Capitolo 1

UML: che cosa è, che cosa non è

Beati monoculi in regno caecorum

Introduzione

L'obiettivo del capitolo è chiarire sin da subito, e in modo inequivocabile, che cosa *sia* e che cosa *non sia* lo UML e, quindi, quali siano le relative aree di competenza e gli obiettivi. A tal fine vengono presentate anche altre tecnologie strettamente correlate con lo UML, nella convinzione che ciò, aiutando a definire più distintamente i confini dello UML, concorra a fornire una visione più completa e precisa dello stesso. Logica conseguenza è che il capitolo risulta essere decisamente corposo e non sempre di facile comprensione. Il fattore positivo è che si può procedere tranquillamente nella lettura del testo pur non comprendendo appieno quanto riportato. Il consiglio per tutti coloro che si accostano allo UML per la prima volta è di leggerlo comunque, magari molto rapidamente, una prima volta, con l'intento di tornare a rileggerlo in un secondo momento, quando si sarà in possesso di una maggiore familiarità con il linguaggio stesso. Allora si potranno apprezzare maggiormente gli argomenti illustrati e si potrà comprendere pienamente il rapporto che intercorre tra lo UML e le tecnologie correlate.

Dopo una breve riflessione sull'importanza della modellazione nei processi di sviluppo del software, si è ritenuto utile fornire un conciso quadro "storico" della situazione presente prima e durante l'evoluzione dello UML, al fine di evidenziare le motivazioni teoriche che hanno spinto i cosiddetti Tres Amigos (Grady Booch, James Rumbaugh, Ivar Jacobson) ad affrontare il superbo progetto del linguaggio di modellazione unificato.

Sebbene sia opinione comune di molti tecnici che lo UML rappresenti "semplicemente" una notazione standard per la creazione di modelli Object Oriented di sistemi, in

realtà è molto di più. Esso possiede una definizione rigorosa di metamodello, istanza del meta-metamodello noto con la sigla MOF (Model-Object Facility). Si tratta di concetti molto eleganti e affascinanti, a elevato livello di astrazione. Tali concetti meriterebbero, in effetti, una trattazione più rigorosa (le specifiche ufficiali constano di oltre 500 pagine) rispetto a quella riportata, ma ovviamente — e fortunatamente — ciò esula dagli obiettivi del presente libro.

Il problema che si pone a questo punto è che il metamodello dello UML è descritto per mezzo dello UML stesso: un'istanza che definisce la classe (ciò rievoca il ricordo del compilatore del linguaggio C scritto in linguaggio C).

Si tratta indubbiamente di una scelta molto elegante, ma che, come inconveniente, può creare seri problemi a tutti coloro che affrontano lo UML per la prima volta: è come andare a lezione di Inglese senza conoscerne una parola e assistere a spiegazioni esclusivamente in lingua impartite da un insegnante di madrelingua che non faccia altro che parlare e parlare.

La presentazione dell'architettura dello UML è stata inserita in questo contesto solo al fine di rendere i lettori consapevoli delle specifiche formali, nonostante la relativa illustrazione venga ripresa e argomentata nel corso di tutto il libro. Se i vari concetti dovessero risultare troppo enigmatici non c'è da preoccuparsi più di tanto: si può utilizzare tranquillamente lo UML senza averne capito il metamodello (si può tranquillamente programmare in C senza sapere come sia stato realizzato il relativo compilatore).

I paragrafi successivi a quelli relativi al metamodello sono dedicati alla trattazione dei metodi di sviluppo (i famosi processi) sia da un punto di vista generale — illustrazione delle principali "dottrine" (Use case Driven, Architecture Centric e Iterative and Incremental) —, sia da un punto di vista particolare — presentazione dei processi più diffusi (The Unified Software Development Process, frutto del lavoro dei Tres Amigos, Use case Driven Object Modeling with UML, OPEN Process patrocinato dall'OPEN Consultorium e Object-Oriented Software Process).

Per questioni di completezza viene fornita una brevissima introduzione dell'eXtreme Programming (XP) con tutte le relative perplessità generate e la relativa evoluzione fornita dall'Agile Modeling, in cui, fortunatamente, l'attenzione viene nuovamente spostata sulla fase di modellazione.

La modellazione

Ogni qualvolta, in una disciplina dell'ingegneria, vi sia la necessità di realizzare un "manufatto", indipendentemente dalla dimensione e dal settore di interesse (una casa, un grattacielo, un particolare meccanismo, un ponte, un dipartimento di un'azienda, e così via) si procede cercando di realizzarne un modello.

L'obiettivo è produrre, in tempi relativamente brevi e soprattutto a costi contenuti, una versione razionalizzata e semplificata del sistema reale che, tuttavia, consenta di evidenziarne l'aspetto finale e di studiarne prestazioni, affidabilità, comportamento, ecc…

I modelli, importantissimi in tutti i processi di sviluppo, trovano la loro più completa legittimazione nell'ambito di sistemi di dimensioni medie e grandi. In tali circostanze la consueta complessità intrinseca del sistema è amplificata dalla dimensione al punto che tentare di affrontarlo nella sua interezza è un'impresa assurda. È quindi maggiormente avvertita la necessità di avvalersi di modelli in grado di descrivere, in maniera semplificata, sistemi comunque complessi.

Si vedrà come lo UML, grazie alla sua organizzazione in "viste" (*view*), risponda alla logica necessità della mente umana di concentrarsi, in ogni istante, su un numero limitato di aspetti del sistema ritenuti importanti per la particolare fase del processo di sviluppo, rimandando a momenti successivi l'analisi degli aspetti rilevanti per le altre viste.

Oltre ai motivi già citati, i modelli sono basilari poiché permettono di ricevere il feedback dei committenti fin dalle primissime fasi del ciclo di vita del software. Ciò permette di definire i requisiti del sistema in maniera attendibile e, con la cooperazione del proprio gruppo, di razionalizzarne il processo di sviluppo. I vantaggi che se ne ricavano sono diversi: adeguate analisi dei tempi, migliori stime dei costi, piani più precisi di allocazione delle risorse, distribuzioni più affidabili del carico di lavoro, e così via.

Si riesce quindi a risparmiare tempo e denaro, a ridurre i fattori di rischio presenti in ogni progetto, a studiare la risposta del sistema a particolari sollecitazioni, e via di seguito.

Nonostante il grande valore apportato dalla modellazione all'ingegneria del software, troppo spesso, in molte organizzazioni, questa meravigliosa tecnica rimane ancora una chimera. A nessuno appartenente al settore dell'ingegneria edile verrebbe in mente di costruire interamente un grattacielo, per poi studiarne la risposta a sollecitazioni di carattere sismico. Il buon senso —risorsa purtroppo sempre rara — suggerisce di procedere con la realizzazione di una versione ridotta sulla quale condurre tutti gli studi del caso. Non necessariamente un processo di modellazione genera un oggetto tangibile: talvolta si tenta di rappresentare un complesso reale per mezzo di eleganti sistemi di disequazioni e quindi l'intero modello si risolve in una raffinata astrazione.

Tutto ciò che è fin troppo scontato in molte discipline dell'ingegneria non lo è nel settore dell'informatica. In molte organizzazioni — e qui non si pensi a un limite riscontrabile solo nelle piccole realtà — la produzione del software è ancora un'attività, per così dire, "artigianale" in cui il relativo processo di sviluppo del software prevede tre fasi: analisi dei requisiti, implementazione e test… Forse.

Si provi a immaginare che cosa potrebbe accadere se si avviasse la progettazione di un ponte a partire da specifiche sommarie, magari comunicate verbalmente o, peggio ancora, se si partisse subito a costruirlo materialmente, magari affidandosi all'esperienza di qualche costruttore. Tutto ciò, benché possa sembrare una caricatura, molto spesso è prassi comune nel mondo dell'ingegneria informatica.

Malgrado siano stati versati fiumi d'inchiostro sull'argomento, e svariati gruppi di ricerca lavorino a tempo pieno per la definizione di nuovi standard di analisi, in molte organizzazioni, soprattutto italiane, anche di "comprovato" prestigio, tale attività è anco-

ra considerata prettamente accademica, vezzo di giovani neolaureati perché, in ultima analisi, è di scarsa utilità nel mondo reale, dove l'obiettivo è produrre codice e l'esperienza è in grado di sopperire a tutto.

L'inevitabile risultato è che spesso si procede cominciando paradossalmente da una delle ultime fasi del processo di ingegnerizzazione del software, ossia dalla stesura del codice (paragonabile alla costruzione del ponte di cui si parlava nell'esempio).

Si inizia concentrandosi prematuramente sul modo in cui redigere il codice ed, eventualmente, alla fine si scrivono "due righe di analisi", magari demandando il tedioso incarico all'ultimo arrivato nel team, al baldo giovane di turno. I rischi che si corrono sono noti a tutti e possono generare delle "veniali" anomalie note in letteratura informatica come "crisi del software".

In molte occasioni — l'autore potrebbe citarne diverse, ma preferirebbe evitare vitto e alloggio a carico dello Stato… — può accadere che, a progetto "ultimato e pronto per la consegna", ci si accorga dell'errata architettura. Non è da escludere, nel peggiore dei casi, che, per via di uno sviluppo interamente incentrato sulla fase di codifica, e quindi privo della visione globale che solo un buon processo di modellazione può apportare, sia praticamente impossibile integrare i moduli prodotti in parallelo o che sia necessario costruire estemporanei moduli di interfacciamento o, ancora, che il sistema preveda percorsi così tortuosi da renderlo praticamente inutilizzabile.

L'unico vantaggio in questo modo di procedere è che si placano, almeno sul momento, gli stati d'ansia di taluni manager, i quali diventano improvvisamente irrequieti quando il team non genera codice, anche se nel frattempo si stanno affrontando delicate e impegnative fasi di analisi e/o disegno.

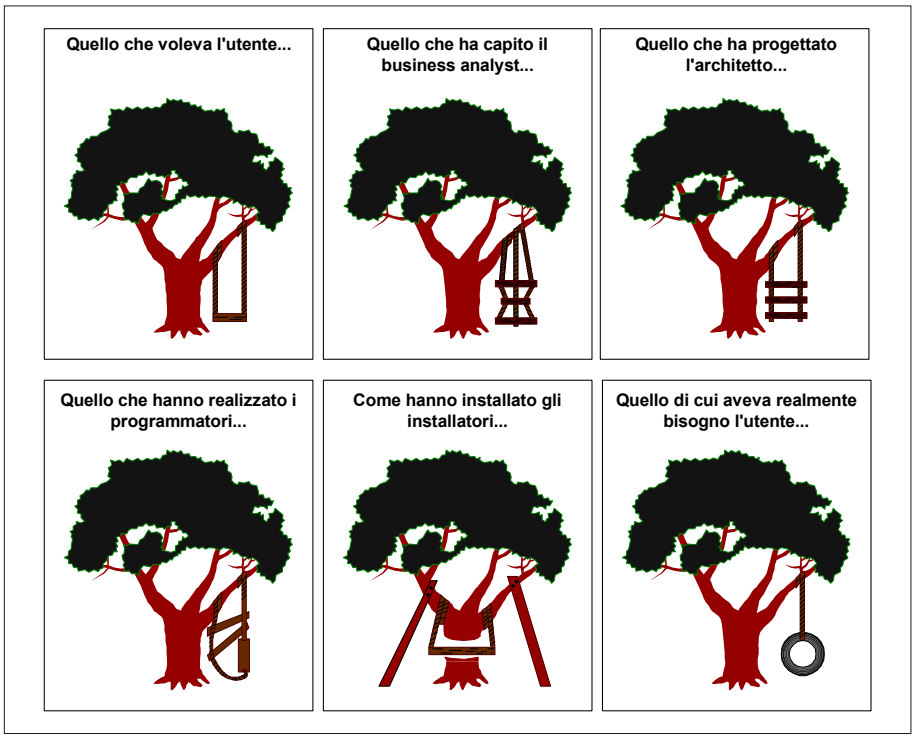
L'alibi che viene rispolverato è lo scarso tempo a disposizione, dimenticando quanto incida il processo di manutenzione sull'intero ciclo di vita del software: il famoso "risparmio della massaia".

A onor del vero, gli eventuali fallimenti non sono sempre imputabili a tecnici/commerciali che "vendono" l'impossibile o a manager informaticamente poco lungimiranti. Molto spesso — è qui doveroso procedere con un po' di sana autocritica — gli stessi progettisti si sentono più a loro agio quando si deve affrontare un problema di programmazione, piuttosto che quando si deve affrontare la modellazione di un sistema.

Probabilmente si tratta anche dell'eredità di una dannosa abitudine: all'inizio i sistemi non erano così complessi, le metodologie di progettazione erano quasi inesistenti, i problemi e gli imprevisti dell'implementazione erano di quantità e entità tutto sommato limitate, tali da rendere quasi giustificabile la contrazione del processo di modellazione. Eppure da allora alcune cose dovrebbero essere cambiate… E pensare che, da oltre un decennio, in ambito accademico si afferma che "la programmazione è un incidente — anche molto piacevole se si vuole — dell'informatica".

Nella realtà, troppo spesso si liquidano prematuramente le fasi di analisi e disegno del software per tuffarsi a testa bassa nella programmazione. Come nella celebre "vignetta

Figura 1.1 — *La vignetta dell'altalena*



dell'altalena", tutti sono felici e contenti, almeno fino al momento di appenderla al ramo dell'albero. La temporanea soddisfazione viene vinta dalla triste constatazione che, a causa del modo superficiale in cui si è operato, è necessario operare un taglio ortogonale alla base del tronco dell'albero per poter ospitare il seggiolino.

In altre realtà, può succedere di realizzare un'analisi adeguata, di produrre modelli sofisticati ad elevato grado di eleganza da dover poi fornire a un team di sviluppatori "object disoriented" producendo unicamente il silenzio (meglio restare in silenzio e dare l'impressione di essere stolti che aprire bocca e togliere ogni dubbio).

Questo per evidenziare che gli strumenti da utilizzare in fase di analisi, purtroppo, possono anche dipendere, in gran misura, dal team con cui si lavora.

Premesso ciò, va comunque ribadito che il processo di analisi e modellazione è un'attività estremamente creativa, strettamente dipendente dalle capacità, dalla cultura e dall'esperienza dei singoli: insomma un vero e proprio bene soggettivo. Il bello — si fa per dire — è che non esiste un manuale delle soluzioni corrette con il quale confrontarsi alla fine del proprio studio: solo il tempo fornirà i verdetti.

Per terminare, sintetizzando al massimo il concetto, si può dire che si modella essenzialmente per due motivi:

- **aumentare la propria comprensione** di un particolare dominio, di un problema, delle sue eventuali soluzioni, ecc.;
- **comunicare**.

Qualità di un modello

Illustrato il concetto di modello — cui si ricorrerà ampiamente in tutto il libro — si ritiene opportuno evidenziarne brevemente le qualità peculiari di interesse nel contesto dello sviluppo di sistemi software. Queste dovrebbero essere tenute a mente e guidare le attività di progettazione, al fine di produrre modelli di migliore qualità. Un modello dovrebbe garantire:

- **accuratezza**: deve descrivere il sistema correttamente, completamente e senza ambiguità;
- **consistenza**: le diverse viste devono completarsi vicendevolmente per formare un insieme coerente (la completezza di un'erma bifronte…);
- **semplicità**: deve poter essere compreso, senza troppi problemi, da persone estranee al processo di modellazione;
- **manutenibilità**: la variazione dello stesso deve essere la più semplice possibile.

Nascita e sviluppo di UML

La babele dei metodi

Uno degli obiettivi che hanno caratterizzato il lavoro dei Tres Amigos è stato realizzare un linguaggio di modellazione che unificasse e incorporasse le caratteristiche migliori dei linguaggi esistenti intorno agli anni Novanta.

In particolare, come citato esplicitamente nel documento delle specifiche [BIB07], lo UML è scaturito principalmente dalla fusione dei concetti presenti nei metodi di Grady Booch, OMT (Object Modeling Technique di cui Rumbaugh era uno dei principali fautori) e OOSE (Object Oriented Software Engineering /Objectory di cui Ivar Jacobson era uno dei più importanti promotori).

Agli inizi degli anni Novanta, ossia poco prima dell'avvio dei lavori per lo UML, i suddetti metodi erano probabilmente quelli più apprezzati, a livello mondiale, dalla comunità Object Oriented.

> Il **metodo di Booch**, diventato famoso nel settore con il nome di "metodo delle nuvolette", definisce una notazione in cui il sistema viene analizzato e suddiviso in un insieme di viste, ciascuna costituita dai diagrammi di modello. Il metodo non si limita a un linguaggio di modellazione, ma contiene anche un processo, in base al quale il sistema viene studiato attraverso micro- e macroviste di sviluppo, secondo un classico schema incrementale e iterativo. Il metodo di Booch, a detta degli esperti, sembrerebbe molto efficace soprattutto in fase di disegno e di codifica, mentre presenterebbe qualche lacuna nelle varie fasi di analisi.
>
> L'**OMT** è un linguaggio di modellazione, sviluppato presso la General Electric, rilevatosi particolarmente efficace nella fase di esplorazione dei requisiti. In maniera speculare al metodo precedente, quest'ultimo sembrerebbe essere carente nella fase della formulazione delle soluzioni, mentre risulterebbe particolarmente accurato nell'esplorazione delle specifiche nel dominio del problema. L'OMT prevede che il sistema venga descritto attraverso un preciso numero di modelli che si completano vicendevolmente. Grazie alla sua accuratezza nella fase di analisi dei requisiti, il modello fornisce un valido ausilio anche alla fase di test di sistema.
>
> Infine i metodi **OOSE** e Objectory (in gran parte dovuti al lavoro di Jacobson) si basano, quasi interamente, sugli Use Case (casi d'uso). Poiché gli Use Case permettono di definire i requisiti iniziali del sistema così come vengono percepiti da un attore esterno allo stesso, i metodi OOSE e **Objectory** si sono dimostrati particolarmente efficaci nello spazio dell'analisi nel dominio del problema. Anche questi metodi forniscono una serie di informazioni e linee guida su come passare dall'analisi dei requisiti al disegno del sistema.
>
> Altri metodi degni di essere menzionati sono Fusion (elaborato presso i laboratori della Hewlett-Packard) e il metodo Coad/Yourdon (noto anche come OOA/OOD – Object Oriented Analisys / Object Oriented Design) che vanta il primato di essere stato uno dei primi metodi di analisi e disegno di sistemi Object Oriented.

Negli anni intercorsi tra il 1989 ed il 1994, oltre ai metodi appena citati, considerati tra i più importanti, se ne contarono addirittura una cinquantina. Questi andarono ad alimentare quella che fu definita "guerra dei metodi": la famosa "Babele" tra i modelli software con la conseguente incomunicabilità. Ovviamente ciascuno di questi presentava punti di forza, lacune, un proprio formalismo, una nomenclatura proprietaria e un peculiare processo di sviluppo.

Tutto ciò finiva per minare alla base lo sviluppo di processi di progettazione più accurati e rendeva difficoltosa la realizzazione di opportuni tool di supporto. Altri inconvenienti, sempre frutto dell'incomunicabilità dei metodi, erano legati all'impossibilità di far circolare informazioni tra team di aziende diverse, alla difficoltà di rendere rapidamente

produttive nuove risorse allocate al progetto e così via. In ultima analisi si finiva per fornire un pretesto ai vari team "analisi-disegno repellenti".

Le motivazioni

Il presente libro viene pubblicato dopo più di nove anni da quando Grady Booch e James Rumbaugh iniziarono i lavori alla Rational Software Corporation (1994). Probabilmente si è trattato della tecnologia giusta al momento giusto (ciò potrebbe far impallidire anche il buon Murphy), visto il grande successo riscosso e le prestigiose collaborazioni che hanno contributo alla sua realizzazione. Tra i partner si contano alcune delle aziende più importanti nel settore della Computer Science, tra le quali IBM, Oracle, Microsoft, Sun Microsystem, Texas Instruments, MCI Systemhouse, Hewlett-Packard e così via.

Il clamoroso successo riscosso è molto probabilmente dovuto alla necessità, avvertita da tutta la comunità Object Oriented, di disporre di un unico linguaggio di modellazione che mettesse finalmente termine alla proliferazione degli anni precedenti. La "guerra dei metodi" ha finito per costituire un serio limite all'evoluzione di una rigorosa metodologia di sviluppo del software. Le aziende che producevano tool di supporto al processo di sviluppo del software, i famosi CASE (Computer Aided Software Engineering), avevano notevoli problemi nello scommettere sul linguaggio da adottare. Si trattava di affidare la politica di sviluppo dell'azienda al successo di un metodo: vera e propria roulette russa. L'idea di produrre plug-in per il supporto degli altri linguaggi risultava uno sforzo troppo impegnativo e costoso, vista la significativa disomogeneità degli stessi.

Dal canto loro le aziende di Information Tecnhnology non riuscivano a orientarsi bene nello scegliere un metodo da utilizzare come standard interno. L'investimento per la produzione di processi proprietari e per l'addestramento del personale risultava così elevato, che lo spettro di una scelta sbagliata rendeva di fatto molto difficile qualsiasi decisione e quindi finiva per produrre una certa immobilità. Le grandi aziende finivano per crearsi un proprio processo proprietario, basato sul bagaglio delle esperienze accumulate, e quindi assolutamente non supportato. Nelle imprese medio-piccole, molto spesso, tutto era demandato alla capacità e alla saggezza dei singoli architetti. Il risultato era una totale incomunicabilità tra i vari team di sviluppo e una discrepanza di metodi presenti in progetti sviluppati in tempi diversi e sotto la direzione di persone diverse. La confusione imperante generava diversi problemi anche ai singoli tecnici, desiderosi di utilizzare un linguaggio di modellazione che agevolasse il lavoro di tutti i giorni e che fosse supportato da tool commerciali.

Tali situazioni prepararono il terreno ideale alla sollecita accettazione del progetto di unificazione dei metodi. Non a caso, il progetto avviato alla Rational Software Corporation suscitò l'immediato consenso di tutta la comunità OO. Altro fattore non trascurabile è che i fautori (i Tres Amigos) del progetto dello UML erano padri dei metodi più conosciuti ed

utilizzati. Chiaramente l'accettazione e il supporto del progetto da parte di altri metodologi non coinvolti nel lavoro fu tutt'altro che scontato: ma questo è un altro discorso.

Se a tutto ciò si aggiunge il fatto che il linguaggio è *nonproprietario* si può ben capire che la sua affermazione era garantita sin dall'inizio della sua invenzione. Si tratta infatti di un linguaggio completamente aperto, tanto che le aziende sono incoraggiate a integrarlo nei propri metodi di sviluppo del software e che le imprese produttrici di CASE possono liberamente produrre software di supporto allo UML.

La genesi

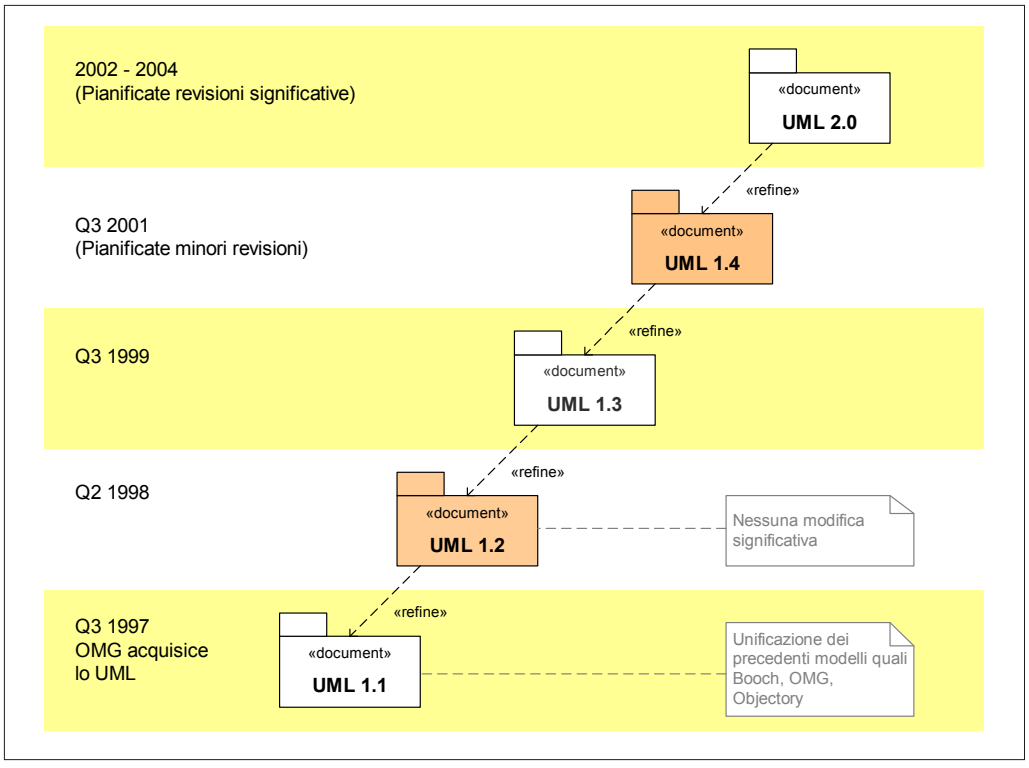
Originariamente il lavoro fu iniziato dai *Dos Amigos* Grady Booch e James Rumbaugh, con l'intento di produrre un nuovo metodo, detto "metodo unificato", che raccogliesse il meglio dei metodi Booch e OMT-2, del quale Rumbaugh era stato uno dei principali promotori. Nel 1995 si unì a loro Ivar Jacobson, fautore del metodo denominato OOSE (Object Oriented Software Engineering): il terzetto si era così costituito. L'azienda che promuove il progetto è la Rational Software Corporation che, dal canto suo, provvede anche ad assorbire la Objective Systems, azienda svedese che aveva sviluppato e distribuito il software Objectory. A questo punto il quadro era completo e lo standard in lavorazione fu ribattezzato Unified Modeling Language. La prima versione, la celebre 1.0, fu disponibile nel gennaio 1997.

Lo UML è uno strumento molto versatile, nato per risolvere le problematiche connesse con la progettazione Object Oriented del software, ma che ben si adatta a essere utilizzato negli ambienti più disparati.

> Nella primissima versione è stato, per esempio, utilizzato alla Cadence per la produzione di un dispositivo di compressione vocale operante a livello di porta fisica. Ancora, una delle aziende fornitrici della US Navy ha utilizzato lo UML come linguaggio di progettazione di un sistema d'arma di nuova generazione. Un'azienda sanitaria si è avvalsa dello UML nella realizzazione di un modello per il trattamento dei pazienti, e così via.

Visto l'enorme successo riscosso nell'applicazione industriale e nel mondo accademico e considerato il relativo riconoscimento a livello di standard (UML 1.0 è stato proposto all'Object Management Group nel gennaio 1997), gli stessi ideatori, i Tres Amigos, dichiararono ormai conclusa la loro esperienza in questo ambito tanto da dedicarsi a nuove sfide… Allo stato attuale lo sviluppo dell'UML è affidato a una task force appartenente all'OMG, la famosa RTF (Revision Task Force… però che fantasia), diretta da Chris Kobyrn. Obiettivo di tale gruppo è accogliere e analizzare suggerimenti provenienti dalle industrie, correggere inevitabili imperfezioni (bug), colmare eventuali lacune e così via.

Figura 1.2 — *Diagramma dei componenti dell'evoluzione dello UML. Nel diagramma sono riportati due stereotipi, etichettati rispettivamente con i termini inglesi* refine *e* document. *Per ora basti sapere che uno stereotipo è una specializzazione di un elemento standard UML. In particolare* refine *è una versione della relazione di dipendenza il cui significato è piuttosto intuitivo. In generale l'asserzione che un oggetto B dipende da un oggetto A (visualizzata per mezzo di una freccia tratteggiata in direzione dell'oggetto A), indica che una variazione all'oggetto indipendente (A) produce la necessità di revisionare ed eventualmente aggiornare l'oggetto dipendente (B). Lo stereotipo* document *rappresenta una particolare versione dell'elemento* package*, che, in questa fase, può essere considerato come un contenitore di oggetti. La similitudine con le cartelle del file system è piuttosto immediata. Una directory (o cartella che dir si voglia) è in grado di contenere eventuali altre cartelle e file delle più svariate tipologie (audio, filmati, grafici, testo, e così via).*



Attualmente è disponibile la versione 1.4 (risultato dell'attività della seconda RTF) e si sta lavorando alla versione 2.0 come mostrato nel paragrafo seguente. L'evoluzione dello UML è mostrata in fig. 1.2, attraverso uno degli strumenti messi a disposizione dal linguaggio stesso.

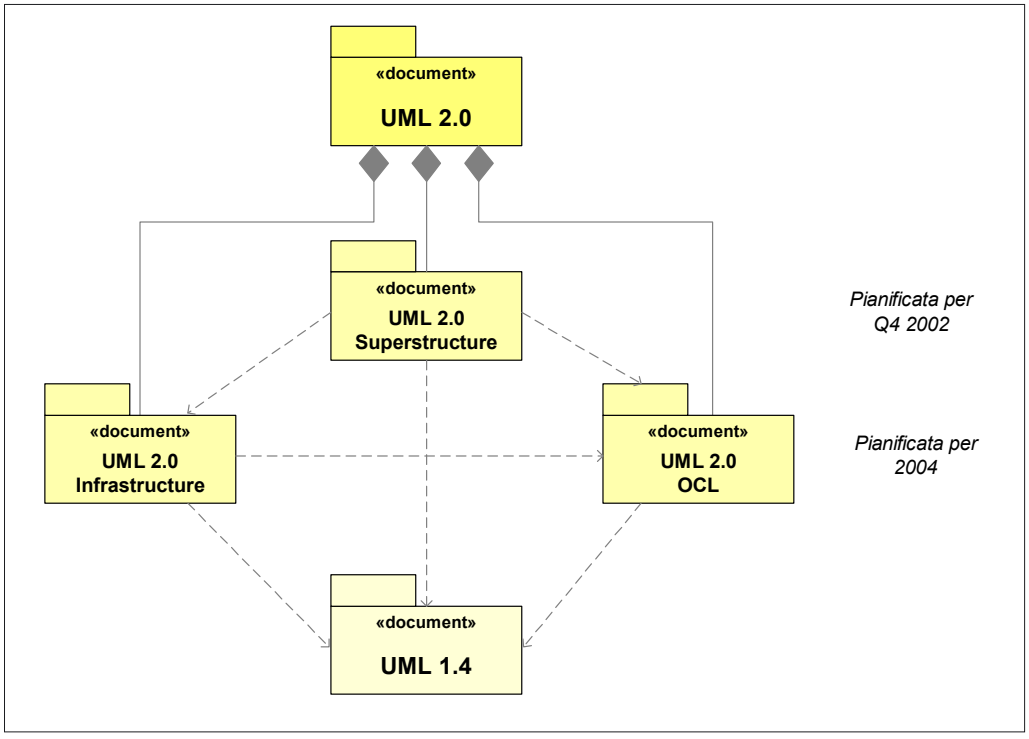
UML 2.0

A questo punto viene da interrogarsi legittimamente su come stiano procedendo i lavori per la versione 2.0 (etichettata spaventosamente come "major revision"), che cosa sia lecito attendersi per i legami con la serie 1.x dello UML, e così via.

Molto probabilmente uno degli impegni più importanti consiste nel proseguire nella direzione, già intrapresa con la versione 1.4, dello studio delle necessità dei sistemi component-based. Ciò al fine di adeguare sempre più lo UML allo stato dell'arte della tecnologia e quindi di fornire soluzioni concrete alle esigenze dei tecnici coinvolti nella realizzazione di sistemi basati sui componenti.

Si consideri il diagramma riportato nella figura precedente. Come annunciato in precedenza dallo OMG, la RTF per la versione 2.0 è stata suddivisa (questa volta ufficialmente) in tre parti.

Figura 1.3 — *Decomposizione delle attività dello UML 2.0 in sottogruppi. Le linee culminanti con un diamante rappresentano relazioni di composizione dette anche whole-part (tutto-parte). In particolare, nel contesto oggetto di studio, indicano che la versione UML 2.0 è costituita dalla documentazione prodotta dalla RTF Infrastructure, Superstructure e OCL.*



Infrastruttura (Infrastructure RFP, OMG document ad/00-09-01)

Come suggerisce il nome, obiettivo di questa RTF consiste nel curare miglioramenti relativi alla struttura base del metamodello UML, riguardanti il core, i meccanismi di estensione, le facility del linguaggio ecc. In particolare, i relativi obiettivi sono di allineare l'architettura dello UML agli altri standard gestiti dall'OMG (quali per esempio il MOF, Meta Object Facility, Meta-data Interchange XMI), ristrutturare il linguaggio al fine di renderlo più comprensibile ed estenderlo, mantenendo però la semantica e la notazione attuale. (Per ulteriori informazioni consultare l'appendice A).

Superstruttura (Superstructure RFP, OMG document ad/00-09-02)

Obiettivo di questa RTF è elaborare proposte atte a incorporare le *best practices* in aree particolarmente sensibili, come lo sviluppo di sistemi basati sui componenti, i modelli architetturali, quelli eseguibili e dell'area business. In particolare è necessario studiare ulteriormente le soluzioni per:

- la gestione degli eventi nei diagrammi delle attività;
- migliorare l'illustrazione dell'incapsulamento con particolare riferimento ai diagrammi di stato e di sequenza;
- ridefinire la semantica di relazioni come la generalizzazione, associazione, ecc.
- il supporto dello sviluppo component-based di sistemi software;
- il supporto per architetture a tempo di esecuzione, favorendone la descrizione del comportamento dinamico e l'eventuale rappresentazione gerarchica.

Object Constraint Language (OMG document ad/00-09-03)

Il nome di questa RTF evidenzia chiaramente quale sia la relativa area di competenza: l'OCL. Gli obiettivi sono di presentare proposte atte ad aumentare il livello di specializzazione dell'OCL per l'utilizzo UML. Ciò dovrebbe semplificare il lavoro degli utenti, favorendo l'aumento di formalità dei vari modelli prodotti.

Da diverso tempo è in corso un dibattito all'interno dell'OMG stesso, teso a investigare la necessità di introdurre un'ulteriore Task Force con l'obiettivo di studiare l'area relativa allo scambio di modelli UML prodotti da tool diversi. Il nome di tale gruppo dovrebbe essere Diagram Interchange RFP. Sebbene in tale direzione ci sia già stato un impegno iniziale della seconda RTF, il relativo interesse è stato focalizzato principalmente sulla semantica e non sulla problematica specifica dello scambio concreto di diagrammi.

Come ben noto a tutti i tecnici informatici, l'aumento del parallelismo nello svolgimento di attività presenta evidenti vantaggi — riduzione dei tempi grazie al lavoro in paralle-

lo, maggiore specificità delle tematiche affrontate, ecc. — a fronte dei quali però, sono presenti tutta una serie di problematiche. In questo caso sono relative essenzialmente all'incremento del lavoro necessario sia per l'integrazione delle proposte avanzate dai singoli gruppi, sia per il mantenimento della coerenza tra le varie parti costituenti le specifiche finali.

Pertanto, ulteriore preoccupazione della RTF per la revisione 2.0 dello UML è controllare lo sviluppo parallelo delle varie task force affinché le varie componenti lavorino nella stessa direzione e producano la *major revision* preservando la coerenza e le strutture delle precedenti versioni mostratesi particolarmente efficaci.

Obiettivi dello UML

Vengono di seguito descritti gli obiettivi che i Tres Amigos si sono prefissi di raggiungere attraverso lo sviluppo dello UML. La relativa descrizione è stata tratta direttamente dal documento ufficiale delle specifiche [BIB07]. Scopi principali dello UML sono:

- Fornire agli utenti un linguaggio di modellazione visuale pronto ad essere utilizzato per sviluppare e scambiare modelli espressivi. Chiaramente, risulta molto importante che un linguaggio di analisi e disegno Object Oriented standard (OA&D Object Analysis & Design) fornisca tutti i concetti necessari alla sua diretta utilizzazione nelle fondamentali attività del processo di produzione di un modello software. Lo UML ha consolidato un insieme nucleare di concetti di modellazione, necessari in molte applicazioni reali, presenti in molti metodi di progettazione e tool di modellazione disponibili sul mercato.

- Fornire meccanismi di estensione e specializzazione al fine di accrescere i concetti presenti nel nucleo. Uno dei punti fermi che ha guidato il lavoro dei Tres Amigos è stato realizzare un linguaggio di modellazione quanto più generale possibile, in modo da non relegarne l'utilizzo a un dominio specifico. In prima analisi, si sarebbe potuto raggiungere tale obiettivo corredando lo UML di un numero di elementi molto elevato, in modo tale da fornire gli strumenti specificamente necessari a tutte le esigenze dei vari ambienti e delle varie architetture. Si sarebbe potuto, ad esempio, includere nello UML tutta una serie di elementi specifici per la modellazione di sistemi CORBA, EJB, ecc. Tutto ciò però avrebbe sicuramente portato a un conflitto con un'altra esigenza, altrettanto importante di quella dell'applicazione dell'UML in campi specifici: mantenere il linguaggio il più semplice possibile, al fine di massimizzarne l'accettazione da parte della comunità Object Oriented. La soluzione a queste esigenze contrastanti è stato realizzare un nucleo del linguaggio basilare, valido per ogni ambiente, corredato da una serie di strumenti che consentano di aggiungere nuova semantica e sintassi al linguaggio stesso

per utilizzi specifici dello UML. Il risultato è che i concetti fondamentali del linguaggio vengono utilizzati così come sono, senza ulteriori estensioni, per la maggior parte del sistema: vige, anche in questo contesto, la famosa legge empirica dell'80/20: l'80% del progetto utilizza il 20% dei concetti dello UML. Per quanto riguarda la restante parte e per progetti molto specifici è possibile sia aggiungere nuovi concetti e notazioni, al fine di modellare opportunamente le aree non coperte dal nucleo dello UML, sia specializzare quelle già esistenti per particolari domini dell'applicazione.

- Supportare specifiche che risultino indipendenti da un particolare linguaggio di programmazione o processo di sviluppo. Lo UML è stato concepito con l'obiettivo di supportare plausibilmente tutti i linguaggi di programmazione. Quindi è stato reso idoneo all'utilizzo nei più svariati ambienti produttivi, che ricorrono a una vasta gamma di tecnologie, e inoltre si è fatto in modo di preservare la sua adattabilità a sviluppi futuri. Chiaramente lo UML risulta particolarmente indirizzato a linguaggi di programmazione Object Oriented. Pertanto parte della sua efficacia viene attenuata da linguaggi che non realizzano tale paradigma. Come solitamente succede, molto dipende dall'utilizzatore. Per esempio si può programmare in C seguendo il paradigma Object Oriented, al limite aiutandosi con un sistema evoluto di macro (le prime versioni di compilatori C++ altro non erano che preprocessori del linguaggio C, C-front). Si è altresì cercato di rendere lo UML idoneo a essere utilizzato con molteplici metodi di sviluppo: può essere proficuamente adoperato per esprimere i "prodotti" generati dalle varie fasi di cui ogni processo è composto.

- Fornire le basi formali per comprendere il linguaggio di modellazione. Un linguaggio di modellazione deve necessariamente essere preciso e al contempo offrire una complessità contenuta. I formalismi non dovrebbero ricorrere all'utilizzo di nozioni matematiche di basso livello e distanti dal dominio del modello. Tali insiemi di nozioni teoriche e di definizioni operative risulterebbero essere una forma diversa di programmazione e implementazione, non idonea a tutte le fasi del processo di sviluppo del software. Lo UML fornisce una definizione formale del modello, utilizzando un metamodello espresso attraverso il diagramma delle classi dello stesso UML. Esso presenta pertanto un adeguato formalismo e una complessità contenuta.

- Incoraggiare la crescita del mercato dei tool Object Oriented. L'esistenza di un linguaggio di modellazione standard doveva — e così è stato — eliminare tutti quei problemi analizzati nel paragrafo dedicato alle motivazioni dello UML: difficoltà per le aziende di selezionare standard da adottare, frustrazione da parte dei tecnici di fronte alla proliferazione dei metodi, difficoltà di circolazione dei vari modelli prodotti, impedimenti per le aziende creatrici di CASE di individuare metodi da

automatizzare, ecc. Pertanto, una volta rimossi i problemi alla base dello sviluppo formale dell'OO, venne incrementata la crescita del mercato dei prodotti commerciali per il supporto dello UML. Si trattò di un vero e proprio "toccasana" per tutta la comunità Object Oriented.

- Supportare concetti di sviluppo di alto livello come componenti, collaborazioni, framework e pattern. Ulteriore obiettivo dello UML fu la chiara definizione dei concetti succitati poiché ciò è essenziale per trarre i massimi benefici dal paradigma Object Oriented e in particolare da quello della riusabilità.

- Integrare e rendere possibile l'utilizzo delle best practice. Un'altra motivazione alla base dello UML è stata integrare le pratiche mostratesi vincenti nella realizzazione di progetti reali. Ciò è risultato particolarmente utile al fine di offrire sin da subito un linguaggio maturo che inglobasse il meglio delle tecniche di comprovata validità.

Che cosa è lo UML

Secondo le specifiche [BIB07] lo Unified Modeling Language è un linguaggio per specificare, costruire, visualizzare e documentare manufatti sia di sistemi software, sia di processi produttivi e altri sistemi non strettamente software. UML rappresenta una collezione di best practice di ingegneria dimostratesi vincenti nella modellazione di vasti e complessi sistemi. Lo UML permette di visualizzare, per mezzo di un formalismo rigoroso, "manufatti" dell'ingegneria, consentendo di illustrare idee, decisioni prese, e soluzioni adottate.

Tale linguaggio favorisce, inoltre, la **divulgazione delle informazioni**, in quanto standard internazionale non legato alle singole imprese. In teoria, un qualunque tecnico, di qualsivoglia nazionalità, dipendente della più ignota delle software house, con un minimo di conoscenza dell'UML dovrebbe essere in grado di leggere il modello di un progetto e di comprenderne ogni dettaglio senza troppa fatica e, soprattutto, senza le ambiguità tipiche del linguaggio naturale. Come al solito qualche problema può sempre emergere, ma si tratterebbe comunque di problemi di poca entità. Un conto è non comprendere qualche dettaglio, un altro è non comprendere assolutamente cosa voleva realizzare l'autore; è situazione tipica di alcuni progetti ribattezzati "temi da scuola superiore": si tratta di documenti tipicamente corposi — quando non si è sicuri è meglio scrivere tanto — in cui il 99% del contenuto è relativo alla descrizione, assolutamente non organica, di nozioni teoriche copiate da appositi libri e il restante 1% a embrioni di soluzioni.

I vantaggi che derivano dal poter disporre di un modello del sistema sono notevoli e fin troppo evidenti. Basti pensare alla non indifferente semplificazione del processo di manutenzione che, da solo, tipicamente incide per più del 50% nel ciclo di vita dei sistemi

software ben progettati; alla possibilità di allocare risorse aggiuntive in corso d'opera, riducendo il rischio che ciò diventi controproducente anche se spesso si tratta di un finto rimedio a cui si ricorre in prima istanza.

Disporre di un linguaggio per **descrivere un sistema** costringe il progettista stesso ad analizzare, con maggior minuzia, aspetti del sistema, anche di un certo rilievo, i quali, viceversa, potrebbero incautamente venir trascurati da un'analisi non molto rigorosa.

Per ciò che concerne l'utilizzo dello UML per **specificare**, bisogna tener presente che in questo contesto l'espressione si riferisce alla possibilità di realizzare modelli completi, precisi e non ambigui. Lo UML dispone di tutti i meccanismi necessari per la specifica di qualsiasi dettaglio ritenuto rilevante in ogni fase del ciclo di vita del software e quindi, in ultima analisi, per produrre modelli accurati.

Lo UML permette di **realizzare modelli** che si prestano ad essere implementati con diversi linguaggi di programmazione, sebbene risulti particolarmente efficace per la progettazione di sistemi Object Oriented. In effetti è possibile realizzare un mapping esplicito tra un modello UML e un linguaggio di programmazione. Chiaramente, tale legame risulta più immediato per i linguaggi fortemente basati sul paradigma Object Oriented, quali C++, Java, Small-Talk, Ada, ecc.

Sul mercato sono presenti diversi tool in grado di generare codice a partire dal relativo modello, sia interattivamente durante la fase di disegno, sia su richiesta. L'esistenza di queste funzionalità, sebbene ancora non del tutto mature — è solo questione di tempo — , dovrebbe far capire che l'implementazione è veramente un dettaglio del disegno, specie con linguaggi come Java.

La generazione automatica di una prima implementazione del sistema risulta particolarmente utile quando il modello deve essere realizzato parallelamente (cioè sempre!), poiché fornisce, ai diversi sviluppatori, lo scheletro — eventualmente con qualche metodo implementato — delle classi fondamentali del sistema che dovrebbero presentare un'interfaccia stabile e ben definita.

Il mapping tra modello e linguaggio di programmazione permette anche la realizzazione di funzioni di reverse engineering: fornendo a un opportuno tool i codici sorgenti o, talune volte anche quelli compilati, questo è in grado di ricostruire a ritroso il modello fino, ovviamente, alla fase di disegno. Purtroppo non si è ancora riusciti a realizzare un tool in grado di ricostruire i requisiti del cliente: un vero peccato!

Il processo diretto (*engineering*) e quello inverso (*reverse engineering*) determinano quello che in gergo viene definito *round-trip engineering*. Nel mondo ideale, la funzione di reverse dovrebbe essere utilizzata raramente… In quello reale è invece molto apprezzata, e tutto dipende dall'uso che se ne fa.

In fase di disegno, probabilmente non è opportuno disegnare tutto dettagliatamente; verosimilmente è opportuno lasciare qualche margine ai programmatori (tutto in funzione delle loro capacità). Sono ritenute assolutamente naturali e accettabili modifiche del

modello in fase di codifica, fintantoché queste non stravolgano il modello stesso. Durante la fase di codifica può anche accadere di accorgersi che una data libreria non funziona come dovrebbe, o che c'è qualche lacuna nel modello, o che risulta opportuno cambiare qualche strategia al fine di ottenere un codice più efficiente… Tutto ciò è normalissimo. Tuttavia, nell'eventualità che le modifiche generino uno stravolgimento del modello, piuttosto che procedere nell'implementazione sarebbe forse opportuno introdurre un'opportuna iterazione della fase di disegno e successiva codifica.

Per ciò che concerne l'utilizzo dello UML per la fase di **documentazione**, la tentazione di considerare il tutto fin troppo ovvio è forte; ma la famosa vocina dell'esperienza si fa sentire. Un progetto, per quanto ben congegnato, potrebbe perdere gran parte del suo fascino se poco documentato, o addirittura potrebbe finire per non essere compreso e in futuro non venire correttamente aggiornato.

Per terminare, lo UML fornisce sia dei meccanismi molto formali, sia del testo libero da aggiungere, ogni qual volta lo si ritenga necessario, a parti ritenute poco chiare o particolarmente complesse, al fine di aumentarne il livello di dettaglio.

Che cosa non è lo UML

Riportata la definizione formale del linguaggio, si ritiene opportuno ribadirne il concetto da un altro punto di vista: che cosa lo UML non è. Potrebbe sembrare ridondante e invece probabilmente è il caso del *melius abundare quam deficere.*

In primo luogo, sebbene ciò dispiaccia a molte persone, lo UML non è un linguaggio di programmazione visuale. Quando la tecnologia sarà matura, il modello di disegno verrà tradotto automaticamente in codice quasi completamente eseguibile tramite opportuni tool commerciali, ma questo è un discorso diverso. In ogni modo, chi ritiene che la modellazione sia inutile in quanto coincidente con la codifica sappia che lo UML non gli sarà di particolare aiuto. Il linguaggio di modellazione definisce un metamodello semantico ma non un tool di interfacciamento o di memorizzazione, e nemmeno un modello in grado di essere eseguito. La documentazione dello UML include molti suggerimenti utili alle aziende che si occupano di realizzare dei tool di supporto, ma non stabilisce ogni necessario particolare. Per esempio, non vengono presi in considerazione dettagli quali la selezione dei colori da utilizzare nella costruzione di diagrammi, la navigazione all'interno del modello, le modalità con cui memorizzare le informazioni, e così via.

Infine lo UML non è un processo di sviluppo, sebbene alcuni tecnici tendano a confondere i ruoli: lo UML è "solo" un linguaggio di modellazione e, in quanto tale, si presta a rappresentare i prodotti generati nelle varie fasi di cui un processo è composto. Pertanto lo UML, contrariamente ai processi, non fornisce alcuna direttiva su come fare evolvere il progetto, né attraverso quali fasi farlo transitare, né tantomeno su quali siano i manufatti da produrre, o su chi ne sia responsabile, ecc.

Metamodello e meta-metamodello

Definizioni di metamodello e meta-metamodello

Contrariamente a convinzioni comuni a molti tecnici, lo Unified Modeling Language non è unicamente una notazione standard per la descrizione di modelli Object Oriented di sistemi software; si tratta bensì di un metamodello definito rigorosamente che, a sua volta, è istanza di un meta-metamodello definito altrettanto formalmente. Si provvederà ora a illustrare brevemente i succitati concetti e le inerenti relazioni, senza avere la pretesa di fornirne una descrizione dettagliata, per la quale si rimanda a testi specializzati e in particolare al documento di specifica dell'OMG (BIB07). Ancora una volta si tratta di una materia così vasta che, probabilmente, meriterebbe un libro a sé stante. Ma l'argomento ha un'importanza e un fascino così elevato che sarebbe stato un vero peccato escluderlo dalla trattazione del presente libro.

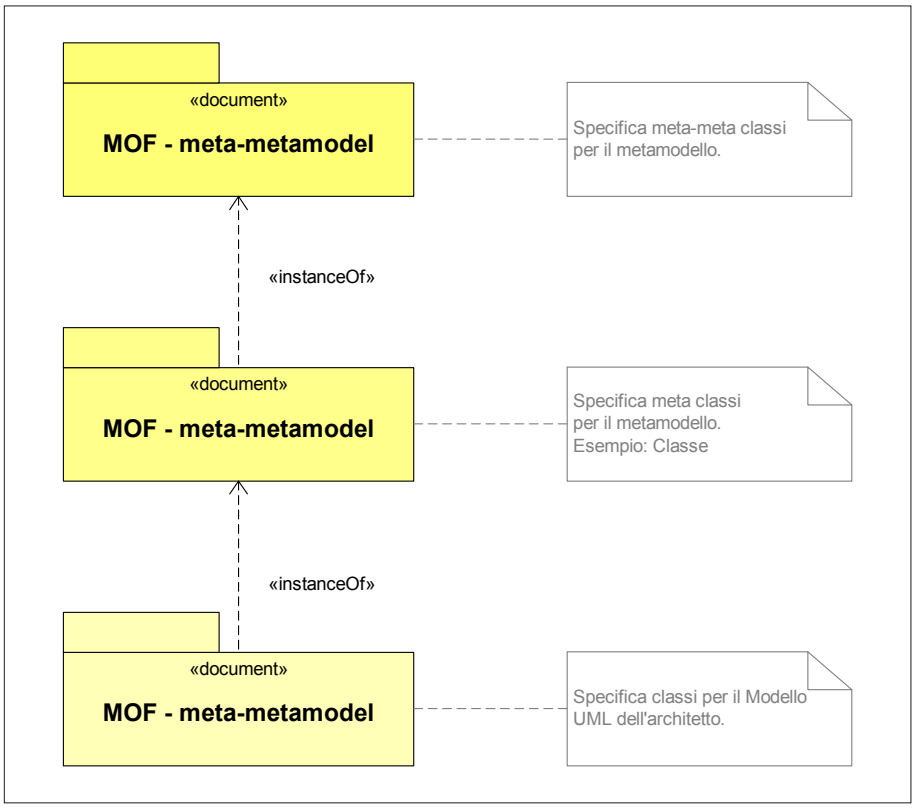
Un metamodello è un modello a sua volta istanza di un meta-metamodello, fruibile per esprimere una sua istanza di modello: l'UML (metamodello) permette di realizzare diversi modelli Object Oriented (modelli definiti dall'utente). Il metamodello dello UML definisce la struttura dei modelli UML. Un metamodello non fornisce alcuna regola su come esprimere concetti del mondo OO, quali ad esempio classi, interfacce, relazioni e così via, ma esso rende possibile avere diverse notazioni che si conformano al metamodello stesso.

Per esempio, in UML una classe viene rappresentata da un rettangolo suddiviso in tre sezioni. Un'altra notazione (in modo molto originale ma poco appropriato) potrebbe stabilire di rappresentare il concetto di classe per mezzo di un apposito pseudo-codice e di visualizzare le relazioni graficamente. Entrambe le notazioni, se definite rigorosamente, rappresentano istanze del metamodello, e sono quindi valide a tutti gli effetti. Un metamodello può specificare che una relazione di associazione deve avere due o più terminazioni di associazione, ma non prescrive assolutamente che essa vada rappresentata per mezzo di una linea che connetta due classi.

Così come avviene nella relazione che lega le classi agli oggetti — una volta definita una classe si possono avere svariate istanze della stessa (oggetti) — analogamente è possibile progettare un numero infinito di varianti dello UML (istanze del metamodello). Al fine di evitare un'ennesima proliferazione di dialetti — sembrerebbe che non appena si lasci un minimo spazio di azione, i generatori naturali di entropia tendano a prendere il sopravvento — lo OMG ha anche definito la notazione standard conforme al metamodello, comunemente denominata UML.

> A dire il vero le cose non sono proprio andate così; si è piuttosto utilizzata una metodologia Bottom-up: partendo da un oggetto concreto (lo UML) e procedendo per astrazioni successive, è stata definita la classe (il metamodello). Nelle sue prime apparizioni ufficiali lo UML era composto semplicemente da una notazione grafica, fruibile per rappresentare modelli di sistemi Object Oriented. Quando poi è giunto il momento della sottomissione allo OMG, per il

Figura 1.4 — *Meta-metamodello, metamodello e modello UML.*



riconoscimento ufficiale di standard (gennaio 1997), si è reso necessario conferirgli una veste più formale. Così a partire dalla versione 1.1 lo UML definisce rigorosamente un metamodello e la notazione atta alla formulazione di modelli Object Oriented conformi a esso.

Fin qui quasi tutto chiaro… Forse. Volendo però procedere oltre, i concetti diventano un po' più articolati. In effetti, così come in molti linguaggi di programmazione Object Oriented esiste il concetto di metaclasse (classe le cui istanze sono costituite da altre classi), anche il metamodello è un'istanza di un'entità di livello di astrazione superiore: il meta-metamodello.

Un meta-metamodello è un modello che definisce un linguaggio per esprimere un metamodello. Chiaro no? La relazione tra il meta-metamodello e il metamodello è paragonabile a quella esistente tra il metamodello e il modello. Lo UML è definito in termini di un meta-metamodello denominato MOF: Meta Object Facility.

Nella fig. 1.4 viene illustrato graficamente quanto emerso fino a questo punto: relazioni esistenti tra il meta-metamodello, il metamodello e il modello dell'utente. Scorrendo il diagramma dall'alto verso il basso si assiste a una graduale diminuzione del livello di astrazione: se si fosse deciso di visualizzare un ulteriore livello, si sarebbero trovate entità istanze della classe del modello dell'utente: oggetti.

Se per esempio si realizzasse un prototipo di un sistema di commercio elettronico, a livello del modello realizzato dall'architetto si troverebbero classi, opportunamente relazionate tra loro, del tipo `Categoria`, `SottoCategoria` (probabilmente l'organizzazione in categorie si presta ad essere rappresentata per mezzo del pattern Composite), `Prodotto`, `Utente`, `Profilo`, e così via. Se poi si volesse visualizzare l'ulteriore livello, l'istanza dello user model, bisognerebbe inserire oggetti istanze di tali classi, come per esempio il `Prodotto` di codice X, avente prezzo Y, della categoria Z, e così via.

Figura 1.5 — *Esempio delle relazioni ai vari gradi di astrazione tra i modelli UML.*

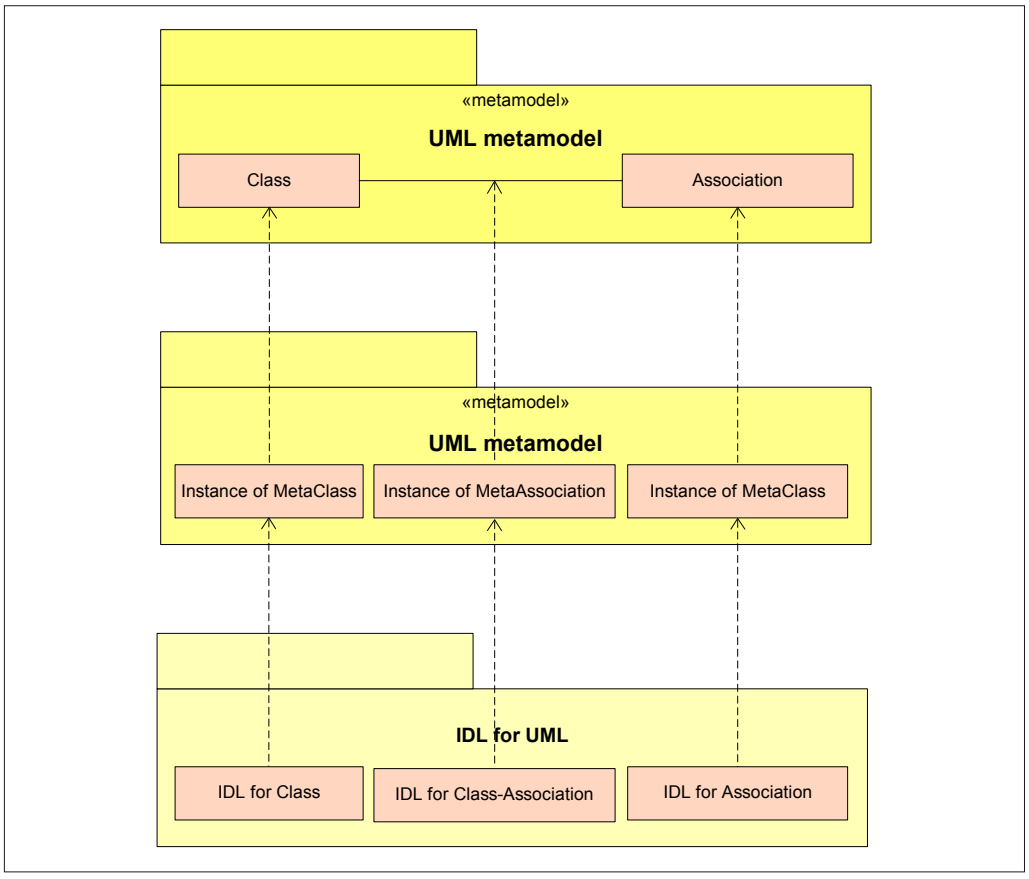


Tabella 1.1 — *La scala di astrazioni: dal meta-metamodello agli oggetti utente.*

Livello		Descrizione	Elementi
M3	*MOF Core*	Definizione degli oggetti utilizzati per specificare metamodelli.	MetaClassi MetaAttributi MetaAssociazioni Mof::Classi Mof::Attributi Mof::Associazioni
M2	*Metamodello*	Metamodello definito in termini degli elementi appartenenti al core del MOF.	UML: Classi, Attributi, Associazioni Data Warehousing: Base di Dati, Tabelle, ennuple,…
M1	*Modello*	Il modello è la descrizione delle informazioni del dominio del problema, effettuata attraverso lo UML. A dire il vero è molto di più…	UML classi: Cliente, Item, Ordine, … UML associazioni: Prodotti ordinati (Business Entities and processes)
M0	*Oggetti "utente"*	Istanze degli elementi del modello.	Cliente #00DNLMDR Cliente #01BRZRIO Item #00HSFRNT Item #01HITECH Ordine #0000003 Associazione tra Ordine #0000233 e Item #00HSFRNT

Per avere un'idea più chiara, si consultino la tab. 1.1 e il diagramma riportato nella fig. 1.5.

Si consideri il diagramma riportato in fig. 1.5. Come si può notare, nel modello di analisi compare una classe denominata `Ordine` appartenente al dominio del problema. La classe è un'istanza della metaclasse, presente nel package UML metamodello e contiene attributi ed operazioni che, a loro volta, sono istanze rispettivamente delle metaclassi `Attributi` e `Operazioni`. Se anche in questo caso si fosse deciso di visualizzare un ulteriore livello si sarebbero trovate le istanze delle classe `Ordine`, ossia oggetti del tipo: `00000312 : Ordine`.

Linguaggi e processi

A questo punto è opportuno chiarire la relazione che lega i linguaggi di modellazione, con particolare riferimento allo UML, ai metodi di sviluppo del software: spesso si tende a confonderne le relative funzioni.

Lo UML è un linguaggio di progettazione e, come tale, è "solo" parte di metodi più generali per lo sviluppo del software: lo UML è un formalismo utilizzato dai processi per realizzare, organizzare, documentare i prodotti generati dalle fasi di cui il processo si compone. Un metodo, tra i vari princìpi, è formato dalle direttive che indicano al progettista cosa fare, quando farlo, dove e perché. Un linguaggio invece è carente di questo aspetto. Va da sé che i processi svolgono un ruolo assolutamente fondamentale nella realizzazione di progetti di successo, e non solo in campo informatico. Talvolta, nonostante tutti i migliori prerequisiti, si assiste al fallimento di un progetto perché lo sviluppo non è stato guidato da un processo ben definito o perché non è stato ben gestito: spesso i processi costituiscono la differenza tra progetti iperproduttivi e progetti fallimentari.

Le varie ipotesi di processi di produzione del software, in ultima analisi, rappresentano tentativi di conciliare la produzione della "testa" (analisi e progetto) con quella delle "mani" (codifica e test).

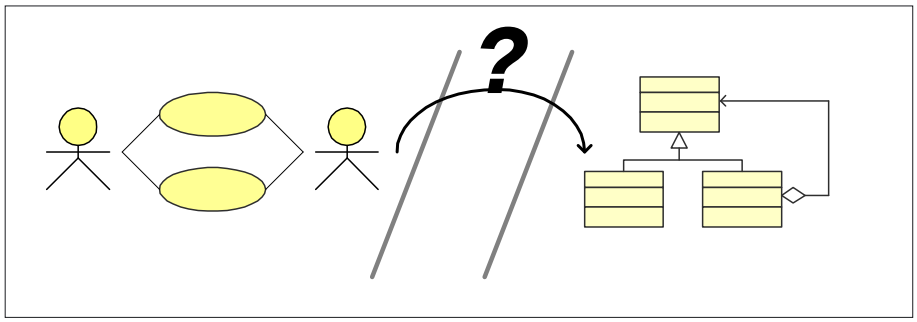
Non esiste il processo universalmente valido o che risulti ottimale per ogni progetto. Alcune volte è necessario conferire particolare attenzione ai requisiti del cliente, altre all'architettura da utilizzare, altre volte ancora è necessario tenere sotto controllo i rischi più rilevanti del progetto e così via. Il processo, per sua natura, deve essere rapportato all'organizzazione, alla relativa cultura tecnologica, al particolare dominio del problema, alle capacità del team, e così via. Non a caso i processi formali moderni sono disegnati in modo universale e prevedono un'attività propedeutica all'utilizzo che consiste nell'adattare i processi alle peculiari esigenze dello specifico progetto, dell'organizzazione, dell'area di business, ecc.

Un linguaggio è uno strumento — nel caso dello UML è un insieme di strumenti — da utilizzarsi per realizzare, descrivere e documentare i modelli prodotti nelle varie attività di cui si compone un modello. Un linguaggio è costituito da una sintassi, una semantica e dei paradigmi.

Nel linguaggio naturale la **sintassi** è costituita sia dalle norme che regolano il coordinamento delle parole (in UML, i simboli dei diagrammi) nelle proposizioni sia da quelle che specificano come organizzare tali proposizioni all'interno di un periodo (in UML, come combinare i simboli, quali possono essere associati e quali no, come, ecc.).

La **semantica** si occupa del significato delle parole (simboli), considerate sia singolarmente, sia nel contesto nel quale vengono utilizzate (significato dei simboli in un particolare diagramma). Infine, un particolare **paradigma** potrebbe contenere le direttive atte a conferire determinate forme, variabili a seconda dei casi, aggiungendo quindi chiarezza e precisione ad alcuni elementi della lingua (in UML, come realizzare modelli chiari, leggibili, eleganti, ecc.).

Figura 1.6 — *Il gap della progettazione del software.*



Il famoso gap: *mind the gap*

Nel libro *Use Case Driven Object Modeling with UML* [BIB05] viene proposta un'immagine, simile a quella in fig. 1.6, che di per sé chiarisce, più di molte parole, articoli e libri, uno dei principali problemi della progettazione: il salto del gap, il passaggio dalle specifiche del cliente alla realizzazione del modello di disegno del sistema (questi argomenti vengono trattati in maniera approfondita nel relativo capitolo).

Alcune persone confidano sul fatto che il problema sia passare dal disegno alla codifica: niente di meno vero! Tale passaggio è abbastanza diretto a patto di non avere i famosi programmatori "object disoriented". Il problema reale è esprimere in termini di componenti software sia il dominio del problema sia l'infrastruttura necessaria per sostenere il tutto. Per quanto concerne l'infrastruttura, le nuove architetture (come per esempio J2EE), cercano di semplificare sempre di più il problema della realizzazione dell'infrastruttura).

Chi scrive ha avuto modo di constatare quanto sia gravoso superare il famoso gap: sperimentandolo in prima persona attraverso la realizzazione di progetti Component Based di grandi dimensioni, attraverso il preziosissimo feedback maturato nell'ambito della collaborazione con la rivista web MokaByte e, infine, nell'ambito dell'insegnamento di UML. Il problema tipo riscontrato è che i neofiti, pur avendo compreso come funzioni lo UML, eventualmente con qualche lacuna, non lo sanno poi applicare alla pratica. Come si può ben intuire, ancora una volta il problema è strettamente connesso all'esperienza e al processo utilizzato, piuttosto che al linguaggio di per sé. Un buon supporto è rappresentato dai vari libri dedicati ai Design Pattern e ai processi.

È il caso di sottolineare che, così come conoscere un linguaggio non significa necessariamente essere bravi programmatori, conoscere lo UML non significa assolutamente essere in grado di produrre buoni disegni né tantomeno essere bravi architetti.

Un valido aiuto viene fornito dal processo il quale deve:

1. fornire le linee guida e il relativo ordine delle attività che il team deve svolgere;

2. specificare quali manufatti debbano essere prodotti in ogni fase;

3. fornire direttive allo sviluppo sia dei singoli programmatori sia del team;

4. offrire dei criteri per controllare e misurare i prodotti e le attività del progetto.

La trattazione di tale argomento necessiterebbe di un apposito testo, che evidentemente non è quello che state leggendo. Ciò nonostante nei seguenti paragrafi si tenterà di affrontarlo, seppur molto parzialmente, consigliando lo studio dei libri [BIB05] e [BIB06] a tutti coloro che desiderassero approfondire l'argomento.

Processi Use Case Driven

I sistemi software, in primo luogo, dovrebbero essere realizzati per soddisfare specifiche esigenze dell'utente. Logica conseguenza è che, per produrre un sistema il quale effettivamente realizzi gli obiettivi per il quale è stato finanziato, è indispensabile chiarire la prospettiva del cliente, le relative necessità, le aspirazioni più o meno recondite, ecc.: tutto ciò che comunemente viene indicato con "specifiche utente". Occorre inoltre tenere bene a mente queste ultime durante tutto il ciclo di vita del software: l'utente non è sempre l'idiota che inserisce i dati nelle caselle di testo e successivamente preme il tasto `OK`. I team tecnici, tipicamente, tendono a non provare sufficiente rispetto per le esigenze degli utenti finali.

Come si vedrà nel Cpitolo 3, dedicato ai casi d'uso, non sempre è così semplice tenere conto di tutte le specifiche utente, per una serie di motivi: difficoltà di stabilire una piattaforma di intesa comune, background e prospettive diverse, esigenze contrastanti, ecc. Focalizzare, fin dalle primissime fasi del ciclo di vita del software l'aderenza dei vari modelli alle richieste del cliente, continuando a monitorarle, può risultare una strategia vincente.

È da tener presente che non sempre gli attori con cui interagisce il sistema sono operatori umani; spesso si tratta di un altro sistema o di un qualsiasi dispositivo fisico. In generale si tratta di un'entità esterna al sistema che interagisce con esso. Se, per esempio, si volesse realizzare un sistema di elaborazione automatica e smistamento di documentazione in formato digitale, il quale, a partire da modelli di documenti riesca a completarne il contenuto, richiedendo ai vari enti le informazioni mancanti (dati anagrafici, penali, pensionistici, ecc.), i vari attori del sistema sarebbero, oltre che diversi operatori, altri enti in grado di produrre, completare, ricevere e inoltrare file di documentazione. Se si volesse poi rendere possibile memorizzare tali documenti, corredati da opportune firme digitali in apposite smart card di proprietà degli utenti, anche queste risulterebbero attori.

Tipicamente un sistema, a fronte di un'iterazione avviata da un utente, risponde eseguendo una sequenza di azioni in grado di fornire all'attore stesso i risultati desiderati: produzione del documento, completamento dello stesso con l'aggiunta dei dati richiesti,

inoltro verso opportuno destinatario, e così via. Tali interazioni costituiscono quello che viene denominato "caso d'uso" (Use Case), si tratta cioè di una parte delle funzionalità del sistema che fornisce all'utente determinati servizi.

La sommatoria di tutti i casi d'uso costituisce una proiezione dello Use Case Model, il quale, oltre a specificare quali siano le funzionalità che il sistema deve realizzare — e fin qui non ci sono grosse novità rispetto ai metodi tradizionali, eccezion fatta per il formalismo grafico —, mette in rilievo gli attori coinvolti e le relative interazioni con il sistema. Quindi, ancora una volta, gli Use Case risultano particolarmente focalizzati sul ruolo degli attori.

La centralità degli attori dovrebbe spingere il progettista a pensare il sistema non solo in termini delle funzionalità da realizzare, ma anche in termini del valore da fornire agli utenti. Dunque, i casi d'uso non vengono utilizzati unicamente come formalismo per la specifica formale dei requisiti utente, ma come valido supporto alle altri fasi, ossia disegno, implementazione e test. In altre parole essi guidano l'intero processo di sviluppo. Logica conseguenza è che i prodotti generati nelle varie fasi sono modelli che realizzano, in qualche misura, i casi d'uso. Per esempio, i programmatori dovrebbero continuamente rivedere il codice, al fine di verificarne l'aderenza con quanto sancito nei casi d'uso codificati, mentre i tester dovrebbero provare le funzionalità del codice, per accertarsi che il sistema realizzi effettivamente quanto sancito nelle specifiche.

Sintetizzando, l'intero ciclo di vita ruota attorno alla Use Case View, vale a dire che i casi d'uso sono progettati, determinano gran parte del modello di disegno, vengono tradotti in codice e infine, nei test di sistema, si verifica che il sistema realizzi quanto stabilito. Nel paragrafo dedicato al processo ICONIX, viene fornita un'istanza di processo Use Case Driven.

Processi Architecture Centric

Il ruolo dell'architettura di un sistema software può essere, per molti versi, ricondotto a quello svolto dall'architettura nelle costruzioni civili. L'edificio viene studiato da parte del team di progettisti da diversi punti di vista: struttura, sicurezza, servizi, tubature, riscaldamenti, e così via, in modo che se ne abbia una visione completa prima di avviare la costruzione effettiva: perla di saggezza!

Allo stesso modo l'architettura di un sistema software è descritta per mezzo di diverse viste prima che venga iniziata la codifica del sistema. I relativi concetti comprendono gli aspetti più significativi della proiezione statica e dinamica del sistema. L'architettura, in ultima analisi, dovrebbe venire progettata con l'intento di fornire l'infrastruttura necessaria per il soddisfacimento dei requisiti utente e, come tale, dovrebbe essere funzionale alla Use Case View.

L'architettura, chiaramente, è influenzata da molti altri fattori tra i quali: la piattaforma sulla quale il sistema dovrà funzionare (architettura di rete, sistemi operativi presenti, ge-

stori di basi di dati, modalità di comunicazioni tra sistemi esistenti, …); riusabilità di determinati componenti (framework, interfacce utente, …); considerazioni di "dispiegamento" (adattamento dei componenti sull'architettura fisica); presenza di legacy system; requisiti non funzionali (prestazioni, robustezza, efficiente utilizzo delle risorse, …), e così via.

L'architettura è la vista del disegno nella sua globalità, grazie alla quale si evidenziano gli aspetti di maggiore importanza, mentre gli altri dettagli vengono lasciati appositamente fuori. La valutazione di quali aspetti siano più interessanti e quali, invece, lo siano meno è un metro soggettivo e come tale dipende dall'esperienza dei singoli. Comunque ciascun metodo di sviluppo offre linee guida come comprensibilità, flessibilità, capacità di adattamento a futuri cambiamenti, riusabilità, e così via.

Processi iterativi e incrementali

Lo sviluppo di un sistema software è un processo che può richiedere mesi o anni, mentre la relativa manutenzione, se il software si rivela di successo, può richiedere un ordine di grandezza superiore.

Queste considerazioni di carattere empirico hanno finito inevitabilmente per fornire un prezioso feedback ai processi di sviluppo: l'intero processo viene infatti suddiviso in diversi miniprogetti, ognuno dei quali è un'iterazione che produce un incremento significativo del progetto globale.

Le varie iterazioni non dovrebbero essere frutto del caso o dell'umore degli architetti, dovrebbero bensì essere debitamente pianificate e controllate, vale a dire che dovrebbero essere veri e propri miniprogetti completi. La scelta di cosa realizzare a ogni iterazione dovrebbe dipendere da due fattori:

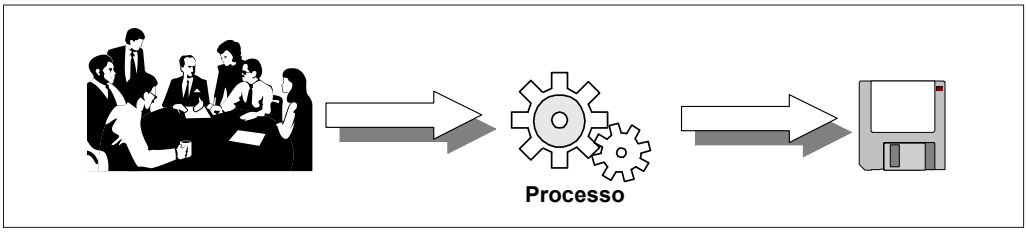
- selezione del gruppo di use case in grado di estendere ulteriormente le funzionalità del sistema, considerando il punto di evoluzione raggiunto — o da raggiungere — nell'iterazione precedente;

- rischi più pressanti.

Giacché ogni iterazione è un miniprogetto, questa dovrebbe includere tutte le fasi relative: analisi, disegno, implementazione e test. Non sempre un'iterazione produce un aumento quantitativo del modello; ma spesso tale aumento si può risolvere nella sostituzione di parti di disegno/implementazione con altre più dettagliate e/o più sofisticate (iterazioni di refactoring).

Infatti può accadere di realizzare velocemente parti del disegno, e relativa implementazione, senza curarne troppo la struttura interna, rimandando quindi la relativa reingegnerizzazione a ulteriori iterazioni. Ciò si rivela particolarmente utile quando tali sottosistemi forniscono l'infrastruttura o le funzionalità utilizzate da altri. Può infatti

Figura 1.7 — *Schematizzazione di un processo software.*



risultare conveniente realizzare prime versioni, seppur grezze, di un particolare sottosistema, al fine di dare modo alle altre parti dipendenti di poter evolvere liberamente. L'intento è quello di reingengerizzare tali versioni prototipali in un secondo momento, disponendo di maggior calma e migliori informazioni (magari applicando opportuni criteri, come quelli specificati nel libro *Refactoring* di Fowler [BIB06]).

Ancora una volta è possibile ricorrere a tale approccio attraverso l'utilizzo intelligente delle interfacce. Si potrebbe, per esempio, progettare e realizzare una prima versione del sistema di sicurezza — magari in grado di fornire risposte costanti — senza che esso realizzi alcun meccanismo specifico, solo per fornire agli altri sottosistemi la possibilità di poter eseguire dei test.

Nella realizzazione dei casi d'uso ritenuti fondamentali per l'iterazione in atto (Use Case Driven), è necessario utilizzare come guida anche l'architettura stabilita (Architecture Centric), e quindi realizzare il disegno in termini di componenti compatibili con essa. Al termine di ciascuna iterazione è indispensabile verificare che essa abbia effettivamente raggiunto gli scopi prefissati, ossia che la nuova versione del sistema realizzi correttamente i casi d'uso selezionati. In caso affermativo si può procedere con l'iterazione successiva, altrimenti è necessario introdurre una fase di revisione, nella quale, può rendersi necessaria anche una completa revisione dell'approccio utilizzato in precedenza.

Per avere migliori possibilità di riuscita, un progetto dovrebbe seguire l'iter pianificato, pur con minime deviazioni dovute a imprevisti, i quali dovrebbero comunque sempre essere tenuti in considerazione all'atto della pianificazione del processo stesso. I vantaggi nell'utilizzo di metodi di sviluppo iterativo e incrementale sono:

- riduzione dei costi dovuti a errori o imprevisti generatisi durante una singola iterazione. Se per qualche motivo si rende necessaria l'introduzione di una nuova iterazione, l'organizzazione impegnata nello sviluppo viene penalizzata "solo" per il relativo contesto senza però intaccare l'intero prodotto.

- riduzione del rischio di non produrre il sistema nei tempi previsti. Identificare fin dall'inizio i rischi maggiori, e assegnare un congruo periodo di tempo per la realiz-

zazione della relativa soluzione, dovrebbe consentire alle varie persone coinvolte di trattare il rischio con la dovuta calma, evitando paradossalmente di aggravare la situazione per via di eventuali stati d'ansia dovuti a potenziali ritardi. Seguendo i metodi tradizionali è comune osservare come nel momento in cui un un problema viene evidenziato si generi l'affanno di risolverlo, temendo di produrre un ritardo nei tempi di consegna. Come direbbe Murphy: "se esiste la possibilità che una cosa vada male, sicuramente andrà male!".

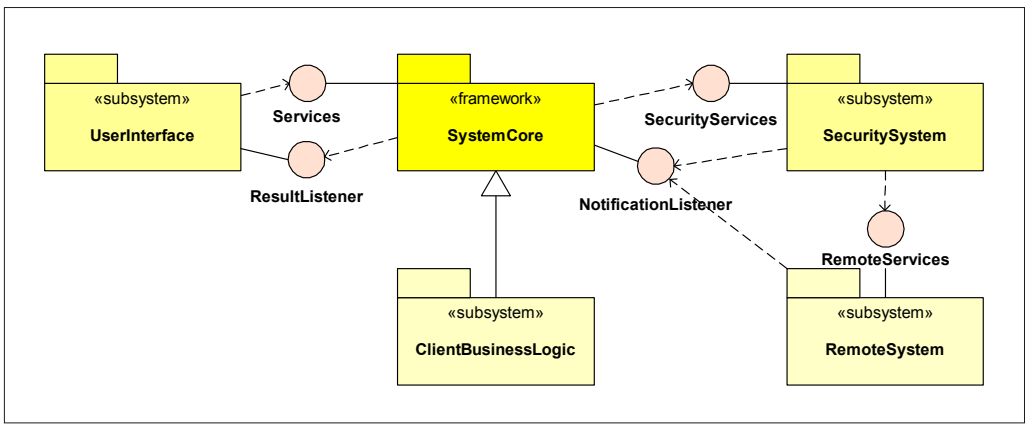
- riduzione del tempo necessario per rilasciare il sistema globale. Ciò dovrebbe essere il risultato di un lavoro più razionale e dominato da obiettivi più immediati e ben riconoscibili.

- maggiore flessibilità in caso di modifiche dei requisiti utente. Nella pratica si verifica che molto raramente si riescano a identificare sin da subito e correttamente le specifiche del cliente, anche per via di sue inequivocabili responsabilità. Paradossalmente la comprensione di esse, nella loro globalità, si raggiunge a progetto ultimato. La suddivisione in iterazioni e la realizzazione di insiemi ben identificati di casi d'uso, rende più facile la possibilità di verificare il sistema in maniera incrementale e di evidenziarne eventuali anomalie.

The Unified Software Development Process

Nei prossimi paragrafi viene presentato, sin troppo brevemente, il processo di sviluppo del software, ideato dai Tres Amigos presso la Rational e denominato "processo unificato

Figura 1.8 — *Esempio di suddivisione del sistema in sottosistemi.*



di sviluppo software" (Unified Software Development Process). Quanto riportato è tratto dal libro [BIB08] al quale si rimanda per una trattazione più approfondita dell'argomento così indispensabile per la produzione di sistemi software di elevata qualità.

L'utilizzo consapevole di un buon processo può, da solo, produrre la differenza tra un progetto fallimentare e uno altamente produttivo, sebbene in molte organizzazioni la produzione del software sia ancora basata su processi ideati ai tempi delle schede perforate oppure standardizzati da personale commerciale o addirittura inesistenti o demandata alla responsabilità e all'esperienza dei singoli (leggasi inesistente). Chiaramente si tratta di una condizione necessaria, ma non sufficiente. Un processo di sviluppo software è costituito dal complesso di attività necessarie per trasformare i requisiti utente in un sistema software.

Lo Unified Software Development Process, in realtà, non è semplicemente un processo bensì un *framework* di un processo generico. Esso si presta a essere proficuamente utilizzato per classi piuttosto estese di sistemi software, differenti aree di applicazioni, diverse tipologie di organizzazioni, svariati livelli di competenza e progetti di varie dimensioni.

Una delle linee-guida del processo unificato è di essere component-based: il sistema da realizzare viene basato su componenti e interconnesioni fra i componenti stessi, fondate su opportune e ben definite interfacce. In questo contesto, per "componente" si intende una parte fisica e sostituibile di un sistema che si conforma e fornisce la realizzazione di un insieme ben definito di interfacce. Queste ultime sono collezioni di operazioni, utilizzate per specificare un servizio offerto da una classe o da un componente.

Vista l'importanza e l'eleganza di questi concetti e la loro centralità nella produzione di sistemi Object Oriented, si è deciso di trattare tali argomenti in dettaglio nel capitolo appropriato, ossia quello dedicato ai diagrammi delle classi (Capitolo 8). Per ora basti sapere che tale approccio si basa sulla suddivisione del sistema globale in diversi sottositemi, comunicanti per mezzo di interfacce che, in questo contesto, assumono la valenza di un contratto. Un sottosistema si impegna a fornire i servizi esposti nelle relative interfacce, mentre un altro le utilizza impegnandosi a rispettarne le condizioni.

Il processo unificato, ovviamente, utilizza lo UML per produrre e documentare i manufatti prodotti in ciascuna delle fasi di cui si compone. Altra caratteristica peculiare, offerta dal processo, è la fusione razionale dei tre principali metodi esistenti: Use Case Driven, Architecture Centric e Iterative and Incremental. Forse ciò non solo lo rende unico ma, verosimilmente, anche raro da utilizzare nella sua versione originale. Se da una parte, infatti, l'insufficienza cronica di tempo non può e non deve giustificare la produzione di sistemi software non guidati da un processo e tantomeno può giustificarne la produzione incentrata sulla sola fase di codifica, dall'altra bisogna ammettere che il problema tempo esiste ed è molto sentito. Forse, se è possibile avanzare una critica al processo unificato, si può dire che esso necessita di una notevole quantità di tempo, non sempre disponibile, per poter essere attuato così come proposto.

Integrazione tra Use Case Driven e Architecture Centric

Ogni prodotto possiede delle funzionalità e una forma fisica. Il problema sta nel trovare il giusto equilibrio per riuscire a generare un valido prodotto. Nel caso di sistemi software le funzionalità vengono modellate per mezzo della vista dei casi d'uso mentre la forma è rappresentata dall'architettura.

Tra le due viste esiste una forte interdipendenza: il sistema che realizza la Use Case View deve conformarsi all'architettura fisica, la quale, a sua volta, deve fornire lo spazio necessario al soddisfacimento dei requisiti utente, sia quelli attuali, sia quelli futuri… Le due viste dovrebbero evolvere parallelamente attraverso una successione di sincronizzazioni. L'architetto, nel conferire la forma al sistema (progettazione dell'architettura), deve prestare attenzione sia a fornire un appropriato ambiente per la versione iniziale, sia a consentire a eventuali versioni future di adattarsi senza grandi problemi.

Per individuare la forma migliore, è necessario essere in grado di comprendere le funzionalità che il sistema dovrà fornire; chiaramente non è necessario conoscerle perfettamente tutte. In particolare l'architetto dovrebbe:

- generare un disegno iniziale di architettura che non sia specificamente connessa ai requisiti utente (*use case independent*), sebbene ne debba possedere una certa conoscenza. Ciò è conseguibile iniziando a tener presenti i vincoli dettati dalla piattaforma, dal network che si utilizzerà, e così via. Tipicamente si ragiona in termini di pattern architetturali o, più semplicemente, si cerca di ricondursi ad architetture — o a opportuni segmenti di esse — dimostratesi efficaci in precedenti progetti;

- a questo punto è necessario lavorare con forte aderenza ai casi d'uso. Vengono selezionati quelli ritenuti più importanti o significativi e, di ciascuno di essi, vengono forniti i dettagli in termini di sottosistemi, classi e componenti;

- non appena i casi d'uso sono stati ben specificati e presentano una certa stabilità, si procede con una verifica dell'architettura che a sua volta potrebbe fornire nuovi elementi per migliorare gli stessi casi d'uso (come per esempio requisiti non fattibili oppure servizi espletabili utilizzando un approccio diverso, ecc.).

Tale processo viene iterato fin quando anche l'architettura viene considerata stabile.

Ciclo di vita del processo

Lo Unified Software Development Process integra le tre principali strategie di sviluppo tra cui l'Iterative and Incremental; pertanto la realizzazione del sistema avviene attraverso cicli di iterazioni controllate, ognuna delle quali termina con la produzione di una nuova versione del sistema.

Ogni ciclo è costituito da quattro fasi: iniziale, elaborazione, costruzione e transizione, ognuna delle quali è, a sua volta, suddivisa in iterazioni. Risultato di ogni ciclo è la produzione di una nuova versione del sistema virtualmente pronta per la consegna. Grosso vantaggio di questo criterio è che anche il processo di verifica è iterativo e incrementale: ogni volta che una nuova versione è disponibile viene sottoposta a relativa fase di test. Ciò dovrebbe favorire la produzione di sistemi di migliore qualità.

Nel mondo dell'ideale, ogni versione dovrebbe consistere in codice incapsulato in opportuni componenti eseguibili, corredati da manuali, documentazione e così via. Sebbene dal punto di vista del cliente i componenti eseguibili siano i manufatti più importanti, essi da soli non sono sufficienti. Ciò è dovuto al fatto che l'ambiente tipicamente è destinato a variare: espansione dei sistemi hardware, aggiornamenti di sistemi operativi e database manager, ecc.

Gli stessi requisiti cliente diventano più chiari e precisi con il procedere nel ciclo di vita del software e pertanto sono soggetti anch'essi ad aggiornamenti. Non a caso, un'idea che spesso si ingenera nei team di sviluppo a consegna avvenuta, è di essere in grado di poter rifare l'intero sistema più adeguatamente e in minor tempo. I prodotti che devono accompagnare la versione finale sono:

- il modello dei casi d'uso, con evidenziate tutte le relazioni tra i diversi use case e i relativi attori e gli scenari;

- il modello di analisi che dovrebbe conseguire, principalmente, due obiettivi: *a*. definire i casi d'uso in maggior dettaglio; *b*. orientare un iniziale disegno del comportamento del sistema per mezzo dell'individuazione degli oggetti basilari che lo caratterizzano;

- il modello di disegno che definisce: *a*. la struttura statica del sistema in termini di sottosistemi, classi, interfacce; *b*. la realizzazione dei casi d'uso per mezzo di collaborazioni attraverso i sottosistemi, le classi , ecc. evidenziati nella proiezione statica al punto precedente;

- il modello di implementazione che include i componenti (raggruppamento di codice) e il mapping degli stessi con le relative classi;

- il modello di dispiegamento (*deployment*) che definisce come i componenti software vadano allocati sui nodi fisici (computer, server, reti, ecc.) previsti dall'architettura hardware;

- il modello di test il quale definisce i casi di test (Test Case) che verificano gli Use Case;

- la rappresentazione dell'architettura.

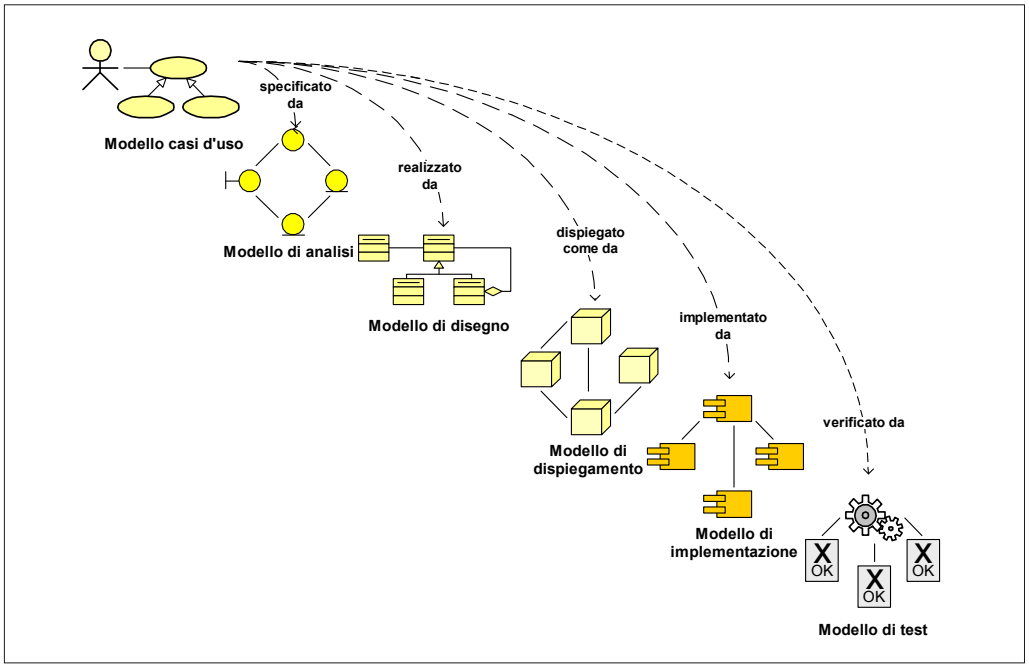
La totalità di questi manufatti rappresenta il sistema nel suo insieme.

Ogni ciclo necessità di un'appropriata porzione di tempo che viene suddivisa in quattro fasi che sono: iniziale, elaborazione, costruzione e transizione. Ciascuna fase può a sua volta essere ulteriormente suddivisa e termina con il raggiungimento del *milestone* pianificato: realizzazione dei manufatti programmati.

I *milestone*s sono importanti per diverse ragioni. In primo luogo permettono ai manager di prendere cruciali decisioni prima di considerare conclusa una fase e passare alla successiva. Successivamente risultano utili anche per gli sviluppatori in quanto permettono di monitorare la progressione delle attività assegnate. Infine, tenendo traccia del tempo speso e delle problematiche emerse in ciascuna fase, è possibile raccogliere tutta una serie di informazioni da utilizzarsi nella pianificazione dei progetti futuri.

Nella fase iniziale di ogni ciclo (detta *inception*) si ipotizzano diverse soluzioni e l'idea migliore viene sviluppata in una visione del prodotto della quale vengono forniti gli scenari di utilizzo. Obiettivo della fase di inception è essenzialmente fornire risposte adeguate agli interrogativi riportati di seguito:

Figura 1.9 — *I modelli previsti dallo Unified Process.*



- quali sono le funzioni basilari che il sistema deve fornire per ciascuno dei principali attori?

- qual è il modello di massima dell'architettura necessaria al sistema?

- qual è il piano e quanto dovrebbe costare lo sviluppo del prodotto?

Il primo quesito trova risposta in un modello semplificato degli Use Case che ne contempli i più critici. Per ciò che concerne il secondo punto va tenuto presente che nella prima fase l'architettura è poco più di un'ipotesi da sviluppare in dettaglio nelle fasi successive, pertanto è sufficiente che contenga i principali sottosistemi. Molto importante invece è riuscire ad identificare prima possibile i rischi principali, assegnare loro la giusta priorità ed elaborare la pianificazione della fase in dettaglio.

La seconda fase, nota come elaborativa (*elaboration*), prevede la specifica di dettaglio di tutti i casi d'uso e il disegno dell'architettura, la quale riveste un ruolo di primaria importanza e viene espressa per mezzo delle viste fornite da tutti i modelli del sistema. Ciò equivale a dire che esistono viste architetturali del modello: dei casi d'uso, di analisi, di disegno, di implementazione e di deployment. La vista del modello di implementazione include componenti per dimostrare che l'architettura è eseguibile.

In questa fase i casi d'uso più critici, identificati nella fase precedente, vengono realizzati. Risultato di questa fase è l'Architectural Baseline, ovvero un insieme rivisto e approvato di manufatti che:

- rappresentano una base stabilita per futuri sviluppi ed evoluzioni;

- possono essere modificati solo attraverso una procedura formale di configurazione o di cambio di gestione.

Al termine della fase elaborativa il capo progetto si trova nelle condizioni di poter pianificare le attività e di stimare le risorse necessarie per completare l'intero progetto. I pericoli di questa fase possono essere adeguatamente sintetizzati dalle seguenti inquietudini: i casi d'uso, l'architettura e i vari piani sono sufficientemente stabili? I rischi sono abbastanza sotto controllo perché lo sviluppo dell'intero progetto in contratto sia in condizione di procedere?

Terminata la seconda fase si passa a quella di costruzione (*construction*), in cui, come è lecito aspettarsi, il prodotto viene costruito. Nel corso dell'evoluzione del prodotto l'architettura di base cresce all'aumentare dei dettagli del sistema e il prodotto evolve fino ad essere pronto per la trasformazione per gli utenti finali. Durante questa fase le risorse precedentemente ipotizzate vengono impiegate. L'architettura del sistema assume una

sua stabilità anche grazie al lavoro degli sviluppatori che in corso d'opera ne verificano la validità ed eventualmente ne suggeriscono opportune varianti. Terminata la fase di costruzione il prodotto contiene tutti i casi d'uso concordati per la versione generata dalla fase in corso. Come pratica insegna, raramente si tratterà di una versione senza difetti: questi sono oggetto di indagine nella fase successiva.

Il milestone della fase di costruzione si raggiunge quando si è certi che il prodotto risolva sufficientemente le necessità di qualche utente tanto da poterne consegnare la versione per il test.

L'ultima fase, di transizione (*transition*), comprende il periodo necessario al prodotto per divenire una versione beta. Un ristretto numero di utenti prova la versione e compila un documento con l'elenco di difetti e deficienze. Questo documento viene quindi fornito a un gruppo di sviluppatori che ricontrolla il prodotto e provvede a risolvere i problemi esposti. A questo punto il prodotto è pronto a essere somministrato a un gruppo di prova più ampio.

La fase di transizione dovrebbe prevedere attività quali la manutenzione del prodotto per accogliere eventuali suggerimenti, l' addestramento del personale, la fornitura di assistenza, e così via. I suggerimenti ricevuti, tipicamente, vengono suddivisi in due categorie: quelli urgenti che quindi necessitano di essere risolti il prima possibile e quelli da demandare alle fasi successive (i famosi *must*, *should*, *could*).

RUP: Rational Unified Process

La Rational ha presentato un processo di sviluppo, denominato RUP (Rational Unified Process) che, lentamente, ambisce a divenire lo standard di mercato per la produzione del software. Il principale fautore è Philippe Kruchten, Lead Architect della Rational. La prima versione fu rilasciata a dicembre 1998 frutto, verosimilmente, della rielaborazione di quello presentato dai Tres Amigos (illustrato nel paragrafo precedente) e del processo Objectory (frutto della medesima organizzazione di Jacobson). Al momento in cui viene scritto il presente libro è disponibile la versione 5.5. Brevemente i vantaggi offerti dal RUP sono:

- è basato su princìpi di ingegneria del software dimostratisi efficaci: approccio Iterative and Incremental, Use Case Driven, Architecture Centric;

- si tratta di un processo proprietario e verosimilmente leader del mercato, pertanto è frutto di continue evoluzioni grazie agli investimenti della Rational;

- prevede una serie di meccanismi, come per esempio i prototipi funzionanti da realizzare alla fine di ogni interazione del ciclo, i punti di decisione (go/no go) che permettono di stabilire, alla fine di ogni fase, se abortire il processo: chiaramente se il processo è destinato a fallire è meglio rendersene conto e quindi abortirlo il prima possibile;

- include un sistema semplificato di agevolazione dell'utilizzo grazie alla descrizione basata su tecnologie web.

Gli svantaggi sono:

- è molto focalizzato nel processo di sviluppo e trascura altri flussi importanti nello sviluppo di sistemi software (come spiegato di seguito);
- essere iterativo e incrementale è un vantaggio ma uno svantaggio al tempo stesso: richiede tutta una serie di attività supplementari dovute alle varie iterazioni;
- non si integra perfettamente in organizzazioni, quali le software house, che, tipicamente, realizzano più progetti contemporaneamente;
- essendo un prodotto proprietario è guidato dalle esigenze di mercato, quindi aspetti meno pressanti sono trascurati (tool per il disegno dell'interfaccia utente, per la modellazione dei dati, ecc.).

> Del RUP è disponibile una versione "migliorata" elaborata presso la Ronin International Inc. il cui fautore principale è Scott Ambler — autore, tra l'altro, di testi come *Building Object Applications that Work* (1998) e *Process Patterns* (1999) — che ne è il presidente. Tale versione, denominata EUP (Enhanced Unified Process, processo unificato miglioraro) è animata dall'obiettivo di perfezionare il RUP aggiungendo flussi mancanti ed espandendo opportunamente, qualora necessario, quelli esistenti. Per esempio è presente una quinta fase (oltre Inception, Elaboration, Construction e Transiction) denominata Production (produzione). Si tratta di un'unica iterazione atta a gestire le attività necessarie per mantenere il sistema operativo fino a quando sia disponibile una nuova versione o, addirittura, il sistema venga rimpiazzato da uno completamente nuovo. Un'altra innovazione è data dalla presenza di un workflow denominato Operations and Supports che, come indicato dal nome, è designato a gestire operazioni di sviluppo e piani di supporto. Per esempio classiche attività di supporto necessarie a sistema rilasciato sono le attività di backup, l'esecuzione di specifici lavori batch, ecc.

La Rational supporta e tenta di divulgare il processo RUP fornendo un apposito sistema software. Non si tratta di un applicativo vero e proprio bensì di una base di informazioni messe a disposizione dell'utente: una sorta di sito web dello Unified Software Development Process, corredato da tutta una serie di consigli, linee guida, best practice, e così via. Per il software di supporto al processo di sviluppo che tutti sognano, probabilmente bisogna attendere ancora un po'…

A detta della stessa casa produttrice, il Rational Unified Process è un software di supporto al processo di sviluppo, fornito di un sistema di conoscenza fruibile con tecnologia

web based (si tratta di un vero e proprio sito web), in grado di incrementare il livello di produttività dei team e quindi ridurre i tempi necessari per la consegna del software grazie alle linee guida fornite.

Il processo RUP è basato essenzialmente sui tre concetti: *workers* (lavoratori), *artifacts* (manufatti) e *activities* (attività) illustrate da appositi *workflow*.

Un **worker** definisce il comportamento e le responsabilità di un individuo o di un team in termini del ruolo ricoperto e quindi le relative attività possono essere svolte da un singolo individuo o da un team. I lavoratori sono suddivisi in categorie quali analisti, sviluppatori, tester, manager, ecc. che a loro volta sono suddivisi in ruoli. Tipicamente la categoria degli sviluppatori include architetti, disegnatori, programmatori e integratori di sistemi.

Ai vari lavoratori sono affidate diverse **attività**: unità di lavoro che devono espletare nel contesto del progetto. La granularità delle attività può spaziare dall'ordine delle ore a quello dei giorni: un'eccessiva minuziosità richiede una lunga e laboriosa programmazione che spesso può risolversi in un inutile spreco di energie, mentre una programmazione superficiale può condurre a una errata stima dei tempi. Come al solito si tratta di individuare il giusto mezzo.

Ciascuna attività è decomposta in tre fasi: *thinking* (elaborazione mentale), *performing* (esecuzione) e *reviewing* (revisione) i cui obiettivi sono ben comprensibili dai nomi stessi. Il sistema fornisce tutta una serie di linee guida, consigli e tecniche pratiche al fine di aiutare i lavoratori a conseguire le relative attività. Le azioni da svolgere per implementare le varie attività sono descritte in termini dei prodotti (*artifact*) di input e di quelli da produrre in output. Per esempio, le linee guida fornite dal sistema (RUP) a un programmatore che deve implementare un componente, consistono nell'eseguire le seguenti attività:

- codificare le operazioni;
- implementare gli stati;
- utilizzare la delegazione al fine di riutilizzare il codice;
- realizzare le associazioni;
- codificare gli attributi;
- fornire un opportuno feedback al disegno del modello;
- valutare il codice.

Al fine di espletare le attività assegnategli, ciascun lavoratore necessita di informazioni. Per esempio, per produrre un componente, uno sviluppatore necessita di un disegno con

le specifiche. La realizzazione di un'attività produce un qualche tipo di risultato, come le classi prodotte dallo sviluppatore. Le informazioni di input e i risultati prodotti sono comunemente definiti manufatti (*artifact*). Alcuni esempi di manufatti sono: il modello dei casi d'uso, il modello di analisi, il modello di disegno, il modello di implementazione, il piano dei test, il glossario, il documento di analisi dei rischi, il documento architetturale e così via.

Storicamente il ciclo di vita del software è organizzato a cascata: il progetto passa attraverso una successione di fasi (*workflow*) a partire dall'analisi del dominio del problema procedendo via via per le fasi di analisi, disegno, implementazione, test e consegna. Nella pratica, i workflow si sono dimostrati non ottimali in quanto eventuali lacune o problemi presenti in una fase emergono tipicamente solo alla fine, durante i test.

La sistemazione di tali anomalie può richiedere costosissime e non sempre risolutive iterazioni del workflow stesso, magari perché per colmare una lacuna bisognerebbe scardinare le fondamenta del modello di disegno.

I processi più recenti prevedono un approccio iterativo e incrementale: ogni iterazione perfeziona ed espande il sistema che via via assume una forma sempre più completa. In sostanza ogni iterazione prevede lo svolgimento dell'intero workflow.

Le prime iterazioni sono decisamente incentrate sull'analisi del dominio del problema e sulla cattura dei requisiti utente, mentre le ultime risultano molto implementative. Man mano che il progetto matura, si assiste a una graduale diminuzione delle attività di analisi e progetto a favore dell'espansione delle attività di sviluppo e test.

Dal punto di vista della gestione, il progetto può essere suddiviso nelle quattro fasi esaminate nel paragrafo dello Unified Software Developmente Process: inception, elaboration, construction e trasition. Ogni workflow è costituito dalle seguenti parti:

- **introduzione**: illustra gli obiettivi del workflow stesso e le relative relazioni con gli altri;
- **spiegazione** dei concetti necessari per comprendere il workflow;
- **descrizione** di dettaglio del workflow;
- **overview delle attività**: viene fornita la lista delle attività da svolgere corredata dai lavoratori coinvolti;
- **overview dei manufatti**: elenco dei manufatti da produrre corredato dall'attribuzione delle responsabilità;
- **overview delle linee guida**: spiegazioni e consigli su come produrre e utilizzare i manufatti coinvolti nel processo in corso.

Pertanto ogni iterazione del workflow descrive lavoratori, manufatti e attività coinvolte nell'iterazione del ciclo di vita del software. Il sistema prevede diverse sezioni, tra le quali le classiche best practice, white paper — chissà come mai il nome di questa sezione finisce per rievocare inesorabilmente memorie delle straordinarie interpretazioni di Totò — form, tool mentor, …

La sezione dedicata ai modelli (*forms*) è particolarmente utile in quanto, fornendo template in molteplici formati per tutta una serie di attività, permette di risparmiare molto tempo. La sezione denominata *tool mentor* fornisce una serie di consigli e istruzioni su come utilizzare al meglio il software Rational Rose durante tutta la fase di sviluppo del sistema. Ovviamente non sono presenti consigli su come realizzare i vari modelli o come organizzare viste, classi, ecc. ma unicamente quali funzioni utilizzare, dove muovere il mouse e quale pulsante premere.

In conclusione si può dire che questo sistema rappresenta una valida risorsa per tutti coloro che desiderano sviluppare seriamente sistemi software Object Oriented. In ultima analisi, si tratta di una fruizione ipertestuale della rivisitazione del libro dei Tres Amigos corredata di qualche riferimento al software Rose.

ICONIX: un processo Use Case Driven

In questo paragrafo viene introdotto brevemente il processo di sviluppo Use Case Driven denominato ICONIX Unified Object Modeling. Per uno studio più completo, si rimanda il lettore a [BIB05] e [BIB06].

I processi di sviluppo del software basati sulla vista dei casi d'uso sono probabilmente i più popolari nelle aziende di informatica. In ultima analisi, sono quelli utilizzati più o meno formalmente da sempre: si tratta di conferire la massima importanza ai requisiti funzionali di un sistema e ciò da sempre ha costituito un criterio guida nello sviluppo del software.

Poiché il ciclo di vita viene completamente basato sulla vista dei casi d'uso, logica conseguenza è che la use case view assume un'importanza e una criticità eccessive: un errore nella formulazione dei requisiti potrebbe determinare il fallimento dell'intero processo. Si ricordi che la difficoltà di comunicazione tra clienti e personale tecnico e la laboriosità nel tentare di definire una piattaforma comune hanno generato l'insuccesso di molti progetti. Spesso accade che certi team siano riusciti a realizzare progetti fantastici con tecnologie ultramoderne ma con l'unico neo, affatto trascurabile, di non risolvere minimamente i problemi del cliente.

Per ridimensionare il più possibile tale rischio, il processo viene frequentemente associato ad altri e spesso si utilizza un approccio incrementale e iterativo: lo sviluppo richiede diverse iterazioni del modello del dominio al fine di identificare ed analizzare completamente i vari casi d'uso. Altre iterazioni avvengono durante l'intero ciclo di vita del software tra le viste di cui si compone. Lo stesso modello statico viene definito incrementalmente a seguito delle successive iterazioni attraverso il modello dinamico.

Rappresentando le fondamenta del processo di sviluppo, la Use Case View necessita di un'eccezionale attenzione; in particolare è necessario stabilire con assoluta precisione ed esattezza:

- chi sono gli attori del sistema e quali interazioni esercitano con il sistema (Use Case View);
- quali sono gli oggetti del dominio del problema (l'ambiente che si sta automatizzando) e quali relazioni esistono tra di essi (diagramma concettuale delle classi);
- quali oggetti sono coinvolti in ogni caso d'uso;
- come gli oggetti collaborano all'interno di uno Use Case (diagrammi di interazione e collaborazione);
- come gestire le problematiche di controllo del mondo reale (diagrammi di stato);
- come costruire fisicamente il sistema (diagramma di dettaglio delle classi);

Come si vedrà in seguito, lo UML fornisce tutti i formalismi necessari per modellare quanto sancito nei punti precedenti nel contesto di un'unica notazione.

Il processo Use Case Driven come ulteriore vantaggio offre un elevato livello di "tracciabilità". Con ciò si intende dire che durante tutto il ciclo di vita è sempre possibile risalire, più o meno direttamente, ai requisiti del cliente. Così si può realizzare il disegno del sistema senza venire meno alle necessità dell'utente, è sempre possibile seguire come gli oggetti evolvono dai casi d'uso all'analisi fino al disegno, è più immediato verificare l'impatto delle richieste di cambiamento dei requisiti, ecc.

Il processo di sviluppo guidato dai casi d'uso proposto nel libro *Use Case Driven Object Modeling with UML* (*cfr.* [BIB05]) prevede quattro "pietre miliari" (*milestones*):

1. riesame dei requisiti utente;
2. revisione del disegno preliminare;
3. verifica del disegno di dettaglio;
4. consegna.

Durante la **prima fase** è necessario identificare gli oggetti del dominio del problema e le relazioni che intercorrono tra di essi. Tale primissimo processo dovrebbe concludersi

con la realizzazione di diagrammi delle classi di alto livello. Laddove i tempi e la complessità del sistema lo permettano, un forte valore aggiunto potrebbe venire dalla realizzazione di un rapidissimo prototipo. La riesamina dello stesso alla presenza del cliente fornisce di solito un'ottima piattaforma di argomentazione. Ulteriore vantaggio è che si riesce a placare temporaneamente il cliente facendolo trastullare con il nuovo giocattolo…

A dire il vero i prototipi possono risultare anche rischiosissimi: può capitare che, da qualche capo progetto avventuriero, esso venga fatto passare come prodotto ultimato, per la gioia di tutto il team di sviluppo; credete che sia solo un rischio e che non sia già capitato? Oppure — il che sostanzialmente si riconduce al caso precedente — può succedere che il cliente sviluppi l'idea che il passaggio dalla fase prototipale a quella definitiva avvenga con accelerazione infinita, ossia tempo zero.

Può accadere che alcune menti semplici pensino che per edificare la propria casetta ci vogliano pochi giorni o settimane, dal momento che, per immaginarsela impiegano pochi minuti. Poi discutono con architetti e muratori e tornano rapidamente sulla terra: si rassegnano ad aspettare il tempo necessario. Nel software, tanto per cambiare, le cose funzionano in modo leggermente diverso: le menti semplici immaginano, per esempio, complicati sistemi di e-commerce e credono di averli già tra le mani, poi discutono con chi il software lo deve realizzare e, strano ma vero, restano comunque della propria opinione.

A questo proposito si ritiene opportuno tributare una meritatissima lode a uno dei padri dell'Object Oriented: l'eccelso Bjarne Stroustrup. Nel suo libro *C++*, capitolo "Progetto e sviluppo", nel contesto "Sperimentazione e analisi" egli esprime un concetto molto interessante: afferma che all'inizio di un progetto importante è semplicemente *impossibile* ipotizzare con successo i dettagli della sua realizzazione; pertanto occorre sperimentare per acquisire quell'esperienza che, riguardo alla problematica, ancora non si possiede. Il prototipo è uno dei principali strumenti di sperimentazione — sicuramente uno dei più utili — ma ha il terribile difetto di somigliare troppo alla soluzione finale. Le menti semplici, già messe a dura prova dalla discrepanza tempi mentali / tempi fisici, non riescono a distinguere il dito dalla luna: ma se un cliente può essere opportunamente edotto riguardo la reale natura di ciò che vede, nulla si può fare quando l'assurdità si annida nella testa dei cosiddetti coordinatori, manager o capi progetto. In tal caso, l'unico consiglio da dare ai malcapitati sviluppatori coinvolti è quello di abbandonare la nave al più presto, prima che l'infausta marea li travolga. Al riguardo, è possibile citare Fowler: "Se non puoi cambiare la tua azienda, cambia azienda".

Sempre durante la prima fase è essenziale identificare i casi d'uso utilizzando i relativi diagrammi o appositi schemi e organizzarli in gruppi. Risultato di ciò è la produzione di primi diagrammi dei componenti. Per terminare la prima fase e quindi raggiungere il milestone della riesamina dei requisiti del cliente, è necessario allocare i requisiti funzionali agli oggetti del dominio del problema e ai relativi casi d'uso.

La prima attività del **secondo milestone** prevede la redazione della descrizione di dettaglio dei casi d'uso emersi nella fase precedente. In particolare, per ogni caso d'uso, è

necessario descrivere lo scenario relativo al caso ideale, quello in cui tutto funziona bene (mainstream) e non si verifica alcuna anomalia, e quelli relativi ai casi meno frequenti e alla gestione delle situazioni di errore che possono verificarsi. Durante questa fase si procede verso l'interno del sistema.

A questo punto è necessario eseguire la famosa analisi di robustezza. Per ogni caso d'uso è necessario individuare tutti gli oggetti che permettono di realizzare i vari scenari e aggiornare il diagramma delle classi del dominio del problema con quelle nuove, gli attributi e i metodi, a mano a mano che vengono individuati. Durante questa fase si procede dall'interno del sistema verso il suo esterno utilizzando un approccio Top Down: si aumenta il livello di dettaglio per mezzo di iterazioni successive. La seconda pietra miliare viene raggiunta con l'aggiornamento del diagramma delle classi al fine di adeguarlo ai concetti emersi nelle attività della seconda fase.

La **terza fase** prevede l'allocazione del comportamento. Per ogni caso d'uso si identificano i messaggi scambiati dai vari oggetti e i metodi da invocare. A tal fine risulta conveniente utilizzare i diagrammi di interazione Sequence o Collaboration. Si tratta di diagrammi equivalenti; quello che cambia è l'aspetto al quale si conferisce maggior risalto: l'iterazione temporale nel primo, la collaborazione tra oggetti nel secondo. Nel tracciare i diagrammi, è necessario continuare ad aggiornare il class diagram a mano a mano che vengono introdotti nuovi concetti.

Terminata anche questa fase occorre completare il modello statico aggiungendo le informazioni dettagliate di disegno, eventualmente ricorrendo ai design pattern. Per raggiungere anche questa milestone bisogna procedere con la verifica del disegno: è indispensabile accertarsi che soddisfi i requisiti precedentemente identificati.

Giunti qui, non resta che concentrarsi sulla **quarta** e ultima **fase**: il delivery. In primo luogo è necessario produrre i diagrammi di implementazione: componenti e dispiegamento. Quindi si passa alla codifica che dovrebbe risultare abbastanza automatica. Si eseguono i test di unità e di integrazione. Si esegue il famoso test di accettazione: utilizzando i casi d'uso e agendo sul sistema con una strategia a scatola nera, si verifica che il sistema faccia effettivamente quello per cui è stato finanziato.

Terminata questa fase il progetto è concluso e quindi si passa alla relativa manutenzione che, nel caso in cui il sistema si riveli di successo, occuperà circa il 50% dell'intero ciclo di vita.

XP (eXtreme Programming): tutto e il contrario di tutto

Viene qui presentato un particolare processo di sviluppo del software, di recente formulazione e in continua evoluzione, incentrato sulla fase di codifica: il teatro dell'assurdo.

Si tratta di camminare sulla lama del rasoio. La comunità informatica — e lo stesso team che ha collaborato alla realizzazione del presente libro — è percorsa da vivaci dibattiti tra sostenitori dell'XP e coloro che invece lo considerano come un vero e proprio anti-processo.

Onde evitare di essere tacciato di ignavia, l'autore dichiara onestamente di aver sempre osservato l'XP con non poche riserve mentali, riserve poi attenuate dall'aumento di formalità apportato grazie alle collaborazioni di personaggi del calibro di Martin Fowler e Erich Gamma (uno della "combriccola dei quattro", Gang Of Four [BIB04]).

Prima di fornire qualche dettaglio sull'eXtreme Programming (XP), si consiglia a tutti i lettori che stessero casomai leggendo in piedi questo libro di sedersi comodamente e di rilassarsi: in questo paragrafo vengono azzerati diversi sforzi prodotti finora nel tentativo di conferire il giusto rilievo alla fasi di analisi e disegno.

Il problema principale è che di questo metodo si tende a fare abuso con estrema facilità. Esso finisce involontariamente per fornire giustificazioni a tutta una serie di gruppi "modello-repellenti". Spesso, collaborando con varie aziende e implorando i diversi team di poter prendere visione dei modelli prodotti, ci si sente rispondere che non sono stati realizzati in quanto il sistema è stato sviluppato con un approccio di tipo XP e che, eventualmente, alcuni diagrammi delle classi verranno prodotti per mezzo di reverse engineering… Come se questi fossero gli unici "manufatti" necessari.

Team "furbetti" di questo tipo trascurano qualche particolare: a differenza dei processi "model free" (presenti in molte aziende), l'XP attribuisce elevata importanza agli unit test (test di unità): prima di scrivere il codice effettivo è assolutamente necessaria la scrittura dei relativi casi di test (quindi, se si utilizzasse veramente un approccio di tipo XP, come minimo dovrebbe essere disponibile una serie di package di test).

La realizzazione di test di unità equivale a dire che il comportamento viene analizzato e "modellato" a priori, però sempre attraverso codice: il che non è assolutamente vietato, ma forse non privo di tutta una serie di svantaggi. Per esempio si ritiene complicato sottoporre il modello ad altre persone, magari non esperte del particolare linguaggio, al fine di ricevere qualche considerazione; risulta più complicato considerare i modelli stessi come risorse preziose per il futuro: si pensi a Use Case o modelli di analisi e disegno presenti in organizzazioni bancarie che rappresentano veri e propri investimenti indipendenti dal linguaggio utilizzato per l'implementazione, di un valore elevatissimo per futuri sviluppi e reingegnerizzazioni.

Chiaramente non si può accusare uno strumento di cattivo funzionamento se lo si utilizza per fini non contemplati; come dire che non si può attribuire all'Aspirina alcuna colpa se la si utilizza per curare gastriti: del resto, non si tratta della cura per tutte le malattie; sicuramente è efficace per tutta una serie di patologie ma in altri casi non solo non produce alcun effetto, ma rischia anzi di aggravare la situazione.

Con ciò si intende dire che ogni prodotto, e quindi anche i processi, deve essere utilizzato per i fini per il quale è stato ideato: l'XP può dare buoni risultati in contesti ben definiti: progetti di dimensioni medio-piccole, team di buona qualità, persone in grado di capire i limiti e le virtù del processo stesso, e così via. Si provi invece a pensare di utilizzare l'XP per sviluppare un complesso sistema bancario…

L'idea di fondo che legittima l'intero processo è l'insufficienza cronica di tempo tipica dei progetti: invece di non seguire assolutamente un processo, il che rischierebbe di produrre risultati assolutamente non desiderabili, probabilmente è più opportuno seguirne uno "intuitivo", leggero, pragmatico e quindi molto operativo. Ciò può risultare particolarmente utile lavorando a progetti con cicli di consegna molto compressi e con fattori di rischio tecnologico molto elevati: indagini e sperimentazioni condotte codificando possono rivelarsi ottimi espedienti.

L'autore ritiene preferibile, anche in casi estremi, applicare versioni *light* di processi come il RUP; ma va riconosciuto che l'XP può risultare un ottimo strumento a supporto di persone intelligenti, preparate e frustrate dai soliti manager sempre convinti di saper far girare correttamente gli ingranaggi del sistema di produzione del software, magari perché riescono a consegnare qualcosa di eseguibile ai vari clienti.

Uno dei punti deboli dell'XP è che il team deve comprendere persone di talento, cooperative, dotate di buon affiatamento, coordinate da un Team Leader dotato di esperienza e capacità personali non del tutto comuni. Per le persone meno brave del team si prospettano due alternative: elevare rapidamente la propria esperienza o uscirne fuori a gambe levate.

Un altro punto da menzionare è che l'XP non porta a produrre tutta una serie di modelli estremamente importanti per l'organizzazione in cui il sistema funzionerà. Infatti, sebbene l'obiettivo del ciclo di vita del software sia produrre un sistema "eseguibile", i processi del tipo RUP permettono di realizzare, oltre al sistema stesso, tutta una serie di manufatti di estremo interesse, come per esempio il modello del business. Esso può essere utilizzato per aggiornare il sistema, per future reingegnerizzazioni, per insegnare a nuovi dipendenti l'area di business, ecc. Per capirne l'importanza, basti pensare che mentre la tecnologia tende a rinnovarsi molto rapidamente, non altrettanto vale per il business, e quindi un buon modello può risultare valido per decine di anni.

Nel processo XP, per così dire *code centric*, tutte le fasi del ciclo di vita del software vengono compresse a favore dell'attività di codifica: l'evoluzione delle API del sistema si acquisisce leggendo direttamente il codice, il comportamento di oggetti complessi viene definito per mezzo della codifica di appositi casi di test, gli inconvenienti vengono mostrati attraverso i problemi emersi dall'esecuzione dei casi di test, e così via.

In tale ottica tutto il processo di sviluppo si risolve in una continua reingegnerizzazione del software e quindi ecco che le tecniche di refactoring illustrate da Fowler [BIB06] risultano un validissimo strumento di lavoro. Questo approccio può risultare particolarmente utile nel realizzare opportuni Framework: la collezione di classi via via prodotte viene immediatamente verificata ed eventuali lacune vengono alla luce rapidamente. Il processo non rifiuta completamente la fase di disegno, ma ne prevede l'utilizzo in casi estremi, quando non sia possibile codificare: in tutti gli altri casi l'attività di implementazione ha la precedenza.

Da quanto emerso risulta chiaro che un processo code centric è applicabile quando si dispone di team di dimensioni medio-piccole formati da personale particolarmente esperto e il progetto da affrontare è anch'esso di dimensioni piuttosto contenute. Si tratta comunque di processi in grado di rendere felici taluni manager i quali ogni settimana, o poco più, si vedono consegnare nuove masse di codice.

XP viene presentato come un processo leggero, efficiente, a basso rischio, flessibile, scientifico con un approccio divertente allo sviluppo del software. Quest'ultima qualità è sicuramente la più allettante: a tutti i tecnici piace divertirsi con nuovi giocattoli. Certamente è un metodo coraggioso — e per questo merita di essere considerato con il dovuto rispetto — che fa e farà discutere. Tale riflessione ha già generato dei risultati positivi: uno dei meriti indiscussi dell'XP è quello di aver assestato un forte scossone ai processi di sviluppo accademici tradizionali, arroccati in un'eccessiva burocrazia e con un esagerato grado di formalismo, costringendoli a rinnovarsi e a rendersi più duttili.

L'autore nutre diversi dubbi sull'efficacia nello sviluppo parallelo. Anche trovandosi nel caso più ideale possibile (requisiti utente ben chiari, team di elevata qualità tecnica, ecc.) il dubbio è che lo sviluppo parallelo del software possa risultare piuttosto macchinoso in quanto, evidentemente, nessun elemento del team può sapere in anticipo cosa produrranno gli altri: non c'è il modello di disegno. Ciò, in ultima analisi, dovrebbe rendere difficile l'integrazione dei vari sottositemi prodotti da elementi diversi del team. Chiaramente il processo richiede personale intelligente in grado di capire che le aree di collegamento tra i vari sottosistemi andrebbero realizzate per prime e non prevede che il personale debba chiudersi in una stanza fino a sviluppo terminato; però la macchinosità del processo potrebbe risultare notevole.

Ulteriore considerazione è che si tratta di un processo di produzione un po' "cieco": non si tenta in alcun modo di prevedere e affrontare i problemi che si potrebbero incontrare. Come dire, si comincia a costruire la strada, prima di edificarne un tratto si producono i vari test, ogni elemento del personale ne costruisce un pezzo, magari qualche kilometro più a est o più a ovest, magari comunicando frequentemente per informarsi vicendevolmente circa il punto geografico in cui ognuno sta lavorando, e così via. Però… che succede se poi alla fine ci si trova di fronte a una montagna o a un altro elemento geografico che complica il raccordo delle strade?

I restanti processi

Uno dei processi nati sull'onda dell'innovazione generata dall'XP è l'Agile Modeling ideato da Scott W. Ambler (`www.agile-modeling.com`). Si tratta di un processo che tenta di riutilizzare quanto di positivo emerso grazie all'eXtreme Programming, riportando finalmente l'attenzione dalla fase di programmazione a quella di modellazione: un metodo decisamente più vicino alla *forma mentis* che sta alla base del presente libro… Sarà per la presenza della magica parolina *modeling* nel nome?

Sia chiaro che l'autore non ripudia assolutamente la programmazione, che tante gioie gli ha dato e continua a dargli: chi scrive è anzi convinto che spesso qualche sperimentazione programmativa (specie quando si gioca con nuovi "giocattoli") prima di procedere con l'attività di modellazione possa far risparmiare molto tempo. Si dubita invece dell'approccio diametralmente opposto, come suggerito dall'XP: ricorrere alla modellazione solo nei casi in cui non sia possibile codificare.

I propositi dell'Agile Modeling sono:

- definire come mettere in pratica una collezione di valori, princìpi e pratiche per la modellazione del software applicabile in progetti per lo sviluppo di sistemi software in modo efficace e leggero. Il segreto non sono le tecniche di modellazione in sé stesse quanto piuttosto il modo in cui metterle in pratica;

- determinare come il processo possa essere utilizzato agevolmente ed efficacemente in progetti di sviluppo di sistemi software utilizzando anche criteri provenienti dall'eXtreme Programming;

- determinare come il processo possa essere applicato seguendo le direttive e i modelli previsti dello Unified Process.

Altre informazioni sono reperibili nel sito web `http://www.agilemodeling.com/`.

Un altro processo degno di considerazione è l'OPEN Process. Il nome del processo già di per sé dovrebbe fornire diverse indicazioni circa i relativi obiettivi e gli artefici dello stesso. In effetti, si tratta di un processo ideato presso l'"OPEN Consortium" (consorzio aperto), che raggruppa un insieme di persone e di organizzazioni animate dall'obiettivo comune di accrescere e migliorare l'utilizzo della tecnologia Object Oriented. Dati i presupposti, è evidente che i destinatari principali dell'Open Process sono le organizzazioni interessate nello sviluppo di sistemi Object Oriented o Component Based.

L'Open Process è indipendente dai particolari linguaggi di modellazione, e pertanto qualsiasi linguaggio in grado di rappresentare modelli Object Oriented è idoneo, quantunque alla fine gli unici utilizzabili siano lo UML e l'OML (Object Modelling Language). L'OML è un linguaggio di modellazione ideato anch'esso dall'OPEN Consortium, che a lungo si è opposto allo UML nella corsa alla conquista della standardizzazione. Chiaramente si trattava di una competizione dall'esito scontato: opponeva allo UML, linguaggio sostenuto dagli investimenti di un'azienda che su di esso ha basato il proprio successo, lo OML affidato alla buona volontà delle persone che hanno collaborato al progetto.

Il ciclo di vita del sistema software è denominato *contract driven*, in quanto ciascuna attività è vincolata a contratti dichiarati, che quindi finiscono per rendere il processo guidato dalle responsabilità (*responsibility driven*).

Uno dei pregi del progetto è la sua capacità di adattarsi a organizzazioni che, generalmente, sviluppano più progetti contemporaneamente: esempio tipico sono le software house.

Il processo, in prima analisi, può essere scisso in due grosse categorie di attività: quelle da svolgersi per il singolo progetto e quelle trasversali, ossia comuni a tutti i progetti. Le attività appartenenti alla seconda categoria supportano la "gestione del programma" (Programme Management), dove per programma si intende un insieme di progetti e/o versioni.

Come gli altri processi anche l'Open si presta a essere adattato alle esigenze delle varie organizzazioni e progetti. I punti di forza di questo processo sono:

- copre quasi tutti gli aspetti della produzione ingegneristica del software;
- è uno standard aperto a cui partecipano liberamente persone di notevole esperienza e capacità.

A dire il vero il secondo punto è un vantaggio/svantaggio. È un pro in quanto permette al processo di evolvere indipendentemente dalle strategie di mercato, e quindi per esempio gli permette di prevedere strumenti per problemi considerati di minore importanza (disegno della GUI per esempio); però al tempo stesso è un punto a sfavore in quanto, non essendo sostenuto da un'azienda investitrice, verosimilmente è destinato a condividere la sorte dell'OML.

Un ultimo processo da menzionare è Object-Oriented Software Process (OOSP, processo di sviluppo software object oriented). Verosimilmente l'iniziale elaborazione è attribuibile a James Coplien (*A Generative Development-Process Pattern Language* e *Pattern Languages of Program Design*) sebbene i recenti miglioramenti e successi siano stati congegnati da Scott Ambler (*Process Patterns* e *More Process Patterns*).

Il processo OOSP è basato sul concetto di pattern di processi: ossia soluzioni dimostratesi valide nella soluzione di problemi tipici presenti nello sviluppo di sistemi software. Ciascuno di essi è costituito da una collezione di tecniche, azioni e attività che permettono di risolvere problemi specifici del processo software prendendo in giusta considerazione fattori ed eventuali vincoli presenti. Tali processi operano su tre diversi livelli di astrazione: fase (*phase*), stadio (*stage*) e attività (*task*).

L'OOPS è basato, essenzialmente, su quattro fasi:

1. iniziale (*initiate*) composta dagli stadi: giustificazione (*justify*), definizione e validazione dei requisiti iniziali (*define and validate initial requirements*), definizione iniziale dei documenti di gestione (*define initial management documents*) e definizione infrastruttura (*define infrastructure*);

2. costruzione (*construct*) formata dagli stadi: modellazione (*model*), verifiche in piccolo (*test in the small*), generalizzazione (*generalize*) e programmazione (*program*);

3. consegna (*deliver*) costituita dagli stadi: verifiche in larga scala (*test in large*), rielaborazione (*rework*), rilascio (*release*) e assestamento (*assest*);

4. mantenimento e supporto (*maintain and support*) che come sancito dal nome è modellata dagli stadi di supporto e identificazione dei difetti e migliorie (*indetify defects and enhancements*).

Anche l'OOSP è un processo iterativo in cui le iterazioni avvengono nel contesto delle fasi, che vengono eseguite in serie. Come l'open process, l'OOSP prevede una serie di attività da svolgersi attraverso i progetti, ossia a livello di "programma". Queste attività prevedono controllo qualità, gestione del progetto, formazione del personale, gestione dei rischi, gestione del personale, ecc. Chiaramente molte attività richiedono di essere svolte sia nell'ambito del singolo progetto sia in quello, più generale, di tutti i progetti attivi presso l'organizzazione. Per esempio è importante effettuare la valutazione (e gestione) dei fattori di rischio dei progetti sia nel dominio dei singoli, sia nella loro totalità: un insieme di progetti relativamente rischiosi potrebbe generare un rischio molto elevato a livello di organizzazione. I vantaggi del processo sono:

1. operare a livello di singolo progetto e a quello di portafoglio di progetti;

2. considerare tutti i processi necessari per lo sviluppo, la consegna e il mantenimento in esercizio dei sistemi prodotti;

3. includere tecniche dimostratesi valide nella realizzazione dei processi e nella segnalazione di condizioni che potrebbero generare effetti indesiderati.

I tool

Tool UML

L'intento predominante che ha guidato tutto il lungo processo di stesura del presente libro è stato quello di tentare di suscitare, nel maggior numero possibile di lettori non ancora espertissimi dello UML, un qualche interesse nel disegnare modelli di sistemi OO utilizzando approcci formali, ma snelli, allo sviluppo del software.

Sia chiaro che l'importante è progettare un sistema, usando poi un qualsivoglia formalismo, ma visto e considerato che esiste uno standard universalmente accettato, elegante, ben provato, perché non utilizzarlo?

A questo punto il passo successivo consiste nel reperire un tool commerciale da utilizzarsi come ausilio nella modellazione di sistemi software… Fino a qualche tempo fa non c'erano molte alternative: nelle varie aziende di informatica era possibile trovare quasi esclusivamente Rational Rose o TogetherJ. Ultimamente si è assistito a tutto un proliferare di tool che ammontano ora a decine. Nella tab. 1.1 si riportano alcuni dei principali strumenti.

Tabella 1.1 — *Alcuni tra i tool UML per la modellazione di sistemu software.*

Argo/UML	http://argouml.tigris.org/
Enterprise Architect	http://www.sparxsystems.com.au
Ideogramic UML	http://www.ideogramic.com/products/
iUML	http://www.kc.com/products/iuml/index.html
JVision	http://www.object-insight.com/html/product_info.html
MagicDraw UML	http://www.nomagic.com/magicdrawuml/
MetaEdit+	http://www.metacase.com/
Model Prototyper	http://www.objexion.com
ModelMaker	http://www.modelmaker.demon.nl/
mUML	http://www.mfcomputers.com/
Object Domain	http://www.objectdomain.com/domain/
Object Engineering Workbench	http://www.isg.de
Objecteering UML Modeler	http://www.objecteering.com/us/produits.htm
Poseidon for UML	http://www.gentleware.com
Rational Rose	http://www.rational.com/rose/
Real-time Studio	http://www.artisansw.com/products/products.asp
Rhapsody	http://www.ilogix.com
SoftModeler	http://www.softera.com/
StP/UML	http://www.aonix.com/content/products/stp/uml.html
StructureBuilder	http://www.tendril.com/
System Architect	http://www.popkin.com/products/product_overview.htm
Together	http://www.togethersoft.com/
UML2COM	http://www.arion.gr/uml2com/
Visual Case	http://www.visualcase.com/
Visual Paradigm for UML	http://www.visual-paradigm.com
Visual UML	http://www.visualuml.com/

Ricapitolando…

Il presente capitolo è stato dedicato all'illustrazione di che cosa sia e che cosa non sia lo UML e di quali siano le relative aree di applicabilità. L'intento è stato quello di chiarire il prima possibile quali siano gli obiettivi, le aree di interesse e i confini dello UML: nella comunità Object Oriented non sono infrequenti opinioni poco chiare che conferiscono allo UML funzionalità tipiche dei processi di sviluppo del software o che, in maniera diametralmente opposta, lo riducono a semplice linguaggio per la rappresentazione dell'organizzazione statica del software.

Secondo le specifiche formali, lo UML è un linguaggio per specificare, costruire, visualizzare e documentare manufatti sia di sistemi software, sia di processi produttivi e altri sistemi non strettamente software. L'UML rappresenta una collezione di *best practices* di ingegneria dimostratesi vincenti nella modellazione di vasti e complessi sistemi.

Il capitolo si apre con una digressione sul concetto di modello e sulle proprietà a cui si è interessati nel contesto della progettazione di sistemi software, ossia: accuratezza, consistenza, semplicità e manutenibilità. Si illustra il grande vantaggio di ricorrere ai modelli: essendo rappresentazioni puntuali ma semplificate di sistemi reali, permettono comunque di studiarne le proprietà di interesse ma, in quanto rappresentazioni, possono essere realizzati molto rapidamente e soprattutto a costi contenuti. Inoltre forniscono una base formale di argomentazione con i clienti. Sebbene i modelli offrano vantaggi indiscutibili, a tutt'oggi, in molte organizzazioni informatiche semplicemente se ne dimentica l'importanza per concentrarsi prematuramente sull'implementazione del sistema.

Negli anni che hanno preceduto la presentazione dello UML, la comunità Object Oriented, limitatamente alla sfera dei linguaggi di progettazione, viveva un momento di smarrimento: agli inizi degli anni Novanta si contavano una cinquantina di diversi linguaggi che finivano per alimentare la cosiddetta "guerra dei metodi". Ciò creava problemi a tutti gli attori operanti nel mondo dell'Object Oriented — dai singoli tecnici, ai clienti, dalle aziende produttrici di prodotti CASE, alle varie consulting, e così via — e finiva per costituire un serio limite al fiorente sviluppo della tecnologia stessa. Lo UML è nato dallo straordinario lavoro di Grady Booch, James Rumbaugh e Ivar Jacobson, presto divenuti famosi nella comunità informatica internazionale con il nomignolo di Tres Amigos. Inizialmente, l'impegno profuso è stato indirizzato essenzialmente nel tentativo di realizzare un linguaggio di progettazione che unificasse, come suggerito dal nome stesso, quanto di buono era presente nei metodi esistenti o almeno in quelli di maggior successo.

In particolare, lo UML è scaturito, principalmente, dalla fusione dei concetti presenti nei metodi di Grady Booch, OMT (Object Modeling Techique di cui Rumbaugh era uno dei principali fautori) e OOSE (Object Oriented Software Engineering /Objectory di cui Ivar Jacobson era uno dei promotori). Lo UML è uno dei pochi esempi di standard nati nel mondo dell'industria (è stato interamente finanziato dalla Rational Software Corporation) per poi approdare ai riconoscimenti accademici: nel 1997 è stato ratificato come standard e posto sotto il controllo dell'OMG (Object Management Group).

Contrariamente a convinzioni comuni in molti tecnici, lo Unified Modeling Language, non è unicamente una notazione standard per la descrizione di modelli Object Oriented di sistemi software; si tratta bensì di un metamodello definito rigorosamente che a sua volta è istanza di un meta-metamodello definito

altrettanto formalmente. Nelle specifiche formali, la proiezione statica dello UML è specificata dai diagrammi delle classi: una delle tipologie di diagrammi previste dal linguaggio stesso. In altre parole, lo UML è definito per mezzo di sé stesso, o meglio la relativa definizione formale utilizza il linguaggio stesso.

In alcuni gruppi di lavoro vi è una certa confusione sul rapporto che lega i linguaggi di progettazione ai processi di sviluppo del software: lo UML è un linguaggio di progettazione e, come tale, è "solo" parte di metodi più generali per lo sviluppo del software. Lo UML è un formalismo utilizzato dai processi per realizzare, organizzare, documentare, … i prodotti generati dalle fasi di cui il processo si compone. Un metodo, tra i vari princìpi, è formato dalle direttive che indicano al progettista cosa fare, quando farlo, dove e perché: un linguaggio invece è carente di ciò.

I processi di sviluppo più celebri, tipicamente, sono fondati su un insieme ben definito di "filosofie" quali: Use Case Driven, Architecture Centric e Iterative and Incremental.

La prima, come suggerito dal nome, è completamente imperniata sui casi d'uso, ossia sulle funzionalità del sistema così come vengono percepite da entità esterne al sistema stesso. Focalizzare fin dalle primissime fasi del ciclo di vita del software e continuare a monitorare l'aderenza dei vari modelli alle richieste del cliente può evitare tutti quei problemi derivanti da un'inefficace dialogo tra il gruppo di tecnici e i clienti. Si tratta della strategia decisamente più ricorrente e, più o meno consciamente, utilizzata da sempre: i sistemi software in primo luogo dovrebbero essere realizzati per soddisfare specifiche esigenze dell'utente.

I processi Architecture Centric tentano di creare il sistema partendo dalla progettazione dell'architettura del sistema, la quale viene posta al centro dell'intero processo di sviluppo. Molto importante quindi è stabilire il prima possibile un'architettura ben definita e quindi un embrione di prototipo per poterla valutare. Chiaramente la versione finale del sistema e dell'architettura stessa viene raggiunta attraverso una serie di iterazioni, ma le variazioni dovrebbero essere confinate a rifiniture e non rappresentare stravolgimenti della versione base. Da tenere presente che nella pratica non esistono processi unicamente basati sull'architettura di un sistema: in genere sono ibridi che ospitano anche le direttive derivanti da dottrine Architecture Centric.

Gli approcci Iterative and Incremental prevedono di produrre il sistema finale attraverso una serie di successive versioni generate alla fine di opportune iterazioni. Ciò equivale a dire che il progetto si risolve in una successione di miniprogetti completi ognuno con opportune e prestabilite versioni del sistema. I vantaggi derivanti dall'utilizzo di metodi di sviluppo iterativo e incrementali sono riduzione dei costi dovuti a errori o imprevisti generatisi durante una singola iterazione, riduzione del rischio di non produrre il sistema nei tempi previsti, riduzione del tempo necessario per rilasciare il sistema globale, maggiore flessibilità in caso di modifiche dei requisiti utente.

I processi di sviluppo del software più interessanti sono molto probabilmente The Unified Software Development Process (processo unificato di sviluppo software) e ICONIX Use Case Driven. Tipicamente i processi non possono essere utilizzati nella loro forma originale: è necessario eseguirne determinate personalizzazioni al fine di adattarli alla realtà organizzativa, al team che si ha a disposizione, alla natura del progetto, e così via. Il paragone più immediato è con gli abiti: una volta acquistati è necessario effettuarvi dei ritocchi per adeguarli alla propria struttura, stile, ecc.

Lo Unified Software Development Process progettato dai Tres Amigos presso la Rational in realtà non è "semplicemente" un processo bensì si tratta di un *framework* di un processo generico; si presta a

essere utilizzato proficuamente per classi piuttosto estese di sistemi software, differenti aree di applicazioni, diverse tipologie di organizzazioni, svariati livelli di competenza e per progetti di varie dimensioni. Caratteristica peculiare offerta dal processo è la razionale fusione dei tre principali metodi esistenti: Use Case Driven, Architecture Centric e Iterative and Incremental. Forse ciò non solo lo rende unico ma, verosimilmente, anche raro da utilizzare nella sua versione originale: richiede un notevole investimento in termini di tempo.

Il processo presentato dalla ICONIX è anch'esso basato sulla vista dei casi d'uso e utilizza un approccio iterativo e incrementale, però risulta essere decisamente più flessibile, "leggero" e facile da applicare… Magari un po' meno formale di quello presentato dai Tres Amigos.