Lezione 7

Lunedì 21 Ottobre 2013
Luca De Franceschi

Introduzione a UML

Linguaggio usato dal mondo professionale (e didattico) che è molto utile conoscere.
Tutto ciò che vedremo su UML servirà soprattutto nel modulo B e nello scritto.
Vista la grandezza del progetto c'è necessità di un linguaggio formale per poter dialogare con i colleghi. Passare da un *linguaggio naturale* a un *linguaggio compilato* è difficile e servono degli strumenti per effettuare questa transizione.

UML è un linguaggio per modellare concetti nel mondo ad oggetti. Una serie di notazioni puramente *grafiche* che hanno dei **vincoli**, c'è una sintassi e un meta-modello che serve a dare un significato alle notazioni grafiche (semantica). UML è puramente visuale è ha alcune caratteristiche:

- **Astratto** → ognuno ha esperienze differenti su diversi linguaggi, quindi UML deve essere indipendente dal linguaggio di *implementazione*.

- **Flessibile** → si può parlare di qualsiasi cosa con UML.

- **Moderno** → progettato per sw *scalabili*, *distribuiti* e *concorrenti*. Scalabilità *orizzontale* e non *verticale* (in cui aggiungo risorse).

Ci sono molti strumenti che forniscono delle *utility* per scrivere con questo linguaggio. Traduco ciò che ho disegnato in un linguaggio di programmazione. UML lo useremo in tutte le fasi di sviluppo sw. E' un *standard* ed è mantenuto dall'OMG, che incorpora molte aziende al suo interno.

Il linguaggio naturale è troppo astratto, UML concretizza il linguaggio naturale. Ho aziende molto grandi che partecipano alla definizione di UML. CI sono vari modi di utilizzarlo.

All'inizio è un po' vago, quindi si parla dell'utilizzo di UML **come "abbozzo"**. All'inizio si parla solo di interfacce in modo astratto. UML definisce una specifica tecnica nella prima fase di progetto. Serve per la documentazione e per descrivere porzioni di sistema. Man mano che il progetto progredisce bisognerebbe scrivere documentazione (in base al tempo che ho). Ma non tutto può essere parallelizzato... La documentazione è molto importante farla a priori e non a posteriori (a meno di emergenze).

Posso avere un approccio più ingegneristico in cui andrò a scendere più nel dettaglio per vedere se le mie idee hanno senso. UML non più come abbozzo ma disegna **diagrammi più dettagliati** (compito del progettista) in modo da facilitare i programmatori e "imbrigliare" la loro capacità creativa. Bisogna dare delle specifiche esatte e precise, senza ambiguità. La documentazione deve essere in questo caso più stabile, perché altrimenti l'onere della manutenzione diventa eccessivo.

UML come **linguaggio di programmazione**. E' possibile generare codice a partire dalle notazioni; è un approccio ottimale ma meno utilizzato. Può essere inoltre usato sia per parlare di sw che di *concetti*. Abbiamo un grado conoscitivo più alto, perché dovremmo insegnare UML anche a persone che non sono informatiche.

UML è nato nel 1997 ed è ancora relativamente “giovane”. Due tipologie di gruppi di sviluppo:

1. Sistema gerarchico piramidale; il cliente sta fuori (contattato solo all'inizio e alla fine). Un approccio chiuso CLIENTE-FORNITORE. Modello usato fino a 6-7 anni fa. Nel basso sta chi va a scrivere codice.

2. Può succedere però che il cliente voglia avere risultati parziali; in questo caso è meglio il metodo *agile* con due vincoli:

    - devo avere un gruppo di persone *molto intelligenti* (mai oltre 10 persone esperte). Le persone sono tutte alla pari e collaborano insieme, e il cliente è uno di loro.

    - deve essere molto intelligente anche il cliente.

Ogni tot giorni si fa un *ciclo* e si “butta fuori qualcosa” (si produce *prototipi*) e si fanno emergere nuovi requisiti. In questo caso UML è più difficile da usare.

Con UML non ho vincoli prescritti, non dice “cosa non dobbiamo fare”, si basa su precise regole descrittive. Bisogna tenere un buon grado di dettaglio. Bisogna fin da subito mettere in pratica. I diagrammi UML sono tantissimi, non ne vedremo una parte, quelli più importanti.

Diagrammi dei casi d'uso

Sono fra i più complessi perché vengono usati maggiormente in *fase di analisi*. Definisce in modo formale le caratteristiche che il prodotto deve avere (usato dagli *analisti*). Vengono usati nell' **analisi dei requisiti**, descrive le caratteristiche in modo formale producendo un documento. E' un documento che si dà al cliente, il quale lo deve approvare e firmare. I diagrammi ci vengono in aiuto per definire i requisiti funzionali. Si parla di funzionalità **verso l'esterno**, non ci concentriamo su come i componenti devono essere implementati (es. “bisogna fare la pagina di login”) → pura interfaccia verso l'esterno. Le nostre funzionalità possono essere chiamati *scenari*, sequenze di passi che il sistema e gli utenti (*attori*) devono fare per arrivare al proprio scopo. Ci possono essere **scenari principali** o **alternativi**, che si verificano sulla base di alcune condizioni.

- **Scenario**
- **Attori**
- **Scopi**

Un caso d'uso è un insieme di scenari che hanno in comune uno scopo per un utente. Permette di disegnare la *facciata* del sistema “fregandosene” dei dettagli implementativi.

Attori → il mio sistema deve interagire sia con persone umane sia con sistemi esterni che interagiscono con il mio (es. login tramite facebook). Attori visti come raggruppamento delle mie funzionalità di sistema. Per identificare gli attori ci si chiede se è un qualcosa che sta interagendo dall'esterno. Gli attori sono **sempre esterni** al sistema.

I diagrammi dei casi d'uso sono **puro testo**. Si parte a vederne la descrizione, poi in realtà ci si attacca uno “schemetto” per facilitarne la lettura. Componenti:

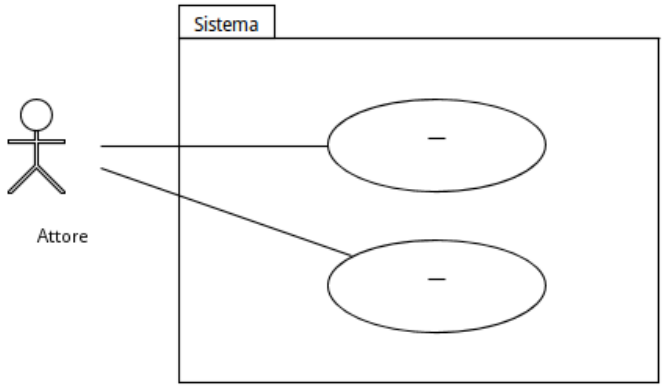
- **Nome/Identificatore**

- **Scenario principale** → es. autenticazione.

- **Scenari alternativi** → es. messaggi di errore o eccezioni.
- **Pre-Condizioni**
- **Effetti/Garanzia** → post-condizioni.
- **Trigger** → evento scatenato dal caso d'uso.
- **Attori principali** → chi vuole ottenere lo scopo.
- **Attori secondari** → es. facebook quando autentico esternamente.

Ho sempre un solo scenario principale per ogni caso d'uso ma posso avere diversi scenari alternativi. Inizialmente andrò a definire dei casi d'uso di alto livello. Successivamente ho un processo di raffinamento e vado a dettagliare il modello, scendendo di livello. E' meglio intervallare al testo degli elementi grafici, ricordando comunque che la componente testuale è la più importante.

I diagrammi dei casi d'uso sono dei *grafi*, in cui i nodi sono o un attore o un altro caso d'uso.



Relazioni:

- **Associazione** → linea solida senza frecce;
- **Inclusione** → freccia tratteggiata che va da un caso d'uso a un altro;
- **Estensione** → freccia tratteggiata che va da scenario alternativo a scenario principale;
- **Generalizzazione** → freccia solida con punta “cava”.

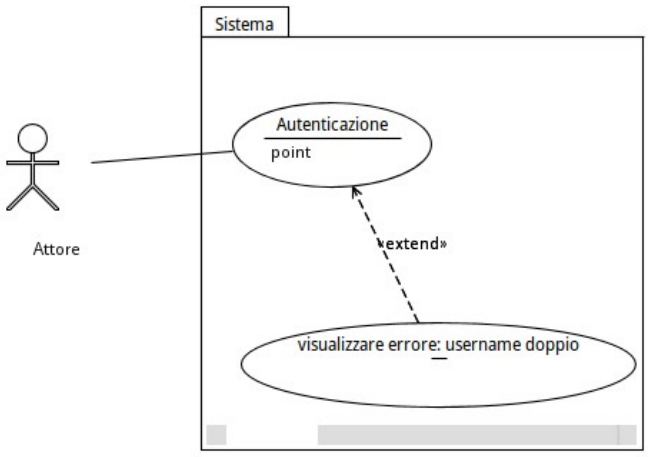
Vi possono essere relazioni anche tra i casi d'uso, che sono sempre tratteggiati e con un verso.

Relazione di inclusione → stiamo dicendo che un caso d'uso A ogni qual volta viene utilizzato da un attore scatena l'utilizzo di un caso d'uso B (*sempre*). Spesso un caso d'uso viene *riutilizzato* in più casi d'uso e quindi posso scriverlo una volta sola. L'inclusione lavora su casi d'uso che sono allo stesso grado di astrazione.

La fase di progettazione dell'interfaccia, di come l'utente interagisce nel sistema è compito

dell'*analista*.

L'**estensione** è diametralmente differente rispetto all'inclusione. La direzione è opposta. Non va immaginata come l'"*extends*" di Java, non stiamo parlando di ereditarietà. Se durante l'esecuzione di un caso d'uso A accadono alcune condizioni, allora l'esecuzione di A termina e parte B. Relazione *condizionale*. Per la maggior parte dei casi è un modello che descrive errori o eccezioni.



Inclusione ed estensione hanno degli aspetti in comune. L'*inclusione* è senza alcuna condizione mentre l'estensione è *condizionale*.