Capitolo 11

Anche gli aspetti "fisici" sono importanti

L'insieme indistinto di questi semi viene organizzato dall'intelletto (nous). L'intelletto è l'anima che muove ogni cosa.
ANASSAGORA

Buone recinzioni fanno buoni vicini.
ROBERT FROST

Introduzione

Questo capitolo è dedicato alla presentazione della vista fisica del sistema, la cui versione finale costituisce la concretizzazione di quanto prodotto attraverso il processo di sviluppo del software nelle fasi a maggiore carattere concettuale. Per comprendere appieno il dominio di questa vista, si consideri l'analogia tra l'edificazione di costruzioni civili e la produzione di sistemi software formulata da Booch [BIB01]: "il prodotto finale di un'azienda di costruzioni è l'innalzamento di un edificio fisico che quindi esiste nel mondo reale. Si producono modelli logici per visualizzare, specificare e documentare le decisioni circa la costruzione, il posizionamento di pareti, porte e finestre; il percorso dei cavi elettrici, il sistema idraulico e il complessivo stile architettonico. Quando poi si costruisce fisicamente l'edificio, queste pareti, porte, finestre e altri oggetti concettuali vengono trasformati in entità reali, in oggetti fisici […]. Lo stesso accade quando si costruiscono sistemi software. Si realizzano modelli logici per visualizzare, specificare, documentare decisioni circa il vocabolario del dominio oggetto di studio, si modellano le proprietà strutturali e comportamentali mediante cui queste entità collaborano, ecc. Si dà poi luogo alla modellazione fisica per costruire il sistema eseguibile. Mentre le entità logiche vivono nel mondo concettuale, le entità fisiche vivono nel mondo "reale", si tratta di ciò che alla fine

risiede nei nodi fisici e che può essere eseguito direttamente o che può, in qualche maniera indiretta, partecipare all'esecuzione del sistema".

Pertanto la vista fisica modella l'implementazione della struttura del sistema eseguibile secondo due ben precise proiezioni: l'organizzazione del sistema in componenti e il relativo dispiegamento (*deployment*) nei nodi che costituiscono l'infrastruttura a runtime. Nella maggior parte dei casi, si esegue un mapping tra classi, i relativi componenti implementativi e i nodi destinati ad ospitarli. Chiaramente non sempre è necessario specificare le classi che albergano in un componente.

La *implementation view* è costituita dai diagrammi dei componenti e quelli di dispiegamento, indicati genericamente con i termini di "diagrammi di implementazione". Sebbene nella maggior parte dei casi li si utilizzi in maniera distinta, è possibile realizzare veri e propri diagrammi implementativi unendo le due notazioni: diagrammi dei componenti calati direttamente nella struttura fisica del sistema.

I diagrammi di implementazione possono, eventualmente, essere utilizzati in modo più ampio nella modellazione del dominio business. In questo caso i componenti si prestano a rappresentare procedure business e manufatti (artifact), mentre i nodi rappresentano le unità dell'organizzazione e le risorse (umane e non) del business.

Il concetto di componente (come evidenziato nel paragrafo successivo) è notevolmente variato negli ultimi tempi, soprattutto grazie all'introduzione (abbastanza recente) dei sistemi component-based. Questa nuova tecnologia ha generato la necessità di espandere (legittimamente) il concetto di componente, e dei relativi diagrammi, verso spazi a carattere più concettuale. Quindi, non più un componente come generico contenitore di qualsiasi tipo di file necessario all'esecuzione del sistema, bensì come entità a sé stante, ben definita, che bisogna progettare fin dalle prime fasi del processo di sviluppo del software.

Prima di procedere con l'illustrazione della notazione di questi diagrammi, si è ritenuto doveroso presentare brevemente quella che è stata l'evoluzione del concetto di componente. Si tratta di un'area dello UML che ha generato notevoli critiche, a volte feroci, da parte della comunità informatica, attenuate in parte con la presentazione della versione 1.4. Il problema era insito nella definizione, non sempre chiara e assolutamente restrittiva, di componente: ne confinava l'utilizzo alla vista fisica, veniva confuso con altri concetti quali quello di manufatto (*artifact*), classe, ecc.

Si trattava di una manchevolezza tangibile che aveva il suo peso, specie considerando che lo UML è per eccellenza il linguaggio di modellazione di sistemi OO e component-based. A dire il vero, questa lacuna era il risultato ultimo della mancanza di una visione comune e di una certa confusione regnante nella stessa comunità OO (o component-based che dir si voglia). Ciò è giustificato considerando che solo recentemente la tecnologia component-based sembrerebbe aver intrapreso la direzione verso lo stato di maturità.

> In merito alla questione della definizione formale di componente si è assistito, specie negli ultimi anni, a un vivace dibattito, portato avanti a suon di articoli da personaggi di rilievo del

settore. In particolare, hanno preso parte direttamente al dibattito (forse sarebbe più veritiero definirla "contesa") persone del calibro di Betrand Meyer (autore di svariati testi tra cui [BIB20]), Clemens Szyperski (anch'egli autore di diversi libri, tra cui [BIB30], [BIB31]) e, indirettamente attraverso i loro libri, esperti del calibro John Daniels e John Chessman (autori del libro [BIB15]), Scott Ambler, gli autori del testo [BIB18], e altri ancora. Questo dibattito ha portato a una più intima comprensione del componente come entità a sé stante, sebbene, per avere una definizione finale, sintetica, chiara e precisa sia necessario attendere ancora altro tempo.

La RTF competente per la definizione della versione 2 dello UML sembrerebbe più che al corrente di questa problematica e, dai comunicati ufficiali, si direbbe intenzionata a investire molto per risolvere completamente il problema. Già nella versione 1.4 si sono fatti considerevoli passi avanti, soprattutto separando nettamente concetti come classi e manufatti da quello di componente; ora si attende che la versione 2 elimini completamente tutte le ombre.

Considerate le premesse, si è deciso di utilizzare un approccio assolutamente pragmatico, ponendosi nell'ottica di coloro che devono realizzare i sistemi software utilizzando i concetti di componente, nonostante le varie limitazioni esistenti. Quindi si è cercato di non dilungarsi troppo sulla problematica, cercando invece di illustrare chiaramente cosa sia un componente e come utilizzarlo ai fini della modellazione di sistemi reali, mentre si demandano ad altre sedi digressioni sulla definizione di componente (è sufficiente navigare in Internet per trovare molta documentazione).

Da tener presente che anche la versione iniziale e restrittiva del concetto di componente non ha impedito di modellare sistemi component-based a tutti gli architetti desiderosi di utilizzare un approccio preciso alla progettazione. A tal fine basti prendere in considerazione l'appendice relativa al profilo EJB in cui i componenti sono rappresentati per mezzo dell'elemento sottosistema.

Illustrati i formalismi dei diagrammi dei componenti e di dispiegamento, nonché le immancabili sezioni dedicate ai consigli relativi allo stile, il capitolo si conclude con una brevissima presentazione di un metodo (derivante essenzialmente da quanto esposto nel libro [BIB15]) dimostratosi particolarmente semplice ed efficace nel difficile compito di decomporre il sistema in componenti.

Evoluzione del concetto di componente.

Questo paragrafo si è reso necessario al fine di illustrare come l'evoluzione del concetto di componente abbia cambiato il modo di concepire i sistemi e, conseguentemente, l'utilizzo di alcuni strumenti dello UML, come i diagrammi dei componenti. Si tratta pertanto di un paragrafo di carattere "storico", non sempre brillante, e non indispensabile per la comprensione di quanto riportato in seguito.

Molti tecnici sono ancora legati alla visione iniziale (quasi primordiale) che sanciva l'utilizzo della notazione dei diagrammi dei componenti esclusivamente nel contesto della vista "fisica", magari prima del rilascio delle versioni iniziali del sistema o per avere un'idea più precisa del deployment. La visione moderna, scaturita da necessità reali emerse nella di progettazione di sistemi basati sui componenti — ancora una volta il mercato, la pratica che determina la teoria — conferisce al componente anche aspetti concettuali, rendendone possibile l'utilizzo anche nelle fasi concettuali della progettazione.

> Questa breve digressione di carattere "storico" aiuta a meglio comprendere le radici del problema. Sembrerebbe che le attuali lacune (tanto per cambiare), siano conseguenza di un pessimo retaggio storico, di un concetto introdotto per esigenze non correlate a specifiche attività di modellazione. Agli inizi, i sistemi software venivano organizzati intorno a un unico blocco monolitico, eventualmente supportato da qualche servizio basilare. A fine anni Settanta ci fu una prima significativa evoluzione: i sistemi cominciavano a essere strutturati in termini di *macromoduli* relazionati in qualche modo, mentre negli anni 80 si assistette alla graduale affermazione dell'OO. Dal canto suo l'architettura cominciò ad essere organizzata in strati realizzati attraverso una miriade di classi interconnesse. Non troppo tempo fa si è assistito a quella che, secondo molti esperti del settore, è la naturale evoluzione del paradigma OO: sistemi component-based, in cui l'architettura è organizzata in termini di componenti, eventualmente appartenenti a diversi strati, che interagiscono tra di loro. Pertanto, al contrario di alcune aziende che gradirebbero il contrario, è legittimo che la tecnologia OO fornisca le naturali fondamenta per la costruzione di sistemi component based.

Secondo molti, la notazione UML continua — a dire il vero sempre di meno — a risentire di un'errata eredità che ha impedito una definizione ex novo, moderna e organica di "componente": un pessimo retaggio storico che vedeva il componente come un elemento generico, una sorta di contenitore introdotto artificialmente per raggruppare elementi necessari per l'esecuzione del sistema (script, file eseguibili, ecc.), in cui l'enfasi era tutta orientata sulla manifestazione "fisica" dello stesso.

Il problema è probabilmente legato al fatto che il concetto di componente e dei relativi diagrammi è stato inizialmente ideato non tanto per soddisfare problematiche concrete evidenziate dalla progettazione di sistemi, ma per mera convenienza di rappresentazione. A questo punto le necessità di mantenere compatibilità con il mondo del passato, di salvaguardare la comodità di disporre di un contenitore generico con cui indicare diversi elementi (script, file di configurazione, documenti testo da visualizzare, ecc.) che prendono parte a un sistema in esecuzione, hanno minato alla base la notazione dei diagrammi dei componenti, relegando il componente ai soli aspetti fisici del sistema.

Senza aver la pretesa di essere esaustivi, come riprova di quanto affermato poc'anzi si considerino le definizioni di componente presenti nei testi dei Tres Amigos. Si tratta di

definizioni che, chiaramente, non vanno lette con occhio troppo critico, giacché sono state proposte in tempi in cui appena si cominciava a parlare del paradigma component-based secondo l'accezione moderna.

"Un componente è una parte fisica e sostituibile di un sistema che si conforma a un insieme di interfacce delle quali fornisce la realizzazione." [BIB01].

"Un componente è una parte fisica e sostituibile che impacchetta implementazione e si conforma a un insieme di interfacce delle quali fornisce l'implementazione". "Un componente rappresenta un pezzo fisico di implementazione di un sistema, includendo codici software (file sorgenti, binari e eseguibili) o equivalenti, come script o file di comandi. Alcuni componenti hanno un'identità e possono possedere entità fisiche, che includono oggetti a tempo di esecuzione, documenti, database e così via. I componenti esistono nel dominio dell'implementazione […]" [BIB03].

Una primissima constatazione da farsi è relativa alla fantasia necessaria per immaginare un componente, una particella software, come parte fisica. Queste definizioni, risalenti alla fine degli anni Novanta, a detta degli esperti del settore, rispecchiano una visione antiquata e assolutamente restrittiva del componente, lontana da quella utilizzata per costruire sistemi che sfruttano le architetture EJB, .NET e CORBA. Chiaramente non si tratta unicamente di un marginale problema di definizione, di pedanteria, bensì di una seria riduzione della capacità di esprimere importanti concetti, una limitazione all'utilizzo dei componenti e dei relativi diagrammi che ha generato serie ripercussioni sull'utilizzo di queste notazioni nella progettazione dei sistemi. Non si tratta assolutamente di un problema marginale, considerando che il nuovo paradigma di progettazione e sviluppo dei sistemi è appunto quello basato sui componenti.

Le principali critiche a questa visione iniziale dei componenti vertevano essenzialmente sulla confusione e sovrapposizione esistente tra il concetto di componente e altri concetti quali classi e manufatti (*artifact*), la limitazione d'uso del concetto di componente relegato nella sola vista fisica, la confusione intorno alla possibilità da parte di alcuni componenti a possedere altre entità o implementare interfacce (si pensi a documenti o a script), ecc.

> Senza dilungarsi, alcune critiche mosse nei confronti dell'iniziale definizione dello UML sono:
> "I componenti UML generalmente non sono utili per la modellazione di queste cose che l'industria sta chiamando componenti (come gli EJB). Gran parte di questa modellazione è ottenuta utilizzando tagged values e stereotipi. I componenti UML confondono caratteristiche di tipo e istanza; per esempio un componente UML può rappresentare contemporaneamente un file e la relativa immagine in memoria" (Steve Cook – Steve Brodsky, IBM, 1999);
> "La semantica corrente per il costrutto UML di componente è vaga e, come sottolineato in precedenza, presenta diverse sovrapposizioni con la semantica di altri classificatori, quali per esempio classe e sottosistema. Al fine di supportare completamente lo sviluppo di sistemi component-based, la semantica dei componenti dovrebbe essere raffinata e, contestualmente,

> bisognerebbe ridurre la sovrapposizione con gli altri costrutti" (Cris Kobryn, UML RTF OMG — tra l'altro autore del testo [BIB26]);
> "Noi abbiamo la sensazione che l'enfasi conferita dall'UML alla manifestazione fisica del componente sia sproporzionata. La tecnologia dei componenti è la base per la formazione di composizioni e i componenti sono parte di una composizione. Questa integrazione componente/composizione non è espressa in UML. Concentrandosi esclusivamente sugli aspetti fisici della composizione si limita la capacità dello UML di esprimere importanti concetti" (Cory Casanave, Data Access Technologies, 1999).

La definizione incorporata nel documento delle specifiche ufficiali dello UML versione 1.4 ([BIB07]) — riportata nel successivo paragrafo — risolve moltissime lacune, anche se ne lascia irrisolte altre, probabilmente per problemi di compatibilità con il passato. In ogni modo, dalla lettura della definizione ufficiale è possibile evidenziare alcune considerazioni:

- viene rimosso il riferimento alla "fisicità" del componente;
- si chiarisce che su uno stesso componente possono risiedere diversi classificatori, alcuni rappresentanti le interfacce;
- che un componente può essere implementato da uno o più manufatti, ecc.

Il problema della sovrapposizione delle definizioni di alcuni elementi è stato affrontato da diverse parti: sia dal punto di vista della definizione del componente, sia da quello dei manufatti (*cfr* paragrafo *Qualche parola sull'elemento Artifact*), sebbene lasci non poche perplessità l'affermazione che un componente possa essere implementato da più manufatti (fig. 11.1). Probabilmente sarebbe più opportuno pensare ad un manufatto implementato attraverso più componenti. A tal fine è sufficiente considerare una qualsiasi libreria commerciale java/EJB. Queste, tipicamente, sono distribuite attraverso un JarFile (nella versione 1.4 del metamodello UML si tratta di una specializzazione della metaclasse `Artifact`) il quale, nella maggior parte dei casi, è implementato da diversi componenti. Ancora una volta un problema di compatibilità con il passato?

Elementi condivisi

In questa sezione sono presentati gli elementi base cui si fa continuo riferimento nel corso della trattazione del presente capitolo. In particolare si focalizza l'attenzione sui due elementi principe della vista fisica: il componente e il nodo. Si è ritenuto importante chiarirne da subito la definizione nel contesto dello UML, al fine di agevolare la trattazione successiva, evitando continue chiarificazioni e rimandi.

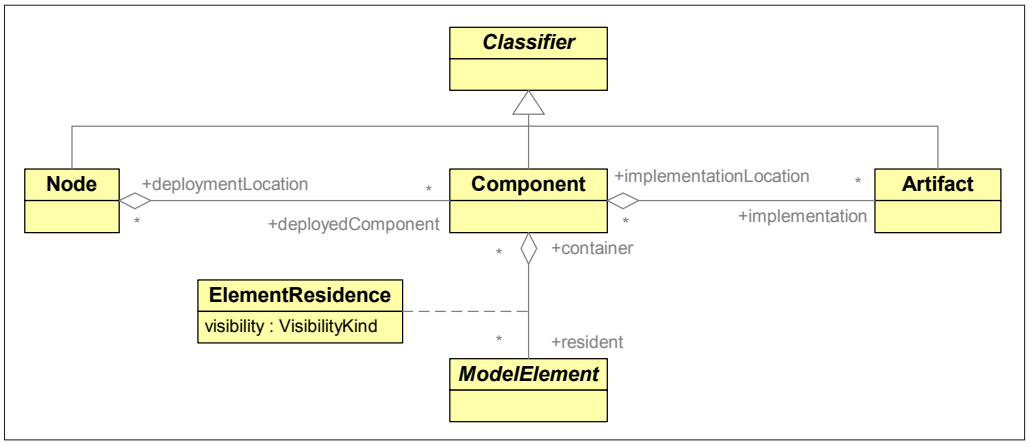
Ancora sui componenti

In questo paragrafo viene illustrato il concetto di componente, corredato dalla relativa notazione, funzionalmente alle necessità di esposizione della notazione UML. In altre parole viene esposto, in modo asettico, quanto riportato nelle specifiche ufficiali dello UML, tralasciando il più possibile le speculazioni riportate nel paragrafo precedente.

La versione 1.4 dello UML sancisce che "un componente rappresenta una parte del sistema, modulare e sostituibile, che incapsula implementazione ed espone un insieme di interfacce. Un componente è tipicamente specificato da uno o più classificatori che risiedono nel componente stesso. Un sottoinsieme di questi classificatori definisce esplicitamente le interfacce esterne al componente. Un componente si conforma all'interfaccia che espone, dove le interfacce rappresentano servizi forniti da elementi che risiedono nel componente. Il componente può essere implementato da uno o più manufatti (*artifact*), come file binari, eseguibili, script, ecc. Un componente può essere allocato (*deployed*) su un nodo."

Dalla definizione emerge chiaramente come nella versione 1.4 dello UML si sia finalmente riconosciuta la presenza e l'importanza concettuale dell'elemento componente. In particolare, è dichiarata esplicitamente la possibilità di inserire componenti sia nei modelli di disegno (come quello relativo alla struttura statica), sia in quelli di carattere

Figura 11.1 — *Frammento del metamodello dello UML relativo agli elementi* `Node`*,* `Component` *e* `Artifact`*, specializzazioni della metaclasse astratta* `Classifier`*. Il diagramma stabilisce che su uno stesso nodo possono essere collocati diversi componenti, così come uno stesso componente può essere collocato in diversi nodi. Un componente può essere la locazione di diversi manufatti e ciascun manufatto può implementare diversi componenti. Infine, diversi elementi possono risiedere in uno stesso componente, con una specifica visibilità indicata dalla classe associazione* `ElementResidence`*.*



implementativo. Nella rappresentazione dei diagrammi dei componenti non è necessario specificare il nodo in cui sono allocati e tanto meno il manufatto che li implementa.

Un componente è mostrato graficamente attraverso un rettangolo che ne ha altri due di dimensioni ridotte sporgenti nel lato inferiore di sinistra (fig. 11.4).

I componenti a livello di tipo sono identificati esclusivamente dal nome del tipo (`component-type`), mentre a livello di istanza posseggono un nome e un tipo (`component-name : component-type`) entrambi opzionali. Qualora non si visualizzi il tipo, chiaramente, non si riporta neanche il simbolo due punti (`:`).

Come da standard UML, nelle rappresentazioni al livello di istanza, la stringa con il nome e il tipo è visualizzata sottolineata. Questa può essere collocata all'interno oppure sotto oppure sopra il simbolo del componente.

Oggetti che risiedono in un'istanza di un componente sono mostrati annidati all'interno del simbolo componente stesso (fig. 11.5) così come le classi che implementano un componente sono mostrate annidate all'interno del simbolo del componente stesso. Chiaramente l'annidamento indica una relazione di residenza e non di possesso (fig. 11.5).

Per gli elementi che risiedono in un componente si pone il problema della visibilità. Come riportato nel diagramma di fig. 11.1, questa è specificata per mezzo dell'association-class `ElementResidence`, la quale stabilisce la visibilità degli elementi ospitati dal componente. Per esempio, la visibilità di un'interfaccia esterna deve essere `public`. La notazione è la stessa utilizzata dall'elemento `package` (si premette il simbolo della visibilità al nome dell'elemento). Il significato, chiaramente, varia in funzione della natura del componente stesso. Per esempio, in un componente di codice di un linguaggio di programmazione, l'attributo di visibilità controlla l'accessibilità dei costrutti del sorgente. Per un componente relativo a codice eseguibile, la visibilità controlla la possibilità del codice di altri componenti di invocare o accedere al codice del componente stesso.

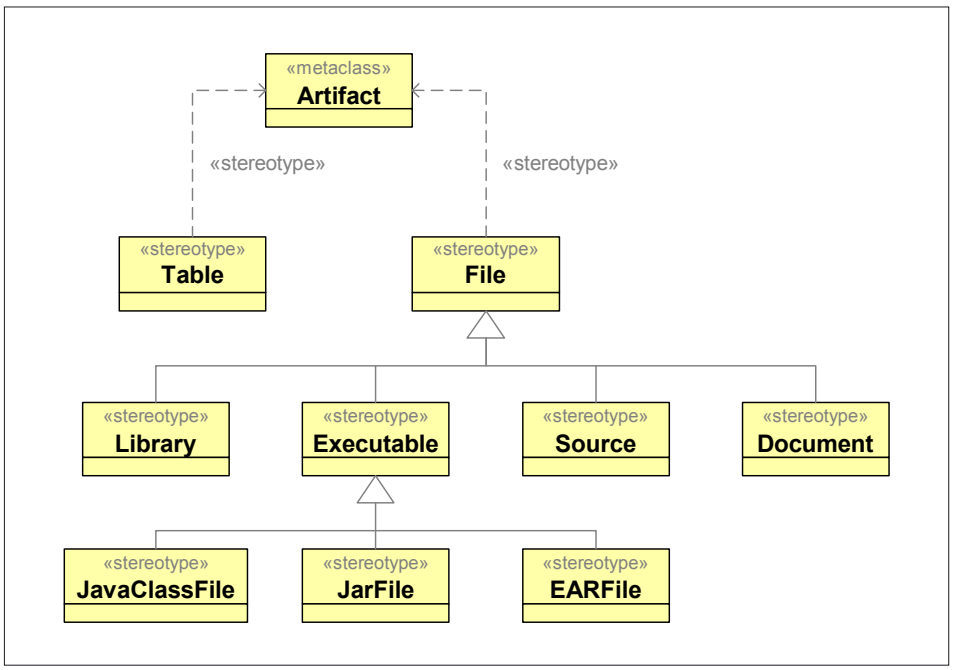
Qualche parola sull'elemento Artifact

> Questo paragrafo è stato inserito nel presente capitolo per una serie di fattori: per il forte legame esistente tra gli elementi `Artifact` e `Component` (come visto, la definizione standard sancisce che `Artifact` implementi uno o più componenti) e perché la definizione di `Artifact`, introdotta nella versione 1.4 dello UML, ha consentito di fare chiarezza nella definizione stessa di componente.

Un manufatto (`Artifact`) rappresenta un pezzo "fisico" di informazione utilizzato o prodotto da un processo di sviluppo del software. Alcuni esempi di manufatti sono: moduli, file sorgenti, script e file eseguibili.

Un manufatto può costituire l'implementazione di un componente runtime. Questa affermazione, tratta dalle specifiche ufficiali della versione 1.4, sembra piuttosto bizzarra

Figura 11.2 — *Classificazione degli stereotipi UML dell'elemento* `Artifact`. *Lo standard giunge fino alla definizione delle sottoclassi dell'elemento file, mentre, nel documento delle specifiche ufficiali, gli elementi inferiori sono proposti come esempi di stereotipi implementativi e specifici della piattaforma.*



e, probabilmente, si deve a problemi di compatibilità delle versioni precedenti dello UML. Probabilmente è più logico attendersi l'opposto: uno o più componenti che implementano un manufatto.

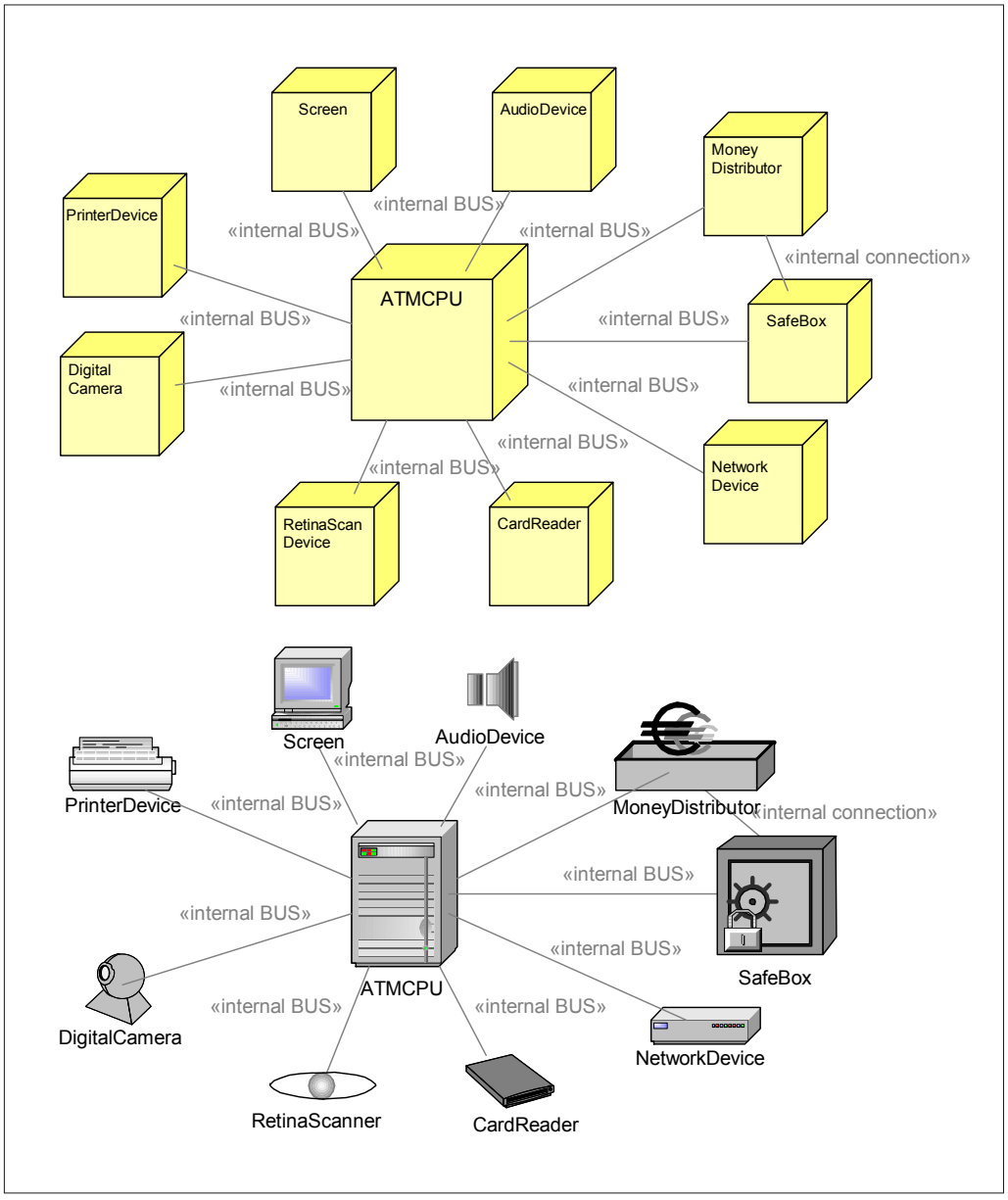
Come riportato nella fig. 11.1, l'elemento `Artifact` specializza la metaclasse `Classifier`. Ciò gli consente di possedere sia proprietà strutturali, sia comportamentali. Per esempio potrebbe disporre dell'attributo che ne indica la sola lettura, così come di operazioni di check in, check out, ecc.

Gli stereotipi standard dell'elemento `Artifact` sono riportati in fig. 11.2.

L'elemento nodo (Node)

Un nodo descrive un oggetto fisico (nel senso stretto del termine) rappresentante una risorsa computazionale. Questa è caratterizzata dal possedere almeno capacità di

Figura 11.3 — *Rappresentazione della struttura interna di un bancomat (ATM) abbastanza futuristico. Del deployment diagram sono mostrate due versioni: una utilizza la rappresentazione standard dell'elemento nodo; nell'altra appositi stereotipi hanno sostituito gli spigolosi cubi.*



memorizzazione e, molto frequentemente, anche capacità elaborative. I nodi sono gli elementi fisici nei quali vengono dispiegati i componenti.

In alcuni contesti, come per esempio l'analisi dell'area business, i nodi possono essere utilizzati per rappresentare elementi diversi da risorse computazionali. Per esempio possono descrivere risorse umane, attuatori meccanici, ecc. A dire il vero si tratterebbe di utilizzi abbastanza desueti.

I nodi prevedono sia una rappresentazione al livello di tipo, sia al livello di istanza. A differenza dei componenti, è possibile realizzare diagrammi dei nodi a livello di istanza senza coinvolgere ulteriori elementi. Qualora si dia luogo a rappresentazioni di questo tipo, è possibile visualizzare gli elementi residenti nel nodo (oggetti e istanze di componenti) che, consistentemente con il diagramma e con la rappresentazione del nodo ospitante, devono essere anch'essi al livello di istanza.

Nel metamodello UML (fig. 11.1) la metaclasse `Node` è specializzazione di `Classifier` ed è relazionata, tramite composizione, con un insieme di `Component` che descrivono i componenti residenti sul nodo. La notazione grafica standard del nodo è abbastanza banale ed è data da un parallelepipedo (fig. 11.3).

Nella realizzazione dei diagrammi di dispiegamento, soprattutto nei casi in cui non sia necessario mostrarvi anche il dettaglio dell'allocazione dei componenti, è fortemente consigliato utilizzare opportuni stereotipi, per rendere la rappresentazione più chiara e accattivante (fig. 11.3).

I nodi al livello di tipo, consistentemente con quanto visto per i componenti, prevedono un nome tipo (`node-type`), mentre al livello di istanza è possibile specificare anche il nome proprio (`name : node-type`).

Il nome, opzionale, rappresenta il nome dello specifico nodo, mentre il tipo specifica la tipologia della risorsa computazionale. Coerentemente con quanto visto in precedenza, entrambi gli elementi sono opzionali e, chiaramente, qualora non sia riportato il tipo, anche i due punti (`:`) vanno omessi.

La visualizzazione degli elementi (componenti) residenti su di un nodo può essere ottenuta in due modi diversi:

- utilizzando una freccia tratteggiata con la parola chiave `deploy` evidenziata;
- includendo direttamente gli elementi all'interno del simbolo che rappresenta il nodo.

Nel primo caso è illustrata la capacità di un nodo di supportare il dispiegamento di componenti al livello di tipo.

> La relazione di `<<deploy>>` rappresenta la metaassociazione del metamodello UML tra l'elemento `Node` e quello `Component` (fig. 11.1).

I nodi possono essere connessi tra loro attraverso una relazione di associazione. Questa evidenzia un percorso di comunicazione tra i nodi associati. Queste associazioni si prestano a essere rappresentate per mezzo di opportuni stereotipi.

I diagrammi dei componenti

Definizione

I diagrammi dei componenti mostrano la struttura del sistema in termini della relativa organizzazione in componenti. In altre parole, mostrano un grafo di componenti connessi attraverso relazioni di dipendenza.

La descrizione dei singoli componenti (come evidenziato nel diagramma di fig. 11.1), include i classificatori che vi risiedono, che specificano il componente stesso (per esempio interfacce e classi di implementazione) e i manufatti che li implementano (come file di codice, file binari, file eseguibili, script ecc.).

I classificatori residenti nei componenti (l'elemento `Classifier` è un discendente di `ModelElement`) possono essere connessi a questi ultimi sia attraverso il contenimento fisico (l'elemento viene visualizzato all'interno del componente di appartenenza), sia utilizzando la relazione `<<reside>>` (fig. 11.4). Si tratta di una relazione istanza della metaassociazione che nel metamodello UML associa l'elemento `Component` a quello `ModelElement`. Un discorso analogo vale per i manufatti che specificano i componenti. Questi possono essere connessi ai componenti o attraverso il contenimento fisico, o per mezzo della relazione `<<implement>>` (fig. 11.4), istanza della relazione che lega gli elementi del metamodello `Component` e `Artifact`.

Sebbene le relazioni `<<reside>>` e `<<implement>>` utilizzino la stessa notazione grafica della relazione di dipendenza, si tratta di relazioni a sé stanti aventi la propria metaclasse nel metamodello UML.

> Nella fig. 11.4 sono stati introdotti gli stereotipi dell'elemento classe `<<auxiliary>>` e `<<focus>>`. Si tratta di elementi utilizzati tipicamente in coppia, introdotti con la versione UML 1.4, specificamente per agevolare la modellazione del sistema in componenti durante la fase di disegno. In particolare, lo stereotipo `<<focus>>` specifica una classe che definisce il nucleo della logica o il controllo del flusso di una o più classi ausiliarie. Nell'ambito di uno stesso diagramma, avendo provveduto a dichiarare lo stereotipo `<<focus>>` per la classe "principale", l'indicazione esplicita di quelli di tipo `<<auxiliary>>` può essere omessa, in quanto la funzione di supporto delle classi ausiliarie è evidenziata per mezzo di una relazione di dipendenza che le lega, nel ruolo di classi indipendenti, alla classe principale evidenziata dallo stereotipo `<<focus>>`. Per esempio, nel diagramma di fig. 11.4 si sarebbe potuto evitare di evidenziare lo stereotipo `<<auxiliary>>` per le classi `CatalogInfo` e `CatalogPK`. Il loro ruolo di classi ausiliarie è implicito dalla relazione di dipendenza che le lega alla classe `Catalog`,

evidenziata dallo stereotipo `<<focus>>`. Lo stesso discorso vale per l'indicazione esplicita dello stereotipo `<<focus>>`, che può essere omesso qualora la relativa classe sia legata, attraverso la relazione di dipendenza, a classi evidenziate con lo stereotipo `<<auxiliary>>`. A questo punto dovrebbe essere chiaro anche il significato di quest'ultimo stereotipo utilizzato per evidenziare classi che ne supportano, implementando logiche secondarie o secondari flussi di controllo, un'altra più centrale o fondamentale, evidenziata attraverso lo stereotipo `<<focus>>`.

Dallo studio delle specifiche ufficiali dello UML si evidenzia che l'elemento `Component` non ha proprietà strutturali e comportamentali (come per esempio attributi e operazioni). Ciò in prima analisi potrebbe sembrare piuttosto strano (la metaclasse `Component` eredita da `Classifier`, che è l'elemento del metamodello che introduce le caratteristiche comportamentali e strutturali); però nell'insieme dei vincoli (*Well-Formedness Rules*) è presente la seguente restrizione: `Component.feature->isEmpty`. Comunque, l'elemento componente si comporta come un contenitore di altri classificatori, eventualmente

Figura 11.4 — *Descrizione della struttura di un componente. Il diagramma, ripreso dalle specifiche ufficiali dello UML (fig. 3-96), è stato selezionato per esporre l'utilizzo delle relazioni* `<<reside>>` *e* `<<implement>>` *. In particolare, viene mostrato un entity EJB che rappresenta il concetto di catalogo.*

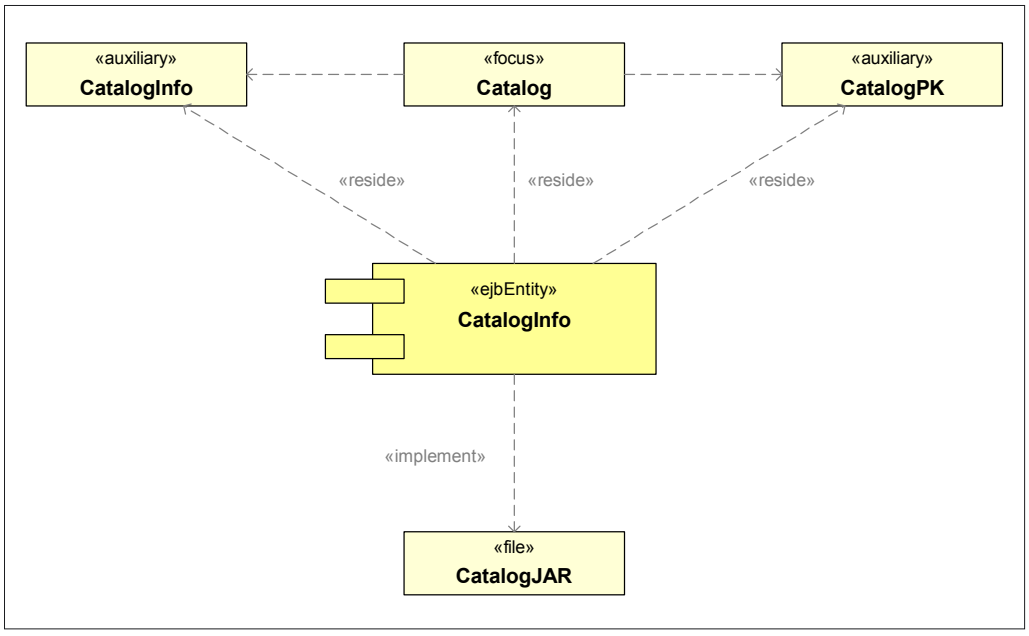
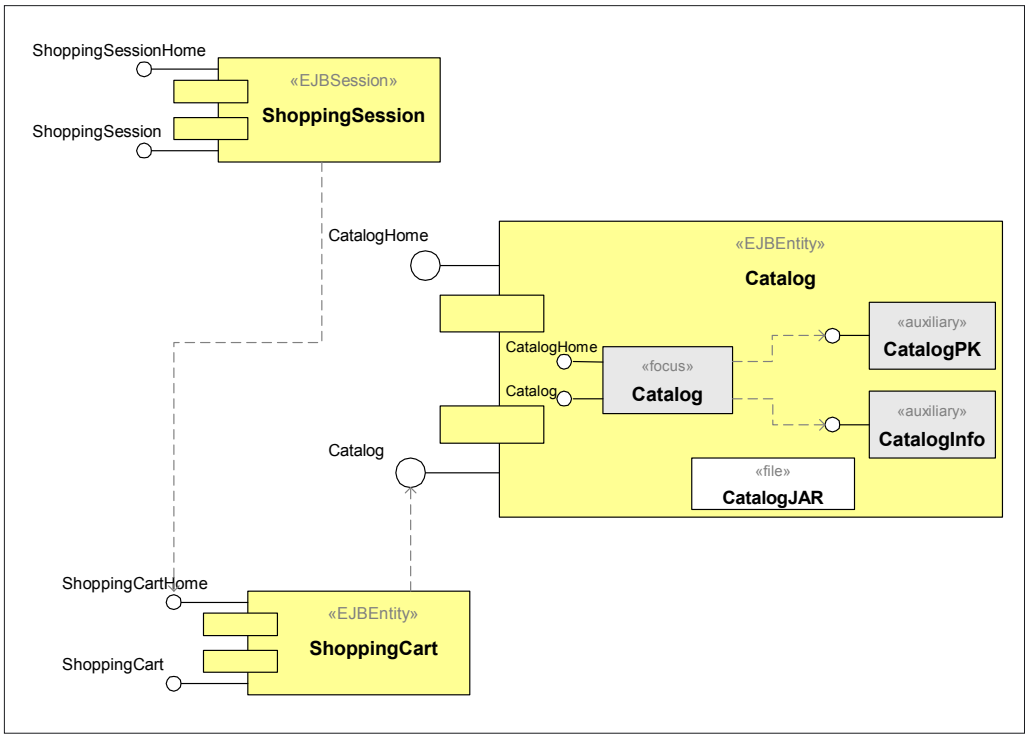


Figura 11.5 — *Esempio, tratto dalle specifiche ufficiali dello UML (fig. 3-95), di un diagramma dei componenti relativo a una porzione di un sistema per il commercio elettronico. In particolare, sono mostrati i componenti:* `ShoppingSession`, `ShoppingCart` *e* `Catalog` *e le relative relazioni di dipendenza. Inoltre, il componente* `Catalog` *è rappresentato con i* `Classifier` *che vi risiedono, e il manufatto che lo implementa.*



definiti con le proprie caratteristiche strutturali e comportamentali. In particolare, i componenti espongono un insieme di interfacce, le quali rappresentano i servizi forniti dagli altri elementi che risiedono nel componente. Per esempio, nel diagramma di fig. 11.5, il componente `Catalog` espone le interfacce implementate dalla classe `Catalog`, che vi risiede. Vista l'importanza di queste interfacce, i relativi simboli sono mostrati esplicitamente.

L'utilizzo dei servizi di un componente da parte di un altro è mostrato attraverso la relazione di dipendenza che parte da un componente e giunge all'interfaccia di un altro.

Sempre nel diagramma di fig. 11.5, è illustrata una porzione dell'organizzazione in componenti di un sistema per il commercio elettronico. In particolare sono illustrati i legami di dipendenze di tre componenti: `ShoppingSession`, `ShoppingCart` e

`Catalog`. Verosimilmente, il primo componente offre una serie di servizi relativi alla gestione del carrello della spesa utilizzato nel sito. Questi servizi includono funzioni di reperimento dei dati di uno specifico carrello, azzeramento dello stesso, aggiunta o rimozione di un determinato articolo, ecc. Pertanto il dominio del componente `ShoppingSession` è costituito da un gruppo complesso di dati rappresentante il concetto del carrello della spesa. Tale macroentità è incapsulata nel componente `ShoppingCart`. Questo a sua volta deve utilizzare dei dati relativi agli articoli trattati, incapsulati nel componente `Catalog`, del quale viene evidenziata la struttura interna. In particolare, il componente espone le interfacce della classe `Catalog`. Questa è un'istanza dello stereotipo `<<focus>>` e dipende da due classi di supporto, quella relativa alla chiave primaria (`CatalogPK`, dove PK indica Primary Key) e quella demandata alla gestione delle informazioni vere e proprie (`CatalogInfo`), che ovviamente sono istanze dello stereotipo `<<auxiliary>>`.

I diagrammi dei componenti (non i singoli componenti) prevedono esclusivamente una forma di tipo e non una istanza. In altre parole non è contemplata la dicotomia tipo–istanza invece prevista da altri diagrammi, come per esempio il diagramma delle classi, la cui versione a livello di istanza è data dai diagrammi degli oggetti. Volendo mostrare istanze di componenti è possibile ricorrere ai diagrammi di dispiegamento, eventualmente realizzando una versione degenerata in cui non viene visualizzato alcun nodo.

Utilizzo

Da quanto riportato in precedenza, la precisa prescrizione dell'utilizzo dei diagrammi dei componenti nel contesto dei processi di sviluppo di sistemi basati sui componenti (CBD, *Component Based Development*) presenta ancora diverse controversie. Di nuovo, ci si muove su un campo minato.

Inizialmente veniva consigliato di utilizzare i diagrammi dei componenti per modellare l'organizzazione dei file sorgenti e delle parti fisiche che costituiscono l'implementazione; in poche parole rappresentazioni a basso livello della configurazione del software Per questi fini, l'impiego dei diagrammi dei componenti risulta poco interessante e, probabilmente, potrebbero essere sostituiti egregiamente dai diagrammi di dispiegamento (permettono sia di descrivere l'organizzazione dei componenti da dispiegare, con le relative riflessioni, sia i nodi in cui allocarli). Con ciò non si intende dire che queste non siano tematiche presenti e spesso di rilievo (basti pensare all'organizzazione del deploy del sistema), bensì che per queste problematiche, verosimilmente, esiste tutta una serie di strumenti automatici più idonei.

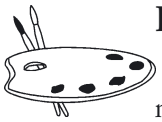
Una cosa certa è che tutti coloro che progettano professionalmente sistemi basati sui componenti sanno perfettamente che il concetto di componente deve essere introdotto già nelle fasi a carattere concettuale della progettazione del sistema, anche prima della definizione dello stesso modello di disegno.

Verosimilmente è opportuno stabilire con precisione come organizzare il sistema in componenti, quanti, quali realizzare, le reciproche dipendenze, prima di poter procedere al disegno del sistema stesso. Una volta definita l'organizzazione del sistema in componenti, considerando i vari strati dell'architettura, la successiva attività di disegno dovrebbe ridursi alla definizione del dettaglio dei singoli componenti. Durante questa attività, spesso, si genera la necessità di rivedere e raffinare il disegno iniziale dei componenti, ma questo è del tutto normale e legittimo se si verifica in fase di disegno: lo sarebbe molto di meno in fase implementativa.

In sintesi, delegare i componenti ai soli aspetti fisici ne limita moltissimo l'utilizzo e l'interesse. Pertanto, come si mostrerà nel paragrafo di conclusione del capitolo, è opportuno dar luogo a una modellazione dell'organizzazione del sistema in componenti non appena le prime versioni stabili del modello dei casi d'uso del dominio e del relativo modello a oggetti presentino una certa stabilità. In altre parole, l'avvio formale dovrebbe avvenire in concomitanza con la baseline della prima versione di tali modelli, sebbene sia opportuno cominciare a sbirciare il modello dei casi d'uso e quello a oggetti del dominio prima possibile. Ciò al fine di cominciare a farsi un'idea e a dar luogo alle versioni embrionali della struttura del sistema in componenti.

Secondo quest'ottica, non solo l'organizzazione statica dei componenti deve essere modellata già nelle prime fasi del processo di sviluppo del software (magari attraverso opportuni diagrammi dei componenti), ma è importante anche realizzare diagrammi relativi al comportamento dinamico, prodotti ricorrendo alla notazione dei diagrammi di interazione.

Nulla vieta, chiaramente, di utilizzare i diagrammi dei componenti secondo la visione classica focalizzata sulla vista fisica, ma questo dovrebbe essere visto esclusivamente come un completamento.



Lo stile

Si presenta una serie di linee guida che permettono di realizzare diagrammi dei componenti di maggiore qualità e quindi più facilmente fruibili.

Utilizzare nomi significativi

Questo consiglio appartiene all'insieme di quelli a carattere generale. In particolare, utilizzare nomi significativi e quanto più possibili rispondenti al dominio oggetto di studio, evitando strane abbreviazioni, permette di rendere i diagrammi più facilmente leggibili dai diversi attori del processo di sviluppo del software: dagli architetti, ai programmatori, ai tester al *deployment team*.

Qualora un diagramma dei componenti modelli un sistema basato su un'architettura multistrato è opportuno stabilire chiaramente e utilizzare consistentemente i suffissi da aggiungere al componente per evidenziarne lo strato di l'appartenenza. Per esempio si

potrebbero utilizzare i suffissi "BS" e "BO" per indicare, rispettivamente, componenti appartenenti allo strato Business Service e a quello Business Object (*cfr* Capitolo 8, paragrafo *Dipendenza del modello di analisi dall'architettura*).

Utilizzare consistentemente gli stereotipi

In questo contesto è ancora fortemente consigliato ricorrere all'utilizzo di opportuni stereotipi, ma questi, tranne rarissimi casi, dovrebbero essere visualizzati semplicemente attraverso opportune stringhe (per esempio `<<EJBEntity>>` e `<<EJBSession>>`) da riportare nel simbolo standard di componente (fig. 11.5).

Poiché la consistenza è una parola chiave nella realizzazione di un qualsivoglia diagramma, anche in questo caso è fortemente consigliato pianificare opportunamente gli stereotipi di cui avvalersi e quindi utilizzarli conseguentemente.

Mostrare chiaramente le interfacce

Come dichiarato nelle specifiche ufficiali "Un componente è tipicamente specificato da uno o più classificatori che risiedono nel componente stesso. Un sottoinsieme di questi classificatori definisce esplicitamente le interfacce esterne al componente".

In questo ambito, si preferisce ricorrere allo stereotipo standard dell'elemento interfaccia (cerchietto); si tratta della versione classica, quindi piu' comprensibile. Inoltre rende il diagramma più compatto (ciò è particolarmente utile perché può capitare di dover rappresentare nello stesso diagramma diversi componenti) e tende a ridurre il numero di segmenti sovrapposti.

Stabilire la posizione delle interfacce

Tipicamente, le interfacce sono posizionate alla sinistra del componente e, quantunque ciò non sia obbligatorio (è possibile attaccarle in qualsiasi parte del relativo classificatore), è consigliabile, per quanto possibile, seguire questa convenzione. Si tratta di una notazione classica e quindi più immediata.

In specifici ambiti, come per esempio nelle architetture J2EE, sia il numero delle varie interfacce, sia il loro significato sono bene definiti. In questi contesti, è consigliabile stabilire una convenzione (per esempio la `home interface` in alto e la `remote interface` in basso) e quindi applicarla nella realizzazione di tutti i diagrammi dei componenti.

Visualizzare le dipendenze tra componenti al livello di interfaccia

Nella decomposizione del sistema in componenti, risulta del tutto naturale che alcuni utilizzino i servizi esposti da altri. Questa situazione viene mostrata collegando i componenti per mezzo della relazione di dipendenza. Molto importante è evidenziare che ciascun componente dipende dai servizi esposti, e quindi dall'interfaccia, del componente fornitore.

Evidenziare la struttura dell'architettura

L'architettura dei sistemi component-based prevede il concetto di multistrato, in cui, ovviamente, ciascuno strato è popolato da una serie di componenti. Nella realizzazione dei diagrammi dei componenti è opportuno cercare di evidenziare questa configurazione, eventualmente, visualizzando i componenti per livelli.

Diagrammi di dispiegamento

Definizione dei diagrammi

I diagrammi di dispiegamento (*deployment diagrams*) mostrano la configurazione a tempo di esecuzione delle risorse con capacità elaborative, eventualmente corredate dagli elementi che vi risiedono e che ivi possono essere eseguiti. Questi elementi tipicamente sono parti software (componenti e oggetti) le cui istanze rappresentano la manifestazione, a tempo di esecuzione, di unità di codice. Tutti quei componenti che non esistono a tempo di esecuzione (per esempio perché inglobati in altri) non appaiono in questa tipologia di diagramma, ma sono di competenza dei diagrammi dei componenti.

Quantunque non comunissimo, il formalismo dei diagrammi di dispiegamento si presta a essere utilizzato anche per modellazioni non strettamente legate al deploy del sistema. Per esempio può essere utilizzato per la modellazione dell'ambiente business. In questo caso i nodi si prestano a rappresentare lavoratori (*worker*) e unità dell'organizzazione, mentre i componenti software possono essere utilizzati per rappresentare procedure e documenti utilizzati dai lavoratori e, in generale, dalle unità dell'organizzazione.

Figura 11.6 — *Versione di diagramma di dispiegamento a livello istanza. Diagramma tratto dalle specifiche ufficiali (fig. 3-97).*

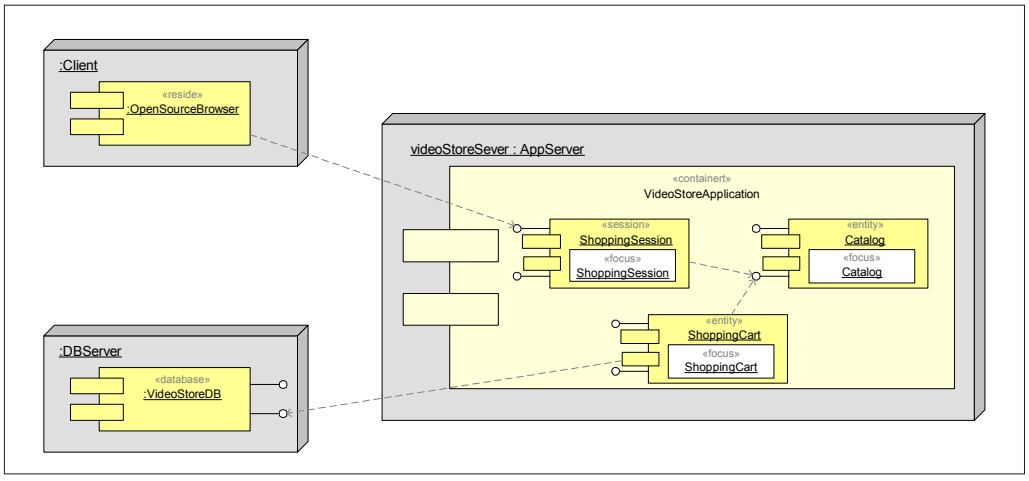
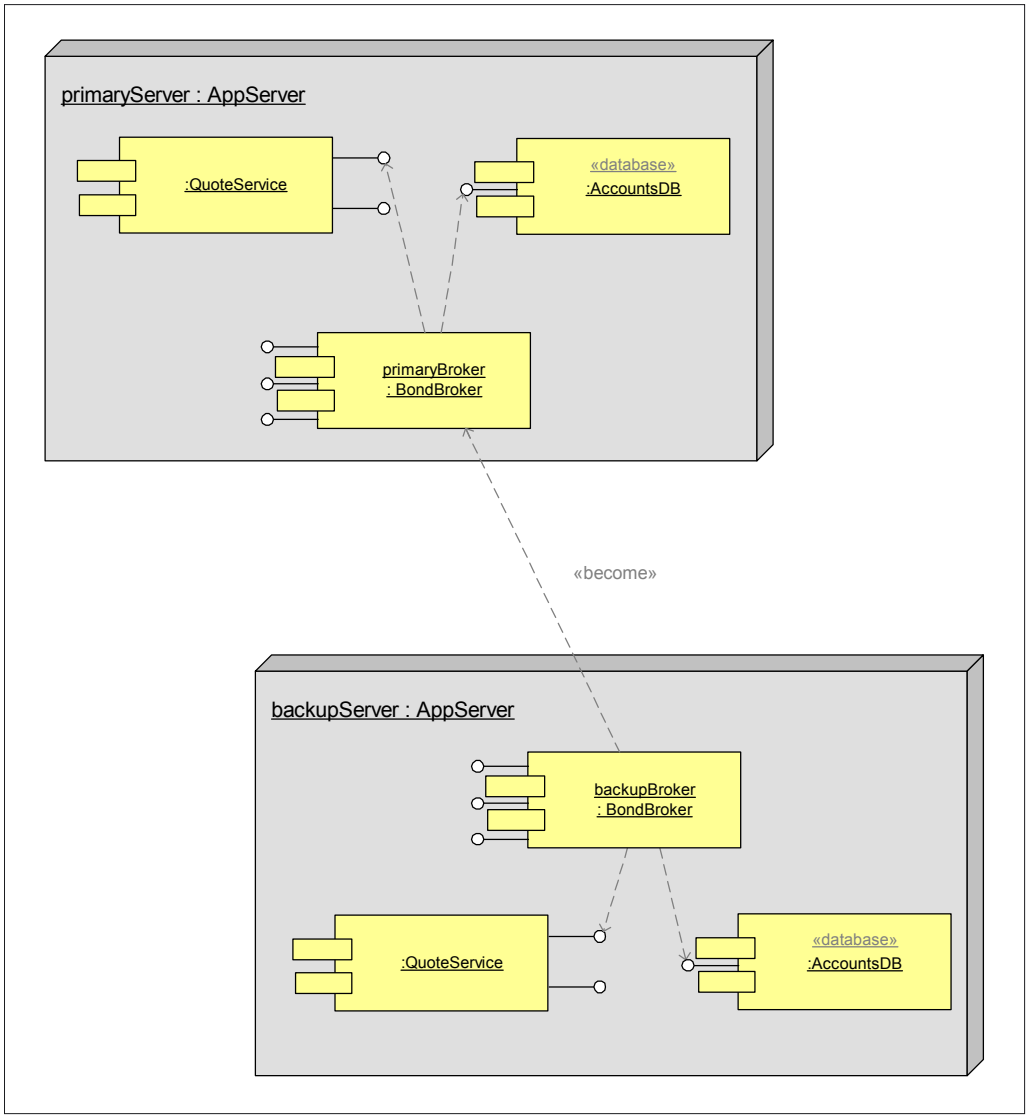
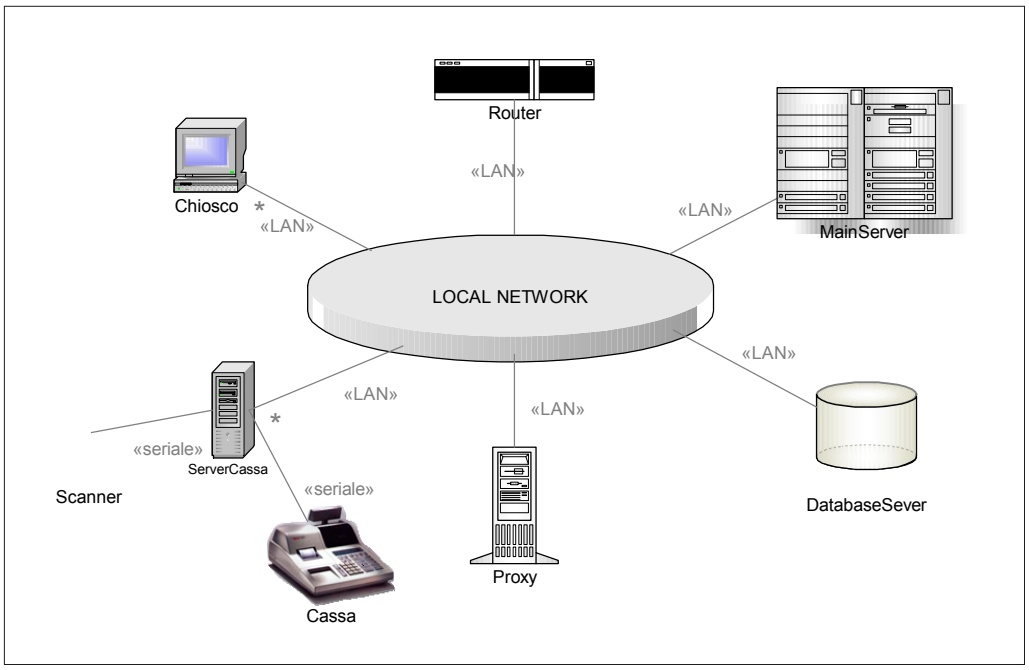


Figura 11.7 — *Illustrazione dello stereotipo* `<<become>>` *della relazione di dipendenza Tratto dalle specifiche ufficiali dello UML (figura 3-98).*



Dal punto di vista della notazione, un diagramma di dispiegamento è costituito da un grafo di nodi interconnessi da associazioni di comunicazione. In questo ambito con il termine nodi, si intende effettivamente un'istanza della metaclasse `Node`. Negli elementi

Figura 11.8 — *Esempio di diagramma di dispiegamento utilizzato per rappresentare la configurazione hardware del sistema di un supermercato. Si tratta di un diagramma al livello di tipo e quindi è possibile riportare la molteplicità delle varie associazioni.*



nodi è possibile visualizzare l'elenco delle istanze dei componenti che vi risiedono e quindi vengono eseguiti. I componenti, a loro volta, possono contenere istanze di classificatori. Ciò indica che tali istanze risiedono nel nodo.

Al contrario dei diagrammi dei componenti, la notazione dei diagrammi di dispiegamento prevede anche la versione a livello di istanze. Questa versione può essere utilizzata per mostrare la versione delle istanze dei componenti (fig. 11.6).

Come visto in precedenza, i componenti sono collegati tra loro (attraverso opportune interfacce) per mezzo di relazioni di dipendenza. Qualora necessario è possibile utilizzare opportuni stereotipi della relazione di dipendenza al fine di aumentare il contenuto informativo.

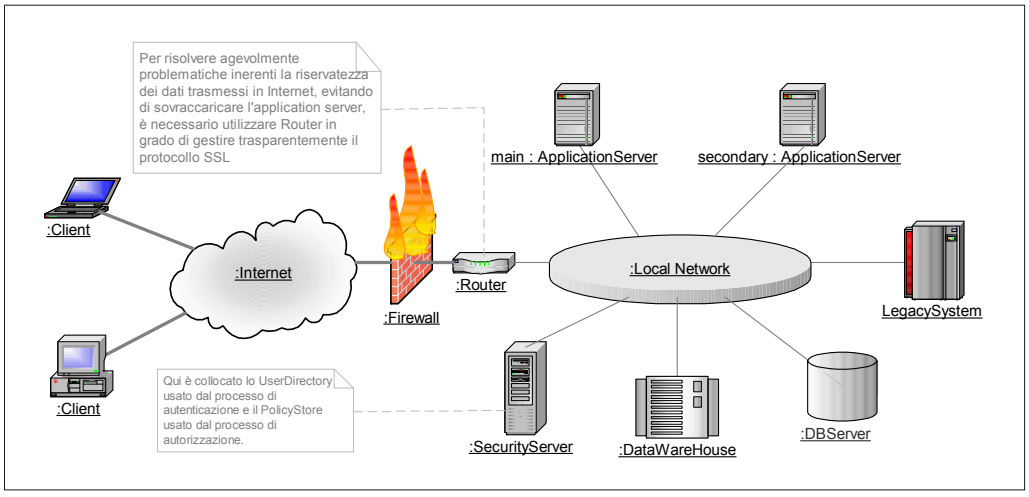
I diagrammi di dispiegamento sono spesso impiegati per mostrare quali componenti risiedono su quali nodi. In tal caso i componenti sono relazionati con i nodi "contenitori" per mezzo di una freccia tratteggiata con riportata la parola chiave `<<deploy>>` (chiaramente il componente è riportato nella coda della freccia), oppure inserendo direttamente il componente all'interno del simbolo del nodo.

Qualora risulti necessario mostrare la migrazione di istanze di componenti da un nodo a un altro, è possibile ricorrere, nuovamente, all'utilizzo della relazione di dipendenza. In questo caso lo stereotipo da adoperare è `<<become>>` (fig. 11.7).

> Le versioni iniziali dello UML menzionavano due stereotipi utilizzabili per i nodi, denominati *device* (dispositivi) e *processor* (processori). La differenza consisteva nel fatto che mentre i *processor* posseggono capacità elaborative (sono in grado di eseguire i componenti che vi risiedono), i *device* tipicamente non hanno questa capacità, almeno al livello di astrazione di interesse e, tipicamente, sono elementi di interfaccia con il mondo esterno. Per esempio, considerando la fig. 11.8, e ipotizzando che gli scanner siano della tipologia senza microprocessore installato, si tratterebbe di un ottimo esempio di device.

Nella fig. 11.9 è mostrato un esempio di diagramma di dispiegamento di una tipico sistema realizzato in architettura J2EE.

Figura 11.9 — *Esempio della configurazione di un sistema realizzato in architettura J2EE. L'esempio di figura è stato selezionato per evidenziare l'importanza di commentare i diagrammi anche di livello fisico. In particolare, alcune volte può essere utile risolvere esternamente dall'Application Server problematiche relative alla riservatezza dei dati. Sebbene, per questioni di sicurezza si vuole che i dati crittografati giungano direttamente all'Application Server, e solo lì siano decodificati, in alcuni contesti ben definiti, è possibile demandare tale incarico a un apposito dispositivo semplificando l'architettura software (nel caso di architetture J2EE non vi è la necessità di realizzare due "main servlet", con tutte le implicazioni relative all'autorizzazione e autenticazione utente, una collegata alla porta HTTP e l'altra a quella HTTPS).*



Gli utenti, attraverso una connessione sicura (SSL), evidenziata nel diagramma di fig. 11.9 dal tratto più spesso, si collegano al sito: l'Application Server assolve anche a compiti di Web Servicing. Superato il `Firewall`, si giunge fino al `Router`, il quale ha le responsabilità di gestire in modo trasparente il protocollo SSL e di inoltrare la connessione all'`Application Server`. A questo punto, se nel corso della medesima sessione l'utente non è stato precedentemente autenticato, procede al riconoscimento fornendo le proprie credenziali. Queste sono controllate e verificate dal `SecurityServer` per mezzo dello `UserDirectory`. Successivamente e in modo trasparente, l'utente continua a utilizzare l'applicazione. Ogni volta richiede di utilizzare un servizio, ha luogo l'autorizzazione, sempre a carico del `SecurityServer`, che in questo caso si avvale del `PolicyStore`. I dati a supporto dell'applicazione sono amministrati dal `DBServer` e dal `LegacySystem`. Il traffico locale tra i vari dispositivi è sempre controllato ed esaminato dai vari `Firewall` "trasparenti", non evidenziati nel disegno per semplicità, il cui utilizzo però è molto importante e serve a rendere più difficili attacchi al sistema.

I lettori più pronti, dopo un'attenta analisi dei diagrammi di dispiegamento riportati nelle figure precedenti, potrebbero essere assaliti da qualche dubbio. In effetti, non emerge chiaramente quale ne sia il dominio; in altre parole, non si capisce se siano diagrammi di dispiegamento al livello applicativo o di sistema. Si tratta di versioni diverse di cui ci si avvale per differenti fini. La prima è utilizzata per mostrare i nodi di stretto interesse per la costruzione del sistema, quelli su cui vanno allocati i componenti sviluppati, mentre la versione al livello di sistema rappresenta la reale architettura fisica installata. Quest'ultima versione presenta, ovviamente, un livello di dettaglio maggiore e, tipicamente, solo in questa compaiono elementi quali router, proxy, ecc., a meno che non siano necessari per chiarificare qualche importante scelta di disegno, come nel caso del router di fig. 11.9.

Un'altra importante differenza è che nella versione applicativa, si attribuisce maggiore enfasi ai componenti allocati nei vari nodi.

Nei diagrammi di figura si è selezionata una via di mezzo. Da un lato sembrava troppo restrittivo visualizzare unicamente l'architettura applicativa, dall'altro si è ritenuto che la rappresentazione di troppi dettagli avrebbe finito il rischio di rendere la fruizione troppo pesante.

Utilizzo

Da quanto riportato nei precedenti capitoli, dovrebbe essere ormai chiaro che l'utilizzo principale dei diagrammi di dispiegamento consiste nel modellare la vista statica della configurazione runtime del sistema. Pertanto sono visualizzati i nodi hardware sui quali vengono allocati i vari componenti, nonché l'infrastruttura necessaria per collegarli.

Sebbene la notazione dei diagrammi di dispiegamento sia idonea per rappresentare l'organizzazione dei processi dell'area di business, questo utilizzo è abbastanza raro.

Le primissime rappresentazioni (ad alto livello di astrazione) della configurazione del sistema hanno luogo sin dalle fasi iniziali del processo di sviluppo del software. Tipicamente il primissimo embrione è prodotto già dai primi colloqui con il cliente e, ahimè, spesso avviene per mano di "ardimentosi" commerciali. Questo schema dovrebbe consistere essenzialmente nel riutilizzo di architetture dimostratesi efficaci in precedenti progetti. È fortemente consigliato affidare l'attività di disegno, anche degli embrioni iniziali della configurazione hardware, ad appositi architetti, mantenendo il personale commerciale il più distante possibile da queste attività… Altrimenti vi potreste ritrovare a dover spiegare che alcuni dispositivi non sono stati ancora inventati. A complicare ulteriormente il problema intervengono poi provvigioni offerte ai commerciali sulla ferraglia piazzata.

Il progetto iniziale dell'architettura è necessario sia per capire di quale hardware e servizi di manutenzione annessi si abbia bisogno per portare il sistema in utilizzo (nelle primissime fasi tutto gira intorno al fattore pecuniario), sia per iniziare a evidenziare importanti vincoli che caratterizzeranno lo sviluppo del sistema (come per esempio risolvere il problema della cifratura via hardware oppure via software).

Il modello iniziale di dispiegamento, focalizzato unicamente sugli aspetti hardware, tende a evolversi durante tutte le fasi del progetto. A essere sinceri, non è sempre il caso. Nella maggior parte dei casi la configurazione runtime è ben definita sin dall'inizio: si ricorre a configurazioni predefinite e dimostratesi efficaci nella pratica; si applicano i pattern architetturali. Quando poi l'organizzazione del sistema in componenti comincia ad assumere una struttura ben definita, è possibile dar luogo (con le accortezze definite nel successivo paragrafo relativo allo stile), ad apposite versioni dei diagrammi di dispiegamento con indicati i componenti residenti in ciascun nodo.

I diagrammi di dispiegamento sono molto utili per i seguenti motivi:

- permettono di capire quanto il fattore hardware, servizi correlativi annessi, incida sui costi del progetto;
- consentono di chiarire fin da subito come affrontare determinati problemi architetturali (alcune soluzioni si prestano a essere risolte sia attraverso opportuni componenti software, sia per mezzo di specifici dispositivi hardware);
- forniscono chiari riscontri circa le problematiche legate alla messa in esercizio del sistema;
- permettono di evidenziare eventuali vincoli e dipendenze (hardware e software) relative a sistemi esistenti;
- rendono possibile visualizzare la struttura interna di dispositivi particolarmente importanti;
- supportano il tracciamento della mappa della configurazione della rete hardware e software del sistema.

Stile

Utilizzo consistente degli stereotipi

Il primissimo consiglio relativo allo stile non poteva che essere quello di utilizzare appositi stereotipi ogniqualvolta sia possibile. Ciò è particolarmente semplice ed efficace qualora si utilizzino dei diagrammi di dispiegamento per mostrare l'organizzazione hardware del sistema, mentre diventa più complesso e quindi da evitare, qualora si voglia visualizzare il dettaglio dei componenti allocati nei nodi. Qualora si decida di ricorrere agli stereotipi, è assolutamente necessario farlo in modo consistente. In particolare è necessario evitare di mostrare uno stesso dispositivo con stereotipi diversi.

Da tenere presente che gli stereotipi non si applicano unicamente ai nodi, ma anche alle associazioni di comunicazione. Molti autori, tra cui Scott Ambler, suggeriscono di utilizzare questi stereotipi per evidenziare i protocolli adottati nelle comunicazioni tra i vari nodi. Secondo questa tecnica è possibile avere stereotipi del tipo: HTTP, HTTPS, rmi, rpc, JDBC, ODBC, ecc.

In generale si tratta di una buona idea che però non sempre può essere sfruttata: il protocollo può essere definito al livello di componente e non di nodo. Spesso, addirittura nell'ambito di uno stesso componente, sono presenti delle parti che comunicano con il medesimo nodo attraverso diversi protocolli. Il caso più semplice è il classico sito per il commercio elettronico. In questo caso il componente demandato alla gestione della GUI, tipicamente, adotta due diversi protocolli per la comunicazione con il client (browser): HTTP e HTTPS, a seconda che le informazioni scambiate siano, rispettivamente, non riservate e riservate.

Sebbene poi non esistano regole precise circa gli stereotipi da utilizzare, è presente tutto un retaggio culturale di simboli che è consigliato utilizzare per semplificare la comunicazione. Per esempio, i database server tendono a essere visualizzati per mezzo di cilindri, i firewall per mezzo di icone a forma di pareti, ecc. Tipicamente è sufficiente utilizzare consistentemente le icone fornite dalla collezione di *clipart* selezionata.

Utilizzare nomi significativi

Questo suggerimento appartiene all'elenco di quelli di applicazione generale. In ogni modo, analizzando diversi diagrammi di dispiegamento è possibile imbattersi in alcuni esempi che hanno nodi etichettati con nomi incomprensibili, spesso legati a quelli degli specifici fornitori. Qualora ciò accade per nodi principali, quali per esempio quello dell'application server, non è un grosso inconveniente, ma quando questi nomi sono selezionati per altri dispositivi quali router, bridge, ecc. la comprensibilità del diagramma potrebbe risultare abbastanza compromessa. Bisogna sempre ricordare che, tipicamente, non tutti i fruitori di un determinato diagramma sono esperti dell'area modellata.

Probabilmente è una buona regola utilizzare nomi più generali, quali per esempio Web server, Application Server, Database Management Server, Client, ecc.

Modellare esclusivamente i componenti di interesse per il sistema

L'elenco dei componenti presenti in un sistema di medie/grandi dimensioni potrebbe facilmente raggiungere l'ordine delle centinaia. Di questi componenti, parte sono quelli strettamente appartenenti al sistema prodotto, mentre altri sono di carattere più generale. Ora rappresentarli tutti potrebbe risultare un'impresa altamente dispendiosa e non sempre utile. Per esempio non sempre aggiungerebbe molto valore mostrare con il simbolo del componente il sistema operativo presente in ogni nodo. Pertanto è opportuno concentrarsi e rappresentare propriamente i componenti di vitale importanza per la comprensione del sistema, demandando a semplici stringhe di testo la specificazione di quei componenti non strettamente connessi con il sistema.

Evitare di modellare dettagliatamente ogni singolo nodo

La possibilità di illustrare i componenti direttamente nei nodi risulta molto affascinante. Il problema però è che, tranne in pochi casi di sistemi particolarmente semplici, raramente si riesce a rappresentare tutti i componenti allocati nei nodi in modo chiaro e comprensibile. A questo punto, invece di rinunciare completamente a questa possibilità, è possibile utilizzare un approccio abbastanza pratico consistente nel rappresentare nei vari nodi solamente i componenti più importanti, omettendo, momentaneamente, quelli di interesse secondario. La mappa completa invece si presta a essere ottenuta realizzando tanti diagrammi, quanti sono i nodi del sistema, e visualizzando per ciascuno di essi i componenti allocati. Questo tipo di strutturazione a livelli fornisce un documento di grande importanza per il team addetto al deployment del progetto. Chiaramente è conveniente dar luogo ai diagrammi di dettaglio dei singoli nodi, solo dopo aver raggiunto una versione abbastanza stabile del modello dell'organizzazione del sistema in componenti.

Un semplice processo per specificare software basati sui componenti

Premessa

Dopo le varie argomentazioni riportate nel corso di questo capitolo, il lettore potrebbe domandarsi se questa sia la sede più consona per illustrare un metodo per la specifica e la progettazione di sistemi basati sui componenti. Si è deciso comunque di riportare questo processo nel presente capitolo per una serie di motivi. In primo luogo a questo punto si dovrebbe avere una comprensione più intima del concetto di componente, con tutte le polemiche esistenti. Inoltre, questa decisione è confacente alla necessità di evitare di affrontare lo stesso argomento, in modo comunque incompiuto, in diverse sedi. Ultima motivazione è che le dimensioni del Capitolo 8 sono già considerevoli.

Come suggerisce il titolo, la tecnica presentata è tratta da quanto esposto nel libro [BIB15]. Chiaramente viene presentata una brevissima introduzione (per una trattazione completa si consiglia vivamente di leggere il libro) frutto sia delle conversazioni avute tra l'autore del presente testo e John Daniels durante la collaborazione presso la banca HSBC, sia di deviazioni scaturite dai riscontri pratici ottenuti dall'applicazione dello stesso metodo.

Sul mercato esistono altri testi e altre metodologie, anche a maggiore livello di formalità (come per esempio quella denominata COBRA illustrata nel testo [BIB18]); ciò nonostante, quella presentata di seguito si fa apprezzare per la semplicità, intuitività e, soprattutto, per la provata efficacia.

Presentazione del metodo

Il punto di partenza è considerare la tecnologia OO come le naturali fondamenta per la costruzione di sistemi basati sui componenti, o meglio, nel considerare i componenti stessi come la logica evoluzione del concetto di classe. Partendo da questi presupposti è possibile estendere i principi utilizzati per la progettazione delle classi alla progettazione del sistema in componenti. In particolare, quando si modella una classe si comincia a focalizzare l'attenzione essenzialmente sulle sue caratteristiche strutturali e comportamentali. In altri termini, inizialmente ci si concentra sul dominio dei dati trattati, o meglio incapsulati, e sulle funzioni che, agendo su questi, permettono di fornire specifici servizi. Inoltre, così come raramente ha senso progettare una classe in isolamento, allo stesso modo non ha senso organizzare il sistema in componenti senza prendere in giusta considerazione la rete dei componenti e la parte di sistema (in termini di classi) necessariamente condivisa. Per la progettazione di sistemi basati sui componenti, è possibile immaginare di applicare le stesse idee, ma su una scala più larga, o meglio a un livello superiore di astrazione. Per quanto riguarda i dati, invece di limitarsi a considerare tipi base, tipicamente, è necessario esaminare grafi di oggetti. Per quanto concerne le funzioni, occorre considerare interi servizi business (end to end).

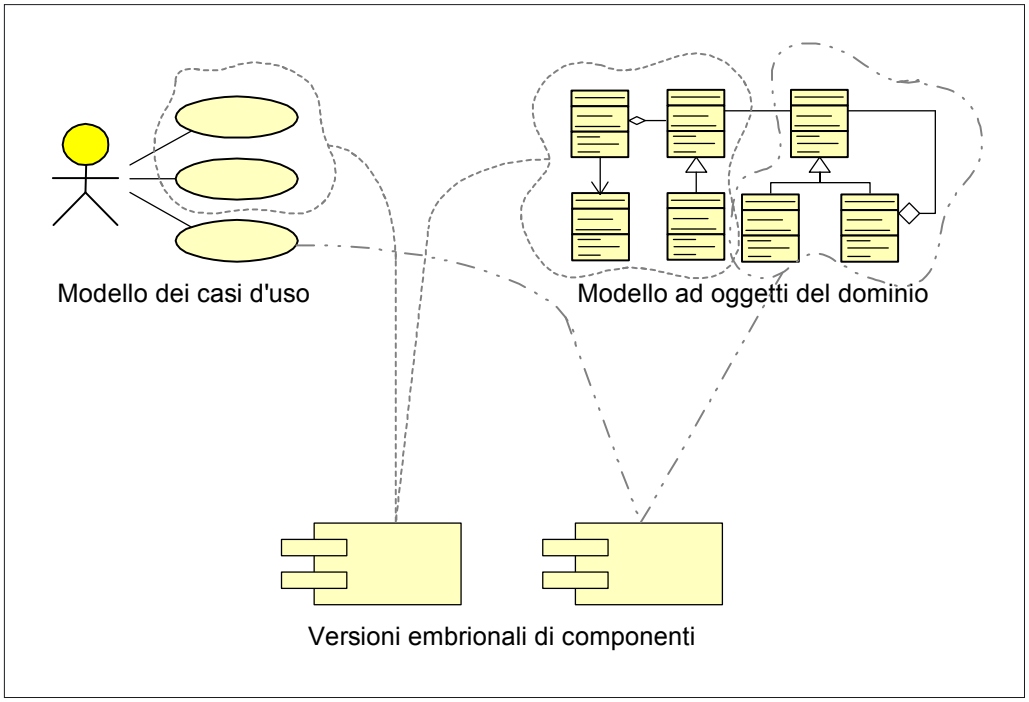
Nell'ambito dei processi formali per lo sviluppo del software, l'organizzazione dei dati e la specifica dei servizi sono affidate a due manufatti molto importanti, rispettivamente il modello a oggetti del dominio e il modello dei casi d'uso. Il primo (secondo la visione *data-driven* sposata dal presente libro) permette di modellare formalmente i dati, corredati dalle mutue relazioni, trattati nell'area business oggetto di studio, mentre il modello dei casi d'uso fornisce le informazioni circa i servizi business da fornire agli attori.

Questi costituiscono i prodotti di partenza per la tecnica proposta. Eventualmente, e non senza molti rischi, limitatamente all'attività di specifica del sistema in componenti, è possibile sostituire il modello dei casi d'uso con un elenco ragionato e dettagliato dei servizi richiesti al sistema, mentre se non si dovesse disporre del modello a oggetti del dominio sarebbe impossibile applicare il procedimento.

L'idea alla base è che i primi embrioni dell'organizzazione del sistema in componenti, o meglio le caratteristiche comportamentali e strutturali dei singoli componenti, a livello concettuale siano ottenute dal raggruppamento di porzioni di dati fortemente connessi, e dalla selezione dei servizi erogabili a partire da questi (fig. 11.10). Pertanto, si tenta di organizzare il modello grazie al contributo delle due *feature*: si raggruppano i dati, si selezionano i servizi che agiscono su di essi (si allocano le responsabilità), si esaminano i restanti servizi al fine di effettuare una sorta di bilanciamento, ossia si verifica se sia o meno il caso di dar luogo a un diverso raggruppamento dei dati (si estendono alcuni gruppi, se ne riducono altri) in funzione degli altri servizi. Pertanto, si inizia con una proiezione basata sui dati e poi la si bilancia con una basata sui servizi. Naturalmente nulla vieta di fare il contrario: si seleziona un insieme di servizi molto coesi e poi si stabilisce quali siano i dati su cui agire. Il metodo esposto sul libro [BIB15] sancisce di avviare il processo con una proiezione basati sui dati. I lettori più attenti dovrebbero aver intuito, dietro a quanto esposto fin qui, che gran parte del metodo si riconduce all'applicazione di due princìpi cardini del disegno OO di sistemi: massima coesione e minimo accoppiamento. Chiaramente questa applicazione avviene con un grado di astrazione superiore, al livello di componente.

Il rapporto che relaziona i casi d'uso ai componenti è chiaramente del tipo *n* a *n*: i servizi descritti in un singolo caso d'uso possono essere implementati attraverso diversi componenti, così come uno stesso componente fornisce una serie di servizi richiesti in diversi casi d'uso. Queste molteplicità prevedono degenerazioni, come per esempio un servizio specificato da un caso d'uso implementato da un solo componente. Molto dipende dalla granularità dei casi d'uso, però la situazione più ricorrente è proprio quella della relazione *n* a *n*. Per esempio, si consideri l'ormai famoso sistema di e-commerce. Ebbene, è abbastanza lecito attendersi una serie di casi d'uso specificanti servizi inerenti la gestione del carrello della spesa, un altro insieme relativo alla gestione degli ordini (*cfr* Capitolo 4), e così via. In questo scenario, orientativamente, è naturale attendersi almeno un componente demandato all'amministrazione dei clienti, uno alla gestione del carrello della spesa, uno relativo alla gestione degli ordini, uno per le offerte commerciali, uno relativo al catalogo, e così via. Pertanto, i servizi descritti dai casi d'uso relativi alla gestione del carrello della spesa dovranno tipicamente venir implementati da diversi componenti: almeno quello della gestione del carrello della spesa, quello relativo al catalogo (per esempio per il reperimento delle informazioni relativi ai prodotti), quello relativo alle offerte, ecc. Gli stessi componenti, poi, si prestano ad essere utilizzati per la realizzazione di molteplici casi d'uso: per esempio il componente demandato alla gestione del carrello della spesa dovrebbe partecipare alla realizzazione dei casi d'uso relativi la gestione degli ordini.

Figura 11.10 — *Utilizzo del modello dei casi d'uso e del modello ad oggetti del dominio per produrre la versione embrionale dei componenti.*



Nel testo [BIB15], il modello a oggetti del dominio è denominato Business Type Model. Chiaramente si tratta semplicemente di una questione di nomenclatura. Quello che serve è un modello che permetta di rappresentare l'organizzazione dei soli dati oggetto di interesse del sistema da realizzare, quindi il modello a oggetti del dominio.

Un passo propedeutico all'applicazione della tecnica presentata consiste nell'eseguire una primissima razionalizzazione del modello. L'obiettivo principale è preparare il modello per lo studio dell'organizzazione del sistema in componenti. In particolare è necessario cercare di separare, qualora possibile, concetti condivisi da diverse aree logiche, razionalizzare relazioni ridondanti, ecc. Non va dimenticato che, teoricamente, il modello a oggetti del dominio è realizzato da personale esperto del business ma con conoscenze non approfonditissime delle leggi dell'OO (i famosi business analyst). Pertanto il modello rappresenta una chiara e veritiera rappresentazione dell'area oggetto di studio, ma con un grado di formalità non sempre elevato e con un'applicazione delle leggi OO piuttosto discutibile.

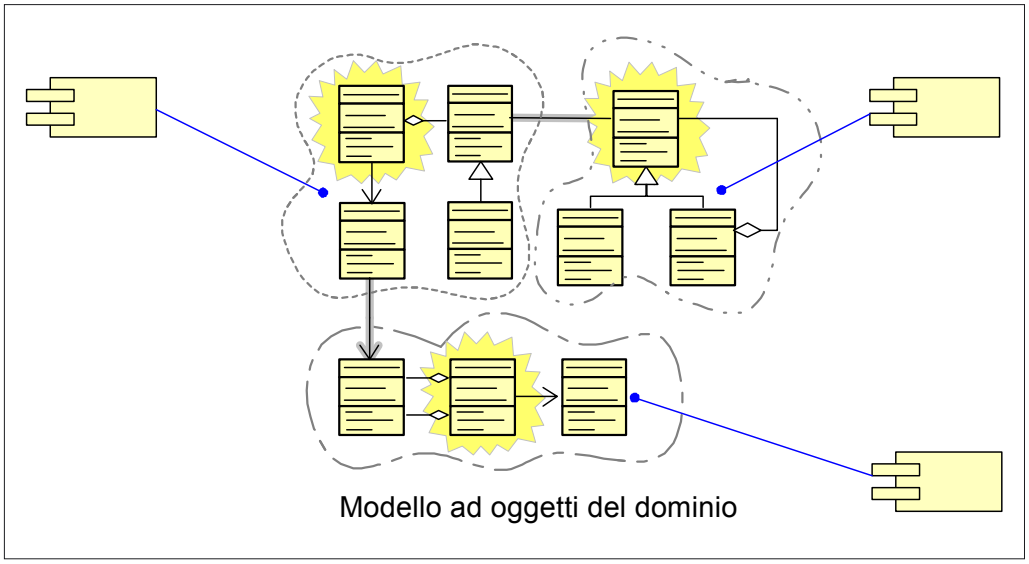
Una volta riorganizzato il modello ad oggetti del dominio, il primo problema che si pone è quello di riuscire a suddividerlo in raggruppamenti di dati a elevato livello di coesione. Il principio suggerito nel testo [BIB15] porta a tracciare una mappa tipo quella del sistema solare: è necessario individuare i vari pianeti e quindi i satelliti che vi orbitano intorno. In termini più informatici, bisogna distinguere i concetti fondamentali (*core*) da quelli a supporto. Chiaramente nessun elemento core deve essere di supporto ad un altro.

> Nel testo [BIB15] le classi identificate come concetti fondamentali sono rappresentate attraverso lo stereotipo `<<core>>` mentre le restanti per mezzo di quello `<<type>>`. Inoltre viene espressamente dichiarata la regola in base alla quale ciascuna classe può appartenere a uno solo di tali stereotipi e `<<type>>` ha la precedenza. Pertanto, in caso di incertezza, `<<type>>` è lo stereotipo da considerare.

L'identificazione degli elementi "nucleo", nella maggior parte dei casi, è abbastanza semplice e avviene a livello intuitivo. In altri casi la situazione non è così immediata, quindi è necessario disporre di alcuni criteri di supporto a questa attività. Il primo da considerare è se, dal punto di vista del business, la classe ha o meno senso qualora dovesse restare isolata. Un altro criterio è se il relativo identificatore è indipendente oppure necessita di un'altra entità. Per esempio, nel sistema di gestione della sicurezza esistono concetti come utente, profilo, ruoli, contesti di sicurezza, e così via. In prima analisi, sia l'entità utente, sia quella profilo potrebbero essere considerate sufficientemente separate, tanto da aggiudicarsi l'assegnazione dello stereotipo `core`. Analizzando più attentamente l'entità profilo, è facilmente ravvisabile che questa non ha molto senso senza la relazione con l'utente che lo possiede. Lo stesso identificatore, ragionevolmente, dovrebbe essere dipendente da quello dell'utente, magari ottenuto come: "codice utente + progressivo profilo" nel contesto dello stesso utente. Per esempio l'utente 0000010, potrebbe disporre dei seguenti profili: 0000010-001, 0000010-002, ecc.

L'identificazione di questi elementi permette di stabilire le dipendenze dei dati, ossia permette di distinguere quelli che potrebbero restare isolati da quelli che invece devono essere relazionati con altri. L'individuazione dei primi è molto importante perché questi tendono a diventare l'elemento base su cui agisce un componente. Ponendo la questione in altri termini, la prima versione dell'organizzazione del sistema in componenti prevede tanti componenti quanti sono gli elementi core (fig. 11.11). La pianificazione iniziale deve poi essere oggetto di un'approfondita analisi per verificare se sia possibile dare

Figura 11.11 — *Identificazione degli elementi core presenti nel modello a oggetti del dominio e delle relazioni intercomponenti.*



luogo a una migliore organizzazione, per pianificare l'introduzione di componenti il cui compito sia costituire il collante tra più componenti, ecc.

Il processo di raggruppamento degli "oggetti" business non è sempre semplice e immediato. Spesso accade che una stessa classe sia correlata abbastanza strettamente a diversi gruppi di "oggetti", pertanto si pone il problema di decidere a quale gruppo assegnarla. Alcune volte il problema si risolve abbastanza agevolmente approfondendo bene le relazioni con le classi appartenenti agli altri gruppi; altre volte la questione è più profonda e quindi la decisione finale dipende da altri fattori, quali per esempio i servizi da fornire partendo da quel dominio di dati. Pertanto dallo studio dei servizi da erogare (specificati nei casi d'uso), nonché dai diagrammi di navigazione della GUI è possibile evidenziare a quale gruppo è più opportuno che appartenga una determinata classe.

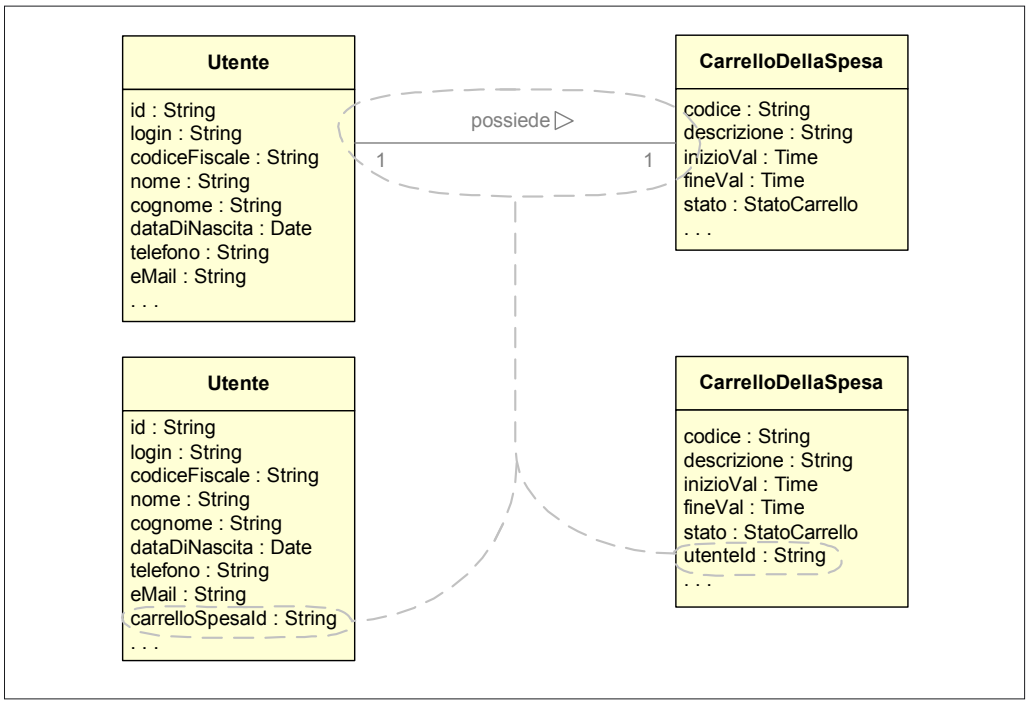
Dall'analisi della schematizzazione riportata in fig. 11.11 è possibile notare che i vari componenti, o meglio le varie parti che andranno a costituirli, sono tipicamente interconnesse: una classe appartenente a un componente è relazionata a un'altra incapsulata in un altro componente. Queste relazioni devono essere chiaramente "spezzate" (si tratta di relazioni tra elementi appartenenti a diversi componenti). La scomposizione di tali relazioni finisce per determinare la rete di dipendenze dei vari componenti e pertanto devono essere analizzate con le dovute accortezze. È appena il caso di ricordare che uno dei princìpi base da seguire nella pianificazione dell'organizzazione del sistema in com-

ponenti è la minimizzazione delle dipendenze: i princìpi della massima coesione e del minimo accoppiamento (*cfr* Capitolo 6) mantengono intatta la loro importanza, sebbene applicati a un livello di astrazione superiore.

Un quesito importante che si presenta è come far in modo che due classi relazionate possano risiedere in componenti distinti. In altre parole come fare a spezzare le relazioni di associazione. La risposta è trasformare relazioni OO (realizzate per mezzo di riferimenti in memoria) in legami relazionali (ottenuti per mezzo della corrispondenza degli identificatori: chiave primaria, chiave esterna).

Si consideri ancora una volta il caso del sito per il commercio elettronico, e in particolare la sezione inerente all'utente e al relativo carrello della spesa. Evidentemente esiste

Figura 11.12 — *Illustrazione della trasformazione di una relazione di associazione tra classi in un riferimento.*



una relazione di associazione tra le due classi ed altrettanto chiaramente le stesse classi si prestano a essere l'elemento core di altrettanti componenti. Ora è possibile eliminare la relazione di associazione includendo l'identificatore univoco di una classe nella lista degli attributi dell'altra (fig. 11.12) e viceversa.

In questo caso lo scambio degli identificatori è stato abbastanza agevole poiché le due classi sono legate da una relazione di associazione del tipo *1* a *1*. In altri casi la situazione potrebbe essere più complessa. Per esempio potrebbe rendersi utile prevedere una lista di identificatori univoci della classe relazionata. Inoltre, è stato necessario dar luogo allo "scambio" degli identificatori tra le due classi, in quanto la relativa relazione di associazione (`possiede`) non prevede un vincolo di navigabilità. Qualora presente, sarebbe stato sufficiente inserire unicamente l'identificatore di una classe (per esempio `Utente`) nell'altra classe (`CarrelloDellaSpesa`).

Da quanto appena citato, emerge una riflessione importante: al fine di minimizzare l'accoppiamento tra gli oggetti, è necessario introdurre vincoli di navigabilità. Si consideri per esempio il solito sistema per il commercio elettronico e due componenti: gestione ordini e amministrazione utente. Evidentemente tra le classi core (utente e ordine) dovrebbe esistere una relazione di associazione (indica l'utente che ha compilato l'ordine); a questo punto è necessario stabilire quale navigazione mantenere: da ordine a utente o viceversa. In altre parole è necessario stabilire se si vuole, dalla conoscenza di un ordine, essere in grado di risalire direttamente all'utente che lo ha effettuato, oppure, situazione opposta, dalla conoscenza dell'utente risalire direttamente ai relativi ordini. Si tratta di due alternative assolutamente legittime, di cui però è opportuno eliminarne una al fine di evitare un accoppiamento stretto tra i relativi componenti. L'introduzione del vincolo di navigabilità non preclude che la navigazione rimossa non possa avvenire in assoluto, bensì sancisce la mancanza di un percorso diretto. Logica conseguenza è che tale "navigazione" dovrebbe poter avvenire, ma attraverso percorsi più tortuosi, il che equivale a dire disegno più complesso e peggiori performance.

A questo punto si pone il problema sul criterio da seguire per selezionare la migliore alternativa. Si è visto che non si tratta di un dilemma shakespeariano: entrambe le alternative sono valide. In questo caso la risposta risiede nei casi d'uso e nel modello di navigazione della GUI. Se, per esempio, la navigazione specifica di selezionare l'utente da una lista e quindi premere un determinato tasto per individuare i relativi ordini, verosimilmente bisognerà lasciare la navigazione utente-ordini; se invece è previsto un servizio di reperimento degli ordini e quindi visualizzazione dei dettagli di quello selezionato (tra cui l'utente), evidentemente la navigazione da mantenere è quella ordini-utente. Nel caso in cui siano presenti entrambe le alternative si può scegliere in funzione di quella che minimizza le dipendenze e quindi migliora la riusabilità (per esempio è più frequente riutilizzare un modulo per l'amministrazione degli utenti piuttosto che uno per la gestione degli ordini), quella che si sposa meglio con altri processi interni al sistema, quella più congeniale per i criteri base dell'architettura, ecc. Se alla fine dell'applicazio-

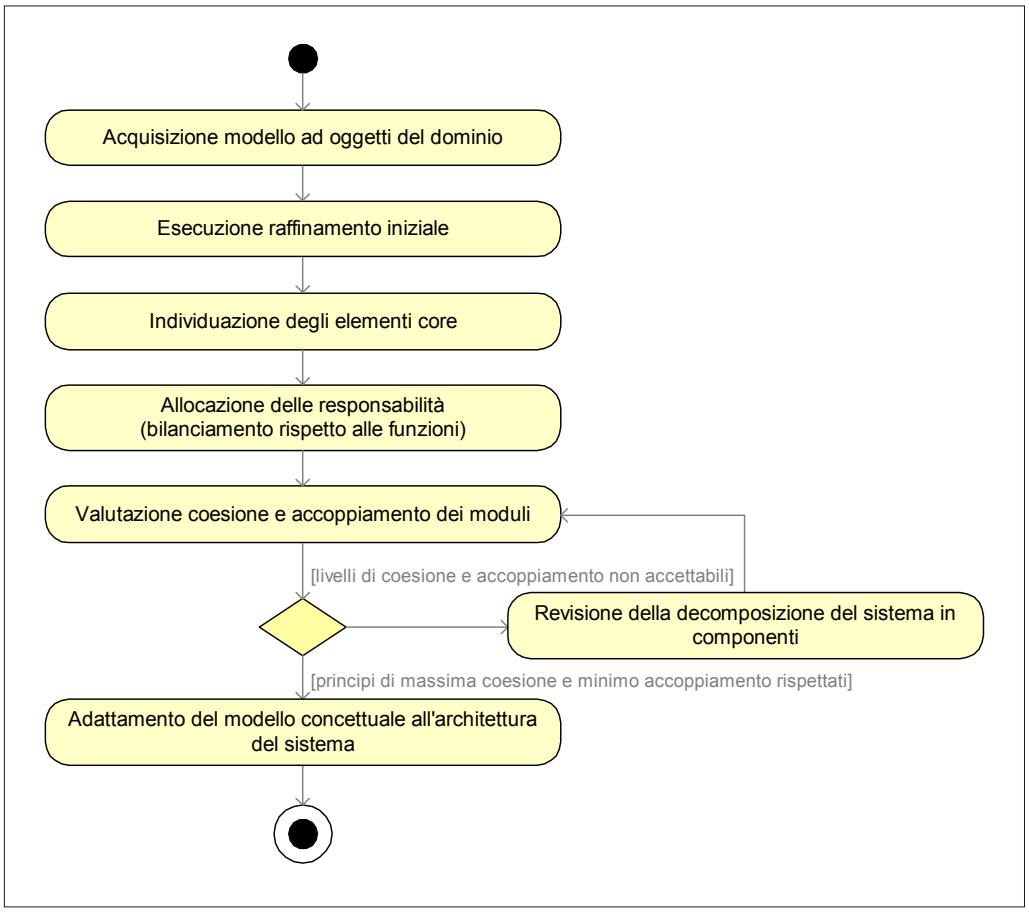
ne di tutti i precedenti criteri non si dovesse essere ancora giunti a una soluzione, si consiglia di lanciare la monetina… Documentandone il risultato.

> Durante il processo di scomposizione del sistema in componenti, è molto importante minimizzare le dipendenze tra componenti. In particolare, è necessario evitare condizioni di accoppiamento stretto e di dipendenze circolari. Queste congiunture si hanno, rispettivamente, quando un componente A dipende da un altro B (ne utilizza i servizi) il quale, a sua volta, dipende dal primo A. La dipendenza circolare si ha quando un componente A dipende da uno B, che a sua volta dipende da uno C, il quale dipende dal componente A. Chiaramente l'accoppiamento stretto è un caso particolare della dipendenza circolare. Si tratta di condizioni sfavorevoli in quanto: complicano il deploy del sistema, lo rendono meno flessibile ai cambiamenti, si possono generare lunghe ripercussioni a catena, rendono il codice più difficilmente riusabile, ecc.
>
> Pertanto, qualora ci si renda conto di aver generato situazioni di dipendenza circolare, si tenga in mente che si tratta di anomalie che possono invalidare la qualità dell'architettura e che è quindi opportuno rivedere.
>
> Chiaramente a questo suggerimento esistono delle (rare) eccezioni. Spesso trovandosi a progettare e realizzare sistemi in cui i requisiti prestazionali siano molto stringenti (per esempio sistemi real time), è indispensabile sacrificare accoppiamenti non minimi e/o assenza di dipendenze circolari a favore di migliori performance.
>
> Su queste tematiche è possibile trovare molte informazioni su testi specializzati. Probabilmente uno dei migliori è [BIB22].

A questo punto si dovrebbe disporre di un'ottima versione iniziale dell'organizzazione del sistema in componenti che però deve essere ancora considerata a livello concettuale. Al fine di renderla un vero e proprio modello di disegno (al livello di specifica naturalmente) è necessario compiere un'attività non sempre immediata: adattare il modello concettuale all'architettura stabilita per il sistema. Si tratta di rivedere il modello in funzione di altri parametri, essenzialmente legati ai requisiti non funzionali (*cfr* Capitolo 5), non considerati in precedenza. Questa attività, tipicamente, si risolve nella variazione della firma di alcuni metodi previsti per i componenti (magari si potrebbe modificare la firma di un metodo per fargli restituire una lista di oggetti invece di uno singolo), altre volte invece si rende necessario rivedere la stessa organizzazione (ulteriore suddivisione di un componente, inglobamento di diversi in uno solo, ecc).

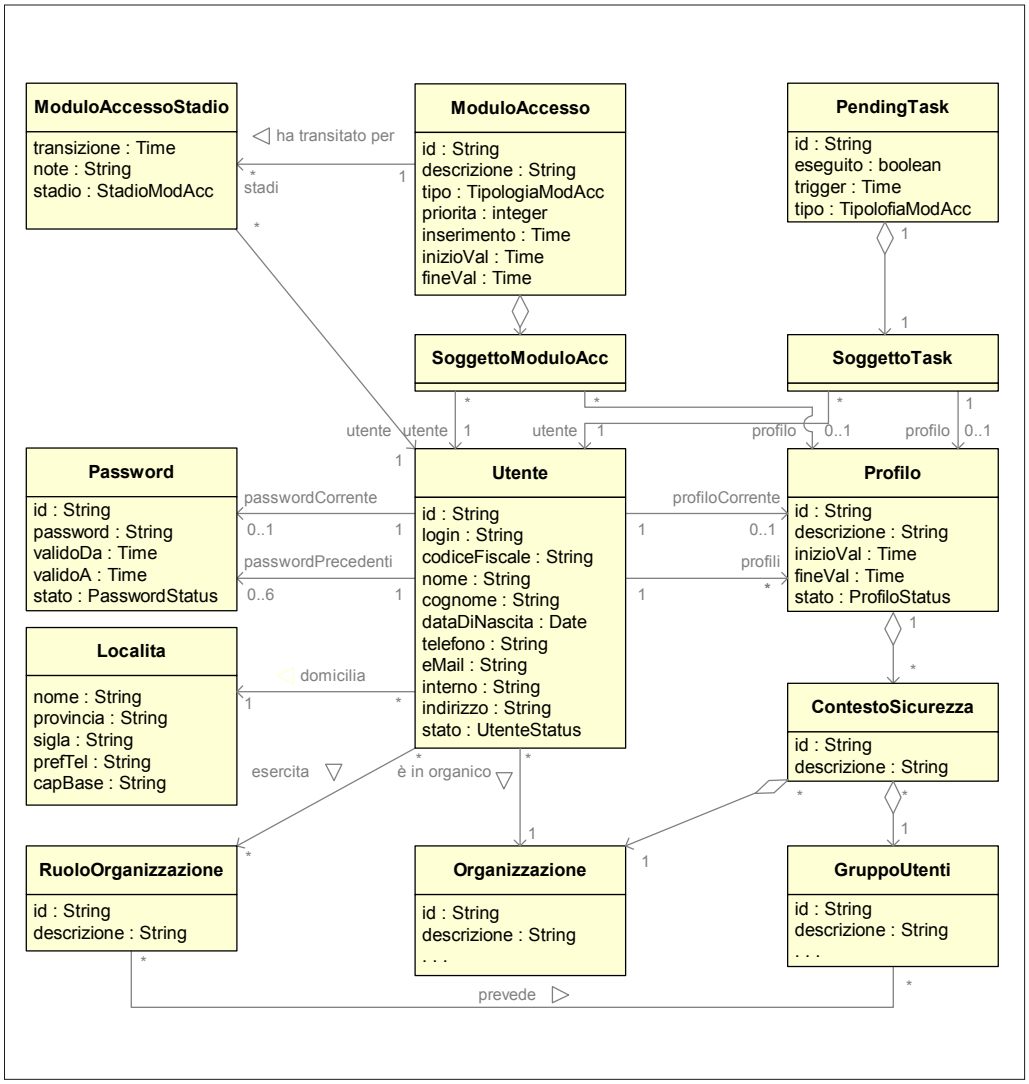
Il passo di adattamento dell'organizzazione all'architettura è necessario perché l'approccio porta alla specifica dei componenti più esterni del livello business. Le architetture moderne però sono organizzate in una serie di strati (*cfr* Capitolo 8), che nei sistemi component-based si traduce in diversi livelli di componenti. Il disegno dei restanti strati dipende dalla piattaforma e dalla strategia selezionata. Per esempio, qualora si decidesse di utilizzare l'architettura J2EE, gli strati successivi prevederebbero il business object

Figura 11.13 — *Illustrazione dei passi principali previsti dall'applicazione della tecnica.*



layer e lo strato di integrazione. Quest'ultimo, per quanto attiene l'integrazione con la base di dati, può essere realizzato sfruttando gli Entity Java Bean, oppure implementando il pattern DAO (*Data Access Object*, oggetto di accesso ai dati; *cfr* Capitolo 8, paragrafo *Esempio*, use case *Esecuzione lavori pendenti*). Per ciò che riguarda invece l'integrazione di sistemi esterni si può utilizzare la J2EE Connector Architecture. A seconda della soluzione selezionata si avrà un'organizzazione dei componenti completamente diversa, ciò che però è molto importante è aver definito la parte più esterna della business logic (in senso ampio): la restante parte della progettazione dovrebbe essere abbastanza lineare. L'intero processo può essere riassunto nel diagramma delle attività riportato in fig. 11.13.

Figura 11.14 — *Porzione del modello a oggetti del dominio relativo all'infrastruttura di sicurezza. Questa versione è il risultato di un primissimo passo di razionalizzazione: è stata aggiunta la classe* `SoggettoModuloAcc`. *Per quanto concerne la struttura dell'organizzazione e i gruppi utente, sono mostrate solo due classi (la classica punta dell'iceberg):* `Organizzazione` *e* `GruppoUtenti`. *Questa però non rappresenta un'ottimizzazione, bensì è una semplificazione utile ai fini espositivi.*



Esempio

A questo punto non resta altro che focalizzare i concetti presentati nei precedenti paragrafi per mezzo di un apposito esempio. Si provi a immaginare quale dominio si prenderà in considerazione. Risposta errata… Il sito per il commercio elettronico è stato ritenuto usurato. Si è invece preferito utilizzare il sistema di sicurezza sia perché fornisce degli ottimi spunti di riflessione, sia perché il relativo modello a oggetti del dominio è stato analizzato in dettaglio nel corso del Capitolo 8.

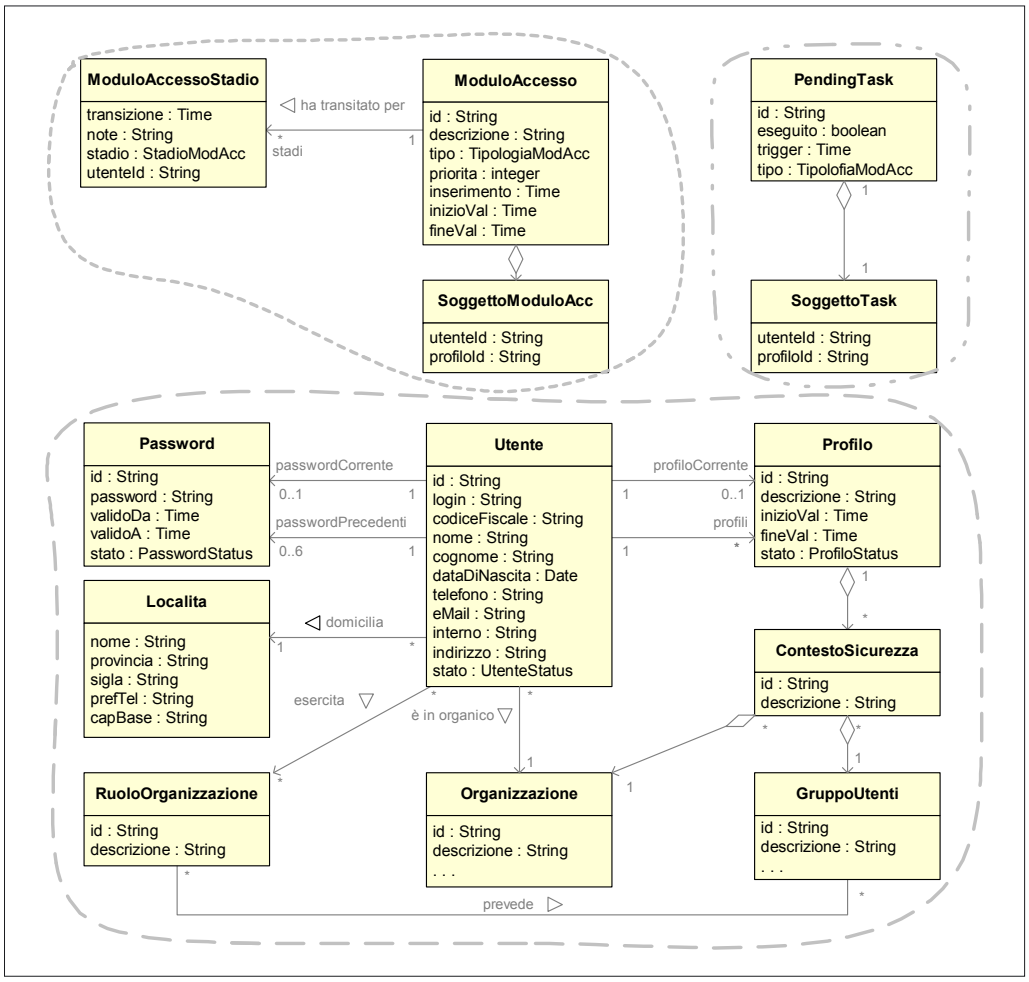
Il primo passo consiste nell'eseguire una prima ottimizzazione del modello ad oggetti del dominio. Ciò si rende necessario per renderlo più formale e più confacente al processo di scomposizione del sistema in una rete di componenti. In merito a quest'ultimo punto, è necessario rimuovere evidenti e inutili ostacoli al raggruppamento delle entità intorno ai concetti core.

Dall'analisi del diagramma in fig. 11.14, è possibile evidenziare l'introduzione di una nuova classe, denominata `SoggettoModuloAcc`. Nel modello a oggetti del sistema originale (*cfr* Capitolo 8), sia la classe `ModuloAccesso`, sia quella `PendingTask`, erano legate per mezzo di una relazione di aggregazione alla classe `SoggettoTask`. Ciò chiaramente non costituiva una situazione ottimale per la decomposizione del modello in componenti in quanto avrebbe generato un'inutile e alquanto gravosa dipendenza del componente atto alla gestione dei moduli di accesso da quello relativo alla gestione dei lavori pendenti. In generale, replicare classi al fine di semplificare la rete delle associazioni non è una buona idea; in questo caso l'introduzione della classe `SoggettoModuloAcc` (e quindi di una ridondanza) non crea alcun problema, giacché si tratta di una classe introdotta per semplificare il modello ad oggetti del dominio e non derivante da necessità emerse dall'analisi del business. Eventualmente, per motivi legati alla riusabilità del codice e di incapsulamento, è possibile dar luogo ad una super-classe della quale `SoggettoModuloAcc` e `SoggettoTask` ne siano specializzazioni.

Il passo successivo consiste nell'individuare le classi core le quali, eventualmente, daranno vita a opportuni componenti. In particolare, (*cfr* fig. 11.15), applicando i criteri evidenziati nel capitolo precedente, è possibile individuare le seguenti classi core: `ModuloAccesso`, `PendingTask` e `Utente`. Inoltre è importante cominciare a eliminare le relazioni intercomponenti. Ciò si traduce nell'introduzione degli identificatori di specifiche classi nella lista degli attributi di altre.

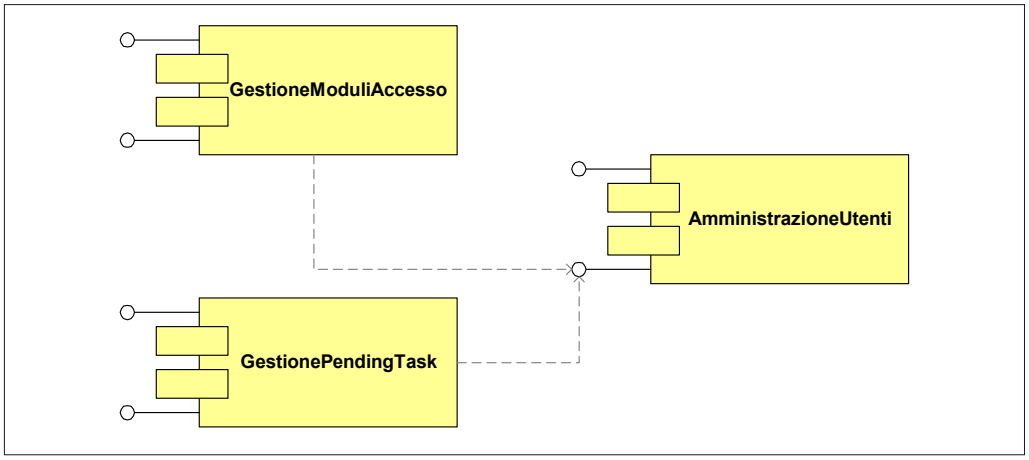
Prima di procedere oltre, è importante sottolineare ancora una volta quanto sia importante disporre di un modello a oggetti del dominio di buona qualità per eseguire l'attività di individuazione dei componenti. Per esempio, in questo contesto, per ovvie questioni legate alla semplificazione della trattazione, è stata presa in esame una sola porzione dell'intero modello. Ciò genera dei risultati (in termini di componenti e mutue dipendenze) diversi da quelli che invece sarebbero scaturiti se si fosse analizzato il modello nella sua interezza. Per esempio, si considerino le classi `Organizzazione` e `GruppoUtenti`. Si tratta di classi che (con la relativa struttura *composite* non evidenziata

Figura 11.15 — *Individuazione delle classi core.*



in questa sede) rappresentano, rispettivamente, la struttura dell'organizzazione e la gerarchia dei ruoli da assegnare agli utenti. Secondo le direttive riportate nel testo [BIB15], queste due classi, per diverse argomentazioni, potrebbero sia essere stereotipi `<<core>>`, sia semplici `<<type>>`. Come visto, in caso di incertezza, la priorità è assegnata al secondo stereotipo, e quindi non danno luogo a un componente. Ora, se si fosse condotta l'analisi considerando l'intero modello ad oggetti del dominio con la totalità dei servizi da fornire (come per esempio servizi autorizzazione dell'accesso ai servizi e filtri sui dati utilizzabili), probabilmente queste classi sarebbero state incapsulate in diversi compo-

Figura 11.16 — *Relazioni di dipendenza scaturite dalla primissima organizzazione del sistema in componenti.*



nenti. Dall'analisi delle relazioni intercomponente è possibile evidenziare le relazioni di dipendenza evidenziate nella fig. 11.16.

Il passo successivo consiste nell'allocare le responsabilità ai vari componenti: si assegnano i servizi specificati dai casi d'uso ai pertinenti componenti. A dire il vero non si tratta di una semplice operazione di assegnazione, bensì si effettua un bilanciamento di quanto modellato. Questa prima versione è essenzialmente frutto di una visione basata sui dati, l'allocazione dei servizi permette di riconsiderare il tutto dalla prospettiva delle funzioni.

Molto spesso l'esecuzione di questo passo si risolve nella conferma di quanto emerso in precedenza, altre volte invece risulta necessario riconsiderare alcuni elementi, come per esempio le dipendenze, la granularità dei componenti, ecc. Nel contesto oggetto di studio, i principali servizi da allocare ai vari componenti sono riportati nelle tabb. 11.1-3.

Tabella 11.1 — *Allocazione dei servizi al componente* `GestionePendingTask`.

Componente	**GestionePendingTask**
Servizi	**Parametri**
Inserimento nuovo task	Oggetto pending task
Eliminazione task	Id. pending task
Aggiornamento task	Id. pending task da aggiornare e oggetto pending con i nuovi dati
Reperimento task	Id pending task oppure altri criteri (utente, profilo, ecc.)
Esecuzione task;	Timestamp corrente (si eseguono tutti i task la cui data e orario di esecuzione è inferiore a quella corrente)

Tabella 11.2 — *Allocazione dei servizi al componente* `GestioneModuliAccesso`.

Componente	GestioneModuliAccesso
Servizi	**Parametri**
Compilazione modulo di accesso	Oggetto ModuloAccesso
Aggiornamento modulo di accesso	Id. Modulo di accesso da modificare, oggetto modulo di accesso con i dati da aggiornare
Validazione modulo di accesso	Id Modulo di accesso, id utente che ha eseguito la validazione (è necessario inserisce una nuova istanza della classe ModuloAccessoStadio)
Approvazione modulo di accesso	Come sopra.
Reperimento moduli di accesso	Id Modulo di accesso, altri criteri (utente, profilo)

Tabella 11.3 — *Allocazione dei servizi al componente* `AmministrazioneUtenti`.

Componente	AmministrazioneUtenti
Servizi	**Parametri**
Inserimento dati anagrafici nuovo utente	Oggetto utente
Aggiornamento dati anagrafici utente	Id. utente da modificare, oggetto utente con i dati da modificare
Aggiornamento stato utente	Id. utente, e codice nuovo stato
Eliminazione dati utente	Id. utente
Reperimento dati utenti	Id. utente e altri criteri (cognome, nome, etc.)
Inserimento nuovo profilo	Id. utente, oggetto profilo da inserire
Aggiornamento dati profilo	Id. utente, Id. profilo da modificare, oggetto profilo con i dati da aggiornare
Cambiamento di stato profilo	Id. utente, Id. profilo da modificare, codice dello stato da impostrare
Eliminazione profilo utente	Id. utente, Id. profilo da eliminare.
Inserimento dati organizzazione	Id. oggetto padre, oggetto dati organizzazione da inserire
Aggiornamento dati organizzazione	Id. oggetto da modificare, oggetto con i dati organizzazione da aggiornare
Eliminazione dati organizzazione	Id. oggetto da eliminare
Reperimento dati organizzazione	Id. oggetto da reperire e altri criteri (nome, id oggetto padre, etc)
Inserimento dati gruppo utenti	Id. oggetto padre, oggetto gruppo utenti da inserire
Aggiornamento dati gruppo utenti	Id. oggetto da modificare, oggetto con i dati del gruppo utenti da aggiornare
Eliminazione dati gruppo utenti	Id. oggetto da eliminare
Reperimento dati gruppo utenti	Id. oggetto da reperire e altri criteri (nome, id oggetto padre, etc)

> Come si può notare, la lista dei servizi riportati nelle tabelle, dovrebbe essere impostata direttamente nel diagramma dei componenti. Si è tuttavia deciso di ricorrere a questo approccio sia per ovvie ragioni di rendering grafico, sia per agevolare l'introduzioni di appositi commenti.

Dall'analisi dei servizi riportati nella tabella è possibile effettuare importanti constatazioni. In primo luogo si può notare che la granularità del componente `AmministrazioneUtenti` è certamente insufficiente. Ciò è evidente sia dal numero dei servizi, sia dallo scarso livello di coesione di alcuni di essi. In particolare si può notare la classica anomalia dovuta alla presenza di gruppi di servizi a elevata coesione, che però presentano pochissima attinenza con i restanti gruppi. La causa principale di questo problema è dovuta al fatto che l'applicazione del metodo (per questioni di esposizione) è stata eseguita considerando una porzione del modello a oggetti del dominio e di quello dei requisiti utente (casi d'uso). Verosimilmente, affrontando accademicamente la modellazione sui modelli nella loro interezza, le responsabilità relative alla gestione dei dati dell'organizzazione e dei gruppi utenti sarebbero state allocate in appropriati componenti. In ogni modo, questa anomalia torna utile per illustrare inesattezze che potrebbero verificarsi anche a seguito di una corretta applicazione del processo e le misure da prendere. Componenti a basso grado di coesione dovrebbero essere evitati per una serie di motivi: rendono difficile la riusabilità del codice, qualora non si voglia parlare in termini di riusabilità, comunque anche la sostituibilità è compromessa, c'è maggiore difficoltà di testing, ecc. In questi casi è opportuno rivedere l'organizzazione iniziale al fine di produrre una migliore decomposizione del sistema in componenti.

Un'altra considerazione molto importante è relativa al modulo amministrazione utente. Come si può notare, non sono stati presi in considerazione diversi servizi, tra l'altro molto importanti, strettamente connessi con i meccanismi di sicurezza (autenticazione utente, autorizzazione servizio, blocco/sblocco utente, sospensione/rimozione-sospensione utente, assegnazione password iniziale, aggiornamento password, ecc). La peculiare "sensibilità" di questi servizi ne rende inopportuno l'inserimento nel componente per l'amministrazione utente, inoltre si tratta di servizi assoggettati a importanti vincoli architetturali.

Per esempio, supponendo che per l'implementazione del sistema si sia selezionata l'architettura J2EE, il componente atto a fornire la sicurezza dovrebbe essere realizzato in accordo con le specifiche del meccanismo di sicurezza previsto dall'application service selezionato e quindi dovrebbe implementare ben definite interfacce. Pertanto, in questo caso, l'adattamento della decomposizione iniziale del sistema all'architettura prevede la progettazione di un ulteriore componente: `SecurityManager`. Un'importante osservazione da farsi è che il nuovo componente, `SecurityManager`, dovrebbe condividere molte caratteristiche con il componente `AmministrazioneUtenti`, come per esempio quelle relative a dati e servizi inerenti agli utenti, al profilo, ecc. Questa situazione si risolve, non come spesso accade di vedere (copiando codice tra vari componenti, minan-

do l'incapsulamento dello stato degli oggetti, ecc.), bensì sfruttando propriamente i vantaggi offerti dall'architettura multistrato. In particolare è sufficiente affidare oculatamente la responsabilità dei servizi condivisi al livello successivo: il business object layer (*cfr* Capitolo 8, paragrafo *Dipendenza del modello di analisi dall'architettura*).

Brevemente, si tratta di uno strato i cui componenti hanno il compito di incapsulare un insieme di oggetti che costituiscono il nucleo delle informazioni business corredate da opportune regole e trasformazioni. Pertanto la soluzione consiste nello specificare nello strato business object i servizi base condivisi e nell'utilizzarli nei componenti dello strato superiore (al livello di business). Dal punto di vista della protezione dei servizi sensibili, i meccanismi di sicurezza presenti nei container permettono di evitare l'invocazione di componenti interni da parte di client che non siano istanze di componenti esplicitamente abilitati ad usufruire dei servizi esposti.

Realizzato anche questo passo, purtroppo non si è ancora giunti alla fine del processo. Molto spesso è opportuno prendere in considerazione ulteriori requisiti funzionali come per esempio performance. In casi limite, si rende necessario rivedere ulteriormente l'architettura disegnata al fine di inglobare diversi componenti (anche al costo di rovinare drammaticamente l'organizzazione dei componenti) per poter far fronte a vincolanti requisiti prestazionali… Un po' come avviene anche per il disegno della base dati: si raggiunge la terza forma normale, eventualmente anche nella versione BC (Boyce-Codd Normal Form, *forma normale di Boyce-Codd*), per poi procedere a una denormalizzazione calcolata, effettuata per migliorare le performance… La pratica è sempre più complessa della teoria… Ma senza teoria la pratica diventa il territorio delle barbarie.

Ricapitolando…

Il presente capitolo è stato dedicato alla presentazione della vista fisica del sistema, la cui versione finale costituisce la concretizzazione di quanto prodotto, attraverso il processo di sviluppo del software, nelle fasi a maggiore carattere concettuale. In particolare, la vista fisica modella l'implementazione della struttura del sistema eseguibile secondo due proiezioni ben precise: l'organizzazione del sistema in componenti e il relativo dispiegamento (deployment) nei nodi che costituiscono l'infrastruttura runtime. A tal fine sono impiegati, rispettivamente, i diagrammi dei componenti e quelli di dispiegamento, indicati genericamente con i termini di diagrammi di implementazione. Quantunque nella maggior parte dei casi li si utilizza in maniera distinta, è possibile realizzare veri e propri diagrammi implementativi unendo le due notazioni: diagrammi dei componenti calati direttamente nella struttura fisica del sistema.

Nello UML, la visione iniziale della notazione dei diagrammi dei componenti prevedeva il loro utilizzo nel contesto della vista "fisica", magari prima del rilascio delle versioni iniziali del sistema o per avere un'idea più precisa del deployment. L'evoluzione della tecnologia component-based, ha esteso l'utilizzo della notazione, conferendo all'elemento componente anche aspetti concettuali. Ciò ne ha reso possibile l'utilizzo anche nelle fasi di disegno del sistema.

Le iniziali definizioni dello UML prevedevano il componente come un elemento generico, una sorta di contenitore introdotto artificialmente per raggruppare elementi necessari per l'esecuzione del sistema (script, file eseguibili, ecc.), in cui l'enfasi era tutta orientata sulla manifestazione "fisica" dello stesso. Chiaramente si trattava di una definizione limitata e spesso confusa che generava una seria riduzione della capacità di esprimere importanti concetti, una vera a propria limitazione all'utilizzo dei componenti e dei relativi diagrammi che ha generato serie ripercussioni sull'utilizzo di queste notazioni nella progettazione dei sistemi. Le principali critiche a questa visione iniziale dei componenti vertevano essenzialmente sulla confusione e sovrapposizione esistente tra il concetto di componente e altri elementi quali classi e manufatti (*artifacts*), la relegazione del concetto di componente alla sola vista fisica, la confusione intorno alla possibilità da parte di alcuni componenti di possedere altre entità o implementare interfacce (si pensi a documenti o a script), ecc.

La definizione incorporata nel documento delle specifiche ufficiali dello UML 1.4 risolve moltissime lacune, anche se ne lascia irrisolte altre. In particolare, tale definizione sancisce che "un componente rappresenta una parte del sistema, modulare e sostituibile, che incapsula implementazione ed espone un insieme di interfacce. Un componente è tipicamente specificato da uno o più classificatori che risiedono nel componente stesso. Un sottoinsieme di questi classificatori definisce esplicitamente le interfacce esterne al componente. Un componente si conforma all'interfaccia che espone, dove le interfacce rappresentano servizi forniti da elementi che risiedono nel componente. Il componente può essere implementato da uno o più manufatti (*artifact*), come file binari, eseguibili, script, ecc. Un componente può essere allocato (*deployed*) su un nodo".

Dalla definizione emerge chiaramente come nella versione 1.4 dello UML si sia finalmente riconosciuta la presenza e l'importanza dell'aspetto concettuale dell'elemento componente. In particolare, è dichiarata esplicitamente la possibilità di inserire componenti sia nei modelli di disegno, sia in quelli di carattere implementativo.

Un componente è mostrato graficamente attraverso un rettangolo con altri due di dimensioni ridotte sporgenti nel lato inferiore di sinistra.

I componenti al livello di tipo sono identificati esclusivamente dal nome del tipo (`component-type`), mentre al livello di istanza, posseggono un nome e un tipo (`component-name : component-type`) entrambi opzionali.

Oggetti che risiedono in un'istanza di un componente sono mostrati annidati all'interno del simbolo componente stesso, così come le classi che implementano un componente sono mostrate annidate all'interno del simbolo del componente stesso. Chiaramente l'annidamento indica una relazione di residenza e non di possesso.

Per gli elementi che risiedono in un componente si pone il problema della visibilità. Questa, nel metamodello UML, è specificata per mezzo dell'association-class `ElementResidence`, la quale stabilisce la visibilità degli elementi ospitati dal componente. La notazione è la stessa utilizzata dall'elemento package (si premette il simbolo della visibilità al nome dell'elemento).

Al fine di comprendere chiaramente la relazione tra gli elementi `Node`, `Component` e `Artifact`, è presentata brevemente anche la definizione di quest'ultimo elemento.

Un manufatto (`Artifact`) rappresenta un pezzo "fisico" di informazione utilizzato o prodotto da un processo di sviluppo del software. Alcuni esempi di manufatti sono: moduli, file sorgenti, script e file eseguibili. Secondo il metamodello UML, un manufatto può costituire l'implementazione di un componente

run-time. Questa affermazione sembra piuttosto bizzarra e, probabilmente, si deve a problemi di compatibilità delle versioni precedenti dello UML. Probabilmente è più logico attendersi l'opposto: uno o più componenti che implementano un manufatto.

Un nodo descrive un oggetto fisico (nel senso stretto del termine) che rappresenta una risorsa computazionale. Questa è caratterizzata dal possedere almeno capacità di memorizzazione e, molto frequentemente, anche capacità elaborative. I nodi sono gli elementi fisici nei quali vengono dispiegati i componenti.

La notazione grafica standard del componente è abbastanza banale ed è data da un parallelepipedo, pertanto è fortemente consigliato utilizzare opportuni stereotipi, per rendere i modelli di dispiegamento più chiari e accattivanti.

I nodi al livello di tipo, consistentemente con quanto visto per i componenti, prevedono un nome tipo (`node-type`), mentre a livello di istanza è possibile specificare anche il nome proprio (`name : node-type`).

La visualizzazione degli elementi (componenti) residenti su di un nodo può essere ottenuta in due modi diversi:

- utilizzando una freccia tratteggiata con evidenziata la parola chiave `<<deploy>>`;
- includendo direttamente gli elementi all'interno del simbolo rappresentante il nodo.

I diagrammi dei componenti mostrano la struttura del sistema in termini della relativa organizzazione in componenti. In altre parole, mostrano un grafo di componenti connessi attraverso relazioni di dipendenza.

I classificatori residenti nei componenti possono essere connessi a questi ultimi sia attraverso il contenimento fisico, sia utilizzando la relazione `<<reside>>`. Un discorso analogo vale per i manufatti che specificano i componenti. Questi possono essere connessi ai componenti o attraverso il contenimento fisico, o per mezzo della relazione `<<implement>>`.

Quantunque nel metamodello UML la metaclasse `Component` erediti da `Classifier`, per via di appositi vincoli, l'elemento `Component` non ha proprietà strutturali e comportamentali. Però, giacché l'elemento componente è un contenitore di altri classificatori, eventualmente definiti con le proprie caratteristiche strutturali e comportamentali, può esporre un insieme di interfacce rappresentanti i servizi forniti dagli altri elementi che risiedono nel componente. L'utilizzo dei servizi di un componente da parte di un altro viene mostrato attraverso la relazione di dipendenza che parte da un componente e giunge all'interfaccia di un altro.

I diagrammi dei componenti prevedono esclusivamente una forma di tipo e non una istanza. In altre parole non è contemplata la dicotomia tipo-istanza invece prevista da altri diagrammi. Volendo mostrare istanze di componenti è possibile ricorrere ai diagrammi di dispiegamento, eventualmente realizzando una versione degenerata in cui non viene visualizzato alcuno nodo.

La puntuale prescrizione dell'utilizzo dei diagrammi dei componenti nel contesto dei processi di sviluppo di sistemi basati sui componenti presenta ancora diverse controversie. Inizialmente veniva consigliato di utilizzare i diagrammi dei componenti per modellare l'organizzazione dei file sorgenti e delle parti fisiche che costituiscono l'implementazione. Si trattava di un utilizzo non molto interessante e sicuramente molto

riduttivo: tutti coloro che progettano professionalmente sistemi basati sui componenti sanno perfettamente che il concetto di componente deve essere introdotto già nelle fasi a carattere concettuale della progettazione del sistema. Verosimilmente è opportuno stabilire con precisione come organizzare il sistema in componenti prima di poter procedere al disegno dettagliato dello stesso. Definita l'organizzazione del sistema in componenti è possibile passare alla definizione del dettaglio dei singoli componenti. Attività che spesso genera la necessità di rivedere e raffinare il disegno iniziale dei componenti.

Per quanto concerne le linee guida relative allo stile, i suggerimenti sono di utilizzare nomi significativi per i componenti, stabilire a priori una serie di suffissi da utilizzare per indicare lo strato di appartenenza del componente in architettura multistrato, utilizzare consistentemente gli stereotipi, limitandosi a visualizzarli per mezzo di apposite etichette, mostrare le interfacce alla sinistra del componente, preferendo la versione basata sullo stereotipo, stabilire una convenzione su come posizionare le interfacce, mostrare le dipendenze tra componenti, passando per le relative interfacce e nella realizzazione dei diagrammi dei componenti, evidenziare la struttura dell'architettura.

I diagrammi di dispiegamento (*deployment diagrams*) mostrano la configurazione a tempo di esecuzione delle risorse che effettuano il processing, con eventualmente specificati gli elementi che vi risiedono e che ivi possono essere eseguiti. Questi elementi tipicamente sono parti software: componenti e oggetti, le cui istanze rappresentano la manifestazione, a tempo di esecuzione, di unità di codice.

Dal punto di vista della notazione, un diagramma di dispiegamento è costituito da un grafo di nodi interconnessi da associazioni di comunicazione. In questo caso, con il termine nodi, si intende effettivamente un'istanza della metaclasse `Node`. Negli elementi nodi è possibile visualizzare l'elenco delle istanze dei componenti che vi risiedono e quindi vengono eseguiti. I componenti, a loro volta, possono contenere istanze di classificatori: istanze risiedenti nel nodo.

I diagrammi di dispiegamento sono spesso impiegati per mostrare quali componenti risiedono su quali nodi. In tal caso i componenti sono relazionati con i nodi "contenitori", per mezzo di una freccia tratteggiata con riportata la parola chiave `<<deploy>>`, oppure inserendo direttamente il componente all'interno del simbolo del nodo.

L'utilizzo principale dei diagrammi di dispiegamento consiste nel modellare la vista statica della configurazione runtime del sistema. Pertanto sono visualizzati i nodi hardware sui quali vengono allocati i vari componenti, nonché l'infrastruttura necessaria per collegare i vari nodi.

Tipicamente il primissimo embrione dell'architettura hardware del sistema è prodotto già dai primi colloqui con il cliente. Questo schema dovrebbe consistere essenzialmente nel riutilizzo di architetture dimostratesi efficaci in precedenti progetti. Il progetto iniziale dell'architettura è necessario sia per capire di quale hardware, servizi di manutenzione annessi, si abbia bisogno per portare il sistema in utilizzo (nelle primissime fasi tutto gira intorno al fattore pecuniario), sia per iniziare a evidenziare importanti vincoli che caratterizzeranno lo sviluppo del sistema. Quando poi l'organizzazione del sistema in componenti comincia ad assumere una struttura ben definita, è possibile dar luogo ad apposite versioni dei diagrammi di dispiegamento con indicati i componenti residenti in ciascun nodo.

La realizzazione dei diagrammi di dispiegamento è molto utile per i seguenti motivi: permette di capire quanto il fattore hardware incida sui costi del progetto; consente di chiarire e pianificare la soluzione di determinati problemi architetturali; fornisce chiari riscontri circa le problematiche legate alla messa in

esercizio del sistema; permette di evidenziare eventuali vincoli e dipendenze relative a sistemi esistenti; rende possibile visualizzare la struttura interna di dispositivi particolarmente importanti; supporta il tracciamento della mappa della configurazione della rete hardware e software del sistema.

Per quanto concerne le linee guida relative allo stile, è importante: utilizzare gli stereotipi, sia per i nodi, sia per le relative associazioni, e farlo in maniera consistente; utilizzare nomi significativi; limitarsi a modellare esclusivamente gli elementi di interesse per l'architettura ed evitare di illustrare ogni singolo dettaglio.

Il capitolo si chiude con la breve presentazione della tecnica per la progettazione dei sistemi component-based, esposta nel libro [BIB15], opportunamente modificata in base alle esperienze maturate. Il punto di partenza è considerare la tecnologia OO come le naturali fondamenta per la costruzione di sistemi basati sui componenti, o meglio, nel considerare i componenti stessi come la logica evoluzione del concetto di classe. Ciò permette di applicare i principi della progettazione OO a un grado più astratto al livello di componente. Per quanto riguarda i dati che un componente incapsula, invece di limitarsi a considerare tipi base, tipicamente, è necessario esaminare grafi di oggetti. Per quanto concerne le funzioni, occorre considerare interi servizi business (*end to end*). Nell'ambito dei processi formali per lo sviluppo del software, l'organizzazione dei dati e la specifica dei servizi sono affidate a due manufatti molto importanti: rispettivamente il modello a oggetti del dominio e il modello dei casi d'uso. Questi, dunque, costituiscono i prodotti di partenza per la tecnica proposta.

L'idea alla base è che i primi embrioni dell'organizzazione del sistema in componenti, o meglio le caratteristiche comportamentali e strutturali dei singoli componenti a livello concettuale siano ottenute dal raggruppamento di porzioni di dati fortemente connessi, e dalla selezione dei servizi erogabili a partire da questi.

Un passo propedeutico all'applicazione della tecnica presentata consiste nell'eseguire una primissima razionalizzazione del modello. L'obiettivo principale è preparare il modello per lo studio dell'organizzazione del sistema in componenti. Questa attività si rende necessaria considerando che le prime versioni del modello a oggetti del dominio spesso sono modellate da personale (business analyst) senza eccessivo skill OO.

Una volta riorganizzato il modello a oggetti del dominio, è necessario riuscire a suddividerlo in raggruppamenti di dati a elevato livello di coesione; bisogna distinguere i concetti fondamentali (*core*) da quelli a supporto. I criteri da considerare per individuare gli elementi core sono due. Il primo è basato su una prospettiva legata al business; in particolare, è necessario selezionare le classi che continuano ad aver senso anche se lasciate isolate. Il secondo è basato su una prospettiva più tecnica, e consiste nel verificare se il relativo identificatore è indipendente oppure necessita di un'altra entità.

L'identificazione di questi elementi permette di stabilire le dipendenze dei dati, ossia permette di distinguere quelli che potrebbero restare isolati da quelli che invece devono essere relazionati con altri. L'individuazione dei primi è molto importante perché questi tendono a diventare l'elemento base su cui agisce un componente.

Un quesito importante che si presenta è come far in modo che due classi relazionate possano risiedere in componenti distinti. In altre parole come fare a spezzare le relazioni di associazione. La risposta è trasformare relazioni OO (realizzate per mezzo di riferimenti in memoria), in legami relazionali (ottenuti per mezzo della corrispondenza degli identificatori: chiave primaria, chiave esterna).

Al fine di minimizzare l'accoppiamento tra gli oggetti, è necessario introdurre vincoli di navigabilità. L'introduzione del vincolo di navigabilità non preclude che la navigazione rimossa non possa avvenire in

assoluto, bensì sancisce la mancanza di un percorso diretto. Logica conseguenza è che tale "navigazione" dovrebbe poter avvenire, ma attraverso percorsi più tortuosi, il che equivale a dire disegno più complesso e peggiori performance.

A questo punto si dovrebbe disporre di un'ottima versione iniziale dell'organizzazione del sistema in componenti che però deve essere ancora considerata a livello concettuale. Al fine di renderla un vero e proprio modello di disegno (al livello di specifica naturalmente) è necessario compiere un'attività non sempre immediata: adattare il modello concettuale all'architettura stabilita per il sistema. Si tratta di rivedere il modello in funzione di altri parametri, essenzialmente legati ai requisiti non funzionali, non considerati in precedenza. Il passo di adattamento dell'organizzazione all'architettura è necessario perché l'approccio porta alla specifica dei componenti più esterni del livello business. Le architetture moderne però sono organizzate in una serie di strati (*cfr* Capitolo 8), che nei sistemi component-based si traduce in diversi livelli di componenti. Il disegno dei restanti strati dipende dalla piattaforma e dalla strategia selezionata.

Il capitolo si conclude con l'immancabile esempio di applicazione del processo.