UPGRADE
The European Journal for the Informatics Professional
http://www.upgrade-cepis.org

Edizione italiana a cura di ALSI e Tecnoteca
http://upgrade.tecnoteca.it

Lo sviluppo basato sui modelli e l'UML 2.0. Fine della programmazione come la conosciamo ora?

di
Morgan Björkander

(Traduzione italiana a cura di Giuseppe Prencipe dell'articolo "Model-Driven Development and UML 2.0. The End of Programming as We Know It?" pubblicato sul vol. IV, n. IV, agosto 2003 della rivista online UPGrade, a cura del CEPIS)

In questo articolo, si analizza una delle promesse dell'Ingegneria del Software: lo sviluppo guidato dal modello (model-driven development), che consente agli sviluppatori di lavorare al livello più alto ed astratto dei modelli software. Viene presentato il contributo di UML 2.0 (Unified Modeling Language) con le relative implicazioni per risolvere il problema del supporto degli strumenti. Vengono anche analizzati alcuni problemi quotidiani dei progetti software.

Parole chiave: *sviluppo per modelli, progetti di software, strumenti di supporto, UML 2.0.*

1. Introduzione

"Sviluppo basato sui modelli" - scandite bene queste parole: sono dieci sillabe che indicano il cambiamento di prospettiva di una intera industria in continua trasformazione. Non stiamo parlando di una rivoluzione effimera, parliamo di un mutamento che permeerà lentamente, ma sicuramente, il modo in cui si sviluppano i sistemi. La spinta è a ridurre l'importanza del codice e a concentrarsi su ciò che realmente serve: su come, cioè, un'applicazione finale deve funzionare assicurando affidabilità e soddisfacimento delle richieste del cliente.

Lo sviluppo per modelli è parte di una visione assai più ampia, rientra nell'Architettura basata sui modelli (MDA, acronimo per Model-Driven Architecture) ed è gestito dall'OMG (Object Management Group). La Model-Driven Architecture fornisce un quadro concettuale per un approccio allo sviluppo per modelli. E comunque, mentre l'MDA, in tutte le sue potenzialità, non è ancora del tutto una vera e propria realtà, lo sviluppo per modelli è già operante qui e ora. In effetti, l'uso di modelli, se pure non molto sofisticati, è già in auge da un bel po' di tempo e, pertanto, non è proprio una novità (a meno che, naturalmente, non si sta ad ascoltare gli addetti al marketing).

2. Nessuna magia

Se lo sviluppo per modelli è davvero valido, perché non saltano tutti immediatamente sul carro dei vincitori? Allora, bisogna innanzitutto dire che lo sviluppo basato sui modelli non è la soluzione di ogni problema: ci deve essere sempre *qualcuno* che ancora verifichi la funzionalità del

sistema e nessuno strumento lo potrà fare per te. Ciò che gli strumenti possono fare è renderti il lavoro più facile e più diretto.

Secondo. Non si cambia un cavallo in corsa, durante un progetto, quando già sono in uso e fortemente radicati dei processi di sviluppo e, cosa più importante, quando devi preoccuparti dell'impatto sulle applicazioni in uso. (E se non sei in questa situazione, fa' un passo avanti e inizia a usare lo sviluppo per modelli: non ne può venire che bene). Inoltre, ci vorrebbe almeno una qualche forma di pianificazione preventiva prima di passare all'approccio per modelli e, di norma, si è disposti a cambiare approccio nei nuovi progetti solo se non si vanno a intralciare gli sforzi che si stanno facendo.

Terzo. È necessario avere un riscontro dalle persone che utilizzeranno gli strumenti (si ha bisogno di strumenti per applicare lo sviluppo per modelli). Gli sviluppatori esitano ad usare i modelli dal momento che "non è programmazione". E per giunta, temono che ostacoli il loro lavoro, pensano forse che questo tipo di sviluppo possa rendere obsolete competenze duramente acquisite. È una paura non del tutto ingiustificata: è infatti vero che lo sviluppo per modelli molto probabilmente farà diminuire la richiesta di tecnici che conoscano dalla A alla Z un certo numero di linguaggi di programmazione. D'altra parte, ogni bravo sviluppatore è, innanzitutto e soprattutto, un risolutore di problemi - ciò che lo spinge è la sfida a fornire nuove e migliori soluzioni ai più impegnativi problemi in esame. La cosa stimolante dello sviluppo per modelli è che permette allo sviluppatore di concentrarsi sulla soluzione di significative sfide progettuali aggiungendo nuove e interessanti funzionalità, invece di spendere un sacco di tempo a correggere errori di sintassi, a intervenire sulle falle della memoria o a sudare su interminabili serie di errori di basso livello.

Il quarto punto è, in verità, un corollario del terzo. Gli strumenti devono essere idonei a svolgere bene il loro compito. Sfortunatamente, i clienti a volte si aspettano troppo, o i fornitori promettono più di quanto questi strumenti possano dare. In entrambi i casi è molto facile che il cliente metta da parte l'idea di affidarsi allo sviluppo per modelli. Si deve essere assolutamente sicuri che gli strumenti possano rispondere adeguatamente alle proprie esigenze.

3. Ingegneria visuale del software

La base di uno sviluppo model-driven sono i modelli e il linguaggio usato per raffigurarli. Un modello dà la possibilità di rappresentare visivamente in maniera coerente differenti viste dello stesso sistema. Un errore comune è pensare allo sviluppo per modelli soltanto come ad una relazione tra un modello e il codice attraverso cui viene realizzato. Anche se questa è gran parte dell'equazione, rimanda, però, ad una visione troppo ristretta.

Uno dei principali scopi del modello è colmare il divario di comunicazione tra le diverse parti coinvolte nel processo di sviluppo: gli ingegneri che devono individuare i requisiti del sistema, gli analisti di sistema, gli sviluppatori del software e i verificatori, tutti parlano lo stesso linguaggio. Attenzione, essi si specializzeranno nei diversi settori del linguaggio, in quelli cioè che più sono congeniali a loro e che più si adattano alle loro esigenze, ma condivideranno alcuni costrutti fondamentali ed avranno anche una comune visione del sistema su cui lavorano. Inoltre, l'uso di un linguaggio comune tende a rendere labili i confini tra i ruoli, facilitando notevolmente, durante le varie fasi di un progetto, lo spostamento delle persone dove ce n'è bisogno. Ci sono anche altre parti interessate a seguire le fasi del lavoro e sono i responsabili del progetto, i manager e i revisori. E, ancora più importante, bisogna dire che il *cliente* ha bisogno di comprendere ciò che gli viene dato e vuole anche essere coinvolto in tutte le fasi del processo di sviluppo parlando con i tecnici impegnati nel sistema. Un linguaggio di modellazione grafica come l'UML permette la comunicazione tra queste componenti aiutando a superare qualche complessità del sistema. Questa possibilità di aumentare il coinvolgimento del cliente, del management e della gran parte dell'organizzazione è una delle più grosse ragioni per cui lo sviluppo model-driven sta attirando l'attenzione degli alti dirigenti di aziende di distribuzione di sistemi e di software per il mercato.

E che dire della programmazione? Non è più necessaria? Ne abbiamo parlato prima, ma ne riparliamo ora con più ampiezza. Una volta immessa sufficiente informazione in un modello, gli

strumenti possono generare la gran parte o tutto il codice richiesto da un sistema. Nota che se tu usi uno strumento atto a generare *tutto* il codice, ciò equivale a *compilare il modello*. In definitiva, si ha un cambiamento di paradigma che è simile a quello avvenuto quando si è passati dalla programmazione in linguaggio Assembler a quella in linguaggio C che, specie dai programmatori dell'Assembler, all'inizio, veniva visto con un certo scetticismo. Con lo sviluppo per modelli parliamo di un cambiamento di paradigma simile, un cambiamento in cui i linguaggi di modellazione si sostituiscono ai linguaggi di programmazione e l'implementazione avviene attraverso il linguaggio di modellazione. Ciò è reso possibile dal fatto che i linguaggi di modellazione stanno diventando abbastanza potenti da permettere di specificare in dettaglio il comportamento del sistema, quasi allo stesso livello di granularità di un comune linguaggio di programmazione. Questo implica inoltre che è possibile eseguire modelli, simulare cioè, da una prospettiva funzionale, come il sistema lavorerà. Possono così applicarsi tecniche di verifica e di validazione per controllare la correttezza del sistema. Nel modello si ignorano anche normalmente fastidiosi dettagli come la distribuzione, i rappresentanti e la gestione della memoria e si lascia che siano gli strumenti a generare quel codice. E ciò significa che si ha anche bisogno di modellare - o programmare, se vuoi - il comportamento del sistema, ma che si può farlo ad un livello più astratto che punti sulle funzionalità importanti.

Il più bravo tra i tuoi programmatori, quello che ha salvato l'azienda tante volte che nemmeno riesci a ricordarle, probabilmente ti dirà: "Farei molto prima a programmare questo sistema appositamente per te". E tu sai che forse ha ragione. Eppure c'è un grosso "ma". *Che cosa* è quello che lui si appresterebbe a programmare? Chi gli ha creato le specifiche? Non fa parte di una squadra? O il sistema è davvero così piccolo che può essere specificato, sviluppato e testato da un solo individuo? E anche se fosse, vorresti che tutte queste informazioni siano solo nella testa di questa persona? Che succede se viene investito da una macchina o riceve da un'azienda concorrente un'offerta che non può rifiutare? E che succede dopo la distribuzione? Potrà il cliente finale modificare l'implementazione? O non è per lui comprensibile? Sarà facile mantenere l'implementazione per le future versioni?

Queste domande ci portano a considerare un altro principale scopo dello sviluppo per modelli. Si tratta di inserire lo *sviluppo* di sistemi e di software all'interno di una *disciplina ingegneristica* del software e dei sistemi. Lo sviluppo per modelli si volge ai sistemi di sviluppo e di manutenzione. Un sistema non è fatto solo della applicazione, ma anche delle diverse parti che rendono possibile la comprensione dell'applicazione stessa. Un modello può contenere parti che sono chiaramente eseguibili, ma avrà quasi sempre altre parti che non sono necessariamente eseguibili, come i requisiti, i primi abbozzi del sistema, i modelli di business e i modelli d'analisi. Tutti questi aspetti evolveranno e saranno aggiornati durante un progetto di sviluppo e sono essenziali per la futura manutenzione.

I moderni strumenti di uno sviluppo per modelli danno la possibilità di eseguire (parti di) un modello e questo rende possibile verificare subito che il sistema funzioni come si voleva. Ciò a sua volta porta a ridurre significativamente i rischi del progetto. La verifica diviene un'attività ancora più importante nello sviluppo per modelli, dal momento che viene applicata molto prima e più frequentemente. In questo modo, si è più sicuri che i diversi componenti di un'applicazione alla fine del progetto convergeranno tutti adeguatamente allo scopo. Intuitivamente, si potrebbe pensare che, per tutto questo lavoro extra, le attività di sviluppo richiedano più tempo, ma l'esperienza dimostra che il tempo per l'immissione sul mercato viene di fatto ridotto. Ci vuole meno tempo nelle fasi dell'implementazione e di verifica e più tempo nelle fasi di analisi e di disegno e, dal momento che si procede iterativamente attraverso queste fasi, il guadagno diventa evidente.

4. Il ruolo dell'UML 2.0

Non sorprendentemente il linguaggio usato per lo sviluppo basato su modelli è l'Unified Modelling Language (UML). Questo linguaggio è diventato uno standard per la prima volta nel 1997 mettendo fine alla "guerra dei metodi" [1] e si affermò velocemente diventando il più popolare

linguaggio di modellazione per "visualizzare, costruire e documentare gli artefatti di un sistema software" [2].

Sono passati da allora sei anni e sia i fornitori degli strumenti del linguaggio che gli utenti lo hanno potuto ampiamente sperimentare. Sappiamo cosa fa e sappiamo cosa deve essere migliorato. Inoltre, la stessa industria del software è in questi anni cambiata e chiede strumenti di supporto per le nuove tecnologie come lo sviluppo per componenti e i modelli eseguibili. Queste esigenze non venivano adeguatamente soddisfatte dalle precedenti versioni dell'UML e, per affrontarle, si diede il via a una approfondita revisione del linguaggio creando l'UML 2.0 che a giugno ha ricevuto il primo esame in vista dell'adozione.

Per quanto riguarda la standardizzazione, l'UML 2.0 si appresta dunque (dovrebbe avvenire entro settembre) ad essere una specificazione *adottata.* Si è formata una task force che lavorerà per lo standard finale e che ha l'incarico di correggere gli errori e di allinearlo con altri linguaggi standard. Molto lavoro di allineamento è già stato fatto e così si pensa che i futuri cambiamenti saranno del tutto secondari. Si noti che una task force per la versione finale è assai simile a quella che ha l'incarico di aggiornare regolarmente le precedenti versioni dell'UML 1.x. Quando la task force per lo standard definitivo ha completato il suo lavoro, si ha il cambiamento di status della specificazione che viene dichiarata *disponibile*, ma questo è principalmente una formalità. Dal momento che l'UML deve rispondere alle richieste delle diverse parti interessate, deve essere un linguaggio davvero completo, ma non è certamente il caso che tutti lo conoscano per intero. Viene appositamente diviso in diverse viste, o diagrammi, che permettono di focalizzarsi su specifiche aree pertinenti per l'interessato. Altri potranno lavorare su altre viste e il modello si manterrà del tutto coerente.

5. Raccolta dei requisiti e test

Uno degli scopi della modellazione è il coordinamento delle diverse fasi del processo di sviluppo. L'UML può essere usato per la fase di raccolta dei requisiti, per l'analisi, per il disegno, per l'implementazione e per il testing. Uno dei più importanti strumenti per questo lavoro è il diagramma di sequenza che è stato sottoutilizzato nell'UML 1.x. Ciò diviene chiaramente evidente nella costruzione di un sistema più grande e quando si scopre che i costrutti del linguaggio non sono sufficienti ad affrontare il grado di scalabilità richiesto. Le interazioni sono utili in molte diverse circostanze, ma sono forse più conosciute per l'uso che se ne fa nelle fasi della definizione dei requisiti e dell'analisi. Un altro comune uso è nel fornire traccia di quanto eseguito, in tal caso le sequenze sono automaticamente generate per illustrare il flusso di comunicazione tra le diverse parti di un sistema. Oltre a ciò, le sequenze sono di grande importanza quando si arriva alla descrizione dei test di verifica. Per esempio il nuovo profilo del testing che emerge dall'OMG è basato principalmente sui diagrammi di sequenza.

Gli ulteriori elementi aggiuntivi che si accompagnano ai diagrammi di sequenza puntano primariamente alla scalabilità. Uno di questi, tra i più semplici, - e forse anche tra i più potenti - è l'introduzione dei riferimenti alle altre interazioni. Questo significa che non è più necessario duplicare le informazioni in diverse interazioni, ed è anche possibile suddividere una interazione in altre più piccole che si rapportino anche ad altri contesti. Collegata fortemente a questa caratteristica è la possibilità di organizzare le interazioni come in uno schema di flusso per indicare l'ordine in cui esse devono essere interpretate. In questo modo, è possibile, ad esempio, ricavare rapidamente nuovi casi di test da quelli esistenti.

Nell'UML 1.x un'interazione poteva mostrare solo singole sequenze; per mostrare casi alternativi o eccezionali, era necessario creare vari diagrammi di sequenza non collegati. Con l'UML 2.0 è possibile raggruppare un certo numero di messaggi in un solo frame e poi specificare come quel gruppo di messaggi deve essere trattato. Per esempio il gruppo può essere opzionale (opt), essere parte di un'alternativa (alt) o può essere eseguito in maniera iterativa (loop). È anche possibile specificare che parecchi altri gruppi dovrebbero concorrere in maniera simultanea (par).

Nella figura 1 si possono vedere alcuni esempi di queste possibilità.

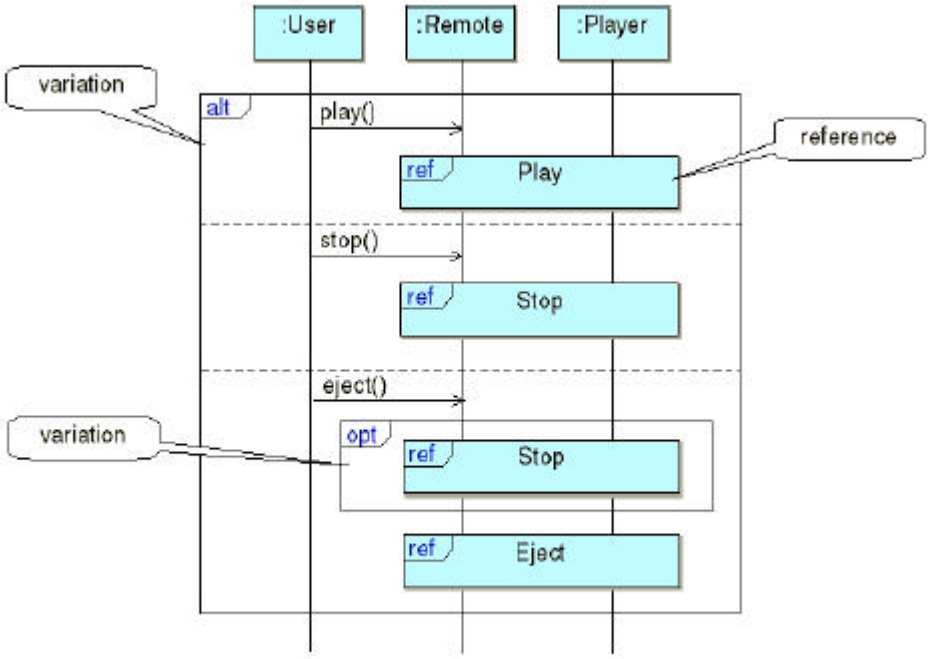

Figure 1: A Sequence Diagram with References to Other Interactions and a Few Variations

Si deve apprezzare il fatto che è possibile legare una interazione ad una singola linea di vita per mostrare le comunicazioni "interne" tra le parti degli oggetti rappresentati. Questa caratteristica si aggiunge alle nuove possibilità di descrivere gerarchicamente architetture di sistemi decomposti, ma possono anche essere usate per filtrare le informazioni non pertinenti a certi gradi di granularità. In relazione ai sistemi integrati vale la pena dire che sono stati aggiunti dei diagrammi di temporizzazione per mostrare nel tempo i cambiamenti di stato.

6. Costruire architetture

Le nuove possibilità di modellare l'architettura dei sistemi che sono state introdotte nel linguaggio sono tra le più importanti e rendono più facile la costruzione di sistemi real-life di notevole complessità.

Dal momento che i sistemi diventano sempre più grandi e più complessi, diventa primario mantenere contratti e interfacce tra le diverse parti del sistema. L'UML è stato significativamente migliorato in quest'area e, tra l'altro, le interfacce che sono realizzate da una classe sono distinte da quelle che sono richieste. Questo è importante perché permette ad una classe o ad una componente di essere vista come un'entità a sé, dove le interfacce danno tutte le informazioni necessarie sul contesto entro cui vanno ad operare. Inoltre, le interazioni forniscono un eccellente meccanismo per descrivere i contratti tra le diverse parti. Non è sufficiente sapere quali servizi sono disponibili, è necessario sapere anche l'ordine in cui essi possono essere invocati (se si dà uno sguardo all'implementazione della classe per rappresentare ciò, si vede che non c'è alcuna difficoltà ad usare le interfacce).

Uno dei più importanti nuovi concetti dell'UML è la porta e uno dei suoi ruoli è fornire un punto di vista di una classe, per esempio poter assegnare gruppi di interfacce a differenti serie di utenti della classe. Un esempio di questo viene mostrato nella sottostante figura 2. Nei sistemi real-time è più comune aver a che fare con classi attive che con classi passive, con classi cioè che hanno

un proprio thread concettuale di controllo. Nella UML 2.0 queste classi attive hanno barre verticali ai lati per distinguerle più facilmente dalle classi passive.

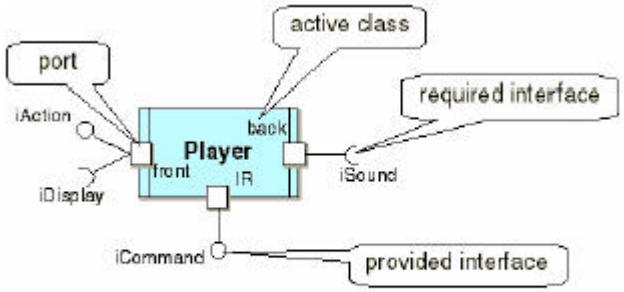

Figure 2: An Active Class with Ports and Interfaces towards Other Classes

Il comportamento delle classi attive è normalmente dato attraverso macchine di stato. Ma sono state adottate dall'UML anche alcune soluzioni atte a descrivere architetture di sistemi e che vengono usate prevalentemente nei domini real-time. La scomposizione gerarchica dei sistemi vi gioca un ruolo importante. Un sistema può essere suddiviso in parti più piccole e più gestibili, e queste a loro volta possono essere ulteriormente suddivise. Nella figura 3 si può vedere un esempio di questo e si può vedere anche come vengono sfruttate le altre potenzialità delle porte, e cioè come punti di connessione che permettono di istituire collegamenti tra le diverse componenti di un sistema. Una volta create, queste componenti possono essere naturalmente riusate in altri contesti in modo simile ai blocchi della Lego™. Per indicare il flusso delle informazioni nei canali si danno dei nomi alle interfacce.

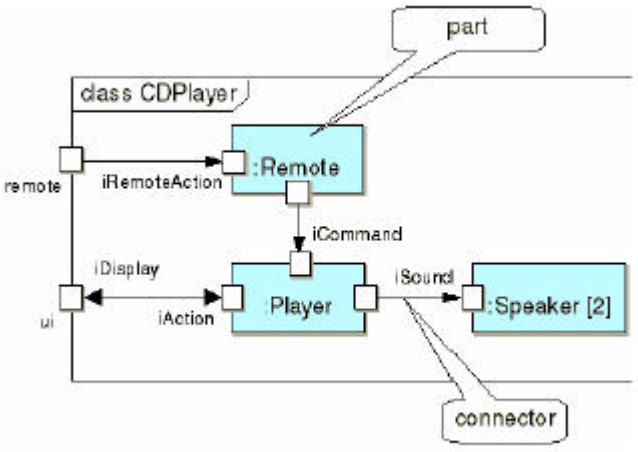

Figure 3: A Class Hierarchically Decomposed into its Parts

7. Considerazioni sugli strumenti di sviluppo

Gli strumenti implementano lo sviluppo per modelli in maniera diversa e forniscono più o meno flessibilità. Il round-trip engineering è un surrogato che non riesce a cogliere i comportamenti in un modello e soffre primariamente del fatto che si è legati a un particolare linguaggio di programmazione e anche del fatto che si ha a che fare con un approccio che si basa principalmente sul codice, per cui il modello viene dimenticato quando si incappa in una strettoia come avviene quando il progetto ha davanti una scadenza improrogabile (sembra essercene sempre una dietro l'angolo). Alla fine di un tale processo si finisce con una sorta di applicazione che funziona e con un

modello praticamente inutile. E allora? Mi vuoi chiedere di nuovo perché è importante ricorrere a un linguaggio di modellazione?

Un modo di venire a patti con i problemi del round-trip engineering è inserire il codice di programmazione direttamente nel modello, ma ciò ti costringe ad aggiornare il modello per ottenere un'applicazione funzionante alla fine del lavoro. E di nuovo si patisce il fatto che si è legati a un particolare linguaggio di programmazione e si devono immettere pezzi del codice in molte diverse sezioni del modello. È proprio come immettere del codice Assembler in un programma C: qualche volta è necessario, ma la manutenzione richiederà il tuo intervento e ti darà problemi.

Data la possibilità di specificare il comportamento del sistema direttamente nel modello, entrambi gli approcci citati sopra diventano irrimediabilmente superati. Si può generare codice automaticamente in ogni linguaggio previsto dalla tua piattaforma premendo un bottone, e ciò significa che si ha portabilità a livello di modello. Tu non hai mai bisogno di cambiare il codice direttamente; questo avverrà al momento della implementazione del modello.

C'è ancora una cosa, comunque, da far osservare. La maggior parte delle aziende di vendita oggi ha una propria mappa di linguaggi e, secondo l'MDA, per andare avanti con successo su questa strada, si richiedono mappe e profili standardizzati per i diversi linguaggi e per le varie piattaforme. La buona notizia è che il futuro è già iniziato. Lo sviluppo model-driven funziona davvero e *cambierà* il modo in cui si sviluppano i sistemi.

Riferimenti

[1]
C. Kobry. UML 2001: A Standardization Odyssey, Communications of the ACM, vol. 42, no. 10, October, 1999.

[2]
OMG, UML Unified Modelling Language Specification, version 1.5, March, 2003
<http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf>.

[3]
OMG, UML 2.0 Infrastructure RFP, OMG Document: ad/2000- 09-01, 2000,
<http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Infrastructure_RFP.html>

[4]
OMG, UML 2.0 Superstructure RFP, OMG Document: ad/2000- 09-01, 2000,
<http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Superstructure_RFP.html>

[5]
OMG, UML 2.0 OCL RFP, OMG Document: ad/2000-09-03, 2000,
<http://www.omg.org/techprocess/meetings/schedule/UML_2.0_OCL_RFP.html>

[6]
OMG, UML 2.0 Diagram Interchange RFP, OMG Document: ad/2001-02-39, 2000,
<http://www.omg.org/techprocess/meetings/schedule/UML_2.0_Diagram_Interchange_RFP.html>

Morgan Björkander lavora come specialista di metodi alla Telelogic, un'azienda con sede centrale in Svezia, <http://www.telelogic.com>. Lavora sulla standardizzazione dei linguaggi di modellazione da cinque anni ed è profondamente coinvolto nella standardizzazione dell'UML 2.0 ad opera dell'Object Management Group. Prima di imbarcarsi nell'odissea della standardizzazione, è stato ingegnere del software ed ha preso parte a vari progetti europei di creazione di servizi nell'ambito delle telecomunicazioni. Può essere raggiunto a questo indirizzo: <Morgan.Bjorkander@telelogic.com>.

Giuseppe Prencipe, insegnante e traduttore freelance. Si è laureato in Lettere all'Università Cattolica "S. Cuore" di Milano e ha approfondito le tematiche relative alla documentazione, alla biblioteconomia e alla gestione delle conoscenze conseguendo il diploma in Archivistica presso l'Archivio di Stato di Bari e il diploma di perfezionamento in Storia moderna presso l'Università di Urbino <giuprencipe@tiscali.it>.