo chiaramente dovrebbe essere effettuato durante tutte le fasi (sottomissione, validazione e approvazione): alcune regole di incompatibilità potrebbero venir inserite o aggiornate tra l'intervallo di tempo che va dal momento in cui un `Modulo di Accesso` sia sottoposto a quello in cui sia validato.

Ogni utente del sistema è autorizzato a possedere diversi profili, però è consentito disporre di uno solo nello stato operativo e quindi utilizzabile. Per esempio, dovendo trasferire temporaneamente un dipendente da un dipartimento o da una filiale ad un'altra, è possibile congelarne il profilo corrente (porlo nello stato sospeso) per assegnarne uno nuovo limitatamente alla durata del trasferimento.

Per semplicità, l'accesso al sistema avviene utilizzando il paleolitico metodo basato sulla coppia (login, password). La trattazione ovviamente si presta ad essere facilmente estesa con tecniche più sicure (smart cards, sistemi di riconoscimento biometrico, e così via).

Per quanto concerne l'entità utente, le informazioni di interesse sono: login (generata automaticamente dal sistema), password (corredata dalla data e orario di scadenza), il

codice fiscale, il nome, cognome, la data e città di nascita, l'indirizzo di posta elettronica ed il numero di telefono interno, il recapito a cui contattarlo e il ruolo esercitato all'interno dell'organizzazione.

I servizi di blocco e sblocco e sospensione e revoca sospensione utente, oltre a poter essere eseguiti attraverso l'iter del `Modulo di Accesso` (in tal caso la richiesta è responsabilità del `Line Manager`), devono poter essere eseguiti direttamente sia dagli addetti alla sicurezza, sia dal supervisore. Le differenze tra Blocco (*lock*) e Sospensione sono che il primo ha una connotazione molto forte (avviene automaticamente qualora un utente fallisca per tre volte consecutive di seguito l'inserimento della password, in caso di licenziamento, sospetto di intrusione, …) e non ha una scadenza, mentre il secondo è meno forte (si applica per esempio per temporanee assenze dal lavoro: malattie, ferie, ecc.) e ha carattere transitorio: è indicata la scadenza.

Produzione del modello a oggetti del dominio

L'obiettivo del presente paragrafo è illustrare un approccio pratico ma efficace all'individuazione delle classi appartenenti al dominio del problema, noto con il nome di "analisi dei nomi e verbi".

Collaborando con molte aziende, è possibile constatare che le domande che più frequentemente ci si sente rivolgere sono relative alle tecniche da utilizzarsi sia per individuare classi appartenenti al dominio del problema, sia, più in generale, per poter materializzare i requisiti utente (magari formalizzati per mezzo dei casi d'uso) attraverso un opportuno modello del disegno. Si tratta di interrogativi piuttosto legittimi le cui risposte, ahimè, sono tutt'altro che banali. In questo paragrafo si focalizza l'attenzione sul primo quesito. Individuare le classi del dominio sta diventando un'attività sempre più cruciale nel processo di costruzione di sistemi basati sui componenti in quanto, per ciò che concerne la realizzazione dell'infrastruttura, si stanno compiendo molti sforzi per renderla sempre più automatica (si considerino per esempio le architetture EJB e .net in cui gran parte dell'infrastruttura è incapsulata direttamente nei tool forniti dai vari vendor).

La realizzazione del modello a oggetti del dominio è un'attività complessa fortemente dipendente da quelle che sono proprietà soggettive del singolo individuo (esperienza, cultura, abilità di analizzare, capacità analitiche, ecc.) in cui spesso anche la fortuna gioca il suo ruolo. Questa forte dipendenza dalle caratteristiche soggettive degli individui determina la difficoltà di formalizzare un metodo rigoroso e meccanico. La buona notizia è che con l'esperienza il processo diviene assolutamente naturale (cognitivo) tanto da essere applicato in maniera istintiva. Una similitudine è relativa all'apprendimento delle lingue straniere (qui parla la voce dell'esperienza dell'autore). Quando si parla una lingua da molti anni, si coniugano i verbi senza neanche pensarci — spesso purtroppo —, il tutto avviene in maniera immediata, i pensieri si sciolgono spontaneamente in discorsi razionali. Quando invece si è novizi, spesso accade che, prima di riuscire a coniugare i verbi, sia

necessario far precedere qualche minuto di riflessione del tipo: il verbo è regolare oppure l'azione non continua nel presente ecc. Ebbene, nell'individuazione delle classi appartenenti al modello del dominio avviene esattamente la stessa cosa: l'esperienza fa sì che sia possibile costruire modelli a oggetti del dominio semplicemente intervistando gli esperti dell'area business o leggendo i documenti di analisi dei requisiti.

Continuando nella similitudine, così come il modo migliore per imparare una lingua è frequentare persone madrelingua, così anche nel mondo OO: il modo migliore per accrescere la propria esperienza consiste nel lavorare con personale preparato ed esperto ed eventualmente studiare gli esempi riportati nella letteratura specifica.

Infine, è possibile osservare che le persone, nel parlare, tendono a riutilizzare determinati costrutti. Nella modellazione, molto spesso, avviene lo stesso procedimento: si tende a riutilizzare pattern dimostratisi validi in precedenti modellazioni (in fondo non è uno dei punti di forza dell'OO?). Anche se i sistemi a cui si lavora risultano sempre diversi tra loro, tipicamente, è sempre possibile individuare delle aree comuni.

Ora, sebbene l'esperienza permetta di sviluppare le capacità necessarie per realizzare validi modelli a oggetti, cosa fare durante il lasso di tempo necessario per acquisire le dovute competenze? In questo paragrafo si presenta una metodo pratico che tenta proprio di risolvere questo problema: agevolare la realizzazione di **versioni embrionali** del modello a oggetti del dominio. Ciò si ottiene fornendo delle buone idee, consigli empirici, e rendendo consapevoli delle trappole nelle quali si può cadere… Il resto, ahimè è a carico del modellatore. Il metodo può essere di grande aiuto, però non si tratta della tanto agognata bacchetta magica, in grado di far emergere le classi dai modelli di analisi dei requisiti.

Come si vedrà si tratta di una tecnica, che sebbene abbia una sua legittimità accademica è stata a lungo oggetto di critiche non sempre giustificate.

> Nel 1990 presso il Tokyo Institute of Technology è stato realizzato un software in grado di effettuare l'analisi di un testo dei requisiti al fine di costruire una versione iniziale del modelli a oggetti. Sebbene di questa tecnica esistano diverse versioni, quella iniziale probabilmente è attribuibile a Abbott.

Chiaramente è necessario inquadrare ogni strumento nel proprio contesto. Se la tecnica promettesse di sostituire quelle più accademiche, allora probabilmente le critiche risulterebbero legittime, se invece l'obiettivo è fornire un approccio semplice, senza avere grosse pretese di formalità, per costruire versioni iniziali del modello a oggetti del dominio, rimandando alle metodologie più formali le attività di verifica e miglioramento del modello stesso, allora verosimilmente i vari attacchi perderebbero di legittimazione.

Le prime iterazioni dei processi iterativi e incrementali di sviluppo del software sono tipicamente incentrate sulla produzione di due manufatti di estrema importanza: il modello dei casi d'uso e quello a oggetti del dominio. Generalmente si preferisce iniziare dal primo, ossia dall'analisi dei servizi (funzioni) che il sistema dovrà fornire ai propri attori, per poi focalizzare l'attenzione sulle entità coinvolte. Chiaramente si tratta di "preferenze

soggettive", nulla vieta di utilizzare l'approccio contrario, l'importante è che alla fine i vari modelli risultino coerenti e consistenti. Un altro manufatto molto importante è il vocabolario (o glossario) dei termini utilizzati nell'area business: in ultima analisi il modello a oggetti del dominio può essere visto come la versione OO di un suo opportuno sottoinsieme. Alla fine dell'analisi dei requisiti è quindi necessario che il modello a oggetti del dominio, gli use case ed il glossario siano mutuamente bilanciati. Ciò si ottiene quando gli use case ed il vocabolario contengono riferimenti (nel primo caso) e descrizioni (nel secondo) di tutte (o quasi) le classi illustrate nel modello a oggetti del dominio ed il viceversa, ossia tutti gli elementi menzionati nel modello degli use case e tutti i termini appartenenti alla sezione business del vocabolario siano modellati nel modello a oggetti.

Qualora non sia ancora disponibile un modello dei requisiti, magari perché si vuole costruire i vari modelli contemporaneamente procedendo con una continua sincronizzazione, è possibile applicare la tecnica utilizzando gli appunti emersi durante i vari colloqui e workshop con gli utenti e qualsiasi documentazione esistente, come per esempio un documento di descrizione dell'area business, dell'impresa ecc. Nel caso limite in cui non si disponga di molta documentazione e non si sappia bene da dove iniziare, un buon approccio consiste nel cominciare a redigere una lista riportante tutte le asserzioni relative allo spazio del problema emerse durante i vari colloqui (frasi del tipo "un utente compila diversi ordini, ciascuno di essi è composto da articoli, gli articoli dispongono del prezzo e possono prevedere delle promozioni dipendenti da particolari regole", ecc.).

Individuazione delle classi

Primo passo

Il primo passo da seguire consiste nell'individuare l'elenco delle ipotetiche classi. In questo stadio si tratta di un'operazione non eccessivamente complessa, quasi meccanica. Più precisamente è necessario eseguire una sorta di analisi grammaticale. Si legge attentamente il documento (use case, appunti presi durante i colloqui con gli utenti, documentazione varia, ecc.), e si evidenziano tutti i nomi e coppie di nomi/aggettivi incontrati strettamente legati all'area oggetto di studio. Terminata la prima lettura, conviene effettuarne una seconda (eventualmente anche una terza) e quindi riportare quanto evidenziato in un'apposita lista. Chiaramente è sempre consigliabile iniziare una primissima cernita eliminando i nomi che palesemente non potrebbero mai assurgere al ruolo di classe nel contesto del dominio di riferimento ma sono utilizzati unicamente allo scopo di illustrare lo stesso (per esempio *entità*, *sistema*, *organizzazione*, ecc.).

Tabella 8.1 — *Elenco delle ipotetiche classi appartenenti al dominio oggetto di studio. Da notare che alcune candidate al ruolo di classe sono state introdotte artificiosamente (codice fiscale, e-mail, …) al fine di poter illustrare alcune regole riportate di seguito, mentre altre sono state volutamente ignorate sempre per fini espositivi.*

N.	Nome	Descrizione
1	CodiceFiscale	Codice fiscale degli utenti del sistema.
2	ContestoDiSicurezza	È parte integrante di un profilo. Accoppia un'istanza di un gruppo utenti alla filiale o dipartimento in cui i servizi specificati dal gruppo possono essere utilizzati. Ciascun profilo dispone di uno o più contesti di sicurezza.
3	Dipartimento	Elemento dell'organizzazione gerarchica interna delle filiali.
4	eMail	Incapsula un indirizzo di posta elettronica.
5	Filiale	Rappresenta una sede distaccata dell'organizzazione.
6	FirmaDigitale	Sistema di riconoscimento utente.
7	GruppoUtenti	Ciascuna istanza di questa classe individua una categoria di utenti del sistema (ruolo) e quindi l'elenco dei servizi fruibili.
8	Indirizzo	Rappresenta il domicilio di un utente.
9	Località	Rappresenta le località a cui si riferiscono le varie istanze della classe Indirizzo.
10	Log	Descrizione di un'azione eseguita dall'utente (auditing), corredate dall'eventuale azione intrapresa dal sistema come risposta.
11	Login	Identificativo univoco di ciascun utente utilizzato come credenziale di riconoscimento.
12	ModuloAccesso	Modulo che consente ai *Line Manager* di richiedere l'esecuzione di specifici servizi relativi agli utenti e relativi profili.
13	Motivazione	Spiegazione relativa al rigetto di una richiesta di accesso.
14	Nazione	Rappresenta le nazioni a cui appartengono le varie località.
15	Notifica	Incapsula la comunicazione che il sistema effettua per notificare agli addetti alla sicurezza ed ai supervisori, l'esistenza di Moduli di accesso da prendere in considerazione e ai Line manager il rifiuto di un modulo da loro sottoposto.
16	Password	Parola segreta dell'utente da utilizzarsi in coppia con la Login per accedere al sistema.
17	PendingTask	Sono le attività che il sistema deve eseguire per rendere operativi gli effetti sanciti nei moduli di accesso approvati dai supervisori della sicurezza.
18	Profilo	Si tratta di una classe necessaria per assegnare agli utenti le autorizzazioni ad eseguire i servizi offerti dal sistema necessari per espletare le proprie mansioni.
19	Record	Gruppo omogenei di dati trattati dal sistema (si fa riferimento al significato di record proprio dei database).
20	RegolaIncompatibilita	Regola dichiarante ruoli, filiali e/o dipartimenti tra loro incompatibili.
21	Report	Elenco di informazioni omogenee estratte attraverso l'applicazione di un criterio ben definito all'insieme dei dati memorizzati nel sistema.
22	RuoloOrganizzazione	Si tratta dei ruoli propri previsti dall'azienda.
23	Schede	Schede compilate dai Line Manager.
24	Servizio	Rappresenta una generica funzionalità fornita dal sistema.
25	StadioModuloAccesso	Serve per tener traccia dell'evoluzione di una richiesta di accesso. In particolare rappresenta uno stadio attraverso il quale è transito il relativo modulo.
26	Utente	Rappresentazione logica del sistema degli utenti registrati e quindi abilitati a fruirne i servizi.

Nel selezionare i nomi delle classi, è molto importante che questi siano il più possibile correlati agli oggetti appartenenti al dominio del problema. Chiaramente più i nomi sono ovvi più è facile comprendere il modello. Si tenga presente anche che il modello a oggetti del dominio è un prodotto di input di diverse fasi, destinato ad essere fruito dalle più svariate tipologie di persone (utenti/clienti, esperti del business, architetti, ecc.) e quindi la facilità di lettura assume un'importanza fondamentale. Quello che andrebbe evitato è il ricorso a nomi che non appartengono all'area oggetto di studio e/o frutto di incomprensibili contrazioni.

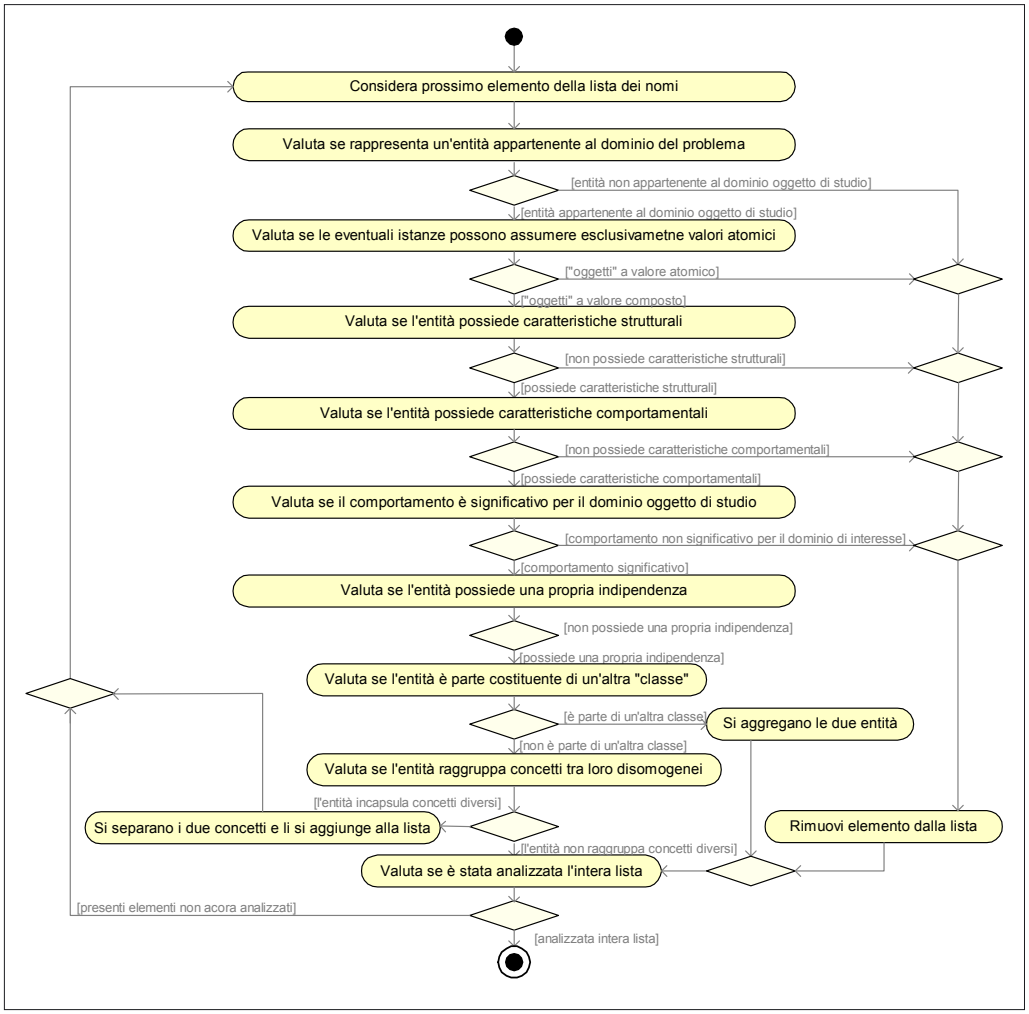
Con riferimento alla descrizione del dominio oggetto di studio, sarebbe possibile compilare una tabella come la tab. 8.1.

Secondo passo

La tabella precedente costituisce il prodotto di ingresso della fase successiva. In particolare, l'obiettivo del secondo passo è filtrare la lista dei sostantivi presenti nella tabella al fine di eliminare quelli che o non possono assurgere al ruolo di classe oppure, pur avendo tutte le credenziali, esulano dagli obiettivi del progetto (*out of scope*). In questo passo si evidenza come saper eliminare idee sbagliate è altrettanto importante di escogitarne delle buone.

Durante questa attività il livello di impegno richiesto comincia ad essere di una certa consistenza. L'ideale sarebbe disporre di uno strumento formale che permetta di stabilire, per ciascun nome inserito nella lista, se si tratti di una classe o meno (una sorta di cartina tornasole delle classi). Il problema, ancora una volta, è che mentre l'esperienza permette di eseguire tale valutazione istintivamente, l'individuazione di un criterio rigoroso non è altrettanto immediato (altrimenti si potrebbe renderlo automatico attraverso un apposito sistema). Chiaramente tutti i tecnici sono in grado di pensare ad alcuni criteri sindacabili come: l'appartenenza o meno al dominio preso in considerazione, la struttura dell'oggetto, e così via. Tuttavia un criterio razionale e completo è meno facile da ideare. Nella fig. 8.6 è riportato un diagramma delle attività che illustra alcuni criteri utili al processo di distinzione tra classi vere e proprie, e supposte tali. Chiaramente, il modo migliore per valutare le classi consiste nel seguire i consigli forniti nel capitolo relativo agli insegnamenti dell'OO (confronta paragrafo "Classi ben disegnate"). Il problema è che si tratta di direttive ad elevato grado di formalità. In questo ambito si vogliono considerare i medesimi metri di giudizio, presentandoli però da una prospettiva decisamente più operativa e quindi più semplice da capire.

Figura 8.6 — *Diagramma delle attività relativo ai criteri da utilizzarsi come riferimento nel processo di validazione delle classi.*



Criterio 1: classi in scope

Come preannunciato una prima domanda da porsi è se la classe appartenga o meno al dominio oggetto di studio che si intende modellare. Una risposta negativa non significa necessariamente che l'entità che si sta analizzando non sia una classe, ma semplicemente che non è il caso di spendere tempo ad investigare, a studiarne la struttura ed il comportamento giacché comunque non è di interesse per il modello.

Questa domanda, sebbene in prima analisi possa sembrare piuttosto banale, in alcune situazioni potrebbe non esserlo assolutamente. La difficoltà di stabilire l'appartenenza o meno di una classe al dominio di interesse, in alcuni casi, è imputabile alla mancanza di chiarezza nello stabilire i confini stessi del sistema. In tali circostanze è opportuno meditare sull'eventualità di compiere il famoso passo indietro nel tentativo di definire con maggiore precisione gli obiettivi del sistema.

Per esempio, sebbene possa essere di notevole interesse la rappresentazione della firma digitale, dai requisiti del cliente emerge direttamente che si tratta di un meccanismo non richiesto nella realizzazione delle prime versioni del sistema. Ancora tentare di rappresentare i record trattati, al fine di evidenziare la proiezione della sicurezza a livello di dati, potrebbe risultare interessante, ma sarebbe abbastanza fuori luogo trattandosi addirittura di dettagli appartenenti allo spazio delle soluzioni. Ancora, mentre potrebbe essere di un certo interesse disporre di entità quali il recapito e la località di ogni utente, quella di nazione probabilmente esula dagli obiettivi del sistema. Nel contesto oggetto di studio, queste informazioni si prestano a divenire attributi della classe città. Se, invece, l'obiettivo del sistema fosse stato, per esempio, realizzare un sistema per la prenotazioni viaggi, evidentemente classi rappresentanti concetti come nazione e continente avrebbero assunto un ruolo ben più importante.

Criterio 2: istanze della classe

La seconda domanda è relativa alle presunte istanze della classe, e in particolare ai valori che queste potrebbero assumere in un ipotetico caso reale tenendo presente che ci si trova nella fase di studio del dominio e non in quella di disegno.

Taluni attributi, come per esempio le date, pur essendo a tutti gli effetti oggetti e non dati semplici, considerarli più di attributi nel contesto della modellazione dello spazio del problema non porterebbe alcun vantaggio, anzi finirebbe per complicare inutilmente il modello.

Premesso ciò, se ci si dovesse accorgere che le istanze della classe oggetto di verifica possono assumere unicamente valori semplici, verosimilmente non è il caso di considerarla tale, indipendentemente dalla relativa rilevanza nel contesto del sistema. Probabilmente è più opportuno considerarla un attributo. Si consideri la *login* assegnata agli utenti; sebbene costituisca un'informazione di vitale importante per il sistema, le relative istanze

assumono valori atomici che si prestano ad essere modellati attraverso un attributo piuttosto che una classe.

Il criterio relativo alle istanze della classe potrebbe essere argomentato affermando che in alcuni casi si può dare luogo a classi con un solo attributo, magari per evidenziare o un valore opzionale (per esempio il commento relativo ad una fase del ciclo di vita di un modulo di accesso), oppure il caso diametralmente opposto: una molteplicità superiore a uno (per esempio gli indirizzi di e-mail di un utente). Questi casi però rappresentano più l'eccezione che la norma, specie considerando che ci si trova in una fase iniziale del processo di sviluppo del software e che per le ottimizzazioni c'é sempre tempo.

Criterio 3: analisi della struttura

Il punto successivo, che a dir il vero è intimamente legato al precedente (e pertanto si configura più come un doppio controllo che come criterio a sé stante), è relativo all'"analisi" dell'eventuale struttura dell'ipotetica classe. In altre parole, è necessario chiedersi se sia possibile identificare una qualche struttura, in termini di attributi, relazioni, ecc. In caso di risposta affermativa allora potrebbe avere senso considerare la classe come tale, altrimenti si ricade nel caso di aver individuato un potenziale attributo.

Per esempio è di importanza fondamentale conoscere, per ogni utente del sistema, il relativo indirizzo di e-mail interno (per esempio per notificargli la password iniziale). Sebbene sia possibile ipotizzare una struttura (nome, cognome, dominio, ecc.), considerare gli indirizzi di e-mail più di un attributo sarebbe piuttosto pretestuoso.

Criterio 4: analisi del comportamento

Dopo essersi interrogati circa la struttura della classe, è possibile spostare l'attenzione sulle eventuali caratteristiche comportamentali (ciò avviene specificando eventuali servizi fornibili dalla classe). La classe `ModuloDiAccesso`, per esempio, potrebbe prevedere servizi atti a sottoporla all'attenzione del sistema di sicurezza (`submit()`), a validarne il contenuto (da parte di un addetto alla sicurezza, `verifyIncompatibilitiesRules()`), a rifiutarla (`reject()`), ecc. Questo criterio è molto delicato e non sempre è il caso di prenderlo in considerazione nel contesto dei diagrammi a oggetti del dominio. Qualora sia possibile identificare caratteristiche comportamentali, allora sicuramente è possibile proseguire con la valutazione dei criteri successivi; in caso contrario, la situazione nasconde diverse insidie.

La difficoltà di individuare caratteristiche comportamentali non autorizza a scartare una classe. Come riportato in precedenza, quando si elabora un modello a oggetti del dominio, si focalizza l'attenzione sui dati e sulla relativa organizzazione, mentre il comportamento assume un interesse marginale (forse è il caso di sottolineare che ciò vale solo per il modello a oggetti del dominio, per gli altri la

mancanza di comportamento costituisce l'elemento più evidente per scartare una classe). Quello che si va ad identificare sono le famose classi entità, ossia veri e propri contenitori di dati che, in qualche modo, somigliano allo schema di tabelle di database relazionali. Queste classi, tipicamente sono costituite da una serie di metodi selettori (`get/set`) utilizzati per accedere ai relativi attributi. Quindi pur non disponendo di particolare comportamento, si tratta di classi di estrema importanza.

Per esempio, volendo rappresentare le filiali ed i relativi dipartimenti, si potrebbe pensare ad attributi come il codice, il nome, la descrizione, ecc. Ora immaginare un comportamento aggiuntivo diverso dai metodi `get/set` dovrebbe rappresentare un buon esercizio di fantasia eppure a questo livello sono classi a tutti gli effetti.

Criterio 5: comportamento in scope

Una domanda che scaturisce dalla precedente è se il comportamento identificato sia rilevante o meno per l'analisi dell'area business che si sta considerando. Si dovrebbe trattare di un'interrogazione inutile qualora l'analisi richiesta dal primo interrogativo sia stata eseguita accuratamente. Molto spesso però, solamente a questo stadio si ha una migliore percezione della struttura e del comportamento della classe e quindi si è in grado di rivalutare accuratamente il quesito del punto 1. Per esempio l'entità `Report` (sebbene violerebbe anche altri criteri soprattutto per via della genericità della definizione) potrebbe possedere del comportamento, però risulterebbe fuori scope, e comunque la relativa definizione risulterebbe ridondante: si presterebbe ad incapsulare parti del modello del dominio.

Criterio 6: valutazione dell'indipendenza

Un ulteriore elemento da considerare è se l'ipotetica classe possegga una propria indipendenza oppure abbia senso esclusivamente nel contesto di una più ampia. Analizzando attentamente il criterio, ci si può facilmente rendere conto della presenza dalla teoria dell'ADT (*cfr.* Capitolo 6). Equivale ad interrogarsi se l'ipotetica "classe" è un nuovo tipo di dato con un insieme ben definito e separato di operazioni, oppure se queste sono o possono naturalmente confluire tra quelle fornite da un'altra classe già accreditata. Eventualmente è possibile porsi la stessa domanda per le classi individuate in precedenza relativamente a quella correntemente analizzata. Qualora una candidata classe sia parte di un'altra è necessario eliminarla e conseguentemente rivedere la definizione di quella "inglobante". Un esempio classico è relativo al codice fiscale degli utenti, sebbene tale entità potrebbe superare diverse delle precedenti fasi (si tratta di un dato strutturato, può avere del comportamento come per esempio `verificaCodice()`, ecc.), nella maggior parte delle situazioni non è il caso considerarlo una classe, probabilmente è più opportuno trattarlo come un attributo di un'altra, quale per esempio "persona".

Tabella 8.2 — *Evoluzione della tabella riportante l'elenco delle classi appartenenti al dominio oggetto di studio. In questa versione è evidenziato l'eventuale motivo che ha portato a scartare alcune classi candidate mentre la descrizione non è stata riportata per questioni di impaginazione della tabella.*

N.	Nome	Esito	Spiegazione
1	CodiceFiscale	✗	Non possiede una propria dipendenza.
2	ContestoDiSicurezza	✓	
3	Dipartimento	✓	
4	e-Mail	✗	Probabilmente la relativa struttura non ne giustifica l'esistenza separata.
5	Filiale	✓	
6	FirmaDigitale	✗	Fuori scope.
7	GruppoUtenti	✓	
8	Indirizzo	✗	In questo contesto ha più senso considerarlo come un attributo.
9	Località	✓	
10	Log	✗	Fuori scope.
11	Login	✗	In questo contesto ha più senso considerarlo come un attributo.
12	ModuloAccesso	✓	
13	Motivazione	✗	Mancanza di indipendenza.
14	Nazione	✗	In questo contesto ha più senso considerarla come un attributo.
15	Notifica	✗	Evento fuori scope.
16	Password	✓	
17	PendingTask	✓	
18	Profilo	✓	
19	Record	✗	Fuori scope.
20	RegolaIncompatibilita	✓	
21	Report	✗	Fuori scope.
22	RuoloOrganizzazione	✓	
23	Schede	✗	Ridondante. Utilizzata come sinomino di Modulo di accesso.
24	Servizio	✓	
25	StadioModuloAccesso	✓	
26	Utente	✓	

Criterio 7: controlli finali

Superata anche l'analisi precedente, le povere candidate classi possono cominciare ad intravedere la luce del canvas di un diagramma delle classi. Gli ultimi controlli da effettuare sono relativi all'eventuale raggruppamento di diversi concetti in uno solo (la classe possiede una coesione molto blanda), al caso opposto, cioè la scomposizione di uno stesso concetto in più miniclassi (questa evenienza dovrebbe essere già stata eliminata dal crite-

rio precedente), all'eventuale presenza di classi ridondanti. Qualora ci si sia imbattuti nel primo caso (in genere ci si accorge dal fatto che le proprietà strutturali e comportamentali presentano uno scarso grado di coesione, in altre parole la classe possiede tante responsabilità/attributi non molto attinenti tra loro), è necessario eseguire una corretta suddivisione delle stesse. Eventualmente, è possibile riapplicare il processo di verifica per le nuove classi come ulteriore controllo.

Sebbene l'iter mostrato in precedenza possa sembrare piuttosto elaborato e non sempre agevole, va considerato che l'esperienza permette di valutare intuitivamente le classi candidate nel 90% dei casi, senza dover passare attraverso un'analisi di questo tipo. Questo processo fornisce comunque validi criteri per tutti i casi di indecisione.

Terminata anche questa fase è possibile inserire altre informazioni nella tabella che quindi assume una forma come quella riportata di seguito. La compilazione della tabella è sempre consigliabile anche perché fornisce una valida documentazione sia per comprendere il modello sia per future reingegnerizzazioni dello stesso. La fase successiva consiste nel cercare di collegare tra loro le classi sopravvissute, iniziando a dar vita ad una rete razionale di collegamenti: la prima versione del modello a oggetti del dominio.

Strutturazione delle classi

Introduzione

Le classi sopravvissute al processo di "selezione della entità" eseguito nel punto precedente avrebbero ben poco senso se destinate ad un ciclo di vita isolato. La situazione normale, al contrario, prevede che ciascuna di esse sia relazionata ad altre e vi scambi messaggi al fine di erogare i servizi tanto cari agli utenti del sistema.

La costruzione del modello, ed in particolare l'incastonamento delle varie classi nella rete di relazioni, tipicamente permette di individuarne ulteriori sfuggite all'analisi precedente o semplicemente non menzionate dagli stessi utenti.

Prima però di procedere nell'illustrazione dei criteri per la costruzione del modello a oggetti del dominio, vengono illustrati alcuni consigli e trabocchetti in cui, molto frequentemente, si rischia di cadere.

Il primissimo consiglio è stamparsi in mente una frase di Donald E. Knuth, tratta dal libro *The Art of Computer Programming*, che uno dei maestri dell'autore usava ripetere spesso: "l'ottimizzazione prematura è la radice di tutti mali". L'obiettivo delle primissime fasi dei processi di sviluppo dovrebbe essere quello di aumentare la comprensione del dominio oggetto di studio e dei requisiti utente. Dunque, è necessario modellare, nel modo più semplice possibile, lo spazio del problema e non quello delle soluzioni. La progettazione delle soluzioni va affrontata nelle fasi successive, quando si è acquisita una buona conoscenza del

dominio del problema e non si deve più fare i conti con le tipicamente ridotte conoscenze informatiche degli utenti. Questo vincolo è invece importantissimo nelle prime fasi del processo di sviluppo del sistema in cui è necessario coinvolgere il più possibile gli utenti ed esperti del dominio, instaurando una piattaforma costruttiva di dialogo tra le parti. In questo ambito, lo scopo fondamentale dei modelli (e quindi dello stesso UML) è comunicare (e poi documentare). Pertanto è necessario tenere a mente che molti utenti ed esperti del business dispongono di una limitata cultura informatica, ed è quindi consigliabile rimandare le ottimizzazioni alle fasi di analisi e disegno.

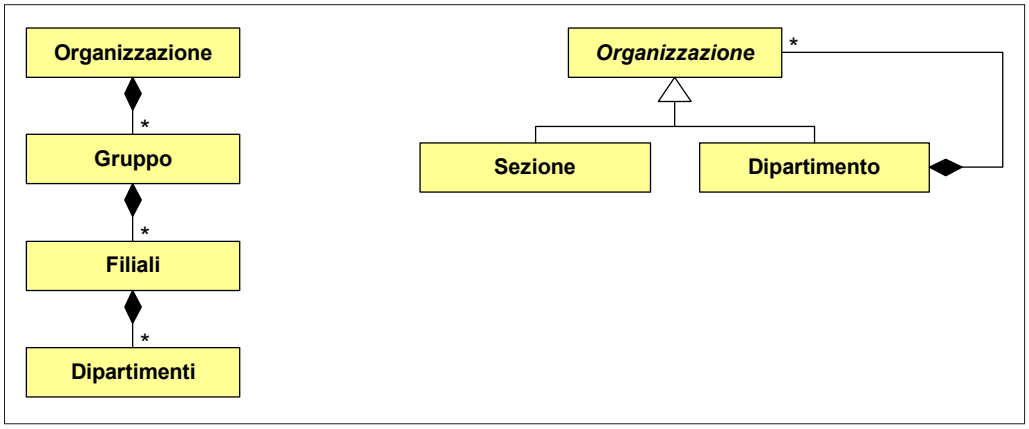
Il modello a oggetti del dominio non è il prodotto da implementare, si tratta di un artifact preziosissimo, ma destinato ad essere rifinito, attraverso altri modelli, nelle fasi successive.

Una delle anomalie che più frequentemente si rischia di generare, specie quando il modello a oggetti del dominio è realizzato da personale a forte connotazione tecnica (come architetti per esempio), è di introdurre ottimizzazioni, utilizzare design pattern e raffinate astrazioni che non sempre hanno molto a che fare con il dominio del problema. Ciò, se da un lato può concorrere ad aumentare l'eleganza e la leggibilità per una platea di fruitori esperti, dall'altro può risultare meno comprensibile invece ad un pubblico tecnicamente meno preparato come quello degli esperti del dominio e degli utenti/clienti. Ciò costituisce un grosso limite considerando che proprio questi soggetti vanno annoverati tra i fruitori principali del modello a oggetti del dominio: sono i migliori conoscitori dell'area business oggetto di analisi e quindi, alla fine, dovrebbero essere loro a validare il modello.

Disporre di un modello a oggetti del dominio non perfettamente OO è un "male" lieve in quanto curabile nelle fasi successive dal personale tecnicamente più preparato. Disporre di un modello che descrive in maniera errata la realtà è invece un problema decisamente più grave in grado di procurare grossi danni a tutto il processo.

Evitare prematuri artifici e raffinatezze è anche consigliato considerando che, generalmente, i modelli del dominio si prestano ad essere modificati diverse volte prima di approdare ad una versione (pseudo) stabile. Spesso gli stessi clienti/utenti non hanno ben chiare le entità che prendono parte nel sistema business e le relative astrazioni. Pertanto una precoce ricercatezza potrebbe presto rivelarsi una perdita di tempo.

Figura 8.7 — *Rappresentazione della struttura gerarchica dell'organizzazione. A sinistra vi è una rappresentazione più fedele ai requisiti, mentre a destra si è ricorsi all'utilizzo del pattern Composite.*



Chiaramente per ogni regola sono presenti le inevitabili eccezioni… In effetti, in alcuni casi proprio non si può fare a meno di usare qualche pattern anche per il modello a oggetti del dominio: parola di architetto! Si consideri l'esempio della strutturazione dell'organizzazione in filiali e dipartimenti. Si esaminino i due diagrammi riportati nella fig. 8.7.

Come si può notare quello di sinistra rappresenta lo spazio del dominio in modo più vicino a quanto riportato nei requisiti senza sofisticazioni, mentre quello di destra utilizza una rappresentazione decisamente più concisa ed elegante basata sul pattern Composite (come si vedrà tra breve, la forte connotazione gerarchica delle entità coinvolte nel dominio oggetto di studio porta alla celebrazione del pattern Composite che verrà utilizzato moltissimo nell'arco dell'intero capitolo). Ora, a meno di altri fattori, probabilmente nel modello del dominio è preferibile ricorrere alla versione base, mentre per modelli appartenenti a fasi più tecniche non si discute, è obbligatorio ricorrere a soluzioni più generali e versatili come quella fornita dall'utilizzo del pattern Composite. Chiaramente non sempre è possibile o conveniente evitare l'utilizzo di pattern o ricercatezze tecniche. In ogni modo è necessario ricorrere a queste soluzioni con molta parsimonia.

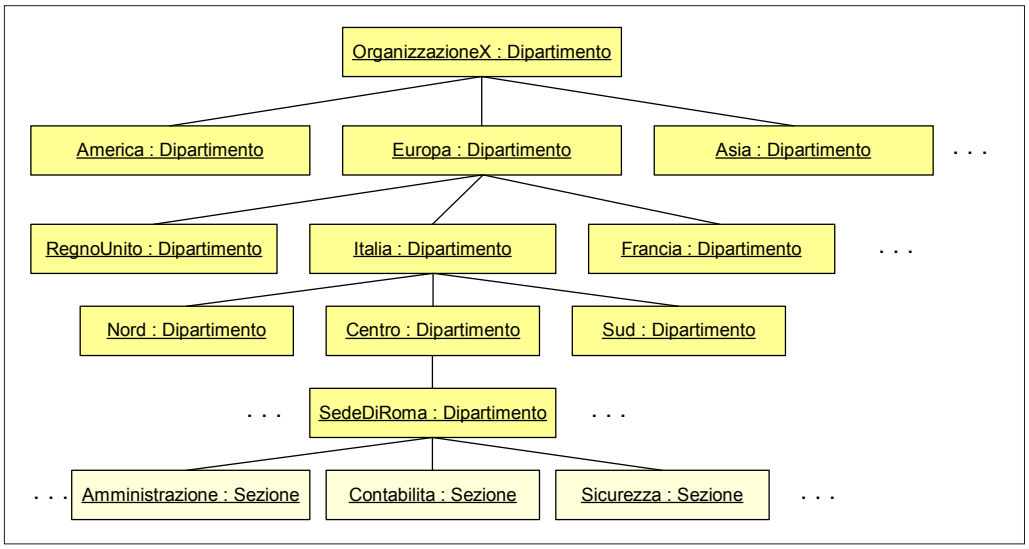
L'esempio mostrato probabilmente appartiene al caso in cui l'utilizzo del pattern è quasi indispensabile, sia perché la strutturazione definita nei requisiti utente non è del tutto chiara, sia perché la struttura delle varie filiali varia molto in funzione anche delle dimensioni. In alcuni casi, per esempio, può rendersi necessario ricorrere a una gerarchia molto dettagliata richiedente molti livelli, mentre in altri potrebbe essere sufficiente un numero esiguo di livelli. Il modello basato sul composite fornisce la flessibilità di poter generare, in maniera molto semplice, una struttura il cui livello di dettaglio è dipendente dalle necessi-

tà delle singole filiali e non da vincoli dettati da una struttura rigida preordinata. I fattori negativi sono la grande difficoltà nel far digerire tale modello a personale con limitate conoscenze tecniche e il dover introdurre nomi di classi che non necessariamente risultano sempre rispondenti. Per esempio (consultare fig. 8.8) la classe atta a rappresentare intere filiali prende il nome di dipartimento, così come l'intera organizzazione.

Qualora si decida di ricorrere all'utilizzo di soluzioni particolarmente sofisticate o pattern di non immediata comprensione, è consigliabile dar luogo a degli esempi atti ad illustrarne l'utilizzo pratico, magari per mezzo di opportuni diagrammi degli oggetti.

Sempre il personale tecnico tende ad inserire nel modello entità che rappresentano processi da eseguire (tipicamente queste classi hanno la caratteristica di possedere un nome che ha tutta l'aria di essere un'azione). Chiaramente non si tratta di un peccato mortale, però probabilmente è più opportuno non complicare inutilmente il modello, anche perché i processi di sviluppo, tipicamente, prevedono tutta una serie di manufatti in cui è possibile specificare sia azioni da svolgere, sia dettagli a carattere più squisitamente implementativo.

Figura 8.8 — *Esempio di applicazione del pattern Composite con particolare riferimento ad una ipotetica filiale dell'organizzazione (*`SedeDiRoma`*).*



Probabilmente non è neppure il caso di spendere un tempo eccessivo nell'investigare la particolare natura ed i dettagli delle associazioni. In questo stadio affermare che una relazione sia un'aggregazione piuttosto che di una composizione non cambia di molto la sostanza, mentre potrebbe creare problemi al solito cliente non tecnico frastornato dalla nuova simbologia. La differenza tra le due tipologie di associazioni ha molto a che vedere con il ciclo di vita degli oggetti partecipanti e quindi su scelte di disegno che, pertanto, esulano dagli obiettivi del modello a oggetti del dominio.

Alcuni autori consigliano anche di non perdere troppo tempo nello stabilire le molteplicità con cui le classi partecipano alle relazioni. L'autore non condivide questo suggerimento. Le cardinalità sono elementi molto importanti del modello a oggetti del dominio, concorrono a definire le regole del business, permettono di evidenziare eventuali lacune di relazioni o, viceversa, relazioni ridondanti, e così via. Per esempio un conto è asserire che un profilo possa contenere un solo contesto di sicurezza, un altro è sostenere che ne può contenere diversi, come esiste una certa differenza nell'asserire che un profilo appartenga ad un solo utente oppure a svariati.

Ciò che probabilmente è possibile, è non attribuire eccessiva importanza alle navigabilità delle relazioni, assumendo che ogni relazione sia navigabile in entrambi i versi. A meno di casi evidentissimi, la selezione della particolare navigabilità dovrebbe essere una scelta da compiersi sempre nella fase di disegno.

Per esempio si consideri la relazione tra un utente e un modulo di accesso, come riportato in fig. 8.9.

Ora, imporre una navigazione da `ModuloDiAccesso` verso `Utente` (il che equivale a porre la freccia all'estremità della classe `Utente` e navigabilità `ModuloDiAccesso` uguale a `false`) impone che, data un'istanza di un modulo di accesso, sia possibile ottenere l'utente a cui si riferisce (magari attraverso il metodo `getUser()`), mentre non è possibile il viceversa. Ciò però non equivale ad asserire che non si possa mai risalire ai moduli di accesso relativi ad uno stesso utente: è ancora possibile ma in modo decisamente più laborioso. Sarebbe necessario reperire dapprima l'istanza del cliente per poi poter individuare tutte le istanze dei relativi moduli di accesso. Ciò potrebbe avere notevoli impatti specie in un sistema component-based, qualora si progetti un componente specializzato per gestire gli utenti e uno per i moduli di accesso.

Figura 8.9 — *Esempio di navigabilità tra le classi* `Utente` *e* `ModuloDiAccesso`.

Prima di iniziare la costruzione del modello a oggetti del dominio, si ritiene opportuno sottolineare (ancora una volta) le parole d'ordine: **approccio minimalista**. Ciò è giustificato non solo al fine di utilizzare al meglio il tempo a disposizione, ma anche perché ci si trova in una delle primissime fasi del processo di sviluppo nella quale, tipicamente, non si dispone di una approfondita conoscenza del sistema. Per cui probabilmente, è meglio rimandare scelte di dettaglio a fasi successive.

Costruzione

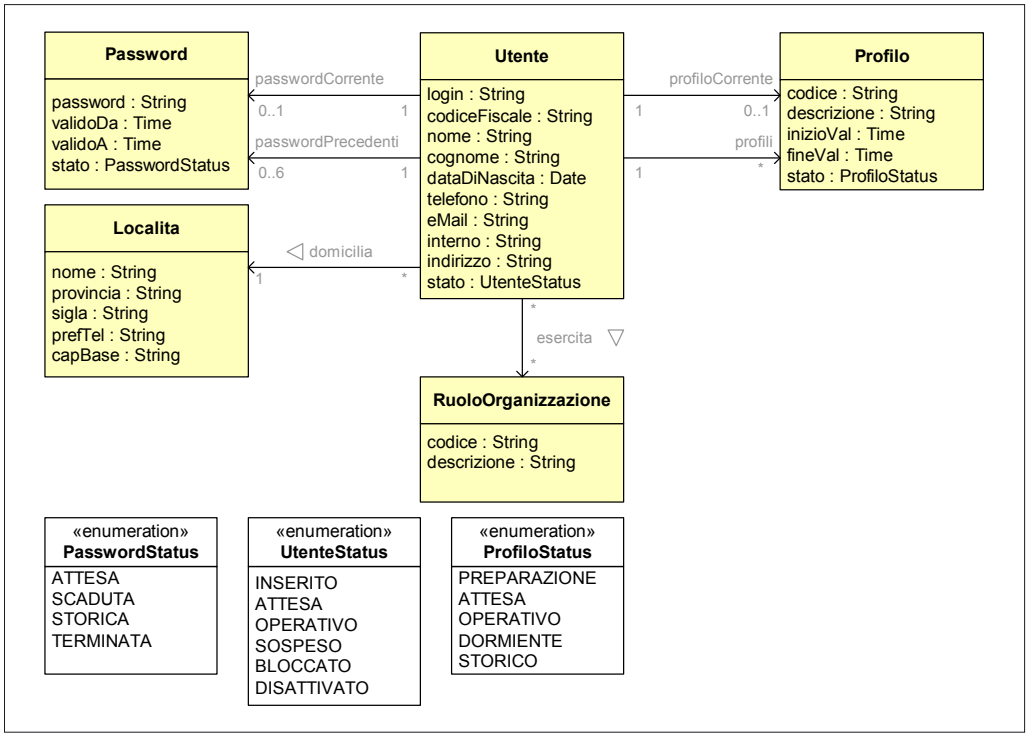
Una volta individuato l'elenco iniziale delle ipotetiche classi, una porzione importante del lavoro è compiuta. Un'altra attività abbastanza complessa consiste nell'organizzare le entità scaturite dal passo precedente in un modello organico. Il processo stesso offre un'ottima possibilità per rivedere le entità stesse alla luce delle relazioni con cui sono interconnesse. Nella pratica è abbastanza frequente che la costruzione del modello porti a scoprire classi mancanti ed a rielaborare quelle evidenziate.

Un buon metodo per dar luogo ad una prima versione del modello consiste nel riconsiderare i casi d'uso (o qualunque altro documento utilizzato nel passo precedente) focalizzando però l'attenzione sulle frasi che coinvolgono le classi evidenziate precedentemente. Durante questo processo è necessario conferire particolare attenzione ai verbi, i quali, tipicamente, rappresentano associazioni tra classi o relativi metodi.

Si cominci la costruzione del modello a partire dagli elementi basilari: gli utenti, i profili e le password (fig. 8.10). La descrizione della realtà oggetto di studio sancisce che un utente può disporre di diversi profili, dei quali però uno solo è nello stato `OPERATIVO` (e quindi utilizzabile) mentre ciascun profilo è riferito ad un solo utente. Di ciascun utente interessa conoscere, oltre i vari dati personali, anche il recapito. Gli utenti accedono al sistema attraverso la coppia (login, password). Quest'ultima ha tutta la dignità di essere considerata una classe in quanto aggrega diversi attributi ed in più una regola del business asserisce che un utente non può riutilizzare una delle sue ultime sei parole chiavi e quindi è necessario memorizzare le password utilizzate in precedenza da uno stesso utente.

Questa prima porzione di modello a oggetti fornisce lo spunto per diverse considerazioni. In primo luogo si considerino le relazioni che associano la classe `Utente` a quelle di `Password` e `Profilo`. In entrambi i casi si è deciso di decomporre la relazione in altre due: una che mostri l'associazione con l'istanza "operativa" dell'altra classe e l'altra che

Figura 8.10 — *Primo embrione del modello a oggetti del dominio, focalizzato sulle entità* `Utente`, `Password` *e (parzialmente)* `Profilo`.



mostri le rimanenti istanze. In un modello di disegno probabilmente non si tratterebbe della migliore scelta, tra l'altro potrebbe rendere il codice meno semplice (si correrebbe il rischio di complicare le iterazioni relative a tali associazioni), mentre nel modello a oggetti del dominio si tratta di una scelta felice in quanto concorre ad evidenziare delle regole di business.

La classe `RuoloOrganizzazione` si presterebbe ad essere sostituita anch'essa dal pattern Composite. In effetti è lecito attendersi una struttura gerarchica di questi ruoli. In questo contesto si è però optato per la soluzione più semplice (lista di ruoli a pari livello).

A questo punto è il momento di considerare il modulo di accesso. In particolare un modulo di accesso si riferisce sempre a un solo utente (richiesta di blocco, sblocco, sospensione, ecc.) ed eventualmente anche a un profilo a questi riferito (richiesta nuovo profilo, modifica di uno esistente, disattivazione, …).

Come visto nel relativo diagramma degli stadi, un modulo di accesso, prima di dar luogo ad eventuali effetti, deve transire attraverso un insieme ben definito di stadi. La

transizione è fornita da una specifica operazione effettuata da utenti appartenenti ad opportuni ruoli di sicurezza. Per esempio un Line Manager sottopone la richiesta, un addetto della sicurezza la valida, il supervisore l'approva. Queste informazioni si prestano ad essere memorizzate nella classe `ModelloAccessoStadio`, il cui compito è di memorizzare l'utente che ha generato una transizione (compilazione, validazione, ecc.), la data e l'orario, le eventuali note aggiuntive e lo stadio in cui il modulo di accesso è transìto. Questa classe tra l'altro è molto importante per effettuare controlli relativi alla segregazione dei ruoli.

Il modello riportato in fig. 8.11 si presta ad essere reso più formale attraverso l'introduzione di un insieme di vincoli che, tra l'altro, permettono di evidenziare business rule molto importanti.

Ogni qualvolta si ha a che fare con dei vincoli, la tecnica di documentazione considerata più opportuna consiste nell'inserire i diagrammi (suddivisi per package) in una o più pagine di un apposito word processor e quindi redigere un'apposita sezione con i vari vincoli. Questi dovrebbero essere specificati dapprima per mezzo del linguaggio naturale e poi attraverso un metodo a maggiore grado di formalità, come per esempio l'OCL.

Esempi di vincoli:

1. il profilo a cui si riferisce un modulo di accesso deve appartenere all'utente a cui fa riferimento lo stesso modulo di accesso;

```
context ModuloAccesso inv:
   self.utente.profiloCorrente <> self.profilo implies
      self.utente.profili->select(Profilo = self.profilo)->isNotEmpty()
```

2. un utente non può eseguire alcuna operazione (compilazione, validazione, approvazione, ecc.) relativa ad un modulo di accesso riferito a sé stesso.

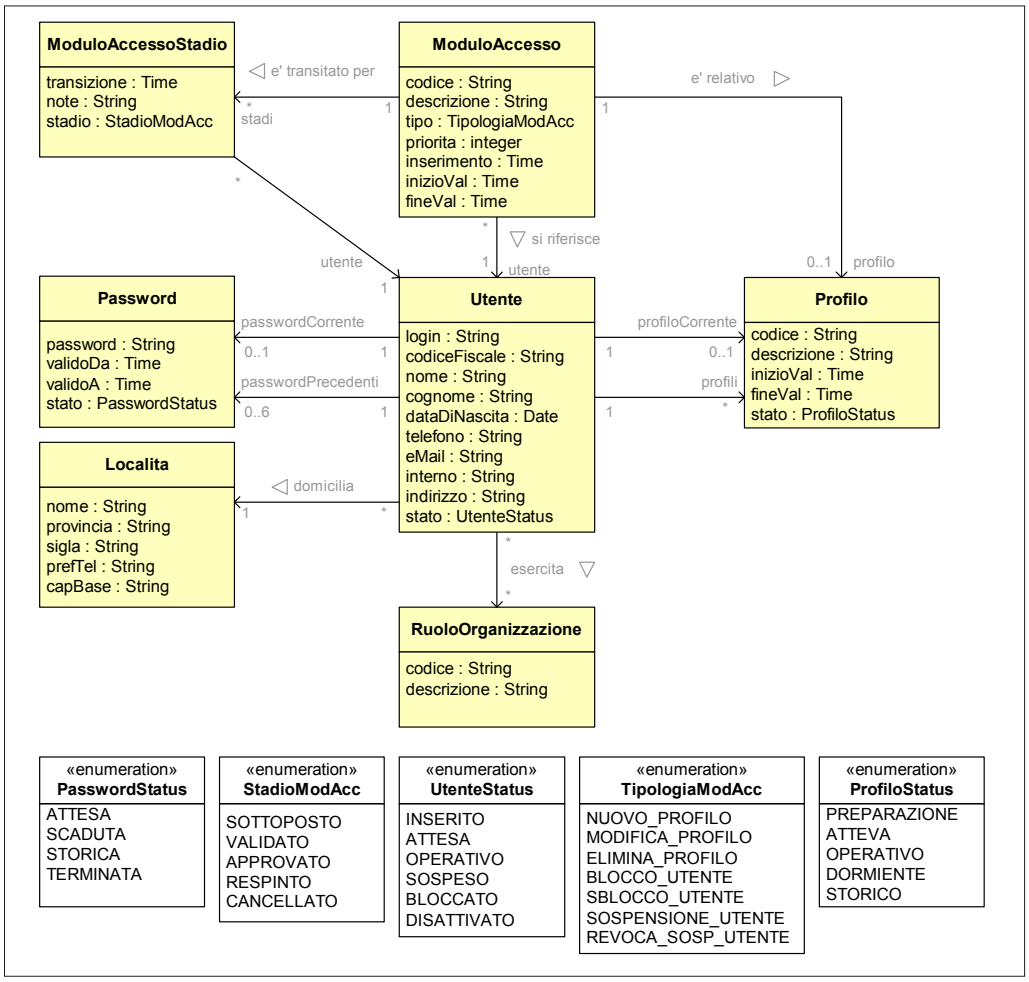
```
context ModuloAccesso inv:
   self.stadi->select(ModuloAccessoStadio.utente = self.utente)->isEmpty()
```

Il termine context definisce il contesto a cui fa riferimento il vincolo, mentre inv rivela un tipo particolare di vincolo denominato invariant (invariante). Brevemente un invariante è un vincolo associato ad un insieme di classificatori e/o di relazioni ed indica che la condizione espressa dal vincolo deve essere soddisfatta da tutte le istanze del classificatore e/o relazioni durante tutto il relativo ciclo di vita.

Queste invarianti, chiaramente, devono essere aggiornate man mano che si procede con il raffinamento del modello.

Dall'analisi del modello è possibile notare che l'attributo `ModuloAccesso`, `TipologiaModAcc` è un'ottimizzazione di una relazione di generalizzazione, in effetti si dovrebbero visualizzare una

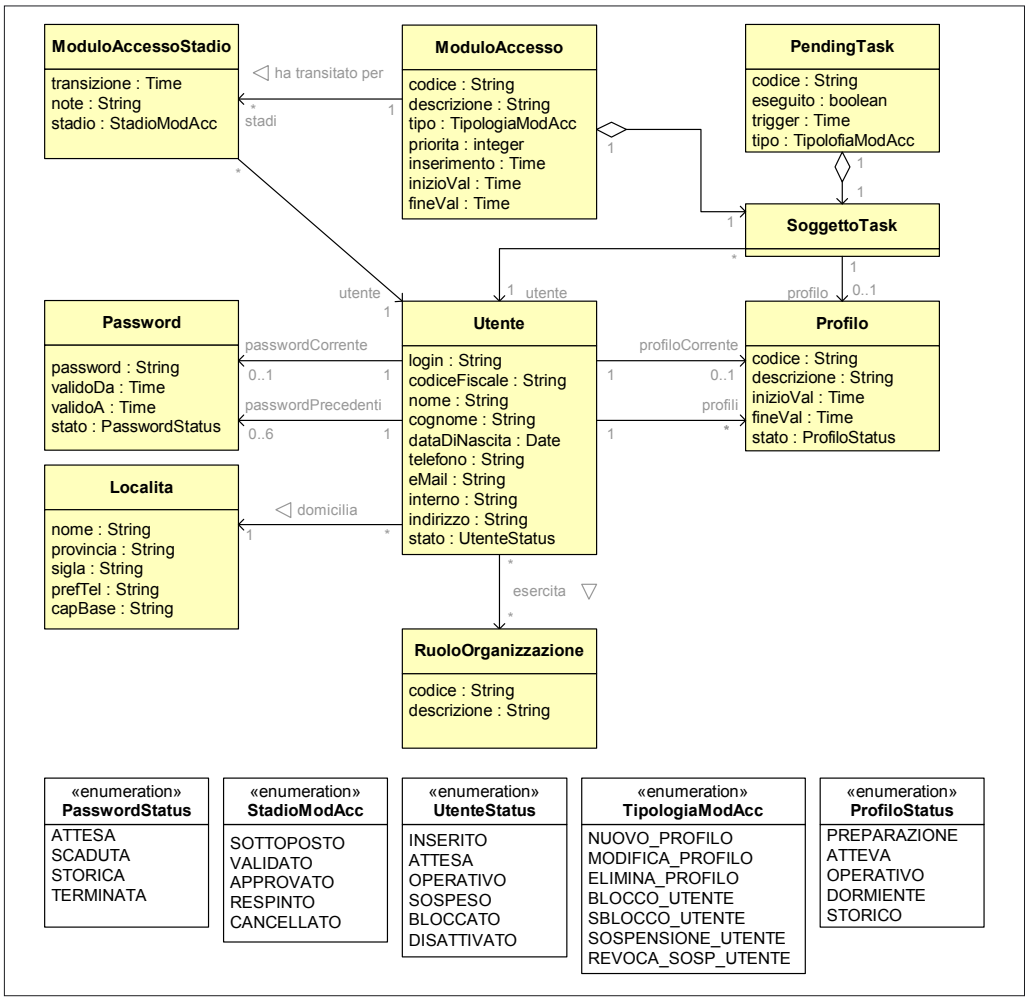
Figura 8.11 — Introduzione dell'entità `ModuloAccesso` nel modello.



serie di specializzazioni della classe `ModuloAccesso`: nuovo profilo, aggiornamento profilo, disattivazione profilo, sospensione utente, fine sospensione, blocco utente, fine blocco utente, ecc. Si è deciso di non visualizzare esplicitamente le varie specializzazioni al fine di non complicare ulteriormente (e forse inutilmente) il diagramma.

A questo punto è possibile considerare la classe le cui istanze rappresentano i lavori da eseguire prevalentemente in modalità batch: i famosi lavori pendenti (*pending task*). Come

Figura 8.12 — *Introduzione della classe* `PendingTask`.



riportato nel paragrafo dei requisiti utente, principalmente le istanze di questi lavori sono generate a seguito dell'approvazione di un modulo di accesso. La durata degli effetti delle richieste è specificata dagli attributi di validità (`ModuloAccesso.inizioVal` e `ModuloAccesso.fineVal`) del modulo di accesso. Questi attributi, congiuntamente con `ModuloAccesso.tipo : TipologiaModAcc`, stabiliscono quanti oggetti `PendingTask` il sistema deve generare. Per esempio un modulo di accesso relativo a una sospensione genera due task: sospensione utente nella data e orario di `inizioVal` e

rimozione sospensione nella data e orario di `fineVal`. Così come avviene nel caso di un nuovo profilo. In alcuni casi, come per esempio una richiesta di blocco utente, o di disattivazione profilo, viene generato un unico task e l'attributo `fineVal` non viene preso in considerazione.

Sempre dal paragrafo dei requisiti utente si evince che alcune azioni (blocco, sblocco, sospensione, …) possono essere eseguite direttamente da un addetto della sicurezza senza dover passare per l'iter del modulo di accesso. In tal caso i lavori non sono prenotati e quindi espletati in modalità batch, ma eseguiti direttamente.

Dall'analisi di fig. 8.12 si può effettuare una serie di considerazioni:

- l'introduzione della classe `PendingTask` ha generato la necessità di ristrutturare la classe `ModuloAccesso`. Volendo si sarebbe potuto pensare di associare direttamente la classe `PendingTask` alla classe `ModuloAccesso`. Ciò però avrebbe generato un modello (e quindi una descrizione del problema) errato giacché non tutti i task sono il risultato della sottomissione di un modulo di accesso. Eventualmente si sarebbe potuto evidenziare il legame inverso: il modulo di accesso e i lavori pendenti scaturiti dalla relativa approvazione. Anche questo è stato evitato in quanto, una volta approvato, un modulo di accesso termina il relativo ciclo di vita;

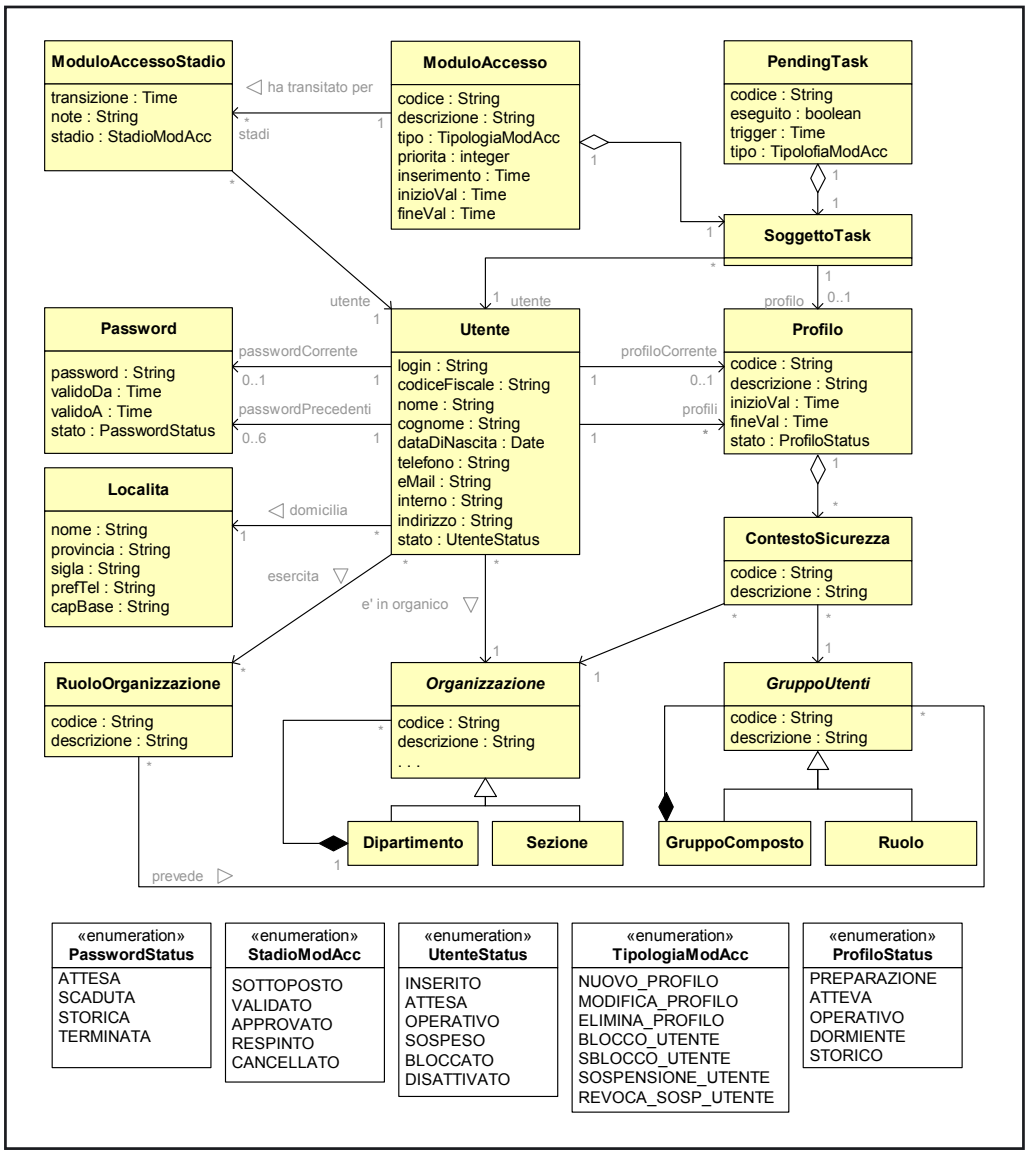
- le due classi sono associate alla classe `SoggettoTask` per mezzo di una relazione di aggregazione. Si sarebbe potuto selezionare anche una relazione di composizione (sebbene la differenziazione appartiene di più ad un modello di disegno). Come ribadito in precedenza, in questa fase non è il caso di investire tempo in questo tipo di speculazioni. Comunque, qualora si fosse deciso di utilizzare una composizione si sarebbe implicitamente sancito l'importante vincolo che una stessa istanza della classe `SoggettoTask` non possa appartenere **allo stesso tempo** a più oggetti.

A questo punto, il passo successivo consiste nel dettagliare il profilo. Qui inizia qualche problemino…

In primo luogo un profilo deve dichiarare i gruppi utente a cui l'utente può appartenere e i dipartimenti (`Organizzazioni`) in cui i relativi servizi sono usufruibili. Entrambe le entità si prestano ad essere modellate attraverso l'ormai famosissimo pattern Composite. Un esempio di utilizzo è fornito attraverso il diagramma degli oggetti riportato in fig. 8.14.

Poiché un utente, tramite un medesimo profilo, può appartenere a più gruppi è stata introdotta la classe `ContestoSicurezza`, la cui funzione è di accoppiare un'istanza del gruppo utenti ad una sezione in cui l'utente è abilitato ad utilizzarla. Come si può notare, ciascuna istanza della classe `ContestoSicurezza` accoppia esattamente un ruolo ad un dipartimento, quindi, qualora lo stesso ruolo possa essere utilizzato in diversi dipartimen-

Figura 8.13 — *Modellazione dettagliata del profilo utente.*

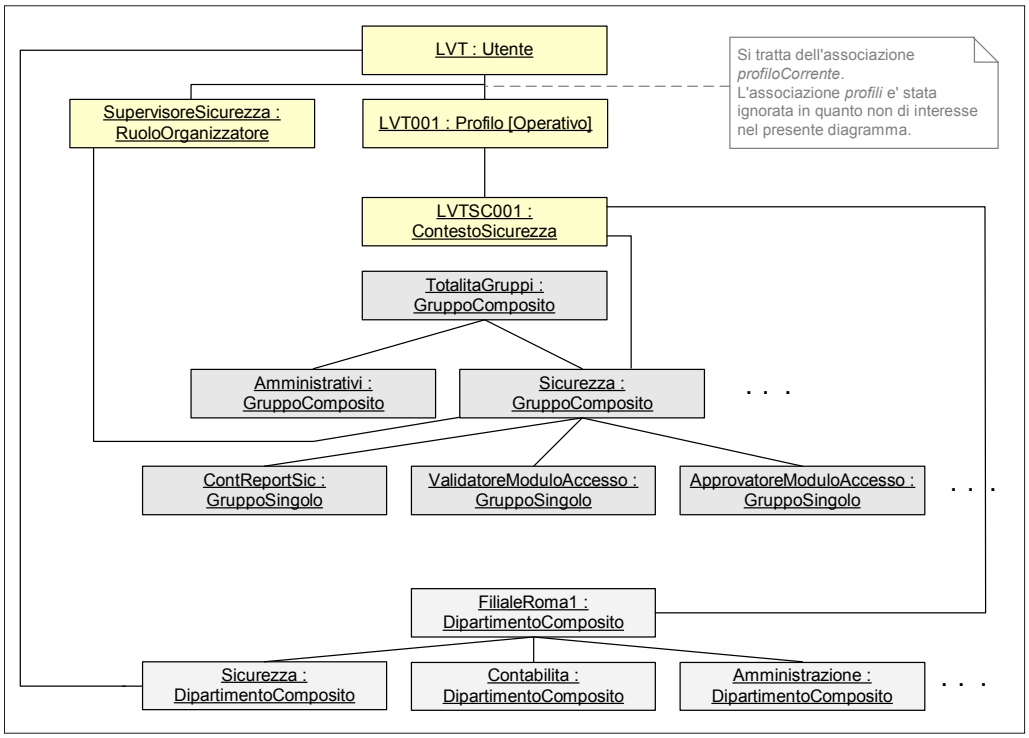


ti, potrebbe rendersi necessario associare al profilo più contesti di sicurezza con il medesimo ruolo e diversi dipartimenti. Molto spesso ciò viene evitato associando il ruolo al dipar-

timento "padre" e quindi includendo automaticamente i vari dipartimenti dipendenti. Un discorso del tutto analogo vale qualora un utente possa utilizzare diversi ruoli nel contesto di uno stesso dipartimento.

Poiché uno stesso ruolo nell'organizzazione può necessitare l'abilitazione delle funzionalità associate a diversi gruppi utente, è stata realizzata un'associazione (*n,n*) tra la classe `RuoloOrganizzazione` e `GruppoUtenti`. Ciò permette al sistema di suggerire i gruppi utente da impostare ogni qualvolta un `Line Manager` compili un `ModuloAccesso` per la richiesta di un nuovo profilo relativo a un suo subalterno: è sufficiente navigare dall'istanza `RuoloOrganizzazione` relativa all'utente a tutte le istanze ruolo utente a cui è associata per mezzo della relazione `prevede`. Questa associazione, tra l'altro, fornisce ulteriori criteri di valutazione agli addetti alla sicurezza. Infatti permette di evidenziare e quindi controllare i casi (eventualmente legittimi) in cui un Line Manager richieda di abilitare, per un proprio subalterno, l'esecuzione di servizi supplementari rispetto a quelli necessari per le mansioni previste dal relativo ruolo nell'organizzazione.

Figura 8.14 — *Diagramma degli oggetti mostranti un primo esempio di profilo.*



Quanto riportato permette di rinforzare un principio molto importante dei sistemi di sicurezza noto come il nome di "minore livello possibile di privilegi" (*Least Privilege*). Questo prevede di assegnare a ciascun utente esclusivamente i privilegi strettamente necessari per eseguire le proprie mansioni. Per rendere operativo questo criterio è necessario dettagliare il più possibile l'elenco dei servizi fruibili, o meglio dei ruoli utilizzabili per accedere al sistema (si parla di granularità fine). Per esempio, invece di disporre di un unico gruppo utente per gli addetti alla sicurezza, è più opportuno considerarne diversi, come *visualizzatori dati utente*, *visualizzatori dati moduli di accesso*, *validatori moduli di accesso*, *visualizzatori dati di log*, e così via. Pertanto il ruolo *organizzazione* resterebbe addetto alla sicurezza, mentre i gruppi utenti dichiarati nel profilo dell'utente attraverso i contesti di sicurezza sarebbero quelli testé citati. Ciò evidentemente comporta un notevole overhead nella gestione del sistema, che però è il prezzo da pagare qualora si voglia limitare il più possibile l'esposizione del sistema e i pericolosi effetti generabili da un attacco riuscito al sistema di sicurezza.

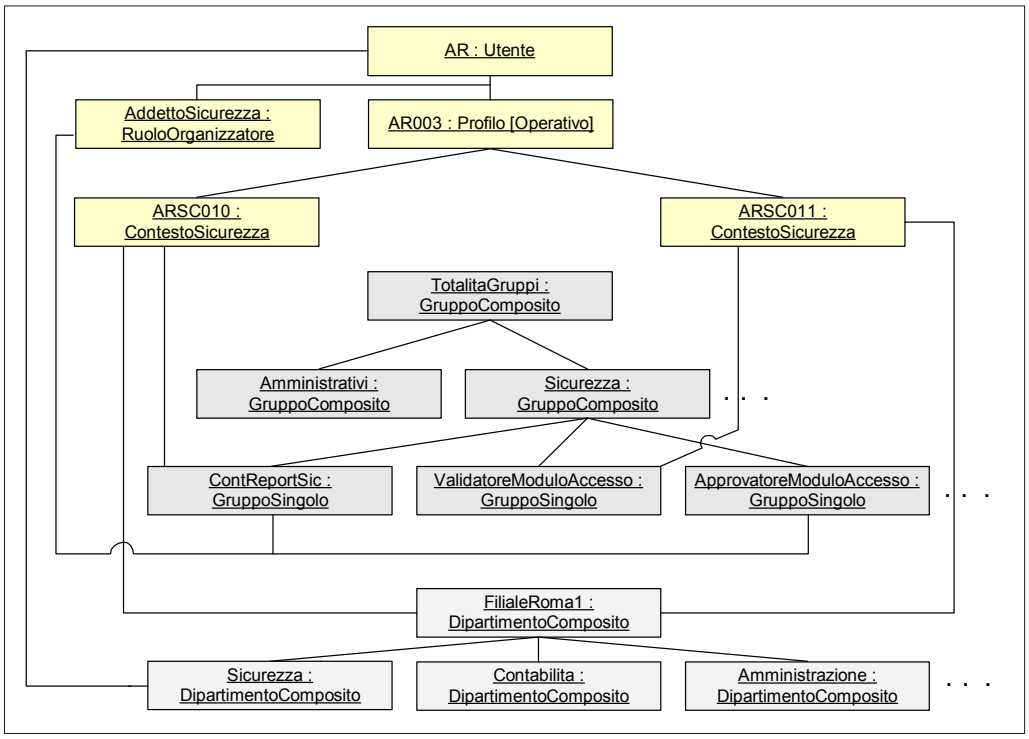
Nell'esempio di fig. 8.15 è mostrato l'utilizzo del pattern Composite, sia quello associato ai ruoli utente, sia quello relativo all'organizzazione dei dipartimenti.

Per quanto concerne il primo, l'insieme di tutti i ruoli è suddiviso, in prima istanza, in quelli appartenenti al settore amministrativo, alla sicurezza e così via. Il sottoinsieme dei ruoli legati alla sicurezza prevede gli elementi di controllore report sicurezza, validatore moduli di accesso, approvatore moduli di accesso ecc. Per quanto riguarda l'applicazione del pattern ai dipartimenti, si è preso in considerazione un primo livello di un'ipotetica filiale. L'utente, all'interno dell'organizzazione, esercita il ruolo di supervisore della sicurezza e pertanto è abilitato ad eseguire tutti i ruoli previsti dal relativo dipartimento. Come si può vedere l'istanza `RuoloOrganizzazione` è associata all'istanza radice del gruppo di ruoli utente appartenenti all'area della sicurezza abilitando automaticamente tutti i ruoli da esso gerarchicamente dipendenti a qualsiasi livello. Il profilo operativo è identificato dal codice `LVT001` che contiene una sola istanza di `ContestoSicurezza`: `LVTSC001`.

Questa abilita l'utente ad usufruire del proprio profilo per qualsiasi dipartimento della filiale di Roma. L'utente poi è (come era lecito attendersi) in organico al dipartimento di sicurezza, pertanto chiunque voglia vederne le relative informazioni ed eventualmente aggiornarle, deve possedere un opportuno ruolo abilitato per il dipartimento di sicurezza o un suo "genitore" nella struttura dell'organizzazione, come per esempio accade all'utente preso in considerazione.

In questo secondo esempio, l'utente esercita il ruolo di addetto alla sicurezza e quindi è abilitato ad esercitare un sottoinsieme più limitato di ruoli. Uno dei vantaggi dell'utilizzo del pattern Composite consiste nella flessibilità che fornisce sia nell'organizzare il grafo gerarchicamente, sia nell'associare le varie classi. Nell'esempio dei ruoli è possibile avere un contesto di sicurezza in grado di comprendere tutti i ruoli dell'organizzazione, tutti quelli di un dipartimento o un ruolo singolo.

Figura 8.15 — *Diagramma degli oggetti mostranti un secondo esempio di profilo.*

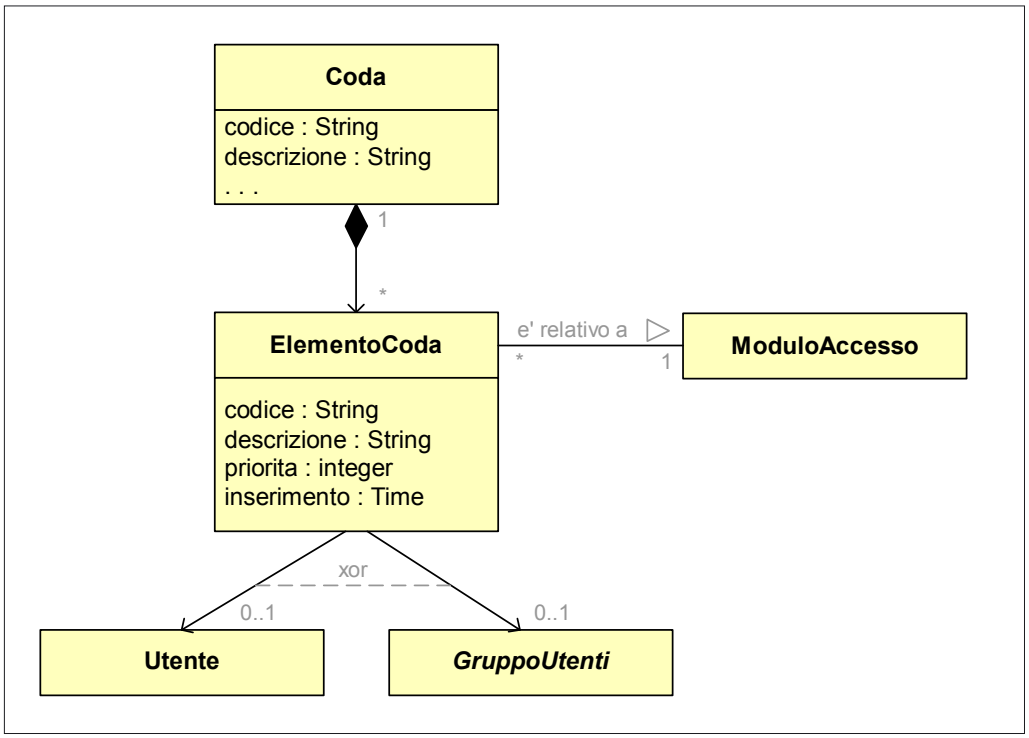


Rispetto alla descrizione del dominio del problema, è necessario modellare due ulteriori aree: quella relativa alle regole di incompatibilità e quella efferente dalle politiche di sicurezza. Entrambe vengono trattate nella paragrafo dedicato alla tecnica alternativa al fine di non appesantire eccessivamente la presente trattazione.

A questo punto, prima di terminare il paragrafo, si vuole cogliere l'occasione per evidenziare un problema che tipicamente si manifesta nell'attività di cattura dei requisiti utente… Spesso accade, per cattiva fede o semplicemente per mancanza di conoscenza, che gli intervistati dimentichino di menzionare alcune importanti entità del modello. In questi casi l'esperienza nella costruzione di sistemi (che funzionano) recita un ruolo di fondamentale importanza. Si analizzi il modello… Forse manca qualcosa…

Se è vero che i moduli di accesso debbano passare attraverso una serie di fasi prima di poter dar luogo ai relativi effetti, come fanno i relativi utenti (l'addetto della sicurezza che ne effettua la validazione, il supervisore che lo approva, il line manager che eventualmente deve modificarlo a seguito di un rigetto, …) a essere avvisati?

Figura 8.16 — *Code di lavori a carico dei vari utenti.*



Come si può notare si tratta di una questione che ha a che fare con i meccanismi da prevedere per la notifica, ma anche con l'organizzazione di specifiche entità. Per quanto concerne i primi (che tra l'altro dovrebbero essere di interesse relativo per il modello a oggetti del dominio) la risposta è abbastanza semplice: in primo luogo bisognerebbe costruire un motore per la gestione dei workflow, e poi bisognerebbe prevedere un sistema di visualizzazione delle notifiche. Quest'ultimo potrebbe essere realizzato prevedendo una funzione sul lato server da eseguire ad ogni richiesta dell'utente. Questa si dovrebbe occupare di verificare se ci sono lavori pendenti per lo specifico utente connesso o per i gruppi utente presenti nel relativo profilo. In caso di esito positivo si potrebbe visualizzare un apposito messaggio o un'icona in una specifica sezione della pagina HTML generata come risposta alla richiesta dell'utente.

Per quanto concerne le entità il discorso è un po' più complesso. In primo luogo si può notare che esistono alcune attività a carico di uno specifico utente: un `Line Manager` sottopone un modulo di accesso; se questo viene respinto, solo il medesimo `Line Mana-`

`ger` è abilitato a decidere il da farsi (modificarlo e quindi riprovare a sottometterlo oppure annullarlo direttamente). Altre invece sono a carico di un gruppo. Per esempio quando un line manager sottopone un modulo di accesso, questo può essere validato da un qualsiasi addetto alla sicurezza che abbia però accesso ai dati del dipartimento a cui appartiene l'utente a cui è riferito il modulo di accesso stesso. Chiaramente i dati relativi al modulo di accesso appartengono al medesimo dipartimento cui appartiene l'utente oggetto del modulo in questione.

Quanto descritto si presta ad essere modellato dal diagramma riportato in fig. 8.16.

Per quanto concerne l'attributo `priorita` della classe `ElementoCoda`, questo potrebbe essere impostato all'atto dell'inserimento di un nuovo oggetto nella coda secondo criteri ben definiti. Per esempio è abbastanza lecito attendersi che moduli di accesso con data inizio validità più ravvicinata abbiano priorità più elevata di altri più in là nel tempo, così come richieste per blocco utente siano prioritarie rispetto ad altre di tipo diverso. La priorità però non dovrebbe essere fissa, ma dovrebbe prevedere una soglia massima iniziale assegnabile all'atto dell'inserimento e quindi aumentare in modo proporzionale al tempo di permanenza di un elemento nella coda. A tal proposito si potrebbe prevedere un servizio (un daemon) che si attivi ad intervalli di tempo prestabiliti il cui compito sia scorrere la coda dei lavori, aumentando eventualmente la priorità degli elementi presenti. Qualora poi il valore della priorità superi dei valori di guardia, il servizio si dovrebbe occupare di emettere opportune segnalazioni di warning affinché si prendano dei provvedimenti per il lavoro rimasto in coda da troppo tempo.

Conclusioni

Come accennato in precedenza, il metodo presentato è stato oggetto di critiche alquanto severe. Prima di analizzarne alcune, l'autore crede che sia necessario inquadrare, ancora una volta, il procedimento nel corretto contesto. Esso non promette di fornire una metodologia sufficiente a garantire l'identificazione di tutte e sole le classi appartenenti all'area business oggetto di studio, l'obiettivo non è la produzione della versione definitiva del modello a oggetti; bensì l'elaborazione di una primissima versione, un embrione, da utilizzare per condurre i propri studi ed instaurare una piattaforma costruttiva di dialogo con gli utenti e gli esperti del dominio.

Dall'analisi delle principali critiche è possibile notare come queste siano relative più ai problemi tipici dell'analisi dei requisiti (investigate nei capitoli relativi ai casi d'uso) che non al procedimento stesso. Chiaramente un'errata analisi dei requisiti tende a generare problemi seri a qualsiasi processo. Tuttavia si è ritenuto opportuno riportare le varie critiche al fine di rendere i lettori consapevoli degli eventuali rischi intrinseci nel metodo, anche al fine di poter pianificare eventuali contromisure, come confronti con gli utenti, verifiche con software esistenti, ecc.

Una delle critiche più forti è relativa al principale manufatto di input: il documento dell'analisi dei requisiti che, come esperienza insegna, non bisognerebbe mai concedere

eccessiva fiducia. Probabilmente un buon modello dei casi d'uso ridurrebbe il rischio. Ovviamente negli esempi si utilizzano costruzioni grammaticali e descrizioni a partire dalle quali chiunque sarebbe capace di individuare classi, relazioni, attributi, ecc. Nella realtà però le cose non sono mai così semplici… All'autore del libro è capitato svariate volte di trovarsi di fronte voluminosi documenti di analisi dei requisiti, e dopo diverse ripetute letture ritrovarsi con uno sguardo smarrito ad interrogarsi su cosa voleva dire il redattore ("ma cosa deve fare il sistema?"). Anche quando il documento e/o il modello dei casi d'uso risultano ben realizzati, non si può nascondere che ambiguità, assunzioni implicite — chiaramente per chi ha redatto il documento — reticenze, ecc. sono peculiari del linguaggio naturale. Queste caratteristiche tendono a generare errate interpretazioni e, come logica conseguenza, possono determinare la produzione di modelli inesatti.

Un'altra critica al metodo è relativa all'ambiguità e flessibilità tipica del linguaggio naturale che, chiaramente, tendono a ripercuotersi negativamente sul modello ottenuto dall'analisi delle specifiche. In questo caso però l'autore non è convinto che si tratti veramente di un problema troppo gravoso. Se si considera che, in questo contesto, l'obiettivo è produrre una versione iniziale del modello a oggetti del dominio che rappresenti una visione statica dell'area oggetto di studio, si capisce come ciò possa rappresentare una preoccupazione relativamente marginale.

Un problema più importante legato sia all'ambiguità e flessibilità del linguaggio naturale sia a reticenze ed assunzioni implicite, è la non descrizione di alcune entità che invece risultano elementi importanti per il modello a oggetti del dominio (si consideri il caso delle code dell'esempio precedente). Come al solito però non si tratta di un problema del metodo bensì della cattura dei requisiti in generale, che può essere minimizzato sottoponendo le versioni iniziali del modello a oggetti del dominio ad architetti esperti nella costruzione di sistemi.

Un metodo alternativo

Nella realtà accade raramente di dover realizzare un sistema inedito per il quale non esista alcun software in circolazione. Molto più frequentemente accade di dover reingegnerizzare un sistema esistente o di dover analizzare un dominio per il quale esistano in commercio software che affrontano problematiche simili. In tutti questi casi è possibile prendere delle scorciatoie traendo vantaggio da queste situazioni.

Per esempio, si può generare il modello del dominio tramite l'analisi dello schema della base di dati. Ciò dovrebbe aiutare a produrre una prima versione del modello o a verificare quella prodotta, per eliminare eventuali classi non necessarie, ridondanti, vaghe o non corrette, per introdurne altre mancanti, e così via. Se si utilizza questa strategia, il modello prodotto va considerato una versione iniziale da discutere e validare con il personale esperto del business.

Sebbene esistano diversi software in grado di eseguire automaticamente il reverse engineering degli schemi dei database, è consigliabile svolgere questo processo manualmente o quantomeno rivedere il modello generato dal tool. Ciò per diversi motivi. In primo luogo, nel modello a oggetti del dominio, verosimilmente, si è interessati ad un numero limitato di tabelle e di campi rispetto a quelli che compongono la base di dati. Tipicamente, molti di questi elementi hanno un significato strettamente legato all'implementazione o alla strutturazione dello stesso, e quindi hanno poco o nulla a che fare con lo spazio del problema. Inoltre è necessario revisionare il modello per conferire un aspetto più "object oriented" di quello che si otterrebbe attraverso una mera traduzione delle tabelle in relative classi. Infine è importante rivedere il modello al fine di adeguarlo alla visione del dominio posseduta dell'azienda committente.

Figura 8.17 — *Porzione di uno schema di un database relazionale utilizzato per memorizzare le regole di incompatibilità.*

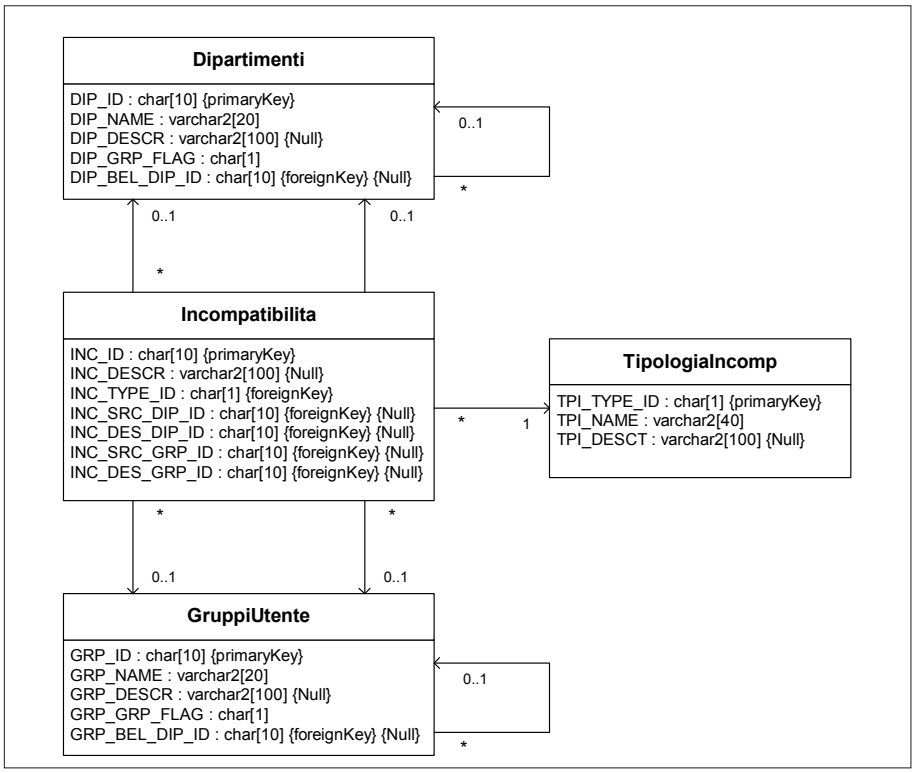
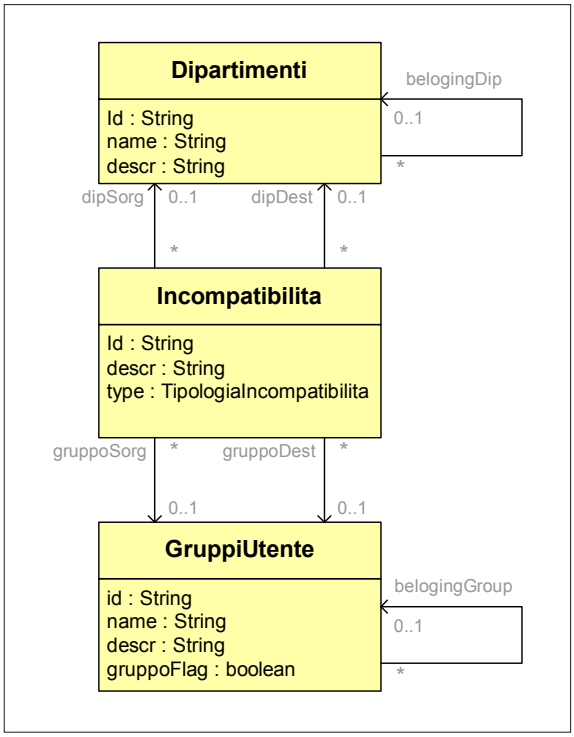


Figura 8.18 — *Esempio di diagramma delle classi ottenuto per mezzo di reingengerizzazione automatica di un frammento della base di dati.*



Durante questo esercizio, è necessario tener ben presenti le regole della particolare tipologia del database management system (relazionale, a oggetti, gerarchico, reticolare) oggetto di analisi. Se, per esempio, il paradigma è a oggetti, allora l'impegno richiesto per "estrarre" il modello del dominio è veramente minimo, mentre negli altri casi l'esercizio può risultare anche molto complesso.

Nella fig. 8.17 è riportato un primo estratto semplificato della base di dati utilizzata per memorizzare le regole di incompatibilità menzionate nella descrizione del dominio. La notazione adoperata è un esempio di come utilizzare i diagrammi delle classi per rappresentare degli schemi di basi di dati relazionali. Chiaramente non si tratta dello strumento più adatto, ma offre il grande vantaggio di realizzare diagrammi facilmente inseribili nel file rappresentante il proprio progetto e di non richiedere tool aggiuntivi.

Prima di iniziare l'analisi di quanto riportato, è necessario tener presente che nei database relazionali il legame tra le tabelle si realizza tramite esportazione delle chiavi primarie. Per esempio ciascun record della tabella `Incompatibilita` può essere relazionato a due

record `GruppiUtente`, ciò si evince dalla presenza nel tracciato record della prima tabella di due ripetizioni della chiave primaria della tabella `GruppiUtente (INC_SRC_GRP_ID, INC_DES_GRP_ID)`.

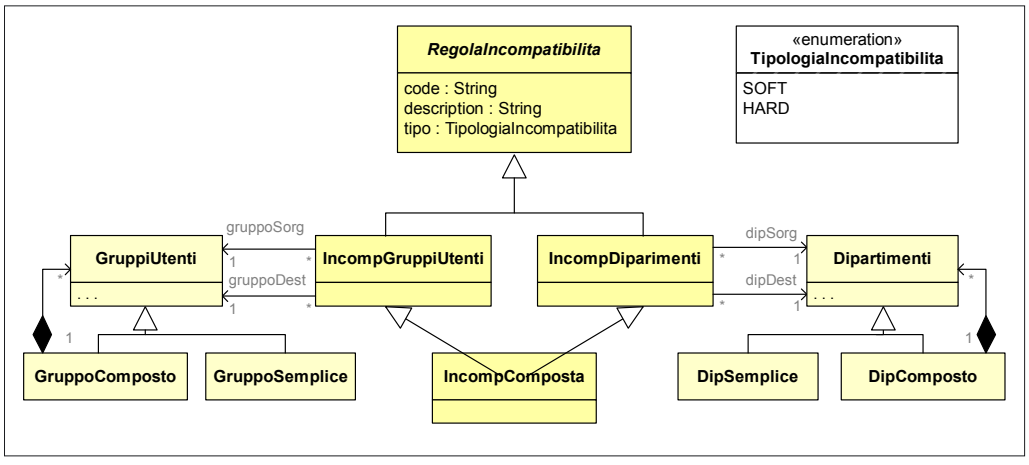
L'esempio delle regole di incompatibilità è stato selezionato per mostrare come una mera traduzione dello schema in un diagramma a oggetti tenderebbe a genere qualche problema. In questo contesto si potrebbe ottenere un diagramma come quanto evidenziato nella fig. 8.18.

Sebbene non si tratti poi del diagramma peggiore al mondo, comunque presenta diverse anomalie, come nascondere alcune regole business di una certa importanza. Per esempio non è chiaro se tutti i legami della classe incompatibilità posso essere omessi (hanno cardinalità del tipo `0...1`), oppure solo alcuni o coppie di essi; ancora, non è chiaro se sia possibile specificarne uno solo relativo al gruppo utenti ed uno solo al dipartimento, e così via.

Nell'analisi di uno schema di un database è necessario tenere presente che molto spesso la struttura finale è risultato di un processo di denormalizzazione effettuato allo scopo di migliorare le prestazioni delle operazioni più ricorrenti (tipicamente si giunge fino alla terza forma normale — 3NF — e quindi si denormalizza). L'effetto di questo processo è che spesso si accorpano alcune tabelle, oppure vengono inserite informazioni ridondanti per evitare ricerche troppo complesse, e così via.

Analizzando attentamente le tabelle relative alle regole di incompatibilità, e in particolare a `Dipartimenti` e `GruppiUtente`, è possibile notare come queste si prestano a memorizzare dati secondo un'organizzazione ad albero. Ciò nel mondo della modellazione

Figura 8.19 — *Esempio di diagramma delle classi ottenuto per mezzo di reingengerizzazione manuale del medesimo frammento della base di dati.*



OO può essere rappresentato attraverso un pattern Composite. Analizzando la tabella `Incompatibilita`, si può notare come la relativa classe in realtà possegga una struttura più complessa.

Sebbene a prima analisi potrebbe venire in mente che il modello di fig. 8.19 complichi inutilmente una situazione semplice (anomalia che tecnicamente viene indicata con il nome di *over-engineering*), le cose non sono esattamente così. Il modello infatti, oltre a presentare un superiore grado di formalità, permette di evidenziare diverse regole di business. Per esempio enfatizza l'esistenza di tre tipologie di incompatibilità: quella relativa ai soli gruppi utenti, quella relativa ai soli dipartimenti e quella composta ricavata dalla combinazione delle precedenti due. Ancora, permette di evidenziare che, per quanto attiene la prima tipologia, un'istanza è valida qualora associata a due istanze della classe `GruppoUtenti`, che nelle incompatibilità relative ai dipartimenti è necessario impostare sia quello sorgente, sia quello destinazione, e via discorrendo. Volendo essere precisi bisognerebbe impostare alcuni vincoli come per esempio che le associazioni gruppo sorgente e destinazione di una medesima istanza della classe `IncompGruppiUtenti` debbano riferirsi a due diversi gruppi utente.

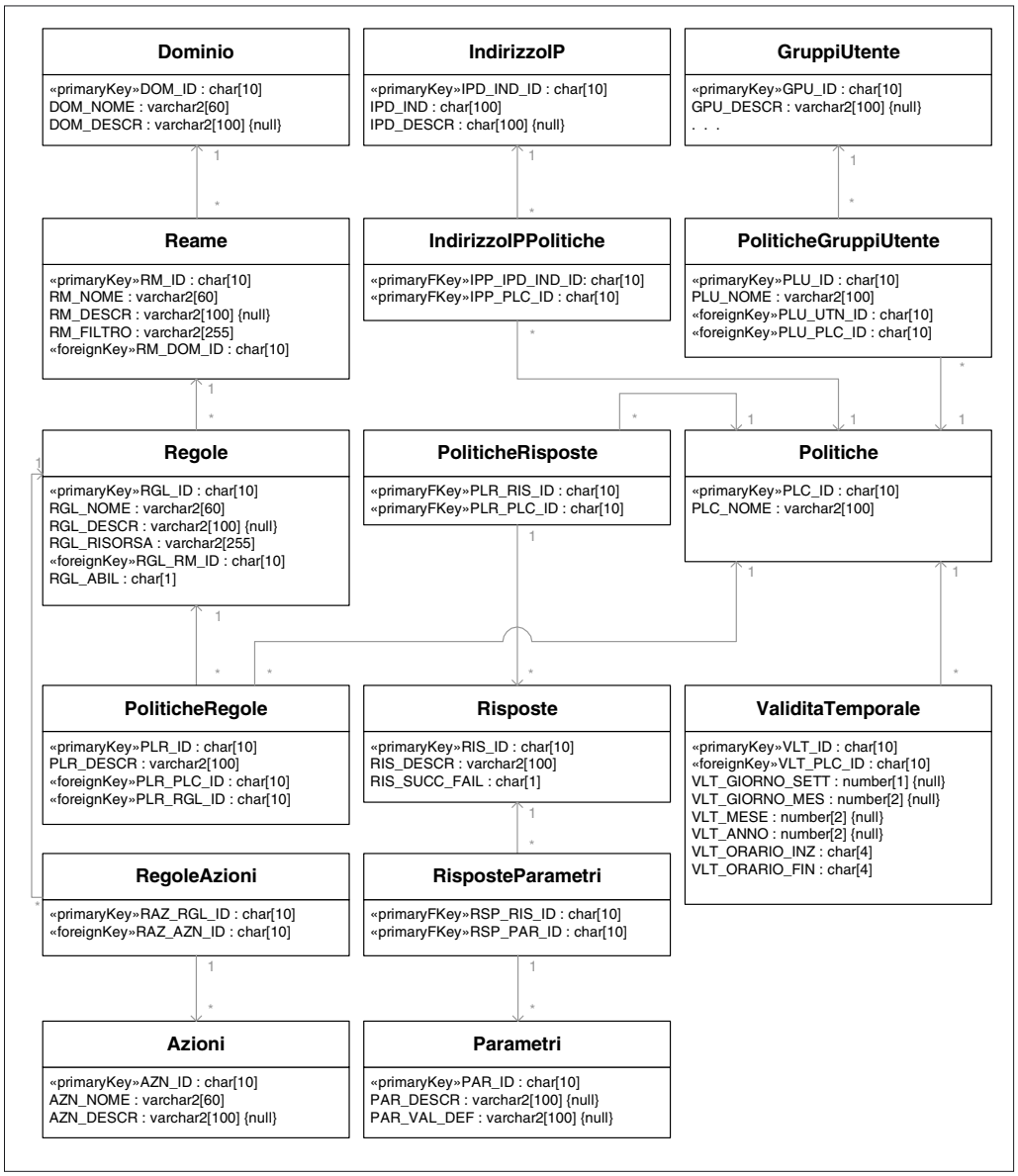
Quanto descritto evidentemente mantiene la propria validità anche per le incompatibilità relative ai dipartimenti.

Un'altra area lasciata inesplorata nella realizzazione del modello a oggetti del dominio è quella strettamente relativa all'organizzazione della sicurezza. Ora, considerando gli obiettivi della presente trattazione e quindi non dilungandosi troppo sui dettagli strettamente tecnici, le strategie di sicurezza utilizzate più frequentemente applicate sono quelle che vanno sotto il nome di ACL (*Access Control List*, Lista di controllo degli accessi) e le *Security Policy* (politiche di sicurezza). Le prime sono organizzate intorno al concetto di risorsa e quindi (molto grossolanamente parlando) definiscono le liste degli utenti abilitati ad accedervi con le relative modalità, le seconde invece pongono al centro dell'attenzione gli utenti che attraverso opportune politiche di sicurezza (come si vedrà di seguito), vengono associati alle risorse. Quest'ultima tecnica tende ad offrire un migliore controllo della sicurezza ed un maggiore grado di flessibilità e quindi è quella presa in considerazione nella presente trattazione.

Si consideri il frammento della base dati riportato nella fig. 8.20. Iniziando l'analisi del diagramma da sinistra, ed in particolare dalle tabelle `Dominio`, `Reame`, `Regole`, si può facilmente notare che si tratta di un'ennesima rappresentazione del pattern Composite. Ora, non è che l'autore sia ossessionato da questo pattern per qualche strana ragione, e tanto meno vuole dare l'impressione che si tratti del pattern "passe partout solvitutto"… La verità è che il Composite trova la sua applicazione ottimale in tutti quei contesti in cui vi sia una forte connotazione gerarchica e ricorsiva dei concetti (in poche parole alberi).

Tornando all'analisi della base di dati, le tabelle permettono di raggruppare le risorse da proteggere secondo un'organizzazione gerarchica a tre livelli: il `Dominio` è suddiviso i `Reami` i quali contengono le `Regole`. La presenza della tabella `Dominio` permette, even-

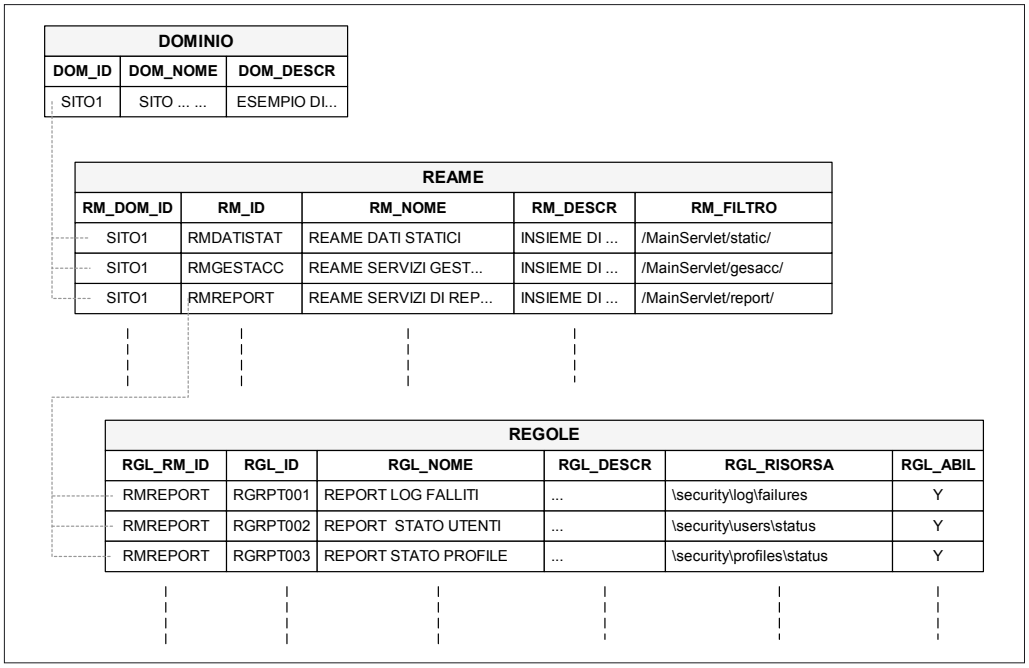
Figura 8.20 — *Frammento di database relativo alle politiche di sicurezza.*



tualmente, di organizzare diversi gruppi di regole (magari perché si vuole proteggere diversi sistemi: URL di un sito web, connessioni, …) o diverse organizzazioni delle stesse.

Un ipotetico esempio di utilizzo di tali tabelle è fornito dalla fig. 8.21.

Figura 8.21 — *Frammento di una possibile organizzazione dei dati relativi al* `Dominio`*,* `Reame` *e* `Regole` *di un sistema Internet.*



DOMINIO

DOM_ID	DOM_NOME	DOM_DESCR
SITO1	SITO	ESEMPIO DI...

REAME

RM_DOM_ID	RM_ID	RM_NOME	RM_DESCR	RM_FILTRO
SITO1	RMDATISTAT	REAME DATI STATICI	INSIEME DI ...	/MainServlet/static/
SITO1	RMGESTACC	REAME SERVIZI GEST...	INSIEME DI ...	/MainServlet/gesacc/
SITO1	RMREPORT	REAME SERVIZI DI REP...	INSIEME DI ...	/MainServlet/report/

REGOLE

RGL_RM_ID	RGL_ID	RGL_NOME	RGL_DESCR	RGL_RISORSA	RGL_ABIL
RMREPORT	RGRPT001	REPORT LOG FALLITI	...	\security\log\failures	Y
RMREPORT	RGRPT002	REPORT STATO UTENTI	...	\security\users\status	Y
RMREPORT	RGRPT003	REPORT STATO PROFILE	...	\security\profiles\status	Y

Le varie regole, organizzate gerarchicamente, sono collegate alle politiche di sicurezza per mezzo di una relazione di tipo "molti a molti".

I database relazionali non sono in grado di realizzare direttamente relazioni *(n, n)* tra tabelle, pertanto è necessario ricorrere a ulteriori tabelle con funzioni di legame che decompongano una relazione *(n, n)* in due del tipo: *(1, n)* e *(n, 1)*. Nell'esempio in questione, questo compito è affidato alla tabella `PoliticheRegole` (come anche il nome lascia presupporre). Tipicamente, tabelle appartenenti a questa tipologia non danno luogo ad altrettante classi nel modello a oggetti a meno che la relazione stessa (non le singole tabelle) non possegga propri attributi. In tal caso, in funzione delle necessità, si può optare per una classe associazione (*association class*) o per una classe vera e propria (come visto nel capitolo precedente).

Per esempio, il legame tra `Politiche` e `Regole` viene modellato attraverso una classe associazione poiché possiede degli attributi, mentre quello tra `Risposte` e `Parametri` viene rappresentato per mezzo di un'associazione semplice.

Prima di analizzare in dettaglio le politiche, si può notare come queste siano nuovamente associate per mezzo di una relazione *(n, n)* con i gruppi degli utenti. Anche in questo caso la relazione va rappresentata per mezzo di una classe associazione per via della presenza dell'attributo `PLU_NOME` nella tabella `PoliticheGruppiUtente`.

Quanto descritto poc'anzi è rappresentato in fig. 8.22.

Dal diagramma di fig. 8.22 è possibile notare l'introduzione di una coppia di classi non presenti nella base di dati: `Risorsa` e `TipologiaRisorsa`. Quest'ultima è necessaria per raggruppare le risorse in gruppi omogenei i quali, per esempio, prevedono un insieme ben definito di azioni. Per esempio se la risorsa presa in esame è un indirizzo (URL) del sito da proteggere, le azioni previste potrebbero essere `get`, `post` e `put`. Questa associazione permette di semplificare il processo di definizione delle regole: una volta selezionata la risorsa a cui si riferisce, il sistema potrebbe automaticamente proporre l'elenco delle possibili azioni selezionabili per la specifica risorsa. Per quanto concerne la classe `Risorsa`, in questo caso si è deciso di non complicare eccessivamente le cose, ma si provi ad immaginare un modo corretto per rappresentarla… Ebbene sì, anche le risorse si prestano ad essere rappresentate attraverso una struttura gerarchica, vedi pattern Composite, che

Figura 8.22 — *Diagramma delle classi relativo al* `Dominio`*,* `Politiche` *e* `GruppiUtente`*.*

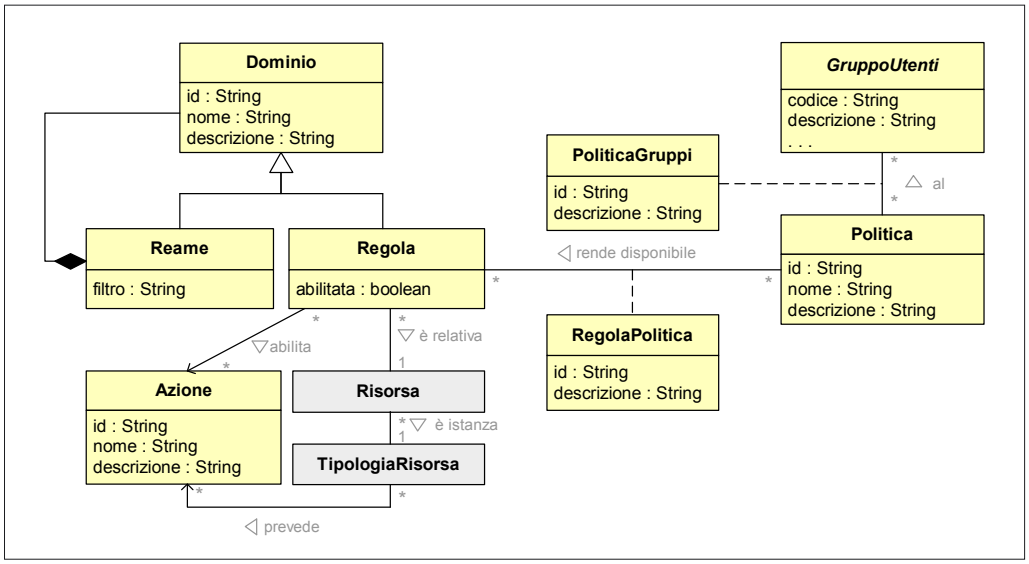
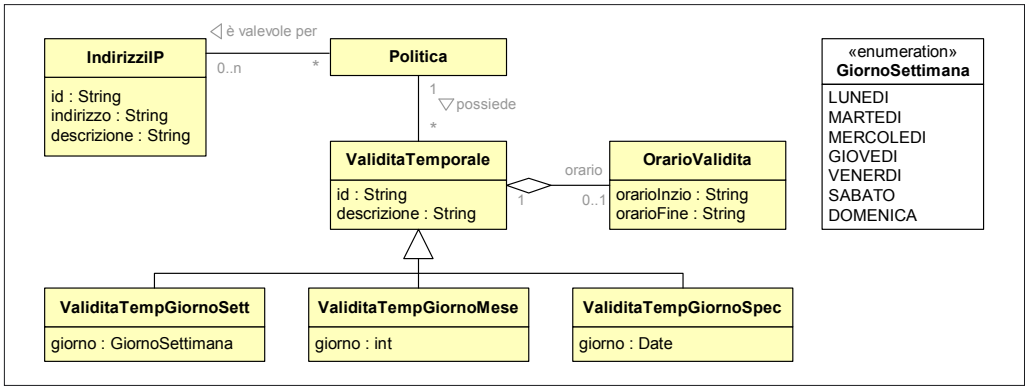


Figura 8.23 — *Primo embrione di modello relativo alle* `Politiche` *concernente la validità temporale e gli indirizzi IP.*



consentirebbe alle regole, tra l'altro, di far riferimento a singole risorse o a gruppi di esse. A questo punto nulla salverà l'autore dal venir tacciato di soffrire di un'ossessione compulsiva relativa al pattern. L'utilizzo del Composite consentirebbe un maggiore grado di flessibilità al sistema, sebbene poi l'eccessiva flessibilità spesso è una forma diversa di entropia.

Si affronti la sezione dedicata alle politiche, che, come è lecito attendersi, sono uno dei concetti fondamentali dell'infrastruttura di sicurezza presa in esame. In primo luogo è possibile associarvi una validità temporale, la quale stabilisce gli intervalli di tempo in cui la particolare politica è valida. Per default lo è sempre, a meno che non venga specificato diversamente, associando opportuni record della tabella `ValiditaTemporale`. In questi record è possibile specificare l'orario di inizio e fine validità da applicarsi tutti i giorni, o solo specifici giorni della settimana/mese/anno. Volendo è possibile restringere ulteriormente le clausole di sicurezza limitando l'insieme di indirizzi IP del computer client, per i quali la regola mantiene la propria validità.

Ancora una volta si è deciso di non eccedere nella complessità. Probabilmente però si sarebbe potuto rendere il sistema più flessibile permettendo di specificare non semplicemente insiemi di indirizzi IP, bensì intervalli, oppure tutti gli indirizzi di un host, e così via.

L'ultimo frammento rimasto da analizzare è relativo all'associazione delle politiche con le risposte. Quest'ultima tabella fornisce la struttura per organizzare i dati di un sistema per la gestione delle risposte da notificare in caso sia di fallimento, sia di successo. Per esempio, in un sistema di un sito web, in caso di fallimento si potrebbe reindirizzare la richiesta ad un'apposita pagina demandata alla gestione delle comunicazione di errori all'utente, associandovi un determinato codice di errore, oppure in caso di successo si potrebbero inserire ulteriori parametri di sincronizzazione, e così via (fig. 8.24). Questi

Figura 8.24 — *Modello relativo al legame tra le* `Politiche` *e le* `Risposte`.

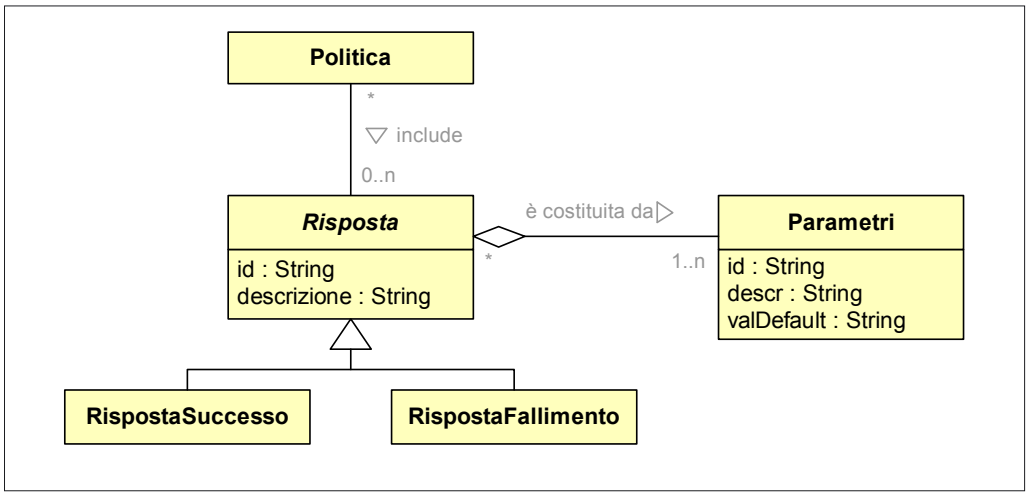
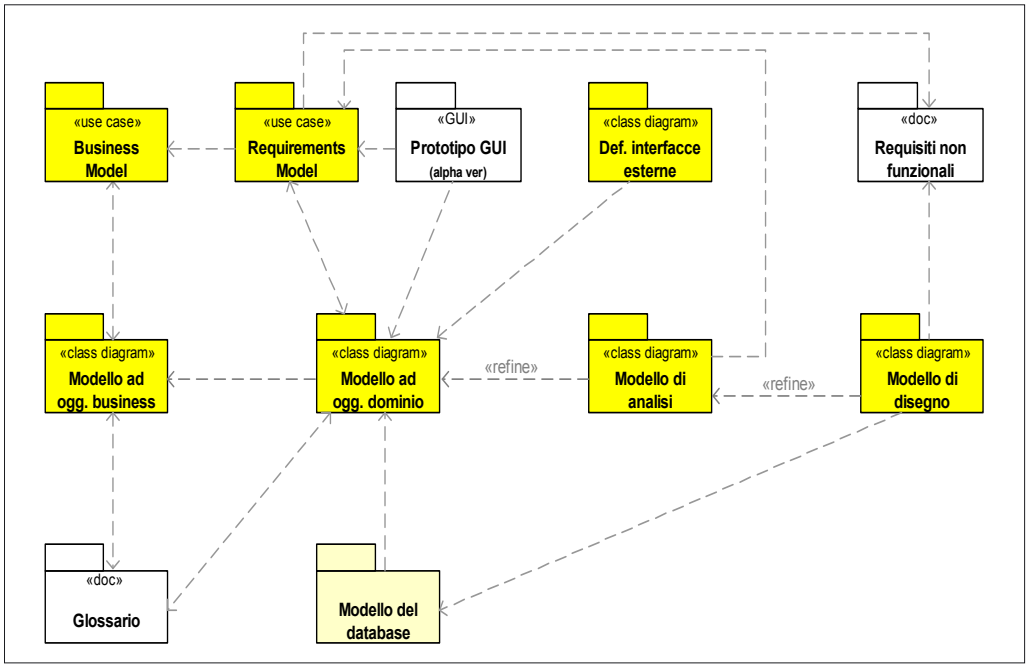


Figura 8.25 — *Modello finale delle politiche di sicurezza. Al fine di non appesantire eccessivamente il diagramma, sono stati omessi diversi dettagli.*



parametri potrebbero essere sia prefissati, sia ottenibili per mezzo dell'esecuzione di specifiche funzioni predisposte. In ogni caso, la relazione tra le risposte ed i parametri è nuovamente del tipo (*n, n*) così come pure quella tra le politiche e le risposte. Ciò permette di definire un insieme di dati basilari utilizzabili ogni volta di cui se ne abbia bisogno.

Dall'analisi del diagramma di fig. 8.25, dovrebbe essere chiaro il perché del nome della strategia modellata.

Pro e contro del metodo

Di fronte ai vantaggi ben evidenti nel ricorrere all'utilizzo dei tracciati del database per produrre modelli a oggetti del dominio — riuscire a produrre velocemente una versione iniziale del modello evitando molti lunghi meeting (non sempre proficui) con gli utenti (sebbene poi questi ne dovranno convalidare i contenuti), risolvere a priori eventuali contenziosi sulle diverse percezioni dell'area business da parte degli utenti, disporre di una versione dell'organizzazione dei dati dello spazio del problema ben investigata e collaudata — esiste una serie di insidie cui porre molta attenzione. Per esempio si può correre il rischio di spendere troppo tempo nel tentare di comprendere la struttura del database senza ottenere grossi risultati, oppure produrre una versione del modello ad oggetto a carattere eccessivamente tecnico, oppure utilizzare una struttura *eccessivamente "denormalizzata"* per questioni di performance (mai capitato di dover analizzare un database di un legacy system datato?) e così via.

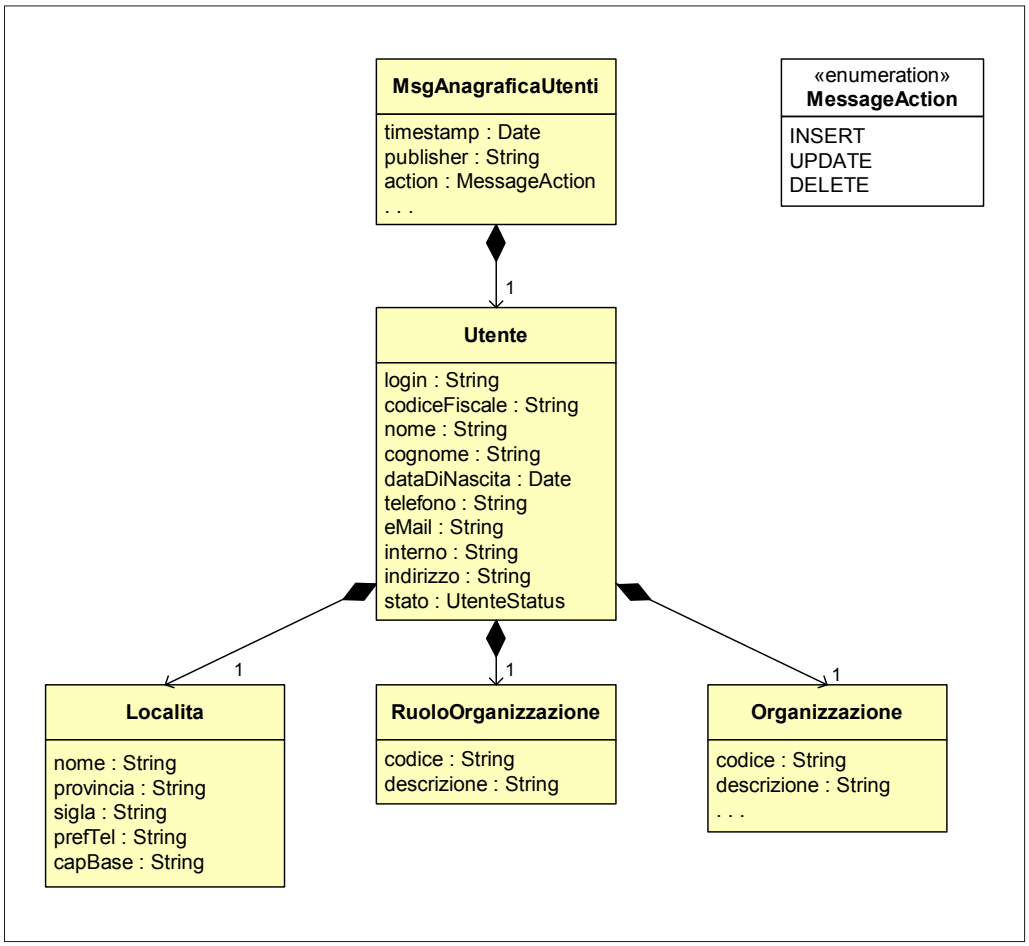
Molti dei rischi citati possono essere attenuati considerando il modello prodotto, ancora una volta, come una versione iniziale da utilizzarsi come piattaforma di colloquio e non come versione finale del modello. Molto importante risulta anche usufruire della continua consulenza di un esperto dell'area business oggetto di studio. Particolarmente importante è ricorrere a risorse interne dell'organizzazione committente sia per sposarne (o per darne sensazione) la relativa visione del business, sia per coinvolgerli durante il processo e quindi ottenere il relativo sostegno.

In ultima analisi si tratta di una scorciatoia che, se utilizzata correttamente, offre molti vantaggi; come ormai da prassi è necessario applicarla mantenendo il cervello costantemente sulla posizione on.

Il modello del dominio e i messaggi

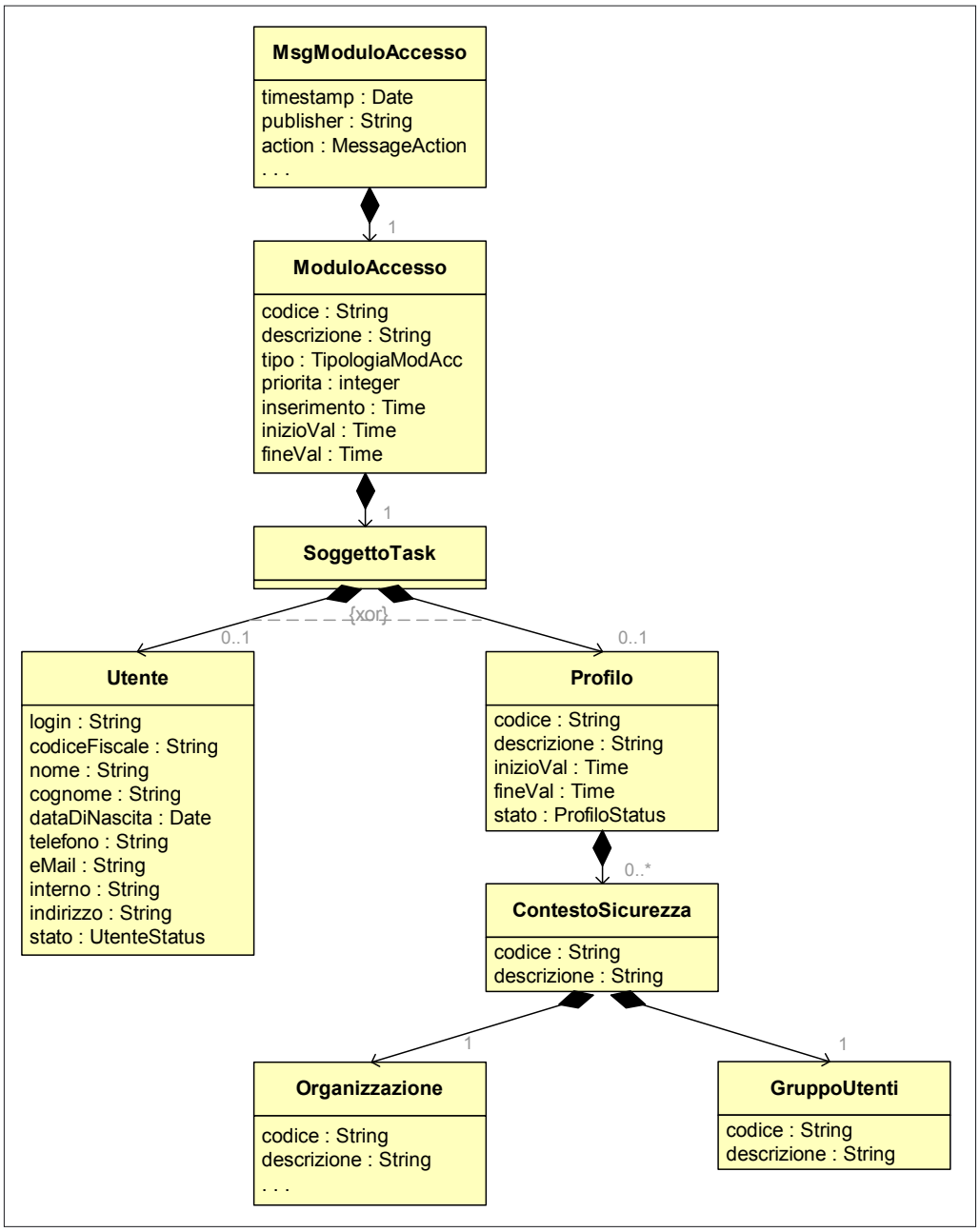
Al momento in cui viene redatto il presente testo i sistemi di messaggistica sembrano assumere sempre di più un ruolo di primo piano nella produzione di sistemi complessi. Si tratta di un'infrastruttura che permette di disaccoppiare le componenti in cui è strutturato un sistema complesso, mantenendole sincronizzate attraverso lo scambio asincrono di messaggi. Questi messaggi rappresentano un po' l'evoluzione del concetto di interfaccia: un componente per interagire con un altro deve produrre opportuni messaggi in accordo

Figura 8.26 — *Esempio di messaggio estratto dal modello a oggetti del dominio.*



allo schema previsto (ancora una versione del concetto di contratto tra cliente e fornitore di servizio). Prima di dar luogo a meccanismi di sincronizzazione delle entità presenti in diversi sistemi è sempre opportuno verificare se ciò sia veramente necessario o se esitano metodi che ne eliminino la necessità, non ultimo il ripensamento della stessa architettura del sistema. In ogni modo, data l'importanza della definizione dei messaggi è opportuno disegnarne versioni iniziali prima possibile. La produzione del modello a oggetti semplifica anche la produzione delle versioni concettuali di tali messaggi: dovrebbe trattarsi dell'estrazione di opportuni gruppi di entità del modello. Se così non fosse, probabilmente il modello del dominio potrebbe non essere strato progettato adeguatamente.

Figura 8.27 — *Messaggio utilizzato per innescare il workflow di un nuovo modulo di accesso.*



Si consideri per esempio il caso in cui nel sistema preso in esame la gestione dei dati relativa ai dipendenti sia responsabilità di uno specifico sistema, e che all'infrastruttura di sicurezza sia vietato l'accesso al relativo database. In tale scenario, il sistema di sicurezza necessiterebbe di possedere un'opportuna replica, mantenuta costantemente sincronizzata, della tabella degli utenti (questi per poter svolgere le proprie mansioni devo possedere opportune credenziali per accedere al sistema). Tra le varie alternative percorribili, una consiste nella pubblicazione di opportuni messaggi destinati all'infrastruttura di sicurezza (*subscriber*, sottoscrittore) da parte del sistema anagrafico (*publisher*, pubblicatore). Obiettivo di tali messaggi è informare sulle variazioni di opportuni dati relativi ai dipendenti. Ora disponendo del modello a oggetti del dominio, una prima versione del messaggio potrebbe essere facilmente prodotto prelevando le entità di interesse.

Come si può notare, data la rappresentazione UML di un messaggio, la produzione del relativo schema dovrebbe essere cosa presto fatta.

Un altro esempio potrebbe aversi qualora la gestione del workflow dei moduli di accesso fosse gestito da un apposito sistema (WMS, *Workflow Management System*, sistema per la gestione dei flussi di lavoro). In questo scenario, l'infrastruttura di sicurezza svolgerebbe il ruolo del pubblicatore mentre il WMS reciterebbe il ruolo di sottoscrittore.

Modello a oggetti business

Introduzione

"Un modello del business mostra l'ambiente dell'azienda e come questa opera in relazione ad esso. Con il termine ambiente si intende ogni cosa con cui l'azienda interagisce al fine di eseguire i propri processi di business, come i clienti, i partner, le aziende appaltatrici e così via. Il modello mostra i dipendenti e, ad ogni livello, cosa deve essere fatto e quando dovrebbe essere fatto" [Ivar Jacobson].

> Quando si parla del modello del business è necessario porre attenzione. Generalmente con lo stesso termine si identificano due manufatti tra loro molto diversi (sebbene concorrano a definire lo stesso contesto da diversi prospettive) ossia: business use case e business object model. In questo ambito si fa chiaramente riferimento al modello a oggetti. Per eliminare ogni possibile ambiguità alla sorgente ci si riferisce al modello con il nome modello a oggetti business.

Qualora in un processo di sviluppo del software si decida di realizzare il modello a oggetti del business, questo dovrebbe essere prodotto durante le primissime fasi del processo stesso, tipicamente prima del modello a oggetti del dominio, sebbene nessuno vieti il contrario. In questo capitolo i due manufatti sono stati presentati in ordine inverso per agevolarne l'esposizione. In effetti, come si vedrà, il modello a oggetti del business può essere visto con un

modello a oggetti del dominio esteso, nel senso che tutte le classi entità del modello del dominio sono presenti nel modello del business (entity); e in più compaiono altri concetti come worker (lavoratori) e work unit (unità di lavoro). Il Business Object Model descrive come un gruppo di utilizzatori (tecnicamente denominati *worker* della business area, ossia gli attori dell'area) utilizzano un insieme di entità di business (le classi già evidenziate nel modello a oggetti del dominio) attraverso opportune unità di lavoro (*work unit*). In sostanza una work unit definisce la logica che specifica i vari servizi erogabili agli attori.

In questo contesto l'attenzione viene focalizzata sull'intera area business e non esclusivamente sul sottoinsieme da implementare. Il concetto è equivalente alla differenza tra sistema informativo e quello informatico, in cui quest'ultimo è un sottosistema del primo. Il sistema informativo considera tutti i flussi, anche quelli destinati a restare manuali, e pertanto non di interesse per il sistema informatico. Nel contesto del sistema preso come riferimento, per esempio, flussi informativi tipicamente manuali potrebbero essere: relativamente alla validazione e/o approvazione del modulo di accesso relativo al profilo iniziale di un nuovo utente, la verifica da parte, rispettivamente, degli addetti alla sicurezza e del supervisore che il nuovo dipendente effettivamente esista e che non sia un tentativo di frode, che i privilegi richiesti siano effettivamente i minimi indispensabili per consentirgli di espletare le relative mansioni, e così via. Quindi le funzioni che si considerano sono i servizi che l'area business fornisce ai propri attori, e pertanto, non tutte le funzioni e gli attori evidenziati in questa fase verranno inclusi nel sistema da automatizzare.

In prima analisi è possibile affermare che un business object model è un modello a oggetti del dominio arricchito attraverso l'inserimento di concetti come *worker* e *work unit* (secondo quest'ottica il modello a oggetti del business dipende da quello a oggetti).

Poiché in questa versione del modello a oggetti compaiono elementi come gli attori e classi strettamente correlate con i business use case (le *work unit* realizzano le funzione definite nei casi d'uso di business), è evidente il legame di mutua dipendenza tra il business object model ed la relativa versione dei casi d'uso. In questi contesto probabilmente il modello Business use case è propedeutico: i casi d'uso che lo costituiscono giustificano tutti gli altri elementi, entità, attori e work unit stesse. Le nuove classi sono aggiunte essenzialmente per fornire le funzionalità descritte nei casi d'uso, pertanto in questa fase è fondamentale specificare anche le operazioni (servizi) che, ovviamente, devono essere di stampo prettamente business. In ultima analisi i servizi sono attuati attraverso specifiche collaborazioni tra opportuni oggetti, realizzate attraverso l'invocazione dei relativi metodi.

Il metodo classico utilizzato per realizzare un modello a oggetti del business prevede:

1. analisi del modello dei casi d'uso di business. Da questo è possibile pre-

levare l'elenco degli attori (si tratta di un primo elenco; non tutti finiranno nel modello dei requisiti, giacché, tipicamente, non tutti interagiscono con il sottosistema informatico), e le work unit. Qualora i casi d'uso posseggano un livello di granularità idoneo, si genera quasi una corrispondenza uno ad uno tra singolo caso d'uso e unità di lavoro;

2. estrazione delle classi entità (per questo motivo spesso si realizza un modello a oggetti del dominio "allargato") da incorporare come entity;
3. riorganizzazione del tutto: gli attori interagiscono con le varie unità di lavoro, le quali forniscono i propri servizi accedendo, ed eventualmente modificando, lo stato delle varie entità. Durante questo processo spesso si rende necessario aggiungere altre classi destinate a realizzare altri fattori, come per esempio particolari regole del business.

Il concetto di work unit si presta ad essere rappresentato mediante il package piuttosto che mediante singole classi. Il motivo principale è che le work unit incapsulano responsabilità e funzionalità con un livello di coesione non sempre elevatissimo (si tratta pur sempre di un modello business), che, verosimilmente, si presta ad essere raffinato suddividendo le responsabilità tra un opportuno insieme di classi correlate.

Si tenga presente che questo modello va prodotto facendo larga economia del tempo, pertanto attendersi un modello propriamente object oriented è decisamente pretestuoso (anche perché la produzione, solitamente, è affidata a Business Analyst, personale non sempre espertissimo degli insegnamenti dell'OO).

Esempio del modello a oggetti del business

Prima di mostrare un esempio del modello a oggetti del business, si vuole ricordare che nella versione 1.4 dello UML è stata integrata la definizione del profilo per la modellazione dell'area business (*UML profile for business modeling*), come anche riportato nel secondo capitolo del libro.

> Per quanto concerne la realizzazione del modello a oggetti del business esistono diverse correnti di pensiero anche per via della mancanza sia di un formalismo universalmente accettato sia di uno standard emergente (probabilmente ciò è un segno del limitato interesse per l'argomento). Studiando la (limitata) letteratura relativa ai modelli a oggetti del business è possibile imbattersi in diagrammi tra i più svariati: ci sono alcuni che realizzano tali modelli come varianti di quelli di analisi (presentati più avanti nel capitolo), altri che sembrerebbero una via di mezzo tra diagrammi delle classi e di collaborazione e così via.
>
> L'autore, anche per questioni di carattere storico, ha sempre adottato una versione simile a quella

presentata dai Tres Amigos nel libro *The Unified Software Development Process* (anche se poi vi è un solo esempio), e pertanto gli esempi riportati riflettono questa scelta.

Tra i vari elementi quelli di maggiore interesse sono riportati nella tab. 8.3.

Poiché non tutti gli attori e le entità dell'area business confluiranno poi nel sistema, e quindi nei restanti modelli, spesso si preferisce ricorrere ad una notazione grafica leggermente diversa da quella tipica, al fine di evidenziare la connotazione "business" delle succitate entità. Alcuni tool, per esempio evidenziano questa versione dei vari concetti attraverso una linea obliqua, come mostrato nella fig. 8.28.

Tabella 8.3 — *Elementi fondamentali del profilo UML per la realizzazione di modelli a oggetti dell'area business.*

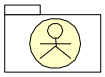
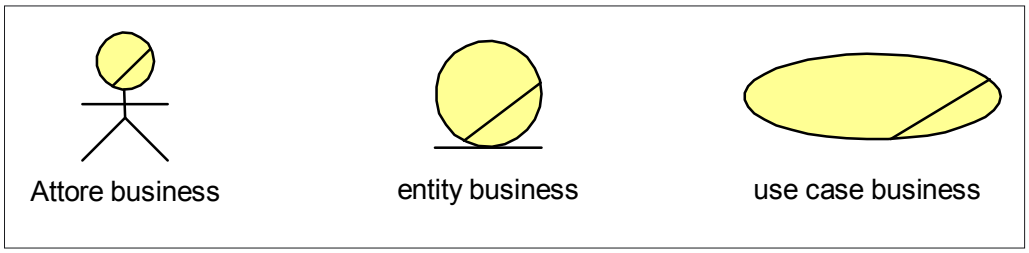
Elemento	Metaclasse del metamodello	Breve descrizione
WorkUnit 	System	Si tratta di un sottosistema che può contenere una o più entità. É un insieme di oggetti raggruppati in funzione della particolare attività da svolgere che rappresenta un tutt'uno per l'utente finale. Eventualmente può disporre di un *façade* in grado di definire le entità della *WorkUnit* rilevanti per l'attore.
Worker	Class	È una classe che rappresenta un'astrazione di un attore umano che interagisce con il sistema. Un *Worker* interagisce con altri attori e manipola *Entità* mentre partecipa alla realizzazione di un particolare caso d'uso.
CaseWorker	Class	Un *CaseWorker* è uno caso speciale di *Worker*, caratterizzato dall'interagire direttamente con gli attori esterni al sistema.
InternalWorker	Class	Si tratta di un altro caso speciale di attore, questa volta caratterizzato dall'interagire con altri *Worker* e *Entità* interne al sistema.
Entity	Class	Un *Entity (Entità)* è una classe passiva che può partecipare a diverse realizzazioni di casi d'uso, e tipicamente, sopravvive alle singole interazioni.
Communicate	Association	Si tratta di un'associazione utilizzata per sancire l'interazione delle istanze dei *Classificatori*, associati.
Subscribe «subscribe»	Association	Un *subscribe* è un'associazione tra due classi che stabilisce che oggetti della classe sorgente (*subscriber*, sottoscrittore) riceveranno un'apposita notifica al verificarsi di particolari eventi negli oggetti della classi destinazione (*publisher*, pubblicatore). In questa associazione viene specificato l'elenco degli eventi che determinano l'invio della notifica ai sottoscrittori.

Figura 8.28 — *Alcuni esempi degli stereotipi utilizzati nell'area business.*



Sebbene non siano standard UML, gli stereotipi mostrati in fig. 8.28 nella pratica sono abbastanza utilizzati. In particolare si fanno apprezzare in quanto permettono di intuire immediatamente il livello di astrazione business dei vari elementi.

Come esempio di modello business, si consideri la sottomissione di una modulo di accesso.

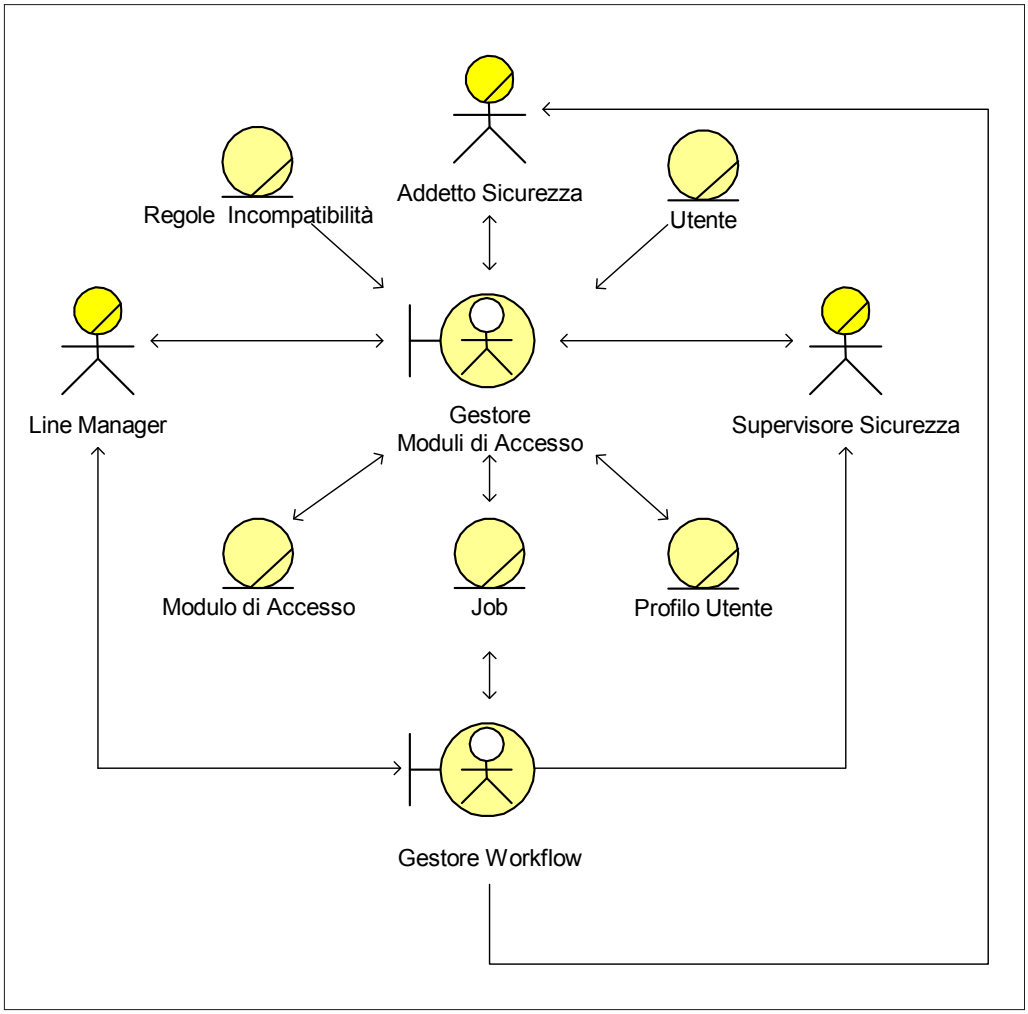
Il diagramma di fig. 8.29 vuole modellare il ciclo di un modulo di accesso, dal punto di vista business:

1. il `Line Manager` inserisce una richiesta, compilando uno specifico modulo di accesso, da sottoporre ad un apposito addetto alla sicurezza. Tale richiesta può essere relativa ad un nuovo profilo, alla modifica di uno esistente, al blocco e sblocco di un utente, ecc.;

2. il `(case worker) Gestore Moduli di Accesso` si occupa di registrare le relative informazioni, eventualmente verificando il rispetto delle regole di incompatibilità;

3. il `Gestore Moduli di Accesso` registra il nuovo `Modulo di Accesso` e contestualmente genera un apposito `Job`;
3. il `Gestore workflow` prende in consegna il nuovo `Job`.
   3.1. se si tratta di un `Modulo di Accesso` sottoposto da un `Line Manager` lo sottopone all'attenzione di un apposito `Addetto alla Sicurezza`;
   3.2. se si tratta di un `Modulo di Accesso` validato da un `Addetto alla Sicurezza` lo sottopone all'attenzione di un apposito `Supervisore alla Sicurezza`;
   3.3. se si tratta di un `Modulo di Accesso` rigettato lo sottopone all'attenzione del `Line Manager` che lo aveva sottoposto;

Figura 8.29 — *Esempio di modello di business relativo alla gestione del ciclo di vita di un modulo di accesso.*



4. l'`Addetto alla Sicurezza` può validare o rigettare un `Modulo di Accesso` posto alla sua attenzione;

5. in entrambi i casi il `Gestore Moduli di Accesso` genera un nuovo `Job` (nel primo caso relativo ad un'attività di responsabilità del `Supervisore della Sicurezza`, mentre nel secondo per avvertire il `Line Manager`);

6. il `Supervisore alla Sicurezza` può approvare o rigettare un `Modulo di Accesso` posto alla sua attenzione;

7. in entrambi i casi, rifiuto o approvazione, il `Gestore Moduli di Accesso` genera un apposito `Job`;

8. il `Line Manager` può cancellare o modificare un `Modulo di Accesso` sottomesso e rigettato;

9. nel caso di modifica il `Gestore Moduli di Accesso` genera un apposito `Job`, altrimenti il viene aggiornato esclusivamente il modulo di accesso per rifletterne il nuovo stato (cancellato).

Modello a oggetti del dominio "contro" quello di business

La stretta somiglianza esistente tra il modello a oggetti business e del dominio finisce per porli su un piano di alternativa. Il problema è che, se si tentasse di produrre tutti i manufatti previsti dai processi più formali per lo sviluppo del software, verosimilmente sarebbe necessario disporre di un lasso di tempo di qualche ordine di grandezza superiore rispetto a quello effettivamente necessario/disponibile. Considerata quindi la canonica carenza di tempo a disposizione, si rende necessario sorvolare su qualche manufatto, pur importante, ma non di importanza vitale per il buon proseguimento del progetto. Da notare che ogni qualvolta si rinuncia alla produzione di un manufatto, implicitamente, si amplifica il fattore di rischio del progetto. Pertanto, per ogni taglio operato, andrebbe considerata un'opportuna contromisura atta a mitigare l'aumento del fattore rischio. Chiaramente la rinuncia dei manufatti principali (per esempio il modello di disegno) è difficilmente compensabile ed il rischio che si corre è di dar luogo ad un magnifico esempio di progetto-kamikaze.

Nella lotta alla sopravvivenza, tipica dei progetti reali, il modello a oggetti del dominio tende ad avere la meglio sul relativo al livello di business… In effetti, lavorando in svariate aziende/progetti, molto raramente accade di realizzare/visionare modelli a oggetti dell'area business, mentre quelli del dominio sono meno infrequenti. La preferenza del modello del dominio è legittimata da alcune interessanti considerazioni:

- è molto utile disporre di una versione dei dati del dominio del problema perché agevola la progettazione della base di dati, a oggetti o di diverso tipo;

- semplifica la scomposizione del sistema in componenti (si individuano gruppi di dati fortemente interconessi e quindi, verosimilmente, destinati ad essere raggruppati in uno stesso componente);

- semplifica la definizione delle interfacce sia utente, sia degli eventuali messaggi utilizzati per far comunicare i vari sottosistemi, ecc.;

- il modello del business può essere (in gran parte) compensato dal modello a oggetti del dominio più la relativa vista dei casi d'uso;

- la fase successiva prevede il modello di analisi molto simile a quello business (con la grande differenza che il primo è focalizzato sul sistema da sviluppare e non esclusivamente sull'intera area business), ecc.

Molti autori preferiscono invece il modello a oggetti business poiché focalizzando l'attenzione sull'intero business si ha un manufatto di valore molto elevato, anche per finalità non strettamente connesse con la costruzione del sistema software, come la materializzazione del quadro generale da utilizzarsi come punto di partenza per generare diversi progetti, addestramento del personale, ottimizzazioni dei flussi lavorativi, ecc.

L'autore, utilizzando un approccio assolutamente pragmatico e forte anche della propria esperienza, crede che il fattore tempo giochi un ruolo fondamentale in pressoché tutti i progetti. Quindi, poiché il modello business, tipicamente, richiede molto più tempo e non è strettamente funzionale alla produzione del sistema, probabilmente può essere trascurato a vantaggio di quello del dominio. Forse si tratta di un bene di lusso che solo poche organizzazioni possono permettersi. Qualora si disponesse delle risorse necessarie (soprattutto tempo e quindi budget), sarebbe molto interessante produrre entrambi i modelli.

Quale viene prima?

Nel paragrafo precedente si è visto come il modello a oggetti del business e quello del dominio, nella lotta alla sopravvivenza imposta dalla risorsa tempo, tendano a diventare antagonisti, in una confronto che vede il modello a oggetti del dominio favorito. Qualora però sia abbia la fortuna di poter realizzarli entrambi potrebbe nascere il dilemma su quale modello realizzare per primo. Non c'è una risposta univoca ed entrambe le alternative presentano dei vantaggi. Probabilmente il metodo migliore consiste nel procedere in parallelo, facendo attenzione però, che limitatamente a questo contesto, allocare la realizzazione dei due modelli a team diversi potrebbe non essere un'idea sempre vincente. Si correrebbe il rischio di dover reiterare le interviste con gli utenti/clienti, di produrre diverse versioni dell'area oggetto di studio, di spendere molto tempo per i riscontri, e così via.

In ogni modo, l'approccio più tipico prevede di iniziare con la realizzazione del modello a oggetti di business. Ciò per una serie di motivi plausibili. In primo luogo permette di acquisire una visione completa dell'intero sistema in cui l'azienda l'opera e quindi di individuare la giusta collocazione del nuovo progetto. Dato il modello a oggetti del business, è

abbastanza agevole ricavare il sottoinsieme di interesse per quello del dominio, e così via. A fronte di questi vantaggi, esistono una serie di controindicazioni. Per esempio è più complicato affrontare da subito lo studio dell'intero sistema; molte volte è più agevole iniziare a concentrarsi su un aspetto più limitato (il modello a oggetti del dominio) e quindi procedere estendendo lo stesso. Durante la produzione del modello di business, difficilmente il processo di sviluppo del software può procedere: non si tratta di un manufatto in grado di innescare direttamente altre fasi, mentre la realizzazione del modello del dominio permette di iniziare lo studio della base di dati, il disegno dei messaggi, le interfacce utenti e così via.

Oltre a questi criteri oggettivi, la scelta su quale manufatto produrre dipende ovviamente anche da altri fattori, per così dire logistici, come per esempio le risorse umane disponibili, i relativi ruoli, ecc. In ogni modo, l'importante è che alla fine i modelli posseggano la necessaria qualità e che riproducano, con prospettive diverse, lo stesso sistema in modo coerente.

Modello a oggetti di analisi

Introduzione

L'Analysis Object Model (modello di analisi a oggetti o più semplicemente modello di analisi), come suggerisce il nome, è prodotto durante la relativa fase del processo di sviluppo del software (fase spesso incorporata in quella di disegno). Si tratta di uno stadio per così dire di "raccordo" tra l'analisi dei requisiti utente ed il disegno del sistema; una sorta di l'anello di congiunzione tra il modello a oggetti del dominio (o del business) ed il modello di disegno del sistema, di cui viene considerato un'iniziale astrazione, sebbene esistano importanti differenze. Una prima divergenza tra questi modelli è relativa alle classi presenti: nel modello di analisi sono ancora focalizzate sui requisiti funzionali del sistema, mentre l'incorporazione di quelli non funzionali è rimandata al modello di disegno. Trattandosi di uno stadio intermedio finisce per presentare un giusto compromesso tra la fase di analisi dei requisiti, i cui principali attori sono utenti e business analyst (personale con estesa esperienza del dominio del sistema e minore conoscenza tecnica), e la fase di disegno, territorio del personale che dovrà costruire il sistema (architetti, programmatori, ecc.). Altre differenze sono relative al livello di astrazione: nella fase di analisi il livello di formalismo è sicuramente superiore rispetto a quello delle fasi precedenti, ma non ancora uniformato a quello delle fasi successive. Gli esperti del dominio tendono ancora ad essere coinvolti sebbene con ruoli piuttosto marginali. Tipicamente ne è richiesto l'intervento per chiarire/analizzate situazioni ritenute ambigue, per colmare eventuali lacune (come per esempio alternative lasciate inesplorate), per "ritoccare" requisiti per così dire futuristici, e così via. Ovviamente non gli si richiede di valicare il modello. L'intervento di questi personaggi si rende necessario poiché la produzione del modello di analisi richiede l'approfondimento

sia dei vari casi d'uso sia del modello a oggetti del dominio. Pertanto, uno degli effetti positivi della produzione del modello di analisi è la revisione formale ed approfondita dei vari modelli prodotti in precedenza. Non è infrequente che alcuni architetti realizzino delle versioni embrionali (usa e getta) del modello di analisi al fine di raggiungere il duplice obiettivo di acquisire una più intima conoscenza del modello dei requisiti e di realizzare una primissima versione del modello di disegno.

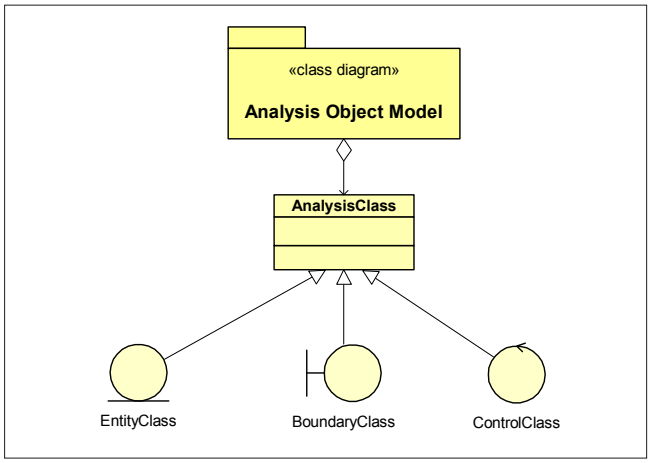
Le classi del modello di analisi, per forza di cose, risultano più vicine allo spazio del dominio, più concettuali e verosimilmente meno granulari delle corrispondenti a livello di disegno. In questa fase non si può ancora pretendere un modello dettagliato e tanto meno un elevato grado di formalità. A meno di particolari eccezioni, l'applicazione degli insegnamenti propri dell'OO sono parzialmente sacrificati a favore del risparmio di tempo. Inoltre, come nel caso del modello a oggetti del dominio, ancora non si è molto interessati a definire le operazioni delle varie classi, ciò sia perché probabilmente non si dispone ancora di sufficienti informazioni, sia perché il tempo richiesto per produrre un tale livello di dettaglio non sarebbe giustificabile nel contesto dell'intero processo di sviluppo del software.

A livello di analisi, tipicamente, non si è molto interessati a concetti come interfacce, sebbene l'attribuzione delle responsabilità rimanga comunque assolutamente necessaria. A tal fine, alcune volte si impiegano strumenti non completamente formali, come per esempio del testo da associare alle classi oppure formalismi come il CRC (*Class – Responsibility – Collaboration*, classe responsabilità e collaborazioni). In questo contesto con la definizione di responsabilità si riferisce ad un sottoinsieme di comportamento ad elevato grado di coesione definito dalle singole classi. In parole più semplici ci si riferisce alle operazioni esposte dalla classe che implementano, o concorrono a implementare, i servizi da erogare all'utente del sistema.

Nel modello di analisi, si è interessati a rappresentare classi appartenenti ad una delle categorie illustrate nella fig. 8.30. Si tratta di stereotipi "standard" dell'elemento "Classe" del metamodello, i quali non introducono alcuna caratteristica supplementare se non un forte significato semantico.

La descrizione degli stereotipi mostrati in fig. 8.30 è riportata nel Capitolo 2, dedicato alla panoramica sullo UML. In particolare, con la versione 1.4 del linguaggio, sono stati inseriti nel profilo standard denominato *Software Development Processes* (processi di sviluppo software).

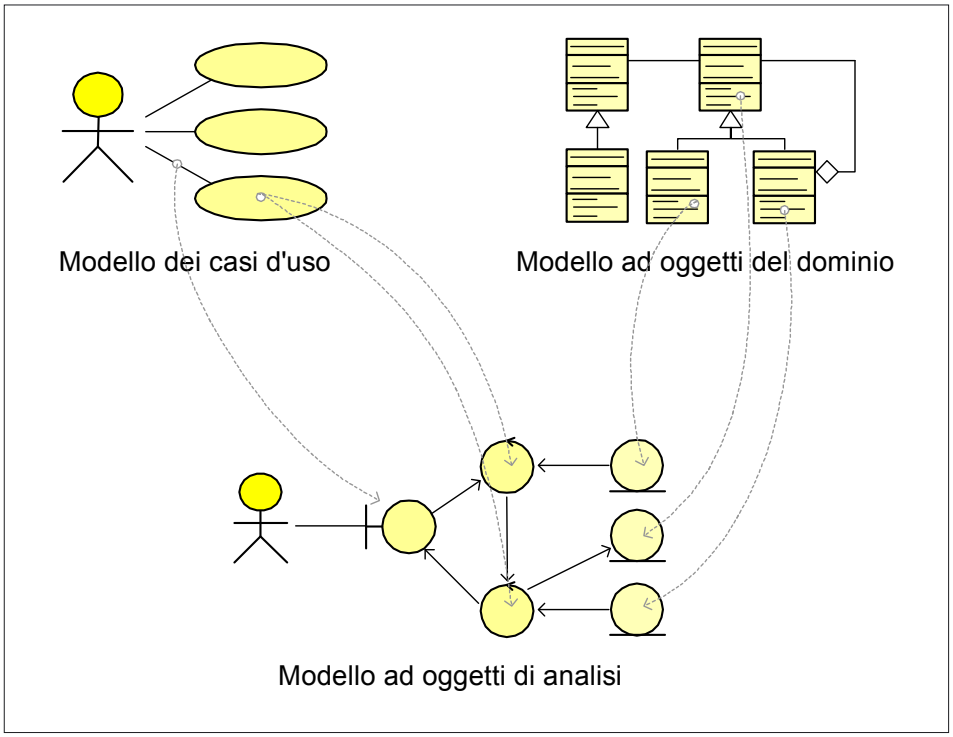
Figura 8.30 — *Stereotipi del modello di analisi*



Una **boundary class** (classe di confine) rappresenta un "punto di accesso" al sistema, una componente dello strato di interfacciamento con i relativi attori. Da tenere presente che l'interazione attore/sistema non si realizza esclusivamente attraverso apposite GUI, ma può avvenire per mezzo di altri meccanismi, come per esempio la ricezione di messaggi (in tal caso l'attore è un altro sistema/dispositivo), attraverso nuovi dispositivi di input, e così via. Le *boundary class* sono parti del sistema fortemente dipendenti dagli attori che le utilizzano. Queste permettono di investigare, chiarire e formalizzare i requisiti del sistema al livello di interfacciamento con entità esterne. L'interazione attore/sistema può richiedere una serie di azioni, quali il controllo dei dati (sia presenti in opportuni messaggi inviati al sistema, sia provenienti dall'immissione dei dati in appositi campi delle GUI), l'invocazione dei metodi previsti dalle classi di gestione dei servizi richiesti (in questo contesto denominate control), la presentazione dei risultati all'utente (sia per mezzo di apposite schermate, sia attraverso l'invio di opportuni messaggi), ecc.

Considerata la fase di questo modello, ed il livello di astrazione ancora abbastanza elevato, verosimilmente non è opportuno dettagliare eccessivamente le classi boundary (per esempio se si tratta di una window non è necessario descriverne tutti i widget), è sufficiente specificare gli obiettivi dell'interazione, i dati che si intendono scambiare, ecc. senza però descrivere come questa poi venga realizzata. La gestione di dettagli di questo tipo è demandata alla fase di disegno.

Figura 8.31 — *Costruzione del modello di analisi. Nel diagramma si è preso in considerazione un solo caso d'uso (quello in basso) e ipotizzando che esso utilizzi esclusivamente tre classi del modello a oggetti del dominio (quelle alla in basso a destra nel modello a oggetti del dominio).*



Per quanto concerne le **entity class** (classi entità), il relativo significato è esattamente quello atteso: ossia classi atte a modellare informazioni gestite dal sistema che necessitano di essere memorizzate in forma permanente. Tipicamente esiste una corrispondenza biunivoca con le classe individuate nel modello del dominio, con la differenza che a questo stadio la struttura deve essere quanto più vicina possibile agli sviluppatori. Un ulteriore livello di dettaglio si ha nella versione del disegno, ove spesso si rende necessario prevedere dettagli implementativi, quali per esempio quelli legati al sistema per il supporto della persistenza. Per loro definizione le classi entity sono passive, sebbene possano disporre di comportamenti anche complessi legati alle informazioni contenute (per esempio negli *Enterprise Entity Java Beans* i dati persistenti vengono definiti attraverso opportuni metodi `get/set`). Uno degli scopi delle classi entity consiste nell'isolare i dettagli della memorizzazione delle informazioni dalle classi che le manipolano (*control*). Ciò offre il

vantaggio di creare uno strato di indirezione e quindi limitare l'impatto di eventuali aggiornamenti apportati al livello dell'organizzazione delle informazioni o alla relativa gestione. In ultima analisi, le classi entity formalizzano la struttura logica dei dati e quindi concorrono a chiarire la comprensione delle informazioni su cui il sistema si fonda.

Le **control class** (classi di controllo) come suggerisce il nome, rappresentano il controllo, nel senso più generale del temine (coordinamento, transizione, successione, ecc.), di altri "oggetti". Tipicamente sono utilizzate per incapsulare la logica di controllo presente nelle funzionalità definite al livello dei casi d'uso. Spesso rappresentano algoritmi di calcolo anche complessi, attuatori delle policy aziendali e regole di business… Chiaramente esula dalla responsabilità di tale classe definire e controllare l'interazione con gli attori del sistema, o strutturare le informazioni che necessitano di essere memorizzate permanentemente nel sistema (responsabilità demandate, rispettivamente alle boundary e alle entity class). Le classi di controllo definiscono formalmente le dinamiche interne del sistema. Ciò è una logica conseguenza del fatto che gestiscono i flussi principali del sistema, coordinando il lavoro svolto da altri oggetti istanze delle boundary e delle entity class. Una delle caratteristiche peculiari delle classi di controllo consiste nell'incapsulare le logiche del sistema, e quindi nel ridurre l'impatto sulle restanti parti dovuto ad eventuali variazioni. Quindi, se delle regole del business dovessero cambiare, gli eventuali aggiornamenti dovrebbero ripercuotersi sulle relative classi di controllo senza ripercuotersi sulle restanti parti del sistema. Tipicamente le classi di controllo tendono ad assumere la fisionomia di "macro-classi" e quindi raramente sopravvivono al modello di disegno, nel quale, ciascuna classe di controllo, tende a dar luogo a svariate classi.

La costruzione del modello a oggetti di analisi dovrebbe avvenire dopo aver realizzato i manufatti delle fasi precedenti come il modello dei casi d'uso, quello a oggetti del dominio, ecc. L'analisi di questi manufatti permette di semplificare notevolmente il disegno del modello di analisi (si consideri la fig. 8.31). In particolare:

- le classi del modello a oggetti del dominio danno luogo alle entity class del modello di analisi. Generalmente vi è una corrispondenza quasi biunivoca, sebbene spesso nel modello di analisi si preferisca avere un dettaglio minore. Per esempio nel modello del dominio una fattura potrebbe essere rappresentata da diverse classi, come `Intestazione`, `TotaliFattura`, `Riga`, ecc. mentre nel modello di analisi si potrebbe preferire rappresentare globalmente il concetto di fattura per mezzo di una sola entità;

- l'analisi dei singoli use case permette di derivare le control class. La corrispondenza dipende dal livello di granularità dei casi d'uso, tipicamente però dovrebbe essere del tipo `1, n`: ciascuno di essi dovrebbe dar luogo ad una serie di control class (eventualmente anche una relazione `n, n` potrebbe risultare legittima), un intervallo tipico è 2-5 classi di controllo per caso d'uso, valori eccessivamente al di fuori da

questo intervallo potrebbero essere dovuti a un modello dei casi d'uso non ben realizzato;

- ogni associazione attore/caso d'uso presente relativo modello è la sede di una o più boundary class.

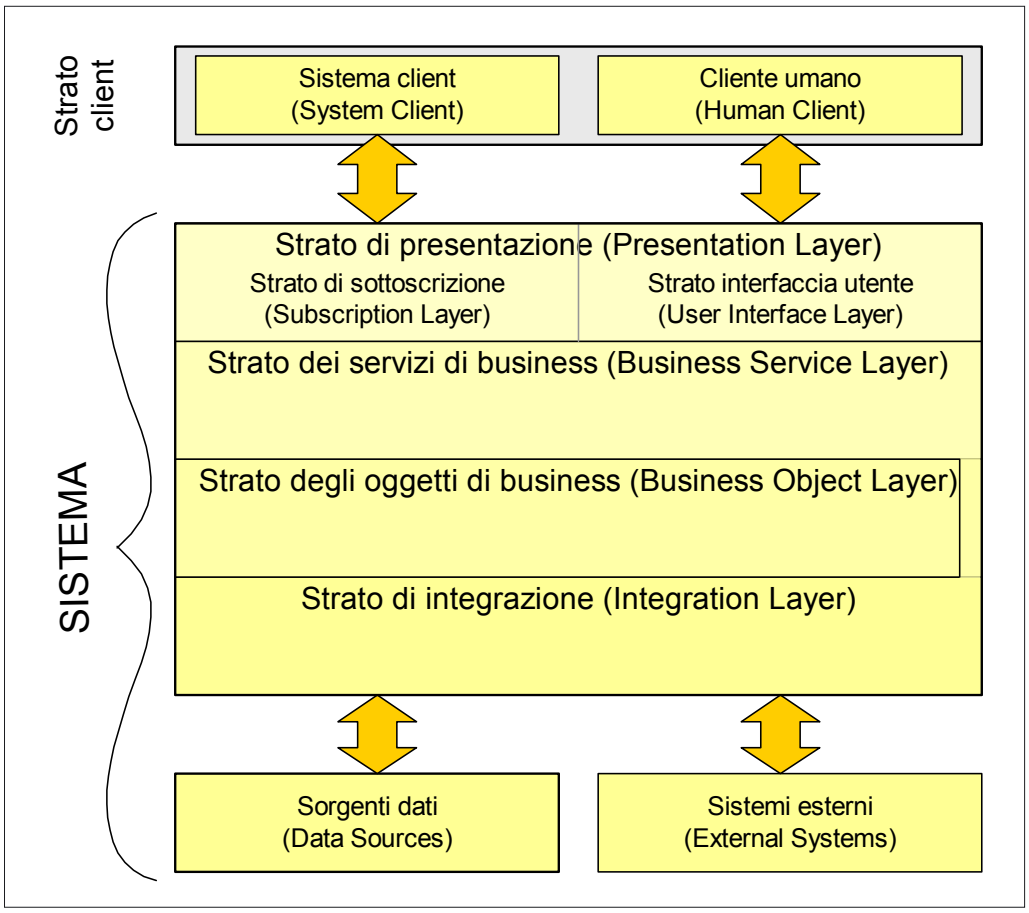
Dipendenza del modello di analisi dall'architettura

Come visto nei capitoli dedicati all'analisi dei requisiti, l'architettura del sistema fornisce importanti riscontri già nella realizzazione del modello dei casi d'uso. Tale influenza diventa più pressante man mano che si procede nelle fasi a maggior contenuto "tecnico/soluzione" del processo di sviluppo del software. Ciò, in prima analisi, potrebbe sembrare un controsenso: come può l'architettura del sistema, risultato ultimo del modello di disegno (manufatto prodotto in una delle fasi finali del processo), influenzare modelli iniziali come quello dei requisiti e di analisi, che invece hanno luogo in fasi precedenti? La risposta è semplice. I processi formali prevedono che fin dai primissimi stadi si inizi a redigere il famoso documento dell'architettura software (SAD, *Software Architecture Document*). Si tratta di un documento "dinamico" che segue l'evoluzione del sistema: si assiste ad un passaggio da versioni iniziali ad elevato livello di astrazione fino a versioni sempre più dettagliate. In pratica, ciò che accade più frequentemente è che la produzione del SAD avvenga riutilizzando versioni prodotte per sistemi realizzati in precedenza (nella speranza che non sia troppo *sad*, triste). Da questa versione è necessario epurare dettagli strettamente legati al sistema precedente (esistono architetti in grado di dimenticarsi di tale dettaglio, e quindi trovarsi a lavorare in un sistema bancario e incontrare riferimenti a un sistema di gestione dei voli aerei!), aggiungere descrizioni/modelli relativi al nuovo sistema, prescrizioni di nuove soluzioni tecnologiche, e così via. Il documento ottenuto viene quindi incorporato nel nuovo processo. Quindi, già dalle primissime fasi del processo, si ha a disposizione un documento di architettura preciso, dettagliato, maturo…

Nella malaugurata ipotesi di non disporre di una precedente versione di un SAD, comunque si assiste al classico ciclo con riscontro: le prime versioni dei requisiti utente permettono di realizzare le versioni iniziali dell'architettura del sistema, le quali, a loro volta, forniscono un prezioso riscontro al modello dei requisiti, che, opportunamente aggiornato, comincia ad avvicinarsi all'architettura. La nuova versione dei requisiti permette di raffinare il disegno dell'architettura, e così via fino a raggiungere un'armoniosa consistenza.

In ogni modo, indipendentemente dalla strategia utilizzata per redigere il SAD e dalla tecnologia che si decida di impiegare (al momento in cui viene redatto questo libro) l'architettura presa come modello universalmente accettato ha una tipica organizzazione multi-strato (*multi-tiered* da non confondersi con le architetture *multi-tired*…). Questa è ca-

Figura 8.32 — *Rappresentazione di un'architettura multi-strato. Lo strato di oggetti business è stato disegnato leggermente più stretto per evidenziare che in casi particolari e con le dovute cautele è possibile passare dallo strato dei servizi business direttamente a quello di integrazione.*



ratterizzata dal suddividere logicamente le responsabilità del sistema in livelli, in modo che ciascuno si faccia carico di una sola tipologia di responsabilità. Il sistema pertanto è il risultato di una sequenza di strati, in cui ognuno fornisce servizi allo strato di livello superiore utilizzando altri forniti dal livello inferiore.

Senza entrare troppo nei dettagli i vari strati che compongono l'architettura hanno responsabilità ben definite.

Strato client. Rappresenta i vari dispositivi (i clienti) che interagiscono con il sistema. Nel caso di attori umani questi possono essere istanze di browser web, applicazioni, ecc.

Nel caso di attori non umani, si può trattare di sistemi che pubblicano messaggi (per esempio via FTP, sistema di messaggistica, …), che invocano servizi on-line, ecc. Il sistema oggetto di studio può, a sua volta, utilizzare meccanismi analoghi per richiedere servizi ad altri sistemi distinti da quelli clienti (esserne un attore). A tal fine è presente lo strato relativo ai **sistemi esterni** (nella figura, in basso a destra). Si è deciso di dar luogo ad uno strato diverso perché l'espletamento di un servizio, tipicamente, prevede di percorrere tutta la pila di strati, arrivare fino alle sorgenti dati o ai sistemi esterni per poi risalire, quindi è corretto inserire le funzionalità per richiedere determinati servizi a sistemi esterni in un apposito strato. Il risultato ottenuto è passato a ritroso per i vari strati fino a giungere all'attore che ha richiesto il servizio. Durante il percorso si assiste all'arricchimento/integrazione del servizio nel contesto di uno più completo.

Strato di presentazione. Si tratta dell'interfaccia del sistema con i propri clienti e quindi incapsula la logica richiesta per servirne le richieste (autenticazione, analisi e decodifica richieste, validazione dati, invocazione servizi business, preparazione dei risultati, ecc.). Giacché i client sono di due tipologie, la struttura dello strato ne riflette la tipizzazione. In particolare vi è quello demandato a gestire i clienti umani (validazione dati, costruzione delle pagine con i risultati, ecc.) denominato Strato interfaccia utente e quello atto a comunicare con altri sistemi (ricezione messaggi, decodifica, ecc.) etichettato Strato di sottoscrizione.

Strato dei servizi business. Questo strato centralizza la logica di business del sistema. Si dovrebbe trattare quindi di servizi derivati direttamente dalle azioni specificate nei flussi dei casi d'uso. Si tratta di uno strato di schermatura (*façade*) tra quello di presentazione, da cui riceve le richieste e ne presenta i risultati, e quello sottostante a cui delega parte delle proprie attività. Nella pratica, in alcune situazioni eccezionale può essere legittimo saltare il livello successivo (strato degli oggetti business) per richiedere direttamente i servizi presenti nello stato di integrazione.

Strato degli oggetti business. Si tratta di uno strato non sempre presente, il cui compito è di incapsulare un insieme di oggetti che costituiscono il nucleo delle informazioni business corredate da opportune regole e trasformazioni. Gli oggetti appartenenti a questo strato dovrebbero essere estratti dal modello a oggetti del dominio, raggruppandoli in una serie di grafi distinti caratterizzati dal presentare un elevato livello di coesione. In sostanza, ognuno di questi grafi dovrebbe costituire il dominio di un apposito componente.

Strato di integrazione. In questo strato vengono inseriti gli oggetti (e/o componenti) che si occupano di interagire con le fonti dati (gestori di database, legacy system, sistemi business-to-business, sistemi esterni per l'autorizzazione di transazioni, etc.).

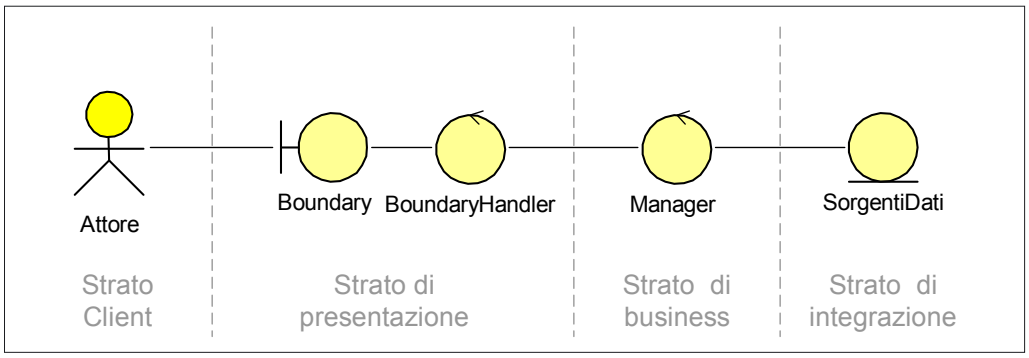
> Da notare che i precedenti periodi sulla definizione dell'architettura del sistema e spiegazione dei vari strati rappresentano un eccellente esempio di un paragrafo appartenente al SAD (*Software Architecture Document*, documento dell'architettura software).

Una volta stabilita l'architettura è opportuno che fin dal modello di analisi se ne rispetti la struttura. Chiaramente non è strettamente necessario farlo a questo livello né tanto meno indispensabile rispettarla sempre nei minimi dettagli, però forgiare il modello di analisi secondo la struttura dell'architettura offre tutta una serie di vantaggi: permette di realizzare una migliore astrazione del modello di disegno (che invece deve assolutamente rispettare le struttura dell'architettura), fornisce migliori informazioni circa la consistenza finale del sistema e quindi dei tempi e costi di disegno ed implementazione, agevola la pianificazione dei componenti, e così via.

A questo punto, stabilita l'architettura, come fa il modello a rispettarne le direttive? Semplice. Ogni servizio va realizzato estendendo uno scheletro basilare come quello riportato nella fig. 8.33.

Dall'analisi di fig. 8.33, si evince che tipicamente, a questo livello, i due strati "business" (service e object) sono incorporati in uno solo. Procedendo per gradi, ciascun servizio può essere richiesto sia da un attore sia umano, sia da un altro sistema. Le richieste devono necessariamente essere effettuate o pervenire ad un'apposita interfaccia. Nel caso di attore umano si tratterà di una pagina HTML, di una schermata di un'applicazione, di un ricevitore di input vocali, ecc. Mentre nel caso di attori non umani, si tratterà di un server FTP, di un modulo di ricezione messaggi ecc. Una sola classe boundary, per servizio, non è sempre sufficiente, alcune volte sono necessarie delle altre per presentare i risultare, per autenticare il cliente, ecc. Ad ogni interfaccia utente è necessario associare un apposito Handler (ciò è particolarmente utile per sistemi web). Questo si occupa di gestire e coordinare le varie interfacce, di richiedere gli appositi servizi al livello di business, ecc. Come ricordato poc'anzi, in questa fase, tipicamente è prematuro suddividere lo strato di business in quello di servizi ed oggetti. Se da un lato è corretto rappresentare il sistema a strati, dall'altro bisogna far attenzione a non esagerare con il livello di dettaglio, che correrebbe

Figura 8.33 — *Template di un servizio descritto dal modello di analisi.*



il rischio di appesantire inutilmente il diagramma, di aumentarne notevolmente la complessità, di richiedere maggiore tempo per la produzione e manutenzione.

Per poter espletare i propri servizi, lo strato di business in genere accede ad opportune fonti dati. Le relative interazioni sono schermate appunto dallo strato di integrazione.

Regole fondamentali per la produzione del modello di analisi

Come si vedrà di seguito, l'iniziale successo del modello di analisi è stato fortemente ridimensionato dall'introduzione dei sistemi component based e dalla standardizzazione di architetture EJB e .net. In ogni modo, esso è parte integrante di diversi processi di sviluppo del software, tra cui ICONIX (documentato nel libro [BIB05] e a cui si è fatto accenno nel capitolo introduttivo) nel quale è integrato in una metodologia nota con il nome di *Robustness Analysis* (analisi di robustezza). Questa prevede di prendere come prodotti di partenza il modello dei casi d'uso e quello a oggetti del dominio. In particolare è necessario leggere ogni singola frase dei vari casi d'uso e quindi aggiornare di conseguenza il modello di analisi con gli attori connessi ad appropriate boundary class che rappresentano le interfacce del sistema, poi si aggiungono le control class che raggruppano specifici insiemi di passi dello use case. Queste accedendo alle entity class prelevate dal modello del dominio e forniscono i servizi richiesti dagli attori (come dettagliato nel paragrafo precedente).

A modello prodotto, chiunque dovrebbe essere in grado, modello degli use case alla mano, di leggere i vari passi di cui sono composti i flussi e quindi individuare le corrispondenze con il modello. Non è infrequente il caso di architetti che indichino esplicitamente nelle control class i passi degli use case che realizzano. Se da un punto di vista di chiarezza ciò produce diversi vantaggi, esistono tutta una serie di controindicazioni, come per esempio un tempo elevato di produzione, una fortissima dipendenza dal modello dei casi d'uso (una piccola modifica in questi genera la modifica del modello di analisi), e così via.

La realizzazione del modello di analisi, secondo la metodologia Robustness, prevede di rispettare le quattro regole fondamentali riportate di seguito (dopo la lettura del paragrafo precedente, queste regole dovrebbero risultare del tutto naturali, quasi superflue).

Le quattro regole sono:

1. **gli attori possono essere associati unicamente a boundary class**. Nel caso in cui l'attore sia un essere umano, la relativa interfaccia è costituita da opportune schermate o da specifici moduli di analisi vocale (chiaramente un attore non si esprime in sequenza di zeri ed uni). Nel caso in cui sia un altro sistema/dispositivo, l'interfaccia è rappresentata da moduli di acquisizione/analisi dei messaggi (per esempio è necessario costruire un modulo sia per sottoscriversi a specifiche code di messaggi, sia per ricevere i messaggi stessi). Ciò è necessario sia per adattare i messaggi ricevuti al proprio sistema, sia per effettuare opportuni controlli: sicurezza, validità dei dati, ecc.;

2. **le classi boundary possono essere associate esclusivamente con attori e control class**. In sostanza questa regola equivale ad affermare che non è possibile associare due boundary class tra loro, così come non è possibile collegare direttamente una boundary class con una entity. Ciò costituirebbe un'evidente violazione dell'organizzazione a strati, e sarebbe fonte di tutta una serie di conseguenze non desiderabili: esposizione delle classi entità, realizzazione di interfacce utente pesanti con incorporate le sezioni relative alla business logic, alla sicurezza, ecc. (queste sezioni tra l'altro dovrebbero essere copiate in svariate interfacce), e così via. Per quanto concerne due boundary, qualora necessario, non è mai opportuno stabilire al loro interno quali altre interfacce richiamare: si tratta di un compito delle control class (è sempre opportuno separare i componenti view dai relativi control). Ciò è opportuno per isolare le interfacce utente da eventuali variazioni delle business rule o delle policy aziendali, per mantenere chiare e limitate le responsabilità dell'interfaccia utente, e così via;

3. **le classi entità possono essere associate esclusivamente con quelle di controllo**. In questo caso si vieta di associare tra loro due *entity class* e di collegarle direttamente o con le *boundary* o, addirittura, con gli attori. Per quanto riguarda le ultime postille le motivazioni alla base sono state ampiamente illustrate. Per ciò che attiene al divieto di associare tra loro le classi entità si tratta di una cattiva organizzazione per tutta una serie di motivi: si muoverebbero nell'organizzazione dei dati, regole del business, causando la perdita del livello di indipendenza, la chiarezza del disegno e delle responsabilità, la schermatura dai cambiamenti dei requisiti, ecc.;

4. **le classi control non possono essere associate con gli attori**. Questa regola dovrebbe essere ormai del tutto evidente: gli attori per interagire con il sistema devono passare attraverso opportuni strati di interfacciamento.

Esempi

Sottomissione modulo di accesso

Dall'analisi di fig. 8.34, si evince come sia possibile raffinare il diagramma introducendo un ulteriore livello gerarchico come riportato nella fig. 8.35.

Dal punto di vista strettamente tecnico, la nuova struttura è chiaramente più elegante e precisa. In poche parole offre un superiore livello object oriented. Nonostante questi vantaggi, l'esperienza insegna che i clienti non esperti del mondo OO (come nella quasi totalità dei casi) tendono a non sentirsi a proprio agio quando si trovano di fronte ad organizzazioni di use case che facciano uso

di astrazioni, e nel diagramma di figura sono presenti addirittura due livelli. Pertanto, strutture di questo tipo spesso vanno purtroppo sacrificate al fine di instaurare una migliore piattaforma di dialogo con il cliente: obiettivo fondamentale delle prime fasi del processo di sviluppo del software. Comunque non c'è poi da rammaricarsi troppo, l'eventuale mancanza di un'organizzazione astratta dei manufatti appartenenti a queste fasi è assolutamente recuperabile nei modelli prodotti negli stadi successivi a maggiore connotazione tecnica.

Figura 8.34 — *Use case* `Sottomissione Modulo di Accesso`.

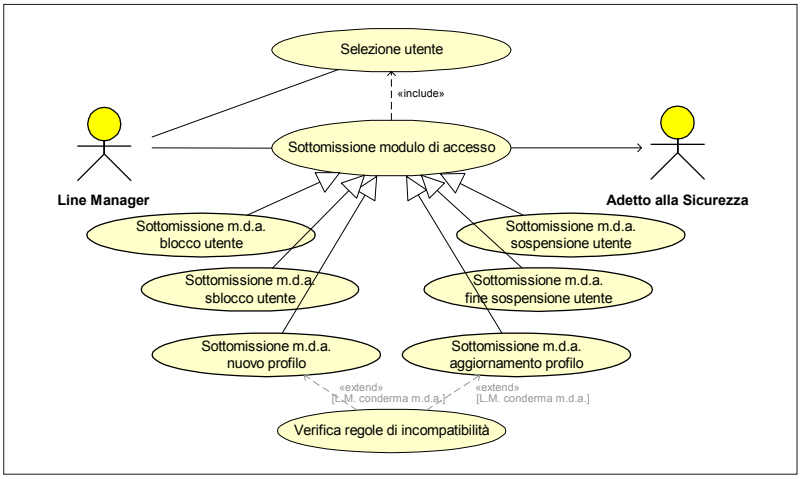


Figura 8.35 — *Use case* `Sottomissione Modulo di Accesso` *riorganizzato.*

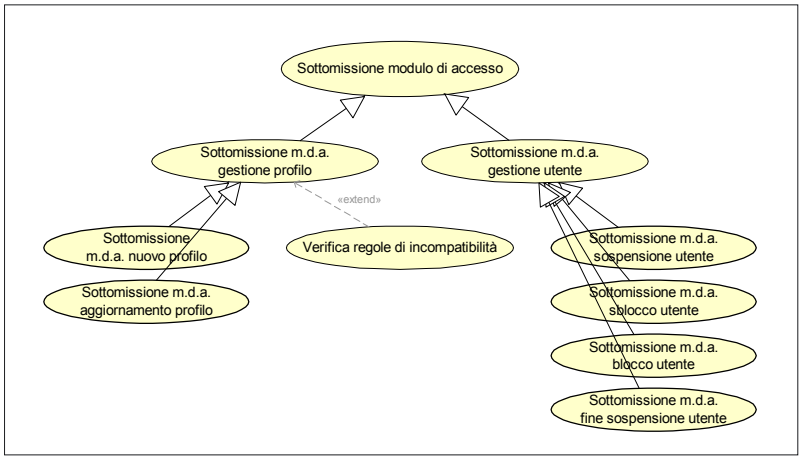


Tabella 8.4

CASO D'USO: **SICSTTMDA**	**Sottomissione Modulo di accesso**	Data: 10/VI/2002 Versione:0.00.001
Descrizione:	Il servizio permette ai *Line Manager* (precedentemente autenticati dal sistema) di sottomettere dei moduli di accesso, i quali sono automaticamente posti all'attenzione di un opportuno addetto alla sicurezza per la validazione (*confrontare state chart di figura 3*). Le tipologie di modulo di accesso previste sono: • richiesta di un nuovo profilo; • variazione di un profilo esistente; • blocco/sblocco utente; • sospensione/fine sospensione utente.	
Durata:	Minuti.	
Priorità:	Elevata.	
Attore primario:	Line Manager	
	Ha interesse nel compilare un nuovo modulo di accesso.	
Attore secondario:	Addetto alla Sicurezza	
	Riceve notifica della presenza di nuovo modulo di accesso da esaminare.	
Precondizioni:	Il Line Manager sia stato precedentemente autenticato.	
Garanzie di fallimento:	Non viene preso in carico dal sistema alcun nuovo modulo di accesso.	
Garanzie di successo:	Il modulo di accesso viene correttamente registrato nel sistema e posto all'attenzione di un opportuno *Addetto alla Sicurezza*.	
Avvio:	Il *Line Manager* esegue la funzione di sottomissione moduli di accesso.	

Per la seguente trattazione il diagramma preso in considerazione è quello di fig. 8.34. Per questioni di spazio, gli unici use case dettagliati sono *Sottomissione modulo di accesso* e *Sottomissione m.d.a. nuovo profilo*. Il comportamento dinamico dei restanti non presenta alcuna caratteristica inaspettata.

Nel punto 12 del caso d'uso di tab. 8.5 ci si riferisce a un *opportuno* gruppo di addetti alla sicurezza. Il termine *opportuno* è dovuto al meccanismo di sicurezza operante a livello di dati/privacy descritta in precedenza. Il nuovo modulo di accesso sottomesso dal Line Manager implicitamente appartiene ad una determinata organizzazione e quindi è strettamente necessario che sia posto all'attenzione di un gruppo di addetti alla sicurezza abilitati per quella specifica organizzazione. In caso contrario il modulo di accesso non verrebbe mai visualizzato come risultato delle ricerche dei membri del gruppo in quanto non ne possederebbe i necessari privilegi. Da notare che l'entità modulo di accesso non possiede direttamente il riferimento ad un'organizzazione di appartenenza (si faccia riferimento al modello a oggetti del dominio). Questa informazione, come lecito attendersi, è desunta dall'organizzazione in cui è in organico l'utente a cui il modulo di accesso stesso si riferisce.

Tabella 8.5

Scenario principale.		
	LINE MANAGER	**SISTEMA**
1.	Richiede l'esecuzione del servizio di sottomissione moduli di accesso.	
2.		*Include(*Selezione Utente*).*
3.		Determina che un utente sia stato effettivamente selezionato.
4.		Verifica che l'utente selezionato non sia il *Line Manager* stesso.
5.		Mostra i dettagli dell'utente selezionato e chiede di confermare la tipologia di modulo di accesso.
6.	Imposta la tipologia di modulo di accesso che intende sottoporre.	
7.		Verifica la tipologia impostata dal Line Manager.
8.		*Punto di astrazione:* *Acquisisce dati modulo di accesso.*
9.		Mostra il dettaglio completo dei dati impostati e chiede conferma a proseguire.
10.	Conferma i dati impostati.	
11.		Memorizza il nuovo modulo di accesso.
12.		Invia notifica della presenza di un nuovo modulo di accesso ad un opportuno gruppo di addetti alla sicurezza.
13.		Comunica all'utente che il servizio è stato portato a termine correttamente.
Scenario di errore. **Il Line Manager non seleziona alcun utente.**		
3.1.	Il sistema termina con insuccesso il caso d'uso.	
Scenario di errore. **Il Line Manager seleziona sé stesso.**		
4.1.	Il sistema comunica il tentativo di violazione delle policy di sicurezza.	
4.2.	Il sistema memorizza i dettagli del tentativo in un apposito file di log.	
4.3.	Il sistema termina con insuccesso il caso d'uso.	
Scenario di errore. **Il Line Manager decide di terminare anticipatamente la sessione.** *La stessa sequenza vale per i punti 6 e 10.*		
6.1.	Il sistema chiede conferma della terminazione.	
6.2.	Line Manager conferma.	
6.3.	Il sistema termina con insuccesso il caso d'uso.	
Annotazioni.		
11.	Lo stato del nuovo modulo di accesso è impostato al valore *SOTTOPOSTO*. Se il modulo si riferisce ad un profilo (nuovo o variazione di uno esistente), lo stato di quello cui il modulo di accesso fa riferimento è impostato al valore *PREPARAZIONE*. Qualora il modulo di accesso sia relativo al primo profilo di un utente, lo stato di quest'ultimo viene impostato al valore *ATTESA*.	

Nel presente template relativo al comportamento dinamico dello use case Sottomissione modulo di accesso e in alcuni dei seguenti, nella sezione relativa alla descrizione dello scenario principale, non è prevista un'apposita colonna dedicata all'attore secondario (Addetto alla Sicurezza), quantunque presente. Pertanto la descrizione dello scenario principale è ripartita nelle sole due colonne: quella dedicata al sistema e quella relativa all'attore principale (Line Manager). Questa decisione è motivata dal fatto che gli attori secondari in questione non eseguono particolari interazioni con il sistema, ma si limitano semplicemente a ricevere delle notifiche. Per questo motivo si è deciso di non pregiudicare la leggibilità del template ripartendo ulteriormente lo spazio con un'altra colonna, in questo caso abbastanza inutile.

Il diagramma riportato in fig. 8.36 rappresenta la proiezione statica del frammento del modello di analisi che realizza lo use case `Sottomissione modulo di accesso`. Sebbene fornisca molte importanti informazioni, non illustra le collaborazioni che si instaurano tra le varie entità, necessarie per erogare il servizio. In altre parole non vi è alcuna direttiva circa il comportamento dinamico. Questo è fornibile attraverso diversi strumenti, come il linguaggio naturale, i diagrammi di interazione, e così via.

Si inizi con il considerare l'illustrazione del comportamento dinamico illustrata per mezzo del linguaggio naturale. Il servizio viene avviato a seguito della esplicita richiesta dell'utente: il `Line Manager` accede allo schermo (`page`) relativo alla gestione dei moduli di accesso (`ModuloAccessoPage`) e ivi seleziona la funzione di sottomissione di un nuovo modulo di accesso. Il controllo passa quindi all'istanza del `ModuloAccessoHandler`, il quale delega all'istanza `SottomissioneMdAHandler` il compito di gestire il servizio. Il primo passo da compiere consiste nel selezionare l'utente oggetto del futuro modulo di accesso, pertanto, il controllo passa al control `SelezioneUtenteHandler`. Confrontando lo use case `Sottomissione modulo di accesso`, il passo consiste nell'inclusione dello use case `Selezione Utente`.

È presentata all'utente la pagina per l'acquisizione dei dettagli dell'utente da reperire (`SelezionaUtentePage`). Il `Line Manager` imposta i valori di alcuni campi da utilizzarsi come criterio per il reperimento e quindi attiva il link di ricerca. Il controllo passa nuovamente all'istanza `SelezioneUtenteHandler`, che deroga la ricerca al control `UtentiManager`. Quest'ultimo si occupa di reperire la o le istanze utente corrispondenti ai criteri di ricerca impostati dal `Line Manager`. I risultati sono rinviati a ritroso fino al `Line Manager` stesso. Questo processo prosegue fin quando il `Line Manager` individua e conferma i dettagli relativi all'utente desiderato, oppure decide di terminare prematuramente il processo.

Reperito l'utente, il flusso degli eventi da esaminare è quello sancito nello use case `Sottomissione modulo di accesso nuovo profilo`. Il controllo torna all'istanza del control `SottomissioneMdAHandler`. A questo punto, quest'ultimo richiede a `ModuloAccessoManager` di fornire i dati necessari per preparare il modulo per l'im-

Tabella 8.6

CASO D'USO: **SICSTTNPR**	**Sottomissione** **modulo di accesso nuovo profilo**	Data: 10/VI/2002 Versione:0.00.001
Descrizione:	Il presente use case specializza il servizio definito nello use case *Sottomissione modulo di accesso*, con il comportamento necessario per permettere al *Line Manager* di richiedere l'assegnazione di un nuovo profilo per l'utente selezionato.	
Durata:	Minuti.	
Priorità:	Elevata.	
Attore primario:	Line Manager	
	Ha interesse nel compilare un nuovo modulo di accesso per richiedere un nuovo profilo per un proprio subalterno.	
Garanzie di fallimento:	Non viene preso in carico dal sistema alcun nuovo modulo di accesso.	
Garanzie di successo:	Il modulo di accesso viene correttamente registrato nel sistema e posto all'attenzione di un opportuno *Addetto alla Sicurezza*. Se l'utente a cui si riferisce il modulo di accesso si trova nello stato di *INSERITO*, questo viene aggiornato in *ATTESA*.	

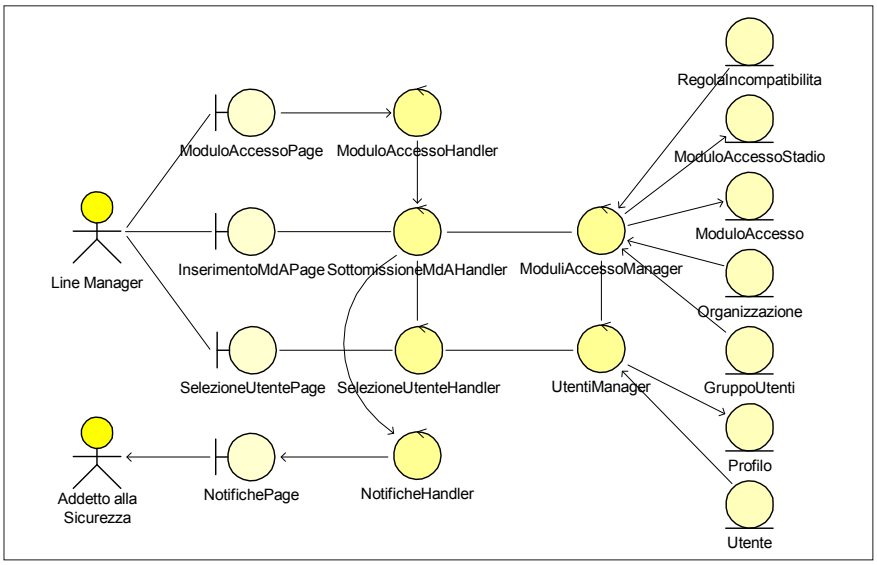
Scenario principale.

	LINE MANAGER	**SISTEMA**
	Definizione punto di astrazione: Acquisisce dati modulo di accesso. *Il Line Manager richiede di impostare un modulo di accesso per un nuovo profilo.*	
1.		Predispone il modulo necessario per impostare un nuovo modulo di accesso relativo ad un nuovo profilo. *Prototipo interfaccia X.Y.Z.*
2.	Imposta i dati presenti. *Confrontare Domain Object Model.* *(Descrizione, priorità, data e orario di inizio e fine validità, ...)* In particolare imposta i vari *Contesti di sicurezza*, ciascuno dei quali accoppia uno specifica istanza di *Gruppi Utente* ad una determinata istanza dell'entità *Organizzazioni*. In sostanza si definiscono le organizzazioni in cui i servizi abilitati dai *Gruppi utente* saranno abilitati.	
3.		Verifica i dati impostati dal *Line Manager*. (*Business Rules x1.y2*)
4.		*Punto di estensione:* *Verifica regole di incompatibilità.* *Condizione:* *Il Line Manager ha impostato i dati correttamente.*
5.		Accerta che la verifica del punto precedente non abbia evidenziato violazioni.
6.		Termina il presente segmento

Tabella 8.7

Scenario alternativo. **I dati impostati dall'utente sono errati** *(confrontare business rules x.y)***.**	
3.1.	Il sistema comunica l'errore individuato.
3.2.	Lo use case riprende dal punto 1.
Scenario alternativo. **Il nuovo profilo viola regole di incompatibilità di tipo "warning".**	
5.1.	L'elenco delle regole violate viene mostrato all'utente.
5.2.	Il sistema chiede all'utente di confermare.
5.3.	In caso di conferma lo use case procede regolarmente.
5.4.	In caso di mancata conferma il sistema esegue il punto 2 dando la possibilità di modificare i valori impostati
Scenario alternativo. **Il nuovo profilo viola regole di incompatibilità di tipo "error".**	
5.1.	L'elenco delle regole violate viene mostrato all'utente.
5.2.	Il sistema esegue il punto 2 dando la possibilità di modificare i valori impostati
Scenario di errore. **Il Line Manager decide di terminare anticipatamente la sessione.**	
3.1.	Il sistema chiede conferma della terminazione.
3.2.	Line Manager conferma.
3.3.	Il sistema termina l'intero use case con insuccesso.

Figura 8.36 — *Porzione dell'Analysis model che realizza lo use case* `Sottomissione modulo di accesso`.



missione di un nuovo modulo di accesso (elenco dei `GruppiUtente` e `Organizzazione` selezionabili, l'identificatore del nuovo profilo, ecc.). Richiede inoltre allo stesso di eseguire il controllo che l'utente selezionato non sia lo stesso di quello connesso. Superato positivamente il controllo ed ottenuti i vari dati, questi vengono utilizzati per impostare la pagina `InserimentoMdAPage` per l'inserimento dei dati. A questo punto il `Line Manager` imposta i vari dati e quindi preme il tasto di conferma. Le informazioni impostate sono passate all'handler `SottomissioneMdAHandler` che le passa al `ModuloAccessoManager` il quale verifica che non ci siano violazioni alle regole di incompatibilità e quindi provvede a memorizzare le nuove istanze delle entità `ModuloAccesso`, `ModuloAccessoStadio`, ecc. Da notare che i servizi di memorizzazione delle informazioni relative al profilo e allo stato dell'utente sono forniti dall'istanza di `UtentiManager`. L'ultimo passo prevede di inviare una specifica segnalazione ad un opportuno `Addetto alla Sicurezza`. Ciò avviene per mezzo della collaborazione tra il control `NotificheHandler` e `NotifichePage`.

Nel caso in cui si verifichi qualche inconveniente, oppure qualche controllo generi un esito negativo, viene presentata al `Line Manager` un'opportuna segnalazione invitandolo ad operare le necessarie correzioni nei dati impostati.

Per questioni di sintesi, nei precedenti periodi è stato illustrato esclusivamente lo scenario di successo, quello in cui tutto funziona perfettamente, l'utente non fa il birichino ed inserisce i dati correttamente, e così via. Chiaramente quando si disegna un modello di analisi è di fondamentale importanza accertarsi che questo sia in grado di gestire anche flussi degli eventi specificati negli scenari alternativi e di errore.

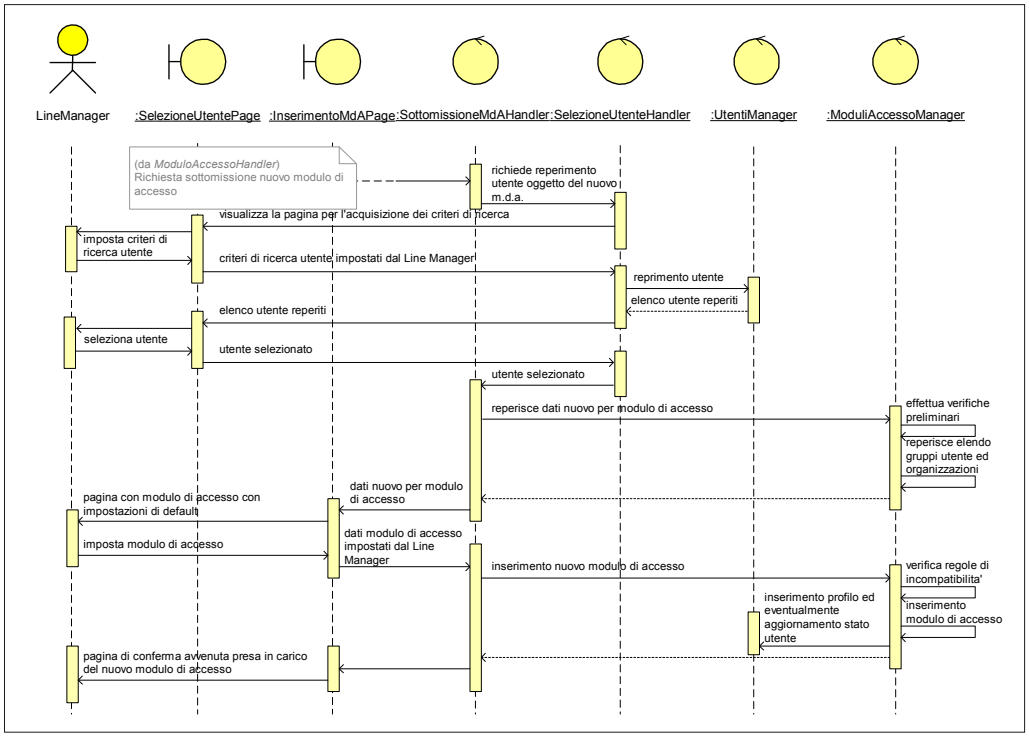
Ora tutta la descrizione riportata è sintetizzabile attraverso un unico diagramma di interazione. In contesti come il presente, tipicamente si preferisce ricorrere all'utilizzo di diagrammi di sequenza piuttosto che a quelli di collaborazione. Probabilmente perché il fattore tempo è ancora quello a cui si preferisce conferire maggiore enfasi.

Come si può notare, il formalismo grafico risulta decisamente più immediato ed accattivante. Un altro punto di forza è la capacità di evidenziare chiaramente la collaborazione temporale tra le varie entità. A fronte di questi vantaggi però esistono tutta una serie di svantaggi, come il limitato livello di dettaglio (dovuto a problemi di rendering grafico), l'elevato tempo necessario per produrre il diagramma e soprattutto per mantenerlo aggiornato, ecc.

Per quanto riguarda il livello di dettaglio, nel diagramma di fig. 8.37 si è deciso (per questioni di rendering grafico) di non visualizzare le varie entity (si tratta di una riduzione importante del contenuto informativo: non viene mostrato come i dati siano manipolati dai componenti del sistema), di mostrare il servizio dal momento in cui il processo è sotto il controllo del `SottomissioneMdAHandler` e di non evidenziare la notifica della presenza di un nuovo modulo di accesso all'`Addetto alla Sicurezza`.

L'esempio di fig. 8.38 è relativo all'approvazione del modulo di accesso. In sostanza viene saltato il servizio di validazione, in quanto ritenuto meno interessante.

Figura 8.37 — *Diagramma di sequenza (non molto dettagliato) relativo allo scenario principale del servizio di* `Sottomissione modulo di accesso nuovo profilo`.



Ancora una volta, un modello più formale, ma probabilmente meno intuitivo per un pubblico di fruitori non tecnici, potrebbe essere quello riportato nella fig. 8.39.

Per quanto riguarda la violazione delle regole di incompatibilità, c'è da notare che sebbene il controllo avvenga anche nelle fasi precedenti (`Sottomissione` e `Validazione modulo di accesso`), è opportuno effettuare nuovamente la verifica, essenzialmente per due motivi:

1. alcune violazioni della tipologia *warning* che potrebbero essere state ritenute accettabili dai precedenti attori (`Line Manager` e `Addetto alla Sicurezza`), potrebbero non esserlo altrettanto per il supervisore;

2. nell'intervallo temporale che intercorre tra una fase e quella successiva, la tabella delle regole di incompatibilità potrebbe subire degli aggiornamenti.

Figura 8.38 — *Use case* `Approvazione modulo di accesso`.

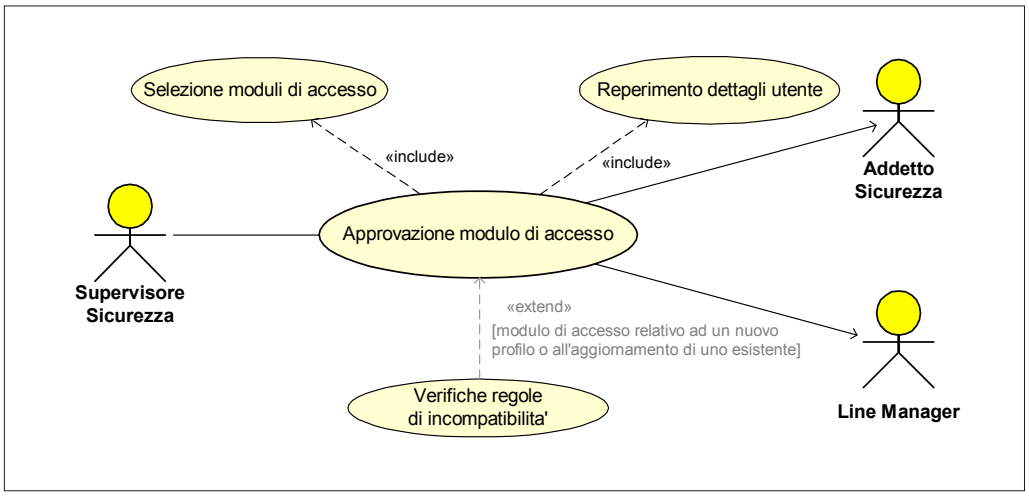


Figura 8.39 — *Use case* `Approvazione modulo di accesso` *a maggiore grado di formalità e, probabilmente, minore leggibilità da parte di un pubblico di non tecnici.*

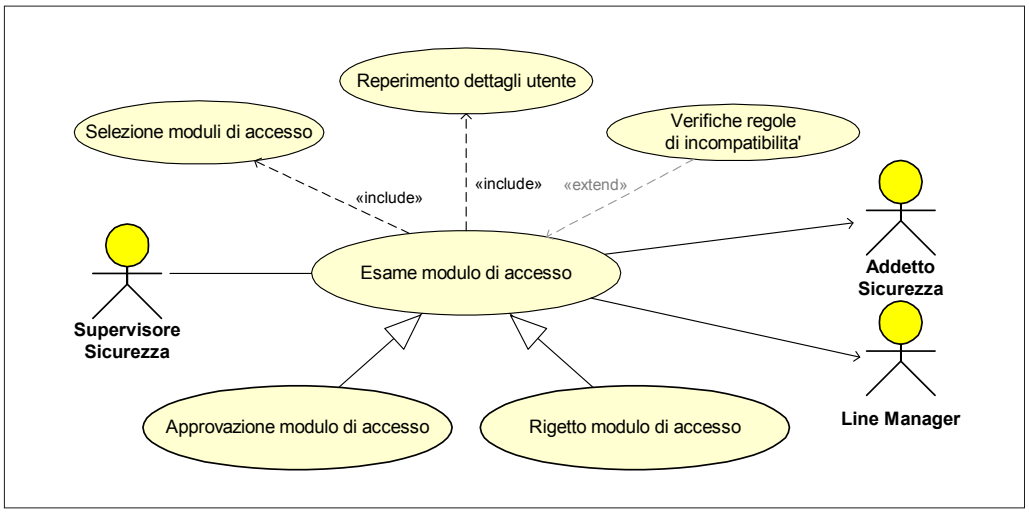


Tabella 8.8

CASO D'USO: **SICAPPMDA**	**Approvazione modulo di accesso**	Data: 10/VI/2002 Versione: 0.00.001
Descrizione:	Il servizio permette ai Supervisori alla sicurezza (precedentemente autenticati dal sistema) di esaminare e quindi approvare o rigettare, i moduli di accesso sottoposti alla loro attenzione. I moduli di accesso giungono all'attenzione dei supervisori alla sicurezza dopo essere stati validati dagli addetti alla sicurezza. *Confronta statechart diagram fig. 3.* Una volta approvato, gli effetti specificati nel modulo diventano operativi alla data di inizio validità. *Confronta modello ad oggetti del dominio.*	
Durata:	Minuti.	
Priorità:	Elevata.	
Attore primario:	Supervisore alla Sicurezza.	
	Ha interesse nell'esaminare il modulo di accesso e quindi approvarlo o rifiutarlo.	
Attore secondario:	Line Manager	
	Riceve notifica circa la decisione del supervisore alla sicurezza. In caso di rifiuto, deve intraprendere opportune azioni (correzione e risottomissione o cancellazione del modulo di accesso).	
Attore secondario:	Addetto alla sicurezza	
	Riceve notifica circa la decisione del supervisore alla sicurezza.	
Precondizioni:	Il Supervisore alla Sicurezza sia stato precedentemente autenticato.	
Garanzie di fallimento:	Non viene variato lo stato del modulo di accesso.	
Garanzie di successo:	Il modulo di accesso cambia di stato in funzione della decisione presa dal Supervisore alla Sicurezza (*APPROVATO* / *RESPINTO*) ed opportuni record sono inseriti nella tabella dei lavori schedulati.	
Avvio:	Il Supervisore alla sicurezza esegue la funzione di approvazione moduli di accesso.	

Scenario principale.

	SUPERVISORE SICUREZZA	**SISTEMA**
1.	Richiede esecuzione del servizio di approvazione moduli di accesso.	
2.		*Include(*Selezione moduli di accesso*).*
3.		Verifica che siano presenti moduli di accesso da approvare.
4.		Visualizza l'elenco dei moduli di accesso reperiti.
5.	Seleziona un modulo di accesso.	
6.		Verifica che il *Supervisore alla Sicurezza* abbia selezionato un modulo di accesso.
7.		*Include(*Reperimento dettagli utente*).*
8.		*Punto di estensione:* *Verifica regole di incompatibilità.* *Condizione:* *Il Modulo di accesso è della tipologia "Nuovo profilo" o "Aggiorna profilo".*
9.		Visualizza il modulo di accesso corredato dai dettagli dell'utente cui si riferisce e dalle eventuali regole di incompatibilità violate.
10.	Approva il modulo di accesso. Eventualmente imposta delle annotazioni.	
11.		Verifica che il modulo di accesso non sia riferito allo stesso Supervisore alla Sicurezza.
12.		Verifica che non ci siano violazioni di tipo "error" delle regole di incompatibilità.
13.		Aggiorna lo stato del modulo di accesso (eventualmente anche quello del relativo profilo) ed inserisce appositi record nella coda dei lavori batch da eseguire.
14.		Invia opportuna notifica relativa all'avvenuta approvazione, sia al Line Manager, sia all'Addetto alla Sicurezza.
15.		Memorizza l'approvazione in un apposito file di log.

Tabella 8.9

Scenario alternativo. **Il Supervisore alla sicurezza rigetta il modulo di accesso inserendo la motivazione.** **Violazione "error" delle regole di incompatibilità (punto 8)**	
10.1.	Verifica che il modulo di accesso non sia riferito allo stesso Supervisore alla Sicurezza.
10.2.	Aggiorna lo stato del modulo di accesso.
10.3.	Invia opportuna notifica relativa all'avvenuto rigetto del modulo di accesso, sia al Line Manager, sia all'Addetto alla Sicurezza.
10.4.	Memorizza il rigetto in un apposito file di log.
10.5	Termina il caso d'uso con successo.
Scenario di errore. **Nessun modulo di accesso reperito.**	
3.1.	Il sistema notifica l'anomalia.
3.2.	Il sistema termina con insuccesso il caso d'uso.
Scenario di errore. **Il Line Manager decide di terminare anticipatamente la sessione.** ***La stessa sequenza vale per i punti 5 e 10.***	
6.1.	Il sistema chiede conferma della terminazione.
6.2.	Line Manager conferma.
6.3.	Il sistema termina con insuccesso il caso d'uso.
Scenario di errore. **Il Supervisore alla Sicurezza tenta di approvare/rifiutare un modulo di accesso riferito a sé stesso.** ***Quanto segue vale anche per il punto 10.1***	
11.1.	Il sistema comunica il tentativo di violazione delle policy di sicurezza.
11.2.	Il sistema memorizza il dettagli del tentativo in un apposito file di log.
11.3.	Il sistema termina con insuccesso il caso d'uso.
Annotazioni	
13.	In funzione della tipologia del modulo di accesso il sistema genera diverse tipologie di record da inserire nella tabella relativa ai lavori schedulati. In particolare: *Nuovo ed aggiornamento profilo*: il sistema genera due record, il primo relativo all'inizio validità del profilo ed il secondo relativo alla fine validità. *Sospensione utente*: generati due record, quello relativo all'inizio della sospensione ed il secondo afferente la fine. *Blocco, sblocco e fine sospensione utente*: generato un solo record relativo, rispettivamente, alla data ed orario di blocco, sblocco e fine sospensione.

Analizzando lo use case è possibile notare la presenza di una incoerenza (evidenziata anche dal diagramma di fig. 8.39). Il nome dello use case è Approvazione modulo di accesso, pertanto l'obiettivo dello stesso, ossia il successo, è dato appunto dall'approvazione. Ora, gli scenari

alternativi sono stati definiti come varianti del flusso degli eventi definiti nello scenario principale che comunque permettono di ottenere il raggiungimento del successo. Come si può notare esiste lo scenario alternativo relativo al rifiuto del modulo di accesso, questo evidentemente non permetterebbe di raggiungere l'obiettivo dell'approvazione del modulo e quindi dovrebbe essere uno scenario di eccezione. Questa anomalia potrebbe essere risolta attribuendo un nome più consistente al caso d'uso, come per esempio analisi del modulo di accesso. Ciò però risulterebbe meno accattivante e mnemonico. Quindi, utilizzando un approccio pragmatico, si è preferito generare una piccola incoerenza per rendere il modello più chiaro e intuitivo.

Per quanto riguarda la selezione dei moduli di accesso validati e quindi eventualmente da approvare, c'è da notare che qualora non ne venga reperito alcuno, ciò non significa necessariamente che non siano presenti. Come per ogni altro utente, anche il profilo del Supervisore alla Sicurezza specifica il ruolo e l'organizzazione in cui il possessore può esercitarlo. Pertanto se presenti nella base dati moduli di accesso validati appartenenti però ad un'organizzazione diversa da quella in cui l'attuale Supervisore alla Sicurezza è autorizzato, queste non passeranno mai il filtro della "riservatezza dei dati".

Figura 8.40 — *Modello di analisi del servizio approvazione moduli di accesso.*

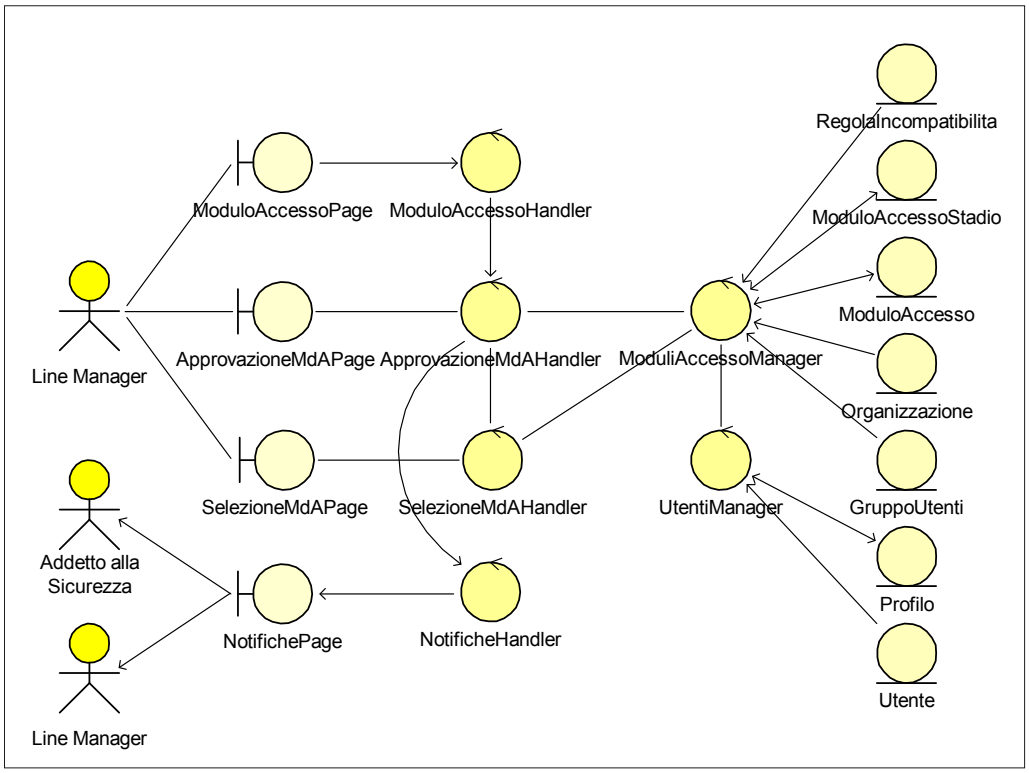
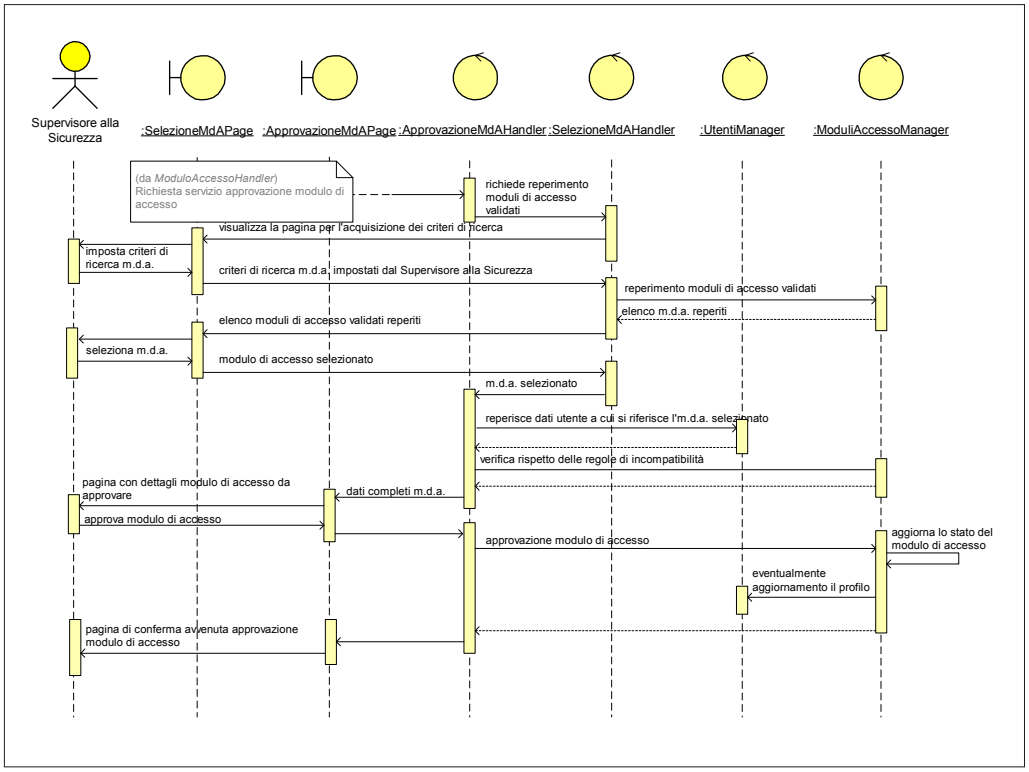


Figura 8.41 — *Diagramma di sequenza del modello di analisi relativo al servizio approvazione moduli di accesso.*



A questo punto il diagramma riportato in fig. 8.40 dovrebbe risultare abbastanza chiaro. In ogni modo in fig. 8.41 ne viene illustrato il comportamento dinamico.

Un'osservazione importante è relativa ai vari control denominati `manager`. Da un'analisi attenta è possibile notare come questi forniscano una prima versione della decomposizione del sistema in componenti. Come si può notare essi offrono una serie di servizi, e per poterli fornire accedono ad un insieme ben definito di entità. Per esempio, `ModuliAccessoManager` si occupa di fornire tutti i servizi relativi ai moduli di accesso: sottomissione, verifica, reperimento, validazione, approvazione, ecc. Per ottenere tali servizi deve accedere ad un insieme ben definito di entità: `RegolaIncompatibilità`, `ModuloDiAccesso`, `ModuloDiAccessoStadio`, ecc. Quando poi viene la necessità di manipolare dati relativi agli utente e ai relativi profili, ecco che è necessario delegare al relativo manager (`UtentiManager`) le varie operazioni.

Pro e contro del modello di analisi

Dalla lettura dei paragrafi precedenti i lettori più attenti avranno intuito che il modello di analisi non rientra nella categoria di quelli sempre strettamente necessari che se evitati amplificano notevolmente il fattore di rischio dell'intero progetto. Diversi processi, tra cui lo stesso RUP della Rational, non prevedono una fase distinta di analisi, che invece è integrata con quella di disegno (*analysis and design*). La decisione se dar luogo o meno ad una fase distinta dipende da molti fattori, non ultimi la dimensione del progetto, il tempo a disposizione, la tipologia, la tecnologia coinvolta, e così via. Qualora però si decida di dar luogo alla fase di analisi, questa dovrebbe aver luogo quasi interamente durante lo stadio di elaborazione (quella successiva alle iterazioni preliminari), e non dovrebbe richiede più di un paio di iterazioni.

L'importanza della fase di analisi è stata recentemente attenuata dalla standardizzazione di architetture del tipo Sun EJB e Microsoft .net. Tale standardizzazione, da una parte, ha generato l'esigenza di adeguare i processi di sviluppo del software alle nuove peculiari esigenze dei sistemi basati sui componenti (alcuni esempi sono forniti in [BIB15], [BIB18]); dall'altro ha concorso a ridurre il rischio tecnologico (gran parte dell'infrastruttura del sistema è resa disponibile dai vari tool, i pattern da applicare sono ben definiti, ecc). Logica conseguenza è che molta dell'attenzione è stata spostata su altri manufatti.

Per molti autori, uno dei vantaggi offerti dal modellare i requisiti utente attraverso i diagrammi delle classi deriva dalla semplicità e rapidità di aggiornamento degli stessi, caratteristica peraltro intrinseca del formalismo. Logica conseguenza è che tutti i modelli istanza dei diagrammi delle classi si prestano bene ad assorbire i temutissimi *change requirements* (cambiamento dei requisiti). Il modello di analisi in più, appartenendo ad un livello di astrazione più elevato rispetto a quello di disegno (permette di visualizzare la struttura del sistema nascondendone i dettagli più tecnici), rende più agevole analizzare nuove richieste, studiarne l'impatto, il tempo necessario e quindi i costi, e così via. L'elevato livello di astrazione (rispetto al modello di disegno) è particolarmente apprezzato anche per attività quali la formazione del nuovo personale tecnico coinvolto nello sviluppo e/o manutenzione del sistema. Disponendo di un modello di analisi corretto e aggiornato, si può risparmiare molto tempo: pur trattandosi di un manufatto che offre una vista concettuale e precisa del sistema sottostante, lo fa in modo più agevole e meno dettagliato del corrispettivo al livello di disegno.

I modelli di analisi assumono un ruolo preziosissimo quando si ha a che fare con sistemi legacy datati o con sistemi prodotti da terze parti, magari carenti di documentazione tecnica. In tali circostanze potrebbe rivelarsi molto utile disporre di una fase distinta di analisi: invece di imbattersi in difficili, costose e probabilmente inutili elaborazioni di modelli di disegno, potrebbe risultare con-

veniente realizzare un modello più concettuale. Una scelta in tal senso sarebbe giustificata da diverse considerazioni. Per esempio l'eventuale evoluzione di sistemi come quelli offerti da enti esterni, non è assolutamente controllabile e quindi mantenere aggiornato il diagramma di disegno potrebbe diventare un lavoro estenuante. Viceversa, se il sistema legacy non prevede cambiamenti o è implementato per mezzo di tecnologia a elevato grado di obsolescenza, allora è evidente che il valore aggiunto da un modello di disegno risulterebbe decisamente relativo.

Ulteriori vantaggi sono legati al fatto che la realizzazione del modello di analisi fornisce un'ottima opportunità per il team degli architetti di acquisire rapidamente e formalmente i requisiti del sistema, di rivedere rigorosamente il modello dei casi d'uso e quello a oggetti del dominio, ecc.

Lo svantaggio principale del modello è sempre lo stesso: il tempo necessario per la produzione e manutenzione. Se il costo della produzione iniziale a volte è giustificabile per i motivi poc'anzi enunciati, la relativa manutenzione lo sarebbe molto meno. Ciò dovrebbe avvenire ogni qualvolta si aggiorni il modello dei requisiti e/o quello di disegno. Questa necessità di mantenere aggiornati i vari modelli è un problema serio, basti pensare al modello di disegno, che pur di fondamentale importanza, difficilmente è mantenuto allineato con il codice. Per questi motivi spesso il modello di analisi viene considerato da molti tecnici un modello "usa e getta" (alcune volte "getta e getta"). Infatti una volta ottenuti i vantaggi legati alla realizzazione del modello iniziale non ci si preoccupa più di aggiornarlo.

Un altro svantaggio del modello di analisi è legato al fatto che nella pratica molto raramente la produzione iniziale del modello di disegno avviene disegnando tutto da zero. In genere si tende a riapplicare strutture, pattern, soluzioni utilizzate in precedenza e dimostratesi soddisfacenti. Si parte quindi cercando di riutilizzare (eventualmente reingegnerizzandole) architetture dimostratesi funzionanti in precedenti progetti. Pertanto i vantaggi offerti dalla produzione del modello di analisi vengono fortemente ridimensionati.

Modello di disegno

Introduzione

Nel corso dell'esecuzione dei processi di sviluppo del software, l'ultimo modello a oggetti che si incontra è quello relativo alla proiezione statica del modello di disegno. Nei processi di sviluppo di tipo iterativo ed incrementale, tipicamente è prodotto nella relativa fase denominata appunto di disegno, che in molti processi (tra cui il RUP) è integrata con quella di analisi (si parla pertanto di analisi e disegno). Tale fase raggiunge il culmine tra la

fine dello stadio di elaborazione (*Elaboration*) e l'inizio di quello di costruzione (*Construction*).

In maniera analoga ai modelli analizzati in precedenza, anche quello di disegno può essere decomposto nei due sottomodelli (proiezioni) relative alla struttura statica ed al comportamento dinamico. Chiaramenti, per la costruzione del sistema non è sufficiente il modello a oggetti (diagrammi delle classi): sebbene di importanza fondamentale, specifica unicamente la struttura statica del sistema, mentre nulla specifica in merito al relativo comportamento dinamico che, in ultima analisi, indica come i vari oggetti (istanze delle classi rappresentate nella proiezione statica) collaborino tra loro al fine di fornire i servizi erogati dal sistema. La descrizione delle dinamiche interne del sistema, tipicamente, è affidata ai diagrammi di interazione (sequenza, *sequence* e collaborazione, *collaboration*), e meno frequentemente, ai diagrammi di attività. Questi si rivelano particolarmente utili qualora si voglia enfatizzare l'esecuzione concorrente di determinati processi.

Resta poi sempre valido il consiglio di produrre diagrammi degli oggetti (quelli propriamente detti) per esemplificare porzioni di diagrammi delle classi ritenuti poco chiari.

Il modello di disegno è indubbiamente uno dei manufatti più importanti del processo di sviluppo del software, sebbene in molti progetti viene prodotto unicamente per scopi documentativi con tecniche di reverse engineering. Verosimilmente è il più noto tra quelli a oggetti tanto che ogni qualvolta si nomina il concetto di diagramma delle classi, l'astrazione che tende a generarsi nella mente degli ascoltatori è relativa ai diagrammi delle classi di disegno: rappresentazioni grafiche dell'organizzazione del sistema in termini implementativi (codice).

Il modello di disegno è decisamente orientato alle soluzione tecniche e pertanto si configura come ambiente naturale di architetti e programmatori. Dopo aver lungamente argomentato le funzionalità che il sistema dovrà realizzare, l'attenzione viene (finalmente) focalizzata sul dettaglio di come ottenerle. I risultati dei vari processi di analisi sono tradotti in soluzioni tecniche a bassissimo livello di astrazione, nelle quali sono condensate sia la struttura statica, sia le collaborazioni dinamiche del sistema.

Le classi presenti nei precedenti modelli a oggetti vengono "incastrate" in una serie di infrastrutture tecniche che rendono possibile realizzare quanto stabilito nella vista dei casi d'uso.

Il modello di disegno concorre a definire la descrizione analitica dell'architettura software del sistema in funzione dei requisiti catturati nelle fasi precedenti e, pertanto, rappresenta il dettaglio delle specifiche per la fase di codifica.

Di modelli di disegno (qualora non si passi direttamente dalle specifiche del cliente al codice) ne esistono diverse versioni o, se si preferisce, ne esiste un'unica che evolve durante lo svolgimento delle fasi di elaborazione e costruzione, passando dal livello di specifica via via a quello di implementazione. Le prime versioni del modello presentano un livello di dettaglio non eccessivamente approfondito, in base anche alle competenze dei programmatori a disposizione. In questo modello trovano posto, sicuramente, tutte le classi princi-

pali ed i metodi più importanti, mentre è buona norma aggiungere classi di marginale importanza in un secondo momento, in funzione delle richieste/suggerimenti provenienti dagli sviluppatori, di eventuali problemi incontrati in corso d'opera, e così via. La stessa sorte conviene al dettaglio di tutti i metodi che tipicamente è ottenuto dall'accoglimento delle decisioni implementative. È invece importante definire fin da subito i metodi pubblici (attributi pubblici non dovrebbero esistere) con la relativa firma in modo da garantire la coerenza del modello, e quindi del sistema, fin dalle primissime fasi evitando di affidarsi eccessivamente all'esperienza dei singoli programmatori. La mancata definizione dell'interfaccia delle varie classi potrebbe causare una perdita di coerenza del sistema, dovuta a scelte operate "localmente" dai singoli sviluppatori che naturalmente non possiedono la visione di insieme del progetto, o quantomeno non sono al corrente delle scelte operate localmente da altri sviluppatori.

La formalizzazione dei metodi pubblici è particolarmente utile specie quando si realizzano infrastrutture necessarie per altri sistemi, come per esempio i famosi framework. In questi casi è auspicabile che l'utente dell'infrastruttura non possa cadere in confusione (per esempio perché diversi programmatori utilizzano differenti convenzioni di "disegno") per via di tecniche non consistenti presenti nel sistema.

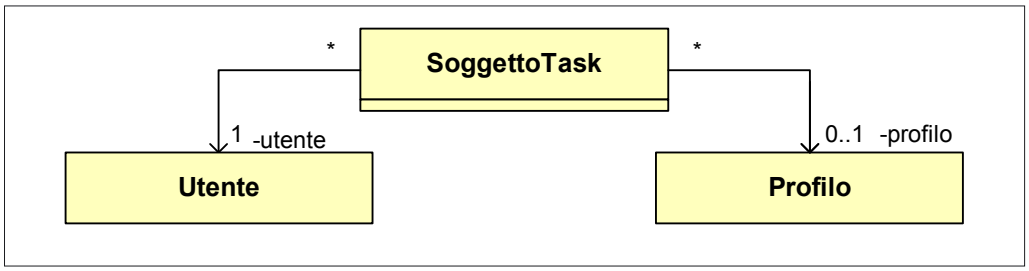
Il modello di implementazione, invece, nella versione finale è una rappresentazione grafica del codice (tenendo sempre presente che "solo il codice è sincronizzato con sé stesso" [S. Ambler]). Spesso è realizzato "raffinando" il modello di specifica attraverso il reverse engineering del codice, oppure integrando "in linea" le varie parti prodotte dagli sviluppatori attraverso lo sviluppo integrato del codice e del disegno.

Uno degli input per la produzione del modello di disegno è costituito dal modello di analisi, che, come visto, fornisce una definizione rigorosa dei requisiti funzionali tralasciando quelli non funzionali, che invece rappresentano una parte fondamentale degli aspetti considerati dal modello di disegno. Sebbene i requisiti non funzionali non siano direttamente considerati nel modello di analisi, c'è da dire che questi fanno sentire la propria influenza fin dalle fasi di analisi dei requisiti. Pertanto, a differenza del modello di analisi, quello di disegno realizza fisicamente i casi d'uso focalizzando l'attenzione su come i requisiti funzionali e non funzionali del sistema e gli altri vincoli si ripercuotono sul sistema in costruzione.

Non è infrequente il caso in cui il modello di analisi venga tralasciato per questioni di economia dei tempi. Chiaramente la decisione di trascurare il modello di analisi comporta un certo rischio che però può essere gestito e mitigato dalla presenza di architetti esperti.

Qualora non si abbia a disposizione il modello di analisi, è necessario realizzare quello di disegno partendo direttamente dai casi d'uso, accogliendo direttive dell'architettura, cercando di riutilizzare soluzioni mostratesi efficaci in precedenti progetti, sfruttando i pattern previsti dalle particolari tecnologie impiegate, ecc. Tipicamente gli architetti più esperti riescono a produrre, senza grossi problemi, prime versioni del modello di disegno a partire dai casi d'uso e dai modelli a oggetti del dominio e/o del business.

Figura 8.42 — *Esempio di utilizzo dei ruoli associazione.*



Poiché il modello di disegno è un'astrazione del codice, è necessario che accolga caratteristiche dipendenti dal linguaggio di programmazione. Ciò è utile considerando che spesso il disegno iniziale viene rifinito dopo e/o durante la codifica per mezzo del reverse engineering o produzione sincronizzata. Se per esempio si dovesse decidere utilizzare Java come linguaggio di programmazione, evidentemente avrebbe poco senso organizzare delle classi per mezzo di relazioni di ereditarietà multipla, in quanto il linguaggio non prevede tale tipologia di generalizzazione.

Ulteriori vincoli da tenere presenti durante la realizzazione del modello di disegno derivano dall'ambiente di produzione del codice, da proprietà quali riusabilità dei componenti, scalabilità del sistema, tecnologie a disposizione (per esempio database relazionali o OO, sistema operativo, ecc.), e così via.

Anche le relazioni con cui le classi sono associate tra loro nel modello di disegno hanno una corrispondenza biunivoca con la relativa rappresentazione in codice. Pertanto, piuttosto che specificare i nomi alle varie associazioni (comunque sempre possibile), è preferibile utilizzare i ruoli che le classi recitano nell'associazione stessa. I nomi di questi ruoli sono i nomi degli attributi utilizzati per realizzare la relazione.

La codifica della classe `SoggettoTask` riportata nella fig. 8.42 genererebbe un codice simile del tipo:

```
Class SoggettoTask {

...
private Utente  utente = null;
private Profilo profilo = null;
...
}
```

Da notare che, poiché la classe `SoggettoTask` non è navigabile in nessuna delle due relazioni, nelle classi `Utente` e `Profilo` non è presente un attributo di tipo `SoggettoTask`.

In questa fase anche le definizione precisa delle diverse tipologie di relazioni ha un significato semantico importantissimo. Per esempio specificare un'aggregazione piuttosto che una composizione ha molte implicazioni in termini della gestione del ciclo di vita degli oggetti istanze delle classi coinvolte nell'associazione stessa. In particolare una classe composta dovrà fornire tutti i metodi necessari per creare e distruggere le istanze delle classi che la compongono.

Qualora si dia luogo al modello di disegno a partire dal modello di analisi, è necessario tenere presenti alcune differenze. Per esempio, nel modello di analisi tutte le classi evidenziate dovrebbero appartenere ad una delle tipologie viste in precedenza (control, boundary, entity), nel modello di disegno si possono avere molte altre tipologie che dipendono sia dall'applicazione di pattern (per esempio factory, pool, ecc.) sia dal linguaggio e dall'architettura utilizzata. Per esempio se si decidesse di basare il sistema sull'architettura EJB, probabilmente si avrebbero stereotipi del tipo `EJBEntity`, `EJBStatelessSession`, ecc. Nel modello di disegno poi, è molto importante definire concetti come le interfacce. Queste sono a tutti gli effetti dei contratti stipulati tra classi server (quelle che offrono i servizi dichiarati nell'interfaccia), e classi client. La loro definizione permette lo sviluppo in parallelo dei vari sottosistemi (l'interazione è definita a priori), aumenta la riusabilità del codice, offre maggiore scalabilità, permette un migliore assorbimento dei cambiamenti di specifiche e così via.

Esempio

Come esempio di modello di disegno si è preso in considerazione il processo temporizzato che si occupa di analizzare i lavori presenti nella tabella dei task pendenti (`PendingTask`) al fine di eseguire quelli la cui data ed orario di esecuzione è stata raggiunta (fig. 8.43).

Figura 8.43 — *Caso d'uso relativo all'esecuzione dei lavori pendenti.*

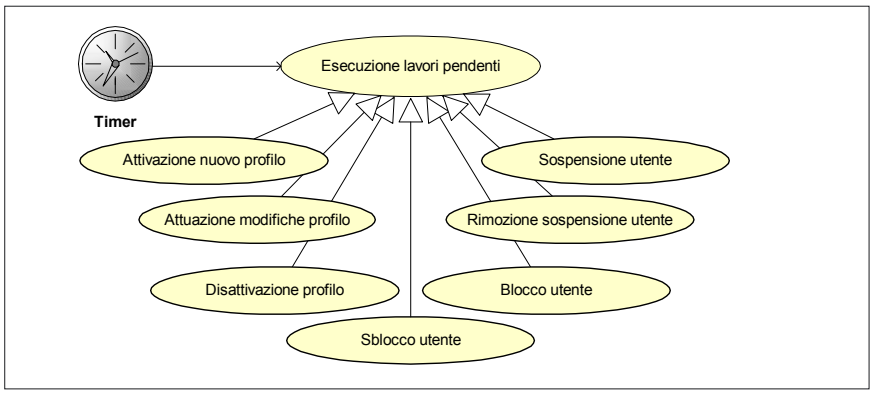


Tabella 8.10

CASO D'USO:	Esecuzione lavori pendenti	Data: 10/VI/2002
		Versione:
SICGSTTSK		0.00.001
Descrizione:	Il servizio si occupa di reperire tutti i lavori pendenti la cui data ed orario di esecuzione precedono la data ed orario attuale non precedentemente eseguiti e quindi di eseguirli. In particolare, i task possono richiedere uno dei seguenti servizi: rendere operativo un nuovo profilo, attuare gli aggiornamenti di un profilo, disattivare un profilo, sospendere un utente, rimuovere la sospensione di un utente, bloccare e sbloccare un utente.	
Durata:	Secondi.	
Priorità:	Elevata.	
Attore primario:	Timer	
	Fornisce lo stimolo temporizzato per eseguire il presente servizio. *L'intervallo tra due stimoli successivi è specificato nella business rule: x.y*	
Precondizioni:	Il timer fornisca gli stimoli ad intervalli di tempi prestabiliti.	
Garanzie di fallimento:	Non viene eseguito alcun lavoro e lo stato del sistema rimane inalterato.	
Garanzie di successo:	I lavori richiesti vengono eseguiti correttamente e lo stato del sistema viene aggiornato conseguentemente.	
Avvio:	Il timer fornisce lo stimolo.	
Scenario principale.		
	SISTEMA	
1.	Riceve la richiesta di esecuzione del servizio.	
2.	Reperisce dalla coda dei lavori pendenti (*PendingTask*), quelli la cui data ed orario di esecuzione è minore della data ed orario attuale e che non siano stati precedentemente eseguiti. *(Consultare modello ad oggetti del dominio).*	
3.	Per ogni lavoro reperito:	
3.1.	*Punto di astrazione:* *Esegue il lavoro reperito in funzione della relativa tipologia.*	
3.2.	Elimina il lavoro dalla coda.	
4.	Termina lo use case correttamente.	
Scenario di errore. **L'esecuzione del singolo lavoro termina con errore.**		
3.2.	Aggiorna il flag di esecuzione del task.	
3.3.	Memorizza l'errore occorso in un opportuno file di log.	
3.3.	Continua l'esecuzione considerando il lavoro successivo.	

Come al solito, dei vari casi d'uso che specializzano quello astratto (*Esecuzione lavori pendenti*), per questioni di sintesi, ne viene riportato solo uno. In questo caso si è deciso di illustrare il comportamento relativo all'attivazione di un nuovo profilo.

Tabella 8.11

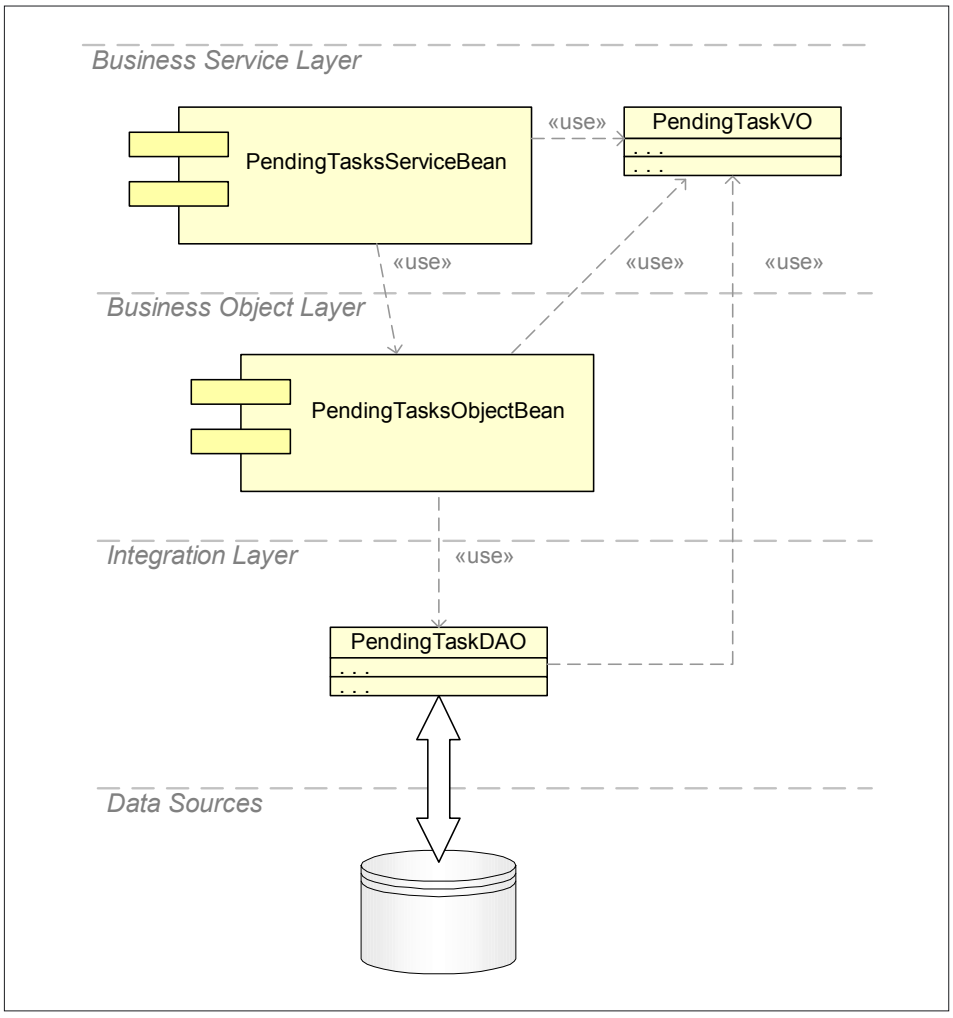
CASO D'USO: SICATTPRF	Attivazione nuovo profilo.	Data: 10/VI/2002 Versione: 0.00.001
Descrizione:	Il presente caso d'uso illustra la sequenza necessaria per rendere operativo il profilo specificato. In sostanza dalla coda dei lavori pendenti viene reperito un lavoro relativo a un nuovo profilo da rendere operativo (data e orario di esecuzione del lavoro coincide con i valori di data e orario di inizio validità del profilo)	
Durata:	Secondo.	
Priorità:	Elevata.	
Precondizioni:	Siano disponibili i dati del task relativo all'attivazione del nuovo profilo.	
Garanzie di fallimento	Non viene cambiato lo stato dell'utente e dell'eventuale profilo corrente a cui il task si riferisce	
Garanzie di successo:	Il nuovo profilo viene correttamente reso operativo	
Avvio:	Selezionato un lavoro relativo ad un profilo da rendere operativo.	

Scenario principale.	
	SISTEMA
1.	Reperisce i dettagli dell'utente a cui si riferisce il nuovo profilo.
2.	Verifica che l'utente non sia stato disattivato (stato utente <> DISATTIVATO)
3.	Verifica che l'utente non si trovi nello stato di ATTESA :
4.	Verifica che l'utente non abbia alcun profilo nello stato OPERATIVO
5.	Reperisce il profilo da rendere operativo.
6.	Aggiorna lo stato del profilo (da stato di ATTESA ad OPERATIVO)
7.	Memorizza l'attività in un apposito file di log.
8.	Termina lo use case correttamente.
Scenario alternativo. L'utente si trova nello stato di ATTESA	
3.1.	Aggiorna lo stato dell'utente (da ATTESA a OPERATIVO).
3.2.	Memorizza l'attività in un apposito file di log.
Scenario alternativo. L'utente dispone di un profilo OPERATIVO .	
4.1.	Reperisce l'attuale profilo operativo dell'utente.
4.2.	Aggiorna lo stato del profilo reperito: - Se la data di fine validità del profilo esistente precede la corrispondente del nuovo profilo (in sostanza il profilo reperito non ha più la possibilità di tornare allo stato operativo), allora ne cambia lo stato da OPERATIVO a STORICO - Altrimenti lo stato dell'esistente profilo cambia da OPERATIVO a DORMIENTE
3.2.	Memorizza l'attività in un apposito file di log.
Scenario di errore. L'utente è stato disattivato.	
2.1.	Memorizza l'errore in un apposito file di log.
2.2.	Termina lo use case con insuccesso.

A questo punto, utilizzando un processo formale, sarebbe lecito attendersi il modello di analisi relativo al servizio oggetto di studio. In questo caso lo si è omesso, essenzialmente per due motivi:

1. nei processi reali non è infrequente il caso in cui si rinunci alla fase di analisi, concentrandosi direttamente sul disegno del sistema, per questioni di risparmio di tempo;
2. in questo contesto, il modello di analisi è facilmente ottenibile da quelli relativi ai modelli riportati in precedenza.

Figura 8.44 — *Schema di massima dell'architettura.*



L'attenzione viene focalizzata sul servizio da realizzare, mentre, per il momento, viene trascurato lo schedulatore (timer) incaricato di fornire gli stimoli (invocazioni) ad intervalli di tempo prestabiliti.

L'architettura presa come riferimento è J2EE della Sun, sebbene la comprensione dell'esempio non richieda competenze specifiche.

Il modello di disegno realizzato rispetta l'organizzazione a strati sancita dall'architettura stabilita nel SAD *(Software Architecture Document)*.

> Dall'analisi del diagramma di fig. 8.44, è possibile notare che il modello presenta un livello di astrazione elevato ed un grado di formalità abbastanza contenuto. Ciò è dovuto a diversi fattori. In primo luogo una rappresentazione di maggior dettaglio avrebbe richiesto diverse pagine (per esempio sarebbe stato necessario modellare propriamente i componenti secondo le specifiche). In secondo luogo, nella pratica, non è infrequente ricorrere a modelli iniziali di questo tipo per fissare le idee, magari disegnandoli su una lavagna, per poi sostituirli con versioni più raffinate, in un secondo momento quando ormai si ha una certa fiducia della bontà della soluzione.

In generale, il livello di dettaglio a cui giungere è un problema tipico di quando si realizzano modelli. Spesso si tende a cadere nel tranello che un diagramma dettagliato rappresenti sempre l'alternativa migliore. Fortunatamente le cose non stanno così. Produrre modelli dettagliati non è sempre una buona idea, perché in primo luogo richiede molto tempo per realizzarli e ciò non è opportuno qualora gli obiettivi siano di iniziare a fissare le idee, esplorare diverse alternative, o illustrare delle scelte, poi, corrono il rischio di distogliere l'attenzione dagli aspetti veramente importanti. Quindi, il livello di dettaglio a cui scendere dipende dagli obiettivi che il modello intende raggiungere ed in ultima analisi dal pubblico a cui è destinato.

L'architettura per il servizio prevede due componenti (stateless EJB): quello al livello di service (`PendingTasksServiceBean`) e quello del livello inferiore: business object (`PendingTasksObjectBean`). I servizi (metodi) esposti dai due componenti a cui si è interessati nel presente contesto sono `executeScheduledTasks` (service), `retrieveTasksToExecute` e `executeSingleTask` (object). Come da direttive architetturali, il compito del primo componente è coordinare la gestione del servizio, mentre il componente di livello inferiore ha la responsabilità di incapsulare una serie di servizi comuni a tutti gli oggetti di tipo `Pending Task`.

L'attore `Timer` e più in generale il servizio di schedulazione si presta ad essere fornito da un opportuno sistema esterno in grado di gestire la programmazione dei vari eventi, il relativo coordinamento, la memorizzazione degli esiti, e così via. Uno stimolo dovrebbe corrispondere all'attivazione di un opportuno oggetto che si configuri come client del servizio e che quindi invochi il metodo `executeScheduledTasks` del service bean.

Una volta ricevuto lo stimolo, il service bean si occupa di caricare in memoria tutti i pending task da eseguire (la cui data e tempo di esecuzione sia minore o uguale a quelli attuali e che non siano già stati eseguiti) e quindi di richiederne l'esecuzione. Entrambi i servizi sono forniti dall'EJB al livello di `Object` il quale, a sua volta, delega le operazioni ad un opportuno modulo di interfacciamento al database (DAO – *Data Access Object*, oggetto di accesso ai dati). Brevemente il ricorso a questi oggetti si rende necessario per sopperire ad alcuni inconvenienti (particolarmente pesanti nelle prime versioni dell'architettura EJB) tipici degli Entity EJB.

Per la trasmissione dei dati relativi ai lavori pendenti si utilizza un apposita classe (`PendingTaskVO`) la cui responsabilità primaria è navigare i dati di pertinenza attraverso i vari strati dell'architettura. Oggetti di questo tipo vengono detti Value Object (oggetto valore). Come si può notare esiste una fortissima somiglianza tra la classe `PendingTaskVO` e la rispettiva al livello di modello a oggetti del dominio. Pertanto, il risultato del reperimento dei lavori pendenti dovrebbe consistere in una lista di `PendingTaskVO` generata dal `PendingTaskDAO`, così come il lavoro da eseguire (trasmesso dal componente al livello service a quello di livello inferiore) dovrebbe consistere in un'apposita istanza della classe `PendingTaskVO`.

Nella descrizione di questo scenario, coerentemente con il diagramma precedente, sono stati omessi diversi dettagli. Per esempio, per richiedere l'esecuzione di un determinato servizio offerto da un altro EJB, un EJB deve reperirne l'interfaccia. Ciò si ottiene attraverso una ricerca in un apposito repository di tipo JNDI. Le direttive Sun prevedono di disporre di un'apposita classe atta a fornire tale servizio, denominata tipicamente `Locator`. Ancora il `PendingTaskDAO`, come si vedrà successivamente, non potrebbe da solo assolvere a tutte le proprie responsabilità ma abbisogna di comunicare con altri oggetti, e così via.

L'architettura organizzata in strati offre la possibilità di effettuare una breve riflessione per quanto attiene alla gestione della transazionalità del servizio. La presenza di due EJB garantisce l'opportunità di gestire ogni pending task in maniera atomica: se l'esecuzione di un particolare task fallisce gli unici effetti da annullare (*rollback*) sono quelli generati dal task stesso, e non quelli generati dai task eseguiti in precedenza o da eseguire in futuro. In sostanza il service bean può prevedere un funzionamento che non richieda alcuna transazione (o al più che la supporti), mentre quello al livello di object deve prevedere la creazio-

ne di una nuova transazione ad ogni invocazione dei suoi metodi. Pertanto se l'invocazione del servizio `executeSingleTask` relativa all'esecuzione dell'*ennesimo* task fallisce, questa non influenza le precedenti (già concluse) e quelle future ancora da creare (nel caso in cui non si abbia a che fare con un errore di sistema).

Tipicamente le tipologie di eccezione che possono generarsi appartengono ad una delle due grandi categorie:

- business: si è in presenza di qualche problema o violazione delle regole business;
- system: si è verificato un problema di tipo hardware (problemi di connessione, non disponibilità di un server, e così via).

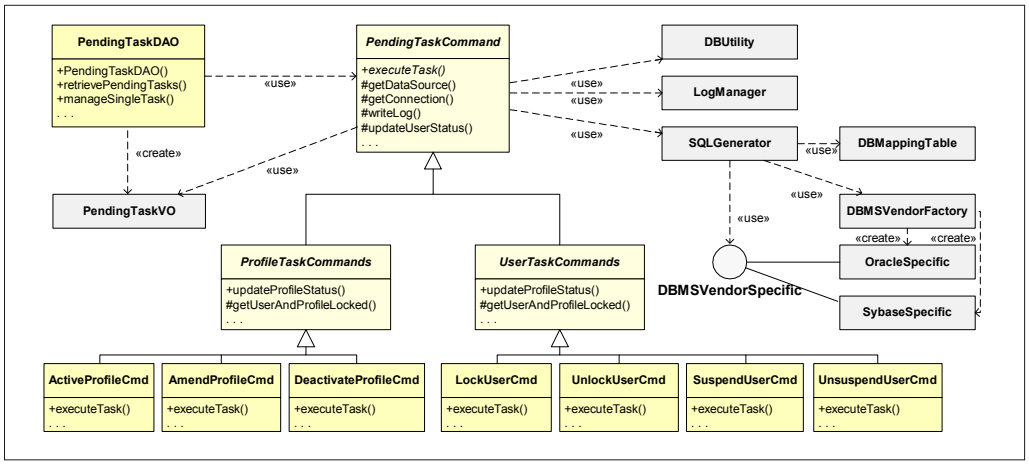
Nel caso si verifichi un'eccezione del primo tipo, può avere senso continuare a eseguire i restanti pending task, mentre nel secondo caso potrebbe non aver senso: verosimilmente lo stesso problema tenderebbe a presentarsi durante l'esecuzione degli altri lavori.

Un'ultima annotazione che potrebbe valer la pena riportare è relativa alla gestione multithreading del servizio. In un'architettura non component-based, uno scenario del genere potrebbe far considerare l'eventualità di eseguire i vari pending task reperiti concorrentemente. Anche se nella maggior parte dei sistemi si ha unicamente una simulazione di concorrenza, comunque politiche di concorrenza offrono il vantaggio di sfruttare tempi di inattività (come per esempio attesa risposte da periferiche). Questa gestione potrebbe ottenersi associando ciascun pending task ad un apposito thread per la relativa gestione. Ora, tecniche di questo tipo sono tipicamente proibite in architetture component based, in quanto la gestione dei thread è interamente demandata al container. Ciò da un lato è vantaggioso: riduce il sistema da dover disegnare, minimizza il rischio di errori di programmazione (come per esempio errata gestione di situazioni di deadlock), ecc., dall'altro è svantaggioso: limita fortemente il grado di libertà del disegnatore.

Una buona alternativa potrebbe ottenersi facendo sì che il componente al livello di service, una volta reperiti i pending task da eseguire, li serializzi e li spedisca a sé stesso. Un opportuno componente si dovrebbe quindi occupare di ricevere questi messaggi e di processarli. In questo modo si otterrebbe una gestione atomica (ogni messaggio ricevuto avvia una transazione a sé stante) ed asincrona dei singoli task. In questo caso si è deciso di non ricorrere a tale soluzione per non rendere incomprensibile un esempio già abbastanza complesso.

Dall'analisi dello use case *Esecuzione lavori pendenti* è possibile notare che ciascun pending task può richiedere una delle seguenti operazioni: rendere operativo un nuovo profilo, attuare gli aggiornamenti relativi a un profilo, disattivare un profilo, sospendere un utente, rimuovere la sospensione di un utente, bloccare e sbloccare un utente. Gli architetti più esperti, analizzando il diagramma del caso d'uso e la relativa descrizione, potranno veder comparire il famoso pattern Command. In sostanza è possibile realizzare

Figura 8.45 — *Porzione di disegno relativa al pattern Command applicato al servizio di esecuzione dei lavori pendenti.*



una gerarchia di classi, ognuna specializzata nell'eseguire uno specifico task (come riportato in fig. 8.45). Queste sono organizzate secondo una struttura gerarchica che prevede il medesimo progenitore (eventualmente un'interfaccia). Questo permette di comandare i vari oggetti in maniera indipendente dalla relativa implementazione come se fossero tutti istanze della medesima classe. Il package della riflessione di Java permette anche di ottenere una creazione elegante e flessibile dei vari oggetti, è sufficiente creare un mapping tra gli indentificatori delle varie tipologie di task e il nome completo delle classi che li gestiscono. Pertanto la classe che si occupa di eseguire gli specifici task deve reperire l'id della relativa tipologia e con questo ottenere il nome della classe da creare dinamicamente a partire dal nome:

```
Class commandClass = Class.forName(classPath);
Object newInstance = commandClass.newInstance();
```

Il modello di fig. 8.45 prevede che l'esecuzione dei lavori pendenti, assegnati dal componente al livello di service a quello di livello inferiore, sia realizzata da quest'ultimo in collaborazione con un'opportuna istanza della classe `PendingTaskDAO`. Questo si occupa di prelevare l'identificatore del lavoro e quindi reperire da un'apposita tabella il nome completo della classe da istanziare. Le possibili alternative sono: `ActiveProfileCmd`, `AmendProfileCmd`, `DeactivateProfileCmd`, `LockUserCmd`, `UnlockUserCmd`, `SuspendUserCmd` e `UnsuspendUserCmd`. Queste istanze, essendo discendenti della classe astratta `PendingTaskCommand`, possono essere considerate e quindi comandate

come istanze di tale classe. Quindi l'oggetto di tipo `PendingTaskDAO`, una volta istanziato lo specifico comando, è in grado di invocarne il metodo `executeTask()`. Da notare che le specifiche del Command prescrivono che nel costruttore sia fornito l'oggetto che contiene i dati che il comando deve conoscere per poter operare. In questo caso si tratta di un'istanza di `PendingTaskVO`.

Nel modello illustrato in fig. 8.45, la presenza della gerarchia intermedia (`ProfileTaskCommands` e `UserTaskCommands`) permette di definire una sola volta il codice utilizzato delle varie classi specializzate.

Sempre dall'analisi del diagramma è possibile notare la presenza di una classe incaricata di generare dinamicamente i comandi SQL (`SQLGenerator`) da utilizzarsi per interagire con il data base management system. La maggior parte di questi comandi sono vendor-independent sebbene talune volte può rendersi necessario ricorrere ad alcune funzioni specifiche dei singoli vendor. Al fine di mantenere la portabilità del sistema, queste funzioni vengono incapsulate in apposite classi (`OracleSpecific`, `SybaseSpecific`, ecc.).

Ancora una volta, il testo riportato nei periodi precedenti si presta a essere rappresentato attraverso un opportuno diagramma di sequenza come quello riportato in fig. 8.46.

Sebbene da un punto di vista teorico un diagramma di interazione (sequenza e collaborazione) sia in grado di illustrare casi/scenari assolutamente generici, nella pratica, non appena l'architettura comincia a superare un certo grado di complessità (presenza di un numero eccessivo di classi, di interfacce/classi astratte…), l'illustrazione del caso generico diviene praticamente impossibile.
Volendo si potrebbe anche tentare di ottenere tale risultato; si correrebbe però il rischio di generare un diagramma così complesso da risultare difficilmente leggibile e praticamente inutile. Il problema viene risolto proponendo uno o più casi particolari. Sebbene si perda di genericità, questi rendono il concetto più facilmente comprensibile e con uno sforzo piuttosto marginale è possibile ricavare il caso generico.

Dall'analisi del diagramma di fig. 8.46 è possibile evidenziare alcune anomalie. In primo luogo è presente un oggetto di tipo `PendingTasksCommand`. Ciò costituisce una macroscopica violazione delle leggi dell'OO (come è possibile avere un'istanza di una classe astratta?). Si tratta invece di una tecnica normale, utilizzata al fine di evidenziare il fatto che gli specifici comandi sono visti dal sistema e quindi pilotati come se fossero effettivamente istanza della classe `PendingTasksCommand`. Anche per quanto concerne gli oggetti relativi agli specifici comandi si è preferito non riportare le singole istanze, bensì semplicemente evidenziare un oggetto generico a mo' di segnaposto. Per terminare, il livello di dettaglio è ancora molto elevato: i componenti

sono mostrati come semplici classi, non sono illustrati i meccanismi che permettono di reperire le varie "istanze" di componenti, non sono presentati gli oggetti che cooperano con gli specifici comandi per ottenere i vari servizi, ecc. Ciò è del tutto legittimo qualora ci si trovi nella fase del disegno di specifica, mentre lo sarebbe molto meno nel caso in cui si trattasse della versione di implementazione.

Chiaramente un modello di disegno non è costituito unicamente dalla proiezione statica (diagramma delle classi), bensì è necessario specificare come, a tempo di esecuzione, i vari oggetti dialogano tra loro per fornire i vari servizi. In altre parole è necessario fornire la proiezione dinamica.

CRC Cards

Presentazione

Una metodologia ampiamente utilizzata (soprattutto in passato) nel processo di progettazione e documentazione di modelli a oggetti è quella nota con il nome *Class – Responsibility – Collaborator Cards* (carte di classe – responsabilità – collaboratrici) o più semplicemente CRC Cards, il cui dominio principale di applicazione è costituito dai modelli a oggetti di analisi e disegno.

Figura 8.46 — *Diagramma di sequenza relativo allo scenario principale del servizio di esecuzione dei lavori pendenti.*

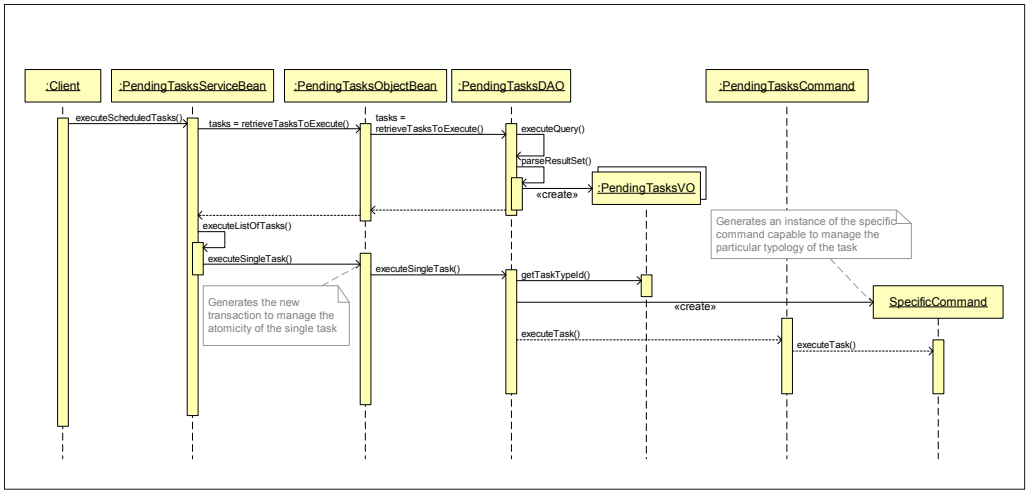
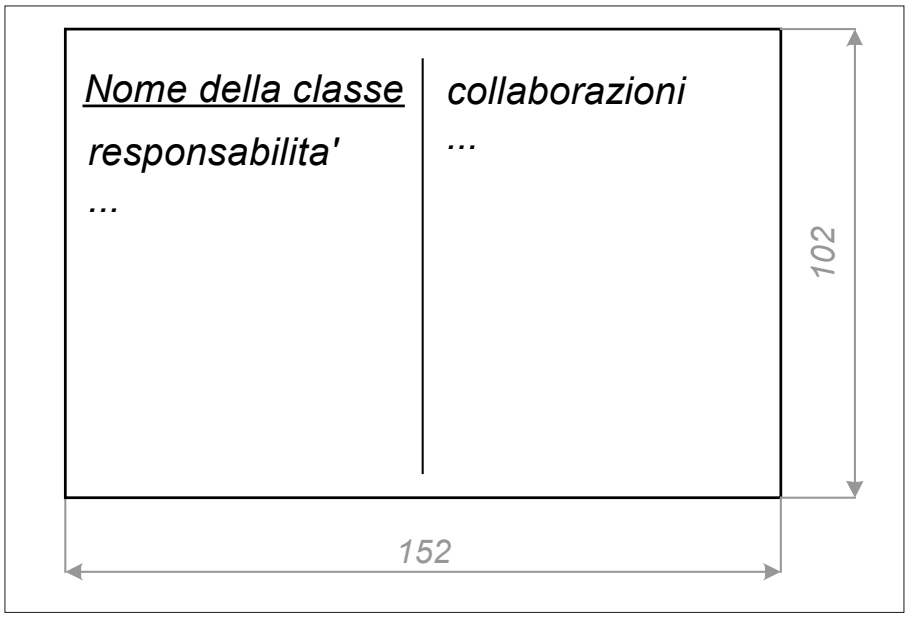


Figura 8.47 — *Formato delle schede CRC.*



Il nome della metodologia descrive perfettamente le caratteristiche su cui si focalizza l'attenzione. Brevemente il termine carte si riferisce ai moduli sui quali riportare le varie informazioni. Le direttive classiche ne specificano il formato e addirittura le dimensioni: 6" × 4" (152 × 102 mm). Queste sono costituite da foglietti divisi nelle tre zone: classe, responsabilità e collaborazione.

Il termine *responsabilità* si riferisce alle informazioni che una classe possiede e i servizi che eroga, sebbene, tipicamente, è sufficiente specificare i servizi forniti (in altre parole si fa riferimento alle caratteristiche strutturali e comportamentali delle classi). Non è infrequente che una classe debba farsi carico di specifiche responsabilità per le quali però non possieda parte o tutte le informazioni necessarie. In tal caso nasce l'esigenza di delegare l'espletamento di parte dei propri servizi ad altre schede (necessita di *collaborare* con altre classi). Per esempio una scheda identificante uno specifico conto corrente potrebbe annoverare tra le proprie responsabilità la fornitura del servizio di generazione dell'elenco dei movimenti mensili. Sebbene la scheda debba avere conoscenza della presenza di tale lista, principalmente per questioni di coesione, non dovrebbe avere conoscenza dei dettagli della relativa gestione. Si genera quindi l'esigenza di creare una collaborazione con una specifica scheda derogata alla gestione della lista. Ciò permetterebbe alla prima scheda di ottenere, dai singoli movimenti, le varie informazioni necessarie (*data*, *destinatario o sorgente*, *importo*, *descrizione*, ecc.).

La versione iniziale della metodologia fu congegnata da Ward Cunningham con il nome di HyperCard presso i laboratori della Apple Computer Inc. negli anni Ottanta. Inizialmente fu elaborata per risolvere problematiche legate alla necessità di documentare le decisioni prese nella fase di disegno, con particolare riferimento alle dinamiche delle collaborazioni tra le classi. Nell'arco del tempo, la metodologia CRC Card ha subito profondi cambiamenti specie con l'avvento di linguaggi di modellazione come lo UML. Quantunque quest'ultimo non sia assolutamente una metodologia, gli strumenti forniti (essenzialmente diagrammi delle classi e di interazione) hanno finito per modificare notevolmente l'applicazione della metodologia CRC che comunque mantiene intatta la propria validità sia come tecnica di brainstorming, sia come strumento di insegnamento e documentazione. Da diversi tecnici è considerata come una valida alternativa alla *Robustness Analysis*. In tal caso, viene utilizzato per produrre modelli di analisi "usa e getta". Così come nel caso dei modello prodotti con al tecnica della Robustness Analysis, una volta realizzate le versioni iniziali del modello di disegno, tipicamente, non è più giustificabile l'investimento di tempo e denaro necessario per mantenere aggiornare le varie schede.

Le CRC Cards si sono dimostrate molto utili anche come strumento di documentazione del codice (per molti autori questo è il ruolo cui sono state relegate dall'avvento delle nuove tecnologie). Tool quali JavaDoc possono agevolare questo tipo di utilizzo. È sufficiente infatti aggiungere, nelle sezioni di intestazione del codice delle classi (header), informazioni supplementari simili a quelle richieste da una scheda CRC (nome della classe, responsabilità, collaborazioni e così via) e quindi esportare tali informazioni in opportuni documenti e/o pagine HTML. Il modo migliore per applicare la metodologia consiste nel cominciare a stampare tutta una serie di schede vuote (come quelle riportate nella fig. 8.47).

Spesso l'insieme delle informazioni standard viene integrato con dati aggiuntivi, come per esempio se si tratta di una classe attiva o meno, se deve persistere, e così via. Ciò non sempre è una strategia vantaggiosa. Per esempio, qualora si decidesse di utilizzare le CRC Cards per il brainstorming iniziale, probabilmente sarebbe conveniente utilizzarle nella forma essenziale senza aggiungervi ulteriori informazioni: si tratta di una fase altamente "volatile" in cui le modifiche sono molto frequenti e tipicamente di notevole impatto. Non è quindi il caso di perdere tempo a specificare informazioni supplementari che, verosimilmente, subiranno continue variazioni. In altri ambiti, per esempio quando si utilizzano a scopi documentativi, è possibile valutare l'opportunità di aggiungere ulteriori informazioni.

Una volta preparate le varie schede vuote è molto utile disporre di una lavagna bianca magnetica con dei piccoli magneti. Si è quindi pronti a cominciare. Un passo propedeutico

consiste nel selezionare gli use case/scenari di cui si intende disegnare il sottosistema atto ad erogarne le funzionalità. Tipicamente si preferisce selezionare quelli che intuitivamente coinvolgeranno un numero maggiore di oggetti e/o realizzano funzionalità particolarmente importanti.

Il "gioco" inizia scorrendo la lista delle azioni che compongono lo scenario. Qualora si incontrino azioni relative a servizi forniti da "schede" individuate in precedenza, o più semplicemente riferimenti ad esse, si posiziona la relativa card sulla lavagna con un apposito magnete e con un pennarello si tracciano le relazioni con le altre schede. Le schede rappresentano sia entità (e quindi classi appartenenti al modello a oggetti del dominio), sia "controller" di determinate funzionalità.

Nel caso in cui nessuna scheda tra quelle individuate in precedenza risulti idonea a gestire il passo dello scenario preso in esame, è necessario iniziare a compilare una scheda vuota relativa alla nuova classe individuata, riportando nella sezione relativa al nome una prima ipotesi di denominazione.

Ogni qualvolta si inizia a leggere un nuovo scenario, le prime carte da considerare tendono ad essere le stesse (coerentemente a quanto visto anche per il modello di analisi): la prima rappresenta un'istanza dell'attore che fornisce lo stimolo iniziale al caso d'uso, la seconda rappresenta l'interfaccia, ossia la classe le cui istanze ricevono gli stimoli forniti dagli attori, ne analizzano i dati, invocano i servizi necessari, ecc., la terza la classe rappresenta la gestione dell'interfaccia utente, ecc. Anche in questo caso tenere a mente l'organizzazione dell'architettura del sistema semplifica notevolmente il processo di individuazione delle classi.

Il processo è intrinsecamente iterativo ed incrementale: man mano che si procede con l'analizzare nuovi scenari si identificano nuove classi e si perfezionano quelle individuate in precedenza. Gli aggiornamenti delle schede si rendono necessari al verificarsi di diverse situazioni: l'attività considerata, sebbene concettualmente appartenente ai servizi forniti da una specifica classe individuata in precedenza (criterio di coesione), non è ancora contemplata nella relativa lista delle responsabilità, oppure una scheda, al fine di espletare nuove responsabilità, necessita di utilizzare i servizi di altre non ancora inserite nell'elenco delle collaborazioni, e così via.

Terminata l'analisi di uno scenario, è importante effettuare tutti i controlli del caso sia al livello delle singole classi che lo costituiscono, sia a livello più generale del disegno. Il processo di revisione aiuta ad individuare e risolvere possibili anomalie come per esempio presenza di diverse classi con responsabilità molto simili che eventualmente vanno inglobate in una sola più generale, oppure carte poco utili e quindi eventualmente di cui fare a meno, e così via.

Qualora ci si dovesse rendere conto che alcune schede siano superflue, invece di gettarle via, è consigliabile porle in un apposito contenitore: ciò evita, a seguito di ripensamenti, di dover riscrivere daccapo le card ricordandone tutte le responsabilità e collaborazioni o, peggio ancora, di dover condurre una ricerca nel cestino della spazzatura dell'ufficio (la voce dell'esperienza). Altri controlli sono relativi all'analisi delle responsabilità delle varie schede. Qualora si evidenzi il caso di classi con troppe responsabilità (tipico sintomo di classi a scarsa coesione) è necessario valutare se siano stati inglobati impropriamente diversi concetti ed eventualmente suddividere tali responsabilità tra più classi ad elevato grado di coesione. Altro controllo, esatto contrario del precedente, consiste nel verificare se esistono classi eccessivamente "deresponsabilizzate". Anche se è tendenza naturale dell'OO creare una miriade di oggetti molto leggeri, non è opportuno distribuire tra vari oggetti responsabilità fortemente correlate tra loro (aumento ingiustificato dell'accoppiamento).

Per quanto concerne l'analisi del modello prodotto, andrebbe eseguita sia nel contesto del caso d'uso oggetto di studio, sia in quello più generale dell'intero modello. Per il primo punto un buon approccio di verifica consiste nel variare le assunzioni iniziali dello stesso, magari ipotizzando diversi parametri di input, diversi stati di partenza del sistema, ecc. L'obiettivo consiste nel verificare che il modello individuato mantenga la propria validità con diversi scenari. In caso negativo, chiaramente, è necessario rivedere il flusso delle carte. Una verifica più creativa consiste nel tentare diversi approcci al problema ("tracciare" percorsi differenti). Spesso può capitare che percorsi inizialmente considerati non molto probabili possano poi rivelarsi soluzioni migliori. L'analisi del nuovo diagramma, nel contesto dell'intero modello di disegno, permette di verificare la consistenza, ossia l'applicazione dei principi base, il rispetto della filosofia adottata, ecc. Ogni volta sia necessario prendere delle decisioni (attività molto ricorrente in questa fase) è necessario che queste non siano limitate unicamente a considerazioni locali al caso d'uso oggetto di studio, bensì è opportuno che tengano conto degli aspetti più generali del sistema (in fondo anche per questi motivi si producono i vari modelli).

Durante questo processo di revisione spesso si dà luogo ad un refactoring, introducendo opportuni oggetti e/o inglobandone altri in uno solo, al fine di migliorare l'organizzazione del disegno, rendere i modelli più stabili, riflettere la struttura a strati delle moderne architetture, e così via.

Una volta raggiunto uno stadio in cui si abbia la confidenza di aver elaborato una buona ipotesi di disegno, è possibile effettuare una verifica formale del comportamento dinamico del modello stesso. Ciò può avvenire tracciando opportuni diagrammi di interazione. Ancora una volta non è necessario essere assolutamente formali; disponendo delle carte, si può utilizzare un approccio decisamente pragmatico, ma non per questo meno rigoroso, per il disegno dei diagrammi di sequenza. Si inizia collocando le carte individuate in alto sulla lavagna lungo un'ipotetica linea orizzontale le une a fianco delle altre, si tracciano le

varie linee normali che rappresentano il trascorrere del tempo e quindi si disegnano le varie interazioni (scambi di messaggi tra i vari oggetti).

Da notare che l'applicazione di questa metodologia porta ad individuare esclusivamente le classi di cui esista effettiva necessità. Questo è positivo per alcuni aspetti (si disegna solo lo stretto necessario, si risparmia tempo, si evitano problemi di over-engineering, si producono modelli più semplici, ecc.), ma negativo per altri (minore flessibilità, maggiori problemi di espansione dello stesso, ecc).

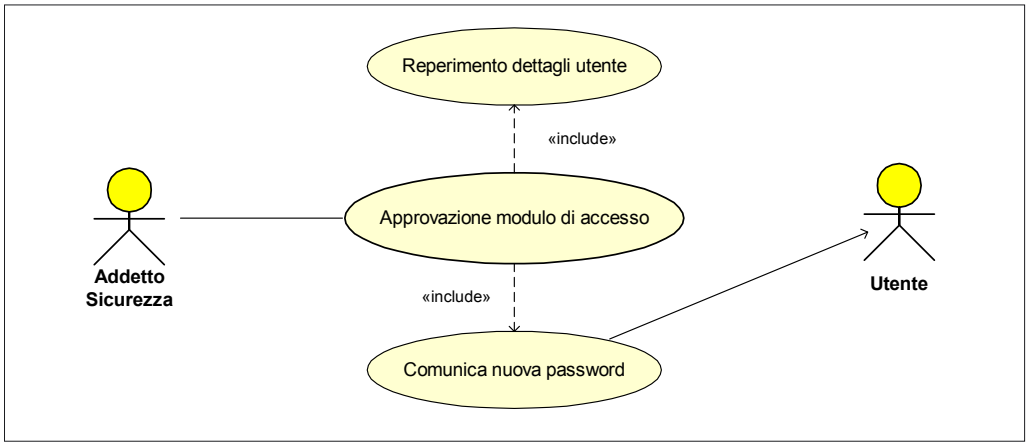
Indipendentemente dalla tecnologia utilizzata, il processo di disegno del sistema richiede di prendere molte decisioni (probabilmente il numero delle scelte da operare in questa fase è superiore a quello di tutte le altre). È molto importante tenere traccia di quanto stabilito, soprattutto per agevolare la manutenibilità del sistema stesso: quando si dovranno disegnare le soluzioni per i nuovi requisiti, la documentazione delle decisioni permette di ricordare (studiare) le scelte che hanno dato luogo allo specifico disegno evitando di percorrere strade già tentate e dimostratesi non soddisfacenti, di realizzare parti del disegno che violano la filosofia del sistema, ecc.

Un'evoluzione della metodologia, consiste nel disporre di carte del "potere" (*power card*), ossia carte rappresentanti soluzioni generiche, eleganti, dimostratesi valide nella soluzione di altri problemi, ecc. in poche parole carte rappresentanti pattern. Per esempio si potrebbero preparare un certo numero di carte con i pattern più diffusi (composite, adapter, commander, proxy, factory, ecc.) da estrarre a mo' di jolly ogni qualvolta se ne ravvisi la necessità.

L'applicazione della metodologia classica prevede che al gioco delle carte (come da buona tradizione) prendano parte diverse figure professionali. Per esempio è possibile coinvolgere un business analyst da interpellare qualora i casi d'uso non siano chiari o si individuino delle incoerenze/lacune, uno o più architetti, programmatori, un esperto della metodologia delle CRC Cards (*CRC Card Facilitator*), ecc. Spesso il team viene corredato da uno o più "scrivani", il cui obiettivo è di compilare le varie schede man mano che i vari elementi sono evidenziati. Alcuni tecnici poi suggeriscono di annettere anche degli osservatori (come per esempio tecnici junior a scopi formativi, alcuni elementi del team che si dovranno occupare dell'implementazione del sistema per renderli consapevoli, il prima possibile, dell'evoluzione dei requisiti utente, e così via). Sebbene ogni persona possa fornire ottimi spunti alla realizzazione del disegno, è necessario tenere presente che riunioni con troppe persone tendono a generare un elevato grado di entropia.

Alcuni tecnici, particolarmente fautori della metodologia delle CRC, si spingono fino al punto di consigliarne l'utilizzo fin dai primi colloqui con gli utenti del sistema. Ciò do-

Figura 8.48 — *Caso d'uso Generazione nuova password.*



vrebbe permettere di produrre contestualmente i casi d'uso ed il modello di analisi, in altre parole si tenta di prendere i famosi due piccioni con una sola fava… L'esperienza insegna che molto spesso si finisce per prendere due fave con un solo piccione! Qualora si convochino troppe persone in una sola riunione cercando anche di definire i casi d'uso, si finisce molto facilmente per generare una grossa confusione che raramente porta a produrre qualcosa di sensato.

Esempio

Come esempio si è deciso di presentare il servizio che permette agli addetti alla sicurezza di far generare al sistema una nuova password per uno specifico utente. Questo servizio si rende necessario poiché gli utenti possono dimenticare la propria password, oppure possono venire bloccati dal sistema a seguito di tre tentativi fallimentari di impostazione della parola chiave durante la fase di log-in nel sistema, e così via.

La metodologia delle carte CRC può essere utilizzata per rappresentare diversi manufatti: in tutti i casi in cui si ha a che fare con un modello a oggetti. Nel contesto di questo capitolo si è deciso di mostrarne un utilizzo per un modello di analisi, in quanto considerato un giusto compromesso: mostra diversi dettagli ma non in quantità elevata come quelli richiesti da un modello di disegno.

> Nelle carte riportate di seguito, spesso sono presenti dei punti di sospensione nella sezione *Collaborazioni*. È un artificio utilizzato per indicare che, sebbene la card illustrata collabori con altre per espletare ulteriori servizi, questi esulano dall'interesse del presente paragrafo. Le dimensioni delle card non rispettano il formato standard per questioni di impaginazione.

Card 1 — *ResetPasswordPage.*

ResetPasswordPage	**Collaborazioni** - ResetPasswordHandler
Responabilità – Realizzare l'interfaccia utente necessaria al servizio reimpostazione password. – effettuare i controlli preliminari sui dati impostati dall'utente. *(Tipologia: boundary.)*	

Card 2 — *ResetPasswordHandler*

ResetPasswordHandler	**Collaborazioni** - ResetPasswordPage - SelezioneUtenteHandler - PasswordManager - NotificheManager
Responsabilità – Coordinare il servizio reimpostazione nuova password: riceve le richieste dalla GUI, invoca i necessari servizi e collabora con *ResetPasswordPage* per visualizzare i vari risultati. *(Tipologia: control.)*	

Card 3 — *SelezioneUtenteHandler.*

SelezioneUtenteHandler	**Collaborazioni** - SelezionеUtentePage - ResetPasswordHandler - UtentiManager
Responsabilità - coordinare il servizio relativo alla selezione di un utente: riceve le richieste dalla GUI, invoca i necessari servizi e con la collaborazione di *SelezioneUtentePage* mostra i risultati. *(Tipologia: control.)*	

Card 4 — *SelezioneUtentePage.*

SelezioneUtentePage	**Collaborazioni** - SelezioneUtenteManager
Responsabilità - Realizzare l'interfaccia utente necessaria alla funzione selezione utenti. - effettuare i controlli preliminari sui dati impostati dall'utente. *(Tipologia: boundary.)*	

Card 5 — *UtentiManager.*

UtentiManager	**Collaborazioni** - SelezioneUtenteHandler - Utente - …
Responsabilità - gestire tutte le operazioni relative ai dati utente (selezione singolo e multiplo utente, inserimento, aggiornamento, eliminazione, ecc.) *(Tipologia: control.)*	

Card 6 — *Utente.*

Utente	**Collaborazioni**
Responsabilità - memorizzare i dati relativi ad un singolo utente (corrisponde orientativamente ad una n-pla della tabelle utenti). *(Tipologia: entity.)*	

Card 7 — *PasswordManager.*

PasswordManager	**Collaborazioni** - ResetPasswordHandler - Password - …
Responsabilità - gestire tutte le operazioni relative ai dati delle password (selezione password, inserimento, aggiornamento, eliminazione, ecc.) *(Tipologia: control.)*	

Card 8 — *Password.*

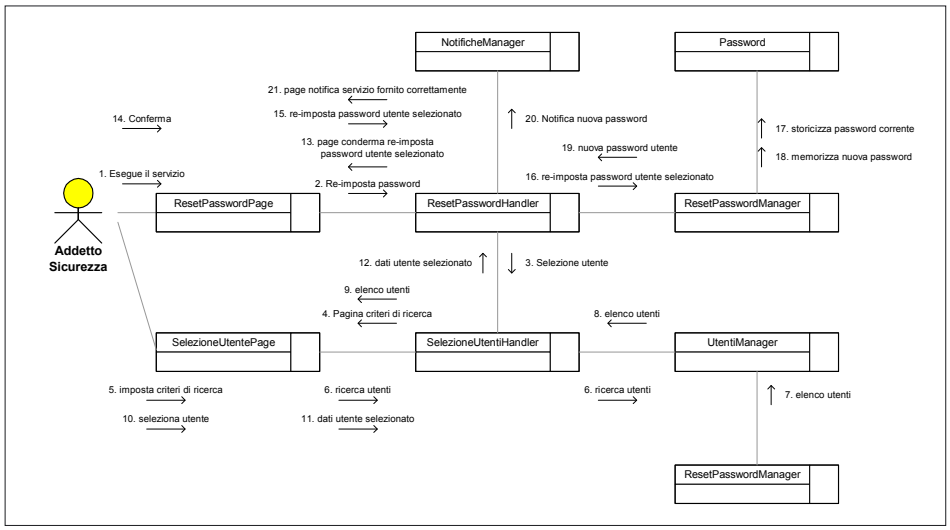
Password	**Collaborazioni**
Responsabilità - memorizzare i dati relativi ad una singola password. (Corrisponde orientativamente ad una tupla della tabelle password) *(Tipologia: entity).*	

Card 9 — *NotificheManager.*

NotificheManager	**Collaborazioni** - …
Responsabilità - Fornire il servizio di preparazione e notifica e-mail (eventualmente crittografate) agli utenti. *(Tipologia: control.)*	

Dall'analisi delle varie carte è possibile constatare che il modello realizzato è completamente consistente con i precedenti modelli di analisi rappresentati attraverso lo UML. Le differenze sono che le CRC cards richiedono molto più spazio e i modelli sono decisamente meno intuitivi, però offrono il vantaggio che impongono di specificare un maggior dettaglio di informazioni (dichiarazione esplicita delle responsabilità). Ciò però è ottenibile anche attraverso lo UML sia attraverso la dichiarazione dei metodi delle classi, sia prevedendo una sezione esplicita (magari visualizzabile su richiesta) in cui riportare le responsabilità.

Figura 8.49 — *Modello reimposta password ottenuto attraverso l'utilizzo delle CRC Card.*



Una volta realizzate le varie carte bisognerebbe mostrarne la relativa collaborazione. Ora supponendo di disporre di una lavagna con i magneti o di una scrivania sufficientemente grande, il risultato che si otterrebbe è simile a quello di fig. 8.49 che, guarda caso, somiglia tantissimo a un diagramma di collaborazione.

Quando un modello di disegno può essere considerato ben progettato?

Questo capitolo si conclude con una parte dedicata al tentativo di rispondere a quello che è uno degli interrogativi più controversi e complicati dell'intero processo di sviluppo del software: quando un modello di disegno OO può essere considerato tale e ben progettato? Si tratta di un interrogativo cui, oggettivamente, è difficile fornire una risposta completa ed univoca, probabilmente si tratta di un vero e proprio argomento da salotto o meglio ancora simposio. Ancora una volta, gli architetti esperti sono in grado di rendersi conto del livello di qualità di un modello fin dalle primissime analisi visive dello stesso. Purtroppo, si tratta di capacità soggettive che si sviluppano sia con l'esperienza nel disegno di sistemi che funzionano, sia con lo studio dei testi sacri dell'analisi e disegno OO, ma che difficilmente si prestano ad essere condensate in un trattato organico. Ciò nonostante, in queste pagine si è cercato di fare proprio ciò: fornire una serie di criteri che permettano di evidenziare errori tipici durante il disegno del sistema, di verificare la qualità degli stessi modelli e di produrne versioni di livello qualitativo elevato. Questo però, senza avere assolutamente la presunzione di essere esaustivi.

Nel processo di produzione e mantenimento del modello di disegno è purtroppo spesso possibile assistere ad uno sviluppo abbastanza tipico. Le prime versioni del modello di disegno rappresentano il paradiso dell'architetto: tutto appare bello, elegante, incontaminato, flessibile, e così via. Questa iniziale visione paradisiaca in genere non riesce a sopravvivere a tutte le fasi dello sviluppo. Durante il processo si assiste ad una sua graduale corruzione che può raggiungere livelli tali da far degenerare catastroficamente l'intera architettura. Uno dei motivi principali della perdita di qualità del modello di disegno è imputabile al cambiamento dei requisiti, che magari avviene in una direzione non prevista dalla logica di funzionamento del sistema. Altre volte invece è generata da motivi piuttosto banali, quali cambiamenti effettuati troppo rapidamente senza un'opportuna mini-iterazione: revisione delle fasi di analisi e disegno, oppure variazioni realizzate da personale non familiare con il disegno del sistema e/o con i principi base (a tal proposito, l'esperienza consiglia di pianificare attentamente le proprie ferie qualora si abbia la responsabilità dell'architettura del sistema), ecc.

Puntare l'indice accusatore sui clienti e relativi cambiamenti dei requisiti è la soluzione più facile, il capro espiatorio sempre valido, spiegare poi a persone intelligenti come il cambiamento dei requisiti possa aver fatto irreparabilmente degenerare il disegno è cosa meno facile. Con ciò non si vuole dire che cambiamenti, anche drastici, dei requisiti siano senza effetti, o che non posseggano la potenzialità di compromettere il disegno, né tanto meno che vadano assorbiti in poche ore o giorni di lavoro. Però se il disegno tende a distruggersi a ogni modifica dei requisiti, probabilmente la sua qualità non è eccelsa.

Prima di addentrarsi nell'esame di una serie di principi utili e linee guida per la realizzazione di modelli di disegno di buona qualità, è interessante presentare alcuni sintomi che permettono di evidenziare situazioni critiche, in cui bisogna necessariamente intervenire (magari con un refactoring a livello di disegno) prima che la qualità del sistema risulti irreparabilmente compromessa. L'analisi delle situazioni di crisi è molto utile anche perché aiuta a comprendere più intimamente la necessità di realizzare modelli di disegno di qualità. Se da un lato stabilire che un modello di disegno sia ben progettato è un'attività piuttosto difficile, dall'altra verificarne la pessima qualità è molto facile.

Sintomi di una cattiva architettura

Uno dei sintomi di malessere dell'architettura più semplici da individuare è la cosiddetta **fragilità**. Con tale termine si fa riferimento alla tendenza dei sistemi di generare malfunzionamenti di sue parti a seguito della realizzazione di aggiornamenti. Ciò che spesso avviene è che a seguito di alcune modifiche apportate in specifici settori del sistema si assiste alla perdita di funzionalità di altre parti apparentemente non relazionate con i settori modificati. È come dire che in un appartamento si modifica il sistema di antifurto ed improvvisamente non funziona più il sistema di riscaldamento.

Il livello di qualità del sistema è avvertito intuitivamente dal personale tecnico coinvolto nello sviluppo dello stesso. L'evidenza più palese che il sistema non sia di elevata qualità è fornita proprio dal loro comportamento. Quando il sistema è fragile si assiste ad una rincorsa alle scuse e delle motivazioni tecniche più impensabili pur di evitare l'approvazione di nuove funzionalità o la rielaborazione di quelle esistenti. Il tutto è peggiorato poi da una caratteristica intrinseca della fragilità del sistema ossia auto-rigenerazione: ogni iterazione e ogni ondata di cambiamenti tende a rendere un sistema fragile ancora più fragile, ad introdurre nuovi problemi e a degradarne il controllo.

Una volta individuata una condizione del genere è opportuno inserire nel processo di sviluppo del software una o più iterazioni (a seconda della quantità di entropia generata) volte a rivedere l'architettura senza aggiungere nuove funzionalità. Altamente sconsigliato è invece procedere disseminando artifici temporanei — leggasi toppe — nella speranza che alla fine il tutto magicamente funzioni. Architetture di questo tipo tendono a sprofondare all'occorrenza dei primi problemi e/o richieste di far evolvere il sistema.

Un'anomalia strettamente connessa con la precedente è quella che va sotto il nome di **rigidità**; con tale termine si indica la resistenza del sistema ai cambiamenti. Chiaramente se ogni cambiamento corre il rischio di distruggere il sistema in parti, questo è fragile e rigido. Come nel caso precedente, ogni aggiornamento apportato al sistema, anche quelli che apparentemente non nascondono alcuna insidia, generano tutta una rincorsa alla stabilizzazione dello stesso. Situazioni del genere, tipicamente, sono evidenziate da periodi prolungati necessari per rendere il sistema stabile.

Un'altra anomalia è nota con il nome di **immobilità**; si tratta del contrario della riusabilità. Evidenzia situazioni in cui risulta difficile riutilizzare parti dello stesso sistema o di altri realizzati in precedenza. Spesso accade di dover realizzare componenti o comunque parti del sistema con funzionalità simili, o addirittura uguali, a quelli implementati per un altro in precedenza e di tentare, legittimamente, di riutilizzare le stesse parti di codice. Tutto ciò, abbastanza naturale dal punto di vista teorico, può risultare molto complicato in pratica. Ciò perché il processo di estrapolazione delle parti da riutilizzare è così lungo, complesso e rischioso (eccessive interdipendenze) che in sostanza, per ridurre il tempo necessario ed il fattore di rischio, diviene più conveniente dar luogo alla riscrittura delle varie parti.

Un ulteriore problema è legato all'evoluzione incoerente del disegno che finisce per rendere lo stesso rigido e, frequentemente, fragile. Quando un architetto inizia a disegnare il sistema o sue parti, utilizza determinati principi, fissa delle colonne portanti dell'archi-

tettura del sistema, e così via: tutto appare perfetto. A seguito poi di aggiunta di nuove funzionalità e/o correzioni, può accadere che le nuove parti violino i princìpi base e/o le regole stabilite in partenza dall'architetto (è pur vero che ci sono architetti che anche quando costruiscono nella propria mente soluzioni riescono a farlo in maniera errata).

Questi problemi sono particolarmente frequenti quando vi è un avvicendamento dell'architetto oppure durante una sua assenza. Il problema di fondo risiede nel fatto che una stessa modifica si presta ad essere ottenuta adottando molteplici approcci. Spesso diversi di questi risultano legittimi, ma non tutti coerenti con le scelte di base, mentre alcuni andrebbero assolutamente evitati (si tratta di soluzioni a corto respiro inserite al volo e bollate come "temporaneamente permanenti"). Questa tipologia di soluzioni può risultare diabolicamente attraente: tende ad apparire più facile e, solo nell'immediato, più veloce da implementare. Ciò fornisce un'attrattiva molto allettante. Sfortunatamente, l'introduzione di soluzioni *temporanee* di questo tipo, finisce per rendere il sistema rigido e fragile. Si tratta di soluzioni a corto respiro, tipicamente non flessibili, da dover risistemare nell'immediato futuro e pronte ad evidenziare tutte le loro debolezze ai primi cambiamenti. Altre volte può capitare che sia veramente necessario adottare delle soluzioni contrarie alla filosofia di base del sistema o di suoi moduli. In questi casi è forse opportuno riesaminare le scelte di fondo e rivedere il disegno prima di lanciarsi nell'attuazione di soluzioni a corto respiro.

Di anomalie se ne potrebbero individuare diverse, sebbene queste siano quelle più ricorrenti. Qualora dovesse capitare di trovarsi di fronte ad un sistema ormai irrimediabilmente corrotto è sempre possibile citare una serie di giustificazioni come: i requisiti continuavano a cambiare, il tempo a disposizione non era sufficiente, ed altre più o meno risibili.

Criteri

Buona qualità degli elementi

Il primo criterio da considerare scaturisce da un ragionamento logico abbastanza semplice: se le singole parti componenti di un sistema sono di buona qualità, allora verosimilmente è più facile che l'intero sistema sia di qualità. Chiaramente sono possibili deviazioni in entrambe le direzioni: alcuni componenti non perfettamente disegnati e sistemi ancora di buon livello qualitativo e viceversa componenti ben disegnati e sistema scadente. Un buon paragone può essere attinto dalla gastronomia. Alcune volte pur disponendo di ingredienti di primissima qualità è possibile cucinare una pietanza orribile, e caso inverso, pur non disponendo di tutti gli ingredienti migliori si riesce comunque a cucinare portate edibili.

Gli elementi base su cui si focalizza l'attenzione nei modelli di disegno sono: le classi, i package ed i componenti. I criteri per verificare la qualità di tali elementi sono stati abbon-

dantemente analizzati nel paragrafo "Classi ben disegnate" del Capitolo 6 (massima coesione, minimo accoppiamento, riusabilità, ecc.).

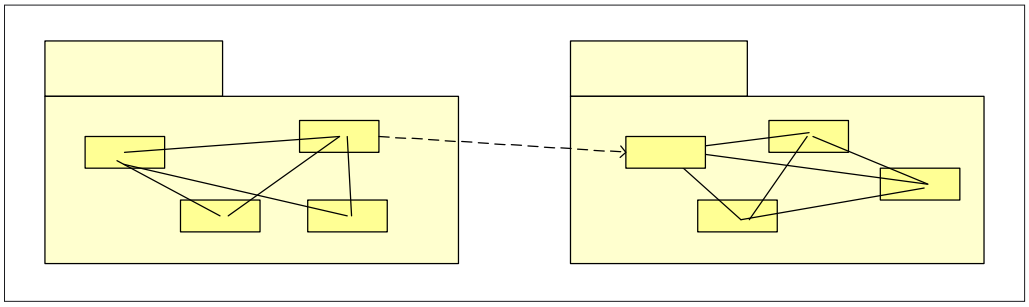
Non bisogna essere eccessivamente rigidi su questo principio: spesso la qualità del disegno di alcune classi viene intenzionalmente sacrificata a favore della qualità del sistema generale. Un esempio evidente (se ne potrebbero trovare diversi altri) è fornito dall'applicazione del design pattern Command [BIB04]. Si tratta di un disegno molto potente, elegante, versatile (un esempio di applicazione è presentato nella fig. 8.45), ma se si analizzassero, utilizzando criteri prettamente accademici, le classi che specializzano quella generale, si potrebbe notare come queste siano create intorno ad una singola funzione e non ad un vero e proprio "oggetto". Ciò non dovrebbe costituire di per sé un buon principio OO, eppure si tratta, nel complesso, di un'eccellente soluzione.

Chiaramente, qualora si tratti di qualche classe non perfettamente disegnata, è possibile sorvolare, quando invece gran parte o tutte le classi sono disegnate male, o alcune sono veramente pessime, allora è inutile procedere nell'analisi dei rimanenti criteri: il modello non va (per quanto si possa essere maghi dei fornelli, con tanti ingredienti marci ci si fa ben poco).

Un altro elemento cui spesso non si attribuisce la dovuta importanza è l'organizzazione in package del sistema. Ciò è molto importante poiché, tipicamente, il riutilizzo delle parti di rado avviene a livello delle singole classi: in genere si cerca di riutilizzare interi package. Errate organizzazione producono svariati problemi: difficoltà di verificare (testare) le singole parti, immobilità del sistema (difficoltà sia di riutilizzarne delle parti, sia di incorporarne altre provenienti da sistemi sviluppati in precedenza), difficoltà di manutenzione, e così via.

Un buon criterio da seguire per strutturare i package deriva dai principi classici: massima coesione e minimo accoppiamento. Le classi inserite nello stesso package devo risulta-

Figura 8.50 — *Esempio di organizzazione in package di una parte del sistema in cui l'accoppiamento interno non è minimo, però lo è quello tra package.*



re "coese" tra loro ed avere un accoppiamento tale da rendere il package stesso scarsamente accoppiato con i restanti. In altre parole si accetta che le classi presenti in uno stesso package possano presentare un livello non minimo di accoppiamento, purché sia invece minimo quello con le classi appartenenti a un altro package (fig. 8.50). Questo criterio assicura, tra l'altro, che i cambiamenti apportati a un package tendano a essere circoscritti allo stesso.

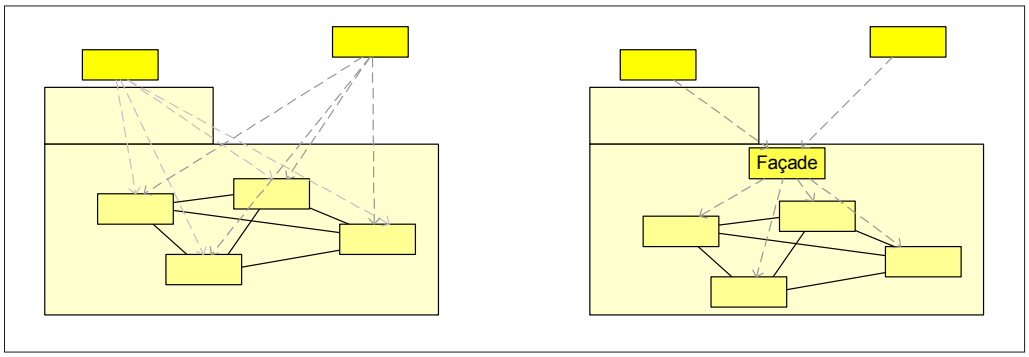
Chiaramente prima di accettare un livello di accoppiamento come quello mostrato in fig. 8.50 bisognerebbe verificare attentamente se ciò sia veramente necessario o se sia esclusivamente conseguenza di un disegno non accurato.

Qualora ci si trovi nella prima ipotesi (forte accoppiamento non riducibile), una buona norma è utilizzare il pattern denominato Façade [BIB04]. Non si tratta di un design pattern con struttura e dinamiche ben definite, è più che altro una norma di buon disegno. L'obiettivo di questo pattern consiste nell'evitare i problemi tipici generati da un forte accoppiamento degli elementi interni ad un package: difficoltà di utilizzo da parte di classi clienti (devono essere consapevoli dell'organizzazione interna), ripetizione del codice necessario all'utilizzo dei servizi esposti dal package nelle varie classi client, e così via. In poche parole il sistema corre il rischio di diventare fragile (è facile commettere errori nell'utilizzo dei servizi, una modifica alla struttura del package si ripercuote su classi clienti, ecc.). Al fine di evitare questi ed altri problemi si ricorre all'utilizzo del pattern Façade, il quale prescrive di introdurre una classe di interfacciamento (una "facciata" appunto, fig. 8.51), che si configuri come unico "punto di accesso" al package, nascondendone i dettagli interni. Ciò permette di evitare che le classi clienti siano consapevoli della struttura interna del package, evita di dover specificare la logica di utilizzo dei relativi servizi in svariate classi clienti e quindi di doverle modificare come conseguenza dell'aggiornamento della struttura del package, ne semplifica l'utilizzo ecc.

Sempre nell'organizzazione del codice in package è molto importante evitare dipendenze cicliche: il `PackageC` dipende dal `PackageB` che a sua volta dipende dal `PackageA` che dipende dal `PackageC`. Questo scenario è da evitare per una serie di motivi: tende ad aumentare l'area in cui eventuali cambiamenti fanno risentire i propri effetti, rende più difficile verificare il sistema, limita la riusabilità del codice, ecc.

Pertanto in tutti i casi in cui sia presente una dipendenza ciclica dei package è necessario andare a interrompere il ciclo vizioso.

Figura 8.51 — Utilizzo del design pattern Façade.



Corretto utilizzo della ereditarietà

Un problema che spesso si incontra nei modelli OO è legato ad un utilizzo non corretto della relazione di ereditarietà. L'analisi degli equivoci dell'utilizzo della relazione di ereditarietà è stato affrontato dettagliatamente nel paragrafo "I tranelli dell'ereditarietà" presente nel Capitolo 6. In sintesi, non è infrequente che alcuni disegnatori, constatando che alcune classi abbiano qualche parametro o responsabilità in comune, decidano di organizzare queste classi attraverso una struttura gerarchica. A volte si tratta di una soluzione corretta (si vuole effettivamente ereditare il tipo), mentre altre no. Sebbene l'ereditarietà sia un principio base dell'OO, l'utilizzo travisato può generare diversi effetti collaterali (riportati nel Capitolo 6):

- la relazione tra le classi è definita a tempo di disegno e non può variare;
- vi è un consistente accoppiamento tra classi genitore e classi discendenti, riduzione dell'incapsulamento, ecc.

Presenza e corretta applicazione dei pattern

L'applicazione dei pattern nel modello di disegno è un fattore che ne favorisce l'aumento della qualità. Gli stessi disegnatori esperti tendono a sentirsi più a loro agio qualora siano incaricati di revisionare modelli di disegno in cui siano applicati correttamente i design pattern. Ciò perché si tratta di soluzioni ben provate che sicuramente offrono un elevato grado di flessibilità, eleganza, funzionalità. Come ogni cosa però, qualora utilizzata in eccesso, o peggio ancora in modo errato, corre il rischio di generare una serie di problemi. Per esempio, utilizzi ingiustificati tendono a generare modelli over-engineered, ossia inutilmente complessi, difficili da comprendere e mantenere, ecc. I design pattern, pur-

troppo, sono oggetto di una corrente di abuso. Nominarli in un discorso o in un articolo (anche a sproposito) sembrerebbe accrescere il livello di professionalità dei vari soggetti. Ci sono persone che vorrebbero trasformare tutto in un design pattern. Per esempio poiché uno dei principi base dell'OO sancisce che gli attributi delle classi debbano possedere visibilità privata (incapsulamento, information hiding, …), ecco creato il "pattern degli attributi privati". Pur con questi casi eccezionali, la presenza di design patter nei modelli di disegno tende ad essere un elemento di qualità in grado di calmare le ansie del revisore dello stesso modello.

Interfacce coese

Questo principio può essere considerato l'applicazione alle interfacce delle leggi della massima coesione e del minimo accoppiamento. In particolare si vuole evitare una serie di problemi derivanti dal disporre di interfacce gravate da un'eccessiva quantità di responsabilità; in altre parole poco coese. Qualora si riconosca una situazione del genere, è opportuno procedere alla decomposizione dell'interfaccia in una serie di interfacce più semplici e con maggiore grado di coesione. In questo modo, ciascuna interfaccia raggruppa un insieme coeso di servizio utilizzato da un ben definito gruppo di classi client. Chiaramente può capitare l'evenienza di dover disegnare una stessa classe che debba fornire determinati servizi specificati in diverse interfacce. In questo caso è buona norma che le classi client percepiscano le istanze di tale classe attraverso le sue interfacce, piuttosto che per mezzo di una sola interfaccia scarsamente coesa. Ciò migliora la qualità del sistema (è più facile individuare le responsabilità delle varie classi implementanti), permette un riutilizzo più agevole delle parti, diminuisce il livello di fragilità.

Un'anomalia generata dalle interfacce poco coese prende il nome di "inquinamento dell'interfaccia". Questa si genera qualora una specifica classe che implementa una determinata interfaccia richieda di essere comandata o di fornire servizi aggiuntivi rispetto a quelli definiti da tale interfaccia. Spesso, trovandosi di fronte a tale situazione, si procede inserendo nell'interfaccia i metodi aggiuntivi necessari anche se poi questi non possiedono alcuna coesione con i restanti servizi esposti dall'interfaccia stessa. Pertanto, al fine di soddisfare le esigenze di una o poche sottoclassi, si finisce per inquinare l'interfaccia comune. A questo punto tutte le sottoclassi devono necessariamente (a meno di non essere dichiarate astratte) fornire l'implementazione anche dei servizi non utilizzati. Questo causa una serie di problemi come, per esempio, rendere più difficoltosa la riusabilità e la comprensibilità del disegno, aumentare il grado di fragilità del sistema (un'eventuale modifica della definizione dei servizi dell'interfaccia, anche di quelli non implementati dalle varie sottoclassi, comunque genera la necessità di modificare tutte le sottoclassi).

Open-Close: un principio facile facile, ma difficile difficile

Uno dei princìpi più famosi del disegno di sistemi OO (e non solo) è attribuibile al solito Bertrand Mayer (l'ideatore del Design By Contract analizzato nel Capitolo 6). Pur

non essendo di recente coniazione (risale al paleolitico periodo di fine degli anni Ottanta) mantiene intatta la propria validità. Si tratta di un principio che, dopo una riflessione iniziale, appare semplice semplice, quasi lapalissiano, la cui applicazione pratica però non lo è altrettanto.

Il principio afferma che: "I componenti del sistema (classi, package, entità, componenti veri e propri, …) devono essere aperti a estensioni, ma chiusi alle modifiche", da cui il nome *Open-Close*, aperto-chiuso.

Da notare che affinché un componente sia chiuso alle modifiche è necessario che il relativo codice non sia modificabile. Sebbene in prima lettura ciò possa sembrare quasi un controsenso — il comportamento delle componenti deve poter essere modificato senza riscriverle! — un'analisi più attenta ne rivela la raffinatezza.

Si proceda con ordine. Per quanto concerne la prima parte, dovrebbe essere ormai chiaro che è insito nel destino dei sistemi cambiare ed evolvere durante l'intero ciclo di vita: non è possibile congelare i requisiti utente e quindi bisogna disegnare il sistema in modo da riuscire, in qualche misura, ad anticipare le variazioni o comunque renderle più indolori possibile. In termini più pratici, si richiede di poter modificare agevolmente il comportamento di determinati componenti al fine di adeguarli a nuovi requisiti, a nuove applicazioni, e così via. Quando poi si ha bisogno di modificare una parte del sistema, si vuole che gli effetti della variazione siano il più possibile circoscritti e non si propaghino in altre parti di codice funzionante. Qualora ciò non avvenga, evidentemente si ha a che fare con un pessimo disegno e probabilmente si è nel caso di una cattiva architettura.

Per quanto concerne la seconda parte, essa sancisce che si desidera aggiornare il comportamento del sistema senza dover cambiare le parti esistenti.

Questo criterio è anche noto come principio base dei framework. In questi sistemi si desidera avere una parte assolutamente non variabile (il *core*) e tutta una serie di meccanismi che permettano di adattare il comportamento dell'infrastruttura alle specifiche realtà in cui viene utilizzato. Il punto chiave per ottenere questo comportamento consiste nell'utilizzare un principio base dell'OO: il polimorfismo realizzato per mezzo di superclassi astratte (e/o interfacce). Ciò permette di definire un'astrazione di un concetto (quello della classe padre astratta) ed una serie di diversi comportamenti definiti dalle sottoclassi (come nel caso del design pattern Command). In questo modo il principio dell'apertura/chiusura è rispettato: la classe genitore è chiusa alle modifiche e la relativa estendibilità è garantita dalla possibilità si aggiungere sottoclassi che realizzino i nuovi comportamenti specializzati desiderati.

Ora, ipotizzare di riuscire a chiudere tutto il sistema è impensabile. Per quanto si tentasse di chiudere un componente, esisterebbero sempre tipologie di variazioni contro le quali il componente risulti non chiuso (ciò è consistente con il primo principio di Murphy). Ciò nonostante una chiusura parziale ben mirata può migliorare notevolmente il disegno. Inoltre (come riportato nel paragrafo successivo), i meccanismi dell'astrazione, per divenire pienamente operativi, oltre al polimorfismo ed alle classi astratte, necessitano di costruire

un'infrastruttura in grado di selezionare (ad un istante di tempo zero o anche dinamicamente) le specifiche estensione da eseguire (classi specializzanti). A questo scopo si utilizzano design pattern quali il Factory (fig. 8.52). Tutto ciò chiaramente ha un costo, sia in termini di aumento del tempo necessario per il disegno/implementazione, sia di complicazione della configurazione, sia della complessità del sistema stesso. Bisogna anche ricordare che la flessibilità, sebbene sia una proprietà molto ricercata, qualora ecceda determinati limiti, può trasformarsi in un problema.

Con riferimento al modello di disegno di fig. 8.45, si può notare che il sistema prevede una serie di comandi SQL specifici per i vari vendor di data base management system. Si vuole chiaramente che il sistema continui a funzionare variando il software di gestione della base dati. A tal scopo è stato applicato una variante del design pattern Factory dato dalle classi `DBMSVendorFactory`, `OracleSpecific`, `SybaseSpecific`, ecc. Ora, quando il generatore di comandi SQL deve generare clausole dipendenti dallo specifico vendor, richiede al factory l'istanza opportuna contenente tali comandi. Il factory può generare una nuova istanza ad ogni richiesta, oppure generarne un'unica all'atto dell'avvio del sistema. Come riesce però a stabilire quale classe specifica costruire? La soluzione tipicamente consiste nel leggere tale informazione da un opportuno file di configurazione all'atto dell'avvio del sistema.

Data l'impossibilità/impraticabilità di "chiudere" l'intero sistema diventa di cruciale importanza riuscire ad individuare le porzioni del sistema la cui chiusura porterebbe risultare molto vantaggiosa. A questo punto il pubblico dei lettori odierà a morte lo scrittore, ma ancora una volta, architetti e disegnatori esperti, illuminati dalla relativa esperienza, riescono ad individuare intuitivamente le aree più "sensibili", basandosi sia sulla relativa

Figura 8.52 — *Factory per l'istanziazione dell' opportuno oggetto DBMS.*

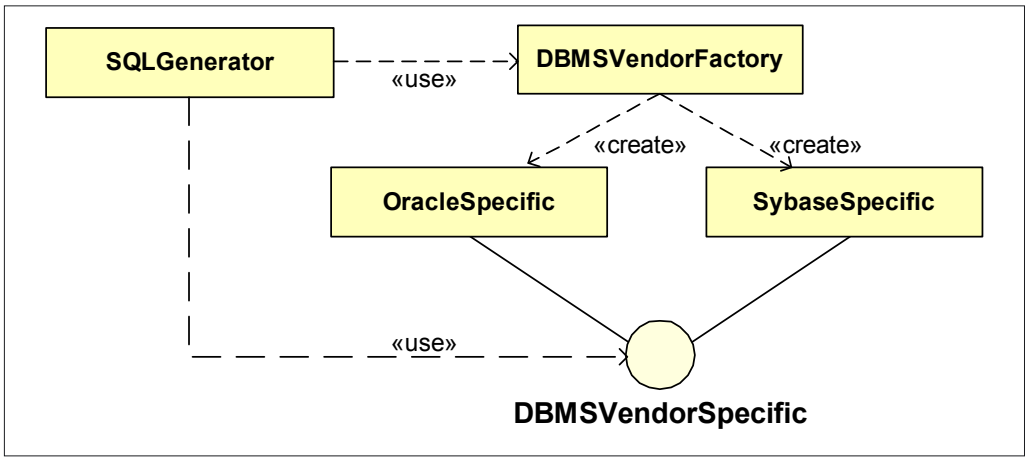
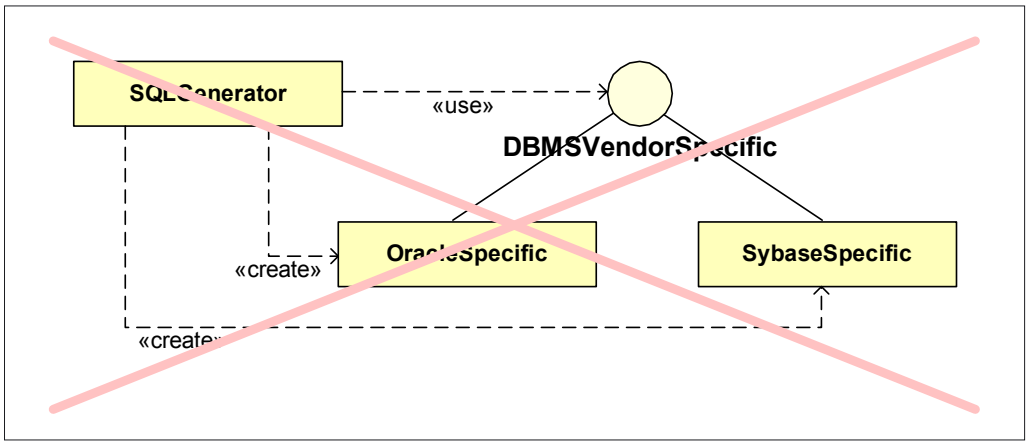


Figura 8.53 — *Errato utilizzo del meccanismo dell'astrazione.*



conoscenza dello stato dell'arte della tecnologia, sia sulla propria intuizione in merito ai requisiti. Per tutti gli altri non c'è troppo da disperare: esistono una serie di criteri e strategie che possono risultare molto utili. Per esempio, è opportuno cercare di applicare il criterio nei casi in cui un componente utilizzi un software o libreria fornita da altri vendor, oppure presenti delle parti specifiche di una determinata tecnologia (come nel caso delle basi dati) e quando la stessa è soggetta ad evoluzioni. Ancora è utile nei casi in cui si ha la sensazione che i requisiti utente possano variare, oppure nei casi in cui è prevista la possibilità di specializzare il comportamento per casi non ancora contemplati.

Utilizzo delle astrazioni

Un altro criterio molto importante da seguire nella realizzazione del modello di disegno, implicitamente presente nei precedenti, è relativo all'utilizzo dei meccanismi dell'astrazione. Si tratta di un buon principio far sì che il disegno (e quindi il codice) dipenda il più possibile da opportune astrazioni piuttosto che da comportamenti concreti. Il criterio quindi suggerisce di affrontare la complessità del sistema conferendo maggiore enfasi alle caratteristiche essenziali dello stesso a spese dei dettagli implementativi.

L'idea alla base è abbastanza semplice: la frequenza con cui variano corpi (concreti) è decisamente maggiore della frequenza con cui cambiano le relative astrazioni.

Come al solito, in sistemi reali è improponibile tentare di far in modo che ogni cosa dipenda da apposite astrazioni (consultare il paragrafo del principio dell'Open-Close). Inserire classi astratte ed interfacce è la classica condizione necessaria ma non sufficiente per garantire la dipendenza dalle astrazioni. Uno degli errori più classici che l'autore ha visto commettere nei modelli di disegno consiste nell'applicare correttamente le astrazioni, dimenticando però di risolvere il problema della selezione (o creazione) degli oggetti. In

altre parole, si disegna un'opportuna interfaccia o classe astratta rappresentante l'astrazione, la si implementa/specializza attraverso un opportuno insieme di classi e poi si commette l'errore di far diventare il sistema dipendente dalle classi concrete anziché dalle relative astrazioni annullando tutti i vantaggi offerti dal ricorso a tale tecnica. Ciò è dovuto al fatto che classi astratte e interfacce non possono essere istanziate direttamente. A tal fine si consideri l'esempio di fig. 8.53.

La realizzazione dell'interfaccia e delle classi specifiche è corretta, così come è corretta la dipendenza (*use*) della classe `SQLGenerator` dall'interfaccia. Il problema è poi che quest'ultima classe genera le istanze delle classi che implementano l'interfaccia (situazione evidenziata dalla presenza degli stereotipi `create` della relazione di dipendenza). Lo stesso problema si ripropone quando si introduce correttamente un Factory, nel quale però viene codificata (*hard-coded*) la conoscenza delle classi specializzate. In quest'ultimo caso si tratta comunque di un modello migliore — nel caso in cui si aggiunga, modifichi o elimini una classe specializzata è sufficiente modificare una sola classe — ma facilmente migliorabile. Per esempio è possibile prevedere un file di inizializzazione da caricare in memoria all'atto dell'avvio del sistema che contenga la conoscenza delle varie classi specializzate presenti, oppure si può utilizzare il pattern del Proxy Dinamico. Questa nuova flessibilità — in caso in cui varino le classi specializzate è sufficiente modificare un file di configurazione e nessuna parte del codice — ha un prezzo che consiste nel prevedere e documentare il file di configurazione, realizzare un modulo per il relativo caricamento iniziale, meccanismi astratti di creazione delle specifiche interfacce.

Criteri generali

Oltre ai criteri citati in precedenza, ne esistono tutta una serie di carattere generale che probabilmente è opportuno tenere in considerazione. Per esempio è opportuno tendere sempre alla semplicità. Un disegno semplice evidentemente è più facile da comprendere, estendere e riutilizzare. Quindi, anche nella modellazione la semplicità è una virtù che, paradossalmente, è molto difficile da raggiungere. Un altro buon principio da seguire è quello che prende il nome di decentramento. Si tratta di una prescrizione intrinseca nelle moderne architetture component-based. Maggiore è il grado di autonomia dei singoli moduli e maggiore è la probabilità di circoscrivere gli effetti di una variazione. Inoltre risulta più semplice reingegnerizzare le varie componenti, riutilizzarle, ecc.

Conclusioni

Nei precedenti paragrafi si è tentato di evidenziare una serie di errori che tipicamente si commettono nella realizzazione dei modelli di disegno, e di fornire un insieme di linee guida da utilizzarsi per verificarne la qualità.

Pur nella consapevolezza di non aver fornito un elenco esauriente (probabilmente non basterebbe un intero volume dedicato all'argomento), si è cercato di condensare i criteri

utilizzati dall'autore del libro sia per realizzare i modelli di disegno, sia per valutarne la relativa qualità. Sebbene l'esperienza giochi ancora un ruolo molto importante, le nuove tecnologie, come per esempio l'architettura EJB, rendono il lavoro meno gravoso: gran parte dell'infrastruttura e dei pattern da utilizzare sono predefiniti e quindi commettere errori gravi è oggettivamente meno facile e richiede un certo impegno.

Caratteristiche di un sistema

Come degna e definitiva conclusione del capitolo si è ritenuto opportuno riportare brevemente le caratteristiche che un software deve possedere per poter essere considerato di buona qualità:

- correttezza. È la capacità del sistema di eseguire correttamente i servizi specificati nei documenti di analisi dei requisiti (*use case*). Si tratta del requisito fondamentale: se un sistema non è in grado di fornire i servizi richiesti, tutte le altre qualità perdono di significato. Quantunque di primaria importanza, la correttezza sembrerebbe un requisito quasi impossibile da ottenere;

- robustezza. È la capacità del sistema di riconoscere e gestire opportunamente situazioni anomale. La robustezza è il logico completamento della correttezza;

- estendibilità. Un sistema è estendibile quando è in grado di assorbire facilmente cambiamenti dei requisiti;

- riusabilità. È la capacità di integrare facilmente componenti provenienti da altri sistemi e di permettere il riutilizzo di proprie parti in altri sistemi;

- efficienza. È la capacità del sistema di eseguire i servizi richiesti dando luogo ad un utilizzo minimo delle risorse (tempo, memoria, canali di comunicazione, ecc.);

- facilità d'uso. Consiste nella facilità di utilizzo del sistema da parte del pubblico degli utenti (pubblico che, tipicamente, contempla persone con diverse qualifiche e diverso livello culturale);

- compatibilità. Si tratta della facilità di combinare diversi elementi software tra loro;

- facilità di gestione. Consiste nella facilità di installare il sistema, controllarne il funzionamento ed intervenire per renderlo nuovamente operativo qualora si verifichi un'interruzione del funzionamento.

- portabilità. È la facilità con cui il sistema può essere trasportato in diversi ambienti hardware e software.

- sicurezza. Capacità del sistema di neutralizzare tentativi di utilizzo e di acquisizione di informazioni da parte di utenti non autorizzati;

- adeguata documentazione. Disponibilità di un buon livello di documentazione del sistema, realizzata con strumenti standard.

Ricapitolando…

Il presente capitolo è dedicato all'illustrazione dell'utilizzo del formalismo dei diagrammi delle classi nel contesto dei processi di sviluppo del software (modelli a oggetti del dominio, del business, di analisi e di disegno). A tal fine sono stati presi in considerazione i processi più formali.

Un primo manufatto di estrema importanza nei processi di sviluppo del software (prodotto durante la fase di analisi dei requisiti utente) è costituito dal modello a oggetti del dominio (Domain Object Model), detto anche modello del dominio. Come suggerito dal nome, rappresenta un modello che ha a che fare con entità "realmente esistenti" nell'area oggetto di studio che il sistema software dovrà automatizzare. Uno degli scopi primari del modello è contribuire alla piena comprensione del contesto in cui il sistema dovrà operare (spazio del problema) focalizzando l'attenzione sulla relativa struttura statica.

Non è infrequente riferirsi allo spazio del dominio del problema con l'espressione "mondo concettuale", in quanto, generalmente, sono presenti "oggetti" che, pur non esistendo apertamente nel mondo reale, da esso "traspirano" oppure possono essere visti come derivati da specifici oggetti attraverso lo studio delle strutture e dei comportamenti.

Il modello a oggetti del domino è un modello statico dello spazio del problema che ne rappresenta un'astrazione in termini delle varie entità presenti corredate dalle relative interconnessioni. Le entità individuate sono ottime candidate di classi persistenti: sono le astrazioni dei dati trattati dal sistema. Secondo le conclusioni dei Tres Amigos [BIB08], le entità presenti nel modello del dominio dovrebbero appartenere essenzialmente alle seguenti categorie:

a. entità "business", ossia oggetti rappresentanti entità (più o meno tangibili) manipolati nell'area business oggetto di studio;

b. oggetti e concetti appartenenti al mondo concettuale che il sistema necessita di manipolare;

c. eventi che possono verificarsi.

Una caratteristica tipica dei modelli a oggetti del dominio consiste nello specificare poche o nessuna operazione nelle varie classi. Verosimilmente, è opportuno rimandare la definizione delle proprietà

comportamentali delle classi a fasi più mature del processo di sviluppo nelle quali si ha una maggiore comprensione del dominio del problema. In questo stadio si è interessati essenzialmente all'organizzazione dei dati e quindi, al livello di classi, alla relativa struttura (attributi e relazioni con altre classi). Il modello a oggetti del dominio, una volta terminato, rappresenta un'asserzione dello spazio del problema o, se si preferisce, il compendio delle regole di business relative all'organizzazione dei dati manipolati. Il DOM è sia un manufatto assolutamente necessario allo sviluppo del sistema, sia un prodotto finale. Si presta ad essere utilizzato per eventuali future reingegnerizzazioni (tipicamente la tecnologia è molto variabile, il business no), per la progettazione di sottosistemi aggiuntivi, per l'illustrazione del business magari a nuovi dipendenti, ecc.

La realizzazione del modello a oggetti del dominio è un'attività complessa fortemente dipendente da quelle che sono proprietà soggettive del singolo individuo come cultura, capacità analitiche, esperienza. Ciò determina la difficoltà di formalizzare un metodo rigoroso e meccanico. La tecnica proposta nel testo è il procedimento denominato metodo dell'analisi dei nomi e dei verbi. Esso è composto essenzialmente da tre attività: individuazione dell'elenco delle ipotetiche classi, scorporazione dell'elenco e individuazione delle relazioni. Il primo passo da seguire è un'attività non eccessivamente complessa, quasi meccanica. Più precisamente è necessario eseguire una sorta di analisi grammaticale. Si legge attentamente il documento dei requisiti (use case, appunti presi durante i colloqui con gli utenti, documentazione varia, ecc.), e si evidenziano tutti i nomi e coppie di nomi/aggettivi incontrati. Questi dovrebbero confluire in un'apposita tabella in cui si riportano, per ogni candidata classe, il relativo nome ed una breve descrizione. Il passo successivo consiste nell'operare una cernita dell'elenco al fine di eliminare le classi candidate che sono fuori scope, oppure che non hanno le caratteristiche delle classi, ecc. La prima domanda da porsi è se la classe appartiene o meno al dominio oggetto di studio. Una risposta negativa non significa necessariamente che l'entità che si sta analizzando non sia una classe, ma semplicemente che non è il caso di spendere tempo ad investigare, a studiarne la struttura ed il comportamento giacché comunque non è di interesse per il modello. La seconda domanda è relativa alle presunte istanze della classe, ed in particolare ai valori che queste potrebbero assumere in un ipotetico caso reale. Se ci si dovesse accorgere che le istanze della classe oggetto di verifica possano assumere unicamente valori semplici, verosimilmente non è il caso di considerarla tale, eventualmente si può trattare di un attributo. Il punto successivo, che a dir il vero risulta intimamente legato al precedente, è relativo all'"analisi" dell'eventuale struttura dell'ipotetica classe. In altre parole, è necessario chiedersi se è possibile identificare una qualche struttura in termini di attributi, relazioni, ecc. In caso di risposta affermativa allora potrebbe avere senso considerare la classe come tale, altrimenti si ricade nel caso di aver individuato un potenziale attributo. Dopo essersi interrogati circa la struttura della classe, è possibile spostare l'attenzione sulle eventuali caratteristiche comportamentali. Qualora sia possibile identificare caratteristiche comportamentali, allora sicuramente è possibile proseguire con la valutazione dei criteri successivi, in caso contrario, la situazione nasconde diverse insidie. Quando si elabora un modello a oggetti del dominio, si focalizza l'attenzione sui dati e sulla relativa organizzazione, mentre il comportamento assume un interesse marginale. Un ulteriore elemento da considerare è se l'ipotetica classe possegga una propria indipendenza oppure abbia senso esclusivamente nel contesto di una più ampia. Ciò equivale ad interrogarsi se l'ipotetica "classe" sia un nuovo tipo di dato con un insieme ben definito e separato di operazioni, oppure se queste possono naturalmente confluire tra

quelle fornite da un'altra classe già accreditata. In caso di risposta affermativa è necessario rivedere la definizione della classe inglobante ed eliminare quella oggetto di verifica.

Le classi ottenute dall'esecuzione dei passi precedenti avrebbero ben poco senso se destinate a rimanere isolate. La situazione normale, al contrario, prevede che ciascuna di esse collabori con le altre al fine di erogare i servizi offerti dal sistema.

L'obiettivo delle primissime fasi dei processi di sviluppo dovrebbe essere quello di modellare, in modo più semplice possibile, lo spazio del problema e non quello delle soluzioni. In questo ambito, lo scopo fondamentale dei modelli è comunicare e documentare. Pertanto è necessario tenere a mente che molti utenti ed esperti del business dispongono di una limitata cultura informatica, è quindi consigliabile rimandare le ottimizzazioni alle fasi di analisi e disegno. Una delle anomalie che più frequentemente si rischia di generare è di introdurre ottimizzazioni, utilizzare design-pattern e raffinate astrazioni che non sempre hanno molto a che fare con il dominio del problema. Ciò può rendere il modello meno comprensibile ad un pubblico tecnicamente meno preparato come quello degli esperti del dominio e degli utenti/clienti. La cosa costituisce un grosso limite considerando che proprio questi soggetti sono tra i fruitori di primaria importanza del modello a oggetti del dominio. Inoltre il modello a oggetti del dominio, prima di assumere una forma stabile, tipicamente subisce molte rielaborazioni e quindi il tempo investito nel produrre raffinate astrazioni ben presto può rivelarsi un notevole spreco. Probabilmente non è neppure il caso di spendere un tempo eccessivo nell'investigare le particolari associazioni. In questo stadio affermare che una relazione sia un'aggregazione piuttosto che una composizione non cambia di molto la sostanza, mentre potrebbe creare problemi al solito utente frastornato dalla nuova simbologia. Nella costruzione del modello a oggetti del dominio la parola d'ordine deve essere utilizzare un approccio minimalista.

Nella realtà accade raramente di dover realizzare un sistema inedito, molto più frequentemente capita di dover di dover affrontare un dominio per il quale esistano in commercio software che affrontano problematiche simili. In tutti questi casi è possibile tentare di generare il modello del dominio per mezzo di una reingegnerizzazione dello schema della base dati. In questo esercizio è necessario tener ben presenti le regole del particolare database management system. Un'altra accortezza è che nella produzione del modello a oggetti del dominio non interessa rappresentare tutte le tabelle con ogni singolo attributo: molti di questi elementi hanno un significato strettamente legato all'implementazione o alla strutturazione, che quindi hanno poco o nulla a che fare con lo spazio del problema.

L'utilizzo di questa tecnica permette di realizzare un modello iniziale che dovrebbe fornire un'ottima piattaforma di dialogo con gli esperti dell'area business.

Di fronte ai vantaggi ben evidenti nel ricorrere all'utilizzo dei tracciati del database (riuscire a produrre velocemente una versione iniziale del modello evitando molti lunghi meeting con gli utenti, risolvere a priori eventuali contenziosi sulle diverse percezioni dell'area business da parte degli utenti, disporre di una versione dell'organizzazione dei dati dello spazio del problema ben investigata e collaudata) esistono tutta una serie di rischi, come per esempio: spendere troppo tempo nel tentare di comprendere la struttura del database, produrre una versione del modello ad oggetto a carattere eccessivamente tecnico, utilizzare una struttura eccessivamente "denormalizzata", e così via. Diverse di queste insidie possono essere attenuate considerando il modello prodotto come una versione iniziale da utilizzarsi come piattaforma di colloquio e non come versione finale del modello.

"Un modello del business mostra l'ambiente dell'azienda e come questa opera in relazione ad esso…" (Ivar Jacobson). Questo descrive come un gruppo di utilizzatori (tecnicamente denominati *worker* della business area) utilizzano un insieme di entità di business (le classi già evidenziate nel modello a oggetti del dominio) attraverso opportune unità di lavoro (*work unit*). In sostanza una *work unit* definisce la logica che specifica i vari servizi erogabili agli attori. In questo contesto l'attenzione è focalizzata sull'intera area business e non esclusivamente sul sottoinsieme da implementare. È possibile affermare che un business object model è un modello a oggetti del dominio arricchito attraverso l'inserimento di concetti come worker e work unit.

Il metodo tipicamente utilizzato per realizzare un modello a oggetti del business prevede di realizzare inizialmente il modello dei casi d'uso di business, da cui prelevare l'elenco degli attori "business" e le work unit. Qualora i casi d'uso posseggano un giusto livello di granularità, si ha quasi una corrispondenza uno ad uno tra singolo caso d'uso e unità di lavoro. Il passo successivo consiste nell'individuare le classi entità (*entity*). L'ultimo passo consiste nel riorganizzazione il tutto: gli attori interagiscono con le varie unità di lavoro, le quali forniscono i propri servizi accedendo allo stato delle varie entità ed eventualmente modificandolo.

La stretta somiglianza esistente tra il modello a oggetti business e del dominio finisce per porli su un piano di alternativa. Se si tentasse di produrre tutti i manufatti previsti dai processi più formali per lo sviluppo del software, verosimilmente sarebbe necessario disporre di un lasso di tempo di qualche ordine di grandezza superiore rispetto a quello effettivamente necessario/disponibile. In questa lotta alla sopravvivenza, tipicamente nei progetti reali, il modello a oggetti del dominio tende ad avere la meglio sul relativo al livello di business. Ciò è dovuto al fatto che è molto utile disporre di una versione dei dati del dominio del problema in quanto agevola la progettazione della base di dati, semplifica la scomposizione del sistema in componenti, agevola la definizione sia delle interfacce utente, sia degli eventuali messaggi utilizzati per far comunicare i vari sottosistemi. Inoltre il modello del business può essere compensato dal modello a oggetti del dominio più la relativa vista dei casi d'uso e la fase successiva prevede il modello di analisi molto simile a quello business.

Molti autori preferiscono invece il modello a oggetti business poiché focalizzando l'attenzione sull'intero business si ha un manufatto di valore molto elevato, anche per finalità non strettamente connesse con la costruzione del sistema software, come la materializzazione del quadro generale da utilizzarsi come punto di partenza per generare diversi progetti, addestramento del personale, ottimizzazioni dei flussi lavorativi.

Il modello a oggetti di analisi è prodotto durante la relativa fase del processo di sviluppo del software e spesso è incorporata con quella di disegno. Si tratta di uno stadio di "raccordo" tra l'analisi dei requisiti utente ed il disegno del sistema. Le classi del modello di analisi sono ancora focalizzate sui requisiti funzionali del sistema, mentre l'incorporazione di quelli non funzionali è rimandata al modello di disegno. Nella fase di analisi il livello di formalismo è sicuramente superiore rispetto a quello delle fasi precedenti, ma non ancora uniformato a quello delle fasi successive. Gli esperti del dominio tendono ancora ad essere coinvolti ma con ruoli molto marginali. Le classi del modello di analisi risultano più vicine allo spazio del dominio, più concettuali e verosimilmente meno granulari delle corrispondenti a livello di disegno. In questa fase non si può ancora pretendere un modello dettagliato e tanto meno un elevato grado di formalità. L'applicazione degli insegnamenti propri dell'OO sovente è sacrificata a favore del risparmio di tem-

po. Inoltre, come nel caso del modello a oggetti del dominio, ancora non si è molto interessati a definire le operazioni delle varie classi, ciò sia perché probabilmente non si dispone ancora di sufficienti informazioni, sia perché il tempo richiesto probabilmente non sarebbe giustificabile nel contesto dell'intero processo di sviluppo del software.

Il modello a oggetti di analisi deve essere realizzato utilizzando esclusivamente tre stereotipi "standard" dell'elemento classe: *boundary*, *entity* e *control*.

Una *boundary class* (classe di confine) rappresenta un "punto di accesso" al sistema, una componente dello strato di interfacciamento con i relativi attori. Si tratta di parti del sistema fortemente dipendenti dagli attori che le utilizzano e permettono di investigare, chiarire e formalizzare i requisiti del sistema a livello di interfacciamento con entità esterne. Considerata la fase di questo modello e l'elevato livello di astrazione non è necessario dettagliare eccessivamente le classi boundary; è sufficiente specificare gli obiettivi dell'interazione, i dati che si intendono scambiare, e così via, senza però descrivere come questa poi venga realizzata.

Le classi entity (entità) sono classi atte a modellare informazioni gestite dal sistema che necessitano di essere memorizzate in forma permanente. Sono passive, sebbene possano disporre di comportamenti anche complessi legati alle informazioni contenute. Uno degli scopi delle classi entity consiste nell'isolare i dettagli della memorizzazione delle informazioni dalle classi che le manipolano (control).

Le *control class* (classi di controllo) rappresentano il controllo, nel senso più generale del termine (coordinamento, transizione, successione, ecc.), di altri "oggetti". Tipicamente sono utilizzate per incapsulare la logica di controllo presente nelle funzionalità definite a livello dei casi d'uso. Spesso rappresentano algoritmi anche complessi di calcolo, gli attuatori delle policy aziendali e regole di business. Una delle caratteristiche peculiari delle classi di controllo consiste nell'incapsulare le logiche del sistema, e quindi nel ridurre l'impatto, sulle restanti parti, dovuto ad eventuali variazioni. Tipicamente tendono ad assumere la fisionomia di "macroclassi" e quindi raramente sopravvivono al modello di disegno, nel quale ciascuna classe di controllo tende a dar luogo a svariate classi.

Nella produzione del modello di analisi è necessario tenere presenti quattro regole fondamentali:

1. gli attori possono essere associati unicamente a boundary class;

2. le boundary possono essere associate esclusivamente con attori e control class (questa regola equivale ad affermare che non è possibile associare tra loro due boundary class così come non è possibile collegare direttamente boundary class con entity);

3. le classi entità possono essere associate esclusivamente con quelle di controllo (si vieta di associare tra loro due entity class e di collegarle direttamente o con le boundary o addirittura con gli attori);

4. le classi control non possono essere associate con gli attori.

Il modello di analisi, pur offrendo diversi vantaggi, non rientra nella categoria di quelli estremamente necessari per la buona riuscita dell'intero progetto. Il problema è che il costo di produzione e manutenzio-

ne non sempre ne giustifica i vantaggi offerti. La decisione se dar luogo o meno ad una fase distinta dipende da molti fattori, non ultimi la dimensione del progetto, il tempo a disposizione, la tipologia, la tecnologia coinvolta, e così via.

Uno dei vantaggi offerti dal modellare i requisiti utente attraverso i diagrammi delle classi deriva dalla semplicità e rapidità di aggiornamento degli stessi. Il modello di analisi, appartenendo ad un livello di astrazione più elevato rispetto a quello di disegno, rende più agevole analizzare nuove richieste, studiarne l'impatto, il tempo necessario e quindi i costi. L'elevato livello di astrazione è particolarmente apprezzato anche per attività quali la formazione del nuovo personale tecnico coinvolto nello sviluppo e manutenzione del sistema: offre una vista concettuale precisa del sistema sottostante in modo più agevole e meno dettagliato del corrispettivo al livello di disegno. I modelli di analisi assumono un ruolo preziosissimo quando si ha a che fare con sistemi legacy datati o con sistemi prodotti da terze parti (in tali casi probabilmente non è il caso di realizzare modelli di disegno e quindi il modello di analisi risulta quello più vicino ai dettagli implementativi). Ulteriori vantaggi sono: rapida e formale acquisizione dei requisiti del sistema da parte del team degli architetti incaricati di realizzare il modello di analisi, possibilità di rivedere rigorosamente il modello dei casi d'uso e quello a oggetti del dominio, ecc. Lo svantaggio principale del modello è sempre lo stesso: il tempo necessario per la produzione e manutenzione. Se il costo della produzione iniziale a volte è giustificabile per i motivi enunciati, la relativa manutenzione lo sarebbe molto meno. Spesso il modello di analisi viene considerato da molti tecnici un modello usa e getta: una volta ottenuti i vantaggi legati alla realizzazione del modello iniziale non ci si preoccupa più di aggiornarlo. Un altro contro del modello di analisi è legato al fatto che, nella pratica, raramente si produce la versione iniziale del modello di disegno disegnando tutto da zero. In genere si tende a riapplicare strutture, pattern, direttive utilizzate in precedenza dimostratisi soddisfacenti e quindi si parte cercando di riutilizzare architetture funzionanti in precedenti progetti.

Nel corso dell'esecuzione dei processi di sviluppo del software, l'ultimo modello a oggetti che si incontra è quello relativo alla proiezione statica del modello di disegno. Il modello di disegno può essere decomposto in due sottomodelli relativi alla struttura statica ed al comportamento dinamico. La proiezione statica si presta ad essere rappresentata per mezzo dei diagrammi delle classi, mentre per la descrizione delle dinamiche interne del sistema è opportuno utilizzare diagrammi di interazione (sequenza, *sequence* e collaborazione, *collaboration*) e, meno frequentemente, i diagrammi di attività, particolarmente utili qualora si voglia enfatizzare l'esecuzione concorrente di determinati processi. Resta poi sempre valido il consiglio di produrre diagrammi degli oggetti per esemplificare porzioni di diagrammi delle classi ritenuti poco chiari.

Il modello di disegno è decisamente orientato alle soluzione tecniche e pertanto si configura come ambiente naturale di architetti e programmatori. Dopo aver lungamente argomentato le funzionalità che il sistema dovrà realizzare, l'attenzione viene focalizzata sul dettaglio di come ottenerle. Le classi presenti nei precedenti modelli a oggetti vengono "incastonate" in una serie di infrastrutture tecniche che rendono possibile realizzare quando stabilito nella vista dei casi d'uso. Il modello di disegno concorre a definire la descrizione analitica dell'architettura software del sistema in funzione dei requisiti catturati nelle fasi precedenti e, pertanto, rappresenta il dettaglio delle specifiche per la fase di codifica.

Di modelli di disegno ne esistono diverse versioni o, se si preferisce, ne esiste un'unica che evolve durante lo svolgimento delle fasi di elaborazione e costruzione, passando dal livello di specifica via via a

quello di implementazione. Le prime versioni del modello presentano un livello di dettaglio non eccessivamente approfondito. In questo modello trovano posto, sicuramente, tutte le classi principali ed i metodi più importanti. È opportuno definire fin da subito i metodi pubblici con la relativa firma in modo da garantire la coerenza del modello fin dalle primissime fasi, evitando ai programmatori l'incombenza di dover operare scelte "locali".

Il modello di implementazione, invece, nella versione finale è una rappresentazione grafica del codice (tenendo sempre presente che "solo il codice è sincronizzato con sé stesso" [S. Ambler]).

Uno degli input per la produzione del modello di disegno è costituito dal modello di analisi. Non è infrequente però il caso in cui questo venga tralasciato per questioni di economia dei tempi. Chiaramente la decisione di trascurare il modello di analisi comporta un certo rischio che però può essere gestito e mitigato dalla presenza di architetti esperti.

Poiché il modello di disegno è un'astrazione del codice, è necessario che accolga caratteristiche dipendenti dal linguaggio di programmazione.

Ulteriori vincoli da tenere presenti durante la realizzazione del modello di disegno derivano dall'ambiente di produzione del codice e da proprietà quali riusabilità dei componenti, scalabilità del sistema, tecnologie a disposizione (per esempio database relazionali o OO, sistema operativo, ecc.).

In questa fase anche le definizione precisa delle diverse tipologie di relazioni ha un significato semantico importantissimo. Specificare un'aggregazione piuttosto che una composizione ha, per esempio, molte implicazioni in termini della gestione del ciclo di vita degli oggetti istanze delle classi coinvolte nell'associazione stessa. In particolare una classe composta dovrà fornire tutti metodi necessari per creare e distruggere le istanze delle classi che la compongono.

La CRC cards (*Class – Responsibility – Collaborator Cards*, carte di classe – responsabilità – collaboratrici) è una metodologia ampiamente utilizzata nel processo di progettazione e documentazione dei modelli a oggetti, il cui dominio di applicazione è essenzialmente quello dei modelli a oggetti di analisi e disegno. La versione iniziale fu congegnata da Ward Cunningham con il nome di HyperCard presso i laboratori della Apple Computer Inc. negli anni Ottanta. La metodologia prevede l'utilizzo di una serie di schede vuote (da cui il nome Cards) le cui dimensioni sono definite nelle stesse direttive, ossia: 6" × 4" (152 × 102 mm). Il processo prevede di selezionare gli use case e/o gli *scenario* di cui si intende disegnare il modello e quindi iniziare a leggere i passi che ne compongono il flusso di azioni. Ogni qualvolta si incontrano riferimenti a oggetti precedentemente individuati (sia oggetti entità e quindi appartenenti al modello a oggetti del dominio, sia "manager" di determinate funzioni) si posiziona la relativa card su un'apposita lavagna. Qualora nessuna delle classi a disposizione risulti idonea per espletare l'azione oggetto d'esame è necessario compilare una nuova scheda: si è individuata una nuova classe. Gli aggiornamenti delle schede sono necessari qualora: 1. l'attività considerata, sebbene concettualmente appartenente ad una classe individuata in precedenza (criterio di coesione), non sia contemplata nella relativa lista delle responsabilità 2. una scheda, per assolvere ad una nuova responsabilità necessiti di utilizzare i servizi offerti da un'altra non menzionata nella propria lista delle collaborazioni. Altri aggiornamenti avvengono durante la revisione del modello qualora esistano classi con troppe responsabilità, diverse molto simili tra loro, altre superflue, ecc. Ogni qualvolta si inizia a leggere un flusso di azioni, le prime carte da considerare tendono ad essere le stesse: la prima rappresenta un'istan-

za dell'attore che ha fornito lo stimolo iniziale al caso d'uso, la seconda rappresenta l'interfaccia, ossia l'oggetto che riceve lo stimolo fornito e quindi avvia di conseguenza il relativo processo, la terza la classe di gestione dell'interfaccia utente, ecc.

Il processo è intrinsecamente iterativo e incrementale: man mano che si procede con l'analizzare nuovi scenari si identificano nuove classi e si perfezionano quelle individuate in precedenza.

Terminata l'analisi di uno scenario, è importante effettuare tutti i controlli del caso sia al livello delle singole classi che lo costituiscono, sia a livello più generale del disegno.

Durante il processo di revisione spesso si dà luogo ad un refactoring introducendo opportuni oggetti, inglobandone altri in uno solo e così via, al fine di migliorare l'organizzazione del disegno, rendere i modelli più stabili, riflettere la struttura a strati delle moderne architetture.

Una volta raggiunto uno stadio in cui si abbia la sensazione di aver elaborato una buona ipotesi di disegno, è possibile effettuare una verifica formale del comportamento dinamico del modello realizzato.

Una evoluzione della metodologia consiste nel disporre di carte del "potere", ossia carte rappresentanti soluzioni generiche, eleganti, dimostratesi valide nella soluzione di altri problemi: in poche parole carte rappresentanti pattern.

L'applicazione della metodologia classica prevede che al gioco delle carte prendano parte diverse figure professionali. Per esempio è possibile coinvolgere un business analyst da interpellare qualora gli use case non siano chiari o si individuino delle incoerenze/lacune, uno o più architetti, programmatori, un esperto della metodologia delle CRC Cards (*CRC Cards facilitator*) ecc. Spesso il team viene corredato dagli "scrivani", il cui obiettivo è di compilare le varie schede man mano che i vari elementi vengono evidenziati. Da tenere presente che, quantunque ogni persona possa fornire ottimi spunti alla realizzazione del disegno, le riunioni con troppe persone tendono a generare elevato grado di entropia.

I vantaggi principali della metodologia delle CRC Cards risiedono indubbiamente nella semplicità e nell'immediatezza di utilizzo. Un altro grande vantaggio consiste nell'economicità con cui è possibile produrre modelli, verificarli, cambiare le fondamenta logiche, e così via.

Grazie a questi vantaggi, la metodologia delle CRC Cards si delinea come un ottimo strumento per scopi formativi. In particolare, la "manualità" della tecnica e la semplicità e rapidità con cui è possibile produrre e aggiornare i propri disegni aiutano ad abbattere molti timori.

Gli svantaggi della metodologia sono dovuti sia alla difficoltà di organizzare meeting produttivi con le varie parti, specie con gli utenti, sia alla presenza di strumenti più adeguati (lo UML) ed architetture standardizzate.

L'ultima parte del capitolo è dedicata all'incosciente tentativo di rispondere a quello che è uno degli interrogativi più controversi e complicati dell'intero processo di sviluppo del software: quando un modello di disegno OO può essere considerato tale e ben progettato?

In particolare si è cercato di fornire una serie di criteri che permettano di evidenziare errori tipici durante il disegno del sistema, di verificare la qualità degli stessi modelli e di produrne versioni di livello qualitativo elevato, senza avere assolutamente la presunzione di essere esaustivi. Prima di addentrarsi nell'esame di una serie di principi utili e linee guida per la realizzazione di modelli di disegno di buona qualità, è stato ritenuto interessante presentare alcuni sintomi che permettono di evidenziare situazioni critiche, in cui bisogna necessariamente intervenire prima che la qualità del sistema risulti irreparabilmen-

te compromessa. L'analisi delle situazioni di crisi è molto utile anche perché aiuta a comprendere più intimamente la necessità di realizzare modelli di disegno di qualità.

Uno dei sintomi di malessere dell'architettura più semplici da individuare è la cosiddetta fragilità. Con tale termine si fa riferimento alla tendenza dei sistemi di generare malfunzionamenti di sue parti a seguito della realizzazione di aggiornamenti. Ciò che spesso avviene e che a seguito di alcune modifiche apportate in specifici settori del sistema si assiste alla perdita di funzionalità di altre parti apparentemente non relazionate con i settori modificati.

Altra anomalia, strettamente connessa con la precedente è quella che va sotto il nome di rigidità, con tale termine si indica la resistenza del sistema ai cambiamenti. Chiaramente se ogni cambiamento corre il rischio di distruggere il sistema in parti, questo è fragile e rigido.

Un altro problema è quello noto con il nome di immobilità, si tratta del contrario della riusabilità. Evidenzia situazioni in cui è difficile riutilizzare parti dello stesso sistema o di altri realizzati in precedenza.

Un ulteriore problema è legato all'evoluzione incoerente del disegno che finisce per rendere lo stesso rigido e, frequentemente, fragile. Quando un architetto inizia a disegnare il sistema, o sue parti, utilizza determinati principi, fissa delle colonne portanti dell'architettura del sistema, e così via: tutto appare perfetto. A seguito poi di aggiunta di nuove funzionalità e/o correzioni, può accadere che le nuove parti violino i principi base, le regole stabilite in partenza dall'architetto.

Per quanto concerne la realizzazione di modelli di buona qualità, un primissimo criterio da considerare scaturisce da un ragionamento logico abbastanza semplice: se le singole parti componenti di un sistema sono di buona qualità, allora verosimilmente è più facile che l'intero sistema sia di qualità. Chiaramente sono possibili deviazioni in entrambe le direzioni: alcuni componenti non perfettamente disegnati e sistemi ancora di buon livello qualitativo e viceversa componenti ben disegnati e sistema scadente.

Non bisogna essere eccessivamente rigidi su questo principio: spesso la qualità del disegno di alcune classi viene intenzionalmente sacrificata a favore della qualità del sistema generale. Chiaramente, qualora si tratti di qualche classe non perfettamente disegnata, è possibile sorvolare, quando invece gran parte o tutte le classi sono disegnate male, o alcune sono veramente pessime, allora è inutile procedere nell'analisi dei rimanenti criteri: il modello non va.

Un altro elemento cui spesso non si attribuisce la dovuta importanza è l'organizzazione in package del sistema. Ciò è molto importante poiché, tipicamente, il riutilizzo delle parti raramente avviene al livello delle singole classi: normalmente si cerca di riutilizzare interi package. Un buon criterio da seguire per strutturare i package deriva dai principi classici: massima coesione e minimo accoppiamento. Le classi inserite nello stesso package devono essere "coese" tra loro ed avere un accoppiamento tale da rendere il package stesso scarsamente accoppiato con i restanti.

Un problema che spesso si incontra nei modelli OO è legato ad un utilizzo non coretto della relazione di ereditarietà. In sintesi, non è infrequente che alcuni disegnatori, constatando che alcune classi hanno qualche parametro o responsabilità in comune, decidano di organizzare queste classi attraverso una struttura gerarchica. A volte si tratta di una soluzione corretta (si vuole effettivamente ereditare il tipo), altre volte no. Sebbene l'ereditarietà sia un principio base dell'OO, l'utilizzo travisato può generare diversi effetti collaterali: la relazione tra le classi è definita a tempo di disegno e non può variare, vi è un consistente accoppiamento tra classi genitore e classi discendenti, riduzione dell'incapsulamento.

L'applicazione dei pattern nel modello di disegno è un fattore che ne favorisce l'aumento della qualità. Si tratta di soluzioni ben provate che sicuramente offrono un elevato grado di flessibilità, eleganza, funzionalità. Come ogni cosa, però, qualora utilizzata in eccesso o peggio ancora, in modo errato, corre il rischio di generare una serie di problemi. Per esempio, utilizzi ingiustificati tendono a generare modelli over-engineered, ossia inutilmente complessi, difficili da comprendere e mantenere.

Un altro criterio consiste nel porre attenzione al disegno delle interfacce. In particolare si vuole evitare una serie di problemi derivanti dal disporre di interfacce gravate da un'eccessiva quantità di responsabilità; in altre parole poco coese. Qualora si riconosca una situazione del genere, è opportuno procedere alla decomposizione dell'interfaccia in una serie di interfacce più semplici e con maggiore grado di coesione. In questo modo, ciascuna interfaccia raggruppa un insieme coeso di servizio utilizzato da un ben definito gruppo di classi client. Un'anomalia generata dalle interfacce poco coese prende il nome di "inquinamento dell'interfaccia". Questa si genera qualora una specifica classe che implementa una determinata interfaccia richieda di essere comandata o di fornire servizi aggiuntivi rispetto a quelli definiti da tale interfaccia. Spesso, trovandosi di fronte a tale situazione, si procede inserendo nell'interfaccia i metodi aggiuntivi necessari anche se poi questi non possiedono alcuna coesione con i restanti servizi esposti dall'interfaccia stessa. Ciò causa una serie di problemi, come per esempio rende più difficoltosa la riusabilità e la comprensibilità del disegno, aumenta il grado di fragilità del sistema (un'eventuale modifica della definizione dei servizi dell'interfaccia, anche di quelli non implementati dalle varie sottoclassi, comunque genera la necessità di modificare tutte le sottoclassi), ecc.

Uno dei princìpi più famosi del disegno di sistemi OO e non solo, è quello che prende il nome di *Open-Close* (aperto-chiuso) coniato da Bertrand Mayer. Il principio afferma che: "Le componenti del sistema (classi, package, entità, componenti veri e propri, …) devono essere aperte a estensioni, ma chiuse alle modifiche".

Da notare che affinché un componente sia chiuso alle modifiche è necessario che il relativo codice non sia modificabile. Per quanto concerne la prima parte (apertura), dovrebbe essere ormai chiaro che è insito nel destino dei sistemi cambiare e quindi bisogna disegnare il sistema in modo da riuscire, in qualche misura, ad anticipare le variazioni o comunque renderle più indolori possibile. In termini più pratici, si richiede di poter modificare agevolmente il comportamento di determinati componenti in modo che gli effetti della variazione siano il più possibile circoscritti. Per quanto concerne la seconda parte, essa sancisce che si desidera aggiornare il comportamento del sistema senza dover cambiare le parti esistenti.

Questo criterio è anche noto come principio base dei framework. In questi sistemi, si desidera avere una parte assolutamente non variabile (il *core*) e tutta una serie di meccanismi che permettano di adattare il comportamento dell'infrastruttura alle specifiche realtà in cui viene utilizzato.

Chiaramente, ipotizzare di riuscire a chiudere tutto il sistema è impensabile. Per quanto si tentasse di chiudere un componente, esisterebbero sempre tipologie di variazioni contro le quali il componente risulti non chiuso. Ciò nonostante una chiusura parziale ben mirata può migliorare notevolmente il disegno.

Un altro criterio molto importante da seguire nella realizzazione del modello di disegno è relativo all'utilizzo dei meccanismi dell'astrazione. Si tratta di una buona norma far in modo che il proprio disegno (e quindi il codice) dipenda il più possibile da opportune astrazioni piuttosto che da comportamenti concreti.

Pur nella consapevolezza di non aver fornito un elenco esauriente (probabilmente non basterebbe un intero volume dedicato all'argomento), si è cercato di condensare i criteri utilizzati dall'autore sia per realizzare i modelli di disegno, sia per valutarne la relativa qualità. Sebbene l'esperienza giochi ancora un ruolo molto importante, le nuove tecnologie, come per esempio l'architettura EJB, rendono il lavoro meno gravoso: gran parte dell'infrastruttura e dei pattern da utilizzare sono predefiniti e quindi commettere errori gravi è oggettivamente meno facile e richiede un certo impegno.

Come degna e definitiva conclusione del capitolo sono state riportare brevemente le caratteristiche che un software deve possedere per essere considerato di buona qualità: correttezza, robustezza, estendibilità, riusabilità, efficienza, facilità d'uso, compatibilità, facilità di gestione, portabilità, sicurezza e adeguata documentazione.