Considerazioni strategiche(11)

Il segmento di ciclo di vita attivato nel progetto didattico va dalla RR *Revisione dei Requisiti* alla RA *Revisione Accettazione*. Il modello di sviluppo interno e scelta autonoma del fornitore e determina piano e strategia di utilizzo delle risorse.
Per il committente non è positivo avere un modello sequenziale, perché non avrà mai visione del prodotto se non alla fine, non riceve prototipi e quindi perde interesse nel progetto, e noi vogliamo un proponente che sia **attivo** e interessato al nostro prodotto. Una volta acquisita l'esperienza necessaria allora posso usare un modello un po' più agile.
Ciascuno di noi deve formare un gruppo portando un proprio **calendario** già fatto, fissando dei vincoli e delle previsioni strategiche. Bisogna far emergere una disciplina ed avere alcune accortezze. Si parla dunque di **pianficazione**. Un *mese-persona* vale circa 142 ore. Il modo in cui gestisco tempo e persone è molto delicato:

- Vi sono componenti di impegno **non comprimibili**, ovvero non posso mettere più persone sullo stesso compito, non posso svolgerlo in parallelo. Non posso frantumare in piccole parti da parallelizzare (es. programmazione o verifica). Le cose sulle quali riesco a comprimere sono poche;
- Vi sono compiti **non partizionabili**;
- La verifica a livello di sistema si fa **solo alla fine**, perché non sono test parallelizzabili e il sistema diventa disponibile solo alla fine dello sviluppo.

Occorre avere una pianificazione che abbia margini e che sia completamente consapevole dei vincoli. Una buona progettazione consente di non cadere nella iterazione non controllata. In questo modo, con queste tecniche si migliora la *mitigazione dei rischi*.
Il modello sequenziale è esattamente coerente con quello che si aspetta il committente, ma non per quanto riguarda il proponente.

Il **modello incrementale** non è iterativo, l'iterazione *distrugge* per sostituzione, potenzialmente pericolosa. La posso avere solo in situazioni di emergenza. Un ciclo incrementale può essere visto come un "*for*". Ogni passaggio di questo ciclo aggiunge cose, mi avvicino alla soluzione per approssimazioni non distruttive. Posso anche non incrementare ma l'importante è che il numero di iterazioni sia noto. Dentro una chiamata di RP posso portare più di un incremento, tuttavia il progetto didattico prevede una singola occorrenza di ogni revisione. Il fornitore deve realizzare altre verifiche interne senza il coinvolgimento del committente.

Modello evolutivo. E' un modello che approssima la soluzione finale ammettendo tante iterazioni, abortendo le versioni intermedie. Per poter attuare un modello evolutivo ho bisogno di tanta energia. E' una tecnica molto interessante ma con un enorme costo, infatti revisioni successive possono avere come oggetto versioni di prodotto diverse.
Con un modello sequenziale combatto per abbattere i rischi mandando molto avanti le funzionalità visibili, l'avanzamento infatti viene fatto solamente quando si è sicuri.
Con un modello incrementale riesco ad avere delle funzionalità molto prima, tuttavia ho molti più rischi possibili.
Il **modello agile** non è facilmente rappresentabile. Si ragiona sulle cose da fare (**backlog**). Fra le cose da fare in un modello agile le persone prendono liberamente quello che faranno (ciascuno pesca un post-it a seconda del proprio estro). In un modello agile l'essenziale è che per ogni cosa fatta l'effetto sia visibile (**incremental build**). Ogni aggiunta rende il prodotto sempre più vicino alle aspettative, anche se non ha un ordine particolarmente ovvio. L'unico ordine è che ci siano tante cose che posso vedere e che rappresentano ciò che il prodotto sarà . E' un modello molto interessante ma difficile da gestire.

Non tutti i problemi hanno una (buona) soluzione. Bisogna fissare con la massima chiarezza:

- **Obiettivi**;
- **Vincoli**;
- **Alternative**;
- **Rappresentazione del problema e delle sue soluzioni**.

la qualità cardine resta comunque **Fattibilità** e **Verificabilità**
Tecniche progettuali:

- **Decomporre** in modo modulare e senza dipendenze. Una buona decomposizione identifica componenti tra loro indipendenti (a basso accoppiamento e funzionalmente coesi). La seconda cosa che voglio fare è nascondere il dettaglio implementativo (**incapsulamento**).
- **Accoppiamento:** è la misura dell'intensità della relazione tra parti distinte.
- **Coesione:** è la misura dell'intensità della relazione all'interno di una singola parte.
- **Astrazione:** omettere informazione per poter applicare operazioni simili ad entità diverse. Non importa la forma esatta che ha una cosa ma che abbia informazioni e funzionalità utili, non voglio sapere tutto, ma le cose importanti in un determinato contesto.
- **Atomicità:** è un altro criterio molto importante. L'utilità dell'astrazione non migliora se divido ulteriormente, perché scomporre più del dovuto ha un costo.

Vediamo ora alcune problematiche critiche della progettazione:

- **Concorrenza:** è molto importante garantire al sistema una concorrenza, ma va ben gestita e se la uso in modo inconsapevole faccio solo danni.
- **Distribuzione:** se e come i componenti sono disseminati su più nodi di elaborazione e come comunicano tra di loro.
- **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.

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. 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.
La **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.

Possiamo gestire due tipi di errori in diversi modi:

- **Errori Dati:**
  - 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).
- **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.