Appendice A

UML e i linguaggi di programmazione non OO

Introduzione

Molte volte si è detto che lo UML è un linguaggio di modellazione che — sebbene ideato per la progettazione di sistemi Object Oriented e component-based — si presta a essere utilizzato per la produzione di sistemi che prevedano linguaggi di programmazione non basati su tali paradigmi. Non solo, lo UML può essere utilizzato proficuamente per modellare sistemi che addirittura non prevedono implementazione. L'esempio più evidente è l'utilizzo dello stesso UML per modellare il metamodello di cui è istanza.

La domanda alla quale si intende rispondere in questa Appendice è come utilizzare i vari strumenti messi a disposizione dallo UML per la modellazione di sistemi non basati sul paradigma OO. A tal fine si è cercato di fornire una risposta inquadrata nel contesto di un processo formale di sviluppo del software.

Analisi dei requisiti

Le prime fasi da affrontare in un qualsivoglia processo di sviluppo del software consistono nell'eseguire l'analisi dei requisiti. I modelli da produrre in questa fase, basati sull'utilizzo dello UML, sono essenzialmente di due tipologie: casi d'uso e modelli ad oggetti. In particolare, in funzione di quale fase si stia lavorando, questi modelli prendono il nome di casi d'uso e modello a oggetti business e casi d'uso e modello a oggetti del dominio. I principali strumenti UML da utilizzare sono quindi i diagrammi dei casi d'uso e quelli delle classi. A questi eventualmente possono essere affiancati quelli di sequenza (utili per

illustrare gli *scenario* dei casi d'uso), quelli degli stati (utilizzati per illustrare il ciclo di vita di particolari "entità" e processi presenti nell'area oggetto di studio) e, meno frequentemente, quelli degli oggetti (utilizzati per illustrare particolari istanze di modelli a oggetti o porzioni meno chiare o meno intuitive).

Premesso ciò, una prima constatazione da fare è che il "paradigma" utilizzato per condurre l'analisi dei requisiti non ha alcuna influenza diretta su quello adottato per disegnare il sistema. Più volte si è posto l'accento sul fatto che il livello di formalità dei modelli prodotti durante le prime fasi deve essere necessariamente "rilassato" per una serie di validi motivi. Il più importante è consentire a utenti e clienti, che tipicamente non sono esperti di informatica e tanto meno di astrazioni OO, di poter collaborare attivamente al processo di analisi dei requisiti (rivedere i modelli, suggerire modifiche, fornire opinioni, ecc.). Questo rilassamento di certo non pregiudica la qualità del disegno: si possono realizzare perfetti sistemi Object Oriented o component-based a partire da specifiche fornite attraverso documenti in linguaggio naturale (che ben poco hanno di OO) così come, situazione opposta, si possono realizzare sistemi non OO — anche involontariamente — con modelli di analisi dei requisiti perfettamente formali.

Il suggerimento pertanto è realizzare l'analisi dei requisiti come se si dovesse realizzare un sistema OO Si tratta di una buona idea anche considerando il ritorno degli investimenti. L'attuale tendenza del settore è costruire sistemi component-based e quindi, prima o poi, specie se il sistema riscuoterà un sufficiente successo, questo dovrà essere reingegnerizzato. In tale evenienza il modello OO dell'analisi dei requisiti realizzato con gli strumenti UML "standard" costituisce un ottimo punto di partenza. Altre considerazioni sono relative alla presenza di tool di supporto che semplificano il lavoro, al rilassamento della formalità delle leggi OO tipico di queste fasi, e via discorrendo.

In sintesi, considerando gli strumenti dello UML da utilizzare, si può dire:

- i diagrammi dei casi d'uso (eventualmente supportati dai diagrammi di sequenza e quelli di stato) si utilizzano normalmente;
- il modello di business (tipo di diagrammi di classi) è molto generale e non strettamente OO;
- il modello di dominio è focalizzato sull'organizzazione di dati e relazioni contestuali; si può dire che somiglia molto a un diagramma entity-relationship, poco utile indipendentemente dal paradigma utilizzato per produrre il sistema.

Per concludere, questa fase del processo di sviluppo del software è abbastanza indipendente dal paradigma selezionato per l'implementazione del sistema: è pertanto consigliabile utilizzare strumenti moderni.

Analisi

In questa fase le cose cominciano a cambiare. Si tratta del vero punto di raccordo tra analisi dei requisiti e disegno. Da un lato si realizza una versione formale dei requisiti utenti e dall'altro si inizia a disegnare il sistema (a tutti gli effetti si realizza una prima versione del disegno, nel quale gran parte dei requisiti non funzionali sono ancora esclusi). Per la realizzazione dei vari modelli, si presentano due possibilità:

1. comportarsi come se si stesse producendo un sistema OO o component-based. A favore di questo approccio c'è da dire che chiunque possieda una buona esperienza OO tende a disegnare/realizzare sistemi come se fossero basati sul paradigma OO, anche se poi si utilizzano linguaggi non OO. A supporto di questa opinione è sufficiente pensare che le prime versioni dei compilatori C++ erano semplicemente compilatori C forniti di macro evolute. In ogni modo, i vantaggi di questo approccio sono essenzialmente gli stessi riportati nel paragrafo precedente (salvaguardia degli investimenti, supporto di tool commerciali). In generale, si tratta di un approccio ancora possibile in quanto, sebbene il livello di formalità sia sicuramente superiore rispetto a quello delle fasi precedenti, non è ancora elevato come quello richiesto dalle fasi successive. Gli svantaggi sono legati al fatto che si rimanda alla fase successiva, già intrinsecamente complessa, una gran mole di lavoro necessaria per rivedere e adattare il modello a una versione OO a elevato livello di formalità.

2. adattare il formalismo dei diagrammi delle classi alle proprie necessità. Quantunque questa attività possa essere complicata, deve comunque essere realizzata nella fase successiva. In questo stadio, le classi disegnate sono di tre tipi: entità, controllori e "di confine". Per quanto concerne la prima e la terza, non ci dovrebbero essere troppi problemi, si tratta di classi che incapsulano, rispettivamente, record di dati e moduli di interfacciamento. Pertanto il particolare paradigma selezionato non dovrebbe influenzare tantissimo il disegno di questi elementi. Qualche inconveniente potrebbe invece sorgere con i controllori (classi controller). Dall'analisi dei moduli nei linguaggi di programmazione non Object Oriented basati sul solo paradigma della programmazione strutturata, è possibile notare che comunque esistono variabili "globali" al modulo, eventualmente globali all'intero sistema, e procedure. Le prime possono essere dichiarate nella sezione dedicata agli attributi, mentre le seconde nel dipartimento destinato ai metodi.

Restringendo l'insieme delle relazioni utilizzabili e modificando il significato dei vari dipartimenti delle classi, è ancora possibile utilizzare il formalismo dei diagrammi delle classi anche per linguaggi non OO. Per quanto concerne i diagrammi di interazione, questi sono applicabili indipendentemente. Invece di rappresentare oggetti istanze di classi, si

visualizzano istanze di moduli. Il vantaggio di questa tecnica è che si comincia a disegnare il sistema spostandosi molto verso la fase di disegno che quindi viene parzialmente alleggerita. Gli svantaggi sono essenzialmente legati all'impossibilità di riutilizzare i modelli prodotti per eventuali successive versioni OO del sistema.

In questa fase pertanto si deve affrontare il primo dilemma: continuare con un approccio OO o abbandonarlo per iniziare ad avvicinarsi alla rappresentazione reale del sistema? L'autore è un sostenitore del primo approccio; la scelta però dipende da molti fattori, non ultime le capacità del personale coinvolto nel processo di sviluppo, la riusabilità di modelli realizzati in precedenza, la necessità di preservare gli investimenti.

Disegno

Qui purtroppo c'è ben poco da fare: è necessario dar luogo alla seconda alternativa evidenziata nel paragrafo precedente: produrre un modello di disegno considerando che il sistema da sviluppare non sfrutta il paradigma OO.

Di versioni del modello di disegno, tipicamente, ne esistono almeno due: specifica e implementazione. In realtà non si tratta di versioni completamente separate, bensì si assiste a un graduale passaggio da un livello di dettaglio elevato a uno sempre più vicino alla codifica. Alla fine si vuole che il modello sia una proiezione grafica del codice.

Il consiglio quindi non può che essere quello di adattare i vari formalismi alla rappresentazione di sistemi non OO.

Per quanto concerne la proiezione dinamica, nei diagrammi di interazione non cambia molto. Per esempio se nella prima riga dei diagrammi di sequenza sono riportate istanze di moduli o package anziché di oggetti, le cose dovrebbero ancora funzionare bene. Per quanto attiene alla proiezione statica, si possono utilizzare i diagrammi delle classi e degli oggetti con gli avvertimenti specificati nel paragrafo precedente. Probabilmente è opportuno porre attenzione alle relazioni utilizzate. Quella che potrebbe creare maggiori problemi è la relazione di ereditarietà. Sebbene possa essere simulata (consultare Capitolo 6), probabilmente in questa fase è più opportuno evitarla.

Restanti diagrammi

Nei precedenti paragrafi sono stati menzionati quasi tutti i diagrammi messi a disposizione dello UML. Restano fuori sono quelli implementativi: diagrammi dei componenti e di dispiegamento. Mentre il primo ha una connotazione ben definita ed è collocabile nella fasi di analisi e disegno, il secondo ha una collocazione più ampia. È opportuno disegnare le prime versioni dell'architettura fisica fin dall'analisi dei requisiti utenti. Ciò per diversi motivi, non ultimo per fornire preziosi riscontri di fattibilità ai requisiti stessi specie quelli non funzionali. La notizia bella è che in questo caso i diagrammi si prestano a essere utilizzati così come sono senza grosse modifiche. In particolare, nei diagrammi di

dispiegamento (*deployment*) è necessario illustrare la struttura fisica del sistema in termini di nodi e relative relazioni che, evidentemente, non hanno dipendenze dal particolare paradigma utilizzato per realizzare il sistema. I vari nodi componenti l'architettura sono la sede in cui sono installati i componenti del sistema. Questi ultimi possono essere sia componenti veri e propri (per esempio EJB) oppure parti più generali dell'architettura (web server, application server, file `.dll`, ecc.). Pertanto i diagrammi dei componenti si prestano a essere utilizzati in maniera molto formale, oppure in modo più pragmatico, come richiesto da un sistema non basato sul paradigma OO.

Conclusioni

Quantunque lo UML sia un linguaggio di modellazione studiato per la produzione di sistemi basati sul paradigma OO, è possibile utilizzarlo proficuamente anche qualora il sistema da produrre non sia basato su tale paradigma.

In questo scenario, una scelta importante consiste nel selezionare accuratamente il "punto di rottura", ossia la fase del processo di sviluppo del software in cui interrompere la produzione dei modelli come se si trattasse di un sistema OO e iniziare a produrre modelli più rispondenti al reale paradigma selezionato per lo sviluppo del sistema. Le alternative sono due: iniziare a realizzare i modelli di analisi secondo il paradigma non OO oppure ritardare il passaggio fino al modello di disegno. L'autore è dell'opinione che comunque è conveniente realizzare un maggior numero possibile di modelli OO, non fosse altro per motivi di ritorno degli investimenti.

In ogni modo, qualche problema di utilizzo dello UML nasce durante la modellazione della struttura statica del sistema. In particolare, è necessario sia ricorrere ad alcuni artefici, sia limitare le tipologie di relazioni utilizzabili.

In ogni modo, anche qualora il sistema da realizzare non sia fondato sul paradigma OO, l'utilizzo dello UML risulta una scelta vincente: migliore ritorno degli investimenti, utilizzo di strumenti "più standard", maggiore supporto di tool commerciali, preparazione per la reingegnerizzazione.