Capitolo 2

UML: struttura, organizzazione, utilizzo

L'esperienza insegna che la capacità di comunicare il comportamento
e il disegno di un sistema prima della sua implementazione
è cruciale per la buona riuscita dello sviluppo.
WALKER ROYCE

Introduzione

Scopo del presente capitolo è fornire una panoramica generale dello UML corredata da opportuni esempi. In particolare viene esposta l'organizzazione in viste e vengono presentati molto brevemente i diagrammi di cui ciascuna di essa è composta. Particolare attenzione è stata attribuita ai metodi forniti dal linguaggio UML per estenderne sintassi e semantica.

Sebbene l'illustrazione dettagliata dei processi di sviluppo esuli dagli obiettivi del libro, durante la stesura del presente capitolo ci si è resi conto che un'illustrazione decontestualizzata da un processo di sviluppo avrebbe probabilmente finito per apportare ben poco beneficio al lettore. Pertanto, senza avere la pretesa di scrivere un libro sui processi di sviluppo, si è ritenuto indispensabile prenderne in considerazione uno di riferimento al fine di illustrare più concretamente l'utilizzo dello UML per la realizzazione dei modelli previsti dallo stesso processo.

Tutti coloro che già conoscono lo UML potrebbero avere la tentazione di procedere direttamente alla lettura del capitolo successivo e forse si tratta di una buona idea. Nonostante ciò, nel presente capitolo sono inserite riflessioni relative ai processi di sviluppo, ai meccanismi di estensione e ai profili che potrebbe valer la pena leggere, magari sull'autobus o in metropolitana

Per tutti i meno esperti il consiglio è di non ostinarsi e perder tempo nel tentar di capire da subito tutti i meccanismi presenti in ciascun diagramma: in questo capitolo hanno valenza unicamente introduttiva (per questo motivo si è cercato di non eccedere con il livello di formalità). L'importante è cominciare a crearsi un'idea delle potenzialità dello UML e prendere familiarità con i vari strumenti messi a disposizione e le relative aree di utilizzo: per ciascuno dei concetti introdotti è presente una trattazione precisa ed esaustiva nel capitolo dedicato all'argomento.

Si è colta l'occasione per proporre un primo esempio di progetto completo: un sistema Internet/Intranet per l'acquisto di biglietti del teatro.

La struttura

Lo UML presenta una tipica struttura a strati; procedendo dall'esterno verso l'interno, essa è costituita da:

- viste
- diagrammi
- elementi del modello

Le **viste** mostrano i diversi aspetti di un sistema per mezzo di un insieme di diagrammi. Si tratta di astrazioni, ognuna delle quali analizza il sistema secondo un'ottica diversa e ben precisa (funzionale, non funzionale, organizzativa, ecc.). La totalità di queste viste fornisce il quadro d'insieme della modellazione del sistema.

I **diagrammi** permettono di descrivere graficamente le viste logiche. Lo UML prevede ben nove tipi di diagrammi differenti, ognuno dei quali è particolarmente appropriato a essere utilizzato in particolari viste.

Per ciò che concerne gli **elementi del modello**, essi sono i concetti che permettono di realizzare i vari diagrammi. Alcuni esempi di elementi sono: attori, classi, packages, oggetti, e così via.

> Durante la fase di definizione dei vari elementi, per i *Tres Amigos* fu necessario affrontare il problema di stabilire quanti e quali elementi dovessero essere inglobati nel linguaggio e se era il caso di iniziare a catalogare tutti quelli immaginabili. Da un lato ciò sarebbe stato utile perché ogni applicazione o tecnologia avrebbe avuto il proprio set di simboli; dall'altro si sarebbero create situazioni non desiderabili: linguaggio rigido, eccessivamente complesso e comunque carente: si trova sempre un progettista *Tizio* che abbia la necessità di esprimere qualche concetto e non trova gli strumenti opportuni per farlo. Addirittura si possono incontrare presunti tecnici, "guerrafondai" non paghi della pace raggiunta dopo decenni di

guerra dei "metodi di analisi", che tentano ancora di riaccendere qualche focolaio inventando propri formalismi.
Il buon senso, ovviamente, portò a preferire un altro approccio: si decise di definire e standardizzare un certo numero di elementi base invarianti (*core*, nucleo), ritenuti fondamentali, e di fornire un insieme di altri meccanismi atti a estendere la semantica del linguaggio (*stereotypes*) per aggiungere documentazione, note, vincoli, ecc.

I vantaggi offerti da tale soluzione furono, essenzialmente, mantenimento della "semplicità" e, contestualmente, conferimento del dinamismo e della flessibilità necessari per rendere lo UML in grado di adattarsi ai più svariati settori.
Come però spesso accade, attraverso riscontri ottenuti dall'applicazione dello UML in progetti reali, fu possibile realizzare che, probabilmente, la soluzione migliore era una via di mezzo: era necessario standardizzare collezioni predefinite di estensioni denominate profili.
Quindi a partire dal nucleo base, utilizzando opportunamente i meccanismi di estensione dello UML, è possibile definire una serie di *plug-in* di elementi standardizzati (appunto i profili) dimostratisi particolarmente utili nell'applicazione dello UML in progetti che sfruttano le architetture ricorrenti (EJB, CORBA, ecc.) o per specifici *workflow* (analisi del business, analisi, ecc.).
È possibile pensare ai profili come alle librerie utilizzabili nella stesura del codice. Sebbene ognuno possa definirsi e riscrivere librerie proprie per risolvere particolari problemi, ne esistono tutta una serie di predefinite (gratuite), ben testate e universalmente accettate… Eventualmente nessuno vieta di estenderle, ma davvero conviene reinventare la ruota?
Il rischio a cui si andava incontro per via della mancanza di tale standardizzazione era una ennesima proliferazione di linguaggi — o meglio dialetti — che avrebbero finito per ridurre molti dei vantaggi offerti dallo UML.

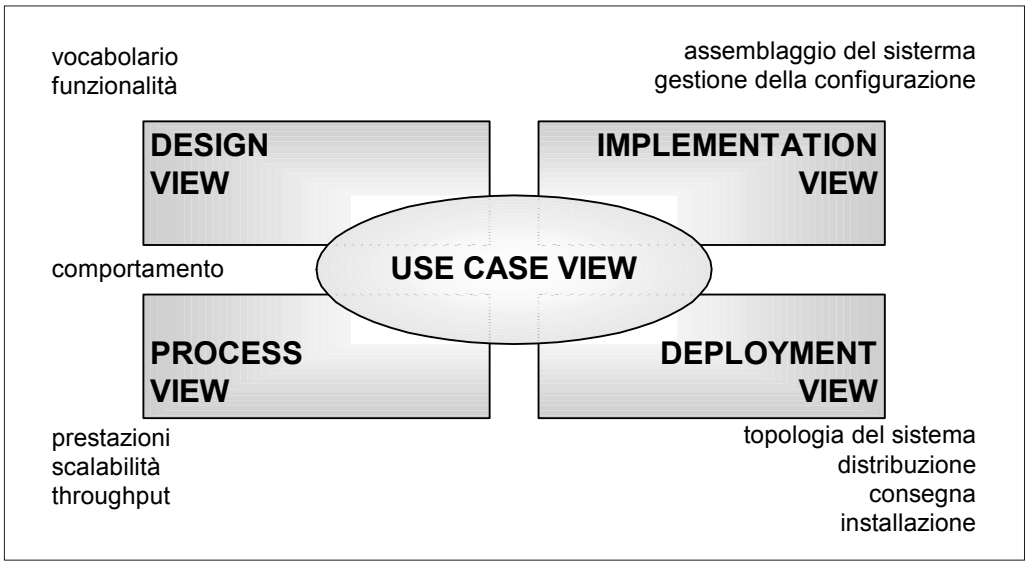
Le viste

In prima analisi, lo UML è organizzato in viste, e in particolare è costituito dalle viste **Use Case**, **Design**, **Implementation**, **Process** e **Deployment** come illustrato nella fig. 2.1.

In breve, la prima vista, **Use Case View** (vista dei casi d'uso), è utilizzata per analizzare i requisiti utente: specifica le funzionalità del sistema come vengono percepite dalle entità esterne al sistema stesso dette Attori. Dunque, si tratta di una vista ad alto livello di astrazione, e di importanza fondamentale perché, nella maggioranza dei processi, guida lo sviluppo delle rimanenti, perché è il manufatto principale utilizzato per ottenere riscontri dal cliente, perché stabilisce le funzionalità che il sistema dovrà realizzare (quelle per le quali il cliente paga).

Obiettivo di questo livello di analisi è studiare il sistema considerandolo come una scatola nera: è necessario concentrarsi sul *cosa* il sistema deve fare astraendosi il più possibile dal *come*: è necessario individuare tutti gli attori, i casi d'uso e le relative associazioni.

Figura 2.1 — *Diagramma delle viste dello UML.*



La vista dei casi d'uso è costituita da due proiezioni: quella statica catturata dai diagrammi dei casi d'uso e quella dinamica rappresentante le interazioni tra gli attori e il sistema.

Importante è dettagliare i requisiti del cliente, carpirne i desideri più o meno inconsci, cercare di prevederne i possibili sviluppi futuri, ecc.

La **Design View** (vista di disegno, talune volte indicata come **Logical View**, vista logica), specularmente alla precedente, descrive come le funzionalità del sistema debbano essere realizzate, in altre parole si analizza il sistema dall'interno (scatola trasparente). Anche questa vista è composta sia dalla struttura statica (diagrammi delle classi e diagrammi degli oggetti), sia dalla collaborazione dinamica dovuta alle interazioni tra gli oggetti che lo costituiscono (diagrammi di comportamento del sistema).

La **Implementation View** (vista implementativa, detta anche **Component View**, vista dei componenti) è la descrizione di come il codice (classi per i linguaggi OO) debba essere accomunato in opportuni moduli (package) evidenziandone le reciproche dipendenze.

La **Process View** (vista dei processi, detta anche **Concurrency View**, vista della concorrenza), rientra nell'analisi degli aspetti non funzionali del sistema e consiste nell'individuare i processi e i processori. Ciò sia al fine di dar luogo a un utilizzo efficiente delle risorse, sia per poter stabilire l'esecuzione parallela degli oggetti (almeno quelli più importanti), sia per gestire correttamente eventuali eventi asincroni, e così via.

La **Deployment View** (vista di "dispiegamento"), mostra l'architettura fisica del sistema e fornisce l'allocazione delle componenti software nella struttura stessa.

Ognuna di queste viste prevede la realizzazione di diversi modelli prodotti da differenti tipologie di lavoratori e quindi, entro certi limiti, questa organizzazione supporta processi di sviluppo del software ad elevato grado di parallelismo.

> I moderni processi di sviluppo del software -come si vedrà a breve- prevedono la costruzione del sistema iterandone lo sviluppo attraverso una serie ben definita di fasi, ognuna delle quali necessita di precisi prodotti (*artifact*, manufatti) di input, genera determinati manufatti di output, richiede l'allocazione di specifiche risorse e l'esecuzione di ben definite attività. Il processo è pertanto strutturato secondo un'organizzazione che non si presta ad essere tracciata direttamente all'impostazione in viste dello UML. Per esempio, tipicamente sono previste diverse versioni del modello di analisi dei requisiti, i quali oltre a comprendere i modelli dei casi d'uso, ne includono anche altri, quali per esempio i modelli ad oggetti atti a rappresentare l'ambiente oggetto di studio. Pertanto, nella stragrande maggioranza dei casi, non è opportuno pensare di studiare e/o adottare un processo di sviluppo in cui le fasi risultino in corrispondenza biunivoca delle viste dello UML, al contrario, è del tutto naturale che ogni processo (che intenda far uso dello UML) pianifichi dettagliatamente come adattare l'organizzazione dello UML alla propria struttura.
> Per avere un'idea di come le viste ed i diagrammi dello UML si adattino ai processi di sviluppo del software formali, si consideri la fig. 2.3.

I diagrammi

Lo UML definisce diversi diagrammi (consultare fig. 2.2).

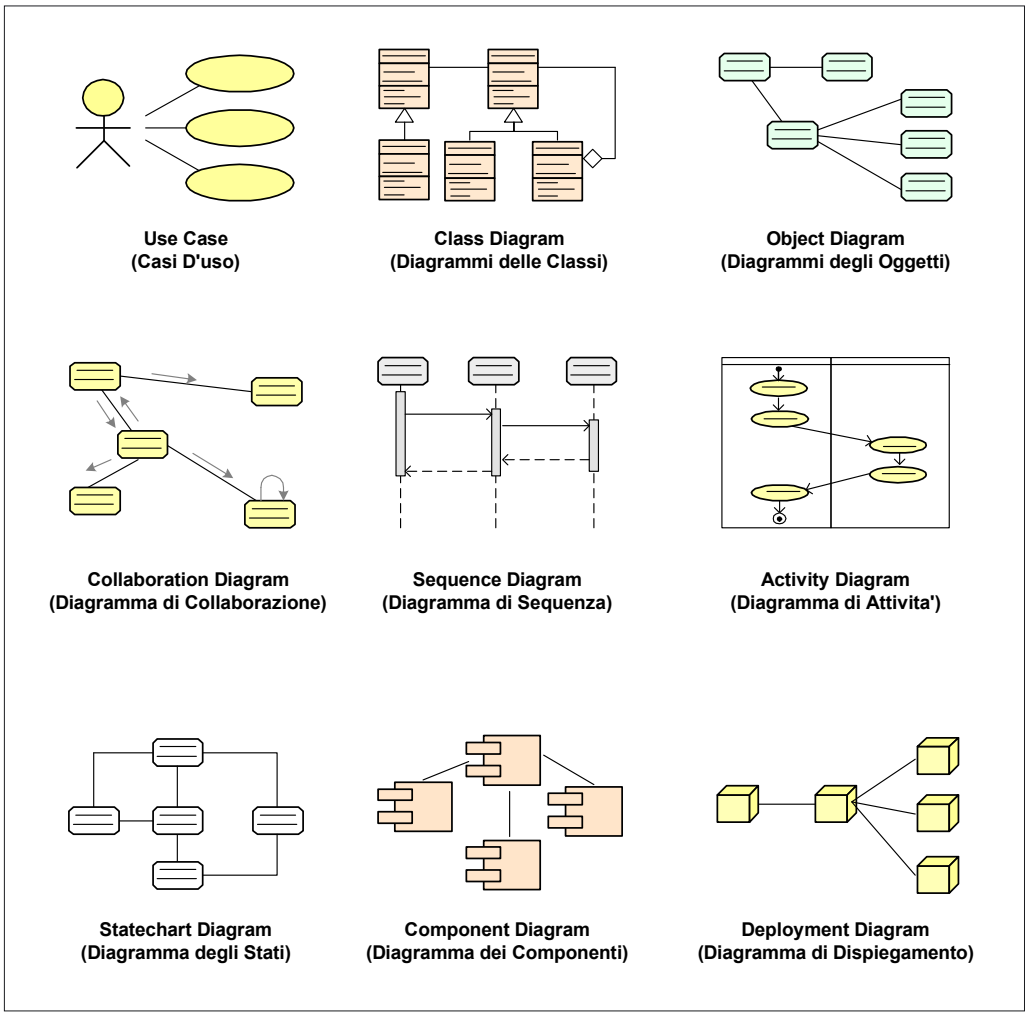
- **Use Case Diagram** (diagramma dei casi d'uso, da non confondersi con la relativa vista);
- **Class Diagram** (diagramma delle classi);
- **Object Diagram** (diagramma degli oggetti).

Diagrammi di comportamento:

- **Statechart Diagram** (diagramma degli stati);
- **Activity Diagram** (diagramma delle attività).

Diagrammi di interazione:

Figura 2.2 — *Diagrammi dello UML.*



- **Sequence Diagram** (diagramma di sequenza);
- **Collaboration Diagram** (diagramma di collaborazione).

Diagrammi di implementazione:

- **Component Diagram** (diagramma dei componenti);

- **Deployment Diagram** (diagramma di "dispiegamento").

Da notare che la stessa tipologia di diagramma si presta ad essere utilizzata per la realizzazione di diversi modelli. Per esempio, la notazione dei diagrammi delle classi permette di realizzare i modelli a oggetti di business, di dominio, di analisi, di disegno, ecc.

Qualche lacuna…

Una prima lacuna individuata dall'autore utilizzando lo UML riguarda il disegno della base dati. Nel mondo dell'ideale non dovrebbero esserci troppi problemi: sia l'architettura software, sia il database dovrebbero essere OO, quindi mapping uno–a–uno o quasi. In queste circostanze il disegno della base dati deriva direttamente dal modello ad oggetti del dominio e quindi si presta a essere descritto attraverso i diagrammi delle classi.

Nella realtà però, i database relazionali sono ancora la stragrande maggioranza per tutta una serie di motivi: molto spesso si ha a che fare con sistemi legacy che — nei casi migliori — utilizzano database relazionali; non sempre i database OO sembrerebbero offrire performance paragonabili ai rispettivi database relazionali; esiste ancora una certa ignoranza e quindi timore nei confronti dei database OO, e così via. Comunque sia, il dato di fatto è che i database relazionali sono molto ricorrenti nei progetti reali, e da qui nasce la lacuna dello UML di non prevedere apposito formalismo per la descrizione del disegno logico e fisico della base di dati.

Sebbene non esista uno strumento realizzato ad hoc per la modellazione dell'organizzazione di database non OO è possibile adattare una serie di strumenti standard per tale utilizzo; in altre parole è possibile realizzare un profilo per questo scopo. Un esempio è fornito nell'appendice B.

Per quanto attiene invece la definizione di un processo formale per realizzare il disegno della base di dati non si tratta di una responsabilità a carico dello UML bensì dei processi di sviluppo del software.

Probabilmente l'autore verrà accusato di essere affetto da un virus di stranezza, o di realizzare software dell'età della pietra, eppure, nel 99% dei sistemi realizzati erano presenti uno o più moduli dedicati all'interfaccia utente.

I sistemi, incredibilmente, erogano servizi fruibili dagli utenti e, "stranamente", l'interazione avviene attraverso opportuni layer di interfacciamento. Si tratta dello strato più esterno: quello che però l'utente vede ed è in grado di valutare, tanto che spesso il giudizio che un utente si costruisce di un sistema è basato principalmente sulla relativa interfaccia utente, sulla sua accuratezza, sulla semplicità di utilizzo, ecc.

Ora viene ancora da farsi la solita domanda: perché lo UML non prevede formalismo dedicato alla descrizione dell'interfaccia l'utente? Eppure il relativo studio, qualora necessario, rappresenta uno dei manufatti di una certa importanza (si pensi a sistemi Internet di

commercio elettronico) da produrre e sottoporre al vaglio dell'utente. Probabilmente esisteranno altri strani fattori traversali che rendono difficile l'individuazione di uno standard.

A questo punto si ritiene divertente riportare una serie di aneddoti relativi al rapporto tra l'autore e la produzione di interfacce utente: *Errare humanum est, perseverare diabolicum*.
Nel primo progetto londinese l'autore temeva che gli inevitabili problemi di abilità linguistiche avrebbero potuto creare notevoli problemi; la situazione era più drammatica del solito: oltre al problema di parlare un *linguaggio* differente dal committente per via delle diverse competenze (cliente esperto di business con scarso bagaglio tecnico), era proprio la *lingua* a essere diversa! Che fare? Ecco la brillante — si fa per dire — idea: realizzare rapidamente e nottetempo un prototipo con uno strumento RAD quale ad esempio Borland Delphi o MS Visual Basic, da utilizzarsi come piattaforma di dialogo. All'inizio sembrava che l'idea fosse veramente buona. Il prototipo, vera primadonna, era al centro di tutti i meeting e forniva un valido ausilio anche al cliente per capire di cosa avesse effettivamente bisogno: si richiedeva di aggiungere informazioni supplementari, di modificarne altre, di variare le dimensioni di alcuni campi, di modificare le business rule ecc. Tutto bene fino al momento in cui il prototipo fu giudicato sufficientemente "rispondente". A questo punto ci si accorse che nella mente del cliente si era drammaticamente instaurata la malsana idea secondo la quale il sistema reale era quasi pronto: bastava "riciclare" e sistemare quanto presente nel prototipo per poi consegnare. In altre parole il cliente si aspettava di avere la versione finale del sistema nell'arco di poco più di un mese! A quel punto la catastrofe si era consumata: ci vollero non pochi meeting per far comprendere ciò che in altri ambienti dell'ingegneria è abbastanza ovvio: un prototipo non è un sistema reale… Chi potrebbe pretendere di camminare sul plastico di un ponte o di un centro commerciale? Chissà come mai la solita vocina dice che ci sono manager i quali farebbero anche questo.
Secondo progetto londinese: memori di quanto accaduto con il progetto precedente, questa volta si tentò con MS PowerPoint: caspita, tutti sanno che si tratta di un prodotto per le presentazioni e non per lo sviluppo del software… l'autore comincia a odiare la vocina che gli urla dentro. Se da un lato non si corse quindi il rischio di ingenerare nell'utente la spaventevole idea che il prototipo fosse quasi il sistema finale, dall'altro non si riuscì — a differenza del caso precedente — a evidenziare le varie procedure del business, la navigazione della GUI, i vari dettagli e così via. In altre parole l'utilità della presentazione realizzata con MS PowerPoint fu molto relativa, a dir poco.
Terzo tentativo… Il problema era il seguente: il prototipo realizzato in Delphi o MS Visual Basic risultava molto allettante, ma proprio nessuno, voleva più correre il rischio di ingenerare nel cliente strane idee… Cosa fare? La soluzione venne all'autore vedendo un negozio di giocattoli: acquistare un bel modellino in scala del famoso London Bridge. A questo punto si realizzò l'ennesimo prototipo in Delphi, ma questa volta l'autore portò con sé nei vari meeting il ponte giocattolo non perdendo alcuna occasione per indicarlo al fine di evidenziare l'oggetto delle varie analisi: un prototipo, solo un prototipo, nient'altro che un prototipo.

Utilizzo dello UML nei processi di sviluppo

Sebbene trattare una materia così complessa e fondamentale come quella dei processi di sviluppo esuli decisamente dagli obiettivi del presente libro — ci vorrebbe almeno un libro solo per questo argomento — si è ritenuto altresì indispensabile prendere in considerazione un processo di riferimento al fine di illustrare in maniera più concreta l'utilizzo dello UML.

Spesso alcuni tecnici tendono a confondere lo UML con il processo di sviluppo: come ripetuto più volte lo UML è "solo" un linguaggio di modellazione e come tale si presta a rappresentare i prodotti generati nelle varie fasi di cui un processo è composto. Pertanto lo UML, al contrario dei processi, non fornisce alcuna direttiva (cosa fare, quali attività svolgere, quando, chi ne detiene la responsabilità e così via) su come fare evolvere il progetto attraverso le varie fasi (si veda il Capitolo 1) così come non specifica quali sono i manufatti da produrre, chi ne è responsabile ecc.

Nel momento in cui viene scritto il presente libro, non esiste un processo standard universalmente accettato né tantomeno esiste un accordo sulle varie fasi e sui relativi nomi. Addirittura non esiste una visione completamente concordante tra il libro dei Tres Amigos *The Unified Software Development Process* ([BIB08]) e il processo proposto dalla Rational, il RUP (Rational Unified Process), che ne rappresenta una rielaborazione. Sembrerebbe comunque che il processo proposto dalla Rational (RUP) stia di fatto prendendo sempre più piede nelle varie aziende seppur con varie difficoltà non sempre ingiustificate.

> Lavorando come liberi professionisti può anche accadere di dover inventare nuovi nomi per indicare alcune fasi e/o modelli (come per esempio *Detailed Design Model*, modello di disegno di dettaglio), al fine di evitare con il rifacimento del lavoro di urtare la suscettibilità dei colleghi… Come dire che, se un manufatto presenta seri problemi però è stato realizzato da una persona importante — magari dal Chief Architect — allora è meglio escogitare un nuovo nome per il rifacimento piuttosto che pestare i piedi di qualche collega.

I processi devono essere sia adattati alle singole organizzazioni in funzione di molti parametri, come per esempio le dimensioni, lo skill tecnico, le diverse figure professionali disponibili e così via, sia dimensionati rispetto alle caratteristiche dei progetti: probabilmente il processo dimostratosi efficace per la costruzione di un grattacielo potrebbe risultare non altrettanto efficace per realizzare una copertura per le macchine.

> Da qualche tempo a questa parte si assiste alla graduale affermazione di una nuova figura professionale definita "ingegnere di processo" (*Process Engineer*) cui compiti sono selezionare il processo più idoneo in funzione dell'azienda e al progetto da realizzare, adattarlo alle peculiarità del contesto, ecc. Chiaramente si tratta di un compito molto complesso, in cui è necessario considerare molte variabili aleatorie, non ultima lo skill a disposizione...
> Sebbene per decenni i vari "lavoratori" siano stati considerati attori, componenti sostituibili

del sistema di sviluppo, solo recentemente si è cominciato a comprendere come in realtà le cose non siano così. Come dire: un processo di sviluppo del software per essere prevedibile necessita di componenti a comportamento altrettanto prevedibile. A. Cockburn addirittura si esprime in termini di processi incentrati sulle persone (*people-centric*).

In queste situazioni di mancanza di standard, gli autori professionali suggeriscono al lettore di accendere il proprio cervello — *switch on the brain…* — e di cercare di prendere tutto ciò che viene percepito positivamente e confacente al proprio contesto, eventualmente arrangiandolo in funzione delle proprie esigenze, conferendo eventualmente un'importanza relativa a ciò che viene ritenuto meno importante. Lo stesso processo proposto dai Tres Amigos, applicato fedelmente, probabilmente richiederebbe il triplo del tempo e delle risorse effettivamente necessarie. In tal caso gli stati d'ansia dei vari manager potrebbero risultare giustificabili.

Figura 2.3 — *Nell'immagine è mostrata una matrice che indica come le viste e i diagrammi dello UML sono impiegati per la produzione dei principali manufatti previsti da un generico processo di sviluppo del software. Per questioni di completezza, è stata riportata una colonna etichettata "Documenti" atta a mostrare manufatti importanti per il processo di sviluppo del software, ma che non si prestano ad essere prodotti per mezzo dello UML. Infine, i modelli visualizzati in nero rappresentano quello fondamentali per la determinata fase, mentre quelli visualizzati in grigio risultano opzionali.*

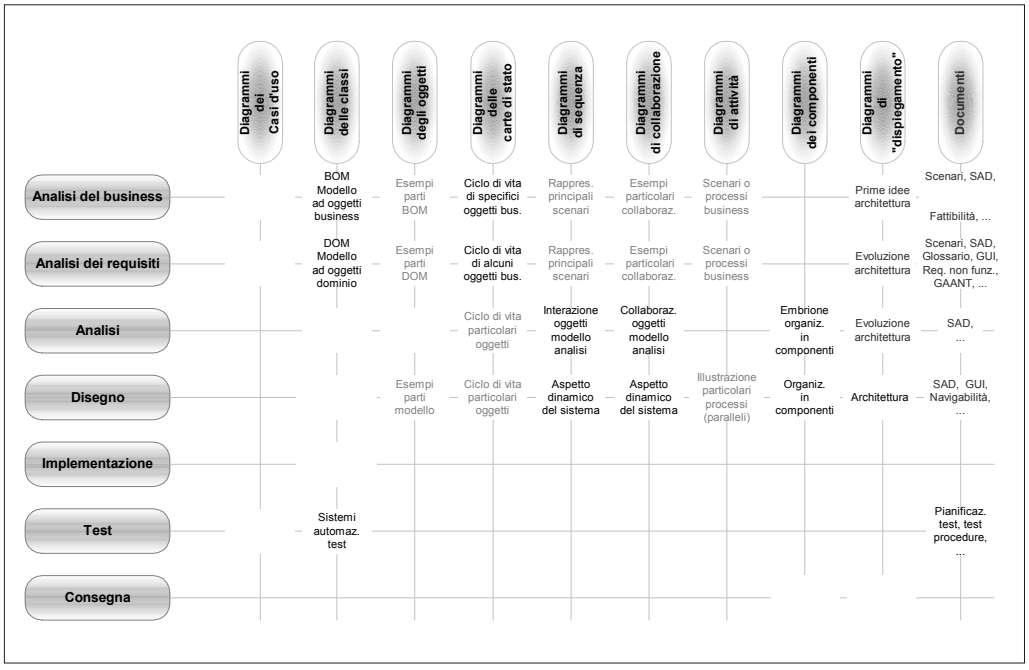


Figura 2.4 — *Utilizzo dei diagrammi UML per produrre i modelli di cui un processo è composto. I modelli evidenziati in figura non corrispondono necessariamente a una successione temporale del processo di sviluppo: molte fasi, che in prima analisi potrebbero considerarsi terminali o in stretta successione, vengono avviate il prima possibile al fine di massimizzare il parallelismo (si veda fig. 2.5). Ciò è utile non solo per logiche legate all'ottimizzazione dell'utilizzo delle risorse, ma anche per aumentare la qualità: ricevere, prima possibile, il feedback da diverse prospettive del modello. Per esempio, le primissime versioni del modello dell'architettura, tipicamente, sono progettate parallelamente alla fase di analisi dei requisiti, al fine di bilanciare i diagrammi dei casi d'uso nel contesto dell'architettura. Si tenga presente che obiettivo del processo presentato è semplicemente avere un riferimento atto a illustrare l'impiego dello UML nell'ambito di un processo organico di sviluppo. Il primo diagramma mostrato nel modello di analisi rappresenta una versione del diagramma delle classi che utilizza particolari stereotipi.*

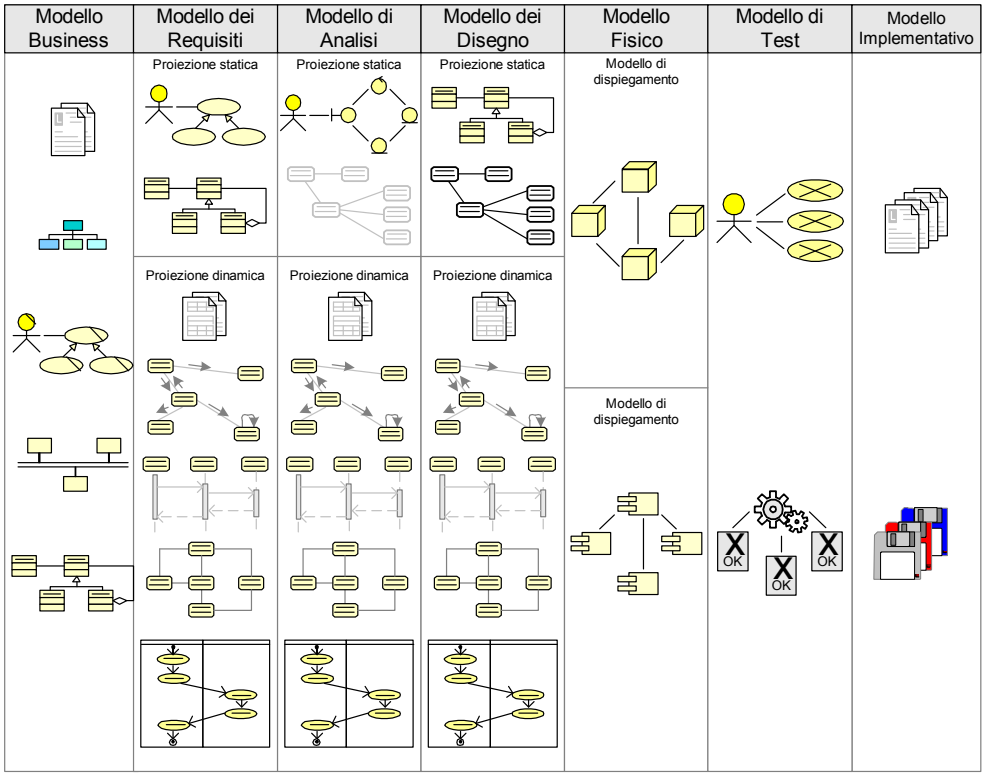
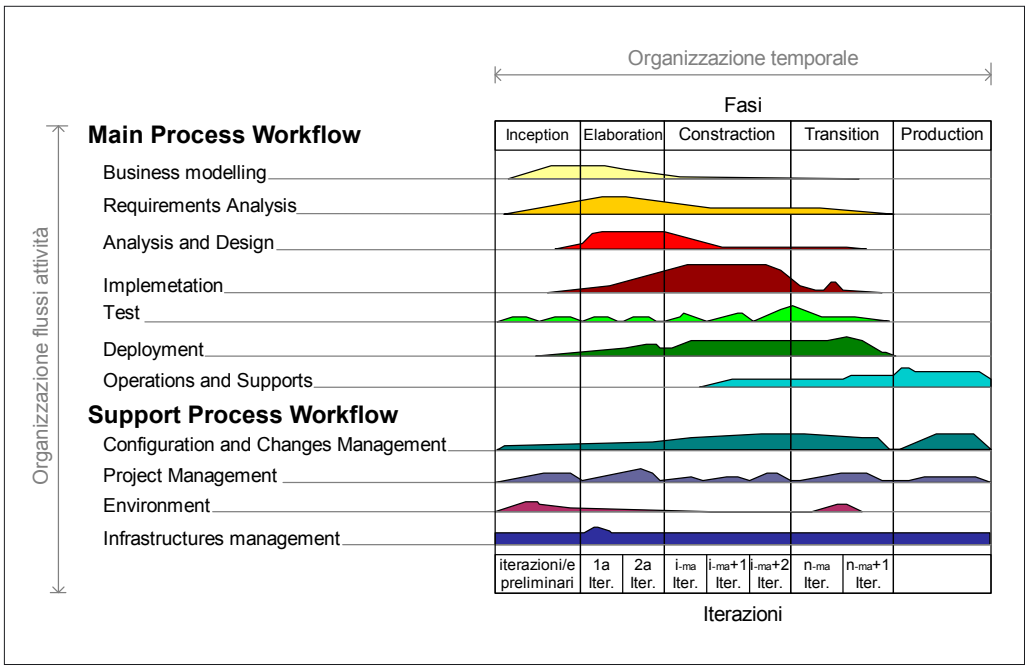


Figura 2.5 — *Ciclo di vita di un processo di sviluppo del software. Il diagramma in figura è una rielaborazione del processo denominato Enhanced Unified Process (EUP) proposto dalla Ronin Internation Inc.*



Nella fig. 2.3 è mostrato sinteticamente come i diagrammi dello UML possano essere utilizzati per produrre i modelli previsti da un generico processo per lo sviluppo del software.

Visti gli obiettivi del libro, il processo illustrato di seguito viene presentato focalizzando l'attenzione sui prodotti generati piuttosto che sul processo stesso. In ogni modo, cercando di interpolare i vari processi, con particolare riferimento al processo proposto dai soliti Tres Amigos, una schematizzazione abbastanza plausibile potrebbe essere quella in fig. 2.4. Come emerge chiaramente dal diagramma , diversi modelli prevedono due proiezioni: una statica ed una dinamica.

Per ciò che concerne la prima, a seconda della fase in cui si trova, è possibile impiegare le seguenti tipologie di diagrammi: casi d'uso, classi, oggetti, componenti, dispiegamento e così via.

Per ciò che concerne la proiezione dinamica, è possibile descriverla (sempre in funzione della fase del processo oggetto di studio) attraverso il linguaggio naturale (più oppor-

tuno nelle primissime fasi), con appositi modelli e/o schede, mediante gli appositi diagrammi dello UML (sequenza, di collaborazione, delle attività e degli stati).

L'evoluzione logica del progetto avviene secondo un approccio classico denominato a cascata: si passa da una fase a quella successiva e i manufatti prodotti in una fase forniscono l'input delle seguenti. Qualora si evidenzino delle lacune (una fase è stata conclusa prematuramente) o errori, è necessario ripercorrere il processo a ritroso fino a raggiungere la fase che permette di colmarla. Chiaramente gli effetti generati dalle correzioni sono proporzionali al numero di fasi a ritroso in cui è necessario risalire: più si risale e maggiore è l'onere necessario per aggiornare i manufatti prodotti.

Qualora si utilizzi un approccio iterativo e incrementale, l'intero processo viene suddiviso in sottoprogetti e quindi l'evoluzione avviene sulla base di opportune iterazioni. In ciascun sottoprogetto sono presi in considerazione un certo numero di casi d'uso o di loro frazioni (*scenario*).

Man mano che si procede verso fasi più tecniche del processo, risulta fortemente consigliabile cercare di utilizzare modelli sempre più formali, come per esempio i diagrammi dello UML.

Per esempio mentre nel modello dei requisiti è del tutto legittimo ricorrere a template per descrivere gli scenari dei casi d'uso, lo sarebbe di meno nel modello di disegno ove è consigliabile utilizzare i diagrammi dello UML dedicati alla descrizione del comportamento dinamico (per esempio Sequence, Collaboration, Activity, ecc.).

Gli stessi diagrammi dovrebbero possedere un livello di astrazione decrescente (aumento del livello di dettaglio) procedendo nelle fasi più "interne" dello sviluppo del sistema: un diagramma di sequenza utilizzato nel modello dei requisiti dovrebbe possedere un livello di astrazione molto elevato, mentre utilizzato nel modello di disegno dovrebbe essere molto dettagliato.

Lo UML può essere efficacemente paragonato all'insieme di attrezzi presente, per esempio, in una falegnameria. Ci sono tutta una serie di utensili (i diagrammi) ognuno dei quali risulta particolarmente indicato per compiti specifici, mentre del tutto inadeguato per altri. Volendo avvitare una vite verosimilmente il cacciavite risulta lo strumento più consono, mentre per conficcare un chiodo in una superficie, probabilmente, lo sarebbe di meno (può capitare però che determinati capi progetto non abbiano alcun problema a richiedere di infilare chiodi utilizzando i cacciaviti).

> Sebbene sia possibile individuare processi più formali o più affini alle necessità delle singole organizzazioni e ai progetti, la schematizzazione proposta in figura dovrebbe risultare piuttosto credibile; eventualmente si potrebbero escogitare nomi diversi, ulteriori manufatti da produrre e così via.

Brevemente (ogni fase verrà ripresa nei capitoli successivi) i vari modelli da produrre come risultato delle varie fasi di un processo sono quelli riportati di seguito.

Modello Business

Si tratta di uno dei primissimi modelli da realizzare ed è costituito da due proiezioni: statica e dinamica, ciascuna delle quali è rappresentata attraverso appositi modelli.

L'obiettivo è iniziare a studiare l'area business in cui operare. Pertanto è necessario capire cosa bisognerà realizzare (requisiti funzionali e non), quale contesto bisognerà automatizzare (studiarne struttura e dinamiche e, a tal fine, è molto utile produrre il Business Object Model, modello ad oggetti del business), comprendere l'organigramma dell'organizzazione del cliente, valutare l'ordine di grandezza del progetto, individuare possibili ritorni dell'investimento per il cliente, eventuali debolezze, potenziali miglioramenti, iniziare a redigere un glossario della nomenclatura tecnica al fine di assicurarsi che, nelle fasi successive, il team di sviluppo parli lo stesso linguaggio del cliente, e così via. È necessario effettuare il famoso studio di fattibilità, redigere una prima versione dei famosi requisiti del cliente sia attraverso la versione iniziale del modello dei casi d'uso (denominato appunto business) sia per mezzo di documenti elettronici (lista dei potenziali requisiti), ecc. Molto spesso viene avviata anche la produzione del documento dell'architettura software del sistema (SAD, Software Architect Document) aggiornato durante tutto il ciclo di vita del sistema.

In questo contesto è di fondamentale importanza riuscire a instaurare una corretta piattaforma di intesa con il cliente: si tratta di una fase non eccessivamente formale in cui tutte le convenzioni utilizzate risultano ben accette purché concorrano effettivamente ad instaurare un dialogo costruttivo con il cliente.

Nel mondo ideale gran parte di questo modello potrebbe essere realizzato attraverso i vari diagrammi messi a disposizione dallo UML… Però quanti manager (anche di società informatiche) sono in grado di capire modelli formulati attraverso lo UML? L'interrogazione retorica potrebbe essere posta anche in altri termini: quanti dei vostri manager sono in grado di comprendere modelli espressi per mezzo dello UML? Si provi allora a immaginare la risposta pensando ai clienti i quali, per definizione, possiedono limitate conoscenze informatiche. In molti processi non è prevista la presente fase: si parte direttamente con il modello dei requisiti. In questo contesto si è deciso invece di riportarla sia perché si fa sempre in tempo a eliminarla dal proprio processo, sia perché, tipicamente, prima di avviare il processo di sviluppo vero e proprio è necessario superare una prima attività relativa allo studio di fattibilità, al fine di accertarsi che esista una piattaforma d'intesa economica con il cliente: chissà come mai gli studi di fattibilità non producono mai un esito negativo.

Modello dei requisiti

Questo modello (spesso detto anche di dominio) è prodotto a seguito della fase comunemente detta analisi dei requisiti (Requirements Analysis) il cui scopo è produrre una versione più tecnica dei requisiti del cliente evidenziati nella fase precedente. Tipicamente, il modello dei casi d'uso elaborato in questa fase (prodotto da System Analyst e Use

Case Specifier) è ancora utilizzato come strumento di dialogo con il cliente: gli Use Case prodotti risultano ancora parlare un linguaggio assolutamente compatibile con il cliente, sebbene il modello cominci ad incorporare feedback provenienti dall'architettura. Spesso è prevista un'ulteriore versione in cui gli aspetti tecnici recitano un ruolo importante. In questo caso ci si riferisce al modello con i termini di System Use Case Model. In questa fase è importante far confluire eventuali vincoli presenti e tenere presenti i cosiddetti requisiti *non funzionali* del sistema (performance, affidabilità, disponibilità, scalabilità, e così via). Sebbene alcuni di essi si prestino ad essere inclusi direttamente nei vari Use Case: vengono raggruppati in appositi documenti (SRS, Supplementary Requirements Specifications, specifica dei requisiti supplementari).

Altri prodotti particolarmente importanti da produrre sono: modello ad oggetti del dominio (rappresentazione dei dati manipolati limitatamente al dominio oggetto di studio), stima dei costi e dei tempi di sviluppo dell'intero sistema, assegnazione delle priorità (e quindi suddivisione del processo in iterazioni), primissime versioni/idee relative alla GUI (Graphical User Inetrface), rielaborazioni dei vari Object Model, del SAD, e così via.

Modello di analisi

Come è lecito attendersi, questo modello viene prodotto come risultato della fase di analisi i cui obiettivi sono:

1. produrre una versione dettagliata e molto tecnica della Use Case View attraverso opportuni diagrammi delle classi, accogliendo direttive provenienti dalle versioni disponibili del disegno dell'architettura del sistema, che a loro volta dipendono da questo modello. Si assiste alla graduale perdita di generalità dei modelli e si passa a un linguaggio più vicino agli sviluppatori e quindi dotato di una notazione più formale. L'obiettivo del modello varia: a questo punto il fruitore principale è il proprio team tecnico e non più il cliente. Si realizza, pertanto, una primissima versione del modello di disegno, generalmente, costituita dalle entità effettivamente presenti nel dominio, interfacce e processi di controllo. In questa fase, tipicamente, non si ha particolare interesse a introdurre ottimizzazioni o razionalizzazioni.

2. analizzare dettagliatamente le business rule da implementare.

3. qualora necessario, si revisiona il prototipo (o comunque una descrizione) dell'interfaccia utente, il SAD, ecc.

In molti processi (come il RUP per esempio) la fase di analisi viene aggregata a quella di disegno.

Modello di disegno

Anche in questo caso il modello di disegno è il prodotto della omonima fase, in cui ci si occupa di plasmare il sistema, trasformare i vari requisiti (funzionali e non funzionali) forniti nel modello di analisi in un modello direttamente traducibile in codice. A tal fine è necessario costruire l'infrastruttura intorno ai diagrammi delle classi prodotti nella precedente fase. Sempre in questo stadio, è importante realizzare un modello completo dell'interfaccia utente e della relativa navigabilità. In sintesi, gli obiettivi della fase di disegno sono:

1. acquisire una profonda comprensione di problematiche relative ai requisiti non funzionali e ai vincoli dovuti ai linguaggi di programmazione, al riutilizzo di eventuali componenti, ai sistemi operativi coinvolti, a problematiche di distribuzione del carico di lavoro e alla conseguente possibilità di svolgere operazioni in parallelo, al sistema di gestione della base dati, alle tecnologie da utilizzare per l'interfaccia con l'utente, a quelle da utilizzarsi per la gestione delle transazioni, e così via.

2. realizzare un opportuno modello completo in grado di fornire tutte le direttive necessarie per l'implementazione del sistema.

3. stabilire il piano per la decomposizione intelligente dell'implementazione del sistema, al fine di massimizzarne l'evoluzione parallela dei vari sottosistemi.

4. disegnare il prima possibile le interfacce esistenti tra le varie componenti del sistema.

5. disegnare opportuni modelli al fine di investigare sulle possibili soluzioni attuabili.

In genere non è necessario eccedere nel disegno del sistema dettagliando tutte le classi di un package e/o tutti i metodi. Qualora il disegno sia sufficiente per la relativa implementazione e non sia fonte di ambiguità, è del tutto accettabile procedere alla codifica senza sprecare tempo. Tipicamente esistono due versioni del modello di disegno, quella iniziale da fornire al team di sviluppo, detta di specifica, e la versione risultata dalla codifica, detta modello di implementazione. Chiaramente il livello di dettaglio a cui giungere è inversamente proporzionale al livello tecnico del proprio team di codificatori: al diminuire del secondo deve aumentare il livello di dettaglio; quando poi lo skill tende a zero, allora conviene implementare in proprio e inventarsi esercizi di stile da assegnare al proprio team. Poiché il modello di disegno è molto vicino all'implementazione del sistema, tipicamente risulta soggetto a correzioni, rifiniture e aumento del livello di dettaglio provenienti sia dall'implementazione del modello stesso sia da eventuale refactoring. Il reverse engineering, posto in questi termini, è del tutto accettabile.

Il round-trip (si modella, si implementa e quindi si ritocca il modello) è del tutto accettabile (d'altronde solo il codice è allineato con l'implementazione del sistema), mentre il reverse engineering da solo, nella mente dell'autore, è percepito come cosa da evitarsi e spesso del tutto inutile. Si pensi al caso in cui si utilizzino tecniche di *reflection* nel codice Java. Tra le varie caratteristiche, le classi di questo package permettono di creare oggetti specificando la classe di appartenenza per mezzo del nome impostato in una stringa. In altre parole si chiede di creare un oggetto non attraverso il classico costruttore `new`, ma specificando il nome (stringa) della classe di appartenenza come parametro di un opportuno metodo. In tal caso i vari tool non sono ancora in grado di ricostruire le dipendenze tra oggetti di questo tipo, e quindi il reverse engineering non mostrerebbe alcun legame tra le classi che incorporano tecniche di reflection e quelle specificate nelle relative stringhe.

Modello fisico

Il modello fisico, a sua volta, è composto essenzialmente da due modelli: Deployment e Component. Non si tratta quindi del prodotto di una fase ben specifica, bensì dei risultati di rielaborazioni effettuate in diverse fasi. Il modello dei componenti mostra le dipendenze tra i vari componenti software che costituiscono il sistema. Come tale la versione finale di questo modello dovrebbe essere il risultato della fase di disegno. Non è infrequente il caso in cui però se ne realizzino versioni preliminari in fasi precedenti con l'obiettivo di chiarire i servizi esposti dai vari sottosistemi e quindi di consentire l'evoluzione parallela dei sottosistemi. Il modello di dispiegamento (Deployment) descrive la distribuzione fisica del sistema in termini di come le funzionalità sono ripartite sui nodi dell'architettura fisica del sistema. Si tratta quindi di un modello assolutamente indispensabile per le attività di disegno ed implementazione del sistema. In genere le primissime versioni del modello di dispiegamento sono create nella fase di elaborazione dei requisiti per consentire di contestualizzare e bilanciare tra loro i vari modelli: gli Use Case perdono di genericità e incorporano riferimenti architetturali. Il disegno dell'architettura fisica dovrebbe essere realizzato in funzione del modello dei casi d'uso, ossia dei servizi che il sistema dovrà fornire, calati nel contesto dei requisiti non funzionali. In realtà, si cerca di partire da soluzioni architetturali mostratesi efficaci in precedenti progetti (pattern architetturali) eventualmente revisionati in funzione di alcune particolari necessità.

Modello di test

Nella produzione di sistemi, con particolare riferimento a quelli di dimensioni medie e grandi, è opportuno effettuare test in tutte le fasi. Eventuali errori o lacune vanno eliminate prima possibile — sarebbe ancor più opportuno prevenire — al fine di neutralizzare l'effetto delle relative ripercussioni sul sistema. Nella costruzione di un ponte, per esempio, è evidente che è opportuno verificare le varie colonne sia dopo la relativa pro-

gettazione sia subito dopo la costruzione. Non si vuole di certo correre il rischio di riscontrare manchevolezze dopo avervi posato la sezione relativa alla strada. Quindi è opportuno effettuare test prima di dichiarare conclusa ciascuna fase. In questo contesto si fa riferimento a uno specifico modello atto a verificare la rispondenza di ogni versione del sistema frutto di un'appropriata iterazione. Il modello di test dovrebbe essere generato non appena disponibile una nuova versione stabile dei casi d'uso. Pertanto dovrebbe essere elaborato o dopo il modello dei requisiti o dopo quello di analisi. Chiaramente è opportuno verificare tutti i manufatti; però meritano particolare attenzione quelli che in ultima analisi verranno eseguiti: classi, package, componenti, sottosistemi, script e così via. Nel realizzare il modello di test è necessario:

1. pianificare i test richiesti a seguito di ciascuna iterazione. In questo contesto si parla di test di integrazione (Integration Test) e test di sistema (System Test). I primi vengono eseguiti ogni qualvolta si integrano diverse parti del sistema al fine di verificarne la corretta "collaborazione". I secondi si effettuano alla fine dell'iterazione e servono per accertarsi che le funzionalità oggetto della iterazione siano rispondenti agli Use Case/scenari presi in considerazione dalla stessa.

2. disegnare e realizzare i test attraverso i Test Case che specificano cosa verificare e le Test Procedure che illustrano come eseguire i test. Quando possibile, sarebbe opportuno creare componenti eseguibili (test component) in grado di effettuare i test automaticamente.

3. Integrare i test necessari al termine dell'attuale iterazione con quelli eventualmente prodotti per le iterazioni precedenti.

Il modello di test quindi specifica come eseguire i vari test (che ovviamente vanno eseguiti nella relativa fase) e come tenere traccia dei risultati e del comportamento. L'obiettivo è essere in grado, in caso di malfunzionamenti, di fornire opportuno feedback per la rigenerazione dell'errore.

Modello implementativo

Questo modello è frutto della fase di implementazione il cui obiettivo è tradurre in codice i manufatti prodotti nella fase di disegno e quindi realizzare il sistema in termini di componenti: codice sorgente, file XML, file di configurazione, script, ecc. In particolare è necessario:

- pianificare l'integrazione necessaria tra i sottosistemi prodotti durante le varie iterazioni (nel caso classico in cui il processo utilizzi un approccio iterativo e incrementale);

- distribuire il sistema come prescritto dal Deployment Diagram (mapping dei componenti eseguibili sui nodi del sistema);

- codificare il modello di disegno in classi. A seconda del livello di completezza del disegno, tipicamente accade che l'implementazione richieda un aumento del dettaglio del disegno catturato attraverso il reverse engineering;

- realizzare dei package, atti a eseguire gli Unit Test, ossia a verificare il corretto funzionamento di sottosistemi prima di renderli disponibili per l'integrazione.

Considerazioni finali

Dalla descrizione del processo sono stati trascurati volutamente tutti i manufatti — tra l'altro importantissimi — relativi alla pianificazione del progetto, in altre parole quelli più a carattere manageriale. Si tratta di una sorta di processo parallelo presente in tutte le fasi del ciclo di vita del software: pianificazione della gestione dei requisiti, assegnazione delle priorità agli Use Case con conseguente pianificazione delle iterazioni, analisi dei rischi — anche questa concorre alla pianificazione delle iterazioni —, continua elaborazione e monitoraggio del SDP (Software Development Plan, piano di sviluppo del software) e così via. Chiaramente la buona riuscita di un progetto dipende molto dalla sua corretta pianificazione, anche se taluni manager credono che ciò sia il risultato della padronanza del tool Microsoft Project: che cosa importano i vari numeri (giorni/uomo) o la pianificazione delle attività? L'importante è disporre di un bel diagramma da appendere al muro…

Altri manufatti da produrre nelle varie fasi sono i documenti con le linee guida (Guideline) con indicazioni sul modo di realizzare i vari manufatti stessi. In altre parole si tratta di prodotti funzionali al processo, da riutilizzarsi anche per eventuali progetti futuri. Per esempio è necessario realizzare le guideline relative agli Use Case (quali stereotipi utilizzare, come descrivere i vari scenari, ecc.). Ogni progetto ha delle proprie problematiche specifiche e quindi si vuole che i vari manufatti alla fine di ogni fase risultino consistenti.

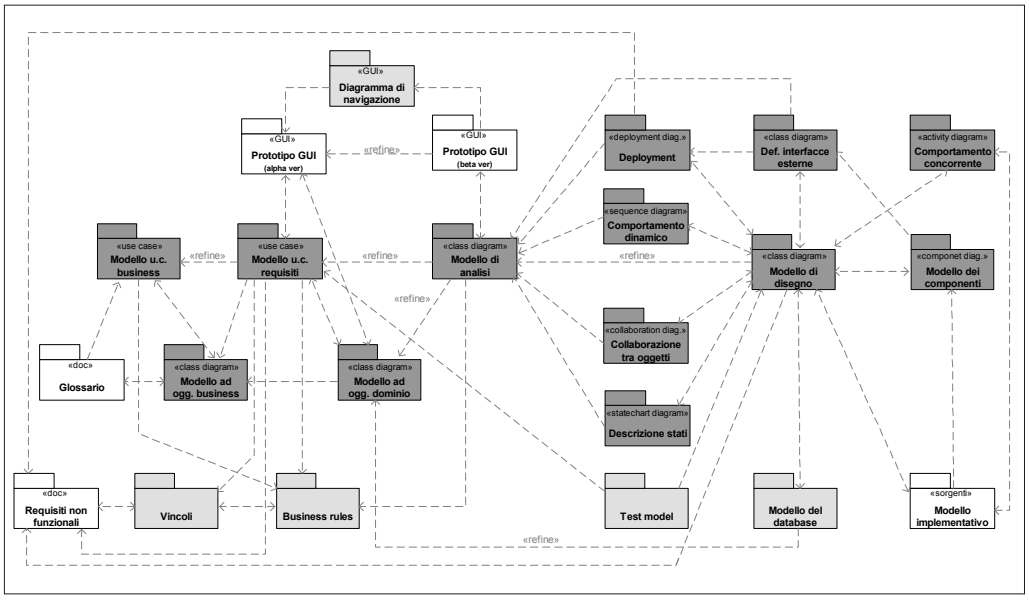
> La critica che più di frequente viene avanzata ai processi formali è di essere eccessivamente burocratici e *Document-Oriented*, tanto da essere addirittura definiti monumentali (Jim Highsmith). Come suggerisce lo stesso M. Fowler, spesso è necessario "mettere a dieta i processi", al fine di evitare la produzione di un enorme quantitativo di documentazione che, oltre a richiedere molto tempo per la realizzazione e l'aggiornamento, finisce, paradossalmente, per scoraggiare i tecnici più volenterosi. Come alternativa si assiste a tutto un fiorire di processi per così dire leggeri (Lightweight), più pragmatici, comunque migliori di "metodologie" completamente imperniate sulla "codifica e correzione". Processi basati su tutta una serie di decisioni a breve o brevissimo termine, validissimi per progetti

di piccole dimensioni, che però, applicati in sistemi di dimensioni maggiori, sono in grado di generale sistemi caotici e chiusi. In tali sistemi l'inserimento di nuove funzionalità implica acrobazie e pene indescrivibili nonché il terrore dell'intero team di sviluppo.

In fig. 2.6 viene riportato un altro diagramma che visualizza le dipendenze tra i vari manufatti prodotti durante un processo. Ancora una volta non sono stati presentati tutti i manufatti, ma solo quelli ritenuti più interessanti e confacenti ai fini della trattazione (il diagramma è già abbastanza complicato così). Alcuni sono stati condensati in uno stesso package e altri sono stati estrapolati in quanto ritenuti interessanti da visualizzare: si ricordi che l'obiettivo è evidenziare i manufatti realizzabili per mezzo dei meccanismi forniti dallo UML.

Nel diagramma di figura 2.6 la relazione di dipendenza (freccia tratteggiata) che unisce i vari package ha un significato piuttosto intuitivo: un aggiornamento dell'entità indipendente (quella puntata dalla freccia) genera revisioni ed eventualmente modifiche su quella dipendente. Per esempio se si apportano delle modifiche al modello di analisi, verosimilmente è necessario rivedere il modello di disegno.

Figura 2.6 — *Diagramma delle classi dei manufatti prodotti dal processo. (Si tratta di una rielaborazione Object Oriented di quello fornito da Scott Ambler nel libro Process Pattern ([BIB12]).*



Non sempre la relazione di dipendenza ha un legame così forte. Per esempio la variazione del modello di analisi non sempre genera modifiche in quello della base di dati, però, tipicamente, ne richiede almeno la revisione.
I vari package (in questo contesto sono collezioni di manufatti) assumono un colore la cui tonalità indica, per così dire, la connotazione UML: quelli evidenziati con una gradazione più scura possono essere specificati interamente attraverso i meccanismi dello UML, quelli a gradazione tenue possono esserlo solo parzialmente, mentre per quelli bianchi è necessario ricorrere a formalismi diversi, quali per esempio i classici documenti.
Per esempio il modello del database, nel caso in cui si consideri un database OO potrebbe essere specificato attraverso diagrammi delle classi e degli oggetti e quindi per mezzo dello UML, mentre negli altri casi è necessario realizzare un mapping tra tali diagrammi e altri più vicini alla natura del database manager con tecniche non presenti nello UML.
I vari package del diagramma tengono conto della proprietà transitiva della dipendenza. Per esempio poiché il modello del database dipende dal modello di disegno che a sua volta dipende dal modello concettuale, è evidente che anche il modello del database dipenda da quello concettuale. Ciononostante, alcune volte, si è deciso di evidenziare tale dipendenza perché, teoricamente, diversi modelli (in questo caso disegno e database) dovrebbero venir sviluppati separatamente e in parallelo per poi essere riallineati.

In fig. 2.7 viene riportata un'altra proiezione del processo di sviluppo focalizzata questa volta sulle attività da svolgere. Il formalismo utilizzato è quello degli Activity Diagram (diagrammi di attività). L'interpretazione è abbastanza intuitiva: i rettangoli arrotondati indicano attività da svolgere, i rombi evidenziano la divisione del flusso mentre le linee orizzontali doppie indicano che le attività incluse tra due successive possono essere eseguite parallelamente.

Il diagramma si configura ancora come una overview: ogni attività dovrebbe essere illustrata per mezzo di un altro Activity Diagram di dettaglio. Ovviamente l'obiettivo è quello di fornire al lettore un'idea delle varie attività da svolgere nelle varie fasi, senza avere assolutamente la pretesa di essere esaustivi.

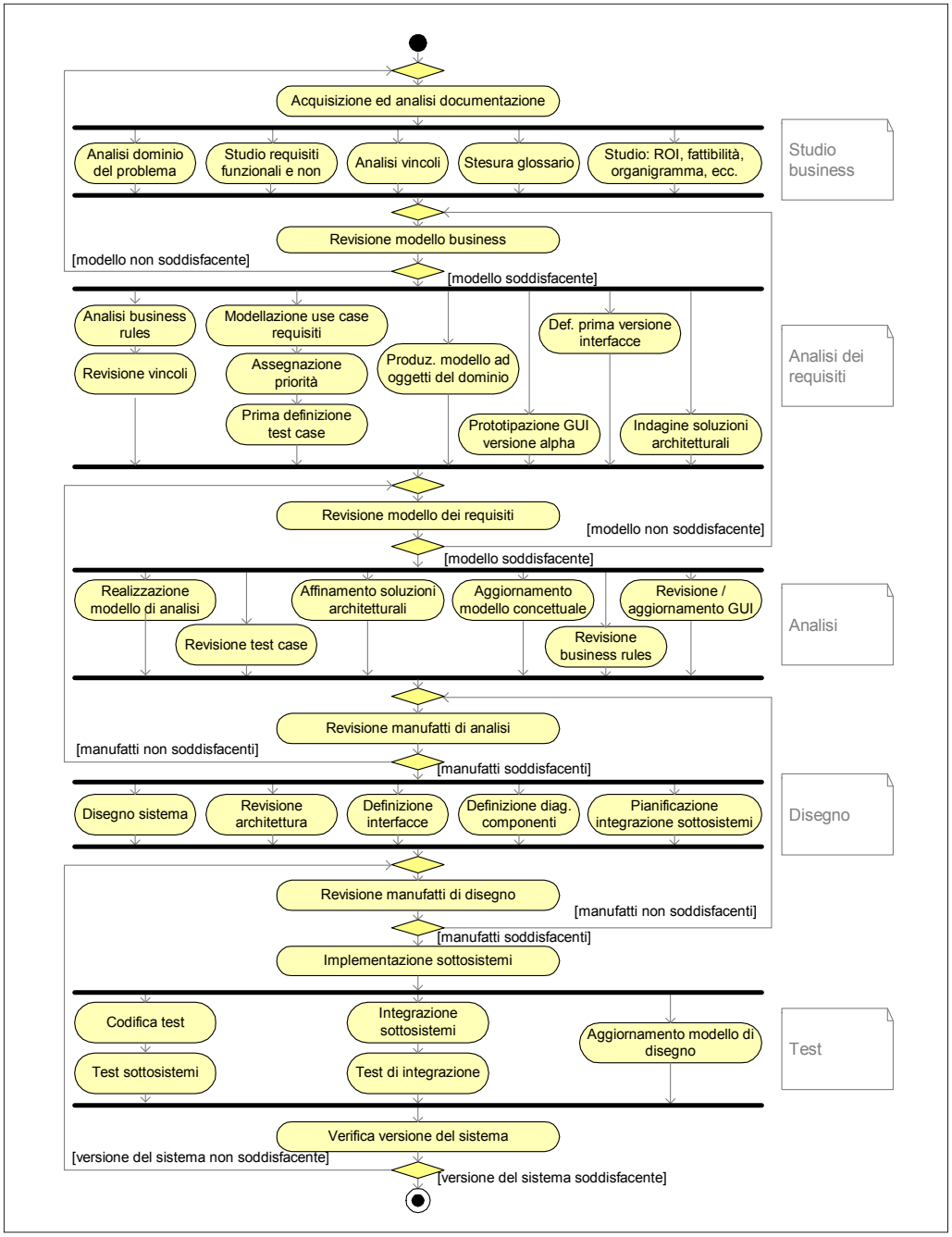
Nella rappresentazione del processo non si è data alcuna enfasi all'approccio iterativo e incrementale solo per evitare di complicare maggiormente il diagramma. Tipicamente le prime due o tre iterazioni (in funzione delle dimensioni del progetto) sono imperniate sulle prime due fasi: analisi del business e dei requisiti.

Le restanti iterazioni invece sono focalizzate sulla realizzazioni di versioni successive del sistema e quindi hanno come dominio le fasi di analisi, disegno, implementazione e test.

Sistemi di sistemi: il progetto che non ti aspetti

Per definizione tutti i progetti sono unici. Anche qualora si lavori nel settore dell'e-commerce, dove i *pattern* utilizzati sono sempre gli stessi, è prassi che ogni progetto abbia business rule del tutto esclusive, integrazioni specifiche, architetture di riferimento

Figura 2.7 — *Diagramma delle attività del processo.*



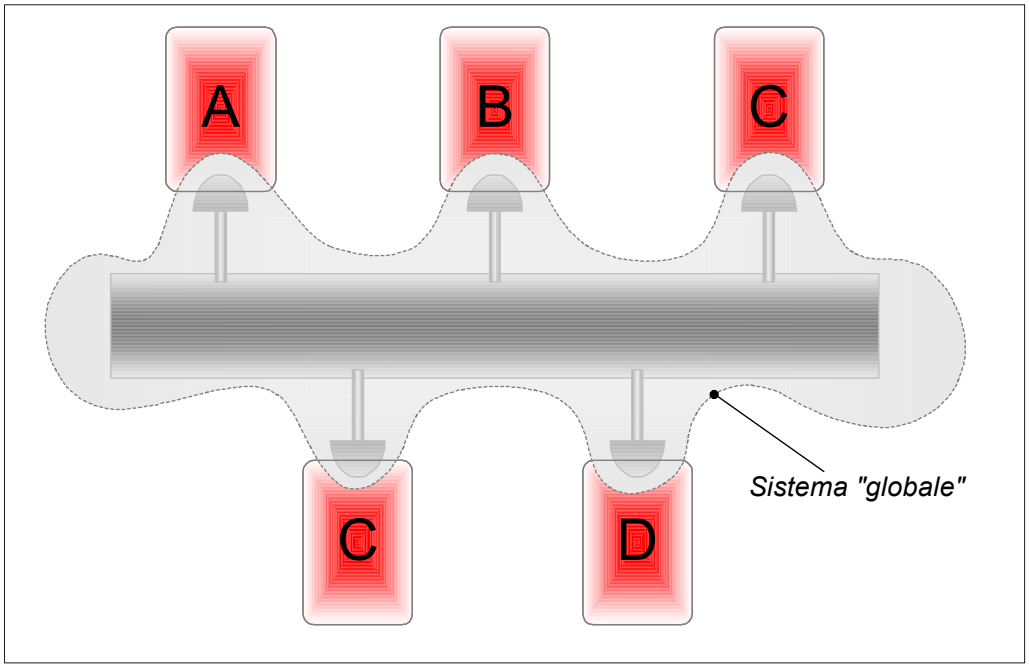
particolari e così via. Capita sempre insomma di dover affrontare problematiche particolari non presenti in nessun testo: a questa legge empirica non si sottraggono certo i processi di sviluppo.

Si prendano in considerazione grandi sistemi quali per esempio quelli delle banche. In questi contesti probabilmente è più opportuno affrontare il processo di sviluppo pensando il progetto in termini di sistemi di sistemi: c'è una serie di sistemi (front office, gestione conti correnti, sistema di banking on-line, treasury, gestione delle eccezioni, e così via) e il sistema globale con funzioni di collante (fig. 2.8).

La realizzazione del sistema finale di solito non prevede la realizzazione ex-novo di tutti i sistemi: sarebbe una follia. Verosimilmente, per la maggior parte di essi risulta "sufficiente" realizzare appositi strati di wrapping: business logic antica in una facciata moderna. In contesti di questo tipo probabilmente utilizzare un processo "tradizionale" potrebbe causare diversi problemi.

Il fatto è che ormai i progetti non sono più come quelli di un ventennio fa in cui c'era tutto da fare e quindi era possibile dividere nettamente le varie aree: nelle organizzazioni moderne i vari impiegati crescono con l'infrastruttura informatica fusa nel loro business; essa diventa parte integrante anche della loro forma mentis.

Figura 2.8 — *Schematizzazione di un sistema di sistemi.*



Pertanto, affrontare il processo realizzando dapprima il modello dei casi d'uso a livello business, poi requirement, il modello di analisi, ecc. potrebbe risultare alquanto complicato. Per esempio i clienti — in questo contesto gli esperti dell'area business della banca — sono generalmente abituati a pensare alle varie funzionalità in termini di collaborazione tra i sistemi informatici presenti; sanno esattamente, per esempio, quando termina il workflow a carico del front office e quando inizia quello relativo al back office; conoscono di quali altri sistemi si avvale la collaborazione (revaluation, messaggistica, market rates, ecc.) e così via.

In queste situazioni, per quale motivo forzare il cliente di astrarsi dalla suddivisione del sistema in sottosistemi? Anzi, magari fossero tutti così i clienti.

Cercare di applicare il processo in modo "accademico" potrebbe addirittura portare a realizzare modelli artificiosi e innaturali per gli stessi clienti. Altri problemi potrebbero venire dal tempo e dall'aumento del fattore di rischio dovuto all'affrontare globalmente anche solo la modellazione dei casi d'uso di un sistema di dimensioni così elevate.

Una buona idea potrebbe essere quella di sfruttare la percezione della scomposizione architetturale del sistema intrinseca nella concezione del cliente e quindi affrontare $n+1$ progetti con n uguale al numero dei sottosistemi esistenti e il fattore aggiuntivo dovuto al sistema globale di raccordo (l'integrazione).

Il rischio più preoccupante in cui si potrebbe incorrere con approcci di questo tipo, è realizzare un sistema in cui le varie parti non risultino facilmente integrabili tra loro. Ricorrendo però a particolari accorgimenti è possibile gestire questo rischio corrispondente alla mancata integrazione.

Alcuni suggerimenti potrebbero essere:

- realizzare la versione business considerando globalmente i vari processi — dall'inizio alla fine — senza preoccuparsi dell'organizzazione in sottosistemi, ossia focalizzando l'attenzione sulla sequenza delle attività da svolgere per erogare il servizio, senza però entrare nel dettaglio dell'allocazione delle varie attività ai sottosistemi;

- pianificare per tempo un "processo" parallelo per lo studio delle aree in comune tra sottosistemi (integrazione);

- definire, prima possibile, le interfacce tra i sottosistemi, e così via.

L'approccio descritto risulterebbe particolarmente valido in tutti quei contesti in cui il cliente possiede una chiara comprensione della suddivisione del flusso tra i vari sistemi ed è in grado di specificare le responsabilità e le funzionalità di ciascuno di essi.

In ultima analisi, dopo la definizione del modello del business, sarebbe necessario applicare, per ogni sottosistema, lo stesso processo di sviluppo. Ognuno di essi sarebbe

caratterizzato dal focalizzare l'attenzione su uno specifico sottosistema e dal considerare i restanti come entità esterne (ossia attori) con cui scambiare informazioni.

Decomponendo il modello generale dei casi d'uso, si verifica che determinati servizi, per essere realizzati, necessitano di attraversare più sottosistemi. Nei punti di attraversamento risiedono le varie interfacce. L'individuazione delle stesse è semplificata dalla successiva versione dei casi d'uso: focalizzando l'attenzione su di un sottosistema, quelli restanti diventano attori per lo stesso. L'associazione tra un "attore sottosistema" e quello oggetto di studio avviene rispettando una specifica interfaccia.

In queste tipologie di progetto, potrebbe risultare vantaggioso adeguare parte dell'organigramma alla struttura del sistema. Per esempio, potrebbe essere una buona idea suddividere il personale in tanti team di sviluppo quanti sono i diversi sottosistemi. Prevedere un architetto leader e un analista leader per ciascuno di essi. Ciò al fine di decentrare le varie responsabilità, aumentare la consistenza, creare aree di competenza, disporre eventualmente di team ridotti (costituiti per esempio da tutti i vari architetti leader) atti a studiare problemi relativi a più sistemi (come per esempio lo scambio di messaggi), a evidenziare e utilizzare componenti comuni e così via.

Questo approccio, che dovrebbe risultare naturale in progetti in cui sia preesistente un'organizzazione in sottosistemi, potrebbe essere utilizzato comunque proficuamente in tutti quei casi in cui il sistema da sviluppare risulti oggettivamente grande.

Situazioni di questo tipo sono facilmente riscontrabili dalle continue lamentele del team di sviluppo considerato in senso generale: architetti, programmatori, ecc. Le frasi tipiche sono: "il modello è troppo astratto", "servono maggiori dettagli", e così via. Il che non significa, necessariamente, che il modello non sia stato realizzato correttamente… Evidentemente però è necessario produrre un'altra versione del modello, che potrebbe ricollegarsi al caso del "sistema di sistemi".

Volendo procedere con questa tecnica, un'attività importante è quella di individuare correttamente i vari sottosistemi. Nel caso "canonico", ciò non era necessario semplicemente perché ci si trovava di fronte a una situazione predefinita, da tenere in conto. Ciò risulterebbe del tutto coerente con le direttive dei processi di sviluppo: tipicamente la versione *requirements* del modello dei casi d'uso prevede un'ulteriore versione detta *system* proprio perché si pone il modello del business nel contesto del sistema da realizzare e si accettano eventuali feed back provenienti dal disegno di architettura.

Di solito gli architetti esperti non hanno enormi difficoltà nel decomporre il sistema generale in sottosistemi. In ogni modo la tecnica generalmente utilizzata è, a tutti gli effetti, basata sulle direttive OO: si cerca di costruire sottosistemi intorno a "gruppi di

dati" relazionati tenendo conto anche delle relative funzioni. Per questo fine il Domain Object Model assume un ruolo molto importante: attraverso l'ispezione visiva permette di raggruppare "oggetti" strettamente relazionati tra loro e relativamente disaccoppiati dai restanti.

Per esempio, se si disponesse di un modello di dominio di un sistema di investimenti bancari, si potrebbe evincere l'esistenza di un gruppo di oggetti strettamente legati ai dati provenienti dalla quotazioni di mercato, di un altro relativo a gruppi di dati (relativamente) statici, di un gruppo relativo alla gestione dei trade, di un altro afferente alla gestione della messaggistica e così via. Chiaramente ciò non significa assolutamente che i vari "mondi" non siano relazionati tra loro.

Una buona verifica consiste nel "bilanciare" le ripartizioni in sottosistemi in funzione dei servizi da erogare. Da tener presente che, una volta decomposto il sistema, la fornitura di diversi servizi richiede in genere l'interazione dei sottosistemi attraverso opportune interfacce.

Il bilanciamento dalla prospettiva delle funzioni, permette di verificare che un sottosistema soddisfi i criteri base dell'ingegneria dei sistemi:

- forte coesione interna;
- minimo accoppiamento tra sottosistemi;
- comunicazione ridotta al minimo tra sottositemi;
- servizi logicamente accomunabili;

e così via.

Use Case Diagram

I diagrammi dei casi d'uso mostrano un insieme di entità esterne al sistema, dette *attori*, associati con le funzionalità, dette a loro volta *Use Case* (casi d'uso), che il sistema dovrà realizzare. L'interazione tra gli attori e i casi d'uso è espressa per mezzo di una sequenza di messaggi scambiati tra gli attori e il sistema. L'obiettivo dei diagrammi dei casi d'uso è definire un comportamento coerente senza rivelare la struttura interna del sistema. Nel contesto dello UML, un attore è una qualsiasi entità esterna al sistema che interagisce con esso: può essere un operatore umano, un dispositivo fisico qualsiasi, un sensore, un legacy system, e così via.

All'interno dei diagrammi dei casi d'uso sono illustrati un insieme di servizi, gli use case appunto, che il sistema dovrà fornire. Si faccia attenzione a non fare confusione: sia i diagrammi, sia le loro funzionalità sono chiamati Use Case.

I casi d'uso possono essere connessi tra loro per mezzo di una serie di relazioni quali generalizzazione, inclusione ed estensione.

I diagrammi dei casi d'uso costituiscono il fondamento della Use Case View (ancora una volta use case) e in particolare ne rappresentano la proiezione statica. Tipicamente la vista dei casi d'uso è composta da un insieme di diagrammi use case, i quali sono indipendenti gli uni dagli altri, pur condividendo alcuni elementi.

Per illustrare i vari diagrammi forniti dallo UML si è deciso di prendere in considerazione un unico esempio: il sito Internet/Intranet con funzioni di commercio elettronico di un teatro.

> In questo caso si tratta di un sistema frutto della fantasia e quindi potrebbero evidenziarsi lacune e incongruenze tipiche di sistemi non realizzati: solo il passaggio attraverso le varie fasi del processo permette di effettuare revisioni formali e quindi di evidenziare le immancabili lacune e incoerenze.

Il primo diagramma che viene visualizzato è una sorta di indice, o meglio di ricapitolazione delle varie funzioni fornite dal sistema: per ogni caso d'uso visualizzato è lecito attendersi il relativo diagramma di dettaglio.

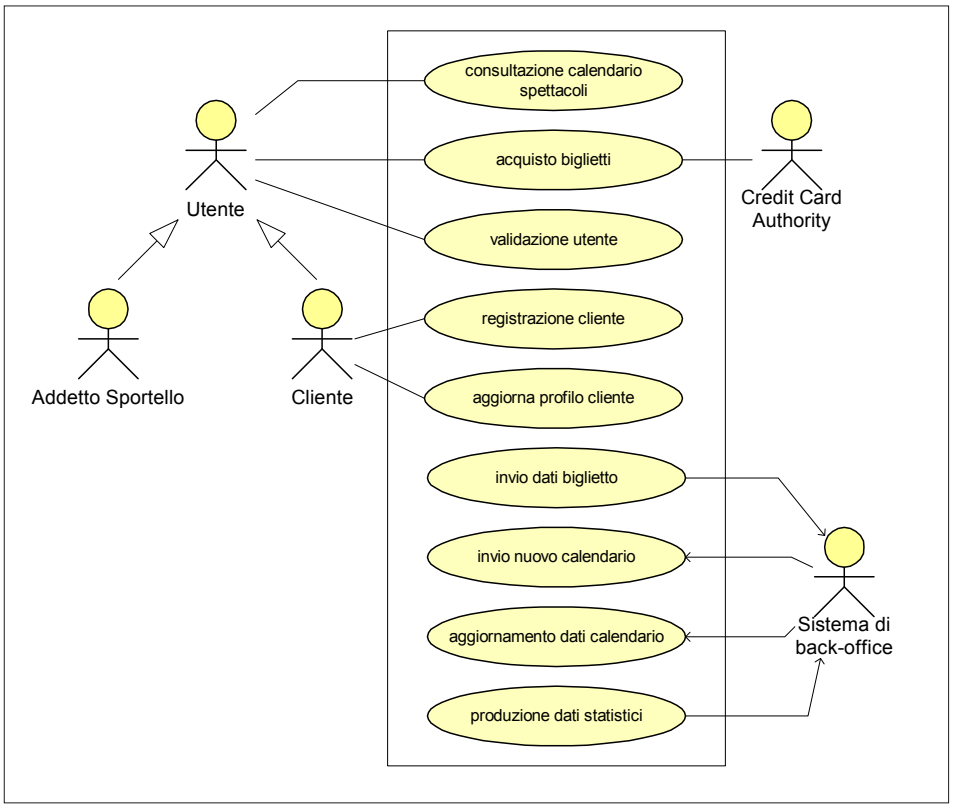
È buona pratica visualizzare il diagramma iniziale di overview, sebbene di solito non sia buona pratica procedere per affinamenti successivi e mostrare diagrammi via via di maggiore dettaglio: si rischia di disegnare il sistema in fasi troppo premature e con strumenti sbagliati (non si devono applicare metodologie quali il Data Flow Diagram).

In esso compaiono Use Case con funzioni di riepilogo e quindi non completamente coerenti con quelli di dettaglio: ciò è legittimo considerando il relativo fine di overview. Per esempio viene mostrato che l'attore `Credit Card Authority`, è associato allo Use Case `vendita biglietti`, mentre nella realtà ne è previsto uno appropriato: `ottenimento autorizzazione`.

Analizzando il diagramma dei casi d'uso mostrati in fig. 2.9, si può notare che, in primo luogo, gli attori possono essere suddivisi in due macrocategorie: quelli umani (`Addetto allo sportello` e `Cliente`) e quelli non umani (`Credit Card Authority` e `Sistema di back office`).

La suddivisione degli attori umani del sistema in `clienti` e `addetti allo sportello` è dovuta, come si vedrà meglio nel diagramma di deployment, alle modalità con cui sarà possibile fruire del sistema: attraverso qualsiasi stazione remota Internet e per mezzo di apposite postazioni ubicate nelle varie biglietterie.

Figura 2.9 — *Diagramma dei casi d'uso: overview delle funzioni del sistema.*



Le direttive Object Oriented consigliano, qualora possibile, di inserire attori, eventualmente astratti, atti a raggruppare il comportamento comune di altri.

In questo contesto ciò si traduce con l'aggiunta di un attore astratto, denominato genericamente `Utente`, specializzato dai due attori concreti: `Addetto allo sportello` e `Cliente`.

Dall'analisi degli attori non umani, si può notare che:

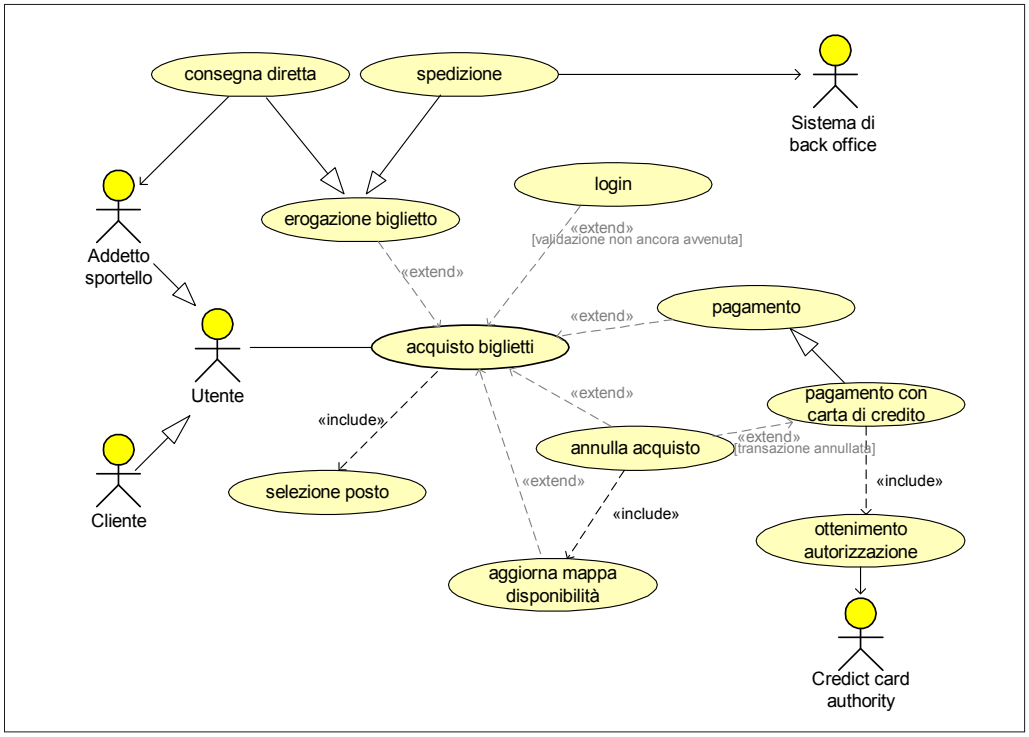
- l'acquisto on-line di biglietti è fruibile solo ai possessori di carta di credito e le relative transazioni sono subordinate all'autorizzazione fornita da apposito sistema: Credit Card Authority;

- il sistema rappresenta una sorta di front office, mentre tutte le attività di contabilità, amministrative, ecc. vengono effettuate da apposito sistema: Sistema di back office (BCS). I due sistemi dialogano tra loro attraverso invio di opportuni messaggi. Per esempio il BCS invia messaggi contenenti aggiornamenti relativi ai calendari degli spettacoli e riceve quelli relativi ai biglietti venduti.

Nel diagramma di fig. 2.10 viene riportato lo Use Case Diagram relativo alla vendita dei biglietti teatrali.

Una delle leggi molto applicate nella pratica è nota come "legge del 7" [BIB28]. Si tratta del consiglio, proveniente direttamente da studi di psicologia relativi alla capacità di concentrazione e comprensione, che propone di limitare il numero massimo degli elementi presenti in ogni diagramma a 7 unità.

Figura 2.10 — *Diagramma dei casi d'uso: dettaglio Use Case vendita biglietti.*



In questo caso non è stata rispettata per semplici motivi legati alla necessità di presentare un unico diagramma e di presentarlo in un contesto organico.

Sebbene gli Use Case Diagram verranno presentati in dettaglio nei prossimi due capitoli, ne viene comunque fornita la chiave di lettura. Le *frecce tratteggiate* che uniscono i vari casi d'uso rappresentano relazioni di *include* ed *extend*. Si tratta di relazioni molto dibattute, argomentate dettagliatamente nell'apposita sede. Per ora basti considerare che un'inclusione ha un significato del tutto equivalente a una invocazione di procedura effettuata in un programma scritto in un qualsiasi linguaggio di programmazione. Ad un certo punto dell'esecuzione di un caso d'uso compare la richiesta di inclusione di un altro incluso, quindi il flusso passa a questo ultimo che viene eseguito completamente e in seguito il controllo ritorna allo Use Case chiamante. Per esempio la relazione di inclusione che lega il caso d'uso `annulla acquisto`, a quello di `aggiorna mappa disponibilità`, sancisce che la funzione di annullamento di un acquisto, durante la propria esecuzione, eseguirà un aggiornamento della mappa delle disponibilità dei posti rendendo nuovamente disponibili i posti precedentemente riservati. La relazione di extend è molto simile ma più potente: è possibile associarvi un'espressione booleana che deve essere vera per abilitare l'esecuzione dello Use Case: in altre parole l'esecuzione di uno Use Case estendente è subordinata al soddisfacimento di una condizione di guardia. Nel caso in cui tale espressione non sia presente (indipendentemente dal fatto che sia visualizzata o meno), ciò equivale a una condizione sempre verificata. Tipicamente, le relazioni di extend sono utilizzate per distinguere il comportamento opzionale da quello obbligatorio visualizzato per mezzo della relazione di include.

Per esempio lo use case `pagamento con carta di credito`, nel caso in cui l'autorizzazione venga rifiutata, prevede l'esecuzione della funzione di annullamento della vendita (`annulla acquisto`). Anche se la direzione della freccia della relazione di dipendenza potrebbe risultare innaturale è invece corretta: la relazione di extend prevede che la funzione utilizzata (come per esempio `annulla vendita`) venga incorporata nell'esecuzione del caso d'uso "invocante" (`pagamento con carta di credito`).

> Di seguito viene riportata la descrizione dell'intero diagramma dei casi d'uso. Per esercizio, si consiglia al lettore, di provare a descriverlo per poi verificare la propria descrizione con quanto riportato di seguito.

In primo luogo il caso d'uso di vendita biglietti prevede l'autenticazione dell'utente (`login`). Si tratta di un extend — e quindi di un comportamento opzionale — in quanto è necessario eseguire il caso d'uso solo se l'utente non sia già stato precedentemente riconosciuto nell'arco della stessa sessione. Per questioni di leggibilità, molte delle condizioni presenti nelle relazioni di estensione non sono state riportate.

Nel caso in cui tutto proceda bene, l'utente seleziona lo spettacolo desiderato e quindi i posti desiderati tra quelli disponibili. Come si può notare fin da questi primi scampoli,

i diagrammi dei casi d'uso, da soli, non sono in grado di fornire alcuna indicazione circa la dinamica dell'esecuzione, l'ordine di esecuzione degli stessi casi d'uso, né tantomeno permettono di descrivere il comportamento che ciascuno di essi possiede. A seguito della selezione del posto viene eseguita la funzione di aggiornamento della mappa di disponibilità: i posti "prenotati" non devono ovviamente poter essere selezionati da altri utenti.

Selezionati anche i posti, è necessario effettuare il fatidico pagamento. Nel caso la fruizione del sistema avvenga da uno sportello vendita biglietti, il pagamento può avvenire o tramite contante o per mezzo di carta di credito, in tutti gli altri casi (da casa o attraverso apposito chiosco) solo per mezzo di carta di credito.

Nel caso in cui il cliente opti per un pagamento tramite carta di credito è necessario richiedere autorizzazione a un apposito sistema esterno, indicato genericamente con il nome di `Credit Card Authority`. Nel caso in cui l'autorizzazione sia negata, è necessario effettuare una sorta di rollback: la vendita viene annullata e quindi i posti precedentemente prenotati vengono nuovamente resi disponibili.

Effettuato con successo anche il pagamento, è possibile effettuare l'erogazione del biglietto. Nel caso dello sportello è possibile fornirlo direttamente al cliente, eventualmente con opportune stampe su biglietti di formato predefinito (come avviene nelle agenzie di viaggi quando si acquista un biglietto aereo). Nel caso in cui il cliente stia fruendo del sistema attraverso le altre modalità, potrà decidere se richiedere il recapito presso l'indirizzo impostato o ritirarlo in uno degli uffici oppure direttamente al teatro.

I diagrammi dei casi d'uso illustrano la proiezione statica del sistema; quella dinamica è invece affidata ad altri strumenti (come si vedrà nei capitoli successivi sugli Use Case), quali i diagrammi di interazione, di attività, i flussi di attività od opportuni moduli.

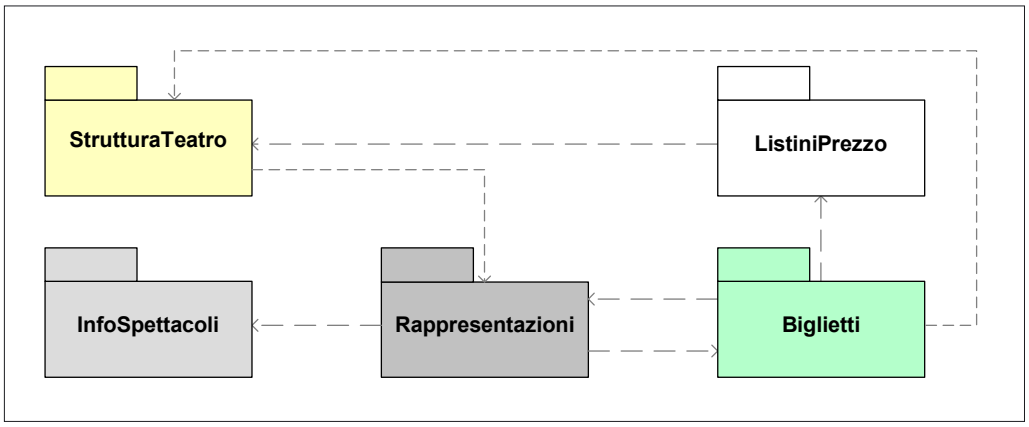
Class Diagram

Il diagramma delle classi è probabilmente una delle tipologie di diagramma più note e affascinanti dello UML. L'utilizzo più importante è legato alla realizzazione del modello di disegno, il quale, una volta completato, rappresenta graficamente l'implementazione del sistema eseguibile.

> Coloro che provengono dal mondo dell'analisi strutturata potranno notare la stretta somiglianza con i diagrammi Entità-Relazioni, di cui i Class Diagram rappresentano l'evoluzione OO (vi è il fattore comportamentale). Ovviamente il passo non è diretto: nella catena evolutiva dei diagrammi delle classi sono presenti diversi significativi anelli, quali per esempio il formalismo OMT e quello di Booch, apprezzabile nel libro capostipite dei Design Pattern ([BIB04]).

Obiettivo principale dei diagrammi delle classi è visualizzare la proiezione statica del sistema: sono utilizzati per realizzare i modelli a oggetti del dominio del business, di

Figura 2.11 — *Organizzazione in package del modello del dominio.*



analisi e di disegno. Qualora si utilizzino Database Management System Object Oriented, i diagrammi delle classi ne rappresentano la struttura (schema logico) utilizzata per la persistenza del sistema (negli altri casi è necessario realizzare un mapping); inoltre supportano i diagrammi degli oggetti, del deployment (dispiegamento), gli interaction ecc.

Come suggerito dal nome, i diagrammi delle classi sono costituiti da un insieme di classi, interfacce e relazioni tra tali elementi. Esistono diverse tipologie di relazioni con le quali è possibile associare classi, come, la dipendenza, l'associazione, la specializzazione, il raggruppamento in package e così via.

Il diagramma riportato nella fig. 2.11 rappresenta una possibile organizzazione in package di un'opportuna sezione del modello ad oggetti del dominio. Probabilmente il livello di granularità è eccessivo, ossia alcuni package raggruppano non più di un paio di classi e potrebbero essere tranquillamente inglobati in altri. In questo contesto si è comunque deciso di mostrarli per poter meglio illustrare i formalismi dello UML. Questo approccio è stato utilizzato in tutto il libro: elevata granularità a fini espositivi.

Se, per ragioni di riusabilità del codice, è opportuno seguire il principio di disaccoppiamento al livello delle classi, lo è ancora di più a livello dei package. Probabilmente è più importante poter riutilizzare interi package piuttosto che singole classi. Da notare che spesso non si parla più di riusabilità del codice, bensì di sostituibilità di opportuni "componenti" al fine di rendere più semplice far "rincorrere" al sistema l'evoluzione del business del cliente.

La mutua dipendenza esistente tra package (nel modello di fig. 2.11 tra `Biglietti` e `Rappresentazioni`) non è, tipicamente, un buon esempio di modellazione. Nella fase di disegno, si cerca di ridurre il grado di accoppiamento per mezzo dell'introduzione di opportune interfacce. Le ottimizzazioni dovrebbero essere introdotte con attenzio-

ne nei modelli relativi al dominio del problema, il cui scopo principale è rappresentare appunto il dominio del problema e cercare di ottenere riscontri dal cliente su quanto modellato.

Le relazioni tra package mostrate in figura (frecce tratteggiate) illustrano le relative dipendenze: se un package A dipende da uno B, ciò implica che modifiche a quest'ultimo package (indipendente) si ripercuotono sul primo (dipendente). Per esempio una revisione dell'organizzazione interna del package dei `ListiniPrezzo` verosimilmente richiede una verifica di quello dei `Biglietti`.

Nel diagramma di fig. 2.12 viene proposto un frammento di quello che potrebbe essere il Domain Object Model (modello ad oggetti del dominio, trattato approfonditamente nel Capitolo 8). Questa particolare tipologia di modello ha alcune caratteristiche peculiari:

Figura 2.12 — *Frammento del Domain Model del sistema teatrale.*

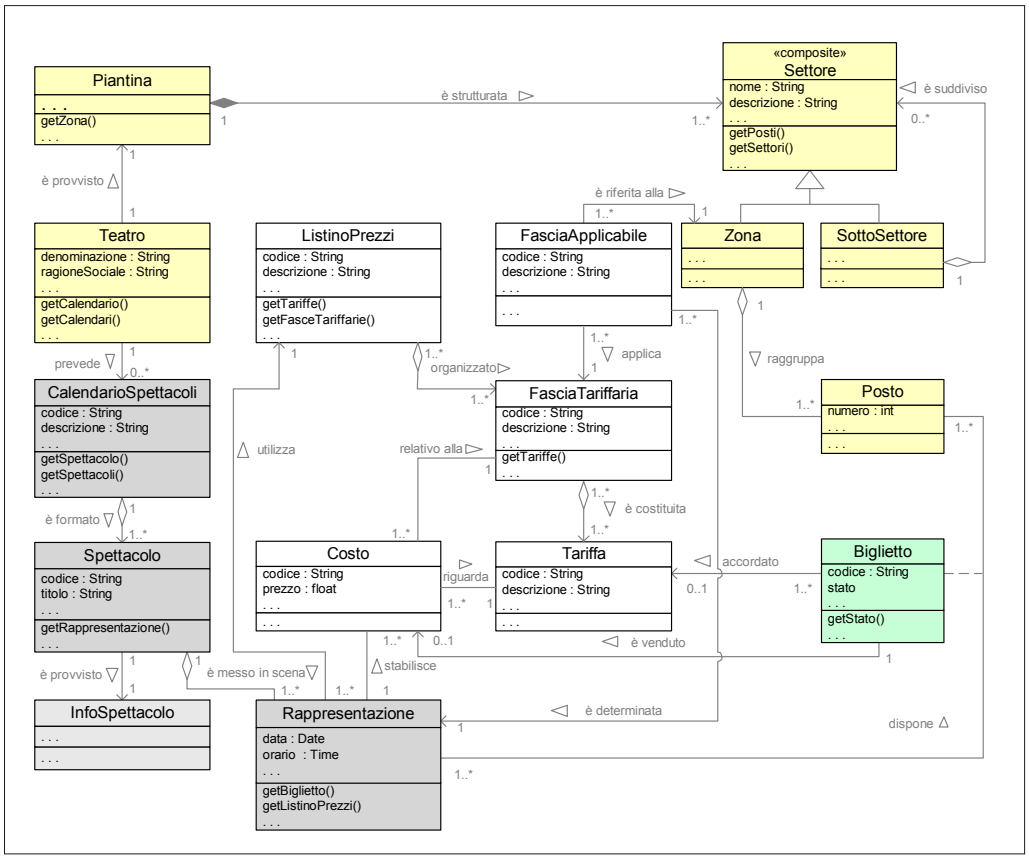
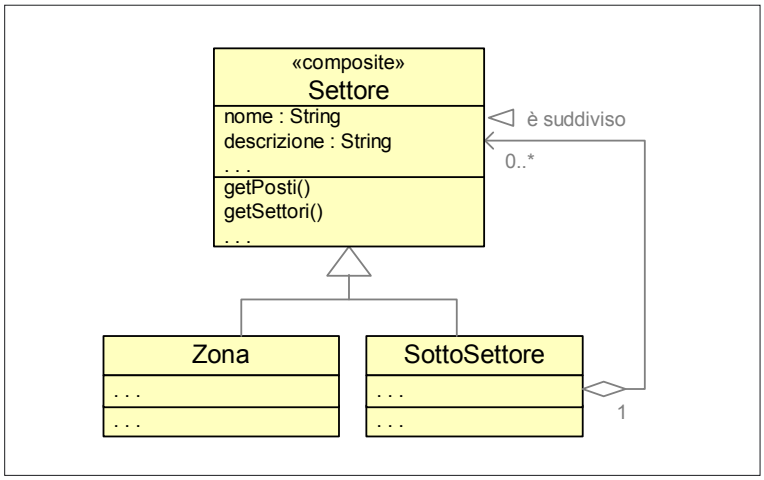


Figura 2.13 — *Frammento del diagramma delle classi relativo all'utilizzo del design pattern Composite.*



- sono presenti unicamente entità effettivamente esistenti nel mondo reale;
- nelle varie classi vengono riportati principalmente gli attributi ritenuti più significativi, mentre l'interesse per i metodi è decisamente più marginale.

I rettangoli mostrati in figura rappresentano classi, mentre i segmenti descrivono relazioni di associazione. Una classe descrive un insieme di oggetti che condividono struttura (attributi), comportamento (operazioni) e relazioni, mentre le relazioni di associazione indicano legami strutturali tra oggetti. Di questa relazione esistono diverse versioni (aggregazione, composizione, ecc.). Una relazione di associazione tra due classi indica che — a meno di ulteriori vincoli come la navigabilità — è possibile navigare dagli oggetti di una classe a quelli dell'altra e viceversa.

Il diagramma rappresentato risulta piuttosto interessante in quanto permette di illustrare buona parte del formalismo previsto dallo UML per i disegnare i diagrammi delle classi.

Si proceda con l'analisi del diagramma a partire dalla classe `Teatro`, e in particolare si inizi con il considerare la relativa rappresentazione strutturale. Un `Teatro` dispone di una `Piantina` la cui unica istanza è utilizzata come raggruppamento, ossia come punto di partenza della struttura. Quest'ultima è rappresentata attraverso un celebre Design Pattern denominato Composite (*cfr* [BIB04]).

Brevemente, il pattern Composite permette di modellare complessi grafi ad albero (intrinsecamente ricorsivi) in cui compaiono relazioni gerarchiche "tutto–parte". In particolare si distinguono due elementi fondamentali: quello composto (dà luogo alla

ricorsione) e quello base, detto foglia (*leaf*), che pone fine alla ricorsione. Nel contesto oggetto di studio la foglia dell'albero (nodo senza figli, elemento "parte" della relazione gerarchica) è rappresentato dalla `Zona`, mentre i nodi con figli sono rappresentati dai `SottoSettori` (fig. 2.13). Quindi, in accordo al diagramma, la `Zona` è l'elemento più semplice della struttura, quello che raggruppa direttamente i posti.

Il ricorso al Composite permette di asserire, per esempio, che la struttura del teatro è costituita da livelli (istanze della classe `SottoSettore`) che a loro volta sono suddivisi in altri `SottoSettori` (centrale, centrale-est, est, centrale-ovest e ovest) suddivisi per `Zone` (palchetto centrale, palchetto centrale di destra, palchetto centrale di sinistra, ecc.).

Si tratta ovviamente di un caso e nulla vieta di disporre di diverse configurazioni, come per esempio che il teatro possa essere diviso in `Livelli` e `Platea` (primo livello dell'albero) e quindi in `Zone`. In un successivo paragrafo viene fornita una visualizzazione di quanto esposto attraverso l'utilizzo dei diagrammi degli oggetti (fig. 2.17).

L'utilizzo del pattern Composite ha permesso di risolvere elegantemente ed efficacemente una struttura altresì complessa con poco sforzo intellettivo: è sufficiente ricercare nel relativo catalogo una soluzione generale al proprio problema e quindi adattarla alle proprie necessità.

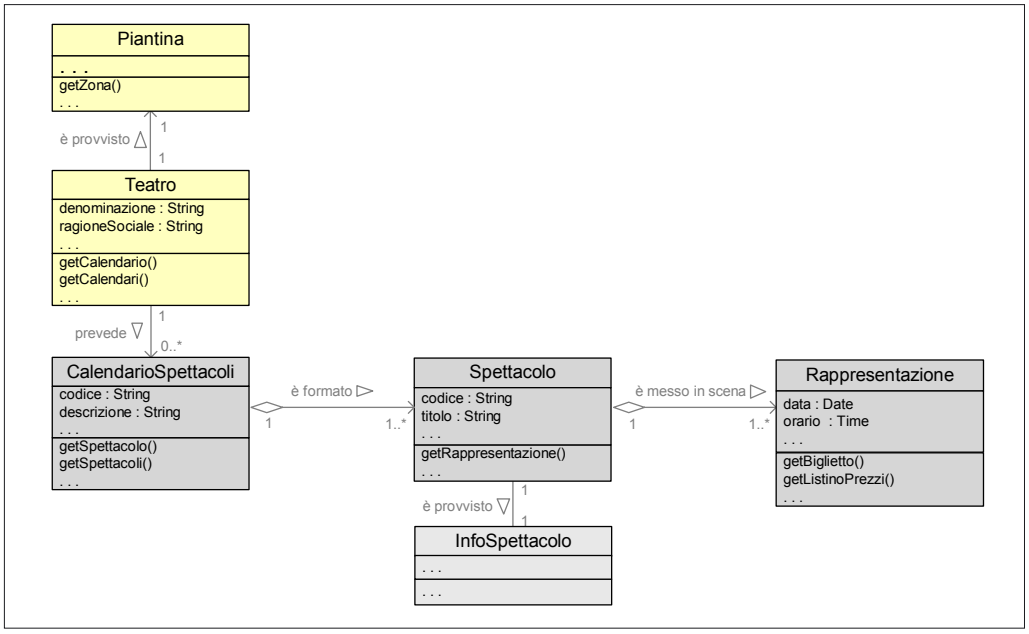
L'organizzazione gerarchica, e in particolare il raggruppamento dei `Posti` per `Zone`, è particolarmente importante in quanto permette di assegnare opportunamente le `Tariffe`: è lecito attendersi che un palco centrale abbia una tariffa leggermente diversa da un posto nel loggione.

Dall'analisi del diagramma in figura 2.12, si può notare che spesso le classi vengono interconnesse da una particolare associazione, detta *aggregazione*, visualizzata per mezzo di un rombo vuoto posto ad una estremità. La relazione di aggregazione specifica una relazione binaria, detta *whole-part* (tutto–parte), tra un elemento detto *aggregato* (il "tutto") e uno detto *costituente* (la "parte"). Nella rappresentazione grafica il rombo viene collocato nell'estremità relativa all'entità "tutto". La peculiarità rispetto a una relazione di associazione semplice è la semantica: l'aggregazione sancisce una relazione decisamente più forte tra le parti associate.

Tornando alla classe `Teatro` (*cfr* fig. 2.14) essa può essere collegata a *n* istanze della classe `CalendariSpettacoli`: si supponga che lo stesso teatro possa essere utilizzato per diverse tipologie di spettacoli: rappresentazioni teatrali, concerti, show ecc., oppure che gli stessi possano essere divisi per stagioni: il calendario invernale, quello estivo e così via.

Ogni calendario è ovviamente costituito da un insieme di spettacoli (*La Tosca*, *La Traviata*, *La Locandiera*, il concerto dei Vaia Con Dios, lo show *Barracuda* di D. Luttazzi…) le cui informazioni salienti sono riportate in un'apposita classe denominata `InfoSpettacolo`. Volendo descrivere propriamente queste ultime informazioni sarebbe stato necessario realizzare un apposito modello: per questo nel diagramma dei package

Figura 2.14 — *Frammento del diagramma della classi relativo a* `Spettacoli` *e* `Rappresentazioni`.



è mostrato un apposito package denominato `InfoSpettacoli`. In questo caso si è deciso di trascurarlo per non complicare ulteriormente la spiegazione.

> In prima analisi si potrebbe distinguere la tipologia di spettacolo (rappresentazione teatrale, concerto, show, ecc.) verosimilmente attraverso una relazione di generalizzazione con tante specializzazioni quante sono le tipologie. Poi si sarebbero dovute formalizzare informazioni come: autori o compositori, direttori o registi o coreografi, gruppo teatrale o orchestra sinfonica o corpo di ballo, ecc.

Ogni `Spettacolo`, tipicamente, prevede diverse `Rappresentazioni`.

A questo punto si giunge alle informazioni relative ai listini prezzi (fig. 2.15): un `Teatro` ne può prevedere diversi che possono differenziarsi sia per questioni lucrative, sia per struttura.

L'intento principale seguito per la modellazione di questa sezione è stato realizzare un modello abbastanza generale da potersi applicare al maggior numero di casi possibili senza però complicare eccessivamente la struttura.

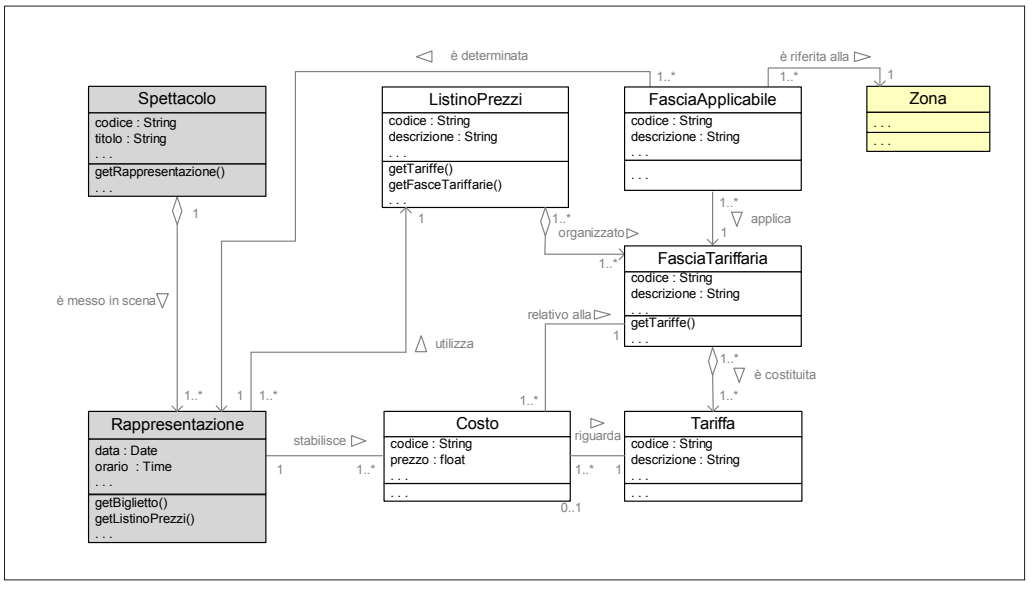
Un altro elemento preso in esame è l'effettiva analisi della realtà — quella dei teatri londinesi — dove in particolare si può notare che la struttura dei tariffari tende a rimane-

re fissa (tariffa extra lusso da applicarsi ai palchetti centrali e primissime file della platea, tariffa normale per la restante parte della platea, tariffa scontata per il loggione, ecc.) così come gli stessi prezzi tendono a prevedere poche alternative (tariffa di lusso equivalente a 150 sterline per spettacoli più importanti, 100 per quelli medi, 75 per quelli meno blasonati o in periodi dell'anno meno richiesti, ecc.).

Se la seguente trattazione dovesse risultare oscura, è possibile far riferimento ai paragrafi successivi e alla fig. 2.18), in cui viene riportato un esempio di listino. In prima analisi un listino prezzi può essere suddiviso in diverse fasce tariffarie (per esempio: `Imperiale`, `Lusso`, `Normale`, `Economica`, ecc.). Uno dei meccanismi alla base del listino prezzi modellato in figura è dato dal legame degli oggetti `Zona` con quelli `FasciaTariffaria`. In particolare è necessario quotare, per ogni zona, la fascia tariffaria da applicare. Questa "associazione" però è più complicata di quanto potrebbe sembrare; in effetti, ciascuna quotazione è fissata in funzione di tre fattori: la `Zona`, il `ListinoPrezzi` e la `Rappresentazione`.

> Volendo rappresentare propriamente questa regole di business, sarebbe necessario modellare tale congiunzione per mezzo di una relazione ternaria tra i suddetti elementi con in aggiunta una classe associazione in grado di individuare la fascia tariffaria applicata (consultare capitolo 7). Considerato il carattere introduttivo di questo capitolo si è cercato di semplificare il più possibile i modelli presentati evitando l'introduzione di meccanismi particolarmente complessi.

Figura 2.15 — *Frammento del diagramma della classi relativo ai listini prezzi.*



La precedente condizione si presta ad essere semplificata notando che ciascuna `Rappresentazione` utilizza un solo `ListinoPrezzi`, quindi specificato un oggetto di tipo `Rappresentazione`, attraverso la relazione `utilizza`, è possibile risalire al relativo `ListinoPrezzi`. Ciò permette di ridurre la relazione ternaria ad una binaria con in aggiunta una classe associazione. Nel modello si è preferito mostrare direttamente una classe di collegamento, `FasciaApplicabile`, la quale, data una particolare zona e una specifica rappresentazione, definisce la `FasciaTariffaria` da applicare.

Le fasce tariffarie, a loro volta, sono ulteriormente suddivise in tariffe. Ciò permette di definire diverse tipologie di prezzi per una stessa fascia tariffaria. Per esempio, nell'ambito della fascia tariffaria `standard`, potrebbe essere utile differenziare il costo dei biglietti per le categorie: studenti, anziani e così via.

Per comprendere appieno i principi base del modello presentato fin qui, si prenda in considerazione un ipotetico servizio di impostazione dei listini prezzi. Questo deve essere strutturato in modo tale da richiedere all'utente di selezionare la rappresentazione di cui specificare il listino e il listino stesso da utilizzare (definizione della relazione `utilizza` tra `Rappresentazione` e `ListinoPrezzi`). Fatto ciò, per ogni zona in cui è stato organizzato logicamente il teatro, l'utente dovrà definire quale fascia tariffaria applicarvi (generare istanze della classe `FasciaApplicabile`).

Terminata anche questa attività, dovrà definire il costo di ciascuna tariffa.

Da notare come il vincolo molto importante per cui una zona può prevedere una sola `FasciaTariffaria` all'interno di uno stesso listino non si desume dalla lettura del diagramma (uno stesso oggetto `Zona` può essere associato a diverse istanze `FasciaApplicabile`). Ciò è ottenibile utilizzando meccanismi più complessi, come la relazione ternaria oppure descriverlo esplicitamente, per mezzo dell'OCL (Object Constraint Language).

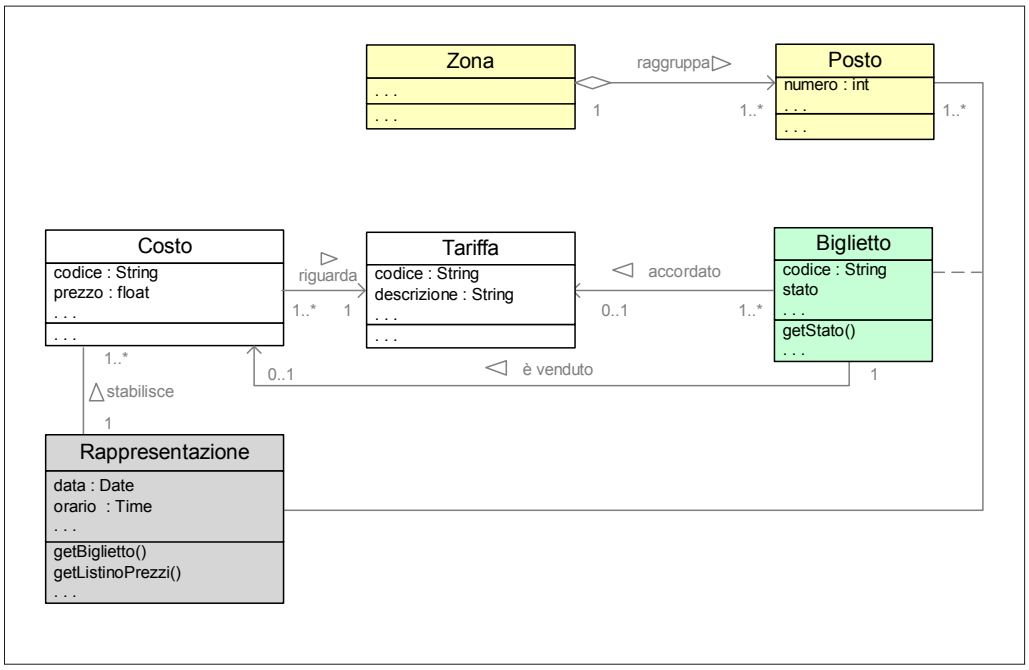
> Nel modello la classe Biglietto non è associata direttamente a un'altra, bensì a una relazione tra due classi e pertanto è un esempio di Association Class (classe associazione). Da un punto di vista implementativo, chiaramente, non è possibile associare una classe a una relazione. Si tenga presente che, in ultima analisi, un'associazione è l'esportazione e/o l'importazione di un indirizzo di memoria di un oggetto (pardon… di un reference) in un particolare attributo di un altro oggetto. La sua realizzazione, tipicamente, prevede che la classe associazione contenga i riferimenti a entrambe le classi che darebbero vita all'associazione da cui dipende. Nel caso oggetto di studio Biglietto, dovrebbe memorizzare i reference degli oggetti istanza delle classi Posto e Rappresentazione. La domanda potrebbe essere la seguente: "perché allora non modellare la classe associazione direttamente con due associazioni alle classi da cui dipende?". La risposta è semplice e risiede nella semantica: la classe associazione sancisce che, se l'oggetto istanza di una delle due classi a cui è associata l'istanza della classe associazione viene distrutto — o dereferenziata in caso si

consideri Java — anche la classe associazione deve essere distrutta di conseguenza. Con due associazioni a due classi non si evincerebbe questa regola a meno di non inserire esplicitamente appositi vincoli. Questi argomenti verranno debitamente trattati nel capitolo relativo ai Class Diagram.

Per ciò che riguarda la classe `Biglietto` (fig. 2.16), si può notare che questi si riferiscono ad un particolare `Posto` nel contesto di una specifica `Rappresentazione` e, una volta venduti, sono associati alla tipologia della `Tariffa` applicata, nonché al relativo `Costo`. Per esempio il biglietto acquistato da uno studente applica la relativa tariffa, che per quella particolare `Rappresentazione` prevede il costo di *x* sterline.

In questo contesto i `Biglietti` esistono indipendentemente dal fatto che siano stati acquistati o meno (non a caso esiste l'attributo `Stato`), il che dovrebbe essere abbastanza rispondente alla realtà. Naturalmente quando si acquista un biglietto lo si fa in riferimento a una particolare `Rappresentazione`, e per un posto appartenente a una ben specifica `Zona`.

Figura 2.16 — *Frammento del diagramma delle classi relativo ai biglietti e alle relative tariffe.*



Ecco che la classe `Biglietto` rappresenta una classe associazione e dipende appunto dal legame tra la classe `Rappresentazione` e quella `Posto`. Quindi, ricapitolando, per ogni `Posto` e per ogni `Rappresentazione` è prevista un'apposita istanza della classe `Biglietto` che, una volta acquistato, definisce una specifica `Tariffa` applicata e il relativo costo.

Object Diagram

I diagrammi degli oggetti rappresentano una variante dei diagrammi delle classi, tanto che anche la notazione utilizzata è pressoché equivalente con le sole differenze che i nomi degli oggetti vengono sottolineati e le relazioni vengono dettagliate.

Gli Object Diagram mostrano un numero di oggetti istanze delle classi con i relativi legami riportati in modo esplicito. Per esempio, se nel diagramma delle classi una classe A è associata con *n* istanze della classe B, nel diagramma degli oggetti vengono visualizzati un opportuno insieme di oggetti istanza della classe A ed esplicitamente tutti i legami che lo connettono con gli altrettanti oggetti istanza della classe B.

Anche questa tipologia di diagramma si occupa della proiezione statica del sistema e mostra un ipotetico esempio di un diagramma delle classi. Si tratta della famosa diapositiva "scattata" a un istante di tempo preciso, riportante un ipotetico stato di esecuzione evidenziato dagli oggetti presenti in memoria e dal relativo stato.

Mentre un diagramma delle classi è sempre valido, un diagramma degli oggetti rappresenta una possibile istantanea del sistema valida in un istante di tempo ben preciso.

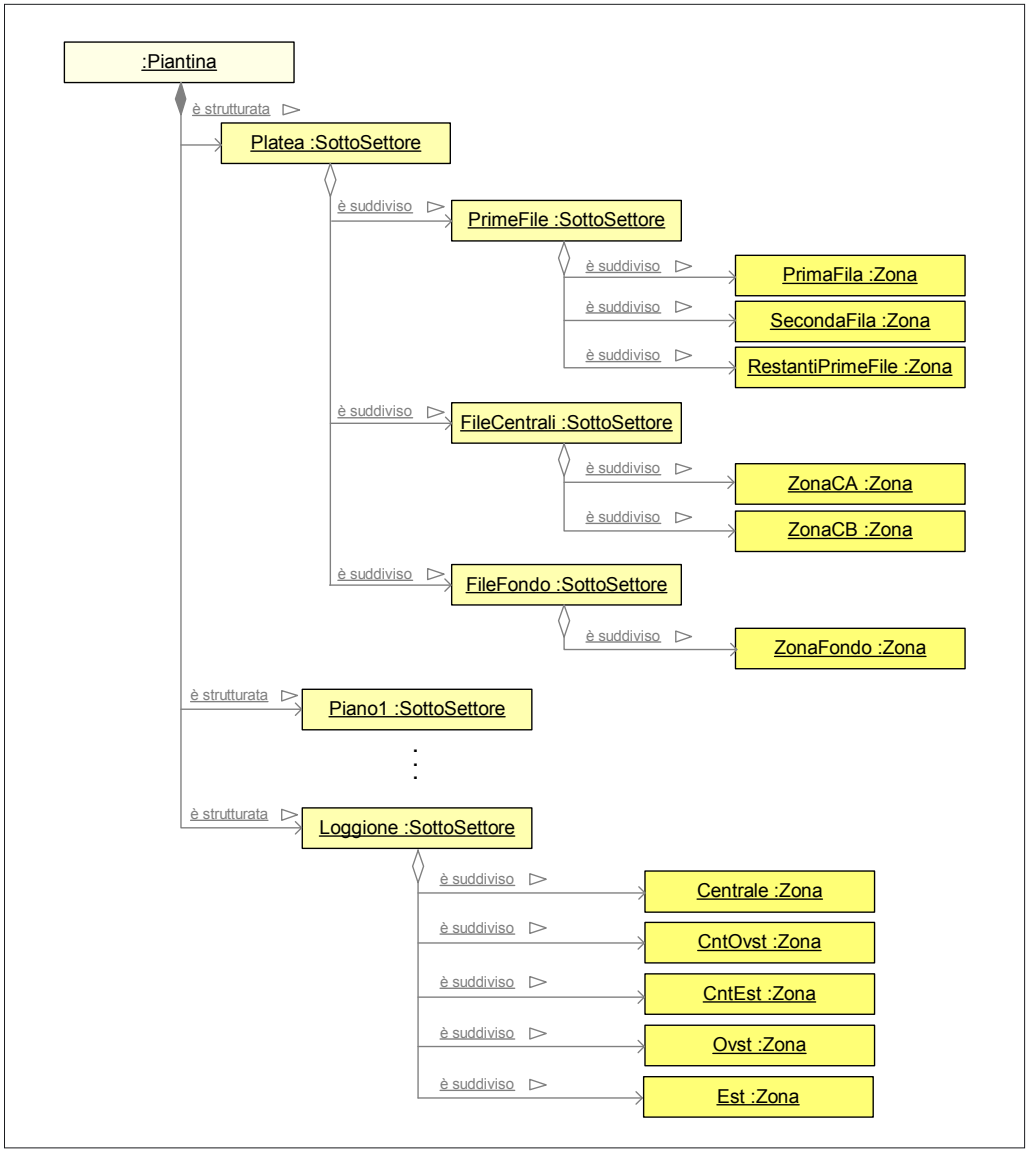
I diagrammi degli oggetti vengono utilizzati essenzialmente per dettagliare le relazioni presenti in diagrammi delle classi ritenute poco chiare o particolarmente complicate: ciò permette di verificare la correttezza logica dei diagrammi delle classi realizzati. Qualora non si riescano a concretizzare formalmente determinate relazioni o non si abbia la certezza che quanto modellato sia effettivamente ciò che si desiderava, è consigliabile realizzare un paio di diagrammi degli oggetti al fine di visualizzare come, in una situazione a regime, i vari oggetti siano relazionati gli uni agli altri.

L'utilizzo dei diagrammi degli oggetti nel corso di una modellazione è tipicamente limitato ma non per questo non importante: talune volte da solo è più esplicativo e preciso di molte linee di commento di un Class Diagram.

Il problema? La maggior parte dei tool commerciali (al momento in cui viene scritto il presente libro) sembrerebbe sottovalutarne l'importanza. Per realizzare i diagrammi de-

gli oggetti bisogna ricorrere sempre a qualche artificio, come simularlo attraverso un Class Diagram o un Collaboration. Tipicamente la soluzione migliore è realizzare i diagrammi degli oggetti per mezzo dei diagrammi di collaborazione.

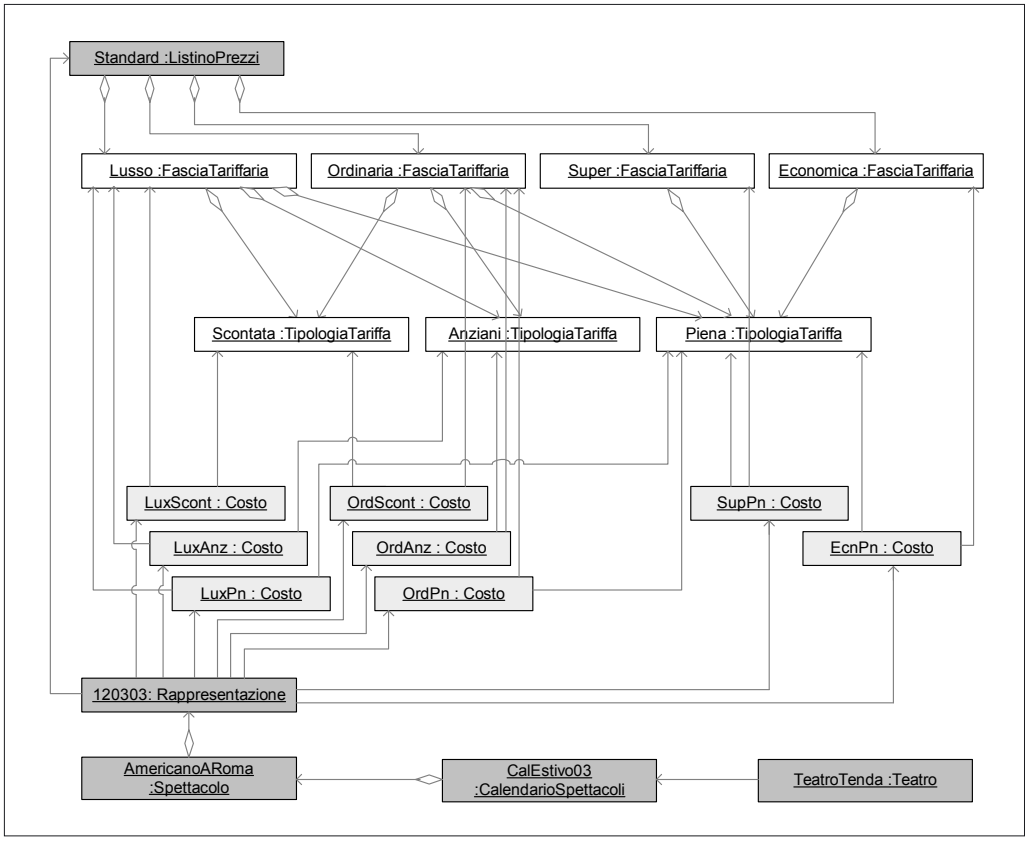
Figura 2.17 — *Esempio di diagrammi degli oggetti: frammento di un'ipotetica struttura di un teatro.*



Il diagramma riportato nella fig. 2.17 mostra una possibile organizzazione della struttura di un teatro. In una situazione reale, questa dovrebbe essere definita, attraverso un opportuno tool, all'atto della configurazione del sistema e memorizzata in un database. Il sistema, all'atto dell'avvio, dovrebbe provvedere a caricarla in memoria. Il diagramma visualizza una ipotetica struttura di un teatro attraverso un opportuno grafo ad albero e quindi dimostra l'efficacia del pattern Composite.

L'oggetto radice dell'intero albero è un'istanza della classe `Piantina`, alla quale si è ritenuto non necessario attribuire un nome in quanto ne esiste una sola istanza e quindi non si corre il rischio di confusione. I nodi di primo livello sono `Platea`, `Piano1`, `Piano2`, e così via fino a `Loggione`. Si tratta chiaramente di istanze della classe `SottoSettore`, specializzazione di `Settore`. Per questione di rappresentazione grafica, non viene data la descrizione di tutti gli elementi. La `Platea`, a sua volta, viene

Figura 2.18 — *Esempio di diagrammi degli oggetti: illustrazione dei listini prezzi.*



suddivisa in altri settori: PrimeFile, FileCentrali e FilediFondo ancora istanze della classe SottoSettore. Il settore PrimeFile, infine, è suddiviso in PrimaFila, SecondaFila e RestantiPrimeFile, istanze della classe Zona, che a sua volta è una specializzazione del Settore. Per il Loggione invece è stata decisa un'organizzazione diversa: si passa dal Settore direttamente alle Zone senza SottoSettori intermedi.

Il diagramma degli oggetti proposto in fig. 2.18, mostra un esempio di un listino prezzi utilizzato nella rappresentazione del 12 marzo 2003, dello spettacolo "Un Americano a Roma", appartenente al calendario degli spettacoli estivi del Teatro Tenda.

Il listino prevede quattro fasce tariffarie così composte:

- Lusso, suddivisa in tariffa Scontata, Anziani e Piena;
- Ordinaria, suddivisa in: tariffa Scontata, Anziani e Piena;
- Super che prevede unicamente la tariffa Piena;
- Economica anch'essa dotata unicamente della tariffa Piena.

L'applicazione del ListinoPrezzi alla specifica rappresentazione prevede:

- LuxScont per la tariffa Scontata della fascia tariffaria Lusso;
- LuxAnz per la tariffa Anziani della fascia tariffaria Lusso;
- LuxPn per la tariffa Anziani della fascia tariffaria Lusso, e così via;

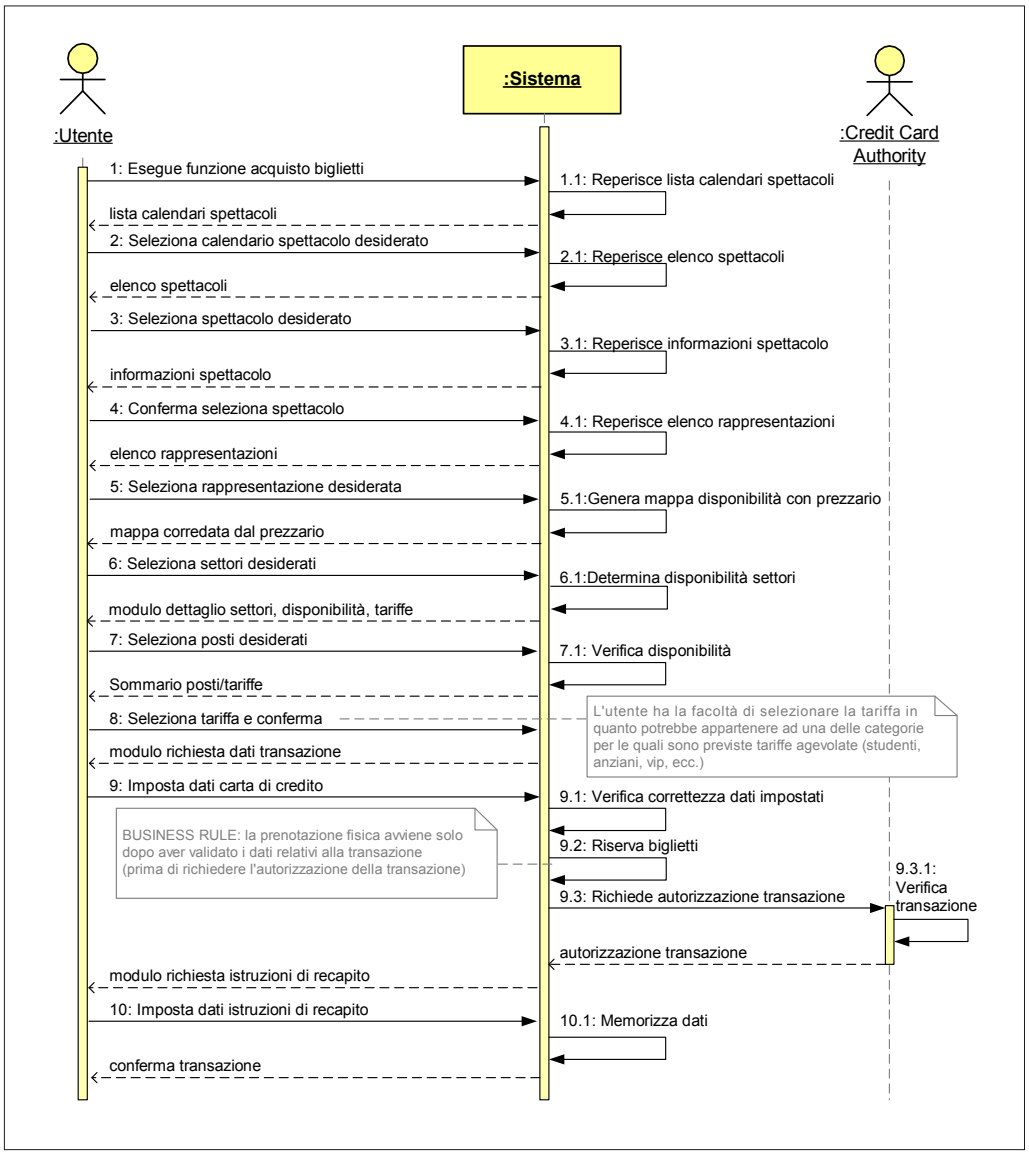
Interaction Diagram

I diagrammi di sequenza e collaborazione — detti anche di interazione in quanto mostrano le interazioni tra oggetti che costituiscono il sistema e/o con attori esterni allo stesso — vengono utilizzati per modellare il comportamento dinamico del sistema.

I due diagrammi risultano molto simili e si può passare agevolmente dall'una all'altra rappresentazione (isomorfi). Alcuni tool permettono, dato un Sequence Diagram, di generare l'equivalente Collaboration e viceversa.

Sequence e Collaboration si differenziano per via dell'aspetto dell'interazione a cui conferiscono maggior rilievo: i diagrammi di sequenza focalizzano l'attenzione sull'ordine temporale dello scambio di messaggi, i diagrammi di collaborazione mettono in risalto l'organizzazione degli oggetti che si scambiano i messaggi. Possono essere utilizzati con diversi livelli di astrazione in funzione degli obiettivi che si intende raggiungere e delle fasi in cui sono utilizzati.

Figura 2.19 — *Sequence diagram.*



Li si utilizza nella Use Cases View per modellarne la proiezione dinamica, ossia per fornire l'illustrazione grafica degli scenari (esempi di utilizzo dei casi d'uso). In tale contesto il livello di astrazione deve essere necessariamente elevato: ciò non significa che

si ignorano i dettagli del funzionamento del processo. Semplicemente l'astrazione è relativa alla mancanza di particolari a carattere più "implementativo", come per esempio i metodi di una classe da invocare.

In questo contesto i diagrammi di sequenza risultano particolarmente utili, mentre quelli di collaborazione lo sono molto meno: verrebbero visualizzati due/tre oggetti con molte connessioni.

I diagrammi di interazione vengono utilizzati anche in fase di analisi e disegno (livello di astrazione molto basso) per documentare l'utilizzo di classi, per illustrare come funzionalità complesse siano realizzate per mezzo dell'interazione (scambio di messaggi) tra più oggetti, e così via.

In fig. 2.19 viene riportato il Sequence Diagram che illustra l'interazione dinamica tra un cliente e il sito internet per l'acquisto dei biglietti. Il grande vantaggio offerto, come si riscontra facilmente, è legato alla semplicità di lettura e comprensione; pertanto il diagramma di sequenza si presta a essere utilizzato per illustrare dei comportamenti da sottoporre all'attenzione del cliente. In tal caso i diagrammi devono possedere un elevato livello di astrazione.

Come si vedrà meglio nel Capitolo 4, lo scenario illustrato viene comunemente denominato *main success scenario* (scenario principale di successo) in quanto illustra un'interazione in cui tutto — magicamente — "funziona bene" e non si verificano errori o eccezioni di sorta. Per esempio le varie verifiche di disponibilità non danno mai esito negativo, così come la richiesta di autorizzazione della transazione viene sempre accordata ecc.

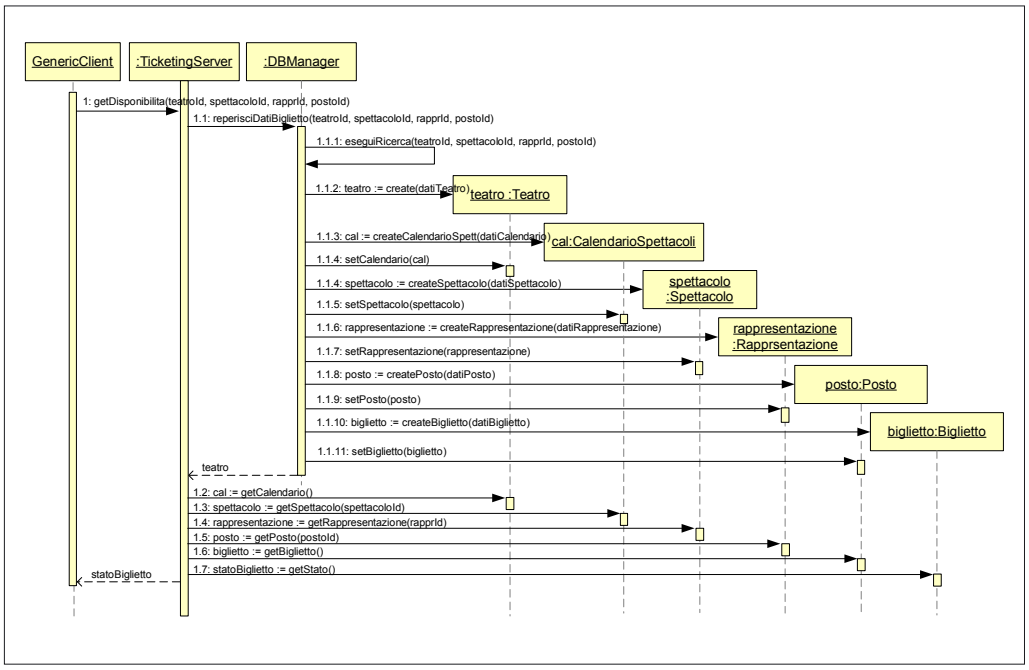
Per la visualizzazione delle anomalie e la relativa gestione, a meno di casi semplici, si preferisce realizzare altri diagrammi. L'obiettivo è evitare di realizzare diagrammi confusi e quindi meno leggibili: ciò ne invaliderebbe la peculiarità.

Però, se da un lato disporre di una serie di diagrammi permette di mantenere gli stessi semplici e lineari, dall'altro crea problemi nella manutenzione: spesso una modifica al flusso descritto nel main scenario implica l'aggiornamento di tutti i diagrammi. Questo è uno dei motivi per cui spesso gli scenari vengono descritti attraverso opportuni moduli di testo.

Nel caso dei diagrammi di sequenza, nella prima riga vengono riportati gli oggetti che partecipano all'interazione. Da ciascuno di essi parte una linea tratteggiata verticale che rappresenta il lasso di tempo in cui l'oggetto è in vita, mentre le varie frecce rappresentano lo scambio esplicito dei messaggi. Come si vedrà nel Capitolo 9 esistono diverse tipologie di messaggio.

Una prima obiezione che si potrebbe fare è la seguente: "Come è possibile realizzare diagrammi di interazione, i cui elementi principali sono gli oggetti (istanze di classi) nella vista dei casi d'uso? In altre parole, se concetti come classi e oggetti sono oggetto di studio di fasi successive del processo di sviluppo del software (analisi, disegno), come è possibile prevederli già in durante l'analisi dei requisiti utente?". Le risposte possono essere diverse.

Figura 2.20 — *Sequence Diagram a livello del modello di analisi.*



Una prima soluzione consiste nell'indicare globalmente il sistema attraverso un unico oggetto (come fatto nel diagramma in figura). Un'alternativa è utilizzare la propria esperienza cominciando a dare una "sbirciatina" all'interno del sistema individuando i primi macrosottosistemi.

Il diagramma di fig. 2.20 rappresenta un esempio di utilizzo della notazione dei diagrammi di sequenza nel modello di analisi: mostra come gli oggetti istanze delle classi collaborino tra loro per realizzare un determinato servizio, in questo caso la verifica della disponibilità di un posto per una specifica rappresentazione.

Come si vedrà nel capitolo 8 i modelli ad oggetti relativi alla fase di analisi prevedono l'utilizzo di appositi simboli (stereotipi) per evidenziare concetti quali: punti di accesso al sistema, gestori di processi ed entità.

In questa fase si cerca di raggiungere essenzialmente due importanti obiettivi:

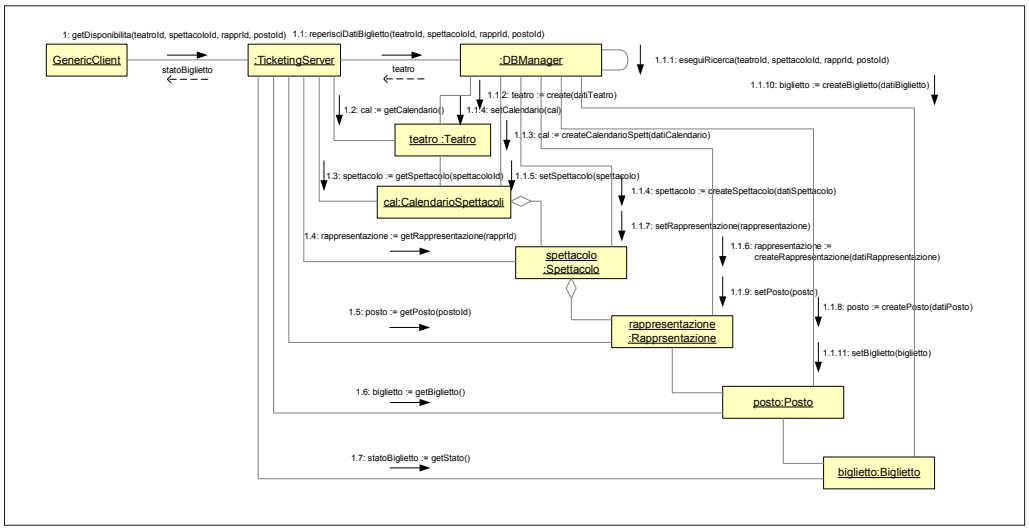
- rappresentare formalmente i requisiti del sistema;
- realizzare una primissima versione del modello del disegno.

Per questo motivo gli oggetti illustrati presentano ancora un elevato grado di concettualità (tanto che spesso sarebbe più corretto considerare questi oggetti come componenti).

In particolare, nel modello presentato in figura, si assume che esista un macro oggetto, `DBManager`, in grado di eseguire ricerche avanzate nel database. I risultati sono restituiti attraverso grafi di oggetti adeguati alla rappresentazione OO dei dati reperiti. Trattandosi di un "componente" generico, restituisce risultati attraverso grafi di oggetti consistenti con i dati reperiti, anche qualora l'oggetto utilizzatore sia interessato ad un solo dato (come per esempio l'istanza biglietto dell'esempio in questione). L'oggetto `TicketingServer` rappresenta un controllore in grado di coordinare l'espletamento dei servizi richiesti, mentre `GenericClient` si presta a essere identificato come un gestore di interfacce utente (per esempio, in un sistema basato sull'architettura J2EE, potrebbe trattarsi di una `Servlet`).

Il Collaboration Diagram riportato in fig. 2.21 illustra la stessa funzionalità del precedente diagramma di sequenza (è possibile passare dall'uno all'altro molto agevolmente), solo che in questo caso l'aspetto a cui viene conferita maggiore enfasi è l'organizzazione strutturale degli oggetti. Per questa caratteristica, e per la capacità di mostrare senza grandi problemi molti oggetti nell'ambito di uno stesso diagramma senza disordinare il diagramma, i Collaboration Diagram vengono preferiti in fase di disegno.
Le differenze principali con la precedente versione sono:

Figura 2.21 — *Collaboration Diagram della funzione* `getDisponibilità`.



- manca un'esplicita dimensione dedicata al fattore tempo, per cui la sequenzialità dei messaggi è desumibile attraverso la relativa numerazione;

- sono mostrate esplicitamente le connessioni tra oggetti.

Il diagramma di collaborazione presentato in fig. 2.22 non presenta grosse novità. Si tratta della rappresentazione del comportamento dinamico del servizio di clonazione listini prezzo. La fase di riferimento è quella di analisi. I nuovi elementi introdotti sono il multiobject (oggetto mostrato con un effetto pila) utilizzato per mostrare un insieme di oggetti dello stesso tipo e l'iterazione dei messaggi evidenziati per mezzo della condizione di iterazione racchiusa tra parentesi quadre.

Statechart Diagram

I diagrammi di stato essenzialmente descrivono automi a stati finiti e pertanto sono costituiti da un insieme di stati, transazioni tra di essi, eventi e attività. Ogni stato rappresenta un periodo di tempo ben delimitato della vita di un oggetto durante il quale l'oggetto stesso soddisfa precise condizioni.

Figura 2.22 — *Collaboration Diagram a livello della fase di analisi relativo al servizio di generazione di un nuovo listino prezzi per clonazione.*

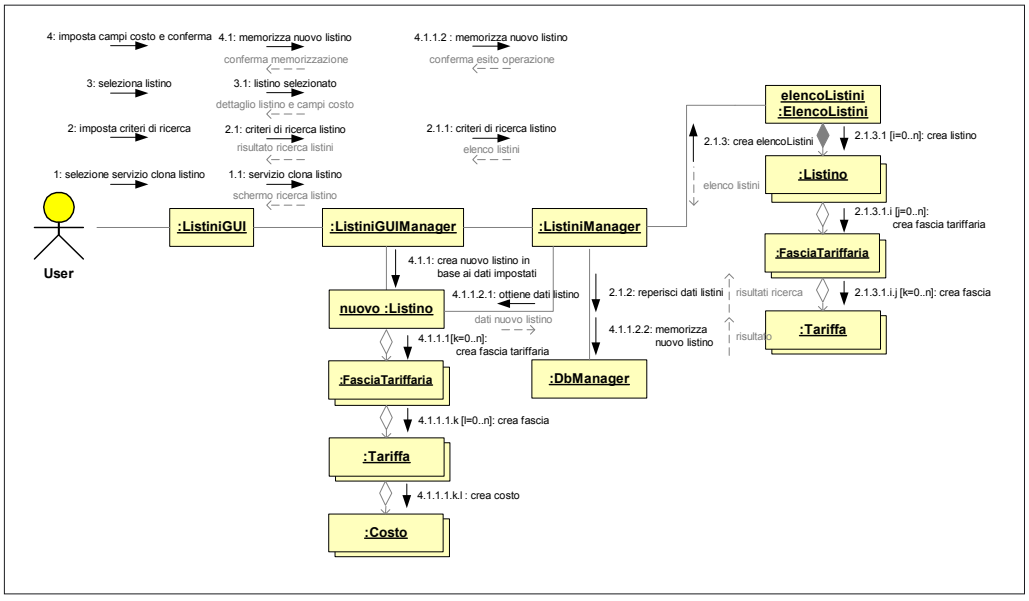
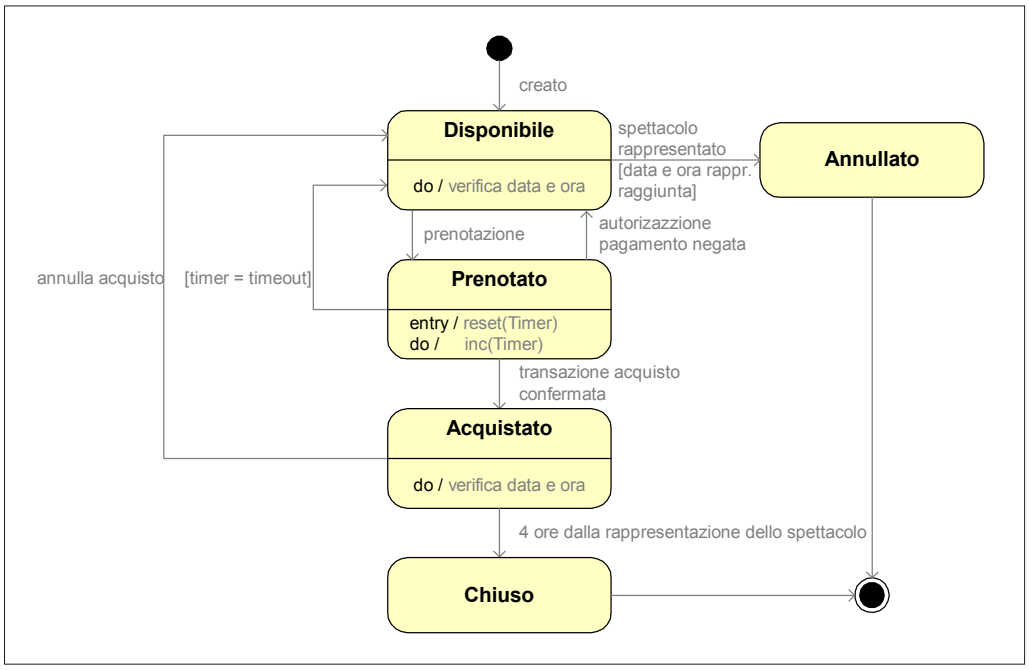


Figura 2.23 — *Statechart Diagram: ciclo di vita di un biglietto.*



I diagrammi degli stati pertanto, modellando la possibile storia della vita di un oggetto, sono utilizzati principalmente come completamento della descrizione delle classi: concorrono a modellarne il comportamento dinamico. Nulla vieta di utilizzarli anche nelle primissime fasi del processo di sviluppo del software al fine di documentare l'evoluzione attraverso i relativi stati di macrooggetti, con l'accorgimento di mantenere elevato il livello di astrazione.

Chiaramente ha senso utilizzare i diagrammi degli stati per dettagliare il comportamento delle sole classi i cui oggetti possono transitare per un ben definito insieme di stati.

Si consideri, per esempio, il ciclo di vita di un biglietto (fig 2.23). Un particolare oggetto biglietto, relativo ad una determinata rappresentazione di uno spettacolo, può trovarsi nello stato `Disponibile` (non è stato ancora acquistato o prenotato e lo spettacolo non è andato in scena), `Prenotato` (il biglietto è stato riservato ed il tempo a disposizione per acquistarlo non è scaduto), `Acquistato` (l'importo del biglietto è stato versato), `Annullato` (il tempo limite per l'acquisto è trascorso) e `Chiuso` (biglietto acquistato e raggiunto il limite massimo per l'annullamento).

Come si può notare esistono due tipi di transazioni:

- provocate dall'esterno (per esempio il Credit Card Authority autorizza la transazione di acquisto e quindi provoca la transizione del biglietto nello stato `Acquistato`);

- che scaturiscono internamente (scade il tempo a disposizione per poter acquistare il biglietto, e lo stesso transita nello stato di `Annullato`).

Il diagramma di fig. 2.23 risulta piuttosto comprensibile: unica nota è che nei vari stati possono essere descritte esplicitamente attività da compiersi all'atto dell'entrata nello stato (`entry`), durante la permanenza (`do`) ed in uscita (`exit`).

Per esempio si potrebbe avere la necessità di notificare l'uscita di un oggetto da un ben preciso stato. In quel caso si potrebbe associare la comunicazione di notifica all'uscita dallo stato (`exit`).

I diagrammi degli stati prevedono una notazione piuttosto ricca, oggetto di studio del Capitolo 10.

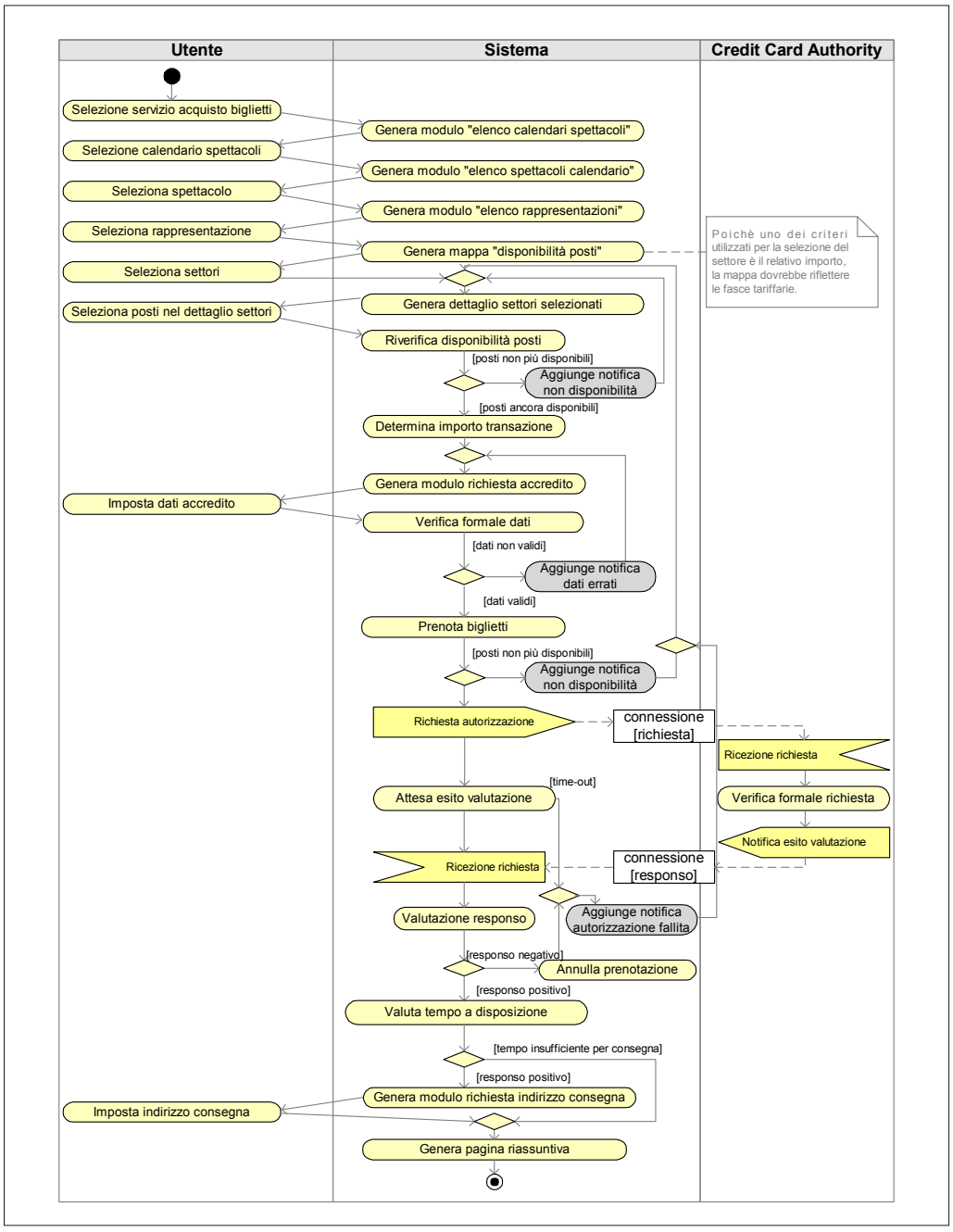
Activity Diagram

Gli Activity Diagram (diagrammi delle attività) mostrano l'evoluzione di un flusso di attività, ognuna delle quali è definita come un'esecuzione continua non atomica all'interno di uno stato. Tipicamente sono utilizzati per mostrare procedure e workflow. Si tratta di una variante dei diagrammi di stato (mostrati nel paragrafo precedente) in cui ogni stato rappresenta l'esecuzione di un'opportuna attività e la transizione da uno stato a quello successivo è generata dal completamento dell'attività stessa (trigger interno). Gli stati evidenziati negli Activity Diagram sono essenzialmente di due tipologie: Activity State (stati di attività) e Action State (stati di azione). I primi consistono in stati che eseguono una computazione la quale, una volta ultimata, genera la transizione allo stato successivo; uno stato di azione, invece, è uno stato atomico (ossia non può essere interrotto).

I diagrammi delle attività possono essere considerati una versione rivista e aggiornata dei "paleolitici" Flow Chart: per questo risultano particolarmente familiari a un vastissimo numero di persone anche con limitato skill tecnico. Logica conseguenza è che si dimostrano un valido ausilio per documentare funzionalità da sottoporre al vaglio di personale non tecnico: clienti e utenti.

Spesso li si utilizza nella fase di disegno, in particolare quando alcune decisioni implementative — tipo "quali attività assegnare agli oggetti istanza di una classe" — risultano difficili da prendere e/o è necessario mostrare processi paralleli. Infatti i dia-

Figura 2.24 — *Activity Diagram acquisto biglietti.*



grammi di attività permettono di enfatizzare la sequenzialità e la concorrenza degli step di cui si compone una particolare procedura.

Il diagramma di figura 2.24 mostra l'utilizzo della notazione dei diagrammi di attività come valido strumento di cattura e verifica dei requisiti utente. In particolare mostra il *main scenario* del servizio di acquisto biglietti teatrali.

Component Diagram

I Component Diagram mostrano la struttura fisica del codice in termini di componenti e di reciproche dipendenze. Insieme ai Deployment Diagram costituiscono la vista fisica del sistema e sono comunemente indicati con il nome generico di diagrammi di implementazione.

In particolare, i diagrammi dei componenti illustrano la proiezione statica dell'implementazione del sistema e sono pertanto strettamente connessi ai diagrammi delle classi. Ciascun componente rappresenta una parte del sistema, modulare, sostituibile, *deployable*, che incapsula implementazione ed espone un insieme di interfacce.

> Recenti indagini hanno evidenziato come la volatilità dei requisiti, tipica nel 90% dei sistemi, ha finito per minimizzare il concetto di riusabilità, una volta baluardo dell'Object Oriented. Allo stato attuale si preferisce parlare di sostituibilità dei componenti (in senso generale). La variazione dei requisiti può essere neutralizzata aggiornando o sostituendo uno o più componenti senza dover modificare altre parti del sistema non affette dalla variazione stessa.

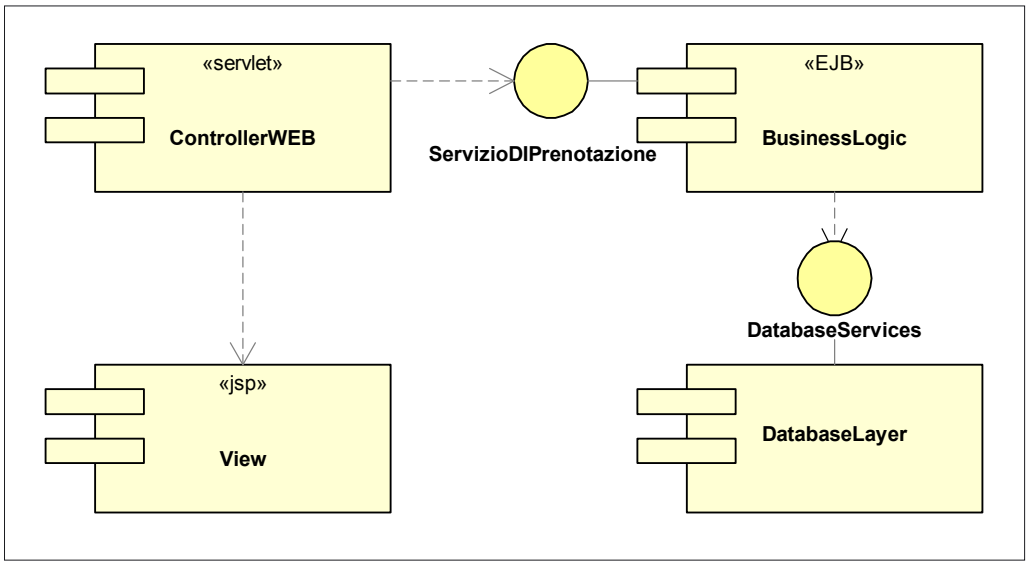
Un componente è tipicamente costituito da un insieme di elementi (interfacce, classi, ecc.) che risiedono nel componente stesso. Un certo numero di questi ne definisce esplicitamente le interfacce esterne, ossia la definizione dei servizi esposti e quindi forniti dal componente.

Un'area dello UML giudicata particolarmente carente dai vari esperti di sistemi Component-Based, che ha suscitato e continua a suscitare non poche critiche, è proprio quella relativa ai componenti. La definizione attuale fornita dallo UML è effettivamente un po' restrittiva; ciò però è abbastanza comprensibile se si pensa che la tecnologia dei sistemi Component-Based è relativamente recente.

L'approccio basato sui componenti è la logica evoluzione della filosofia OO ed è destinata ad avere un impatto profondo su disegno e implementazione dei sistemi. Probabilmente lo stesso UML finirà — magari non a brevissimo termine — per essere considerato un linguaggio di modellazione Component-Based piuttosto che semplicemente OO.

Con la versione 1.4 molto è stato fatto nella direzione dello studio di soluzioni atte a colmare le lacune evidenziate dallo UML nella modellazione di sistemi basati sui componenti. Ciò nonostante, ancora diverse carenze risultano irrisolte. Per esempio sarebbe opportuno definire profili da utilizzarsi per la modellazione di sistemi basati sulle architetture

Figura 2.25 — *Component Diagram del distributore automatico.*



Component-Based più famose (EJB, COM+ e CCM), proporre tecniche per modellare il contesto dell'esecuzione dei componenti (Containers EJB) e relative comunicazioni, individuare tecniche per modellare l'assemblaggio dei sistemi Component-Based e così via.

Nella fig. 2.25 è illustrato il diagramma dei componenti (concettuale) del sistema di ticketing del teatro. Si tratta di una primissima versione da raffinare durante la fase di disegno e da revisionare durante il processo di implementazione.

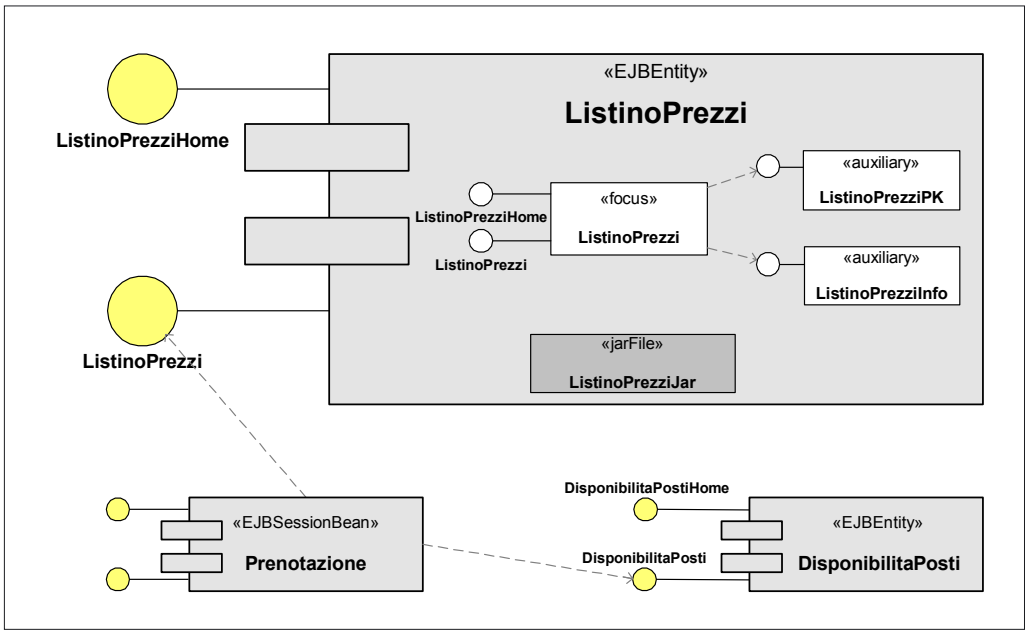
Probabilmente l'esempio è artificioso e i componenti sono assemblati per tecnologia, il che non sempre rappresenta l'idea migliore. In ogni modo è presentata la schematizzazione di una versione del Design Pattern MVC (Model View Control, modello vista e controllo): lo stesso utilizzato dalle Swing Java, adattato al Web. (Per maggori informazioni è possibile consultare il documento Sun's Blueprints for J2EE).

In particolare la o le Servlet dovrebbero incaricarsi essenzialmente di fornire dei servizi comuni come autenticazione, login, gestione degli errori, ecc. e di effettuare opportune redirezioni delle richieste dell'utente (il vero e proprio Controller).

Le richieste del client (browser Internet) dovrebbero giungere all'opportuna Servlet, la quale dovrebbe "girare" la richiesta alla classe appropriata (magari servizio EJB) atta alla relativa gestione: agisce in qualche modo da notificatore eventi (Events Dispatcher).

Le Servlet si dovrebbero anche incaricare di istanziare opportuni JavaBean da trasmettere (in qualche modo) al client insieme alla pagina HTML generata dall'appropriata JSP (pagine HTML con integrato del codice Java).

Figura 2.26 — *Esempio di specifica di un componente.*



Chiaramente la Business Logic del sistema dovrebbe essere fornita dagli Enterprise Java Bean: le Servlet dovrebbero utilizzare anche questi elementi per inserire dati in appositi JavaBeans. Nel diagramma è presentata una generica macrointerfaccia esposta dagli EJB; ovviamente si tratta di una rappresentazione simbolica: in realtà bisognerebbe evidenziare tutta una miriade di interfacce implementate dai relativi EJB.

Per terminare è mostrato un component definito Database Layer, in quanto si assume che la persistenza non venga gestita da Entity Beans bensì da Session. Nella fig. 2.26 è riportato un ulteriore esempio di Component Diagram. Brevemente, il diagramma mostra tre componenti: `Prenotazione`, `ListinoPrezzi` e `DisponibilitaPosti`. Verosimilmente, compito del primo componente è fornire una serie di servizi quali prenotazione di poltrone situate in specifici settori del teatro per determinate rappresentazioni di spettacoli, reperimento di informazioni relative al listino prezzi, verifica disponibilità, ecc.

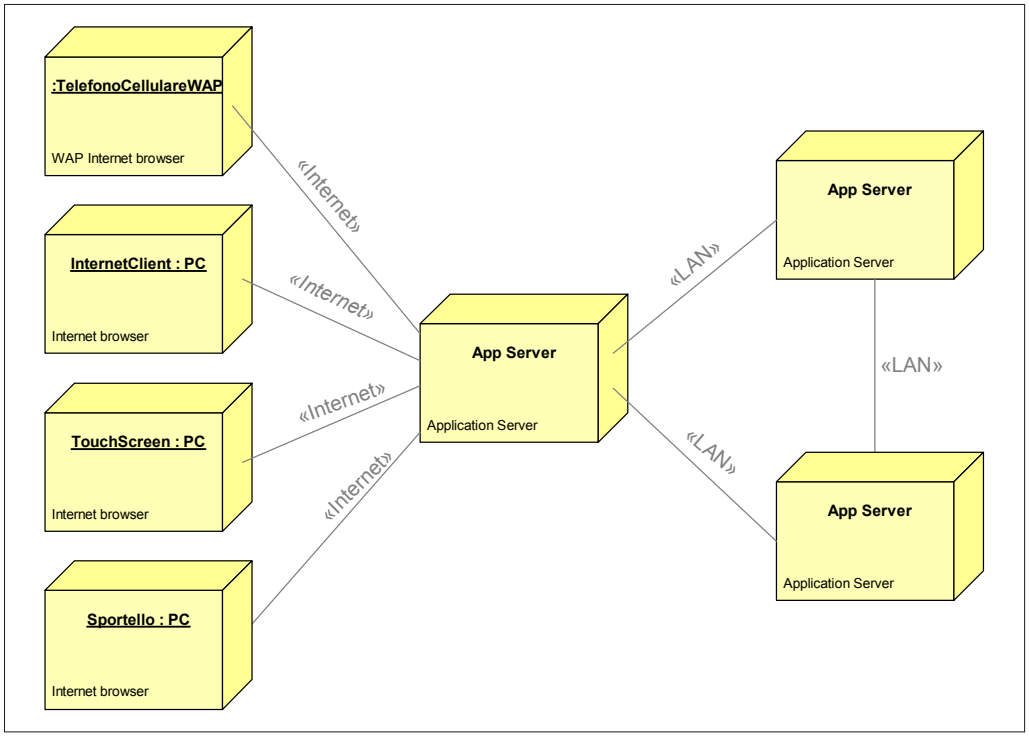
L'esempio, sebbene non completo — probabilmente sarebbe stato opportuno mostrare un maggior numero di componenti opportunamente organizzati — è stato proposto al fine di presentare alcuni elementi dello UML introdotti con la versione 1.4. Questi elementi risultano particolarmente utili per la progettazione e descrizione della struttura dei componenti.

In particolare gli stereotipi (in questo contesto si tratta di classi con particolare significato) `<<focus>>` e `<<auxiliary>>`, da utilizzarsi in coppia, permettono di distinguere classi *core* (fondamentali o centrali per il flusso di controllo) da altre di secondaria importanza (sempre dal punto di vista logico).

Nel diagramma in fig. 2.26 viene conferito particolare risalto alla struttura del componente `ListinoPrezzi`, costituito dalle classi `ListinoPrezzi`, `ListinoPrezziInfo`, `ListinoPrezziPK`. L'acronimo `PK` riportato come suffisso all'ultima classe indica un oggetto che ingloba la chiave primaria (PK, *Primary Key*). Sarebbe stato opportuno considerare un nome diverso, soprattutto al fine di non creare confusione con altre tecnologie, come per esempio quella dei database relazionali, ma questa tecnica permette di specificare l'identificatore univoco dei componenti entity EJB.

Da quanto descritto, si comprende come il componente generale sia realizzato attraverso l'utilizzo di due classi ausiliarie. Gli stereotipi `<<focus>>` e `<<auxiliary>>` risultano particolarmente utili per cominciare a progettare e descrivere componenti già nelle fasi di analisi e disegno direttamente nei diagrammi delle classi.

Figura 2.27 — *Esempio di diagramma di dispiegamento.*



Infine lo stereotipo `<<jarFile>>` permette di distinguere componenti eseguibili (EJB entity bean, EJB session bean, COM object) dai manufatti che li implementano (JAR file, DLL).

Deployment Diagram

I diagrammi di dispiegamento mostrano l'architettura hardware e software del sistema: ne vengono illustrati gli elementi fisici (computer, reti, dispositivi fisici, …) opportunamente interconnessi e i moduli software allocati su ciascuno di essi.

In sintesi viene mostrato il dispiegamento del sistema a tempo di esecuzione sulla relativa architettura fisica, in termini di componenti e relative allocazioni nelle istanze dei nodi.

Dall'analisi di fig. 2.27 è possibile individuare una serie di nodi, rappresentati attraverso parallelepipedi, collegati tra loro per mezzo di opportune connessioni (associazioni). I nodi sono elementi fisici esistenti a tempo di esecuzione che rappresentano risorse computazionali, mentre i collegamenti mostrano le relative connessioni fisiche.

I nodi vengono suddivisi in processori (*Processor*) e dispositivi (*Device*): la differenza risiede nel fatto che i primi dispongono di capacità elaborative (possono eseguire dei componenti), mentre i secondi no e, tipicamente, vengono utilizzati per rappresentare elementi di interfacciamento con il mondo esterno.

I nodi sono entità simili alle classi (nel metamodello UML ereditano dallo stesso genitore: classificatore), per cui è possibile utilizzarne le stesse potenzialità: ruoli, molteplicità, vincoli, ecc. Poiché diagrammi con soli nodi possono risultare decisamente anonimi e poco esplicativi, è possibile utilizzare delle rappresentazioni alternative utilizzandone "istanze" (`TelefonoCellulareWAP`, `TouchScreen`, ecc.) : un concetto del tutto simile a quello dei diagrammi degli oggetti. In tali varianti, è possibile corredare i nodi con un nome, indicarne i processi allocati, e così via.

Nel diagramma in fig. 2.27 è mostrata una prima versione del Deployment Diagram. Nelle versioni più dettagliate sarebbe corretto riportare anche i propri moduli: per esempio tutte le pagine HTML, JSP (Java Server Pages), le Servlet e i JavaBeans andrebbero installate nel `Web Server`, il core del sistema; i servizi EJB e il layer di interfacciamento al database invece andrebbero ubicati nell'`Application Server` e così via.

Meccanismi generali

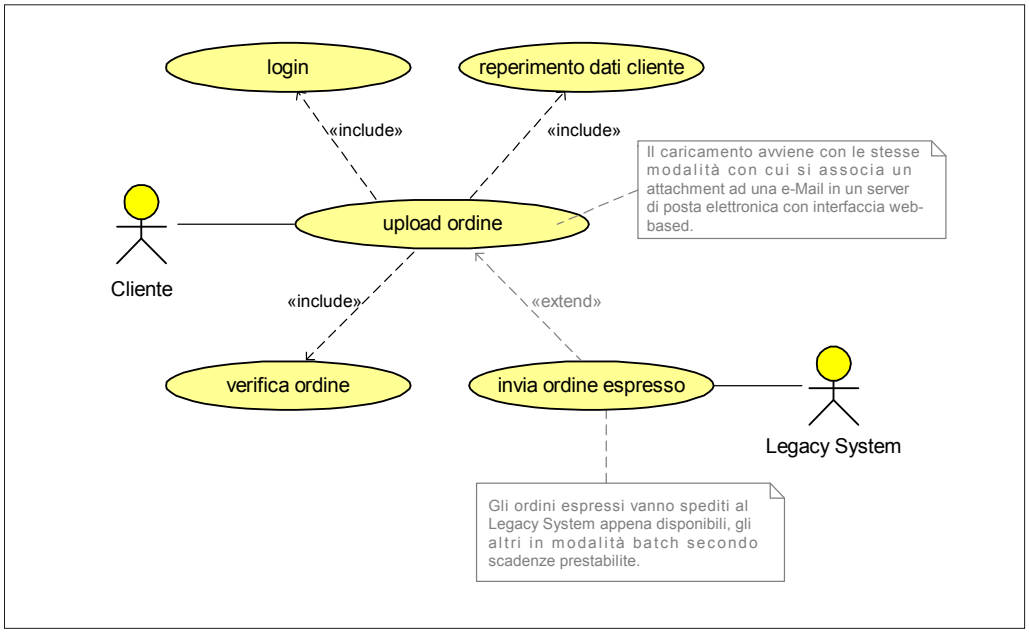
I successivi paragrafi illustrano i meccanismi generali — utilizzabili in qualsiasi vista e diagramma — forniti dallo UML per estenderne sintassi e semantica. Sebbene gli elementi del nucleo dello UML permettano di formalizzare molti aspetti di un sistema, è impossibile pensare che da soli siano sufficienti per illustrarne tutti i dettagli. Come spiegato in precedenza, lo UML fornisce sia meccanismi generali utilizzabili per aggiun-

gere informazioni supplementari difficilmente standardizzabili, sia veri e propri meccanismi di estensione.

L'esempio più classico, già incontrato nel corso di alcuni dei diagrammi presentati nei paragrafi precedenti, è quello delle **note** (*notes*): informazioni supplementari redatte con un formalismo che può variare, a completa discrezione del modellatore, dal linguaggio di programmazione, allo pseudocodice al testo in linguaggio naturale.

La selezione del livello di astrazione da utilizzare deve essere subordinata al miglioramento della leggibilità del diagramma. Poiché la leggibilità dipende anche dal fruitore, ne segue che il formalismo da utilizzarsi dipende dal destinatario principale del diagramma. Per esempio, un'annotazione in codice avrebbe poco significato nella Use Case View, dove il pubblico dei fruitori prevede anche i clienti che, per definizione, non hanno conoscenze tecniche, mentre dovrebbe risultare del tutto naturale in un Class Diagram, i cui fruitori sono tecnici qualificati.

Figura 2.28 — *Utilizzo delle note in uno Use Case Diagram.*



Graficamente le note sono rappresentate per mezzo di un foglio di carta stilizzato associato all'elemento che si vuole dettagliare tramite una linea tratteggiata. Nella fig. 2.28 le note sono state attaccate a due casi d'uso: `Upload ordine` e `Invia ordine espresso`, rispettivamente per specificare la modalità utilizzata per trasmettere un file da un browser client a un Web server e per evidenziare la differenza esistente tra gli ordini comuni e quelli espressi da trasmettere al legacy system appena disponibili. In questo caso le note contengono semplice testo in linguaggio naturale coerentemente con la vista di appartenenza e quindi con i relativi fruitori.

Si noti il ricorso al colore grigio chiaro sia per il testo, sia per il disegno della sagoma delle note. Esso è utilizzato al fine di non appesantire il diagramma e di non distogliere l'attenzione dagli elementi più importanti.

Un altro meccanismo generale è costituito dai cosiddetti **ornamenti** (*adornments*), che permettono di aggiungere semantica al linguaggio. Si tratta di elementi grafici atti a fornire al lettore una cognizione diretta di aspetti particolarmente importanti di specifici elementi.

Con riferimento alla fig 2.29, si può notare che il nome delle classi è stato scritto in corsivo per indicare che si tratta di classi astratte. Nella relazione (associazione) che lega l'`OggettoOsservato` all'`OggettoOsservatore` è presente un altro esempio di ornamento: la molteplicità. In una relazione tra classi, le molteplicità sono i numeri (o simboli) che specificano, per entrambe le classi dell'associazione, quante istanze possono essere coinvolte nella relazione con l'altra classe. Per esempio nel diagramma di fig. 2.29, un'istanza di tipo `OggettoOsservato` può disporre di diversi osservatori, così come un ogetto di tipo `OggettoOsservatore` può osservare diversi oggetti.

Un altro esempio di ornamento sono i simboli utilizzati per mostrare la visibilità di metodi ed attributi delle classi (`+` per elementi pubblici, `-` per privati, `#` per protetti, ecc). Da notare che nella versione 1.4 dello UML è stata aggiunta un'ulteriore visibilità: quella a livello di package (il `friendly` del C++) indicata con il simbolo tilde (`~ package`).

Ulteriore esempio di meccanismo generale è rappresentato dalle **specificazioni** (*specifications*): elementi di testo che aggiungono sintassi e semantica agli elementi dello UML. Un esempio di specificazione è fornito dall'indicazione dell'elenco degli attributi

Figura 2.29 — *Utilizzo delle note per illustrare le operazioni eseguite da un metodo.*

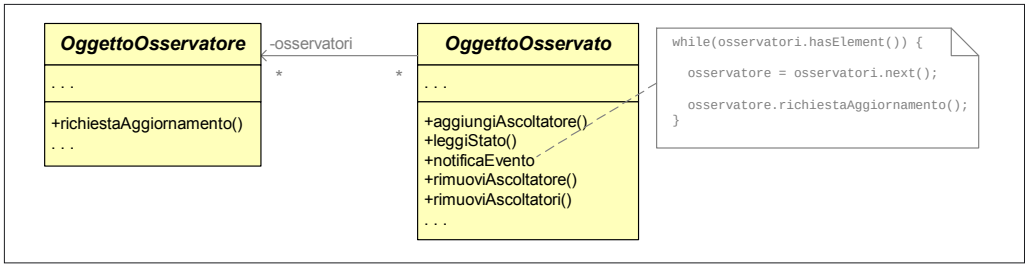
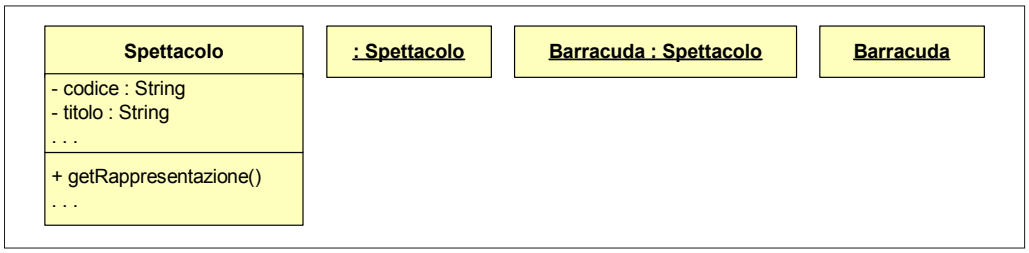


Figura 2.30 — *Distinzione tra classe e relative istanze (oggetti).*



e dei metodi (inclusa la firma completa) presenti nell'icona delle classi. Tipicamente per conferire ai vari diagrammi un livello migliore di chiarezza, vengono visualizzati solo determinati sottoinsiemi degli attributi e metodi dei vari elementi, chiaramente quelli ritenuti più utili nel contesto oggetto di studio. Ciò implica che per una stessa classe, in diversi diagrammi, possano essere visualizzati sottoinsiemi diversi di metodi e attributi.

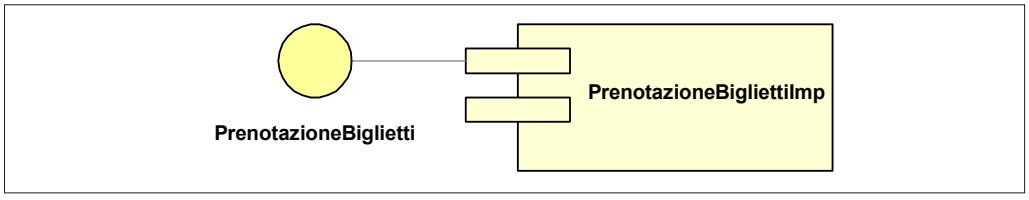
Ultimo esempio di meccanismo generale è costituito dalle **divisioni comuni** (*common divisions* o dicotomia tipo-istanza): nella modellazione di sistemi OO, ogni elemento reale può essere diviso, almeno, in due diverse tipologie.

Il primo esempio è costituito dalle classi e dagli oggetti: una classe è un'astrazione; un oggetto ne è il corpo, ossia una sua concreta manifestazione. Su questi argomenti si tornerà in dettaglio nel Capitolo 7 dedicato ai diagrammi delle classi e degli oggetti, per ora è sufficiente sapere che lo UML fornisce i meccanismi per mostrare la dicotomia presente in molti degli elementi (Use Case e istanze di Use Case, componenti e istanze di componenti. …).

In fig. 2.30 sono rappresentate la classe `Spettacolo` e tre sue istanze: nella prima viene indicata unicamente la classe di appartenenza (`Spettacolo` appunto), la seconda evidenzia anche il nome (`Barracuda :Spettacolo`), mentre la terza solo il nome.

Un altro esempio di divisione comune è fornito dalle interfacce e dalle relative implementazioni: un'interfaccia dichiara un contratto e una sua implementazione rappresenta una concreta realizzazione del contratto stesso, e quindi è responsabile della fornitura dei servizi in esso promessi (fig. 2.31).

Figura 2.31 — *Esempio di Interfaccia e relativa implementazione.*



Meccanismi di estensione

I meccanismi di estensione (Extensibility Mechanisms) consentono di aumentare sintassi e semantica dello UML al fine di permetterne il proficuo utilizzo in aree molto specifiche migliorando la leggibilità dei diagrammi prodotti e, al contempo, contenendo il livello di complessità del linguaggio stesso.

In particolare i meccanismi di estensione forniti dallo UML sono:

- Stereotypes (stereotipi);
- Tagged Values (valori etichettati);
- Constraints (vincoli).

Gli **stereotipi** permettono di estendere il vocabolario dello UML attraverso l'introduzione di nuovi elementi, definiti dall'utente, specifici per il problema oggetto di studio. Chiaramente la definizione degli stereotipi deve seguire regole ben precise: ciascuno di essi deve essere la specializzazione di un elemento presente nello UML (metamodello). In ultima analisi uno stereotipo è un nuova classe che viene (virtualmente) aggiunta al metamodello in fase di progettazione. Chiaramente non è assolutamente necessario andare a modificare il metamodello: è sufficiente che il nuovo stereotipo rispetti le regole previste.

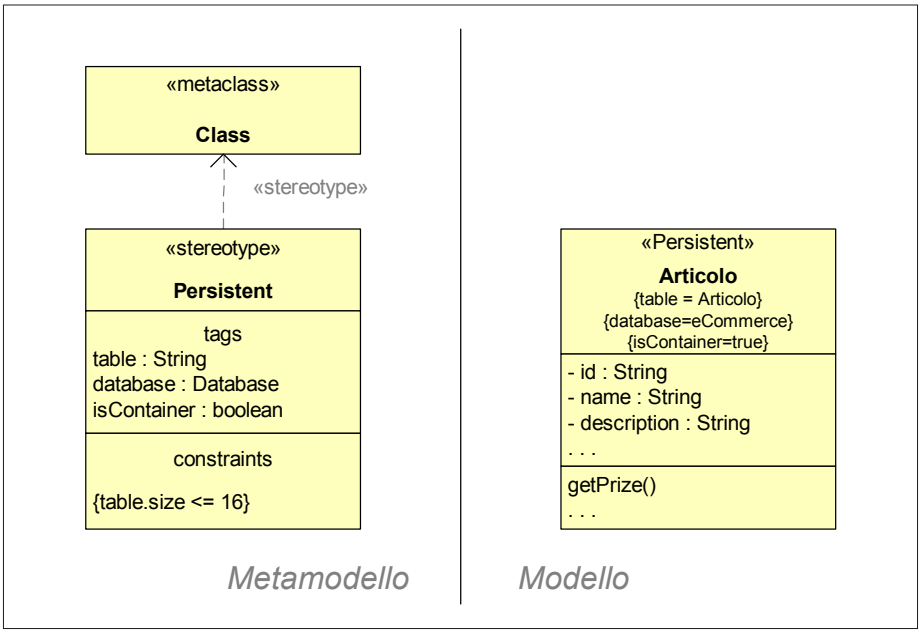
Spesso uno stereotipo è introdotto unicamente per disporre di icone più accattivanti (per esempio se un attore è un legacy system, invece di mostrare l'uomo stilizzato potrebbe risultare più chiaro mostrare un'icona con riportato un vecchio mainframe) o per evidenziare un'etichetta autoesplicativa (come per esempio nel caso di `focus` e `auxiliary`). Altre volte li si utilizza per aggiungere insiemi predefiniti di Tagged Value (associazione tra un attributo e il relativo valore) e vincoli validi per tutte le istanze dello stereotipo stesso. Ovviamente i vincoli, il nome e le altre caratteristiche aggiunte per mezzo di uno stereotipo, non devono entrare in conflitto con la definizione dell'elemento base (metaclasse o stereotipo) che si intende specializzare.

Un esempio di quanto detto è rappresentato dallo stereotipo `<<persistent>>` (fig. 2.32) dell'elemento classe (volendo essere formali si tratta di una specializzazione della metaclasse del metamodello denominata `Classe`). Esso si presta a essere utilizzato per inserire informazioni aggiuntive alle classi le cui istanze necessitino di essere memorizzate in forma permanente dal sistema.

La presenza della etichetta riportante con lo stereotipo (`<<persistent>>`), già da sola aiuta ad aumentare la chiarezza del modello. Infatti permette di distinguere immediatamente le classi (istanze dello stereotipo) i cui oggetti necessitano di essere memorizzati in modo permanente da quelle che invece non ne hanno bisogno.

Questo stereotipo presenta tutta una serie di elementi aggiuntivi, quali:

Figura 2.32 — *Definizione ed utilizzo di uno stereotipo (*`Persistent`*). Formalmente in UML uno stereotipo è rappresentato attraverso un'apposita (meta)Classe il cui stereotipo è* `<<stereotype>>` *appunto. Tale classe è la client di una relazione di dipendenza (a sua volta stereotipizzata con il termine* `<<stereotype>>`*) che la vincola all'elemento del metamodello che intende estendere. Nell'esempio di figura l'elemento che si intende estendere è la metaclasse* `Classe`*. I Tagged Value del nuovo elemento sono rappresentati per mezzo di appositi attributi, talune volte stereotipizzati come* `<<taggedvalue>>`*.*



- Tag: `storageMode` i cui elementi sono di tipo enumerato e possono assumere uno dei seguenti valori: `table`, `file`, `object`;

- Per gli oggetti persistenti su tabella, risulta inoltre possibile specificare informazioni aggiuntive come: Nome della tabella con il vincolo che sia minore di 16, database, ecc.

Chiaramente tutte le classi istanze dello stereotipo `<<persistent>>` devono fornire valori per i Tagged Value riportati poco sopra, nonché rispettarne il vincolo.

La descrizione formale degli stereotipi viene anche data in forma tabulare, per mezzo di apposita tabella (tab. 2.1). In essa sono presenti sei colonne, rispettivamente lo stereotipo (*Stereotype*), l'elemento del metamodello che si "specializza" (*Base Class*), l'elemento

Tabella 2.1 — *Descrizione formale degli stereotipi.*

Sterotype	Base class	Parent	Tags	Constraints	Description
Persistent	Class	N/A	table database isContainer	table.size() < 17	Indica classi le cui istanze devono essere memorizzate in forma permanente.

genitore (*Parent*), eventuali valori etichettati associati (*Tags*), eventuali vincoli (*Constraints*) e la descrizione (*Description*). La colonna *Parent* ha senso qualora lo stereotipo che si intende definire sia la specializzazione di un altro predefinito.

Questa tabella deve poi essere seguita da quella che definisce i Tagged Value associati allo stereotipo. Da tener presente che con la versione 1.4 dello UML si esorta a definire Tagged Value in associazione con gli stereotipi. In sostanza sarebbe obbligatorio, però tale obbligatorietà viene meno per questioni di compatibilità con le versioni precedenti.

Si vuole sottolineare ancora una volta che nella pratica non è assolutamente necessario dar luogo alla definizione formale dei vari stereotipi che si intende utilizzare nei propri modelli: essi possono essere utilizzati all'uopo. Chiaramente le definizioni rigorose diventano necessarie qualora gli stessi stereotipi debbano essere utilizzati da diversi team, in svariati progetti e così via.

Un altro classico esempio di stereotipo è quello delle classi di eccezione: sebbene siano classi a tutti gli effetti, i relativi oggetti sono trattati in modo del tutto particolare.

Gli stereotipi sono indicati o per mezzo del nome racchiuso tra doppie parentesi angolari (da definizione dello UML le doppie parentesi devono essere un solo carattere) oppure fornendo una nuova icona.

In fig. 2.33 viene mostrato come le istanze della classe `SourceParser` possano scatenare eccezioni del tipo `TokenNotValidException`. In questo caso si evidenzia che quest'ultima classe è un'eccezione sia dal nome sia dall'indicazione dello stereotipo (`exception` appunto) racchiuso tra doppie parentesi angolari.

Un'alternativa decisamente accattivante consiste nel definire una nuova icona ad hoc, (fig. 2.34). Invece di una "piatta" stringa `exception` racchiusa tra parentesi angolari si è utilizzata una colorata icona.

Figura 2.33 — *Esempio di stereotipo: classe eccezione.*

Ricapitolando, uno stereotipo può essere visualizzato in UML attraverso una serie di meccanismi, quali:

1. visualizzazione della sola stringa identificante lo stereotipo chiusa tra parentesi angolari doppie (fig. 2.33);

2. visualizzazione di un'apposita icona (fig. 2.34, in alto a sinistra);

3. visualizzazione della rappresentazione grafica prevista per l'elemento che si è esteso, in questo caso una classe, con aggiunta di un'icona specifica (fig. 2.34, in basso a sinistra);

4. visualizzazione sia del nome dello stereotipo, sia dell'icona nella rappresentazione grafica standard, un mix tra tecnica riportata nel punto1 e quella del punto 3 (fig. 2.34, in basso a destra).

Chiaramente è possibile utilizzare stereotipi per tutti gli elementi dello UML, come per esempio particolari Use Case, attributi di classi, messaggi, ecc.

Gli stereotipi possono essere utilizzati anche a livello di elementi più semplici come attributi e metodi (sempre metaclassi al livello di metamodello). Per esempio è possibile specificare che un determinato attributo è di sola lettura premettendo ad esso l'opportuno stereotipo.

Poiché uno stereotipo specializza un elemento base, ne segue che può essere utilizzato in ogni posto in cui viene utilizzato quest'ultimo (è a tutti gli effetti una generalizzazione).

Figura 2.34 — *Esempio di stereotipo: classe eccezione visualizzata attraverso le varie modalità dello UML.*

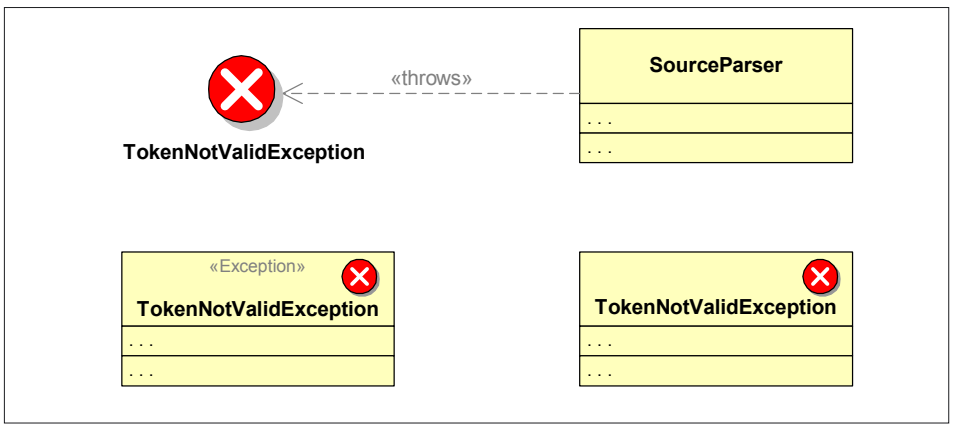
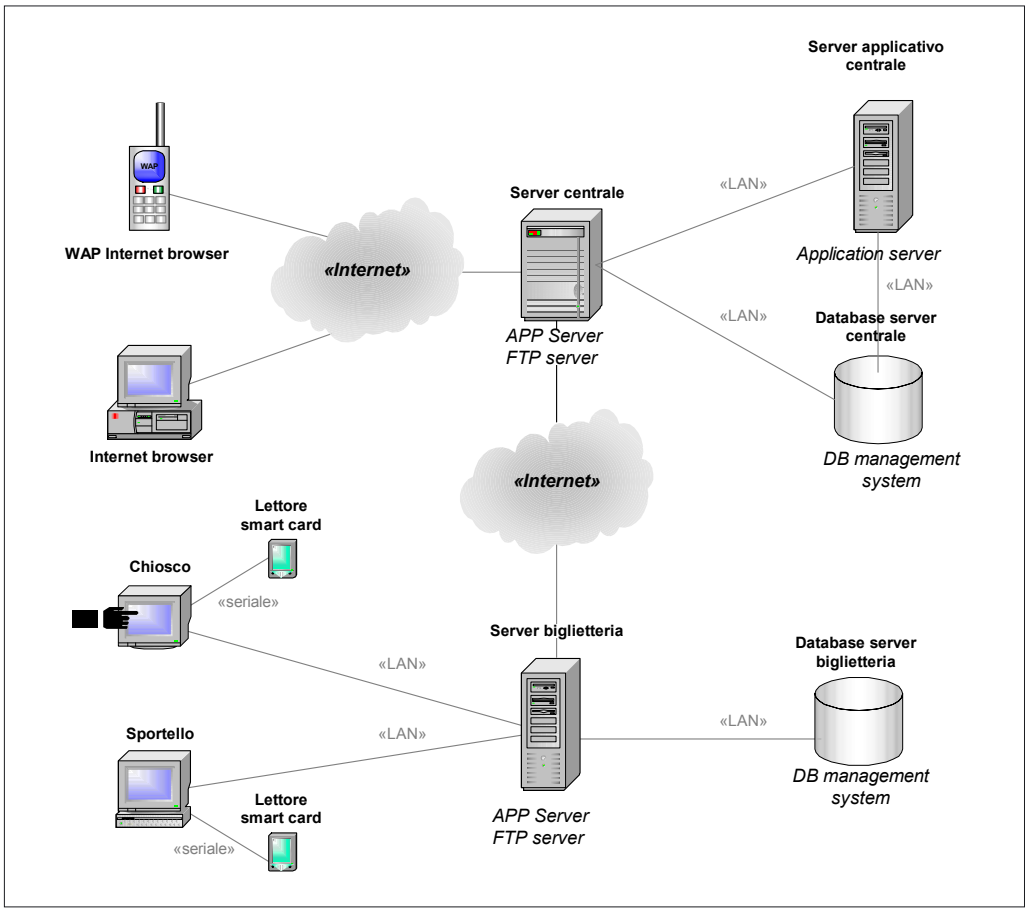


Figura 2.35 — *Esempio di deployment con l'utilizzo di opportuni stereotipi.*



Un ultimo esempio di utilizzo degli stereotipi è relativo al Deployment Diagram mostrato in fig. 2.35: in questo caso il ricorso agli stereotipi genera un effetto decisamente più incisivo degli spigolosi cubi definiti dallo UML, sebbene in un progetto professionale probabilmente il diagramma potrebbe risultare leggermente appariscente.

Il diagramma dovrebbe far riferimento a un'architettura gerarchica costituita da un'installazione centrale alla quale afferiscono sia gli utenti Internet, sia tutta una serie di biglietterie remote. L'architettura di queste è copia di quella presente nel sistema centrale: ciò al fine di non congestionare il sistema centrale per richieste che non coinvolgano dati relativi a prenotazione e acquisti di biglietti. Pertanto le varie biglietterie sono in grado di svolgere tutte le funzioni ordinarie in locale, limitandosi ad interagire con il sistema centrale solo per informazioni che variano in tempo reale.

Nel diagramma in fig. 2.35 sono presenti anche due *device*, rispettivamente il lettore della Smart Card e lettore di Magnetic Card, che si distinguono dai *processor*, in quanto non dispongono di capacità elaborative, almeno non nel senso tradizionale.

Tra i vari stereotipi introdotti, si può notare quello per l'associazione tra nodi remoti (la nuvola Internet), i database server mostrati per mezzo di un'icona a forma di cilindro, il telefono cellulare, il chiosco rappresentato da un *Touch Screen*, e così via.

I **Tagged Value** estendono le proprietà degli elementi dello UML consentendo di aggiungere informazioni supplementari a qualsiasi elemento. Sono a tutti gli effetti esempi di specificazioni. Si tratta di elementi costituiti dalla coppia nome–valore.

L'interpretazione della semantica dei valori etichettati esula — volutamente — dagli obiettivi dello UML. Si tratta di convenzioni interamente demandate al tool e/o all'utente. Non a caso vengono definiti metaattributi virtuali.

Per esempio un tool di disegno potrebbe utilizzare opportuni Tagged Value per definire in quale linguaggio effettuare la generazione del codice. Si possono impostare informazioni specifiche relative a metodi, allo stato di avanzamento del modello, dati necessari ad altri strumenti di sviluppo e così via. Per esempio, è possibile specificare la versione della revisione di un diagramma o di un suo elemento. A tal fine, è sufficiente impostare sotto il relativo nome la stringa : *{*`versione = 1.02`*}*.

Con la versione 1.4 dello UML i Tagged Value diventano tipizzati: le relative istanze possono assumere solo valori vincolati, come mostrato nel caso dello stereotipo `Persistent`. Sempre con la versione 1.4 è stata definita una nuova metaclasse nel package `Extension Mechanisms` del metamodello dello UML denominata `tagDefinition`. Questa specifica l'insieme di Tagged Value che possono essere associati a un elemento. Si tratta di un espediente tecnico per invogliare i progettisti a associare insiemi di valori etichettati a opportuni stereotipi. In altre parole, qualora sia necessario definire dei Tagged Value, dovrebbe essere obbligatorio definire anche un opportuno strereotipo che li raggruppi. Il condizionale è dovuto al fatto che, per questioni di compatibilità con le versioni precedenti dello UML, è ancora possibile definire Tagged Value non associati ad alcuno stereotipo. Inoltre la convenzione sancisce che ogniqualvolta un Tagged Value sia di tipo booleano, la relativa presenza implica un valore `true` (per esempio riportare `isAbstract = true` o semplicemente `isAbstract` è completamente equivalente), mentre l'omissione indica un valore `false`.

La definizione formale dei Tagged Value dovrebbe essere effettuata per mezzo di una tabella come la tab. 2.2. In essa sono presenti cinque colonne, rispettivamente: valore etichettato (*Tag*), lo stereotipo a cui è associato (*Stereotype*), il tipo (*Type*), la molteplicità (*Multiplicity*) e la descrizione (*Description*).

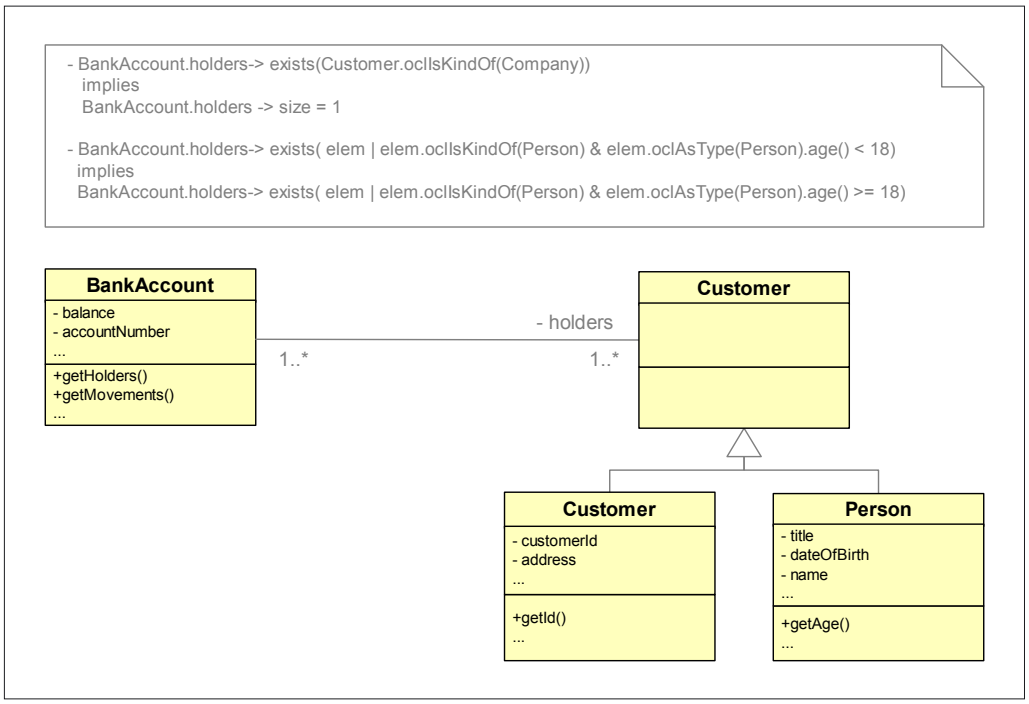
Ulteriore esempio di meccanismo di estensione — si fa per dire — fornito dallo UML è costituito dai **Constraints** (vincoli) che rappresentano restrizioni della semantica dei modelli. Una delle migliori definizioni formali sancisce che un vincolo è una restrizione di uno o più valori (o parti) di un modello OO o di un sistema.

Tabella 2.2 — *Definizione formale dei Tagged Value.*

Tag	Stereotype	Type	Multiplicity	Description
Table	Persistent	UML::Datatypes::String	1	Nome della tabella in cui memorizzare in forma permanente l'oggetto
Database	Persistent	Database::DataBaseEnum(enumeration:{eCommerce, Security})	1	Nome del database a cui appartiene la tabella
isContainer	Persistent	UML::Datatypes::Boolean	1	Valore `true` indica che la persistenza è gestita dal container.

Tipicamente un vincolo può essere rappresentato attraverso qualsiasi formalismo: la selezione di quello ritenuto più idoneo è completamente demandata al progettista. Da notare però che, nella versione 1.3 dello UML è stato incorporato ufficialmente lo OCL (Object Constraint Language) dell'IBM, linguaggio Object Oriented per la formulazione di vincoli esenti da effetti collaterali.

Figura 2.36 — *Esempio di diagramma delle classi atto a rappresentare la relazione tra clienti di una banca e conti correnti.*



Volendo essere precisi occorre dire che il primo embrione dell'OCL va fatto risalire a Syntropy.

Quindi, sebbene non sia strettamente obbligatorio definire tutti i vincoli per mezzo dell'OCL, l'utilizzo è fortemente consigliato.

L'OCL si presta a essere utilizzato per esprimere rigorosamente anche altri concetti presenti nei vari diagrammi dello UML, come precondizioni e postcondizioni (particolarmente utili per il Design By Contract), condizioni di guardia, transizioni di stato, invarianti, e così via.

Con il termine *invarianti* si indicano particolari classi, tipi ecc. le cui istanze sono caratterizzate dal fatto che la valutazione di una specifica espressione fornisce per tutte lo stesso risultato (`true`), durante l'intero ciclo di vita. In altre parole, invariante indica che l'espressione associata a una classe o tipo deve fornire lo stesso risultato per tutte le rispettive istanze in ogni istante del relativo ciclo di vita. Tipicamente le classi invarianti vengono evidenziate attraverso apposito stereotipo `<<invariant>>`.

Un primo consiglio proveniente dall'esperienza quotidiana: probabilmente non è necessario possedere una conoscenza approfondita dello OCL; ovviamente occorre averne le basi concettuali e conoscerne i costrutti principali. Verosimilmente il sistema migliore per utilizzarlo è acquistare una copia del libro [BIB11] e consultarla ogniqualvolta si abbia la necessità di comprendere e/o esprimere formalmente un vincolo. Da tener presente però che la versione incorporata nelle specifiche ufficiali dello UML è sensibilmente diversa da quella illustrata nel libro [BIB11], per cui è sempre opportuno avere qualche riscontro. I Constraint formulati tramite l'OCL sono decisamente molto eleganti, espressi per mezzo di un linguaggio basato sul paradigma OO, definito attraverso una grammatica rigorosa e chi più ne ha più ne metta. Ma, ahimè, a fronte di tutti questi bellissimi vantaggi, esiste uno svantaggio che rischia di invalidarli tutti: "quanti tecnici del proprio team conoscono e riescono a comprendere la semantica di vincoli espressi per mezzo del'OCL?". Si tratta ovviamente di una domanda retorica. Proprio questo è lo svantaggio: ad eccezione di limitati casi semplici, pochissimi tecnici riescono a comprendere vincoli espressi per mezzo della grammatica OCL. Quanti, nel vostro team, padroneggiano l'OCL? Allora cosa fare? Secondo consiglio — sempre gratuito — è di esprimere comunque i vincoli in modo formale, anche perché ciò conferisce un aspetto di maggiore professionalità, ricordando però contestualmente che i vincoli sono inseriti per essere compresi e quindi è comunque necessario spiegarli attraverso il linguaggio naturale.

Si prenda in considerazione il diagramma riportato nella fig. 2.36. Si tratta di un semplice esempio atto ad illustrare la relazione esistente tra i conti correnti delle banche e i relativi clienti. Il diagramma sancisce che un conto corrente può essere intestato a più clienti così come ciascuno di essi può possedere diversi conti correnti (relazione "*n* a *n*"). Il primo vincolo espresso attraverso l'OCL sancisce che, qualora un conto corrente sia intestato a una società, non è possibile disporre di altri cointestatari.

```
BankAccount.holders->exists(Customer.oclIsKindOf(Company))
  implies  BankAccount.holders->size  =  1
```

In particolare la prima riga afferma che se nella collezione degli intestatari (`holders`) del conto corrente è presente una società, allora tale collezione prevede un solo elemento: la società stessa.

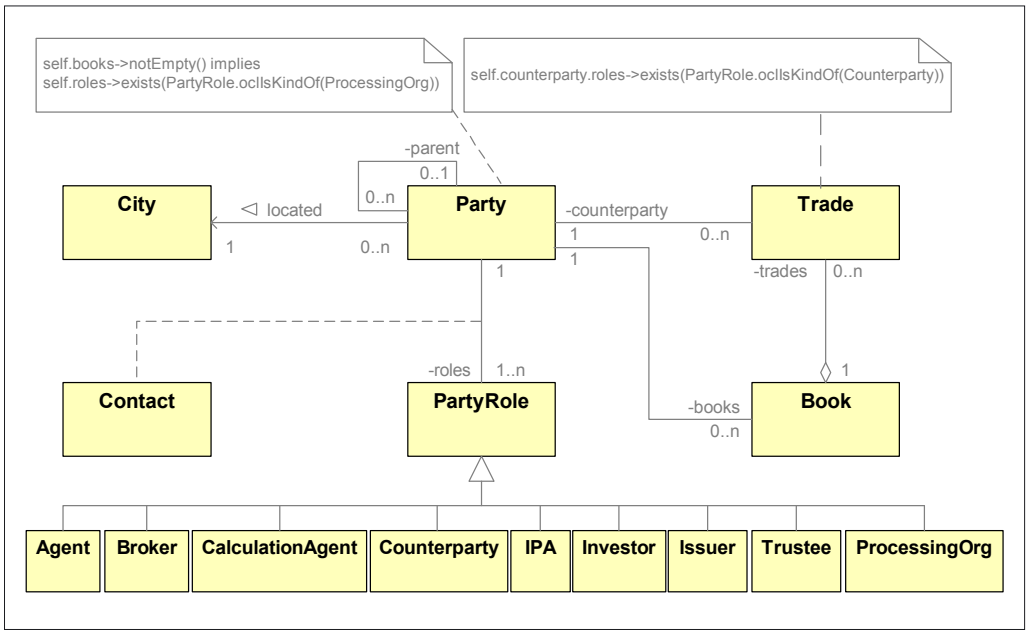
Il secondo vincolo risulta decisamente più interessante. Afferma che se uno degli intestatari di un conto corrente è un minorenne, allora lo stesso conto corrente deve necessariamente essere intestato anche a un'altra persona, questa volta maggiorenne.

```
BankAccount.holders->exists(
     elem|elem.oclIsKindOf(Person)  &  elem.oclAsType(Person).age()  <  18)
implies
BankAccount.holders->exists(
     elem|elem.oclIsKindOf(Person)  &  elem.oclAsType(Person).age()>=  18)
```

Si consideri ora il diagramma delle classi riportato in fig. 2.37. Si tratta di una porzione di un diagramma proveniente direttamente dal mondo degli investimenti bancari (Treasury System). Diagrammi del tipo di quello in figura servono a rappresentare le entità dell'area oggetto di studio (business) e tipicamente sono indicati come modelli a oggetti del dominio (DOM, Domain Object Model). Il relativo obiettivo, come illustrato nel Capitolo 8, è mostrare le varie entità e le relative associazioni presenti nel mondo concettuale che il sistema software deve automatizzare (area business o dominio del problema). Visto e considerato il dominio del problema si è lasciato il diagramma in lingua inglese… *To Cesar what belongs to Cesar*… sebbene la prima banca sia stata ideata ed edificata in quel di Siena.

> Senza entrare troppo nei dettagli — in fondo l'obiettivo è quello di fornire un esempio di utilizzo dell'OCL — il diagramma può essere spiegato come segue.
>
> In primo luogo nel mondo degli investimenti esistono diverse entità (agenti, broker, dipartimenti di banche, clienti, ecc.) aventi ruoli diversi ma comunque accomunabili sia perché condividono un certo sottoinsieme di comportamento sia per evitare di avere miriadi di oggetti simili sparsi per il sistema. Queste entità sono rappresentate dalla classe denominata genericamente `Party`. Da tener presente che il raggruppamento delle relative

Figura 2.37 — *Diagramma concettuale relativo all'entità party e relative associazioni.*



istanze è necessario anche perché ciascuna di esse può recitare diversi ruoli: l'associazione `plays` che connette la classe `Party` a quella `PartyRole` prevede cardinalità (1, n).

Per ogni ruolo che una classe `Party` recita è necessario disporre delle informazioni inerenti i relativi recapiti (`Contact`): indirizzo, telefono, fax, e-mail, indirizzo per la messaggistica interbancaria, ecc.

Un'entità `Party` poi è localizzata in una specifica città (`City`) e può essere strutturata secondo un'apposita organizzazione gerarchica: ogni oggetto `Party` può essere associato a un altro per mezzo dell'associazione `parent`. Una stessa banca per esempio può essere suddivisa in dipartimenti, ognuno dei quali organizzato altri dipartimenti e così via.

A questo punto si considerino le informazioni più direttamente legate agli investimenti. Ciascuno di essi (`Trade`) deve possedere almeno due entità: sorgente (chi vende) e destinatario (chi acquista). Nel caso più generale possono essere coinvolte molte più entità: agenti, intermediari ecc. Questi però sono specificati attraverso altre entità (non riportate nel diagramma) denominate "istruzioni per l'erogazione delle liquidazioni" (Standing Settlement Instructions), che si applicano agli investimenti in funzione dell'ennupla (sorgente, destinatario, prodotto finanziario, moneta, ecc.).

Ogni `trade` appartiene a un solo `book` che tra l'altro permette di individuare il dipartimento che ha stipulato l'investimento.

Il primo vincolo (quello più a sinistra) sancisce che solamente `Party` che svolgono anche funzioni di `Processing Organization` possono amministrare dei `book`.

```
self.books->notEmpty() implies
   self.roles->exists(PartyRole.oclIsKindOf(ProcessingOrg))
```

Precisamente, il vincolo sancisce che se un `Party` gestisce almeno un `book` ne segue che nella lista dei ruoli recitati dal `Party` stesso deve necessariamente comparire quello di `Processing Organization`.

Il secondo vincolo invece è associato alla classe `Trade`. Sancisce che un `trade` può prevedere come controparte unicamente un'istanza della classe `Party` che recita appunto il ruolo di controparte:

```
self.counterparty.roles->exists(PartyRole.oclIsKindOf(Counterparty))
```

Formalmente, nella lista dei ruoli recitati dalla classe `Party` collegata al `trade` per mezzo dell'associazione `counterparty` deve essere necessariamente presente il ruolo di `Counterparty`. Quindi un'istanza di *Trade* rappresenta un *deal* (contratto, affare) stipulato tra uno specifico dipartimento della banca (`ProcessingOrg`), quello che amministra il `Book` a cui il `Trade` è associato, e una controparte.

L'esempio di figura 2.36 è stato introdotto esclusivamente a fini espositivi: nulla vieterebbe di associare gli oggetti `Trade` direttamente alle specializzazioni `ProcessingOrg` e `CounterPary` eliminando alla base la necessità di ricorrere ai vincoli.

Ultimo esempio di meccanismo di estensione è costituito dai **Profili** (illustrati anche nei paragrafi seguenti) che con la versione 1.4 sono entrati a far parte integrante del linguaggio. Attualmente ne sono stati formalmente incorporati due: il profilo per i Software Development Processes (processi di sviluppo del software) e quello per il Business Modeling (modellazione dell'area business).

Un profilo è uno stereotipo di un package contenente un insieme di elementi del modello opportunamente adattati a uno specifico dominio o scopo. Tipicamente lo si ottiene estendendo il metamodello tramite l'uso adeguato di meccanismi di estensione forniti dallo UML (stereotipi, definizioni di Tagged Value, vincoli e icone). In tali casi si parla di profili "leggeri" (*lightweight*), in quanto ottenuti per mezzo dei meccanismi di estensione propri del linguaggio, in contrasto con quelli "pesanti" (*heavyweight*) ottenibili per mezzo dei meccanismi di estensione definiti dalle specifiche del MOF.

La definizione formale di un profilo, per essere consistente all'approccio utilizzato nel documento delle specifiche ufficiali, deve prevedere le seguenti sezioni:

1. insieme delle estensioni standard che definiscono il profilo stesso, ottenuto attraverso l'opportuna applicazione dei meccanismi di estensione dello UML al sottoinsieme di interesse del metamodello;

2. definizione della relativa semantica descritta attraverso il linguaggio naturale (americano, ovviamente);

3. insieme delle regole ben formate (*well-formed rules*), ossia insieme di vincoli, espressi in linguaggio naturale, e qualora possibile per mezzo dell'OCL, appartenenti agli elementi introdotti con il profilo;

4. elementi comuni del modello (*common model elements*), ossia istanze predefinite di elementi del metamodello UML. La definizione di tali elementi può far uso delle estensioni standard definite nel profilo stesso secondo gli eventuali vincoli in esso presenti;

5. eventuali estensioni al MOF (Meta Object Facility). Questa sezione, qualora presente, determina la transizione del profilo dalla caratterizzazione "leggero" a quella "pesante". Si ottiene definendo nuove meta-classi da incorporare nella definizione formale del MOF. Chiaramente, si tratta di estensioni delicate a cui ricorrere solo dopo aver appurato l'inadeguatezza ai fini richiesti dei meccanismi standard.

È possibile pensare ai profili come a librerie di elementi o a plug-in da installare, contenenti insiemi predefiniti di elementi ottenuti dall'estensione di quelli base, da utilizzarsi per modellare specifiche aree business o per determinati scopi.

Figura 2.38 — *Stereotipi di classi definiti nel profilo dei Software Development Processes.*

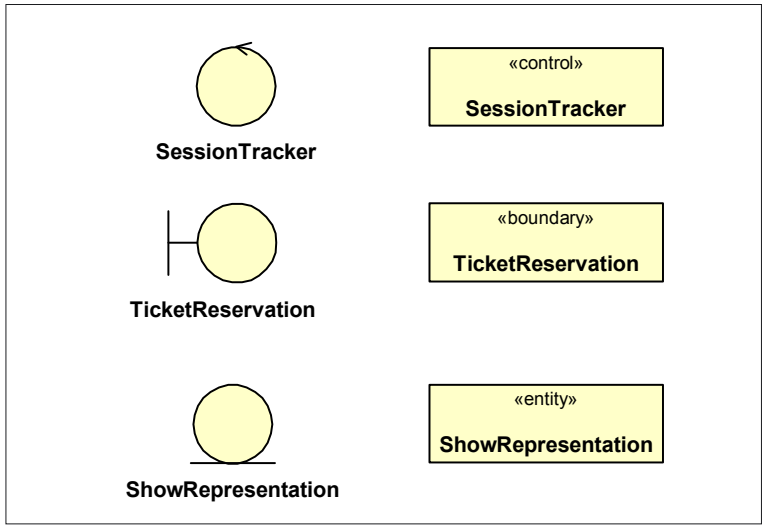


Tabella 2.3 — *Stereotipi presenti nel profilo dei Software Development Processes.*

nome stereotipo	elemento base
UseCaseModel	Model*
AnalysisModel	Model
DesignModel	Model
ImplementationModel	Model
UseCaseSystem	Package
AnalysisSystem	Package
DesignSystem	Subsystem**
ImplementationSystem	Subsystem
AnalysisPackage	Package
DesignSubsystem	Subsystem
ImplementationSubsystem	Subsystem
UseCasePackage	Package
AnalysisServicePackage	Package
DesignServiceSubsystem	Subsystem
Boundary	Class
Entità	Class
Control	Class
Communicate	Association
Subscribe	Association

** Il modello cattura una particolare vista di un sistema fisico, in altre parole, ne è una astrazione realizzata per scopi ben precisi. In particolare lo scopo permette di determinare quali elementi sia interessante riportare e quali invece possano essere trascurati. È possibile definire diversi modelli per uno stesso sistema, dove ciascuno ne rappresenta una vista definita da propri obiettivi e dal relativo livello di astrazione (per esempio modello di analisi, modello di disegno, modello di implementazione, e così via). Tipicamente diversi modelli risultano tra loro complementari e vengono definiti dalle prospettive (Viewpoints, punti di vista) dei diversi fruitori del sistema. Poiché nel metamodello la metaclasse modello — che scioglilingua! — eredita da* `Package`*, ne consegue che rappresenta un raggruppamento di elementi opportunamente organizzati secondo strutture gerarchiche (*`Package` *eredita dal* `Namespace` *e dal* `GeneralizableElement`*).*

*** Un sottosistema (Subsystem) è un raggruppamento di elementi che rappresentano un'unità, aggregabile dal punto di vista del comportamento, del sistema fisico. I sottosistemi sono in grado di esporre opportune interfacce e dispongono di operazioni. Gli elementi che costituiscono i sottosistemi vengono divisi in due categorie: elementi di specifica e quelli di realizzazione. Nel metamodello l'elemento* `Subsystem` *presenta una ereditarietà multipla: eredita sia dal* `Package` *sia dal* `Classifier`*. Quindi oltre alle caratteristiche proprie del* `Package`*, esso possiede tutta una serie di* features *quali operazioni, associazioni, ecc.*

Nella fig. 2.38 vengono riportati alcuni esempi di elementi presenti nel profilo dei Software Development Processes. Questi sono solo alcuni esempi di stereotipi presenti nel profilo: la lista completa è presente nella tab. 2.3.

> Per quanto riguarda gli stereotipi delle classi riportati in figura, non si tratta di elementi nuovi; li si può trovare, tra l'altro nel libro dei Tres Amigos dedicato al processo di sviluppo del software ([BIB08]). Essi si prestano a essere utilizzati nel modello di analisi con i seguenti significati:
>
> `<<entity>>`
>
> Il significato di questo stereotipo è abbastanza intuitivo: gli oggetti entità contengono informazioni che necessitano di essere memorizzate in forma permanente. Quindi, tipicamente, rappresentano entità che esistono (o traspirano) nel mondo concettuale oggetto di studio: si tratta prevalentemente, di raffinamenti delle classi presenti nel modello del dominio. Le istanze dello stereotipo `<<entity>>` sono oggetti passivi (per esempio non iniziano una interazione per loro iniziativa) e tipicamente partecipano alla realizzazione di molti Use Case e sopravvivono alle singole interazioni.
>
> `<<control>>`
>
> Le istanze di questo stereotipo gestiscono interazioni tra collezioni di oggetti. Tipicamente hanno comportamento specifico per un singolo Use Case e i relativi oggetti non sopravvivono a una specifica interazione a cui prendono parte. Queste classi sono anche utilizzate per rappresentare complessi algoritmi di calcolo, regole appartenenti all'area business, ecc.
>
> `<<boundary>>`
>
> Le istanze di questa classe "vivono" nel confine *interno* del sistema, e il relativo compito, tipicamente, consiste nel gestire l'interazione tra attori esterni e altri oggetti boundary, entity e control interni al sistema stesso. Quindi si tratta di parti del sistema che dipendono dai relativi attori e in particolare catturano i requisiti per determinate interazioni.

I profili attesi

Come visto in precedenza, i profili sono delle particolari estensioni dello UML, o meglio collezioni di estensioni predefinite, le quali nascono dall'esigenza di standardizzare l'utilizzo dello UML per domini o scopi o tecnologie ampiamente diffuse come per esempio le architetture CORBA, EJB e COM+.

Lo UML di per sé è un metamodello e quindi una notazione a scopo generale (*general purpose*) particolarmente indicata per esprimere modelli ad oggetti. Uno degli intenti che fin dall'inizio hanno ispirato il lavoro dei Tres Amigos è stato di renderlo così generico da poter essere utilizzato proficuamente in un'estesa varietà di ambienti. Poiché però era impossibile riuscire a immaginare i requisiti di tutti i possibili ambienti fin dall'inizio, è stato deciso di dotare lo UML di meccanismi che ne consentissero l'estensione (consultare i paragrafi precedenti) garantendo così un efficace utilizzo in ogni ambiente specifico.

Il problema emerso era che le estensioni da apportare allo UML al fine di adattarlo a tecnologie particolarmente ricorrenti in progetti OO erano completamente demandate ai progettisti. Il rischio, ancora una volta, era quello di una nuova proliferazione di "plug-in" non standardizzati e completamente disomogenei che chiaramente finivano per minare i grandi vantaggi offerti dallo UML, derivanti dall'elevato grado di standardizzazione.

Per esempio, se si fosse chiesto ad *n* architetti (esperti) di progettare un sistema basato sulla tecnologia EJB attraverso lo UML, verosimilmente tutti avrebbero realizzato altrettanti modelli perfettamente validi e formali ma, probabilmente, ciascuno avrebbe formulato un proprio insieme di estensioni, diverso da quello ideato degli altri. Si correva pertanto il rischio di generare ancora una volta una miriade di dialetti.

Un altro problema è legato, paradossalmente, a una delle caratteristiche più apprezzate del linguaggio: l'essere *general purpose*. Se da un lato è un vantaggio per i motivi più volte elencati, dall'altro la relativa flessibilità può creare non pochi problemi — e qui ce ne sarebbero di esempi da citare — soprattutto a un pubblico di utilizzatori UML non esperti. Molto spesso i progettisti sono sopraffatti dalla flessibilità del linguaggio e preferirebbero disporre di elementi pronti la cui validità sia stata già collaudata da altri.

Onde evitare tutto ciò, lo OMG (Object Management Group) ha deciso di lavorare alla standardizzazione di una serie di profili, tra i quali molto importanti sono quello CORBA e quello EJB per il quale, ovviamente, è stata richiesta l'approvazione della stessa Sun Microsytems.

Si tratta di un passo indietro? Tanto valeva, allora, definire tutti gli elementi direttamente dall'inizio. In realtà le cose non stanno così. La struttura dello UML risulta paragonabile a quella di un qualsiasi linguaggio di programmazione. Esiste un *core* ben definito, basato su un numero di elementi relativamente limitato, e quindi più facilmente accessibile, utilizzabile per costruire tutte le funzionalità di cui si ha bisogno, e in più è possibile utilizzare tutta una serie di librerie predefinite, di provato successo, addirittura gratuite.

L'architettura dello UML è un esempio di ingegneria del software, che possiede le caratteristiche più ambite: flessibilità, formalità, potenza, semplicità, ecc. I meccanismi di estensione, pertanto, rimangono strumenti comunque molto validi, la cui efficacia, tra l'altro, è stata provata durante il lavoro stesso di definizione di profili. Si tratta di un framework utilizzabile per estendere efficacemente il linguaggio per gli usi più svariati secondo direttive ben definite… E poi… nessuno vieta di definire ulteriori elementi non previsti dai profili.

Nell'Appendice C è riportata una breve illustrazione del profilo EJB.

Ricapitolando…

Il presente capitolo è stato dedicato alla disamina del linguaggio UML: una vera e propria overview. L'intento è iniziare a mostrare le potenzialità e i meccanismi dello UML senza avere assolutamente la pretesa di essere esaustivi: per ogni argomento presentato esiste un apposito capitolo in cui i vari concetti vengono illustrati dettagliatamente.

Procedendo dall'esterno verso l'interno, lo UML è composto da una serie di viste (Use Case, Design, Implementation, Component e Deployment) realizzate attraverso opportuni di diagrammi ognuno dei quali utilizza specifici elementi del modello.

La prima vista, Use Case View (vista dei casi d'uso), è utilizzata per analizzare i requisiti utente; specifica le funzionalità del sistema come vengono percepite da entità esterne allo stesso, dette attori.

La Design View (vista di disegno), descrive come le funzionalità del sistema devono essere realizzate; in altre parole il sistema viene analizzato dall'interno.

La Implementation View (vista implementativa, detta anche Component View, vista dei componenti), è la descrizione del modo in cui il codice (classi per i linguaggi Object Oriented) viene aggregato in moduli (package) e di come questi dipendano gli uni dagli altri.

La Process View (vista dei processi, detta anche Concurrency View, vista della concorrenza) rientra nell'analisi degli aspetti non funzionali del sistema, e consiste nell'individuare i processi e i processori.

La Deployment View (vista di "dispiegamento"), mostra l'architettura fisica del sistema e fornisce l'allocazione delle componenti software nella struttura stessa.

Per ciò che concerne i diagrammi, sono previste ben nove tipologie: Use Case, Class, Object, Sequence, Collaboration, StateChart, Activity, Component e Deployment.

Dall'utilizzo dello UML per rappresentare modelli di sistemi reali è possibile riscontrare due lacune: l'interfaccia utente e il disegno di database relazionali. Il problema è che non esistono formalismi per rappresentare due modelli decisamente importanti nella realizzazione di sistemi informatici come l'interfaccia utente e il disegno di database razionali. Per database OO è possibile utilizzare i diagrammi delle classi e quelli degli oggetti.

Sebbene il presente testo non si occupi di trattare in dettaglio i processi di sviluppo, si è comunque preso in considerazione un processo di riferimento al fine di illustrare in maniera più concreta l'utilizzo dello UML.

Spesso alcuni tecnici tendono a confondere lo UML con il processo di sviluppo: come ripetuto più volte lo UML è "solo" un linguaggio di modellazione e come tale si presta a rappresentare i prodotti generati nelle varie fasi di cui un processo è composto. Pertanto lo UML, al contrario dei processi, non fornisce alcuna direttiva su come fare evolvere il progetto attraverso le varie fasi così come non specifica quali sono i manufatti da produrre, chi ne è responsabile ecc.

Il problema è che non esiste un processo standard universalmente accettato e tantomeno esiste un accordo sulle varie fasi e sui relativi nomi: ogni processo va adattato alle caratteristiche delle specifiche organizzazioni, ai progetti, e così via.

Brevemente — ogni fase verrà ripresa nei capitoli successivi — i vari modelli da produrre sono:

- Modello business. Viene generato come risultato della fase di modellazione della realtà oggetto di studio (business modeling). L'obiettivo è capire cosa bisognerà realizzare (requisiti funzionali e

non), quale contesto bisognerà automatizzare (studiarne struttura e dinamiche; a tal fine, spesso, si utilizzano i famosi Domain e/o Business Model), comprendere l'organigramma dell'organizzazione del cliente, valutare l'ordine di grandezza del progetto, individuare possibili ritorni dell'investimento per il cliente, eventuali debolezze, potenziali miglioramenti, iniziare a redigere un glossario della nomenclatura tecnica al fine di assicurarsi che, nelle fasi successive, il team di sviluppo parli lo stesso linguaggio del cliente, e così via.

- Modello dei requisiti. Questo modello viene prodotto a seguito della fase comunemente detta analisi dei requisiti (Requirements Analysis). Scopo di questa fase è produrre una versione più tecnica dei requisiti del cliente evidenziati nella fase precedente.

- Modello di analisi. Questo modello viene prodotto come risultato della fase di analisi i cui obiettivi sono di produrre una versione dettagliata e molto tecnica della Use Case View accogliendo anche le direttive provenienti dalle prime versioni del disegno dell'architettura del sistema (che a loro volta dipendono da questo modello), realizzare un modello concettuale, analizzare dettagliatamente le business rule da implementare, revisionare il prototipo — o comunque una descrizione — dell'interfaccia utente.

- Modello di disegno. Anche in questo caso il modello di disegno è il prodotto della omonima fase, in cui ci si occupa di plasmare il sistema, trasformare in un modello direttamente traducibile in codice i vari requisiti (funzionali e non funzionali) forniti nel modello dei casi d'uso di analisi.

- Modello fisico. Il modello fisico, a sua volta, è composto essenzialmente da due modelli: Deployment e Component. Non si tratta quindi del prodotto di una fase ben specifica, bensì sono risultati di rielaborazioni effettuate in diverse fasi. Il modello dei componenti mostra le dipendenze tra i vari componenti software che costituiscono il sistema. Come tale, la versione finale di tale modello dovrebbe essere il risultato della fase di disegno. Il modello di dispiegamento (Deployment) descrive la distribuzione fisica del sistema in termini del modo in cui le funzionalità sono ripartite sui nodi dell'architettura fisica del sistema. Si tratta quindi di un modello assolutamente indispensabile per le attività di disegno e implementazione del sistema.

- Modello di test. Nella produzione di sistemi, con particolare riferimento a quelli di medio/grandi dimensioni, è opportuno effettuare test in tutte le fasi. Eventuali errori o lacune vanno eliminate prima possibile al fine di neutralizzare l'effetto delle relative ripercussioni sul sistema. È opportuno effettuare test prima di dichiarare conclusa ciascuna fase. In questo contesto si fa riferimento a uno specifico modello atto a verificare la rispondenza di ogni versione del sistema frutto di un'appropriata iterazione. Il modello di test dovrebbe essere generato non appena disponibile una nuova versione stabile dei casi d'uso. Nel realizzare il modello di test è necessario: pianificare i test richiesti a seguito di ciascuna iterazione, disegnare e realizzare i test attraverso i Test Cases che specificano cosa verificare, e le Test Procedure che illustrano come eseguire i test. Quando possibile, sarebbe

opportuno creare componenti eseguibili (Test Component) in grado di effettuare i test automaticamente, integrare i test necessari al termine dell'iterazione in corso con quelli eventualmente prodotti per le iterazioni precedenti.

- Modello implementativo. Questo modello è frutto della fase di implementazione il cui obiettivo è tradurre in codice i manufatti prodotti nella fase di disegno e quindi realizzare il sistema in termini di componenti: codice sorgente, file XML, file di configurazione, script, ecc.

I diagrammi dei casi d'uso mostrano un insieme di entità esterne al sistema, dette attori, associati con le funzionalità che il sistema stesso dovrà realizzare. Le funzionalità vengono espresse per mezzo di una sequenza di messaggi scambiati tra gli attori e il sistema. L'obiettivo dei casi d'uso è definire un comportamento coerente senza rivelare la struttura interna del sistema.

I diagrammi delle classi, a disegno completato, forniscono una rappresentazione grafica dell'implementazione del sistema eseguibile e sono costituiti da un insieme di classi, interfacce, collaborazioni e relazioni tra tali elementi.

I diagrammi degli oggetti rappresentano una variante dei diagrammi delle classi, tanto che anche la notazione utilizzata è pressoché equivalente, con le sole differenze che i nomi degli oggetti vengono sottolineati e le relazioni vengono dettagliate. Gli Object Diagram mostrano esplicitamente un numero di oggetti istanze delle classi e i relativi legami.

I diagrammi di sequenza e collaborazione — detti anche di interazione in quanto mostrano le interazioni tra gli oggetti che costituiscono il sistema — sono utilizzati per modellare il comportamento dinamico del sistema. I due diagrammi risultano molto simili: si può passare agevolmente dall'una all'altra rappresentazione (isomorfismo). Si differenziano per via dell'aspetto dell'interazione a cui conferiscono maggior rilievo: i diagrammi di sequenza focalizzano l'attenzione sull'ordine temporale dello scambio di messaggi, i diagrammi di collaborazione mettono in risalto l'organizzazione degli oggetti che si scambiano i messaggi.

I diagrammi di stato essenzialmente descrivono automi a stati finiti, e pertanto sono costituiti da un insieme di stati, transizioni tra di essi, eventi e attività. Ogni stato rappresenta un periodo di tempo ben delimitato della vita di un oggetto durante il quale esso soddisfa precise condizioni.

I diagrammi delle attività vengono utilizzati per illustrare il flusso di attività coinvolte in un particolare processo. Sono particolarmente utili per mostrare esecuzioni del comportamento del sistema di alto livello che non coinvolgano dettagli interni come lo scambio di messaggi catturati da altri diagrammi.

I Components Diagram mostrano la struttura fisica del codice in termini di componenti e di reciproche dipendenze. Insieme ai Deployment Diagram costituiscono la vista fisica del sistema.

I diagrammi di dispiegamento (Deployment) mostrano l'architettura hardware e software del sistema: vengono illustrati gli elementi fisici (computer, reti, dispositivi fisici…) opportunamente interconnessi e i moduli software presenti su ciascuno di essi. In sintesi viene mostrato il dispiegamento del sistema a tempo di esecuzione sulla relativa architettura fisica, in termini di componenti e relative allocazioni nelle istanze dei nodi.

Sebbene gli elementi del nucleo dello UML permettano di formalizzare molti aspetti di un sistema, è impossibile pensare che da soli siano sufficienti per illustrarne tutti i dettagli. Pertanto, al fine di mantenere il linguaggio semplice, flessibile e potente allo stesso tempo, lo UML è stato corredato sia di meccanismi

generali utilizzabili per aggiungere informazioni supplementari difficilmente standardizzabili (per esempio le note), sia di veri e propri meccanismi di estensione (come per esempio gli stereotipi).

I meccanismi generali forniti dallo UML sono:

- Notes (note): informazioni supplementari, scritte con un formalismo che può variare dal linguaggio di programmazione, allo pseudocodice al testo in linguaggio naturale e pertanto difficilmente standardizzabili;

- Adornments (ornamenti): elementi grafici che permettono di aggiungere semantica al linguaggio al fine di fornire al lettore una cognizione diretta di aspetti particolarmente importanti di specifici elementi (come per esempio le cardinalità);

- Specifications (specificazioni): elementi di testo che aggiungono sintassi e semantica agli elementi dello UML,per esempio l'elenco degli attributi e metodi presenti appartenenti alle classi.

- Common Divisions (divisioni comuni): nella modellazione di sistemi Object Oriented, ogni elemento reale può essere diviso almeno in due modi diversi: l'astrazione e il relativo corpo; si pensi al rapporto che esiste tra classi e oggetti;

I meccanismi di estensioni forniti dallo UML sono:

- Stereotypes (stereotipi): permettono di estendere il vocabolario dello UML aggiungendo nuovi elementi, specifici per il problema oggetto di studio, ottenuti estendendo opportunamente quelli esistenti;

- Tagged Value (valori etichettati): estendono le proprietà degli elementi dello UML, aggiungendo nuove informazioni costituite dalla coppia nome–valore.

- Constraints (vincoli).

Durante il processo di disegno di sistemi che utilizzano particolari tecnologie piuttosto ricorrenti nei progetti, è necessario avvalersi di una serie di estensioni dello UML. Ciò al fine di renderlo in grado di rappresentare adeguatamente le caratteristiche peculiari di suddette tecnologie. Lo OMG ha deciso pertanto di standardizzare questi insiemi di regole, detti profili, come veri e propri plug-in dello UML. Particolarmente utili sono i profili CORBA ed EJB.