UML

Riccardo Prandini

1. COS'E' UML
   Lo Unified Modeling Language, "linguaggio di modellazione unificato" è nato come una notazione per la modellazione e specifica orientata agli oggetti
   Oggi, essendo divenuto uno standard de facto, viene utilizzato ovunque ci si possa ricondurre a concetti OBJECT ORIENTED.

2. UML LINGUAGGIO
   UML è un linguaggio di modellazione , non un metodo.
   Infatti un metodo è in genere costituito da un linguaggio di modellazione e da un processo, dove il processo indica i passi necessari per eseguire il progetto, e il linguaggio di modellazione è una notazione che il metodo usa per rappresentare il progetto stesso.
   UML è quindi indipendente (si pone tale scopo) dal processo utilizzato nello sviluppo del software. Il processo è infatti scelto dal progettista, in base alle sue esperienze e al tipo di applicazione da sviluppare.

3. LA NASCITA
   I linguaggi per la modellazione object-oriented iniziarono a svilupparsi in diversi contesti a partire dagli anni '80. Si trattava di notazioni di varia natura, che consentivano di descrivere la struttura di un sistema software a oggetti (in termini di classi e relazioni fra classi) ed eventualmente il suo comportamento dinamico.
   La proliferazione di queste notazioni diede luogo a quelle che furono poi battezzate "guerre dei metodi" (method wars), con diversi progettisti, o organizzazioni, che adottavano e sostenevano una particolare notazione a scapito di altre adottate altrove.
   Intorno alla metà degli '90 diversi metodi e linguaggi iniziarono a fondersi, e si iniziò a delineare la possibilità di una integrazione dei principali formalismi. Fra le metodologie e le notazioni più apprezzate e diffuse del periodo spiccavano OMT (Object Modeling Technique) di Jim Rumbaugh, e il cosiddetto metodo Booch di Grady Booch, entrambi ricercatori presso Rational Software. Il lavoro di unificazione iniziò con loro; in seguito si unì a questo sforzo Jacobson con la sua software house Objectory.
   Il primo risultato congiunto di questo team fu OOSE (Object Oriented Software Engineering). Mentre i tre operavano per unificare i propri approcci all'analisi e alla progettazione a oggetti, il progetto fu accolto sotto l'egida dell'OMG (Object Management Group), un consorzio fondato con l'obiettivo di creare e gestire standard nel contesto dello sviluppo del software a oggetti. Nel 1995, l'OMG raccolse tutti i principali metodologisti del settore in un incontro internazionale per discutere della notazione unificata. Nel 1996 l'OMG emise una RFP (Request for Proposal) per tale notazione. Nello stesso anno, Booch, Rumbaugh e Jacobson misero a punto le release 0.9 e 0.91 di UML. Il progetto fu ben accolto dalla comunità internazionale e innumerevoli grandi organizzazioni si unirono a Rational per proseguirlo (per esempio Digital, Hewlett-Packard, IBM, Microsoft, Oracle e Unisys). Questo gruppo esteso realizzò, nel 1997, UML 1.0, che fu sottoposto alla OMG come risposta alla RFP dell'anno precedente. La release 1.1 di UML contribuì a consolidare la semantica del linguaggio e incluse elementi tratti da una proposta avanzata indipendentemente all'OMG da un gruppo composto da IBM, ObjectTime, Ptech e altre.
   L'OMG tuttora gestisce lo standard di UML definendo una sintassi e delle regole di interpretazione. UML è potuto così facilmente diventare uno standard de facto e de iure.

4. DIFFUSIONE
   Come già detto l'OMG definisce sintassi e regole di interpretazione di materiale UML, permettendone così un utilizzo al di fuori del campo informatico.
   Il campo dove si è diffuso maggiormente l'UML è proprio quello, degli ambienti integrati di sviluppo per linguaggi a oggetti come Java, C++ o .NET, che comprendono strumenti di

modellazione in UML, eventualmente con meccanismi automatici di traduzione parziale dei diagrammi UML in codice. Viceversa, sono anche disponibili ambienti software sofisticati dedicati alla modellazione in UML che consentono di generare codice in diversi linguaggi. Fra gli ambienti più noti di quest'ultima categoria va senz'altro citato Rational Rose di Rational Software (un'organizzazione a cui appartenevano due dei tre padri di UML) e Together di Borland, che può essere integrato con diversi ambienti di sviluppo altra valida alternativa è Argo UML. Vi sono altri strumenti che si pongono solo l'obbiettivo di permettere di realizzare diagrammi come DIA o VISIO.
Ora UML si sta sempre più diffondendo in ogni settore, come nel mondo delle architetture a componenti, dove si utilizza per specificare i componenti, le loro interazioni e l'integrazione in sistemi completi e coerenti.Si usa nelle fasi della definizione dei requisiti, dell'identificazione, interazione e specifica dei componenti, del provisioning e dell'integrazione. È focalizzato sulla specifica delle caratteristiche esterne dei componenti e sulle loro interdipendenze, piuttosto che sull'implementazione interna(rispettando l'idea di UML).

5. STRUTTURA DELL'UML
   L ' UML presenta una tipica struttura a strati;
   1. Viste: mostrano i diversi aspetti del sistema per mezzo di un insieme di diagrammi.
   2. Diagrammi: permettono di descrivere graficamente le viste logiche.
   3. Elementi del modello: concetti che permettono di realizzare vari diagrammi (es. attori, classi,packages, oggetti, e così via).

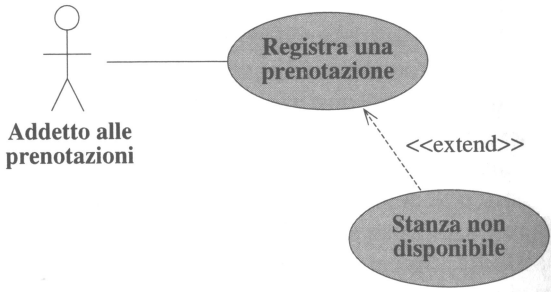
6. LE VISTE
   Lo strato più esterno dell' UML è costituito dalle seguenti viste:
   1. Use Case View (casi d'uso) utilizzata per analizzare i requisiti utente. Obiettivo di questo livello di analisi è studiare il sistema considerandolo come una scatola nera. È necessario concentrarsi su cosa il sistema deve fare astraendosi il più possibile dal come: è necessario individuare tutti gli attori, i casi d'uso e le relative associazioni. Importante è dettagliare i requisiti del cliente, capirne i desideri più o meno consapevoli, cercare di prevedere i possibili sviluppi futuri, ecc.
   2. Design View (progetto o disegno) descrive come le funzionalità del sistema devono essere realizzate; in altre parole analizza il sistema dall'interno (scatola trasparente).
   3. Implementation View (implementazione) descrive i packages, le classi e le reciproche dipendenze.
   2. Process View (processi) individua i processi e le entità che li eseguono sia per un utilizzo efficace delle risorse, sia per poter stabilire l'esecuzione parallela degli oggetti.
   4. Deployment View ("rilascio") mostrano l'architettura fisica del sistema e definisce la posizione delle componenti software nella struttura stessa.

7. I DIAGRAMMI

7.1. Use Case Diagram

-rappresentano le modalità di utilizzo del sistema da parte di uno o più utilizzatori (attori)
-descrivono l'interazione tra attori e sistema(senza rivelarne l'organizzazione interna)
-sono espressi in forma testuale, comprensibile anche per i non "addetti ai lavori"
-possono essere definiti a livelli diversi (sistema o parti del sistema)
-ragionare sui casi d'uso aiuta a scoprirei requisiti

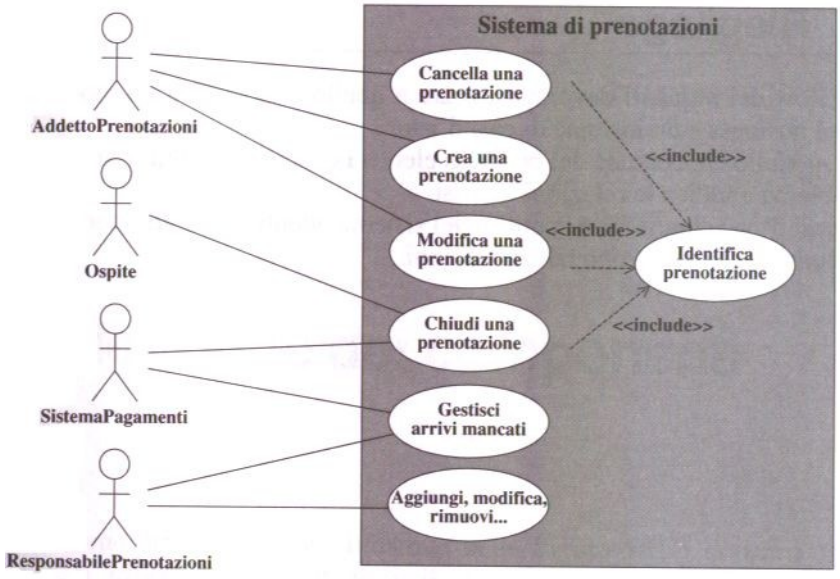
Uno use case descrive una tipica interazione tra l'utente e il sistema in analisi; il diagramma le formalizza. Gli Use Case Diagram (UCD) modellano il comportamento esterno di un sistema in termini delle funzioni che esso mette a disposizione agli attori interagiscono con essi (utenti, altri sistemi software, ecc.). Gli UCD sono il diagramma principale nella Use Case View. In molti modelli di processo software basati su UML, i casi d'uso (la Use Case View) sono la vista principale del sistema (processi "use case driven").
Uno Use Case Diagram può essere utilizzato in modo molto efficace nei primi stadi del ciclo di vita del software, in particolare in fase di analisi. Tali diagrammi consentono di individuare con chiarezza gli obbiettivi dell'utente e le interazioni del sistema con gli elementi esterni ad esso.
Può anche essere utilizzato, vista la sua semplicità sintattica, come supporto nella fase di specificazione dei requisiti.
Ogni use case può potenzialmente descrivere una caratteristica che il sistema deve soddisfare (ad esempio, svolgimento di particolari compiti, ecc.).



Le primitive utilizzate nella costruzione di uno Use Case Diagram sono:

- **actor**: *ruolo* che l'utente ha rispetto al sistema.
- **use case**: entità grafica che rappresenta tutto ciò che riguarda le funzionalità esterne richieste dal sistema. *Specifica il comportamento di un sistema o di una parte di un sistema ed e' una descrizione di un set di sequenze di azioni.* Si usano per catturare il comportamento del sistema in esame, senza dover specificare come il comportamento è realizzato.

Un actor esegue (nel senso che utilizza) uno o più use case. A sua volta uno use case può essere eseguito da più actor. Questa relazione tra actor e use case è stabilita mediante messaggi.
Possono essere presenti diversi tipi di relazioni tra use case:



include : si verifica quando un determinato comportamento si ripete in più casi d'uso e non si vuole ripetere la sua descrizione;

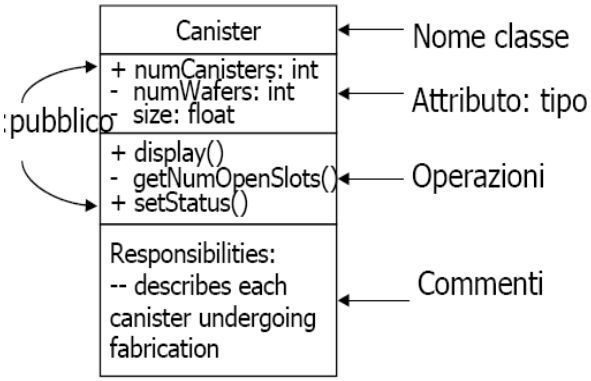
generalizzazione : si utilizza quando un caso d'uso è simile ad un altro ma fa qualcosa di più (generalizzazione nel senso dell'estensione). Questo permette di rappresentare scenari alternativi.
extend: è simile alla generalizzazione, ma ci sono più regole. Quando si utilizza questo costrutto il caso d'uso che estende un caso base può aggiungervi comportamento, ma stavolta il caso d'uso base deve dichiarare dei "punti di estensione" e il caso che lo estende può aggiungere comportamento solamente in corrispondenza dei punti specificati.

7.2. Class Diagram

-rappresenta le classi e gli oggetti che compongono il sistema, con attributi e operazioni
-specifica, mediante le associazioni, i vincoli che legano tra loro le classi
-può essere definito a livelli diversi (analisi, disegno)
-può rappresentare diverse tipologie di oggetti

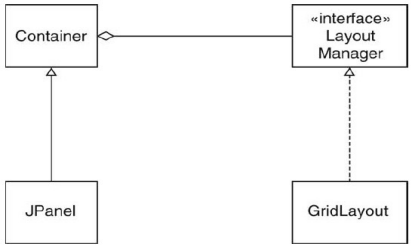
Una classe UML è indicata da un rettangolo suddiviso in tre parti distinte:

- nella prima parte è indicato il nome della classe;
- nella seconda parte sono elencati gli attributi della classe: agli attributi viene associata una visibilità: + public, # protected e – private.
- nella terza parte sono indicate le operazioni della classe(metodi: vale anche in questo caso il discorso della visibilità del punto precedente.



Le classi sono relazionate tra loro mediante associazioni. Le associazioni, oltre ad avere una cardinalità, possono anche avare un nome (role name) e possono essere navigabili (se sull'associazione è specificata una freccia). La navigabilità indica come verrà realizzata l'associazione da un punto di vista realizzativo.

- l'aggregazione, indicata con il rombo vuoto, che rappresenta una relazione di part-of in cui l'oggetto può essere condiviso in aggregazione da altre classi;
- la composizione, indicata con il rombo pieno, in cui si richiede la partecipazione esclusiva del composto nel componente, inoltre il componente è destinato a seguire le sorti del composto quando questo viene distrutto.



- Le gerarchia di generalizzazione (primitiva IS_A) è rappresentata da un collegamento terminante con una freccia vuota.

Nella programmazione i class diagram sono molto utilizzati per implementare varie strategie.

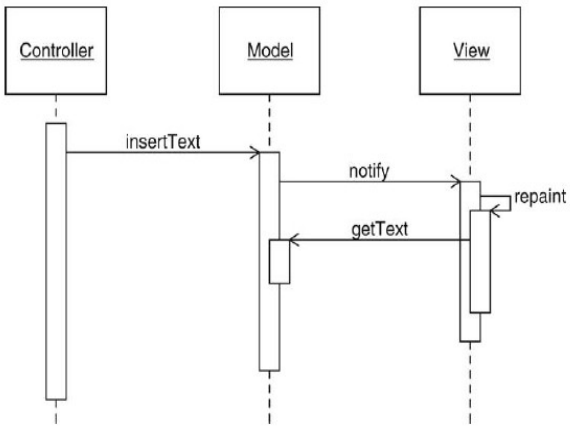
7.3. Interaction Diagram

Sono diagrammi che descrivono come gruppi di oggetti interagiscono in alcuni comportamenti. Si definiscono quindi degli **scenari**. In genere, si cattura il comportamento di Use Case, mostrando alcuni oggetti e i messaggi scambiati.
Ci sono due tipi di Interaction Diagram:

- Sequence Diagram

-evidenzia il modo in cui uno scenario (uno specifico percorso in un caso d'uso) viene risolto dalla collaborazione tra un insieme di oggetti
-specifica la sequenza dei messaggi che gli oggetti si scambiano
-può specificare nodi decisionali e iterazioni

Ogni oggetto è rappresentato da un box (contenete il nome dell'oggetto nella forma *objectName: className*), al di sotto del quale è presente una **lifeline** , la quale rappresenta la vita dell'oggetto durante l'interazione con gli altri oggetti. Ogni **messagge** è rappresentato da una freccia, al di sopra della quale è presente il nome del messaggio stesso. E' possibile la **self-delegation** (un messaggio spedito da un oggetto a se stesso). E' anche possibile indicare delle **condition**, che indicano le condizioni rispetto alle quali il messaggio può essere spedito. Con l'**iteration maker** (indicato con *) è possibile indicare la spedizione di messaggi multipli. A fianco di tale marcatore è presente una condizione che indica il criterio rispetto al quale avviene la spedizione multipla.



E' possibile indicare anche un **return**, mediante linea tratteggiata, che indica il ritorno da un messaggio, non un nuovo messaggio.

- Collaboration Diagram

-specifica gli oggetti che collaborano tra loro in un dato scenario, ed i messaggi che si indirizzano
-la sequenza dei messaggi è meno evidente che nel diagramma di sequenza, mentre sono più evidenti i legami tra gli oggetti
-può essere utilizzato a livelli diversi (analisi, disegno), e rappresentare diverse tipologie di oggetti
Rappresenta la stessa informazione e le stesse interazioni del Sequence Diagram, ma in un'altra forma. Infatti, la sequenza dei messaggi è indicata mediante numerazione progressiva (in vari formati). Molto meno usati poiché complessi ma molto potenti.

Sono diagrammi utili quando è necessario mostrare il comportamento di parecchi oggetti all'interno di un singolo Use Case.
L'utilizzo del sequence o collaboration diagram dipende da quello che si vuole mostrare:

- sequence diagram: enfatizza la sequenza dei messaggi;
- collaboration diagram: utile per vedere come gli oggetti sono connessi staticamente.

Sono diagrammi utili anche per gestire processi concorrenti.

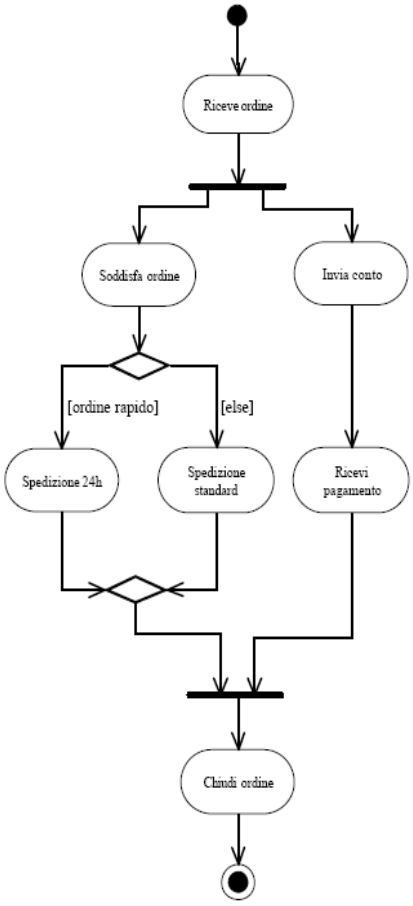
7.4. Package Diagram

Diagramma che mostra raggruppamenti di classi (**package**) e le **dipendenze** fra loro.
Un package diagram può essere visto come un'altra forma di class diagram. Una dipendenza esiste tra due package se esiste tra le classi costituenti il package.
E' possibile anche definire package che contengono altri package oppure definire la generalizzazione tra package (con lo stesso significato presente tra le classi).
Il package diagram è fondamentale per progetti di dimensioni medio grandi poichè permette di lavorare in team molto meglio e su sezioni a portata di progettista.

7.5. State Diagram
-specifica il ciclo di vita degli oggetti di una classe, definendo le regole che lo governano
-quando un oggetto si trova in un certo stato può essere interessato da determinati eventi (e non da altri)
- come risultato di un evento l'oggetto può passare ad un nuovo stato (transizione)

Sono un derivato di un altro linguaggio OMT.
Descrivono come avvengono i passaggi negli oggetti che hanno più stati.
Gli State Diagram sono utili per descrivere il comportamento di un singolo oggetto attraverso diversi Use Case.

7.6 Activity Diagram
-servono a rappresentare sistemi di workflow, oppure la logica interna di un processo (di qualunque livello, dai business process ai processi di dettaglio)
-permette di rappresentare processi paralleli e la loro sincronizzazione
-è un caso particolare di diagrammi di stato, in cui ogni stato è uno stato di attività



L'Activity Diagram descrive la sequenza delle attività e supporta un comportamento condizionale e parallelo. Si tratta di una variante del diagramma di stato in cui tutti gli stati hanno associata una attività. Gli elementi di un Activity Diagram sono i seguenti:

- **Activity**: dal punto di vista concettuale, indicano un lavoro che deve essere svolto. Dal punto di vista più strettamente implementativo, un'activity può essere un metodo di una classe. Da ogni activity possono uscire uno o più **transazioni**, che indicano il percorso da una activity ad un'altra.
- **Start e End Point**: punti di inizio e fine del diagramma. Gli End Point possono anche non essere presenti, oppure essere più di uno indicano dove inizia e termina l'attività in esame.
- **Branch e merge** : il comportamento condizionale è determinato da questi due costrutti. Il branch ha una singola transazione entrante e più transazioni uscenti in cui solo una di queste sarà prescelta. Il merge ha più transazioni entranti e una sola uscente e serve a terminare il blocco condizionale cominciato con un branch.
- **Fork e join**: il comportamento parallelo è determinato da questi due costrutti. Quando scatta la transazione entrante, si eseguono in parallelo tutte le transazioni che escono dal fork. Con il parallelismo non è specificata la sequenza. Per la sincronizzazione delle attività parallele è presente il costrutto di join che ha più transazioni entranti ed una sola transazione uscente.

Un Activity Diagram è particolarmente utile per descrivere:

- workflow process, e quindi business process;
- parallel process.

Da un punto di vista più vicino ad UML, tali diagrammi sono utili per:

- descrivere metodi di classe.
- descrivere use case.

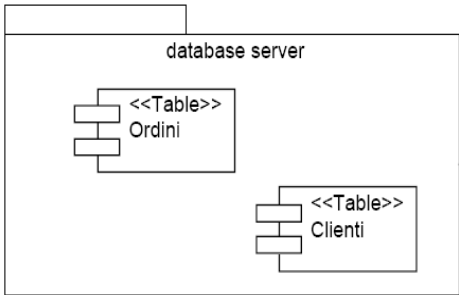
7.7 Implementation Diagram

Mostrano gli aspetti dell'implementazione, inclusa la struttura del codice e le strutture per l'implementazione run-time del progetto. Si dividono in

Component Diagram

-evidenzia l'organizzazione e le dipendenze esistenti tra componenti
-i componenti sono moduli software, dotati di identità e con un'interfaccia ben specificata
-i componenti (come a livello logico le classi) possono essere raggruppati in package

ostra le dipendenze fra componenti software.



I *component diagram* forniscono una vista fisica del modello corrente, mostrando l'organizzazione e le dipendenze fra componenti software, inclusi moduli di codice sorgente, di codice binario e componenti eseguibili. Questi diagrammi mostrano anche il comportamento visibile esternamente dei componenti, mostrandone le interfacce. Le dipendenze sono mostrate mediante opportune relazioni tra componenti ed interfacce.

I *component diagram* sono composti da:

- *component packages*

I component packages rappresentano un insieme di componenti logicamente correlati. In tal modo è possibile partizionare il modello fisico del sistema.

- *components;*

Un componente (component) rappresenta un modulo software con una ben definita interfaccia.

- *interfaces*

L'interfaccia è rappresentata da uno o più elementi d'interfaccia che il componente fornisce. I componenti sono usati per rappresentare dipendenze tra i moduli (a tempo di compilazione, run-rime, di chiamata). E' anche possibile visualizzare le classi implementate nel modulo.
Un sistema potrebbe essere composto da parecchi moduli software di diversa specie. Per distinguere le differenti specie di moduli, è utilizzato il concetto di stereotipo
Un cerchio attaccato all'icona del componente significa che questo supporta una particolare interfaccia. Un cerchio attaccato all'icona del componente significa che questo supporta una particolare interfaccia. Le interfacce sono delle classi appartenenti al componente, e definite nel diagramma delle classi.

- *dependency relationships*

Questo tipo di relazione sta ad indicare che le classi contenute nel componente *client* usano le classi contenute nel componente *supplier.*

Deployment Diagram

-evidenzia la configurazione dei nodi elaborativi in ambiente di esecuzione (run-time), e dei componenti, processi ed oggetti ubicati in questi nodi
-permette di rappresentare, a diversi livelli di dettaglio, l'architettura fisica del sistema

mostra l'integrazione del programma con la struttura hardware e software. Gli elementi costituenti il diagramma sono:

- ***processors***: componente hardware in grado di eseguire programmi. Opzionalmente è anche possibile visualizzare i processi attivi sul processore e il tipo di *scheduling*;
- ***device***: componente hardware che non è in grado di eseguire programmi;
- ***connection***: rappresenta un accoppiamento hardware tra due entità.

8. PER COSA USARE UML

L'utilizzo di UML da parte delle aziende ricade tipicamente in una delle seguenti tipologie:
Utilizzo come strumento di documentazione a *posteriori*.

- Usare UML come *strumento di documentazione* significa procedere con l'analisi, design, implementazione, ecc prescindendo da UML: ovvero, lavorare "come prima". Questo, per molte aziende, significa condurre l'analisi in linguaggio naturale, ed il design in modo misto (linguaggio naturale e codice o pseudo-codice), non di rado in modo indistinguibile dalla codifica stessa. In tal caso UML interviene *a posteriori*, quando vogliamo documentare il risultato dell'analisi e/o del design.
  La documentazione è un compito non semplice, non sempre piacevole, e spesso percepito come un overhead rispetto ad altre attività come la codifica. UML può essere utilizzato come notazione ragionevolmente espressiva e ragionevolmente compatta per sostituire (in parte) la documentazione puramente testuale.
  Questo approccio all'adozione di UML non presenta particolari difficoltà, né richiede particolari investimenti sotto il profilo degli strumenti. In teoria, potremmo usare un tool di diagrammazione generale, non studiato appositamente per UML (in pratica, è spesso più comodo usare comunque uno strumento mirato). Non è neppure necessario che tutti i membri del team di sviluppo imparino a "scrivere" in UML: è sufficiente che tutti imparino a *leggere* i diagrammi UML, e che alcuni imparino a "tradurre" in UML i risultati di analisi, design, e probabilmente di parte della codifica. D'altro canto, come capita con molti investimenti modesti, anche il ritorno potenziale è piuttosto modesto: una documentazione più compatta e (con un minimo di fortuna e capacità) più precisa di quanto avremmo prodotto senza l'uso di UML.
- Utilizzo come strumento *attivo* di analisi e design.
  Il salto dalla modalità precedente ad un uso *attivo* di UML, *durante* le fasi di analisi e design, è apparentemente breve ma in realtà molto profondo. Si tratta di imparare a ragionare con l'ausilio di *rappresentazioni esterne multiple*, anziché in base alla sola rappresentazione interna (mentale). È importante capire la differenza tra *creare un diagramma* come traccia dell'analisi o del design e *usare un diagramma* per *ragionare* sul modello di analisi o design. Si tratta di imparare ad utilizzare *anche* la percezione visivo-spaziale come parte delle tecniche di gestione della complessità; si tratta anche di imparare ad intuire la dinamica di un sistema osservandone una rappresentazione statica.
  Vi è uno stretto legame fra psicologia e *l'apprendimento* dell'uso dei diagrammi come strumenti di pensiero. Tale processo deve essere arricchito e ciò richiede ore tempo, e investimenti: in questo caso, infatti, avere a disposizione strumenti pensati appositamente può rendere il compito più naturale.
  Il risultato dell'approccio "attivo" è duplice. Da un lato, infatti, otteniamo documentazione di analisi e design basata su UML, prodotta durante le fasi in questione, non a posteriori. Dall'altro otteniamo (ed è il punto centrale) possiamo avere maggiori chance di ottenere un prodotto più aderente ai bisogni dell'utente, e dotato di tutte quelle caratteristiche spesso promesse (e meno frequentemente mantenute) dall'approccio object oriented (riusabilità, estendibilità, ecc...).
  Questo è l'approccio che dà i migliori risultati, sia in termini di rapporto costo/benefici, sia in termini assoluti di qualità del prodotto risultante.
- Utilizzo come strumento di sviluppo, attraverso il cosiddetto *round-trip*.
  L'idea di fondo è di dotarsi di strumenti migliori, in grado di generare codice a partire dal modello, e di riportare le modifiche effettuate manualmente nel codice all'interno del modello. Si cercano rappresentazioni sempre più astratte, generando in modo quanto più automatico possibile le rappresentazioni più concrete.
  Questo lega fortemente gli sviluppatori a un tool.

9. COMPLESSITA' DI UML
   UML intende rappresentare qualunque tipo di sistema software, a livelli di astrazione differenziati, introduce quindi un elevato numero di elementi, e in molti casi è possibile scegliere tra forme di rappresentazione diverse.
   UML non suggerisce, né tantomeno prescrive una sequenza di realizzazione dei diversi diagrammi.
10. ESTENDIBILITA E FUTURO
    UML include tre meccanismi che consentono l'estensione della sua sintassi e della sua semantica da parte dell'utente: stereotipi, tagged values e constraints. Questi strumenti possono essere usati nel contesto di un modello per esprimere concetti altrimenti non rappresentabili in UML, o non rappresentabili in modo chiaro, sufficientemente astratto, e così via. I profili UML sono collezioni di stereotipi, tagged values e constraints che specializzano il linguaggio per particolari domini applicativi o per l'uso di UML in congiunzione con particolare tecnologie. Fra i profili riconosciuti ufficialmente da OMG si trovano profili per CORBA, per i sistemi distribuiti, per sistemi con vincoli di QoS e per sistemi real-time.

Fonti:

-Wikipedia: http://it.wikipedia.org/
-UML manuale di stile Dott. Carlo Pescio: http://www.eptacom.net/umlstile
-Introduzione a UML Adriano Comai1998-2001

Un interessante articolo si trova su MokaByte Numero 34 - Ottobre 99
Una buna raccolta di appunti è situata su http://www.morpheusweb.it

Riccardo Prandini