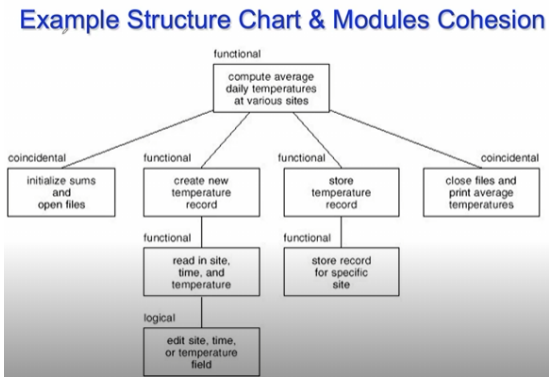
- ***Lezione 21***

La misurazione della coesione ovviamente avviene a livello architetturale, ossia per ciascun modulo individualmente.




A partire dalla descrizione di ogni modulo dobbiamo capire la relativa coesione: vediamo modi con coesione functional in quanto svolgono una funzione ben precisa (es. memorizzare record temperature, crea record temperature etc..), si hanno poi due moduli a coesione coincidentale in quanto svolgono azioni scorrelate (es. inizializza le somme e apri i file, chiudi file e stampa le temperature medie. Si poteva fare di meglio, ad es. creando un modulo dedicato alla gestione dei file, uno a inizializzare le somme e l'altro a stampare le temperature medie), poi un singolo modulo a coesione logica ossia che fa varie cose ma ogni volta che lo utilizzo se ne fa una sola in base a ciò che interessa al modulo chiamante. Ma perché il modulo sopra (read in site, time…) è funzionale mentre quello sotto logico? La differenza non sta nel fatto che uno legge e l'altro modifica, ma che sopra c'è un AND e sotto un OR: il primo è utilizzato per leggere i tre campi, il secondo per modificarne uno solo.
Dobbiamo ora occuparci di misurare il coupling.
Il coupling misura il grado di accoppiamento tra moduli, anch'esso si misura usando una scala, stavolta a 5 livelli (1 è il peggiore e il 5 è il migliore):

1. Content → un modulo fa riferimento diretto al contenuto di un altro modulo, modificandolo o semplicemente accedendovi (livello peggiore, forte dipendenza tra moduli)
2. Common → due moduli accedono in modalità read e write alla stessa struttura dati
3. Control → un modulo influisce l'esecuzione di un altro modulo
4. Stamp → due moduli interagiscono scambiandosi messaggi, in particolare passando come argomento una struttura dati della quale si usano solo alcuni parametri (quindi diciamo che tra i parametri ce ne sono alcuni che non servono all'altro modulo)
5. Data → due moduli interagiscono scambiandosi messaggi, in particolare passando come argomento una struttura dati della quale si usano tutti i parametri

I fattori che influiscono sul grado di accoppiamento sono la quantità di dati condivisi tra moduli, il numero di riferimenti che un modulo ha rispetto ad altri moduli, la complessità dell'interfaccia tra moduli, il livello di controllo che un modulo esercita su un altro.
Stamp e Data coupling sono infine il livello migliore in quanto i moduli si scambiano informazioni tramite un'interfaccia ben definita scambiandosi messaggi contenenti solo informazioni necessarie se Data altrimenti Stamp.

- *Information Hiding* → consiste nel progettare e definire i moduli in modo che gli altri moduli vedano solo quanto serve, nascondendo quindi i dettagli implementativi (procedura e dati) che ad essi non sono necessari.
  Nel caso dell'uso di Information Hiding invece si usano qualificatori di accesso che permettono di rendere privato l'accesso alla struttura dati, in questo caso ad es. se un modulo vuole accedere agli attributi di JobQueue non può farlo direttamente ma dovrà passare attraverso l'interfaccia di classe (i suoi metodi). Ciò come abbiamo detto è utile sia in fase di riuso della classe che in fase di manutenzione: controllo più fine sulle caratteristiche di ogni classe.

- *Riusabilità* → esso fa riferimento all'utilizzo di componenti sviluppati per un prodotto all'interno di un prodotto differente.
  In generale per componente riusabile si fa riferimento a moduli, progetti, parti di documenti, insiemi di test data, stime di costi o tempi etc..
  Tra i principali vantaggi la netta diminuzione di costi e tempi di produzione del software e incremento dell'affidabilità dovuto all'uso di componenti già convalidati.
  Nella fase di Progetto, la riusabilità si applica a:
  - singoli moduli software
  - application framework (la logica che tiene insieme i moduli e che viene utilizzata per usare i moduli)
  - design pattern (a livello progettuale spesso capita di affrontare problemi ricorrenti, quindi si sono definite soluzioni standard)
  - architettura software (all'intera architettura quindi ricombinando il riuso a livello di moduli, application framework e design pattern).
  Ora vedremo specificatamente un metodo di progettazione orientato agli oggetti.

➔ *Nono Blocco*

*Object Oriented Design – OOD*

Come nell'OOA, esistono vari metodi che utilizzano l'approccio object oriented.
Il nostro metodo di progettazione è costituito dalle seguenti sottofasi (già viste in generale per l'approccio alla progettazione):

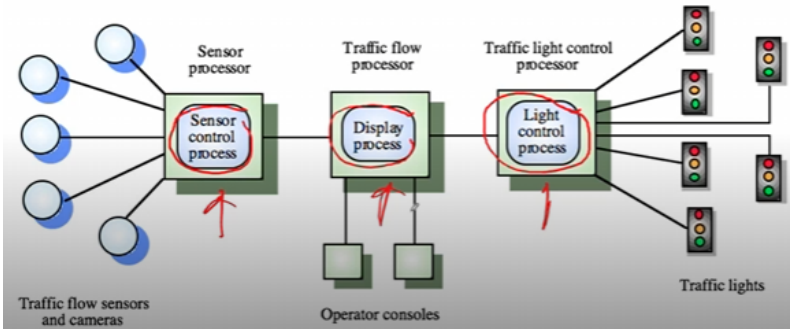
- Progettazione Preliminare (o architectural design o system design) → in questa sottofase si prendono le decisioni legate all'organizzazione d'insieme del software, definendo quindi l'architettura del sistema software.
- Progettazione dettagliata (object design) → si definiscono i dettagli di ciascun modulo in termini algoritmici, di strutture dati etc..

OOD esattamente come OOA lavora in modo iterativo e incrementale, inoltre OOD riutilizza ovviamente i risultati ottenuti in fase di OOA in particolare nella fase di progettazione dettagliata. La prima sottofase invece prevede di definire l'architettura software usando nuovi tipi di diagrammi UML.
Cosa significa definire un'architettura di sistema? Abbiamo capito di dover effettuare la decomposizione modulare, ma come progettiamo la soluzione pensando anche alla piattaforma di esecuzione del software?
L'architettura di sistema definisce la struttura delle componenti insieme alle relazioni che esistono tra le componenti.
Possiamo identificare un insieme di architetture standard, di seguito un elenco dalle architetture più antiche alle più moderne. Le seguenti architetture, dalla 3 alla 6, sono definite architetture distribuite o meglio architetture di sistema software distribuito (ossia per cui la sua esecuzione non è centralizzata ma partizionata e distribuita su vari nodi di esecuzione che sono interconnessi attraverso un'infrastruttura di rete che può essere locale, geografica etc…). Le prime due architetture invece sono esempi di architetture software centralizzate (si usa tipicamente un processo del sistema operativo che, mandato in esecuzione, esegue l'intero software).
Le architetture di cui stiamo parlando sono le seguenti:

1. *Mainframe-based architectures* → un mainframe rappresenta un computer molto potente che esegue sistemi operativi che sono multiutente (a differenza dei classici personal computer. Per realizzare questa multiutenze queste macchine hanno sulla parte posteriore delle porte fisiche tramite le quali sono connessi diversi terminali "stupidi" in quanto hanno soltanto dispositivi di I/O (tastiera e schermo) e non sono in grado autonomamente di eseguire software. Uno di questi terminali è il terminale console utilizzato dall'amministratore del sistema, colui che ha accesso completo alla configurazione del mainframe (definisce account degli utenti e le risorse etc…).
   Tutte le applicazioni vengono eseguite sul singolo Mainframe, quindi l'esecuzione del software è centralizzata.
   Tutt'ora si utilizzano tipicamente in domini critici come quello bancario, in quanto garantiscono maggiore affidabilità e sicurezza (es. possono funzionare per anni senza crashare)

2. *File Sharing* → qua siamo già passati all'introduzione di personal computer e le reti di calcolatori che permettono di fare file sharing. Identifichiamo questo come architettura centralizzata e non distribuita perché i pc sono collegati da una rete (immaginiamo ethernet in un'azienda), all'epoca piuttosto che comprare pc con alte prestazioni e quindi molto costosi se ne comprava solo uno con spazio di memoria significativo che aveva il compito di file server dove venivano salvati tutti i file, che poi venivano recuperati dagli altri computer in caso di necessità. Quindi l'esecuzione è ancora localizzata su un singolo nodo della rete di calcolatori.

3. *Client/Server* → da qui abbiamo architetture del sistema software distribuito. L'esecuzione del software, per architetture di sistema software distribuito, è partizionata in un certo insieme di nodi di esecuzione interconnessa da un'infrastruttura di rete.
   Il fatto che l'applicazione sia distribuita è trasparente per l'utente, non ci si accorge ad es. che si sta comunicando con un server nell'eseguire un'applicazione. Nel passaggio da architetture centralizzate a distribuite ha svolto un ruolo chiave l'utilizzo di tecnologia Middleware (si utilizza questo termine perché si fa riferimento a uno strato software che sta tra quello applicativo e quello del sistema operativo) e rappresenta la tecnologia che gestisce la complessità di un'applicazione distribuita
   (es. RPC remote procedure call, invece di chiamare procedure locali ne chiamo di remote. Devo quindi chiedere a qualcun altro di svolgere la procedura e prendere i risultati, i protocolli che permettano che ciò avvenga appunto sono gestiti dal Middleware, altri esempi MOM e ORB).

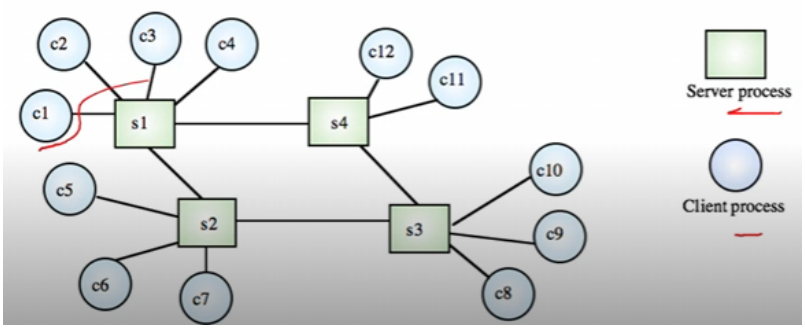



   Qui un esempio di sistema di controllo del traffico che è basato su un sistema software distribuito: diversi nodi in esecuizione che rappresentano i processi, a dx il processo che gestisce i semafori, a sx i sensori di traffico, al centro il processo che visualizza agli operatori le condizioni attuali del traffico (quindi il sistema software complessivo è costituito da più processi in esecuzione contemporaneamente collegati tra loro tramite infrastruttura di rete)
   I principali vantaggi che hanno portato all'utilizzo di sistemi distribuiti:
   Condivisione di dati e risorse tra i nodi in esecuzione, Openness (possibilità di gestire anche risorse eterogenee, si vuole ad esempio che un oggetto scritto in java in una macchina Windows possa invocare un metodo di un oggetto implementato in C++ su una macchina Linux), Concurrency (tutti gli oggetti in esecuzione operano in modo concorrente), Scalability (se non si ha più memoria è sufficiente aggiungere un nuovo

nodo da cui condividere), Load Balancing (si vuole distribuire il carico affinché non vi siano processi troppo appesantiti di lavoro), Fault Tolerance (backup dei nodi nel caso qualcosa vada storto), Trasparenza (diversi tipi di trasparenza, es. l'utente non distingue l'uso di risorse remote o locali oppure l'utente non sa che il nodo non funziona se è operativo quello di backup etc…).
Tuttavia non vi sono solo vantaggi ma anche fattori critici:

- Qualità del servizio (problemi di affidabilità es. se non ho la rete non posso usare l'applicazione, di performance per cui se un'applicazione ha ad es. requisiti di performance molto stretti in fase di progettazione posso creare modelli di simulazione basati sulla conoscenza delle risorse che utilizzo come processore, tempo di accesso al disco fisico etc..
posso fare una predizione accurata perché conosco le risorse che utilizzo, ma se la chiamata finisce in un server remoto fare queste predizioni è difficile perché intervengono molte altre variabili come l'operatore di rete)
- Interoperabilità (è necessario far gestire risorse eterogenee ed è difficile)
- Sicurezza (dati condivisi tra vari nodi)

Architetture Client/Server: si distingue il ruolo di processi client, di processi server e quelli che svolgono entrambi i ruoli. Il processo client interagisce con l'utente ed è quindi responsabile di prendere in carico la richiesta e fornire una risposta. Il client è solo un intermediario, non produce lui la risposta ma sottomette la richiesta ad un processo server per ottenerla.
Il processo server è invece il processo che attende le richieste da eventuali processi client. Egli nasconde la complessità dell'intero sistema all'utente e al processo client (da qui la trasparenza agli occhi dell'utente). Quando riceve la richiesta il processo server può rispondere direttamente o usare server backend per rispondere.




Vediamo il fatto che vi siano processi sia client che server dal momento che i server sono correlati tra loro: se a c2 serve un'informazione contenuta nel server s3 allora richiesta a s1 che dovrà contattare s4 che a sua volta contatterà s3 che risponderà. Quindi s1 e s4 sia client che server.
Come detto prima questi processi sono mandati in esecuzione su nodi interconnessi da una rete che permette la comunicazione. Normalmente un dispositivo possiede sia processi client che server.