- ***<u>Lezione 20</u>***

  ➔ *Ottavo Blocco*

Ci si occuperà, in questa seconda parte del corso, delle fasi successive a quella di specifica dei requisiti.
Una volta capito cosa deve fare il software, saranno i progettisti e programmatori ad occuparsi del come la realizzazione deve aver luogo.
Anzitutto fase di progetto, poi soluzione progettuale da tradurre in codice dai programmatori. La Fase di Progetto prende in input il documento di specifica (analisi requisiti) e produce il Documento di Progetto che guida la successiva fase di codifica.
Ancora una volta, per continuità, faremo uso di un approccio object-oriented per vedere come quanto fatto nella prima parte del corso viene utilizzato per arrivare a progettare il software.
Tuttavia, al di là dello specifico approccio di cui si fa uso, ogni fase di progetto può essere suddivisa in due sottofasi:
- Progetto Architetturale (o preliminare) → dove viene definita l'architettura del software, ossia il prodotto viene partizionato in più componenti (decomposizione modulare) capendo anche come queste interagiscono l'una con l'altra
- Progetto Dettagliato → dove ciascuna componente viene progettata in modo dettagliato, scegliendo algoritmi e strutture dati specifiche.
In questa parte introduttiva, descriviamo anzitutto i principi fondamentali di progettazione del software per poi vederli concretamente applicati al caso object oriented.
Tra questi principi:

- *Information Hiding, Riusabilità* → (entrambi da applicare sia qui che in fase di implementazione)

- *Stepwise refinement* → (procedere per raffinamenti successivi delle varie componenti senza procedere sequenzialmente, già visto anche in fase di analisi dei requisiti). In ambito di progettazione software il procedere per raffinamenti successivi è un processo di elaborazione che aggiunge di volta in volta solo i dettagli necessari per quella particolare attività che deve essere svolta.
  In particolare si parte dalla specifica di una funzione (o di dati) in cui ancora non è descritto il funzionamento interno/struttura interna dei dati, per poi di volta in volta aggiungere un livello di dettaglio maggiore (da qui raffinamento).
  Perché è importante procedere per raffinamenti successivi? Non potrei considerare i dettagli tutti insieme? No, come suggerisce la Legge di Miller (precedente all'ingegneria del software) ogni essere umano può concentrarsi al più su 9 elementi differenti contemporaneamente. Il concetto di Raffinamento è complementare a quello di Astrazione.

- *Astrazione* → (usato anche in fase di analisi dei requisiti). Consiste nel concentrarsi solo sugli aspetti essenziali ignorando i dettagli secondari. Il concetto di livello di astrazione è stato introdotto da Dijkstra parlando di sistemi operativi per descriverne l'architettura a strati. Nell'ambito del processo software, ogni passo rappresenta un raffinamento (scendo più in dettaglio) del livello di astrazione della soluzione.

Due principali tipi di astrazione:
- Procedurale (es. le funzioni: funzione printf stampa e non serve che io che la uso sappia come funziona il codice)
- dei Dati (es. data encapsulation, ossia utilizzo di una struttura dati che astrae l'insieme di azioni eseguite su di essa come Pila meccanismo LIFO.)
Un tipo di dato astratto (abstract data type) combina l'astrazione procedurale e dei dati, identificando un tipo di dato insieme alle azioni eseguite sulle istanze del tipo di dato.
Nell'approccio Object Oriented il tipo di dato astratto è rappresentato dalla classe (che contiene sia dati (attributi) che funzioni (metodi)).
Vantaggioso in termini di manutenibilità e riusabilità in quanto se devo modificare specifiche funzioni che agiscono su variabili mi concentro solo sull'entità aggregata (classe), so già dove andare a cercare.
(quindi l'uso del tipo di dato astratto migliora gli attributi di qualità del software di manutenibilità e riusabilità dove qualità == capacità di rispondere alle aspettative degli utenti)

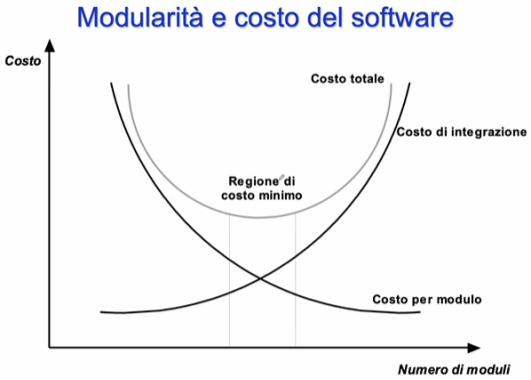
- *Modularità* → (capire il criterio per identificare i moduli). Ne abbiamo già parlato quando abbiamo parlato dei modelli di ciclo di vita del software. In particolare, parlando del modello incrementale, l'idea era di suddividere il prodotto in una serie di incrementi successivi ed ognuno di essi, una volta rilasciato, andava ad aggiungersi alla totalità già rilasciata.
  Ciò forniva molti vantaggi, e sono quegli stessi vantaggi che hanno portato la modularità ad essere un principio fondamentale nella fase di progettazione.
  Se si dovesse mantenere un blocco monolitico per il software si avrebbero difficoltà nel mantenerlo (non sono chiare le responsabilità), correggerlo, capirlo ed eventualmente riusarlo.
  La soluzione a ciò è suddividere il prodotto in segmenti più piccoli detti moduli software.
  Ma quali sono i criteri per identificare i moduli? Lo capiamo dalla definizione standard della modularità: la modularità è il grado di cui un certo software è costituito da un numero di componenti discrete (moduli) tali che la modifica di un componente abbia impatto minimo sugli altri.
  Quindi il criterio che deve guidare l'attività di decomposizione modulare è identificare moduli quanto più indipendenti l'uno dall'altro, in quanto se poi voglio modificare un modulo voglio che questa modifica non abbia grande impatto sugli altri di modo da ridurre l'effort nella manutenzione e riusabilità!

- *Decomposizione Modulare* → Ma cos'è un modulo? Rappresenta un elemento software che:
  - contiene istruzioni, logica e strutture dati (sia definizione variabili che il loro utilizzo)
  - può essere compilato separatamente e memorizzato in una libreria software
  - può essere incluso in un programma
  - può essere usato invocando segmenti di modulo identificati da nome e lista di parametri
  - può usare altri moduli

Esempi del concetto di modulo possono essere le funzioni o le classi (le funzioni in particolare rispettano il quarto requisito perché il segmento che può essere invocato è uno solo e rappresenta la loro stessa firma).
Il risultato ottenuto dopo la decomposizione modulare è detto architettura dei moduli.
Quindi in generale la decomposizione modulare si basa sul principio del "divide et impera" per cui dividere in moduli mi permette di ridurre la complessità e quindi l'effort, in particolare il costo di riaggregazione è tale per cui la complessità e quindi l'effort di risolvere due moduli insieme è maggiore rispetto a risolverli separatamente per poi riaggregarli.

$$C(p1) > C(p2) \Rightarrow E(p1) > E(p2)$$
$$C(p1+p2) > C(p1) + C(p2)$$
$$\Downarrow$$
$$E(p1+p2) > E(p1) + E(p2)$$

Il + rappresenta null'altro quindi che la fase di integrazione successiva a quella di codifica, in cui i vari moduli vengono rimessi insieme.
Abbiamo quindi capito i criteri per definire i moduli, ma come possiamo misurare concretamente questa decomposizione per capire quale è la migliore?
Si utilizzando due metriche: la coesione e il coupling.
L'obiettivo che si vuole raggiungere è massima coesione (cohesion) interna ai moduli e minimo grado di accoppiamento (coupling) tra moduli diversi.
Infatti massima coesione e minimo coupling permettono di incrementare come detto comprensibilità, manutenibilità, estensibilità (come sappiamo la manutenzione più frequente non è quella correttiva ma perfettiva, estendere le funzioni di un prodotto) e riusabilità del prodotto software.
La coesione rappresenta le interazioni interne al modulo e deve essere massimizzata (si vuole che il modulo faccia il più possibile internamente e "tratti argomenti coerenti", senza aver necessità di interagire troppo con altri moduli), il coupling fa invece riferimento all'interazione tra moduli e deve essere minimizzata.
Il problema di queste metriche è che se cerchiamo di massimizzare la coesione operiamo negativamente sul coupling e viceversa. Ad esempio, se prendo un enorme prodotto e lo divido in due moduli allora ottimizzo il coupling in quanto avendo solo due moduli minimizzo le interazioni, ma peggioro la coesione in quanto un singolo modulo farà internamente tante cose diverse l'una dall'altra.
Al contrario dividendo in tantissimi moduli maggiore coesione ma anche maggiore coupling. Si vuole quindi trovare il numero di moduli che offra il miglior tradeoff, e in questo senso facciamo riferimento esclusivamente al costo.



Si torna al grafico visto per il modello incrementale nel cercare di capire il numero adatto di build: fissato il numero di moduli si devono considerare due elementi; il costo per modulo (costo per sviluppare la singola componente) e il costo di integrazione. Sappiamo bene che il costo è proporzionale alla dimensione del software al quadrato → maggiore il numero di moduli minore il costo per svilupparli (da lì la curva decrescente).
Tuttavia maggiore il numero di moduli maggiore il costo di integrazione, da lì la curva crescente.
Sommando questi due costi otteniamo l'andamento del costo totale (parabola) che ci permette di identificare la regione di costo minimo e quindi il numero di moduli che conviene scegliere a1 fine di minimizzare i costi.
Quindi dal punto di vista tecnico misuriamo la decomposizione modulare con coesione e coupling, dal punto di vista di costo con sto grafico.
Ma come si misurano coupling e cohesion?
Per eseguire una funzione sono necessarie varie azioni, che possono esser concentrate in un singolo modulo o sparse in tanti.

La coesione misura come un modulo riesce a svolgere internamente tutte le azioni necessarie ad eseguire una data funzione (ossia senza interagire con le azioni interne ad altri moduli). In altre parole, la coesione misura il grado di interazione interna al modulo tra le azioni di una funzione.
La coesione si misura utilizzando una scala di valori, per un totale di 7 (1 è peggiore, 7 è il migliore):

1. Coincidentale → nessuna relazione tra azioni nel modulo. Questo può succedere in casi specifici che vedremo.
2. Logical → elementi correlati, ma solo uno di essi viene utilizzato dal modulo chiamante
3. Temporal → relazione temporale tra gli elementi
4. Procedurale → gli elementi sono correlati in base a una sequenza predefinita di passi
5. Communicational → leggermente migliore della procedurale, uguale ad essa solo le azioni sono svolte sulla stessa struttura dati
6. Informational → ogni elemento ha una porzione di codice indipendente e un proprio puntio di ingresso e di uscita, inoltre ogni elemento agisce sulla stessa struttura dati.
7. Funzionale → tutti gli elementi sono correlati dal fatto di svolgere una singola funzione

In base al tipo di approccio, possiamo avere come obiettivo o Informational o Functional. Se paradigma OO allora ovviamente il meglio che possiamo fare è Informational (si definiscono le classi, ognuna con una funzione ben precisa che opera su una stessa struttura dati e dove ogni azione è identificata da punto d'ingresso e uscita), se paradigma di programmazione strutturata si può anche puntare alla Funzionale dove si identificano tanti moduli quante sono le funzioni del prodotto. Questi moduli possono avere luogo in contesti come standard di codifica, ossia dove ad esempio per un software a contratto viene chiesto ai programmatori di utilizzare le stesse regole nella codifica del software. Spesso vi sono dei vincoli nella dimensione del numero di istruzioni minime per modulo, quindi in tal caso anche per moduli semplici è necessario aggiungere istruzioni di questo tipo per raggiungere la dimensione minima.