Capitolo 3

Introduzione agli Use Case

Il problema di "questo particolare" progetto è che le specifiche continuano a variare!
LVT

Un software veramente forte è in grado di cambiare i propri utenti
DONALT KNUTH

Introduzione

Il presente capitolo è dedicato all'illustrazione di una delle notazioni fornite dallo UML per supportare il delicato processo di analisi dei requisiti utente: la *Use Cases View* (vista dei casi d' uso).

In questa fase della modellazione del sistema è importante cercare di astrarsi il più possibile dall'implementazione dello stesso: bisogna concentrarsi **su cosa** il sistema dovrà fare e **non sul come**.

Le varie viste dei casi d'uso sono modelli importantissimi nel contesto del ciclo di vita del software, sia perché sono quelli su cui il cliente appone la sua firma, sia perché stabiliscono in maniera inequivocabile — o dovrebbero farlo — quali funzioni il sistema dovrà fornire, le relative modalità di fruizione nonché il modo in cui trattare eventuali anomalie che potrebbero verificarsi.

Costituiscono inoltre, già di per sé un prodotto di eccezionale importanza da consegnare al committente. Il modello dei casi d'uso corredato da altri manufatti — il modello a oggetti del dominio per esempio — condensa l'analisi del business dell'organizzazione. La tecnologia evolve molto rapidamente, mentre in genere l'evoluzione del business — si consideri per esempio un sistema bancario — è decisamente meno rapida. Pertanto le varie viste dei casi d'uso si prestano a essere usate per molteplici funzioni:

implementazioni di nuove versioni del sistema, ottimizzazione dei processi interni, addestramento del personale, ecc.

I processi di sviluppo del software prevedono diverse versioni di modelli dei casi d'uso (*cfr* Capitolo 2), principalmente business e requirements (anche detta di dominio). Quest'ultima può essere ulteriormente raffinata nella versione detta di sistema. Procedendo nelle varie fasi del processo si assiste a una graduale transizione sia del linguaggio, sia del livello di astrazione. Per ciò che concerne il primo, si assiste a un progressivo passaggio dal linguaggio del cliente a quello tecnico. Per quanto riguarda il livello di astrazione, si passa da un livello elevato a uno sempre più dettagliato fino a incorporare suggerimenti derivanti dall'architettura del sistema.

La maggior parte dei processi di sviluppo del software prevedono l'integrazione di approcci Use Case Driven. Ciò implica che la vista dei casi d'uso divenga il punto di riferimento per tutto il ciclo di sviluppo del sistema: dalla progettazione — in termini sia di disegno, sia di architettura hardware — allo sviluppo, ai test.

La criticità della vista viene parzialmente attenuata dall'integrazione di altri approcci, quali, per esempio, Architecture Centric e Risk Driven.

I modelli dei casi d'uso risultano costituiti da due proiezioni: statica e dinamica. La prima viene completamente catturata attraverso i diagrammi dei casi d'uso, mentre per la seconda sono disponibili diverse alternative che variano dal linguaggio naturale a opportuni diagrammi UML (sequence, activity, ecc.).

La proiezione dinamica dovrebbe illustrare sia le sequenze di attività da svolgere quando tutto funziona correttamente (*best scenario*, scenario migliore), sia l'elenco completo degli inconvenienti che potrebbero insorgere, correlato dai provvedimenti che il sistema deve attuare per la relativa gestione.

Quindi i programmatori, durante la fase di codifica, dovrebbero implementare il modello di disegno tenendo sotto controllo i casi d'uso relativi alla parte di sistema oggetto dello sviluppo.

Da quanto detto, va da sé che errori o lacune nelle viste dei casi d'uso determinano problemi in tutte le restanti viste, fino a giungere, in situazioni estreme, all'invalidazione della qualità del sistema finale: magari il sistema è tecnicamente ben congegnato, realizzato rispettando tutti i canoni della produzione del software ma, semplicemente, non risolve i problemi del cliente. Non bisogna sempre pensare all'utente come l'"asino" di turno che inserisce i dati nei campi di input quando il cursore lampeggia.

Al fine di minimizzare i rischi dovuti a possibili analisi dei requisiti approssimative o errate, si preferisce utilizzare processi di sviluppo di tipo incrementale e iterativo, tipicamente guidati dall'attenuazione dei fattori di rischio.

Ciò permette di sottoporre al vaglio del cliente successive versioni del sistema per verificarne la rispondenza alle reali necessità ed eventualmente correggere tempestivamente la "rotta". Lo stesso utente, potendo interagire con il sistema, o con una sua versione, riesce a chiarirsi le idee e a capire quanto i requisiti analizzati siano effettiva-

mente rispondenti alle reali necessità: non a caso spesso si ricorre alla realizzazione di opportuni prototipi.

La vista dei casi d'uso è presentata partendo dalla visione globale; ne viene illustrata la suddivisione in proiezione statica e dinamica, per poi scendere via via nei particolari, con un approccio tipicamente top down.

In questa trattazione si è deciso di non attribuire eccessiva attenzione al flusso delle azioni (descrizione del comportamento dinamico degli use case) così come definito dai Tres Amigos in quanto, nella pratica, non è infrequente il ricorso a tecniche sensibilmente divergenti da quelle "standard" in grado di apportare ambìti benefici. Tali tecniche sono oggetto del capitolo successivo.

Prima di entrare nel merito dell'oggetto di studio del capitolo, si è ritenuto opportuno presentare una breve digressione su cosa si intenda con la famosa — spesso misteriosa — definizione di requisiti utente e in cosa consista il relativo processo di analisi: tutti i tecnici ne hanno una percezione più o meno intuitiva, ma spesso ciò che è chiaro a livello intuitivo non è in definitiva facile da rendere a livello pratico.

> Nel corso del capitolo si utilizzano i termini *use case* e la relativa traduzione in lingua italiana *casi d'uso* con significato assolutamente equivalente.

I requisiti utente

La realizzazione di un qualsiasi manufatto, e quindi anche di un nuovo sistema software, dovrebbe cominciare dalla raccolta dei relativi requisiti, la quale innesca il processo denominato "ciclo di vita del software".

Tipicamente la primissima fase è denominata analisi del business, in cui le informazioni raccolte e opportunamente elaborate (modello del business) costituiscono i prodotti di input della fase di analisi dei requisiti vera e propria. Visto lo stretto legame esistente tra le due fasi e le sovrapposizioni, non sempre è possibile effettuare una netta ripartizione tra le due.

L'iter inizia tipicamente con il team degli esperti analisti, corredato da qualche commerciale, che si reca presso il cliente. A dire il vero, nelle organizzazioni più complesse il compito è interamente demandato ad appositi team composti quasi esclusivamente dai famosi Business Analyst: persone esperte dell'attività economica del cliente.

Obiettivi della prima spedizione sono essenzialmente due:

1. capire (o carpire) le esigenze che determinano la necessità di un nuovo sistema informatico o dell'aggiornamento di quello esistente;

2. analizzare lo scenario del cliente con particolare attenzione alle risorse disponibili: umane, tecnologiche (parco macchine, connessioni, reti, software utilizzati…) e, perché no, anche di budget.

È consigliabile iniziare con una serie di interviste ai manager, sia perché essi sono in grado di fornire una buona visione di insieme dell'organizzazione e della relativa attività economica, sia per evitare di urtarne — da subito — la suscettibilità.

Gli analisti più esperti fanno precedere a ogni intervista l'invio di un documento riportante i quesiti e gli argomenti di cui si intende discutere: proprio come taluni giornalisti politici italiani (trascurabili "pretese" dei cosiddetti politici). In questo contesto, tuttavia, si tratta di un buon espediente… sempre che i manager leggano il questionario.

Terminata la tornata iniziale, si redige un primo documento, corredato da un glossario, con la spiegazione degli acronimi e dei termini tecnici utilizzati (vocabolario del dominio del problema).

Il passo successivo, purtroppo spesso trascurato, consiste nello scendere via via di livello nell'organigramma dell'organizzazione del cliente, fino a intervistare coloro che dovranno interagire direttamente con il sistema: i "poveri" utenti finali. Tipicamente, questi mancano di una visione d'insieme, ma in compenso conoscono ogni dettaglio sul modo in cui le procedure vengono realmente eseguite: dalla teoria alla pratica.

Durante le interviste può sorgere qualche problema: alcuni dipendenti, sentendosi minacciati dall'introduzione di un nuovo sistema e magari indagati in qualche modo sulle relative mansioni, tendono a divenire improvvisamente reticenti ed enigmatici.

Terminata anche questa fase, si redige una seconda versione del documento e… qui cominciano le dolenti note. Il problema che può emergere a questo punto è che le versioni fornite dai manager potrebbero risultare discordanti o addirittura contraddittorie rispetto a quelle fornite da coloro che eseguono fisicamente le varie procedure.

A chi credere? Si tratta di affrontare l'eterno dilemma tra teoria e pratica. Cosa fare? Si reiterano le interviste, cercando di dettagliare maggiormente i punti controversi nel tentativo di rendere il tutto coerente, si riscrivono i vari documenti, ecc. Nei casi più estremi è necessario giungere a confronti all'americana — si mettono i manager faccia a faccia con i subalterni — nei quali, *stranamente*, le versioni dei manager tendono a prevalere.

Si riorganizza di nuovo quello che ormai è diventato un enorme ammasso di appunti e si redige un ennesimo documento che man mano ha acquistato anch'esso di corposità.

In questo documento sono condensati i famosi **requisiti utente**: le richieste legittime del cliente, le sue elucubrazioni mentali — in termini tecnici si possono trovare sinonimi più incisivi — le distorsioni, le aspettative, tanti *omissis*…

Quando ci sono da esprimere pareri o suggerimenti, nessuno riesce a fare a meno di fornire la propria visione: ognuno vorrebbe dimensionare il nuovo sistema a propria

immagine e somiglianza. Quando poi giunge il fatidico momento di firmare il documento delle specifiche, si assiste al gioco del "fuggi fuggi" meglio noto come secondo sport nazionale: lo "scaricabarile". Infine, ottenuta la validazione dal cliente, non è infrequente un'attività di rielaborazione: si comincia a lavorare sul serio.

Il primo embrione di analisi dei requisiti è tuttavia di grande interesse per i commerciali, in quanto dà un'idea del giro d'affari che ruota intorno al progetto: fa capire quanto ci si possa guadagnare…

Obiettivi dell'analisi dei requisiti utente

Gli obiettivi principali che si intende raggiungere con l'esecuzione del processo di analisi dei requisiti utente sono:

- decidere e descrivere le specifiche, funzionali e non, concordate con il cliente le quali confluiscono nel contratto che l'azienda sviluppatrice sottoscrive con il cliente;
- fornire una descrizione chiara e consistente delle funzionalità che il sistema dovrà realizzare: prodotto guida delle altre fasi del ciclo di vita del software;
- generare un modello che consenta di eseguire il test del sistema;
- iniziare a progettare l'interfaccia utente del sistema;
- disporre, attraverso le varie versioni degli use case, di una traccia dell'evoluzione del sistema e/o dei requisiti utente (i quali, tipicamente, si "automodificano" a ogni ciclo lunare);
- realizzare le prime versioni dei modelli a oggetti (business e/o dominio);
- eseguire la stima dei tempi e quindi dei costi;
- pianificare il processo di sviluppo del sistema (attribuzione delle priorità, definizione delle iterazioni, ecc.).

Per poter realizzare la vista dei casi d'uso, è necessario definire precisamente il sistema (in termine dei suoi "confini"), identificare correttamente gli attori, definire le funzioni dei vari diagrammi, determinare le relazioni tra use case e validare il sistema stesso. Si tratta di un'attività che richiede un'elevata interazione con il cliente e con le persone che rappresentano gli attori del sistema stesso.

Ogni volta che viene proposto uno strumento per l'analisi concettuale di un sistema si afferma che esso può essere facilmente compreso da clienti senza esperienza informatica (vedi diagrammi E-R e/o DFD): in realtà ciò si verifica molto raramente. Pertanto l'analisi dei requisiti deve essere comunque espressa anche per mezzo di strumenti non eccessivamente formali, quali per esempio opportuni template o addirittura il buon classico documento in linguaggio naturale. Inoltre è sempre consigliabile pianificare opportune sessioni di training relative al formalismo utilizzato, allo scopo di fornire a utenti/clienti i requisiti necessari per leggere e interpretare i vari modelli che dovranno validare.

Può succedere — ogni riferimento alla vita reale è puramente casuale — di sviluppare elegantissimi diagrammi dei casi d'uso ben strutturati, colmi di relazioni e di generalizzazioni, giungere al momento di revisionarli con il team del cliente, e vedere poi comparire progressivamente nei volti un certo sconforto di fronte al livello di astrazione prodotto.

È bene ricordare che, sebbene anche la vista dei casi d'uso debba essere un modello OO a tutti gli effetti, è talvolta ahimè necessario sacrificare parzialmente la propria formalità al fine di favorire il dialogo con il cliente.

Lo UML (secondo quanto riportato nei libri dei Tres Amigos) prevede di descrivere il comportamento dinamico dei casi d'uso per mezzo del relativo flusso di eventi, il quale specifica quando uno use case inizia, quando termina, quando interagisce con un attore, quali oggetti vengono scambiati, ecc. Il ruolo dei vari diagrammi dei casi d'uso, per tutta una serie di motivazioni, dovrebbe, in qualche modo, venir dimensionato a rappresentarne una sorta di overview. La speranza è che l'utilizzo di un formalismo grafico renda il tutto più accattivante, immediato e comprensibile al cliente.

Raramente è possibile la descrizione completa di tutti i requisiti utente tramite soli grafici e una deficienza grave nell'analisi dei requisiti può generare grandi problemi nel processo di sviluppo del software. Ciò però non deve neanche generare la paralisi del processo di sviluppo per il timore di proseguire nelle fasi successive con requisiti non ancora completi: si utilizzano processi iterativi e incrementali anche per mitigare questi rischi. In ogni modo, la rappresentazione grafica è decisamente utile e accattivante, ma assolutamente non sufficiente.

Un ultimo aspetto — solo in senso temporale di esposizione — da tenere bene a mente è che **i requisiti utente variano**: bisogna rassegnarsi all'idea che, per quanto meticolosi, esperti e scaltri si possa essere, è impossibile prevedere tutti i dettagli fin dall'inizio. Sebbene ciò possa apparire in contrasto con gli insegnamenti ricevuti nei banchi dell'università, la realtà è che l'evoluzione è insita nel destino dell'universo e che i sistemi informatici non esulano da tale regola.

Il sogno di alcuni manager informatici consiste nel congelare le specifiche prima di proseguire nell'attività di codifica. Sebbene ciò possa sembrare logico, si corre "l'insignificante" rischio di realizzare un sistema, magari bellissimo, che però semplicemente non era quello desiderato dal cliente e che non è di alcun aiuto: con un esempio, l'utente ha bisogno di una scaletta per arrampicarsi sull'albero e gli viene fornito un ascensore adatto a un grattacielo.

Invece di investire un tempo eccessivo nell'analisi dei requisiti nel vano tentativo di studiare ogni dettaglio per realizzare un modello "perfetto" prima di procedere alla fase successiva, paralizzando tra l'altro l'intero processo, forse potrebbe essere opportuno cercare sì di fare del proprio meglio, ma comunque avanzare con un processo di sviluppo di tipo iterativo e incrementale. Ovviamente ciò non significa realizzare un prodotto di scarsa qualità… "tanto dovrà cambiare". È necessario anticipare il problema progettando sistemi flessibili, cercando di circoscrivere le aree più a rischio di variazione, studiando opportuni piani per la gestione dei temutissimi "change requirements".

Forse è il caso di tenere sempre ben presenti le parole di E.H. Bersoff: "Indipendentemente dalla fase del ciclo di vita in cui ci si trova, il sistema varierà e il desiderio di cambiare persisterà per tutto il ciclo di vita".

Diagramma di "diritti e doveri" del cliente

Uno dei problemi storici nel mondo della progettazione dei sistemi informatici risiede nel difficile rapporto, in genere di carattere comunicativo, che spesso si instaura tra tecnici informatici e committenti che necessitano di sistemi informatici per migliorare la qualità e l'efficienza del proprio lavoro. Si tratta di problematiche antiche che rivivono sotto forme moderne.

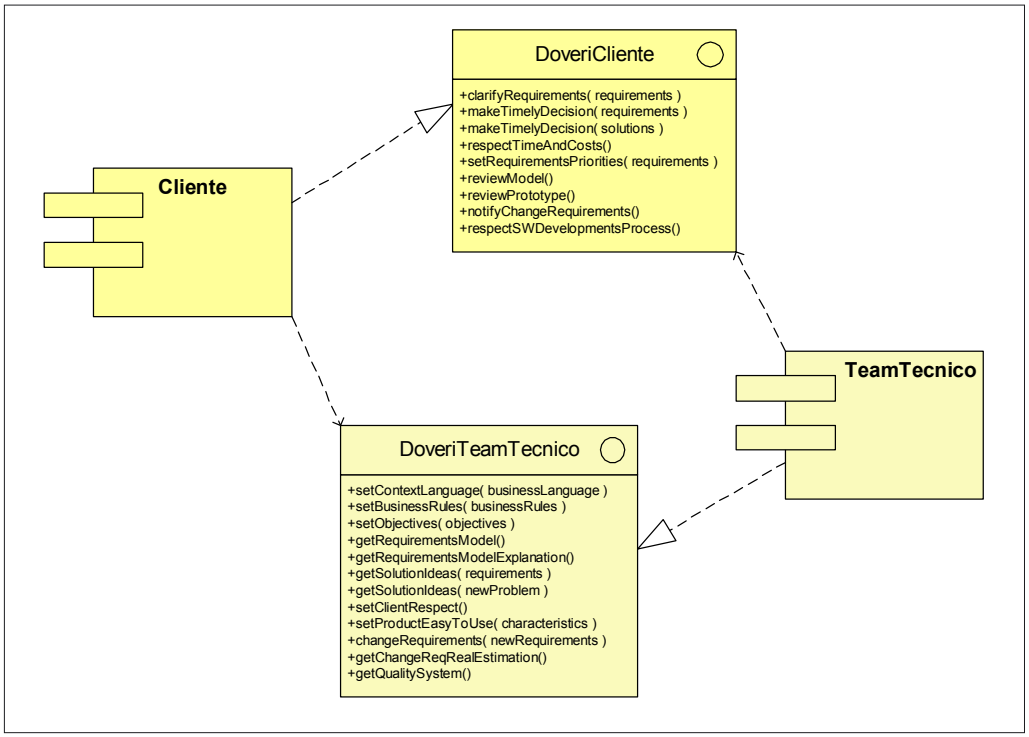
Materia certamente nota: la conoscono tutti fin dai primi corsi di programmazione, eppure si continua a perseverare nell'errore in modo decisamente diabolico (con accezione latina). Obiettivo del presente paragrafo, giocando un po' con i diagrammi dello

UML, è illustrare quelli che dovrebbero essere diritti e doveri delle due parti in causa: team tecnici e utenti/clienti. Si è deciso di presentare l'argomento utilizzando un espediente tecnico — interfacce e componenti — nel tentativo di rendere più piacevole la trattazione. (Ci si augura che ciò non disturbi l'eterno sonno dell'eccelso Cesare Beccaria… A dire il vero ci sarebbero altri motivi ben più reali del presente paragrafo per turbare il riposo del sommo pensatore).

Un'interfaccia (in termini di classi) è a tutti gli effetti un contratto tra classi client che utilizzano i servizi e la classe server che li fornisce.

Nel contesto della presente trattazione, entrambi i componenti (team tecnico e cliente) espongono una propria interfaccia (si impegnano a erogare determinati servizi: quelli appunto definiti nella propria interfaccia) e utilizzano i servizi esposti dall'interfaccia dell'altro componente. Quindi entrambi presentano sia comportamento da client (utilizzano determinati servizi), sia da server (implementano i servizi dichiarati dalla propria interfaccia).

Figura 3.1 — *Diagramma dei componenti di "diritti e doveri dei clienti".*



Il componente `Cliente`, poiché è quello che alloca le risorse economiche, può pretendere di accettare la sottoscrizione da un componente che esponga l'interfaccia desiderata (offre precisi servizi).

> Quanto riportato nei prossimi paragrafi è una rivisitazione tecnica di vari libri e articoli. Il contributo maggiore comunque è dovuto alla rivista "Microsoft Press" e in particolare al Software Requirements di Karl Wiegers.
> Con riferimento alla fig. 3.1, di seguito viene presentata una disamina commentata dei vari metodi esposti nelle interfacce.
> La descrizione dei servizi esposti spesso si presta ad essere illustrata per mezzo della tecnica denominata design by contract (trattato in maggior dettaglio nel Capitolo 6 dedicato ai principi base dell'OO), e in particolare attraverso:
>
> 1. prerequisiti: quando un client generico utilizza un determinato servizio esposto da un server deve impegnarsi a rispettarne le eventuali precondizioni;
> 2. requisiti durante l'utilizzo: responsabilità dei server che espongono servizi è garantire il soddisfacimento di determinate condizioni durante la fruizione del servizio stesso;
> 3. postcondizioni: sempre i server devono assicurare il conseguimento dei risultati garantiti dall'erogazione di ogni servizio esposto, a patto chiaramente che i prerequisiti siano soddisfatti.

```
setContextLanguage(BusinessLanguage : Language)
```

Questo metodo indica che il componente `TeamTecnico` dovrebbe fornire un "servizio" che permetta di sincronizzare il proprio linguaggio con quello utilizzato dal cliente, in altre parole è diritto di questi ultimi attendersi che il team degli analisti parli il linguaggio "tecnico" utilizzato nel proprio ambiente di business.

Ciò però comporta il dovere dei clienti di fornire le risorse necessarie (in termini di personale, di strutture, di documentazione nonché di tempo) affinché il team tecnico possa raggiungere l'obiettivo di appropriarsi del linguaggio.

Utilizzando il `design by contract` si può asserire:

1. prerequisiti: il `Cliente` deve impegnarsi ad allocare le risorse necessarie per insegnare e verificare l'esatto apprendimento del linguaggio tecnico del business;

2. requisiti durante l'utilizzo: il `TeamTecnico` garantisce di compiere ogni "sforzo" necessario per acquisire il linguaggio tecnico;

3. post condizioni: il team tecnico padroneggerà il linguaggio tecnico dell'area business nei limiti imposti dalle relative necessità e dal tempo a disposizione.

```
setBusinessRules(businessRules : businessRule[])
setObjectives(objectives : objective[])
```

In modo completamente analogo al caso precedente, è del tutto legittimo attendersi che il cliente pretenda che il team tecnico acquisisca le regole vigenti nel proprio business e, in qualche modo, ne condivida gli obiettivi finanziari e strategici: i sistemi informatici, essendo parte importante di quelli informativi, recitano un ruolo di primaria importanza nel conseguimento degli obiettivi strategici delle aziende. Basti pensare ad esempio a siti e-commerce o a sistemi per la gestione degli investimenti bancari (area treasury).

```
RequirementsModel : getRequirementsModel()
ModelExplanation : getRequirementsModelExplanation()
```

Altro diritto/dovere del cliente è richiedere di visionare il modello dei requisiti e di ricevere tutte le spiegazioni del caso. Per quanto riguarda il primo metodo (`getRequirementsModel`), le precondizioni che deve rispettare il cliente sono di aver provveduto a invocare i metodi precedenti, ossia aver facilitato l'apprendimento del proprio linguaggio, aver illustrato in maniera precisa e puntuale le business rule, aver spiegato i propri obiettivi ecc., mentre le condizioni durante l'uso, garantite dal `TeamTecnico`, sono che il modello verrà prodotto in accordo alle informazioni fornite dal cliente e la post condizione è che il modello fornito sarà completo, rispondente alle esigenze del cliente e di elevata qualità tecnica.

Per ciò che riguarda il secondo metodo (`getRequirementsModelExplanation`), le precondizioni sono che il cliente abbia eseguito il metodo precedente, ossia abbia ottenuto una istanza del `RequirementsModel`. Le condizioni durante l'utilizzo e le post condizioni sono piuttosto evidenti: il `TeamTecnico` deve adoperarsi per chiarire i dubbi espressi dal cliente e verificare se questi celino qualche requisito non detto e quindi fornire una spiegazione puntuale, precisa ed esauriente.

```
setClientRespect()
```

Questo metodo, in prima istanza, potrebbe sembrare piuttosto banale, e invece è proprio il caso di commentarlo. Spesso i team tecnici sembrano non nutrire il dovuto rispetto per "il cliente" (in questo contesto si fa riferimento all'utente finale).

Troppo spesso si considerano i clienti come gli zucconi di turno agli ordini del cursore lampeggiante, i quali, per definizione, non riescono mai a impostare i dati in maniera corretta. Suggeriscono nulla uno dei primissimi assiomi della programmazione: "il cliente è uno sciocco" o i famosi "test dell'asino" da farsi per verificare se il sistema sia o meno a prova di utente?

Talune volte sarebbe opportuno provare compassione (anche qui, accezione latina ovviamente) per la frustrazione spesso provata dai clienti che interagiscono con nuovi sistemi. Qualora tale frustrazione esista veramente, sarebbe il caso si interrogarsi se si è fatto del tutto per neutralizzarla o quantomeno minimizzarla.

Naturalmente, se gli utenti/clienti forniscono indicazioni (requisiti) errati, difficilmente verificabili, c'è poco da fare (vige l'acronimo GIGO, *Garbage In, Garbage Out* ovvero "entra spazzatura, esce spazzatura"; a questo acronimo viene spesso preferita la più prosaica versione SISO la cui traduzione viene demandata alla fantasia del lettore…), ma se i tecnici si ingannano con un'idea preconcetta basata sulla propria, sopravvalutata esperienza e trascurano l'attività di convalida con gli stessi utenti, allora difficilmente il sistema risulterà soddisfare le reali necessità degli stessi.

Per entrare nel giusto spirito si provi a immaginare di dover realizzare un sistema complesso utilizzando un framework esistente, magari fornito da terze parti, non intuitivo e con scarsa documentazione tecnica: a chi non è capitato?

```
getSolutionIdeas(newProblem : Problem)
getSolutionIdeas(requirement : Requirement)
```

Questi metodi decretano che il team di sviluppo deve essere in grado di comprendere le reali necessità del cliente — eventualmente anche non dette o non pensate dal cliente ma comprese attraverso la sfera di cristallo chiamata esperienza — e provvedere idee e alternative relative sia ai requisiti specificati dal cliente, sia a dubbi e perplessità palesate. In altre parole è diritto del cliente ottenere proposte di soluzioni relative ai propri problemi e requisiti basate su esperienza e competenza del team tecnico. Qualora la famosa "esperienza" manchi, non è comunque la fine del mondo: è possibile sopperire con una buona dose di impegno e intelligenza; ma se mancano anche queste qualità… la situazione diviene davvero complicata.

```
setProductEasyToUse(characteristics : Characteristic[])
```

Visto e considerato che il cliente finanzia il sistema e sarà "lui" a utilizzarlo, dovrebbe essere del tutto comprensibile che si attenda di ricevere un prodotto il quale, ovviamente, assolva alle problematiche che ne hanno generato la necessità e che, al tempo stesso, sia semplice da utilizzarsi in funzione dei propri parametri/necessità.

Talune volte i team tecnici costruiscono interfacce utente incomprensibili e disordinate o magari semplicemente ottimizzate in funzione della struttura del sistema o in base a quelle che erano considerate le esigenze dell'utente dal team tecnico. Si tende banalmente a dimenticare chi sarà il fruitore ultimo del sistema stesso.

Non è infrequente anche imbattersi in situazioni in cui l'utente, per eseguire determinate funzionalità ricorrenti, sia costretto a eseguire tortuosi giri e incomprensibili procedure, mentre sarebbe del tutto legittimo attendersi un utilizzo più lineare, eventualmente ottimizzato in base ai servizi richiesti più frequentemente.

Nei casi peggiori, sempre frequenti, le complicatissime interfacce utente sono solo la punta dell'iceberg… Accade che l'intero modello del sistema sia basato su "intuizioni" tecniche, magari anche interessanti, ma lontanissime dalla realtà… In fase di consegna si assiste a crisi isteriche dell'"intuitore": "Ma come? Non si riesce a forzare la realtà alla struttura del sistema?".

```
ChangeRequirements(newRequirements : Requirements[])
```

Questo è indubbiamente il metodo che incute più paura di tutti: basta il nome per generare tremori nel team tecnico. Sfortunatamente si tratta di una pretesa legittima del cliente: l'obiettivo comune delle parti dovrebbe essere quello di progettare un sistema di qualità che effettivamente soddisfi il cliente e non di "consegnare qualcosa di eseguibile". In altri ambienti, per esempio nella costruzioni di edifici, risultano accettabili incursioni nel cantiere dei committenti con eventuali richieste di modifiche.
Pertanto non c'è da stupirsi che, iniziando a vedere il sistema reale e cominciando a interagirvi, il cliente possa chiederne dei cambiamenti, perché per esempio non era riuscito a capire in anticipo eventuali problematiche o semplicemente perché erano state fraintese. Come al solito è necessario un *mea culpa*: quando l'utente invoca il fastidiosissimo metodo `changeRequirements`, quanta colpa ha veramente il cliente?

Spesso accade anche che il team tecnico, pur avvertendo difficoltà o incongruenze, semplicemente le ignori dimenticando che prima o poi il problema si ripresenterà, ma questa volta amplificato dal fattore tempo. Continuando con l'esempio dell'edificio, è un po' come se la squadra addetta alla costruzione si rendesse conto che gli spazi lasciati per le tubature sono assolutamente insufficienti, ma proseguisse ugualmente nella costruzione, dimenticando che prima o poi bisognerà demolire il tutto e ricostruire e questa volta con pochissimo tempo a disposizione. L'importante è consegnare l'edificio per la data di presentazione prevista, e poi… Poi è sempre possibile inserire delle toppe.

```
Estimation : getChangeReqRealEstimation(newReq : Requirement)
```

Questo metodo è strettamente collegato a quello precedente. In caso di richiesta di cambiamenti dei requisiti, sarebbe naturale da parte del cliente ottenere una stima attendibile e onesta dei tempi necessari per soddisfare tali richieste.

Nella realtà accade che le richieste ritenute più divertenti e meno ostiche tendano a ricevere una stima ottimistica, mentre altre, magari anche molto importanti, tendano a venir sovrastimate. A onor del vero va detto che, indubbiamente, uno dei principali motivi di sollecitudine dei team tecnici consiste nel giocare con nuovi giocattoli. Qualora venga fiutata la possibilità di poter cogliere l'occasione per utilizzare nuove tecnologie, le stime tendono improvvisamente a divenire molto ottimistiche, mentre il caso contrario implica stime funeste.

```
System : getQualitySystem().
```

Il commento di questo metodo viene deliberatamente lasciato al lettore.

Esaurita l'analisi dei diritti del cliente, si passa ora a esaminarne i doveri…

Alcuni di essi sono già stati espressi per mezzo di precondizioni dei metodi del `TeamTecnico`. In particolare è dovere del cliente insegnare il linguaggio tecnico, le regole dell'ambiente oggetto di analisi, cercare di spiegarsi nel modo più chiaro possibi-

le, ecc. A tal fine il cliente deve allocare tutte le risorse necessarie, tra cui il tempo, per illustrare, spiegare, verificare e così via.

```
Clarification[] : clarifyRequirement(requirements : Requirement[])
```

Responsabilità dei clienti è fornire i requisiti al `TeamTecnico`, allocando il tempo necessario per chiarire quelli ritenuti meno evidenti. È importante che i requisiti e le relative spiegazioni siano precise e concise.

```
Decision[] : makeTimelyDecision(requirements : Requirement[])
Decision[] : makeTimelyDecision(solutions : Solution[])
```

Altra responsabilità dei cliente consiste nel prendere decisioni e nel farlo in tempi umani e non geologici. In generale le decisioni del cliente dovrebbero essere relative sia alla selezione delle ipotesi di soluzioni ritenute più valide tra quelle esposte dal `TeamTecnico`, sia relative a requisiti non chiari o di costoso ottenimento.

> In merito alla selezione delle varie alternative di soluzioni viene alla mente il film *La vita è bella* di e con Roberto Benigni… "Cosa gradisce? Del maiale grasso grasso grasso con patate cotte in olio grasso grasso grasso oppure del salmone leggero leggero leggero con un'insalata leggerissima?". Cosa sceglierà mai il cliente? La vocina presente nell'autore afferma che molti clienti sceglierebbero comunque il maiale grasso grasso grasso…

```
respectTimeAndCosts()
```

Questo metodo può essere visto come lo speculare del metodo `setRespect()`: così come il `TeamTecnico` deve rispettare il cliente, allo stesso modo, quest'ultimo deve rispettare la stima dei tempi e dei costi fornita dal `TeamTecnico`, e produrre ogni sforzo per limitare le proprie richieste di ridurre i tempi per la messa in opera del sistema.

> Quando le pressioni del cliente cominciano a essere eccessive, si vede la differenza tra capi progetto all'altezza del proprio ruolo e quelli che credono che il capo progetto è colui che inserisce una serie di numeri nelle caselline del progetto realizzato con Microsoft Project.

Un altro problema tipico, che si verifica di sovente, è che clienti/utenti desidererebbero esporre molto rapidamente le funzionalità richieste, o magari addirittura solamente nominarle, per poi vedersele perfettamente incorporate nel sistema nell'arco di qualche giorno. Spesso i clienti tendono a non rispettare il processo di sviluppo del software adottato dal team tecnico. Da un lato si richiede elevata qualità e dall'altro si pretende di ottenerla subito.

```
setRequirementsPriorities(priorities : RequirementsPriority[])
```

Su questo argomento si ritornerà più in dettaglio nel paragrafo dedicato al processo di sviluppo. Per adesso basti pensare che è necessario che il cliente corredi i vari requisiti

con le relative priorità. È importante però sottolineare che queste devono ritenersi a carattere prettamente indicativo, in quanto l'ordine con cui le varie funzionalità verranno realizzate deve dipendere dal piano di suddivisione del processo in iterazioni: è compito dell'architetto e del capo progetto decidere tale piano in funzione anche di altri parametri, quali per esempio la riduzione dei rischi.

`reviewModel()`
`reviewPrototype()`

Altro compito molto importante dei clienti consiste nel revisionare i modelli prodotti dal `TeamTecnico`. Spesso si tratta di un'attività tediosa e non priva di elementi di frustrazione, però di importanza fondamentale per il processo di sviluppo del sistema. Una volta validati i vari modelli si prosegue con lo sviluppo del sistema ed eventuali errori nelle specifiche possono generare conseguenze notevoli. Chiaramente il colloquio tra il team tecnico e il cliente non si esaurisce qui: è sempre cosa buona e giusta chiedere ai clienti ulteriori chiarificazioni in caso di incertezze relative ai requisiti o proporre nuove idee.

> Ciò dovrebbe essere del tutto naturale: si assiste spesso a persone che volendo costruire la propria casa, cominciano non solo a visionare i vari progetti, ma anche a metterci le mani in prima persona. Chiaramente se dovesse accadere che, per esempio, la cucina risulti troppo piccola, sarebbe compito dell'architetto, grazie alla propria esperienza, evidenziare l'anomalia.

Per ciò che concerne il metodo `reviewPrototype()`, non ci sono dubbi: il cliente adora a sua volta giocare con i nuovi giocattoli e quindi sicuramente spenderà il tempo necessario, se non di più, per analizzare i prototipi in tutti i suoi dettagli. I limiti dei prototipi sono stati evidenziati nel capitolo precedente, e li si può sintetizzare riproponendo il commento di Stroustrup: il loro difetto principale "è di somigliare terribilmente al sistema finale".

`notifyChangeRequirements()`

È responsabilità fondamentale del cliente comunicare i cambiamenti di specifiche non appena avvertiti.

> Nel caso in cui, eseguendo un'incursione nella casa in costruzione, il cliente si accorga che la cucina è effettivamente troppo piccola, lo dovrebbe comunicare immediatamente senza attendere che i vari muri divisori siano edificati e che quindi una loro risistemazione diventi piuttosto gravosa se non impossibile.

`respectSWdevelopmentProcess()`

Spesso una delle manchevolezze dei clienti è di non rispettare il processo di sviluppo del software, nel senso che loro tendenza naturale è di desiderare la botte piena e la

moglie ubriaca: sistema di qualità corredato dai vari modelli e realizzato in un paio di fasi lunari.

I sistemi software, alla stregua degli altri prodotti ingegneristici, prevedono opportune fasi e diversi modelli da produrre, ognuno dei quali richiede un certo quantitativo di tempo. Spesso sono gli stessi clienti che richiedono di codificare direttamente per poter giocare il prima possibile con il sistema.

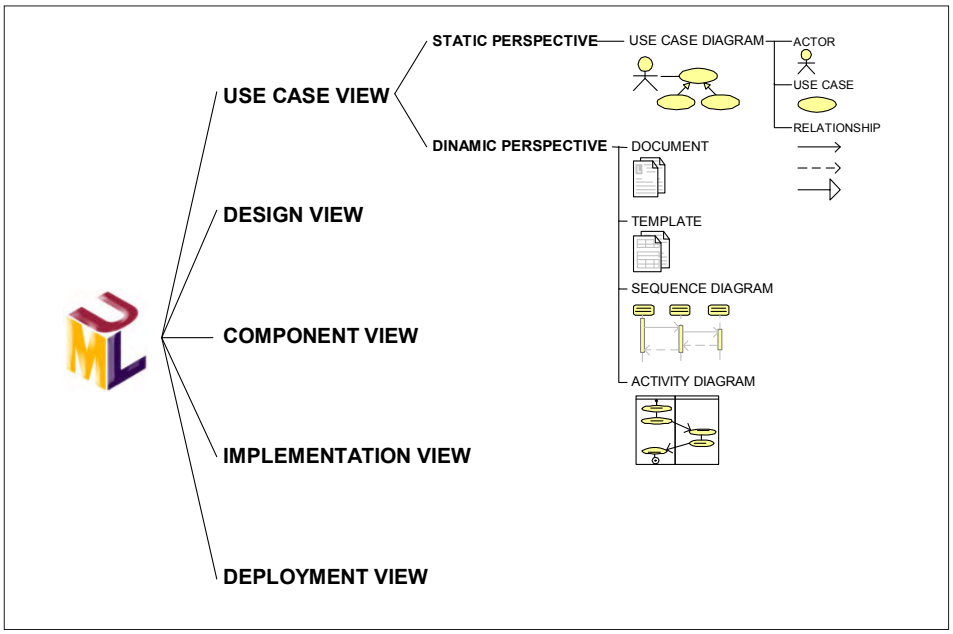
> Il sogno dei clienti/utenti è di comunicare le funzioni da realizzare, magari semplicemente citandone il nome, e poi iniziare a effettuare i vari test dopo qualche giorno. Spesso palesare l'inattuabilità di questo sogno diviene impossibile da parte di qualche commerciale di turno, ansioso forse di mascherare o farsi perdonare qualche *marachella* magari commessa in fase di fatturazione. La relativa decisione, a questo punto obbligatoria, di tenersi buono il cliente porta a promettere cose la cui portata esula assolutamente dalle reali possibilità… Iterato poi il comportamento per un almeno un paio di generazioni, si ottiene la mutazione genetica anche dei clienti.
>
> Si immagini di voler costruire la propria casa e per questo convocare due ditte edili per ottenere altrettanti preventivi. Si supponga ancora che l'addetto commerciale della prima vi dica: "Non c'è problema, in un paio di settimane inizieremo i lavori. La mia azienda dispone di squadre di muratori e carpentieri che sono degli artisti. Lavoreranno come forsennati per consegnarvi la casa dopo sei mesi. Il tutto per un costo di centomila euro, migliaio in più o in meno". L'addetto commerciale della seconda ditta, invece, vi dirà: "In effetti è necessario capire che tipo di casa si vuole e quale sia conveniente costruire, verificare il terreno, esaminare il piano regolatore, realizzare insieme qualche progetto di massima. Solo allora sarà possibile una stima appropriata. In ogni modo lo studio iniziale di fattibilità le costerà sui cinquemila euro". Ora, quanti di voi affiderebbero la costruzione della casa alla seconda azienda?

Use Cases View

Le use case view (viste dei casi d'uso) nei processi di sviluppo del software assumono un ruolo di primaria importanza, sia perché sono sottoposte alla firma del cliente — generalmente quella dei requisiti utente… e, per la firma, si consiglia di richiedere l'utilizzo del sangue come inchiostro —, sia perché le restanti viste si occupano di modellare quanto specificato in quella dei requisiti.

> La proiezione statica della vista dei casi d'uso si basa sugli omonimi diagrammi, frutto del lavoro svolto da Ivar Jacobson durante la sua collaborazione con la Ericsson nell'ambito dello sviluppo di un sistema denominato AXE. Visto il successo riscosso dai diagrammi e percepitene le relative potenzialità, i Tres Amigos hanno pensato bene di inglobarli nello UML.

Figura 3.2 — *Rappresentazione schematica della proiezione statica della use case view.*



Gli use case diagram sono utilizzati per dettagliare le funzionalità del sistema, dando particolare rilievo agli attori che interagiscono con esse. Le singole funzionalità sono rappresentate graficamente attraverso elementi ellittici denominati use case.

A questo punto si corre il rischio di fare confusione. In effetti, la prima vista logica, i relativi diagrammi, o meglio quelli che si occupano della proiezione statica, e alcuni elementi di questi ultimi, quelli che rappresentano le funzioni del sistema, si chiamano tutti, grazie a un sovrumano lavoro di fantasia, use case. Per tentare di far chiarezza si consideri il diagramma riportato in fig. 3.2.

La prima vista è quindi composta da più diagrammi, di cui quelli derogati a modellarne la proiezione statica sono gli use case diagram, i quali permettono di specificare le funzioni del sistema per mezzo di elementi grafici a forma di ellissi detti use case.

La vista dei casi d'uso, come gran parte dei modelli, prevede due componenti: statica e dinamica, di cui solo la prima è rappresentabile per mezzo degli use case diagram. Quindi, al fine di colmare la lacuna e modellare anche la proiezione dinamica, fanno la loro apparizione gli interaction e activity diagram (diagrammi di interazione e diagrammi di attività), utilizzati per rappresentare particolari istanze dei casi d'uso dette scenari. Entrambi sono oggetto di studio di appositi capitoli (rispettivamente 9 e 10); per ora basti sapere che servono per modellare aspetti dinamici del sistema.

Nei paragrafi seguenti si evidenzia che il ricorso a tali diagrammi, sebbene permetta di illustrare il comportamento dinamico delle funzioni del sistema, può risultare decisamente laborioso e non sempre di facile lettura. Probabilmente opportuni template dei casi d'uso finiscono per essere lo strumento migliore: sono chiari, immediati e decisamente facili da aggiornare.

Quando i diagrammi di interazione vengono utilizzati nella use cases view, ne realizzano gli *scenario*: versioni a elevato grado di astrazione dei flussi di azioni racchiusi nei casi d'uso. Tipicamente, degli stessi diagrammi vengono realizzate versioni di maggior dettaglio nella design view allo scopo di rappresentare il flusso dei messaggi che gli oggetti si scambiano per produrre un determinato risultato. Semplificando al massimo il concetto, è possibile affermare la seguente proporzione: gli scenari stanno agli use case come gli oggetti stanno alle classi.

La maggior parte dei processi formali di sviluppo del software prevede, contestualmente alla produzione della vista dei casi d'uso, la produzione di determinati modelli a oggetti. In particolare nella fase di analisi del business si realizza un modello a oggetti atto a rappresentare l'intera area di business, mentre nella fase di analisi dei requisiti si limita l'attenzione al dominio oggetto di studio. Si tratta di modelli di livello concettuale in cui le classi identificate sono appartenenti al mondo reale e non all'infrastruttura del sistema. Modelli del genere — come si vedrà di seguito — permettono di bilanciare i modelli dei casi d'uso: le due diverse proiezioni devono illustrare la stessa entità.

A lavoro concluso, i diagrammi dei casi d'uso devono rappresentare tutte le funzioni del sistema, a partire dal relativo avvio, o richiesta, che tipicamente è generata da un attore del sistema, fino alla relativa conclusione. Le proprietà che tutti i modelli dovrebbero possedere (accuratezza, consistenza, semplicità e manutenibilità) in questo contesto assumono un'importanza imprescindibile.

Gli attori, sono **entità esterne** al sistema che hanno interesse a interagire con esso. Pertanto, contrariamente a quanto lascerebbe pensare il nome, e a memorie derivanti da metodologie quali DFD (Data Flow Diagram, diagramma del flusso dei dati), gli attori sono sia persone fisiche, sia sistemi o dispositivi hardware che interagiscono con il sistema.

Va sottolineato, per l'ennesima volta che, a questo livello di analisi, il sistema va visto come una scatola nera, pertanto si deve descrivere **cosa** fa il sistema senza specificare il **come**: ci si deve occupare dello spazio del problema e non di quello delle soluzioni. L'impegno è proprio nel cercare di isolarsi, di astrarsi il più possibile da dettagli realizzativi il cui studio è demandato alle apposite fasi.

Perché utilizzare i diagrammi dei casi d'uso?

La risposta all'interrogativo posto nel titolo è piuttosto convenzionale: "perché fanno parte dello UML". Allora si provi a porlo in questi termini "perché i Tres Amigos hanno inserito i diagrammi dei casi d'uso nello UML? Quali sono i vantaggi?".

Molto brevemente le principali peculiarità dei casi d'uso possono ricondursi ai seguenti punti:

- si tratta di uno strumento facilmente comprensibile dagli utenti e quindi appropriato per comunicare con essi e per ottenere le necessarie validazioni.

- si prestano a essere utilizzati con diversi livelli di formalità. Generalmente, nelle primissime fasi si è concentrati nel comprendere più intimamente le necessità dell'utente e quindi si conferisce minore importanza ai principi dell'OO, anche perché spesso non facilmente comprensibili dall'utente medio. Nelle fasi successive invece si è anche coinvolti nel processo di sviluppo del sistema e quindi il livello di formalità deve essere necessariamente elevato.

- permettono di definire molto chiaramente i confini del sistema e di evidenziarne gli attori sia umani che non.

- evidenziano il comportamento del sistema, permettendo di evidenziare eventuali incoerenze e lacune nell'analisi dei requisiti.

- permettono di realizzare un'ottima documentazione.

- i casi d'uso opportunamente strutturati si prestano a essere utilizzati come scenari di test (i test case).

Fruitori della use case view

La vista dei casi d'uso è oggetto di interesse per un vasto insieme di figure professionali coinvolte nel ciclo di vita del sistema: dai clienti ai capi progetto, dagli architetti ai tester, dagli addetti al marketing ai commerciali e così via.

I primi, in ordine cronologico, a fruire della vista dei casi d'uso sono i clienti, i capi progetto e i *business analyst*: le use case view specificano le funzionalità che il sistema dovrà realizzare e come queste verranno fruite.

In quest'ottica le viste sono necessarie per riuscire a carpire i requisiti utente: reali necessità, sogni più o meno consci, eventuali reticenze…

In molte organizzazioni sono previsti opportuni team che si occupano unicamente di analizzare l'attività economica del cliente, di capire i requisiti del sistema e di redigere il documento dei requisiti. Sono i "padroni" della use case view. Queste figure professionali vengono tipicamente denominate *business analysts*.

La vista dei casi d'uso è necessaria poi al team degli sviluppatori (architetti, disegnatori, codificatori) poiché fornisce direttive, chiare e precise — un sogno — su come realiz-

zare il modello. In questo contesto, con il termine di architetti si fa riferimento a quelli software, ossia a coloro che si occupano di disegnare il sistema in termini di classi. Ovviamente anche gli architetti hardware hanno interesse a fruire della vista degli use case perché fornisce le prime disposizioni su come organizzare il sistema in termini di dispositivi fisici (server, connessioni, proxy, ecc.).

Nei processi di sviluppo use case driven e architecture centric esiste una mutua dipendenza tra la use case view e la vista fisica del sistema: le funzionalità da realizzare forniscono l'orientamento iniziale su come organizzare l'infrastruttura hardware così come quest'ultima influenza la modalità con la quale le diverse funzioni possono essere fruite.

Da tener presente che la proiezione dinamica, oltre a dettagliare la sequenza di azioni da svolgere, nel caso in cui tutto funzioni correttamente, dovrebbe riportare anche la casistica completa degli inconvenienti che potrebbero verificarsi durante l'esecuzione del caso d'uso e le relative contromisure che il sistema dovrebbe attuare come risposta.
Pertanto durante la progettazione e l'implementazione del sistema, il team di sviluppatori dovrebbe aver ben presente i casi d'uso a cui si riferisce la parte del sistema oggetto di disegno o implementazione.

Altra categoria di utilizzatori è costituita dalle persone che dovranno effettuare le verifiche finali (note anche come test di sistema): a partire da tale vista è possibile generare quella di test che consente di verificare che il sistema prodotto realizzi pienamente ciò che è stato sancito con il cliente.

> Sebbene in molte organizzazioni i test vengano eseguiti solo dagli sviluppatori e dall'organizzazione presente presso il committente, nelle organizzazioni più strutturate sono previsti appositi team (detti "martellatori" in quanto il loro obiettivo è prendere a martellate il sistema…) che si occupano della delicata fase di test.
>
> In generale, non è un buon principio far eseguire i test allo stesso personale che ha prodotto lo specifico artifact: i programmatori non dovrebbero essere i soli a verificare il proprio codice. Il problema è che chi realizza un "prodotto" tende, inconsciamente, a trascurare verifiche di situazioni critiche e meno chiare che, per questo, divengono più soggette a errori. A tal fine, chi effettua i test deve possedere una buona conoscenza dei requisiti utente e quindi della use case view. A dire il vero, solo per la fase di test, si dovrebbero realizzare apposite versioni delle use case view denominate test case.

Dall'elenco dei fruitori si vogliono poi escludere i commerciali? Ma certo che no. Anzi, sarebbe auspicabile che *taluni* commerciali e addetti al marketing dessero un'occhiata a quanto prodotto, giusto quel tanto che basta per capire di cosa dovranno parlare nelle varie riunioni con i clienti ed evitare di basare i propri giudizi unicamente sulla scelta dei colori presenti nelle interfacce utente. Si potrebbe "addirittura" correre il rischio di evitare di vendere congelatori agli esquimesi e impianti di riscaldamento a chi vive all'equatore.

Oppure potrebbero evitarsi frasi del tipo "Perché ci vuole così tanto a realizzare un sistema?". Ma chi glielo fa fare di stare a competere con questi tecnici paranoici e schizoidi? In fondo, esistono centinaia di lavori migliori di questo… Gestire un banco di frutta è difficile, le mele hanno la brutta abitudine di andare a male… Troppo complicato, è meglio il software, lì si può "buttare tutto in caciara" e un responsabile "esterno" si riesce sempre a trovarlo…

Le diverse versioni del modello dei casi d'uso

Il processo di sviluppo del software è una delle risorse — come tale dovrebbe essere considerato — più importanti delle aziende produttrici di software: tra le varie caratteristiche, fornisce al progettista e ai vari elementi del team direttive precise su cosa fare, quando farlo, come, dove e perché. La corretta adozione di un processo, da sola, può fare la differenza tra un progetto di buona riuscita e un insuccesso.

Sebbene non esista il processo valido in assoluto o che risulti ottimale per ogni progetto, non sono infrequenti aziende software che si affidano al famoso processo di sviluppo denominato *IDP* (*Irrational Development Process*, processo di sviluppo irrazionale), tipicamente composto da tre fasi: analisi dei requisiti, codifica, test (eseguito rapidamente prima della consegna del sistema). A complicare ulteriormente la situazione spesso interviene un'analisi dei requisiti sommaria magari effettuata solo parzialmente (in questi casi si parla in termini di *useless case*, casi inutili) da migliorarsi durante il processo di codifica, o addirittura durante e/o dopo i test.

L'imperativo dell'IDP, da trent'anni a questa parte, è sempre lo stesso: liquidare le fasi iniziali per dedicarsi alla codifica e solo alla codifica.

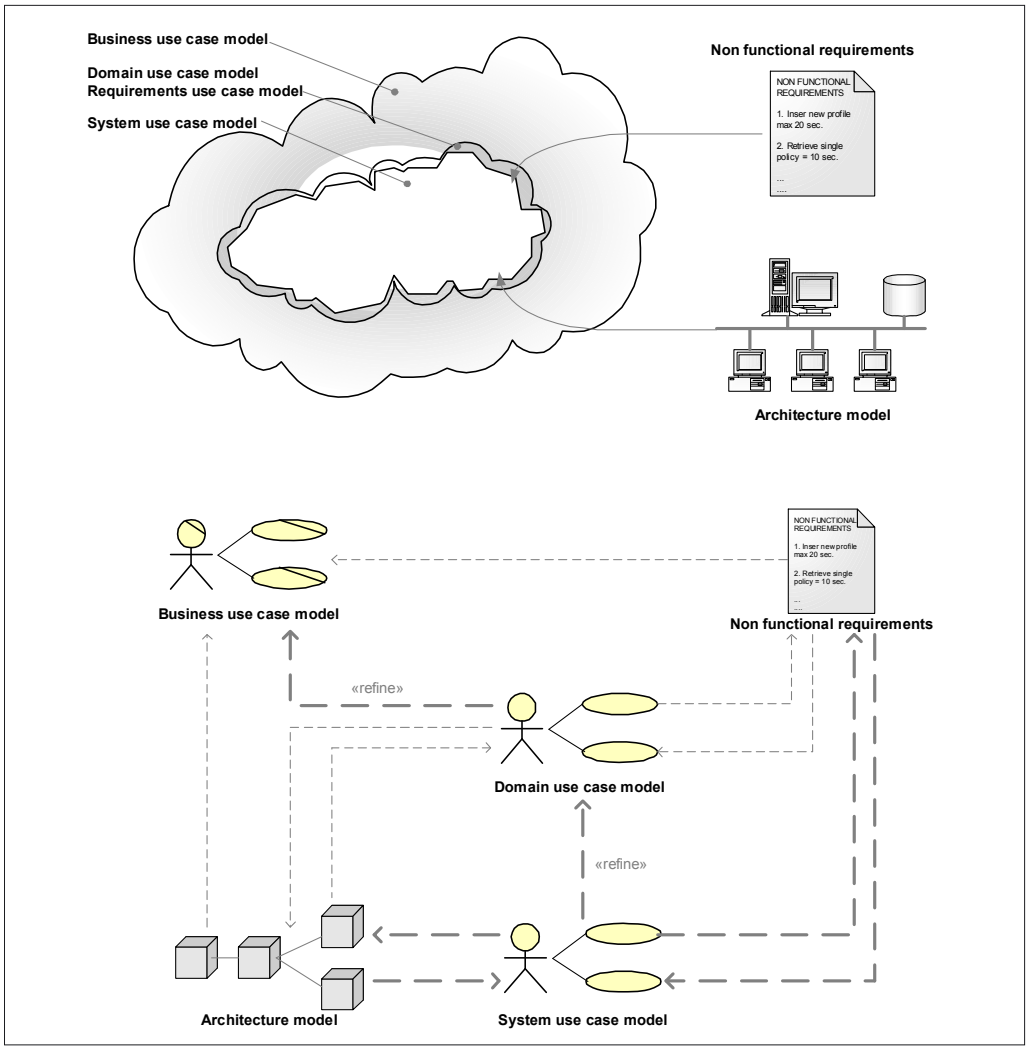
I processi di sviluppo formali prevedono una successione — eventualmente iterata per uno sviluppo incrementale del sistema — decisamente più articolata (analisi business, requisiti utente, analisi, disegno, codifica, test e consegna).

Nei processi iterativi e incrementali, le prime iterazioni (tipicamente due o tre) sono focalizzate quasi esclusivamente sull'analisi dei requisiti, attività che dovrebbe gradualmente ridursi, fino a tendere a zero nelle iterazioni finali (*cfr* Capitolo 2).

L'analisi dei requisiti è un processo complesso che coinvolge diversi modelli. In funzione delle caratteristiche peculiari del progetto (come dimensione, dominio oggetto di studio, ecc.) può essere necessario realizzare fino a tre diverse tipologie di modelli di analisi di casi d'uso (fig. 3.3).

I requisiti non funzionali sono oggetto di studio del Capitolo 5; per ora basti sapere che in questa tipologia fatti rientrare requisiti di natura strettamente tecnica quali quelli attinenti agli aspetti di performance, affidabilità, robustezza, flessibilità, estensibilità, ambiente, e così via.

Figura 3.3 — *I diagrammi in figura mostrano, utilizzando due diversi formalismi (quello in alto decisamente più intuitivo al contrario di quello in basso, più formale), le diverse versioni dei modelli dei casi d'uso, le loro reciproche relazioni, e le mutue dipendenze con altri importanti manufatti, come il modello di architettura e il documento dei requisiti non funzionali. Nel diagramma in basso (a tutti gli effetti basato sul formalismo dello UML) le relazioni di dipendenza sono rappresentate, artificialmente, con un tratteggio di diverso spessore per meglio enfatizzare il grado di dipendenza dei manufatti relazionati. Chiaramente tratteggi più spessi indicano un maggiore grado di dipendenza o, se si vuole, una maggiore influenza esercitata dal manufatto indipendente.*



Indipendentemente dai nomi che si preferisce adottare (la comunità OO sembrerebbe non essere d'accordo sulla nomenclatura da utilizzare) le diverse topologie di modelli dei casi d'uso sono:

- quella tipicamente denominata **business**. È la prima che si incontra e deve il nome al fatto di essere focalizzata sull'intera area business: non è quindi ristretta al solo ambito oggetto del sistema da realizzare (si tenta di studiare l'intero sistema informativo e non esclusivamente quello informatico). Si tratta di un modello ad elevato grado di astrazione nel quale, in teoria, non è assolutamente inglobato alcun riscontro proveniente dallo spazio delle soluzioni. Il livello tecnico informatico è pertanto contenuto, e linguaggio e formalismo sono adeguati a quelli del cliente. Si tratta di un modello funzionale alla comprensione dell'ambito in cui il nuovo sistema dovrà operare, ma non strettamente necessario per la relativa costruzione. Ciò lo rende una sorta di bene di lusso che non sempre è possibile permettersi considerata anche la complessità, il tempo da investire per la sua realizzazione, lo skill richiesto ai vari analisti, ecc. In ogni modo si tratta di un bene prezioso vantaggioso per molti fini, quali istruzione del nuovo personale, analisi/razionalizzazione dei flussi presenti nell'area business, base di partenza per la realizzazione di altri sistemi, ecc. Dall'analisi di questo modello è possibile cominciare a realizzare le prime ipotesi del modello di architettura, nonché il documento dei requisiti non funzionali.

- quella generalmente denominata di **dominio.** Si tratta di un modello abbastanza allineato con quello di business, la cui prima grande differenza è che questa volta l'attenzione è focalizzata unicamente sul dominio da automatizzare (si confina lo studio al sistema informatico). I casi d'uso cominciano ad assumere una struttura più formale e tecnica e si cominciano ad inglobare indicazioni provenienti dal modello dell'architettura e dai requisiti non funzionali.

- quella generalmente denominata di **sistema**. Si tratta di un modello decisamente più particolareggiato derivante sia dall'approfondimento del modello precedente, sia dall'assorbimento di direttive provenienti dal disegno dell'architettura tecnica. Non sempre è un modello completamente diverso da quello precedente: spesso si tratta di un'evoluzione continua, ottenuta sia dall'approfondimento degli stessi requisiti, sia da opportuni aggiustamenti generati da riscontri provenienti soprattutto dal modello dell'architettura. I casi d'uso di questo livello risultano decisamente più formali e precisi, l'organizzazione rispecchia maggiormente le leggi dell'OO, presentano riscontri e correzioni provenienti dallo spazio delle soluzione e non mancano di un certo linguaggio tecnico.

Nei progetti reali, considerata la canonica mancanza di tempo, spesso i tre modelli vengono collassati in uno solo; si fa a meno della versione business e la versione domain viene rapidamente fatta evolvere in quella di sistema.

Da quanto riportato, emerge che diversi modelli dei casi d'uso, e in particolare quello di sistema, devono incorporare vincoli e indicazioni provenienti dal disegno dell'architettura fisica (per esempio requisito non fattibile, servizio realizzabile proficuamente utilizzando un diverso approccio, ecc.). Il problema che potrebbe sorgere è che a sua volta l'architettura del sistema dovrebbe poter essere stabilita solo dopo che i requisiti utente siano stabili e il modello dei casi d'uso sia disponibile. Si corre pertanto il rischio di innescare un circolo vizioso. La soluzione che tipicamente viene adottata consiste nel realizzare prime versioni di casi d'uso delle funzionalità ritenute più importanti e quindi fornirle all'architetto del sistema. Quest'ultimo, analizzando la versione embrionale della use case view e cercando di riadattare precisi pattern architetturali, dovrebbe essere in grado di realizzare un disegno iniziale dell'architettura. Così facendo, lavorando per approssimazioni successive, si rimuove il rischio di generare un circolo vizioso. L'interazione tra le due viste continua fino al completamento di entrambe in cui il tutto deve necessariamente risultare coerente.

> Un esempio, decisamente poco formale, ma molto intuitivo, consiste nel paragonare l'evoluzione dei due modelli a quella di una coppia che si forma in discoteca. Inizialmente i due si sono appena conosciuti, devono "rompere il ghiaccio", sono un po' impacciati e ballano abbastanza distanti. Poi, con il procedere della serata, cominciano a socializzare, a conoscersi meglio, fino a giungere verso fine serata in cui si ballano in maniera decisamente ravvicinata (chissà poi perché nella realtà non va mai così bene).

Sebbene l'esperienza insegni che ogni sistema è unico, esiste tuttavia un certo numero di pattern accomunabili e quindi "riciclabili": tutti i tecnici, a qualsiasi livello del processo di sviluppo, tendono a riutilizzare soluzioni dimostratesi efficaci in precedenti progetti. A questa legge empirica non si sottraggono i casi d'uso e tantomeno i modelli per l'architettura fisica. In altre parole, se i modelli vengono organizzati propriamente, è probabile che diverse parti risultino riutilizzabili in progetti futuri.

Brainstorming inziale

Il processo di analisi dei requisiti richiede esperienza e competenza specifica sul business del cliente e sulla tecnologia di riferimento. È necessario capire di cosa abbia effettiva-

Tabella 3.1 — *Modulo per la ricapitolazione dei requisiti frutto del brainstorming.*

CODICE: *Codice*	*Breve nome mnemonico*		**Data:** Versione: 0.00.000
Descrizione:	*Descrizione del requisito*		
Impatto sul sistema:	*Aree interessate dalla funzionalità e conseguenze.*		
Stima del rischio:	*Basso/Medio/Elevato.*		
Priorità cliente:	*Bassa/Media/Elevata*		
Durata:	*Ordine di grandezza del tempo stimato per la realizzazione.*		
	Personale		**Giorni uomo**
	Figura tecnica richiesta.		*gg*
			gg
			gg
			gg
Costo stimato:	*Questa valutazione deriva direttamente dal punto precedente*		
Iter:	**Autore**	**Stato**	**Data**
	Iniziali autore.	*Proposta/Respinta/ Approvata/Da variare*	*Data*

mente bisogno il cliente: talune volte è necessario convincerlo delle relative necessità, imparare a leggere tra le righe per afferrare ciò che non viene detto — per esempio, razionalizzare un processo potrebbe nuocere a "società amiche" —, suggerirgli eventuali soluzioni — facendole magari passare per sue… — e così via.

Una buona idea per cercare di affrontare nel miglior modo possibile l'analisi dei requisiti consiste nel procedere con un **brainstorming** iniziale allo scopo di redigere un documento con l'elenco di tutti i potenziali requisiti che passano per la mente dei vari personaggi coinvolti nel processo.

Ovviamente tale lista andrebbe rielaborata al fine di catalogare tutte le idee emerse, valutarne la fattibilità, il costo orientativo, il rischio e così via (tab. 3.1). Stranamente, i requisiti più odiati dal personale tecnico riportano dei costi esorbitanti.

Il risultato di questa sottofase dovrebbe consistere in un documento riportante: codice o un breve nome mnemonico, descrizione, stima del rischio e impatto nel sistema, valutazione del personale coinvolto e del tempo necessario (quindi costo), approvazione e priorità.

La priorità dovrebbe essere assegnata dal business analyst in funzione delle direttive fornite dal cliente. Queste dovrebbero avere un valore essenzialmente di tipo indicativo: non ci si può attendere che le funzionalità vengano completamente realizzate secondo

l'ordine stabilito dal cliente. L'"ordine" finale è dato dal piano delle iterazioni nelle quali si intende scindere l'intero processo. È evidente che esso debba tener presente sia le richieste del cliente, sia altri fattori come neutralizzare o perlomeno minimizzare i fattori di rischio, delle dipendenze tra le varie parti componenti, e così via. Quindi la decisione finale delle priorità dovrebbe essere presa con la collaborazione del capo architetto. In genere si cerca di bilanciare le priorità definite dall'utente con quelle dovute a fattori tecnici.

Tipicamente i processi di sviluppo più comuni consigliano di affrontare per primi gli use case (o solo alcuni scenari: magari quelli di successo) più complicati o comunque

Tabella 3.2 — *Esempio di compilazione di un modulo di ricapitolazione.*

CODICE: ***IR_ATSND_0010***	**Autenticazione mittente / certezza integrità messaggio.**		**Data:** 2/II/2000
			Versione: 0.00.002
Descrizione:	Si esige che prima di avviare il processo di elaborazione dei messaggi relativi alle quotazioni finali dell'attività borsistica, il sistema si faccia carico della verifica dell'identità del mittente, della relativa autenticazione, e dell'integrità del messaggio stesso.		
Impatto sul sistema:	Sistema di sicurezza. (Verosimilmente è necessario memorizzare, in maniera centralizzata e crittografata, la "chiave pubblica" dei sistemi esterni abilitati a scambiare messaggi con il sistema e fornire appositi servizi che siano gli unici abilitati ad accedere a tali informazioni). Sistema di interfacciamento con i sistemi esterni.		
Stima del rischio:	Elevato.		
Priorità cliente:	Media (Per le prime release del sistema non è indispensabile)		
Durata:	27 giorni uomo.		
	Personale		**Giorni uomo**
	Business Analyst		4
	Chief architect		3
	Architect		10
	Programmatore		15
	Tester		5
Costo stimato:			20.000
Iter:	**Autore**	**Stato**	**Data**
	L.V.T.	Proposta	2/II/2000
	A.R:	Approvato	10/III/2000

più rischiosi, come dire: "se deve andare tutto a rotoli è meglio accorgersene prima possibile". Un documento riportante schede come quella precedente risulta essere un valido ausilio al project manager per poter effettuare la stima dei tempi e dei costi in maniera meno avventurosa.

Nella tab. 3.2 è presentata la descrizione di un requisito utente che potrebbe scaturire nel contesto della realizzazione di un sottosistema bancario dedicato alla gestione degli investimenti (trading system). In particolare, sistemi di questo tipo necessitano di ricevere da appropriate fonti esterne e memorizzare informazioni relative alle quotazioni di mercato di determinati insiemi di titoli, valute, azioni ecc. Le informazioni delle quotazioni sono necessarie sia in tempo reale, sia come resoconto di fine giornata borsistica per essere in grado di compiere tutta una serie di processi di rivalutazione, analisi rischi, previsioni, ecc. Il requisito specificato nella tabella fa riferimento essenzialmente ai messaggi di fine giornata e in particolare si esige di avere l'assoluta certezza sia in merito all'indennità del mittente, sia all'integrità del messaggio. Si provi ad immaginare i risultati che potrebbero conseguire da un attacco hacker o semplicemente da un errore di trasmissione.

Use Cases Diagram

Il sistema

Il primo passo nella realizzazione dei modelli dei casi d'uso consiste nello stabilire con precisione il confine del sistema; sebbene possa sembrare un'attività piuttosto banale, nasconde diverse insidie. Molto spesso, analizzando modelli dei casi d'uso, non si riesce a comprendere chiaramente quale ne sia il dominio: un sistema informativo, un sistema informatico, un sottosistema, un modulo e così via. Ovviamente tutto è valido a patto che si stabilisca con precisione l'oggetto di studio e si realizzi un modello coerente. Verosimilmente gli attori di un sistema informativo sono ben diversi da quelli di un componente: lo stesso livello di dettaglio dovrebbe risultare completamente differente.

In un tipico processo di sviluppo del software esistono diversi modelli dei casi d'uso e, indipendentemente da quale sia quello oggetto di studio (business, requirements, ecc.), è importante che i confini siano ben definiti. Se per esempio l'oggetto di studio è il modello business dei casi d'uso, verosimilmente l'area di interesse è parte di una struttura economica più vasta, e quindi è evidente l'importanza di definire precisamente la porzione del business che il sistema finale dovrà risolvere.

La corretta definizione dei confini del sistema riduce il rischio di commettere un errore spesso presente nei modelli dei casi d'uso: la modellazione delle entità esterne. Si tratta di un malinteso che comporta una serie di problemi:

- rende difficile l'individuazione degli attori del sistema;

- si corre il rischio di definire comportamenti non rispondenti alla realtà e/o requisiti non concretizzabili in quanto non sotto controllo;

- si spreca tempo, ecc.

Non sempre è immediato stabilire i confini del sistema; non sempre si riesce a determinare esattamente se sia più opportuno eseguire manualmente oppure automatizzare talune attività: quante volte capita di sentire un operatore esclamare che un'attività *x* prima dell'introduzione del computer veniva svolta in pochi secondi, mentre ora, con l'automazione, necessita di ore? Quante volte è colpa esclusivamente degli orizzonti limitati dell'operatore? La vocina interna direbbe quasi mai… Chissà come mai riflessioni di questo tipo sono particolarmente frequenti in istituti ben precisi.

Un'altra considerazione da farsi è quanto grande si vuole realizzare il sistema nelle sue prime versioni. Una buona idea è quella di iniziare con l'identificazione delle funzionalità ritenute indispensabili: il "core" del nuovo sistema, rimandando a sviluppi futuri le restanti. Procedendo con un simile approccio, è possibile concentrarsi su un insieme limitato di funzioni, con la speranza di riuscire a realizzare una prima versione soddisfacente del sistema. Raggiunto l'obiettivo, e convinto il cliente della qualità del lavoro prodotto, è sempre possibile aggiungere altre funzioni ritenute di secondaria importanza (altro passaggio del confine, altro fiorino).

Approcci iterativi e incrementali incontrano tipicamente il consenso del cliente: sebbene nel complesso si trovi a elargire una somma superiore, lo fa a rate e ci sono maggiori garanzie di successo del sistema.

Graficamente il sistema dovrebbe essere evidenziato per mezzo di un rettangolo che ne contenga gli use case e ne lasci all'esterno gli attori: un vero e proprio confine. Per esigenze di "rendering" grafico e lacune di taluni tool, la delimitazione del sistema viene quasi sempre omessa.

Attori

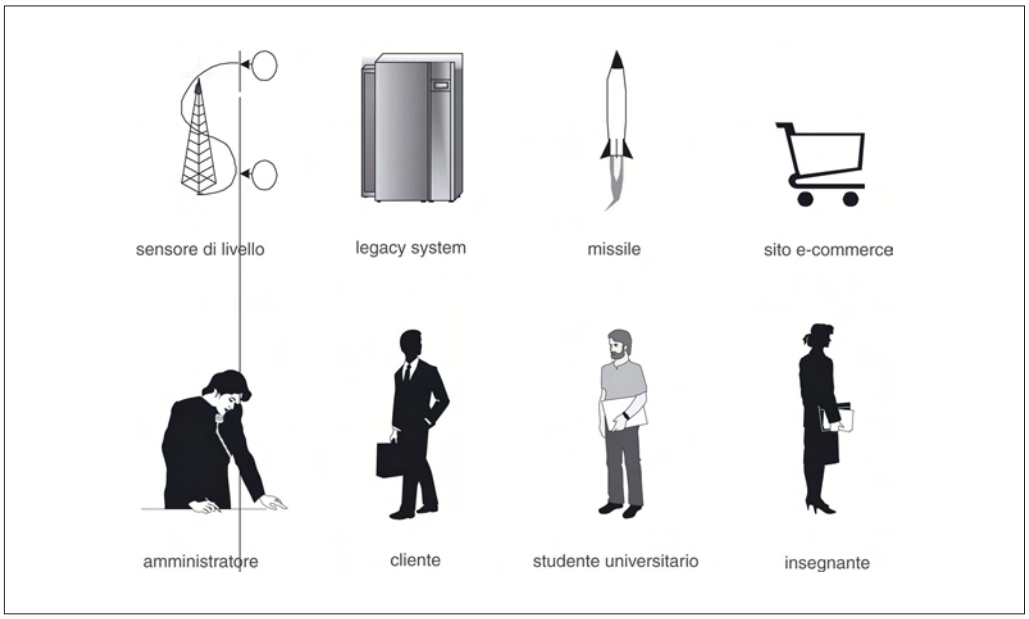
Definizione

Un attore definisce un insieme coerente di ruoli che un "utente" di un caso d'uso recita quando interagisce con esso.

Un attore non è necessariamente una persona, può essere un sistema automatizzato o un qualsiasi dispositivo fisico; in generale è un'entità esterna al sistema che interagisce con esso. Si tratta dell'idealizzazione di un persona, di un processo, o di qualsiasi cosa interagisca con il sistema inviando e/o ricevendo messaggi: scambiando informazioni.

Alcuni esempi di attori, come mostrato in fig. 3.4, possono essere un amministratore, il cliente, uno studente universitario, un insegnante, un sensore di livello, un missile, un legacy system, un sito e-commerce, ecc.

Figura 3.4 — *Esempi di attori.*



Gli attori prevedono una rappresentazione grafica standard ma, come illustrato nel Capitolo 2, nulla vieta di ricorrere a opportuni stereotipi al fine di evidenziare le caratteristiche salienti dei vari attori e le relative funzionalità.

Tipicamente è consigliato rappresentare gli attori attraverso opportuni stereotipi sia per enfatizzarne natura e ruolo (sistemi automatici, sensori, manager, impiegati) sia per rendere i diagrammi più accattivanti (sempre senza esagerare). Chiaramente, una volta stabilito un determinato stereotipo per uno specifico attore è necessario utilizzarlo consistentemente.

Un errore tipico è che spesso si rappresentano persone o descrizioni di incarichi invece che attori veri e propri. Pertanto, nell'individuare gli attori del sistema è necessario prestare attenzione al fatto che in particolari organizzazioni diversi utenti possono impersonare lo stesso attore "logico", così come uno stesso utente può recitare il ruolo di più attori.

In UML, gli attori vengono rappresentati graficamente attraverso sagome di omini stilizzati (*stick man*), detti in gergo tecnico "scarabocchi" — o più teneramente "scarabocchietti" — con il nome riportato alla base. In fig. 3.5 sono riportati alcuni esempi di attori.

Non necessariamente un attore deve essere una persona fisica: anche altri sistemi o dispositivi hardware possono assurgere al ruolo di attore. Se per esempio si dovesse modellare un sistema d'arma di artiglieria contraerei, i vari radar, la sezione lancio e, spesso, i missili stessi avrebbero tutta la dignità per assumere il ruolo di attori. Ancora, se si dovesse progettare un sistema di wrapper (layer di comunicazione) tra un sito per il commercio elettronico e il Legacy System del committente, entrambi i sistemi avrebbero tutta la dignità di essere considerati attori. Un sensore in grado di rilevare e comunicare al sistema il livello dell'acqua presente in una diga è anch'esso a tutti gli effetti un attore per il relativo sistema di monitoraggio e così via.

Un errore tipico che si commette nel realizzare il modello dei casi d'uso, e in particolare nell'individuare gli attori, è utilizzare nomi diversi per identificare lo stesso ruolo e quindi lo stesso attore. Questo problema è particolarmente frequente quando uno stesso modello dei casi sia realizzato da diversi team. Si assiste quindi al proliferare di sinonimi del tipo: "banconista", "addetto allo sportello", "sportellista", ecc. Il problema può essere risolto aggiornando la lista degli attori utilizzati con relativa descrizione, magari inserendoli tutti in uno stesso package condiviso del proprio modello. Pertanto prima di inventare un nuovo attore sarebbe sufficiente effettuare una breve verifica tra quelli individuati.

Figura 3.5 — *Esempi di attori con rappresentazione classica UML.*

Per ciò che concerne le relazioni, un attore viene connesso ai casi d'uso con i quali interagisce per mezzo di una relazione di associazione, e può prendere parte a relazioni di generalizzazione in cui una descrizione astratta di un attore più essere condivisa e specializzata da una più specifica.

Ogni attore, durante la fase di disegno, deve in qualche modo essere rappresentato all'interno del sistema con cui interagisce (come si vedrà meglio nel capitolo dedicato ai class diagram: in ultima analisi un attore è una particolare classe, la cui implementazione interna non è rilevante nel contesto dei casi d'uso).

Gli attori comunicano con il sistema inviando o ricevendo messaggi: forniscono lo "stimolo" (*trigger*) agli use case; ciò equivale a dire che ogni caso d'uso deve essere avviato da un'esplicita richiesta di un attore (eccezion fatta per i casi d'uso introdotti per migliorare la qualità del modello, come per esempio quelli raggruppanti un significativo comportamento comune presente in diversi casi d'uso).

Una tecnica utilizzata per semplificare il processo di individuazione degli attori consiste nel classificarli in primari e secondari. I **primari** sono quelli che utilizzano le funzioni proprie del sistema (per uno use case diagram sono quelli che lo avviano), e pertanto vengono anche definiti attivi; i secondi usufruiscono di informazioni: ricevono comunicazionie la loro partecipazione a un servizio è molto circoscritta, e così via. Questi sono i cosiddetti attori **secondari** o passivi. È da tener presente che attori primari per uno use case possono essere secondari per un altro e viceversa. La differenza sostanziale tra i due tipi è che, mentre gli attori primari avviano delle funzionalità proprie del sistema, i secondari ricevono dei messaggi e non forniscono un vero e proprio stimolo.

L'insieme completo degli attori presenti nei diagrammi dei casi d'uso evidenzia tutte le entità che necessitano di scambiare informazioni con il sistema stesso.

Relazione di generalizzazione tra attori

Nella realizzazione dei diagrammi dei casi d'uso può verificarsi il caso in cui più attori presentino diverse e significative similitudini, per esempio interagiscano con un insieme di casi d'uso con le stesse modalità. In questi casi, come i princìpi del paradigma OO insegnano, è possibile dar luogo a un nuovo attore, magari astratto, che raccolga a fattor comune tali similitudini, esteso dagli altri per mezzo di apposite relazioni di generalizzazione. Chiaramente non sempre è necessario introdurre un nuovo attore; anzi, la maggior parte delle volte si verifica il caso più semplice di un attore che ne estende un altro: un attore "specializzato" eredita il comportamento del "genitore" e lo estende in qualche maniera. Coerentemente con la definizione di relazione di generalizzazione (o meglio con il relativo principio di sostituibilità), un attore estendente può sempre essere utilizzato al posto di quello esteso ("genitore" o "padre").

> Ciò è abbastanza chiaro: se per esempio in un team di sviluppo è necessario allocare un ulteriore analista programmatore OO e viene un laureato in discipline tecniche proveniente

da qualche famosa azienda di consulting, che vanta la partecipazione in mille ruoli di rilievo, magari anche in riviste di prestigio, il tutto dovrebbe funzionare benissimo: oltre a capacità analitiche e programmative, richieste all'attore base, dovrebbe possedere una serie di esperienze (comportamenti e attributi aggiuntivi, particolarmente appetibili). Chissà perché poi le cose nella realtà non sono mai così lineari…

Si consideri il caso di un sistema con due attori: operatore e amministratore. L'amministratore è caratterizzato dallo svolgere tutte le funzioni dell'operatore e dall'essere gravato da un insieme di funzioni aggiuntive ritenute sensibili per il sistema. In questo caso, è immediato asserire che l'attore `Amministratore` eredita dall'attore `Operatore`.

Si consideri l'esempio di fig. 3.6. Si tratta di un sistema automatizzato di vendita utilizzato, dal lato back office, essenzialmente da due attori: `Venditore` e `Supervisore`.

Entrambi possono gestire un ordine, ma l'attore `Supervisore` estende il `Venditore`, in quanto ha la responsabilità di validare ordini che, in base a determinati fattori, sono considerati rischiosi.

La notazione grafica utilizzata in UML per rappresentare una relazione di generalizzazione è quella standard e prevede un segmento che unisce l'attore estendente a quello esteso; in prossimità di quest'ultimo viene riportato un triangolo vuoto a mo' di freccia.

Figura 3.6 — *Esempio di relazione di generalizzazione tra attori.*

Figura 3.7 — *Esempio di mancata astrazione nell'identificazione degli attori.*

Molto spesso analizzando modelli use case è possibile notare una rete densissima di collegamenti tra attori e casi d'uso. Per esempio può capitare di osservare che una serie di use case siano tutti connessi con un insieme ben definito di attori (fig. 3.7)

Fitte reti di collegamento attori/casi d'uso possono costituire sintomi del fatto che non si sia proceduto ad una appropriata organizzazione gerarchica degli attori. Probabilmente è possibile ristrutturare i vari attori secondo un'opportuna relazione gerarchica. Ciò è ottenibile definendo un nuovo attore "padre" connesso con tutti i casi d'uso comuni agli altri e quindi specializzarlo con i vari attori, ognuno dei quali eventualmente connesso ad altri casi d'uso di più specifica pertinenza (fig. 3.8).

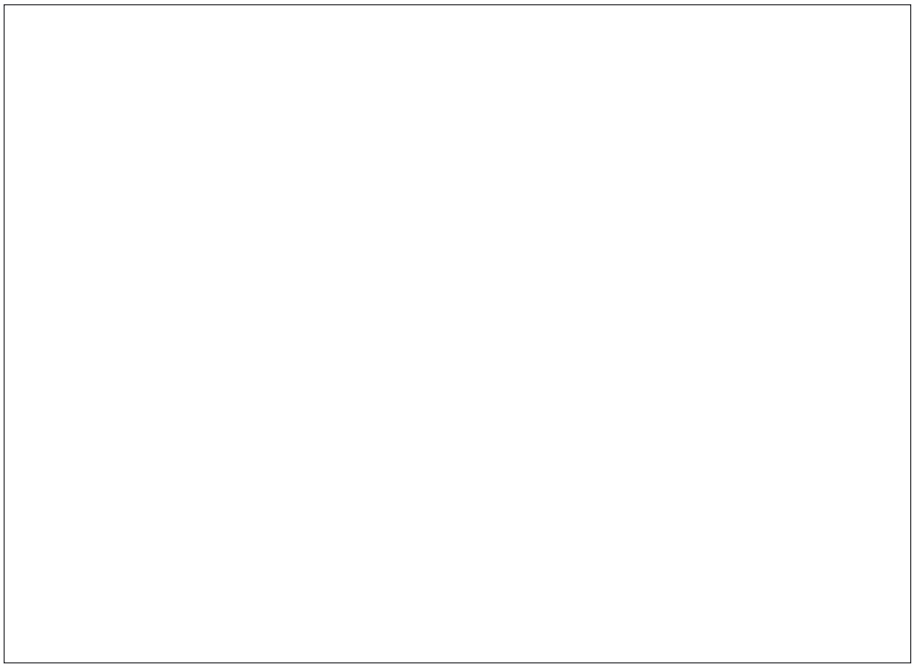
Il diagramma mostrato in fig 3.8 rievoca nella mente dell'autore un "simpatico" aneddoto. Realizzata la primissima versione del modello dei casi d'uso di un sistema piuttosto complesso, bisognava definire chiaramente ruoli e responsabilità dei relativi attori umani. Trattandosi

di un sistema completamente nuovo, l'attività risultava abbastanza importante: si trattava di definire il profilo delle figure professionali da reclutare oltre ai manager già assegnati per l'entrata in esercizio del sistema. La prima versione dell'organizzazione dei ruoli era simile a quella presentata in fig. 3.9.

La logica alla base della struttura di figura era di astrarre ruoli con comportamento comune ossia raggruppare attori che interagivano con un insieme condiviso di casi d'uso. Per esempio, era importante disporre di un attore utente generico (`Utente`) per descrivere le funzioni che tutti gli attori utilizzavano (login, cambiamento di password, richiesta nuova password, ecc.). All'interno di ogni dipartimento era poi importante mostrare le differenze tra gli addetti e i manager in quanto questi, oltre a poter espletare tutte le mansioni dei propri subalterni (ereditano dagli addetti del reparto), sono gravati da responsabilità supplementari tipiche del ruolo manageriale (ereditano dall'attore `Direttore Dipartimento`). Per esempio avviano il servizio di richiesta del profilo utente per i propri dipendenti, hanno l'obbligo di validarne i *time sheet*, ecc.

Una volta sottoposto il diagramma all'attenzione dei "clienti" (manager), successe la rivoluzione... "Orrore!!! Come? I manager nell'organigramma compaiono a un livello

Figura 3.8 — *Diagramma ristrutturato evidenziando la relazione di generalizzazione tra attori.*



inferiore rispetto a quello dei propri subalterni!?". A questo punto il pasticcio era fatto. Tutte le spiegazioni tecniche non sortivano alcun effetto… Cosa fare? Prima idea: ribaltare il diagramma… Per questa soluzione era troppo tardi e comunque rimaneva il concetto poco gradito della doppia piramide. La soluzione fu la realizzazione di un diagramma illeggibile… Si decise di portare tutti i manager su una riga superiore come mostrato in fig. 3.10.

Figura 3.9 — *Frammento di un'organizzazione gerarchica degli attori umani di un sistema.*

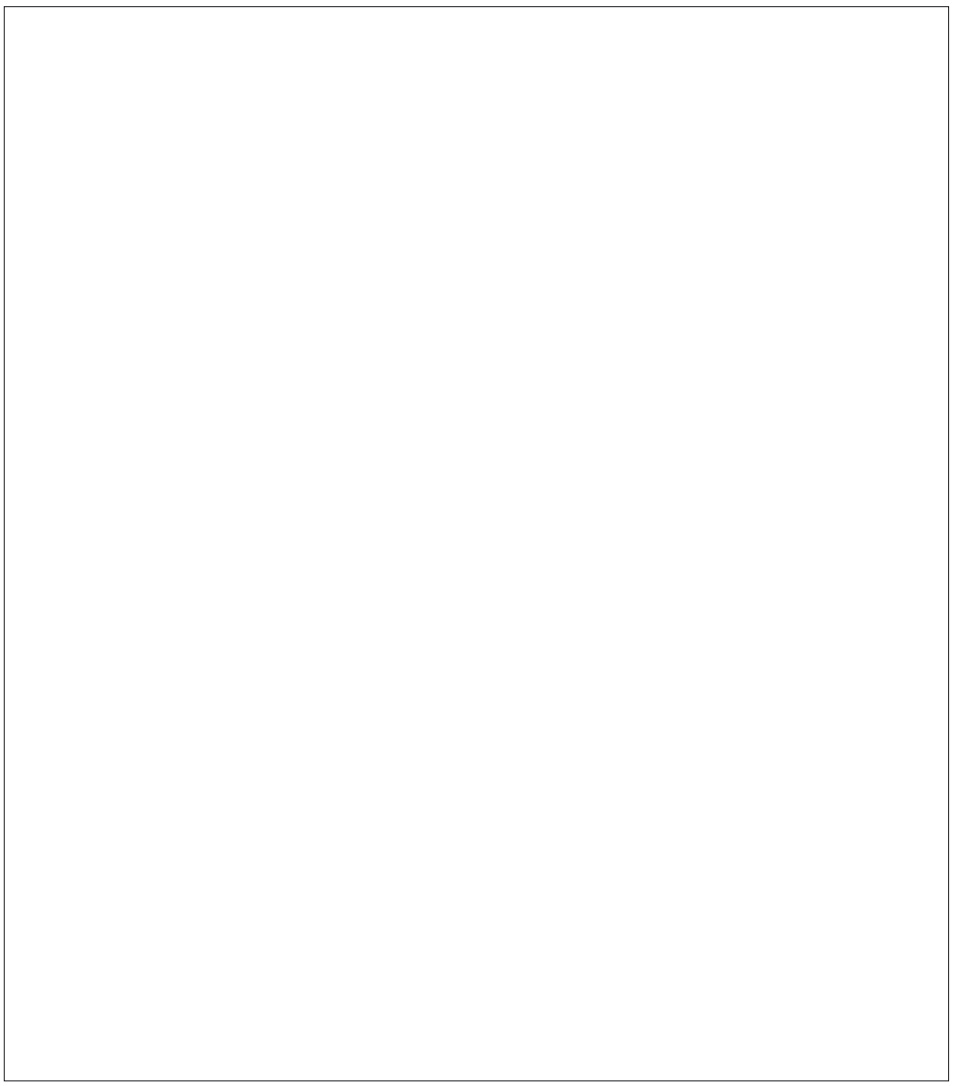
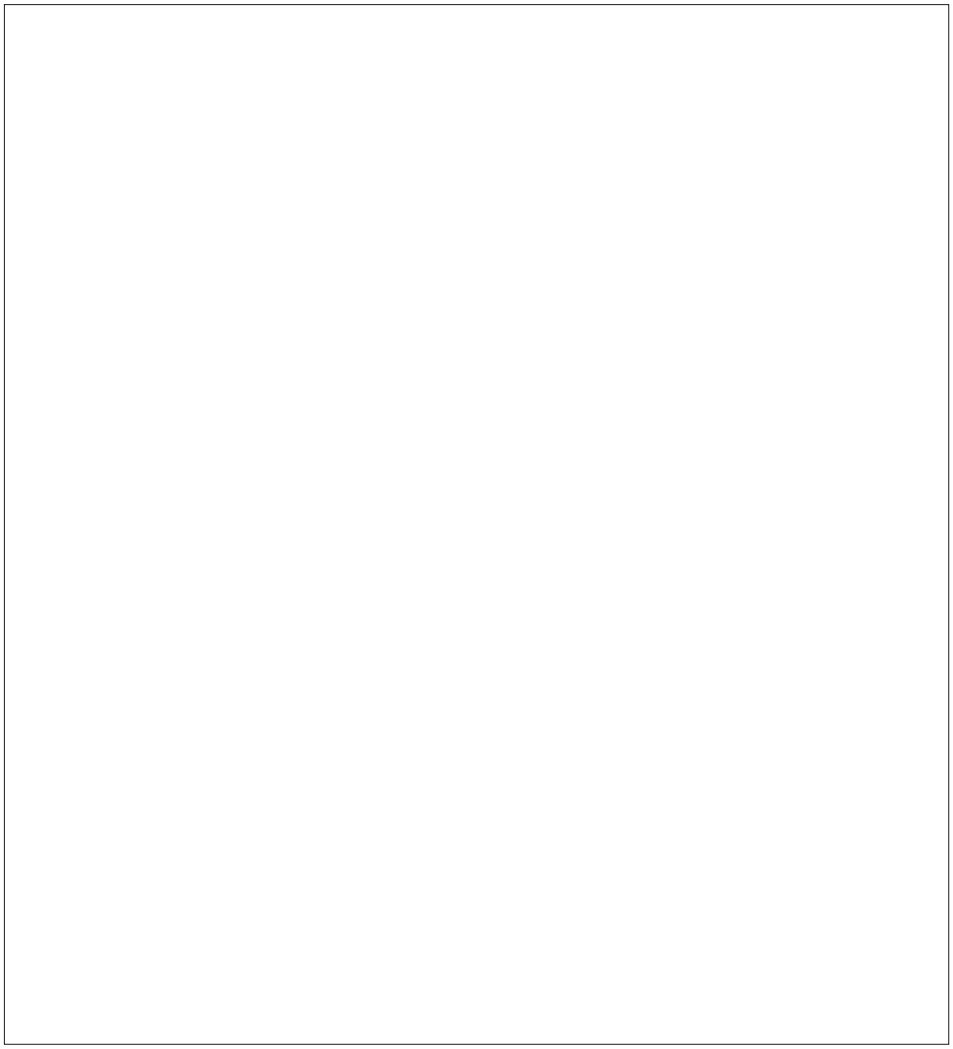


Figura 3.10 — *Modifiche a un frammento dell'organizzazione gerarchica degli attori presentata nel diagramma precedente.*



Relazione di associazione tra attori e casi d'uso

Gli attori sono entità esterne al sistema che interagiscono con esso. Le interazioni si materializzano attraverso scambi di messaggi. Pertanto ogni attore è connesso con un insieme di casi d'uso (funzioni del sistema) che ne ricevono gli stimoli e che producono i dati richiesti dall'attore stesso (disporre di attori non associati ad alcun caso d'uso

Figura 3.11 — *Esempio di relazioni di associazione.*

dovrebbe invitare a riflettere sulla loro necessità). Il legame tra attore e use case è realizzato per mezzo di una relazione di associazione. La situazione più tipica è che un singolo attore è associato a molti casi d'uso. Graficamente le relazioni di associazione sono visualizzate per mezzo di un segmento che unisce l'attore con il relativo caso d'uso.

Nella fig. 3.11 viene riportato un semplice diagramma dei casi d'uso rappresentante una porzione di un sistema automatizzato di vendita online. Per ora si attribuisca alla relazione `include` (inclusione) un significato del tutto intuitivo: l'esecuzione della funzione `Effettua ordine` necessita dell'esecuzione delle funzioni `Reperimento dati cliente` e `Reperimento articoli`.

Dunque, un cliente, previa opportuna identificazione, può compilare degli ordini selezionando gli articoli di interesse. L'attore *Venditore* ha il compito di visionare gli ordini e l'esito del controllo viene comunicato al cliente che ha emesso l'ordine oggetto di verifica.

Navigabilità sì, navigabilità no

Come si vedrà nella descrizione dei diagrammi delle classi, la navigabilità è un vincolo attribuibile alle relazioni di associazione ed è visualizzato graficamente per mezzo di una freccia (fig. 3.14). Qualora in una relazione di associazione non si voglia permettere agli oggetti di un tipo di "vedere" quelli dell'altra, nella rappresentazione della relazione è necessario inserire una freccia indicante il verso di percorrenza. Pertanto gli oggetti posti nella coda dell'associazione (`navigabilità = false`) possono "navigare" negli oggetti puntati dalla freccia (`navigabilità = true`), mentre non è possibile il contrario.

Nei modelli dei casi d'uso non sempre è opportuno visualizzare il vincolo di navigabilità nelle relazioni di associazione tra attori e casi d'uso. Il problema è che tali vincoli sono interpretati come limitazioni alla direzione del flusso delle informazioni. Nel 99% dei casi il flusso di dati tra un determinato attore ed uno specifico caso d'uso non è esclusivamente monodirezionale; normalmente si tratta di un'interazione prolungata nel tempo (il sistema richiede informazioni, l'attore le inserisce e quindi le sottopone al sistema, quest'ultimo elabora i dati, esegue specifici servizi e quindi presenta altri dati all'attore, e così via). Nel diagramma di fig. 3.11, per esempio, si sarebbe potuta inserire la direzione nell'associazione tra l'attore `Cliente` allo use case `Effettua ordine` con il verso dall'attore al caso d'uso. Ciò avrebbe agevolato la lettura del diagramma ma, allo stesso tempo, avrebbe corso il rischio di creare confusione. Il vincolo di navigabilità pertanto, può essere utile per mostrare l'attore principale di un caso d'uso (chi fornisce lo stimolo iniziarle), però è opportuno valutarne bene l'utilizzo evitandolo nei casi in cui possa ingenerare confusione tra i fruitori del modello.

Come identificare gli attori del sistema

Identificare gli attori del sistema è un'attività molto importante: si identificano le entità interessate a usare e interagire con il sistema. Tipicamente si tratta di un'attività non smisuratamente complessa che l'esperienza tende a rendere ancora meno complicata. Tuttavia esistono casi particolari in cui anche il processo di individuazione degli attori può generare più di qualche incertezza.

Nel caso in cui l'attribuzione del nome possa causare qualche problema, ciò *potrebbe* essere un buon campanello di allarme che qualcosa non funziona correttamente: si sono raggruppati più attori in uno solo (magari perché nella realtà di interesse diversi ruoli sono fisicamente recitati da una sola persona) o, problema diametralmente opposto, lo stesso ruolo è diviso in più attori (caratteristica peculiare di particolari istituti).

L'errore tipico che si commette nell'individuare gli utenti del sistema è che si prendono in considerazione unicamente gli operatori umani: i famosi signori con i manicotti seduti di fronte ai monitor. Spesso ci si dimentica che un attore è qualsiasi entità che ha interesse a interagire con il sistema: anche dispositivi fisici.

Un altro errore ricorrente è quello di considerare il sistema da subito e unicamente con una prospettiva molto tecnica e non con una più vicina al business del cliente. In altre parole, spesso ci si dimentica dei veri utenti del sistema i quali, molto frequentemente, sono i clienti del committente stesso.

Premesso ciò, rispondere alle domande riportate di seguito può aiutare ad identificare/verificare gli attori del sistema:

- Chi sono gli utilizzatori delle principali funzioni del sistema? Ossia, chi sono gli attori principali?
- Chi usufruirà giornalmente del supporto del sistema?
- Chi si occuperà di mantenere e amministrare il sistema?
- Esistono sistemi esterni (sia software, sia hardware) che necessitano di interagire con il sistema? Quali di essi iniziano il colloquio con il sistema e quali invece ricevono i risultati dell'esecuzione di opportuni processi?
- Esiste qualche ulteriore entità che ha interesse a essere informata sui risultati prodotti da opportune elaborazioni del sistema?

Modello per la documentazione degli attori

Spesso è conveniente, già durante i colloqui iniziali con gli utenti del sistema, cominciare a redigere una lista riportante gli aspiranti attori. Tale elenco è destinato a essere riesaminato e corretto man mano che si procede con le varie interviste e quindi, più in generale, con la comprensione dell'area di business che il sistema, in qualche percentuale, intende risolvere.

Una volta terminata la serie delle interviste è necessario eseguire una fase di verifica della lista degli aspiranti attori al fine di individuarne eventuali "ridondanti", "ambigui", non ben definiti, mancanti e così via.

Tabella 3.3 — *Scheda per la definizione degli attori del sistema.*

Nome dell'attore	**Versione:**	**Data:**
Responsabilità **1.** **2.**		
USE CASE CON CUI INTERAGISCE		
Nome use case	**Primario**	**Frequenza**
	☐	
	☐	
	☐	

Effettuata anche questa verifica la lista costituisce un ottimo prodotto di input per la costruzione dei vari diagrammi dei casi d'uso. Ciò, tra l'altro, dovrebbe favorire la realizzazione di un glossario preciso circa gli attori del sistema.

Al fine di supportare il processo di costruzione del modello dei casi d'uso, si ritiene utile, per ogni attore, compilare una scheda come quella riportata in tab. 3.3.

Molto importante è definire le **responsabilità** di un attore anche se tipicamente si tende a specificarne le **operazioni**. Queste permettono di chiarire le regole del business vigenti nel sistema oggetto di studio. Talune volte, invece di specificare i casi d'uso in cui un attore è coinvolto, si preferisce descriverne le aspettative. Ciò risulta particolarmente utile quando ci si trova in una delle primissime fasi del ciclo di vita, quando i modelli dei casi d'uso non sono ancora definiti.

Per ciò che concerne la frequenza dell'utilizzo dei casi d'uso, si tratta di un'informazione utilissima in molti contesti, quali ottimizzazione delle operazioni, pianificazione dei test, ecc.

I casi d'uso

Definizione

Uno use case (caso d'uso) rappresenta una funzionalità completa così come viene percepita da un attore. Si tratta di un costrutto utilizzato per definire il comportamento di un sistema o di un'altra entità semantica senza rivelarne la struttura interna.

In termini più formali, un caso d'uso è un tipo di `Classificatore` (nel metamodello UML `UseCase` eredita da `Classifier`) che rappresenta un'unità coerente di funzionalità fornita da una specifica entità (sistema, sottosistema, classe) e descritta sia dalla serie di messaggi scambiati tra l'entità stessa e un'altra interagente esterna (attore), sia dalla sequenza di azioni svolte. Ogni caso d'uso è pertanto definito da una sequenza di azioni (comportamento dinamico) necessarie per erogare un servizio che l'entità esegue interagendo con il relativo attore. Tale sequenza può prevedere delle varianti come sequenze alternative, comportamento eccezionale, gestione degli errori, e così via.

Gli use case specificano servizi che un'entità fornisce ai relativi utilizzatori: ognuno di questi servizi deve essere completo e avviato da un attore. Le implicazioni della proprietà di completezza sono essenzialmente due:

1. l'entità, dopo aver erogato il relativo servizio, deve transitare in uno stato tale che il servizio possa essere fruito nuovamente;
2. più casi d'uso che specificano la stessa entità non possono comunicare tra loro, perché ciascuno di essi deve, individualmente, descrivere completamente l'utilizzo dell'entità stessa.

I casi d'uso possono essere raggruppati in gruppi (*package*) per questioni di comodità e facilità di reperimento.

Sebbene gli use case siano impiegati quasi esclusivamente per specificare i requisiti esterni di un'entità, essi possono anche essere utilizzati per documentare le funzionalità offerte da un'altra entità (esistente) del sistema, anche se per tali fini sarebbe preferibile utilizzare altri diagrammi messi a disposizione dallo UML (diagrammi di sequenza, collaborazione, attività e di stato).

Indirettamente ciascun caso d'uso stabilisce i requisiti che ogni entità esige dai relativi utilizzatori, come per esempio le modalità con cui essi devono interagire con l'entità stessa per fruire dei relativi servizi offerti. Nel caso che l'entità sia il sistema o un sottosistema, i relativi utenti sono gli attori; se invece si fa riferimento a sottosistemi e classi interne al sistema, tipicamente, gli "utenti" non sono definiti.

La notazione grafica dello UML prevede di rappresentare gli use case attraverso ellissi con all'interno specificato il nome — eventualmente è possibile riportarli di sotto — e che possono essere connessi ad altri use case o agli attori. In quest'ultimo caso la "connessione" prende il nome di "associazione" o "comunicazione di associazione".

Nell'individuazione degli use case, l'esperienza insegna che molto importante è la selezione del relativo nome. Si tratta dell'elemento più discusso dagli utenti/clienti. I nomi dovrebbero essere unici, sintetici e al tempo stesso capaci di definire completamente il relativo concetto (a tal fine è molto importante utilizzare verbi con significato preciso, piuttosto che generici, come *fare*, *dare*, ecc.). Ciò è particolarmente utile per i modelli a livello di dominio in cui il nome da solo dovrebbe essere in grado di giustificare il caso d'uso dal punto di vista del business. Oltre a facilitarne la lettura, nomi di questo tipo assicurano che il relativo use case rappresenti un reale requisito e non uno supposto o un'incomprensione.

Probabilmente la sintassi ideale prevede una breve frase formata da un sostantivo indicante l'azione e da un sostantivo indicante l'oggetto della stessa (*Inserimento ordine*, *Invio conferma*, *Annullamento ordine*, *Inserimento utente*, *Variazione profilo*, *Selezione domicilio*, ecc.).

Poiché un modello dei casi d'uso, in ultima analisi, rappresenta l'insieme dei servizi forniti agli attori del sistema, nella selezione dei nomi dei casi d'uso è opportuno mettersi nell'ottica degli attori.

Relazione di associazione

Si faccia riferimento al paragrafo *Relazione di associazione tra attori e casi d'uso*.

Relazione di generalizzazione

Nel disegnare i diagrammi dei casi d'uso si verifica spesso che diversi use case presentino delle somiglianze e condividano del comportamento (analogamente a quanto illustrato per gli attori). Come in precedenza, gli insegnamenti dell'OO prescrivono di rag-

Figura 3.12 — *Esempio di relazione di generalizzazione tra use case: Upload ordine.*

gruppare il comportamento comune in un apposito use case genitore (eventualmente astratto) e di estenderlo per mezzo di altri casi d'uso figli che lo specializzino per gli usi originariamente individuati. La proprietà a cui si fa riferimento è ovviamente l'eredità e la relazione dello UML è la *generalizzazione*.

Nel contesto dei casi d'uso, la generalizzazione (*Generalization*) è una relazione tassonomica tra uno use case, figlio, e un altro, chiamato genitore, che descrive le caratteristiche che il caso d'uso divide con altri use case che hanno lo stesso genitore. Un caso d'uso genitore può essere specializzato in uno o più figli che ne rappresentano forme più specifiche. Lo Use Case figlio eredita gli attributi, le operazioni, il comportamento del genitore, e pertanto può sovrascrivere (*override*) o aggiungere comportamento (attributi, operazioni, e così via).

Come da standard UML, anche in questo contesto la notazione grafica utilizzata per la relazione di generalizzazione prevede una linea continua, terminata con un triangolo vuoto posto a mo' di freccia in prossimità dello use case genitore. Da notare che la presenza di una relazione di generalizzazione implica che l'esecuzione di use case "fratelli" può avvenire unicamente in alternativa.

Si consideri l'esempio di generalizzazione illustrato nella fig. 3.12. Si tratta di uno use case che descrive la funzione che supporta l'invio (*upload*) automatico ordini, tramite browser Internet, ad una determinata organizzazione commerciale. Si immagini un sito

per il commercio elettronico il quale, tra le varie funzionalità, permetta di ricevere ordini preparati dagli utenti secondo opportuni formati (testo ASCII, XML, ecc. ecc.). Questa funzionalità di upload potrebbe funzionare in modo del tutto equivalente al modo con cui i server di posta elettronica, fruibili via comune browser, consentono di allegare file (attachment) in una e-mail: si seleziona localmente il percorso del file da allegare e quindi si preme il tasto di upload.

Si supponga che il sistema preveda due tipologie di ordini: normali e prioritari. La peculiarità dei secondi è di richiedere una gestione immediata: appena caricati vengono spediti al sistema di back office (*Legacy System*) che provvede a gestirli immediatamente, mentre i primi prevedono un iter rallentato dovuto sia a vincoli architetturali presenti nel sottosistema Internet sia in quello di back office operante presso l'organizzazione del cliente.

Il diagramma in figura mostra il caso d'uso di *Upload ordine*. In particolare la presenza delle relazioni di generalizzazione implicano che è possibile eseguire uno solo di tali

Figura 3.13 — *Flusso degli eventi di due casi d'uso uniti dalla relazione di generalizzazione.*

casi: l'esecuzione di un "fratello" esclude quella dei rimanenti. Quindi o si esegue l'upload di un ordine standard oppure si esegue quello di un ordine prioritario.

Il diagramma in figura indica unicamente la proiezione statica della funzionalità: specifica che il caricamento di un ordine richiede la validazione dell'utente e la verifica dell'ordine stesso; evidenzia che l'utente può richiedere due tipologie diverse di servizio: standard e prioritario. Nel secondo caso è previsto l'immediato invio al sistema di legacy.

Il diagramma evidentemente non specifica nulla circa la dinamica dell'operazione, non specifica altresì precondizioni e postcondizioni, non fornisce alcuna direttiva su come trattare i casi anomali: cosa fare se il formato dell'ordine risulta errato o se il prezzo di un prodotto presente nell'ordine non corrisponde a quello impostato nel sistema? E si potrebbe continuare.

Per sopperire a tale lacuna è possibile specificare il flusso delle attività illustrato in fig. 3.13. Dall'analisi del diagramma è possibile notare che il caso d'uso base A può prevedere dei punti astratti combinati a generici passi di esecuzione. Uno use case specializzante può sia definire il comportamento per le sezioni astratte (utilizzo canonico), sia sovrascrivere quello sancito in passi generici, sia aggiungere altro comportamento dopo la fine del flusso di eventi del caso d'uso base.

Per esempio, con riferimento allo use case di fig. 3.12, `Upload ordine`, un possibile flusso di eventi potrebbe essere quello riportato di seguito.

Use case base Validazione utente:

1. Il sistema visualizza il modulo richiesta credenziali utente
2. L'utente imposta la propria login e password
3. Il sistema verifica i dati impostati
3.1. **se** selezionata l'opzione termina **allora**
3.1.1. termina il caso d'uso con errore
4. Il sistema procede con la verifica della coppia (login, password)
4.1. **se** utente autenticato **allora**
4.1.1. il sistema mostra messaggio di avvenuta autenticazione
4.1.2. termina il caso d'uso con successo
4.2. **se** autenticazione fallita per tre volte consecutive **allora**
4.2.1. il sistema mostra messaggio di fallita autenticazione.
4.2.2. il sistema esegue il lock dell'utente.
4.2.3. il sistema registra l'evento nel file di log.
4.2.4. termina lo use case con errore.
4.3. **altrimenti**
4.3.1. il sistema memorizza il numero di tentaivi di log-in falliti.
4.3.2. il sistema mostra messaggio di fallita autenticazione.

```
   4.3.3. il sistema registra l'evento nel file di log.
   4.3.4. torna al punto 1.

Use case base Upload ordine:
   1. include (Validazione utente)
   2. se riconoscimento utente terminato con errore allora
   2.1. termina il caso d'uso con insuccesso
   3. Il sistema visualizza il modulo per il caricamento del file.
   4. L'utente imposta il percorso del file da caricare.
   5. Il sistema valuta i dati impostati dall'utente.
   5.1. se selezionata l'opzione termina allora
   5.1.1. termina il caso d'uso con insuccesso
   6. Il sistema esegue il caricamento del file.
   7. se verificato errore nel caricamento allora
   7.1. Il sistema mostra opportuno messaggio.
   7.2. Il sistema registra l'evento nel file di log.
   7.3. torna al punto 3.
   8. Il sistema memorizza temporaneamente il file (ordini da verificare)
   9. include (Verifica ordine)
   10.  se verifica fallita allora
   10.1. Il sistema mostra opportuno messaggio.
   10.2. Il sistema elimina l'ordine.
   10.3. Il sistema registra l'evento nel file di log.
   10.4. torna al punto 3.
   11.  Il sistema memorizza permanentemente il file (ordine verificato)

Use case Upload ordine standard:
   11.  Il sistema memorizza permanentemente il file
        nella directory ordini standard.

Use case Upload ordine prioritario:
   11.  Il sistema memorizza permanentemente il file
        nella directory ordini prioritari.
   12.  include (Invio ordine prioritario).
```

Si può immaginare il sistema strutturato in modo tale da prevedere un servizio cadenzato che si occupi, a intervalli prestabiliti di tempo (una o due volte al giorno) di prelevare gli ordini standard per inviarli al legacy system (sistema automatizzato di back office presente presso l'organizzazione del cliente: la struttura che in ultima analisi fornisce i prodotti che vengono venduti tramite il sito di e-commerce).

Lo use case di una funzionalità così concepita offre un interessante motivo di riflessione: l'avvio dello use case avverrebbe automaticamente (a istanti temporali predefiniti) senza lo stimolo di un vero attore esterno. A questo punto nasce un dilemma: il tempo può essere considerato un attore o no? Probabilmente si tratta più di uno sceneggiatore o di un regista.
Applicando rigorosamente le direttive dello UML, il tempo non dovrebbe poter essere considerato un attore; come punto a favore si può invece asserire che si tratta di un'entità indubbiamente esterna al sistema e altrettanto sicuramente non controllabile dallo stesso. In altre parole l'attore Tempo non solo è un'assunzione decisamente accettabile ma contribuisce anche a rendere i diagrammi più leggibili… In ultima analisi gli stessi non dovrebbero venir sempre avviati da un attore?

Nella fig. 3.14 è stato riportato l'esempio del tempo come attore evidenziandolo per mezzo di apposito stereotipo al fine di far risaltare ulteriormente il concetto dell'avvio temporizzato. Per ciò che concerne invece gli ordini prioritari, la specifica del comportamento dinamico ne prevede l'immediato invio al legacy system.

Come si vedrà successivamente, probabilmente il formalismo utilizzato per specificare il comportamento dinamico dei casi d'uso (così come proposto dai Tres Amigos) nella pratica non risulta essere il più efficace: forse sarebbe più opportuno utilizzare appositi template scegliendoli tra quelli proposti.

Come si può notare gli use case figli ereditano la sequenza comportamentale del genitore e vi aggiungono comportamento specifico. Potenzialmente sia il caso d'uso genitore, a meno che non preveda opportune sezioni astratte, sia quello figlio sono istanziabili.

Differenti specializzazioni dello stesso genitore sono completamente indipendenti, contrariamente a quanto avviene con le relazioni di estensione (illustrato nell'apposito paragrafo) in cui i diversi use case estendenti modificano il comportamento dello stesso use case e possono essere invocati in sequenza.

Figura 3.14 — *Invio ordini standard.*

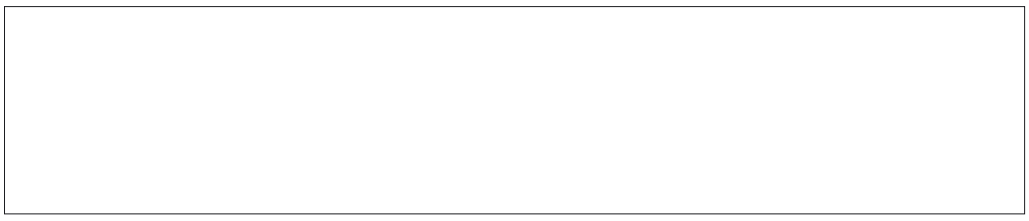


Figura 3.15 — *Flusso degli esempi della relazione di inclusione.*

Il comportamento specifico di un caso d'uso figlio può essere indicato inserendo opportune sezioni o sovrascrivendone altre, in modo del tutto analogo a quanto avviene con i metodi delle classi, nella sequenza di azioni ereditate dallo use case genitore. In questo contesto è consigliabile ricorrere con parsimonia al meccanismo di sovrascrittura onde non stravolgere gli intenti dello use case genitore.

Quando il caso d'uso genitore è astratto, nella relativa sequenza di azioni sono previste apposite sezioni lasciate volutamente incomplete e alle quali lo use case ereditante deve provvedere.

La proprietà di sostituibilità propria della relazione di generalizzazione applicata ai casi d'uso implica che la sequenza di azioni dello use case figlio deve includere quella del genitore.

Relazione di inclusione

L'inclusione è una relazione tra uno use case base e uno incluso che specifica come il comportamento di quest'ultimo possa essere inserito in quello definito nello use case base, il quale, può avere visione dell'incluso e, tipicamente, dipende dagli effetti da esso generati durante la relativa esecuzione.

In termini della sequenza delle azioni, che ciascun caso d'uso esegue per erogare un servizio, la relazione `include` comporta l'esecuzione della sequenza di azioni dello use case incluso sotto il controllo e nella locazione specificata dello use case base, come mostrato nella fig. 3.15.

Il concetto di inclusione è per molti versi analogo a quello di invocazione di funzione: viene eseguita la sequenza di azioni dello use case base; quando viene raggiunto un punto di inclusione, il controllo viene passato al caso d'uso ivi indicato (incluso) e ne viene eseguita completamente la sequenza delle azioni, quindi il controllo ritorna allo use case base.

> Volendo essere più precisi bisognerebbe paragonare la relazione di inclusione al meccanismo delle macro presente in molti linguaggi di programmazione: si definisce una macro e la si copia (si espande il "codice") in tutti i punti in cui ci si riferisce alla macro stessa. Lo use case incluso può avere accesso agli attributi ed alle operazioni di quello base.

Figura 3.16 — *Esempio di relazione di inclusione: use case per la compilazione di un ordine.*

> La relazione di inclusione ha avuto una genesi piuttosto complicata; inizialmente si trattava di uno stereotipo della relazione di generalizzazione e, pertanto, veniva rappresentata per mezzo dello stesso simbolo grafico con a fianco l'etichetta `<<uses>>(nome dello stereotipo)`. Nella versione 1.3 dello UML, per fortuna, le cose sono un po' cambiate e l'inclusione è stata più opportunamente convertita in una relazione a sé stante. La notazione grafica prevede una freccia tratteggiata con l'etichetta `<<include>>`. Probabilmente era poco opportuno indicare una relazione di uso o di inclusione attraverso uno stereotipo della relazione di generalizzazione in quanto quest'ultima implica la proprietà di sostituibilità; nel contesto della relazione di inclusione indicava che lo use case "utilizzatore" poteva sempre essere sostituito al posto di quello utilizzato, il che evidentemente è lungi da rispondere a verità.

Una relazione di inclusione indica che lo use case base (l'utilizzatore) incorpora esplicitamente il comportamento di un altro use case (l'utilizzato), il quale, tipicamente, non vive di vita propria, ma deve necessariamente essere istanziato come parte di un altro caso d'uso. Lo use case "utilizzato" ha visione completa del suo utilizzatore con il quale può scambiare parametri e comunicare risultati.

La relazione di inclusione è molto utile in quanto evita di dover ripetere più volte uno stesso flusso di sequenza. In particolare, per specificare nel flusso dello use case utilizzatore il punto in cui inserire quello dello use case incluso, è sufficiente premettere l'identificatore `include` seguito dal nome dello use case.

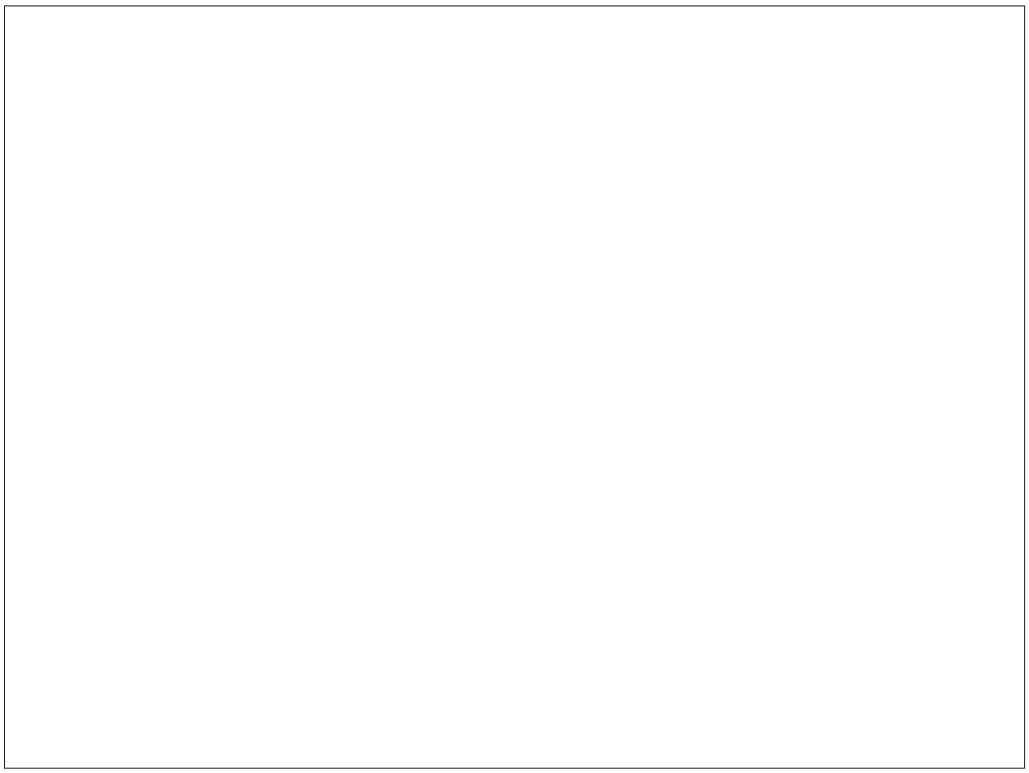
Chiaramente, poiché il caso d'uso incluso è a tutti gli effetti uno use case, può essere associato ad altri per mezzo di proprie relazioni di inclusione, estensione e così via. Si consideri l'esempio riportato in fig. 3.16. Si tratta di uno use case per la compilazione ordini. In particolare vi è l'attore `Cliente` che inserisce gli ordini nel sistema. Come si può evidenziare, la funzionalità `Compila ordine` prevede il `Reperimento dei dati del cliente` e la ricerca degli articoli (`Reperimento articolo`) che andranno a costituire l'ordine stesso.

Nel redigere i diagrammi dei casi d'uso è necessario porre attenzione a non essere eccessivamente prescrittivi: ciò non sarebbe di alcun aiuto e, paradossalmente, finirebbe con il rendere il diagramma inutilmente complicato e con l'obbligare eccessivamente il team di sviluppo durante le fasi successive a rispettarne i dettagli; si ricordi che la use case view è anche un documento utilizzato nei test di accettazione.

Per esempio nello use case di fig. 3.16 si sarebbero potute evidenziare ulteriori funzionalità (fig. 3.17) come `Verifica disponibilità articolo` o `Reperimento prezzo cliente`, generalizzazioni della funzionalità di ricerca articolo: `Ricerca articolo per codice`, `Ricerca articolo per descrizione`, `Ricerca articolo per categoria`, ecc. Ciò sarebbe stato del tutto ingiustificato e avrebbe complicato uno use case intrinsecamente semplice.

L'inconveniente può non sembrare così serio nel diagramma di figura, ma si tenga presente che, in una situazione reale, esso sarebbe parte di un caso d'uso decisamente più complesso; è consigliabile, quindi, non perdere tempo nel dettagliare eccessivamente gli use case diagram. Si corre unicamente il rischio di renderli inutilmente complessi: come si suol dire "ogni cosa a suo tempo". Un altro problema in cui si rischia di incorrere è quello di dar luogo ad una decomposizione funzionale del sistema. Ciò è fortemente sconsigliato poiché si comincia a disegnare il sistema in fasi premature in cui si hanno pochi dettagli relativi al sistema e con una notazione non opportuna. La situazione di modelli di casi d'uso generati attraverso un processo di decomposizione funzionale è ravvisabile in tutti quei casi in cui è presente un caso d'uso A che ne include uno B che ne include uno C ecc. Già tre livelli di inclusione potrebbero essere sospetti. Assumendo che l'utente sia stato precedentemente autenticato dal sistema e autorizzato a eseguire la funzione, il flusso di azione del diagramma (fig. 3.16) potrebbe assumere una forma del tipo:

Figura 3.17 — *Come complicare un "affare" semplice.*



Use case base Compila Ordine:

1. **include** (Validazione utente)
2. **se** riconoscimento utente terminato con errore **allora**
2.1. termina il caso d'uso con insuccesso
3. Il sistema reperisce i dati del cliente
4. Il sistema imposta l'intestazione dell'ordine
5. Il sistema visualizza il modulo reperimento articolo
6. L'utente imposta i dati
7. Il sistema valuta i dati impostati dall'utente
7.1. **se** selezionata l'opzione termina **allora**
7.1.1. termina il caso d'uso con insuccesso
8. **include** (Ricerca articolo)
9. **se** reperimento fallito **allora**
9.1. mostra messaggio

```
  9.2.     registra l'evento nel file di log.
  9.3.     torna al punto 5.
  10. L'utente seleziona l'operazione da eseguire sull'ordine
  10.1. se operazione = annulla ordine allora
  10.1.1. mostra messaggio
  10.1.2. registra l'evento nel file di log.
  10.1.3. termina il caso d'uso con insuccesso
  10.2. se operazione = aggiorna quantità articolo allora
  10.2.1. l'utente imposta la nuova quantità dell'articolo desiderato.
  10.2.2. il sistema ricalcola i totali.
  10.2.3. il sistema aggiorna l'ordine.
  10.2.4. torna al punto 10.
  10.3. se operazione = elimina riga allora
  10.3.1. l'utente seleziona la riga da eliminare.
  10.3.2. il sistema ricalcola i totali.
  10.3.3. il sistema aggiorna l'ordine.
  10.3.4. torna al punto 10.
  10.4. se operazione = aggiunta ulteriore articolo allora
  10.4.1. torna al punto 5.
  10.5. se operazione = conferma ordine allora
  10.5.1. Memorizza ordine
  11. Fine use case

Use case Reperimento articolo:
  1. il sistema acquisisce il codice articolo.
  2. il sistema tenta di reperire i dati dell' articolo per codice
  3. se reperimento fallito allora
  3.1.  il sistema acquisisce la descrizione articolo
  3.2.  il sistema tenta di reperire i dati articolo per descrizione
  4. se reperimento fallito allora
  4.1.  termina con insuccesso
  5. il sistema tenta di reperire il prezzo articolo
     da applicare al cliente
  6. se reperimento fallito allora
  6.1.  termina con insuccesso
  7. il sistema riporta dati reperiti
```

Volendo si sarebbe potuto rendere i flussi di azione decisamene più eleganti e complessi, ma proprio non è il caso; anzi è consigliabile impiegare meglio il tempo nel contesto dell'economia dell'intero progetto.

Figura 3.18 — *Differenza di notazione tra la relazione di inclusione e quella di estensione.*

Relazione di estensione

Una relazione di estensione (*Extend*) è una relazione tra un caso d'uso estendente e uno base che specifica come il comportamento definito dal primo (l'estendente) debba essere inserito nel comportamento di quello base.

A questo punto i lettori non molto esperti della vista dei casi d'uso potrebbero essere scossi da più di qualche turbamento: il comportamento dello use case estendente viene incorporato in quello base? In effetti è così. Il problema risiede unicamente nel nome scelto per tale relazione: *Extend*. Considerato — erroneamente — come relazione di estensione canonica, il comportamento sarebbe assolutamente contrario alle convenzioni più classiche dell'OO, ma non bisogna farsi confondere dal nome e ricordare che per specificare relazioni di ereditarietà tra casi d'uso è prevista l'appropriata relazione di generalizzazione.

Si tenga comunque presente che la vista dei casi d'uso prevede come fruitori anche persone che, per definizione, hanno pochissima o nessuna esperienza del mondo OO: se si è fortunati, tali individui appartengono unicamente alla categoria dei clienti, e la relazione di estensione così definita finisce probabilmente per risultar loro più naturale e comprensibile.

Nel diagramma riportato in fig. 3.18 è illustrata la differenza tra la notazione di una relazione di inclusione e una di estensione. Si considerino due use case generici A e B, in cui il primo necessita in qualche modo del secondo. Nel caso in cui:

- lo use case `Annulla prenotazione` **include** quello `Aggiorna disponibilità`, la freccia va dal primo al secondo;

- lo use case `Aggiorna disponibilità` **estende** quello `Annulla prenotazione`, la freccia va riportata esattamente al contrario del caso precedente.

Una relazione di estensione contiene una lista dei nomi dei punti in cui lo use case base può e deve essere esteso (consultare figura seguente). Chiaramente ciascuno di essi deve essere esteso da opportuni segmenti presenti nello e nei casi d'uso estendenti. Un punto di estensione rappresenta una locazione o un insieme nello use case base, nel quale l'estensione può essere inserita.

Come evidenziato nella fig. 3.19, lo use case base A prevede due punti di estensione, mentre il caso d'uso estendente ne fornisce l'apposita definizione. Il punto esclamativo serve a indicare che l'effettiva estensione avviene sotto il controllo di una condizione di guardia.

Per questo motivo, spesso le relazioni di estensione sono utilizzate per modellare delle parti di use case che rappresentano delle azioni facoltative in modo da distinguerle esplicitamente dal flusso delle azioni non opzionali. In questo modo è possibile distinguere il comportamento opzionale da quello obbligatorio del sistema.

Figura 3.19 — *Flusso degli eventi della relazione di estensione*

Il funzionamento prevede che quando un'istanza dello use case base raggiunge una locazione referenziata da un punto di estensione, la condizione viene valutata: se l'esito è positivo (valore `true`) il flusso delle azioni specificate nel relativo caso d'uso estendente viene eseguito, altrimenti si procede oltre nell'esecuzione del flusso delle azioni dello use case base.

L'assenza di una condizione corrisponde a un valore sempre `true`. Nel caso in cui la relazione di estensione preveda diversi punti di estensione, la condizione viene verificata solo la prima volta, ossia prima dell'esecuzione del primo frammento. Chiaramente uno stesso use case può essere esteso da diversi altri, così come un caso d'uso può estenderne molteplici.

Spesso, analizzando i vari flussi degli eventi che descrivono il comportamento dinamico dei singoli casi d'uso, ci si imbatte in vere e proprie procedure scritte in linguaggio di programmazione (la patologia della "programmite" è sempre in agguato) infarcite di comandi `for`, `while`, `if`, ecc.
Questa soluzione tipicamente non riscuote i consensi dei clienti i quali, per definizione, non sono tecnici informatici e tendono a sentirsi a disagio quando il formalismo tecnico prende il sopravvento. L'esperienza insegna che è preferibile cercare di utilizzare un linguaggio quanto più vicino possibile a quello dei clienti e rendere le varie descrizioni dei casi d'uso lineari. Per esempio, qualora uno use case sia molto esteso o troppo complicato, è possibile semplificarne la struttura inserendo alcuni flussi alternativi in appositi caso d'uso, connessi a quello oggetto di studio per mezzo di una relazione di estensione. Ovviamente, utilizzando questo espediente bisogna sempre fare attenzione a non eccedere nella granularità.

Il diagramma riportato in fig. 3.20 corrisponde a una funzione che permette a studenti universitari di prenotarsi per gli appelli delle sedute d'esame previste, magari comodamente seduti nella poltrona di casa, oppure interagendo con opportuni chioschi presenti presso le varie facoltà dell'università (!?).

Come si può notare, il caso d'uso `prenota appello` prevede due punti di estensione denominati rispettivamente `transazione abilitata` e `verifica idoneità`.

Il comportamento del primo punto viene definito dallo use case `seleziona esame`. Nella relativa relazione di estensione viene esplicitamente dichiarato il punto, o meglio, il segmento che il caso d'uso estendente specifica nello use case base: `transazione abilitata`. Sempre nella relazione di estensione è definita la condizione che deve essere soddisfatta affinché il flusso del caso d'uso possa essere eseguito, ossia: *lo studente deve essere stato preventivamente riconosciuto e quindi abilitato dal sistema*.

Figura 3.20 — *Prenotazione appelli di esami universitari: esempio di relazioni di estensione.*

Un discorso del tutto analogo vale per il caso d'uso `verifica idoneità studente` il cui obiettivo è quello di verificare che l'utente sia in possesso di tutti i requisiti (di tipo sia amministrativo, sia di curriculum) per potersi prenotare per la sessione di esami prescelta. In questo caso la condizione da soddisfare è che lo studente abbia precedentemente selezionato l'appello di interesse e il punto in cui viene esteso il caso d'uso base è `verifica idoneità`.

Ancora una volta, probabilmente non è il caso di sprecare energie andando a definire nei diagrammi i punti in cui ogni caso d'uso estendente estende quello base: è possibile fare ciò nel flusso di azioni. Probabilmente, non solo si perde molto tempo, ma si finisce anche per complicare inutilmente i diagrammi.

Il flusso di azioni dello use case di fig. 3.20 potrebbe assumere una forma del tipo:

```
Use case base Prenota appello:
   1. include (validazione studente)
   2. se riconoscimento utente terminato con errore allora
   2.1. termina il caso d'uso con insuccesso
```

3. il sistema tenta di reperire i dati studente
   (facoltà, anno di corso, ecc.)
4. **se** dati studente non reperiti **allora**
4.1. il sistema mostra messaggio
4.2. il sistema termina sessione con insuccesso
4.3. registra l'evento nel file di log.
5. <transazione abilitata>
6. il sistema acquisisce appello selezionato dallo studente
7. <verifica idoneità>
8. fine

Extension use case Seleziona esame:
definzione segmento <transazione abilitata>
1. il sistema visualizza il modulo inserimento anno
2. l'utente imposta l'anno di corso relativo all'esame di interesse.
3. **se** nessun dato impostato **allora**
3.1. il sistema termina caso d'uso con insuccesso
4. il sistema visualizza elenco degli esami
   relativi all'anno impostato
5. l'utente seleziona l'esame selezionato
6. **se** nessun esame selezionato **allora**
6.1. il sistema termina caso d'uso con insuccesso
7. il sistema memorizza l'esame selezionato
8. il sistema mostra appelli previsti per l'esame selezionato
9. **se** nessun appello previsto **allora**
9.1. il sistema mostra messaggio
9.2. torna al punto 1.
10. l'utente seleziona l'appello desiderato
11. **se** nessun esame selezionato **allora**
11.1. il sistema termina caso d'uso con insuccesso
12. il sistema memorizza l'appello selezionato
fine definzione segmento <transazione abilitata>

Extension use case Verifica idoneità studente:
definzione segmento <verifica idoneità>
1. il sistema verifica idoneità amministrativa.
2. **se** anomalie amministrative riscontrate **allora**
2.1. il sistema visualizza messaggio
2.2. termina termina caso d'uso con insuccesso
3. il sistema verifica propedeuticità esame selezionato

```
4. se riscontro propedeuticità fallito allora
4.1. il sistema visualizza messaggio
4.2. termina termina caso d'uso con insuccesso
5. restituisci esito verifica positivo
fine definzione segmento <verifica idoneità>
```

Dall'analisi del flusso degli eventi relativi al caso d'uso `Verifica idoneità studente` si può notare che non viene fornito alcun dettaglio circa i controlli da effettuare per la verifica della situazione amministrativa dell'utente e per il rispetto delle propedeuticità dell'esame selezionato. Come si vedrà nel Capitolo 5, regole del genere sono definite *business rule* e dovrebbero essere inserite in un apposito documento al quale i casi d'uso dovrebbero far riferimento. In un documento in formato elettronico ciò si traduce nell'inserimento di un *link*.

Lo Use Case estendente può avere accesso e modificare gli attributi definiti nello use case base mentre quest'ultimo non ha visione dei casi d'uso estendenti e quindi non può avere accesso ai relativi attributi/operazioni.

Con riferimento al caso presentato, lo use case `seleziona esame`, per poter visualizzare l'elenco degli esami di interesse dello studente per l'anno specificato, deve essere in grado di accedere ad informazioni presenti nello use case base come la facoltà a cui lo studente risulta iscritto.

Quando si ricorre ad una relazione di estensione è possibile immaginare lo use case base come un framework modulare in cui le estensioni possono essere inserite, modificandone implicitamente il comportamento, senza che esso ne abbia visione.

La differenza tra la relazione di generalizzazione e quella di estensione dovrebbe essere fin troppo chiara ma si ribadisce — *repetita juvant* — che nel primo caso il comportamento specializzato è presente nelle istanze che generalizzano lo use case mentre nel secondo caso è riportato direttamente nello use case esteso.

Come già menzionato, la relazione di estensione è rappresentata graficamente per mezzo di una freccia tratteggiata corredata dall'etichetta `<<extend>>`.

Tipicamente è consigliabile organizzare i diagrammi dei casi d'uso estraendo il comportamento comune (utilizzo delle relazioni di inclusione) e distinguendo le varianti (relazione di estensione). Ciò a patto di non cadere nei tipici errori legati ad un utilizzo inesperto della notazione dei casi d'uso: tentativo di descrivere lo spazio delle soluzioni, decomposizione funzionale, eccessiva granularità, ecc.

Inclusione o estensione?

Nel processo di progettazione degli use case diagram, è prassi comune strutturare i casi d'uso associandoli tra loro per mezzo di relazioni di inclusione ed estensione. Ciò permette di realizzare diagrammi più eleganti, chiari e quindi, in ultima analisi, più facilmente comprensibili.