Lezione #22

Mercoledì, 27 Novembre 2013

Tullio Vardanega, 09:30-11:15

Luca De Franceschi

Un problema grosso nella produzione del codice è che non siamo in generale preparati ad avere una buona strategia di **gestione degli errori**:

- **Relativi al flusso di dati**, non posso assumere che l'utente inserisca sempre dati corretti, ma allo stesso tempo non posso accettare *spazzatura*. Sugli input non posso fare assunzioni ottimistiche. Problematiche relative alla disponibilità di un dato.
- **Relativo al flusso di controllo**, le azioni che avvengono, gestione di eccezioni che io sollevo o che il sistema genera. Le eccezioni sono problemi che rendono il flusso di controllo instabile, e devo avere una strategia di gestione, altrimenti possono essere dannose.
- **Relative al trascorrere del tempo**, ci possono essere degli impegni che il programma si è preso nei confronti dell'utente (es. mi aspetto che il programma risponda in tot secondi). Devo mettere quindi un limite rispetto al quale mi aspetto qualcosa. L'utente si immagina che il sistema garantisca delle funzionalità in un tempo finito.

Mai programmare e progettare immaginando che le cose vadano a buon fine, fare sempre delle assunzioni pessimistiche. Si può fare progettazione parallela solo se ho costruito un'architettura progettuale che garantisca il disaccoppiamento. Devo garantire l'integrità concettuale, desiderabile in ogni architettura di sistema. L'architettura non è *assembleare*, si fanno scelte su buone norme di progettazione che stanno all'architetto (progettista). Il raccordo tra programmazione e progettazione deve essere esplicito, ci deve essere coerenza. **Enforce intentions**, l'atto di assicurarsi conformità, "fai in modo che il codice realizzi precisamente le indicazioni della progettazione". Ciò che dico nell'architettura deve essere vero nel codice. La programmazione non deve fare scelte libere, ma rispettare la progettazione. Fare **programmazione difensiva**, essere esplicitamente pronti a trattare errori, si fa in due modi:

- **Incapsulamento degli errori**, in tutti i luoghi del codice dove so che posso non avere successo devo poter gestire l'errore. Programmare esplicitamente il trattamento dei possibili errori, **errori dei dati in ingresso** ed errori logici;
- La strategia di trattamento va prevista nella progettazione.

Metodi di trattamento:

- Attendere fino all'arrivo di un valore legale;
- Assegnare un valore predefinito (*default*);
- Usare un valore precedente;
- Registrare l'errore in un *log*, per avere un registro di manutenzione;
- Sollevare eccezioni (se ho un gestore delle eccezioni);
- Abbandonare il programma (se proprio non ce la faccio).

Fonti di errori logici:

- **Aritmetica in virgola mobile**, la sua approssimazione può cumularsi e condurre a errori importanti e a confronti svianti. Essa è intrisecamente imprecisa, sono operazioni spesso fonti di errore;
- **Puntatori e limiti delle strutture**, i puntatori sono in generale molto pericolosi. Nel momento in cui manipolo indirizzi porto cose ovunque (*segmentation fault*). Devo avere una programmazione che mi garantisca che queste cose non accadano. Voglio imporre che ai dati ci si acceda attraverso *metodi di interfaccia*;

- **Allocazione dinamica della memoria**, può portare a esaurimento della disponibilità e alla sovrapposizione di aree sensibili (il garbage collector arriva di tanto in tanto...). CHi fa un programma deve sapere quante *new* può fare (oppure non le fa);
- **Ricorsione**, può portare all'esaurimento della memoria o alla non terminazione. La ricorsione in ogni applicativo ragionevolmente serio è un concetto **proibito**, consuma lo stack. La combinazione di *new* e ricorsione è ancora peggio;
- **Concorrenza**, se mal progettata può condurre a errori.

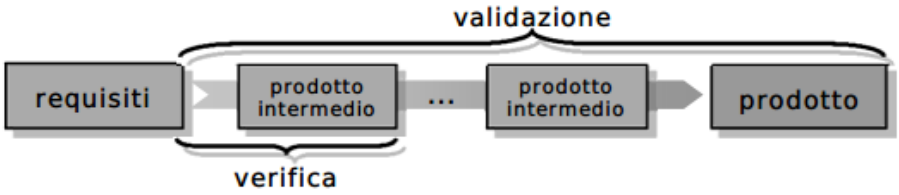
Verifica e validazione

- **Verifica**, "ho fatto il sistema nel modo giusto";
- **Validazione**, "ho fatto il sistema giusto".

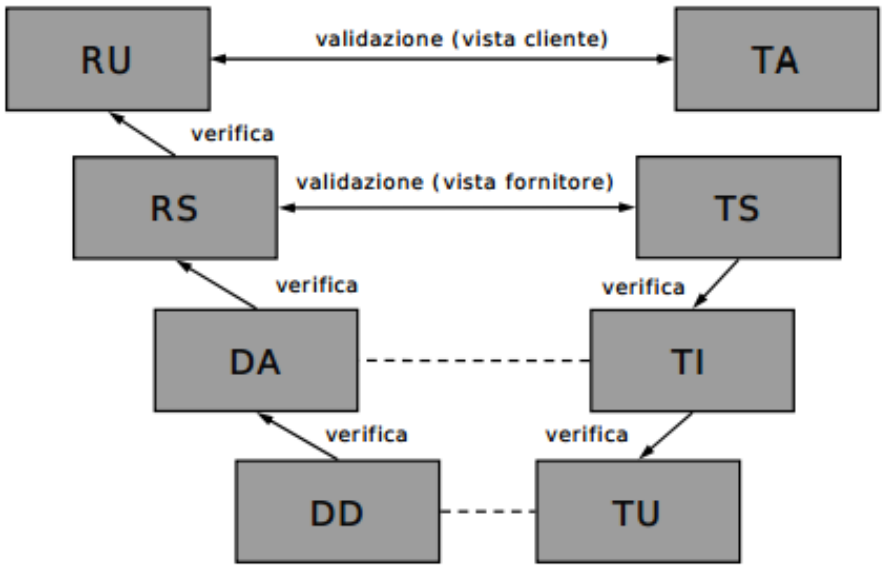
La *software verification* ricerca la completezza e la correttezza del software e tratta ciò che lo supporta. Consente di valutare di conseguenza che il sw sia validato. La verifica è a supporto della validazione e la validazione è l'ultima cosa che faccio in un progetto. La verifica è un attività che svolgo durante **tutto lo sviluppo** fino all'ultimo istante dove farò validazione, che servirà a dire che ciò che ho fatto è la cosa giusta. La verifica va fatta per impedire che la risposta finale non sia sbagliata. Devo garantire tre cose importanti:

- **Consistenza**, "sono ciò che vi attendevate fossi";
- **Correttezza**, "ciò che ho conseguito è corretto rispetto alle norme";
- **Completezza**, "tutto ciò che ho creato è tutto ciò era atteso".

Sono tre caratteristiche di cui devo accertare l'esistenza su tutti i prodotti parziali dello sviluppo. Il verificatore impara le norme e dice che quello che è stato fornito è fatto come richiesto. Non corregge nè rifà il lavoro ma controlla solo che tutto rispetti le tre caratteristiche. La validazione conseguentemente è una conferma **by examination**, mostra copertura dei requisiti (utente e sw).



Le milestone nel progetto didattico sono minimo le revisioni, poi ciascun gruppo può fissarne di intermedie.

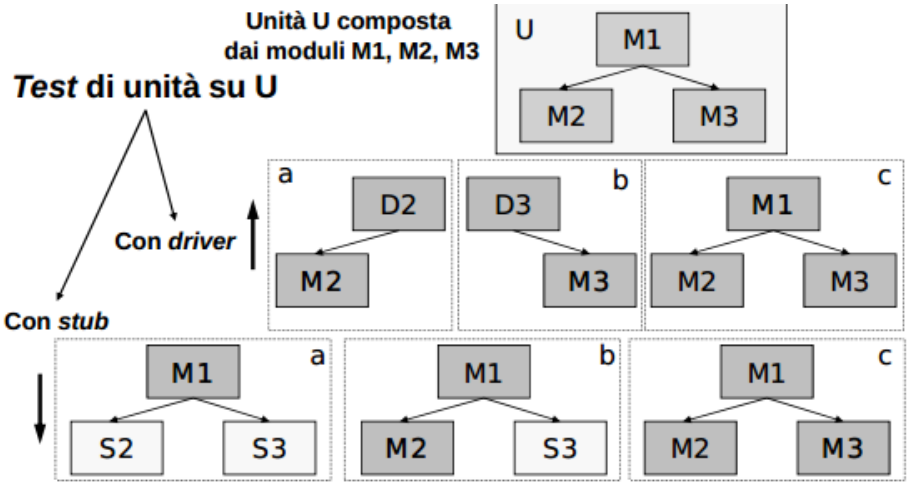


Per il verificatore ho due forme di **analisi**:

- **Analisi statica**, non richiede l'esecuzione del programma, studia le caratteristiche del codice sorgente (e a volte anche del codice oggetto);
- **Analisi dinamica**, richiede l'esecuzione del programma, viene effettuata tramite **test**, usata sia nella verifica che nella validazione.

Fissare alcune caratteristiche dell'ambiente dinamico:

- **Ripetibilità**, è un requisito essenziale. Dobbiamo assumere uno stato inziale prima dell'esecuzione. IL test deve essere **deterministico** ed eseguire le cose secondo un ordine noto. Lo stato iniziale è tutto ciò che ha influenza diretta o indiretta sull'esecuzione, e devo sapere qual'è. **Specifica di un test**;
- **Strumenti**:
  - **Driver**, componente attiva fittizia per pilotare una parte;
  - **Stub**, componente passiva fittizia per simulare una parte;
  - **Logger**, componente non intrusiva di registrazione dei dati di esecuzione per l'analisi dei risultati. Ogni tanto deve lasciare traccia del suo esito;
- **Unità**, un insieme piccolo di classi che è possibile assegnare a un programmatore. La più piccola unità sw che è conveniente verificare singolarmente.



Una volta verificata ogni singola unità metto insieme quelle raggruppate e faccio **test di integrazione**.