Lez 21 (4/03)

Ci si occuperà, in questa seconda parte del corso, delle fasi successive a quella di specifica dei requisiti (al di là di quella di Pianificazione di cui si è già parlato alla fine del primo modulo).
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 progettazione**, *poi soluzione progettuale da tradurre in codice dai programmatori.*
Ci occuperemo solo della fase di progetto.

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, come quanto fatto nella fase di definizione e analisi dei requisiti, descriviamo anzitutto i **principi fondamentali di progettazione del software** per poi vederli concretamente applicati al caso object oriented.
Tra questi principi:
- ***Stepwise Refinement*** (procedere per raffinamenti successivi delle varie componenti senza procedere sequenzialmente, già visto anche in fase di analisi dei requisiti)
- ***Astrazione*** (usato anche in fase di analisi dei requisiti)
- ***Decomposizione Modulare***
- ***Modularità*** (capire il criterio per identificare i moduli)
- ***Information Hiding, Riusabilità*** (entrambi da applicare sia qui che in fase di implementazione)

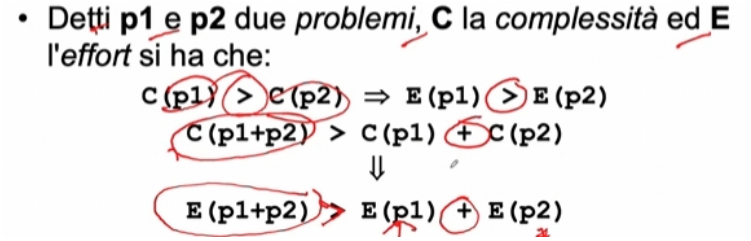
Stepwise refinement: in ambito di progettazione software il procedere per raffinamenti successivi è un processo di elaborazione che aggiunge di volta in volta solo i dettagli necessari per quella particolare attività che deve essere svolta.
In particolare si parte dalla specifica di una funzione (*o 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 alla ISW) *ogni essere umano può concentrarsi al più su 9 elementi differenti contemporaneamente*.

Il concetto di Raffinamento è complementare a quello di **Astrazione**.
Infatti l'Astrazione 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) è una struttura che unisce l'astrazione dei dati e delle operazioni, definendo un insieme di dati e le operazioni che possono essere eseguite su di essi.
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: 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!

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**

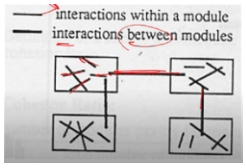
Esempi del concetto di modulo possono essere le funzioni o le classi (le funzioni in particolare rispettano il quarto requisito perché il segmento che può essere invocato è uno solo e rappresenta la loro stessa firma).
Il risultato ottenuto dopo la decomposizione modulare è detto architettura dei moduli.
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.



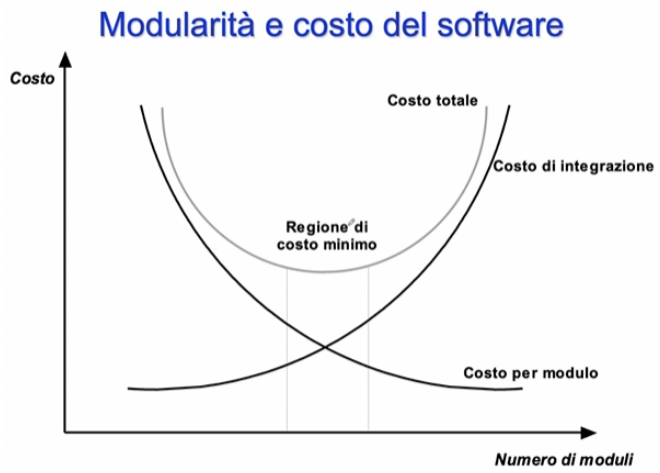
Il + rappresenta null'altro quindi che la fase di integrazione successiva a quella di codifica, in cui i vari moduli vengono rimessi insieme.

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 es. 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.

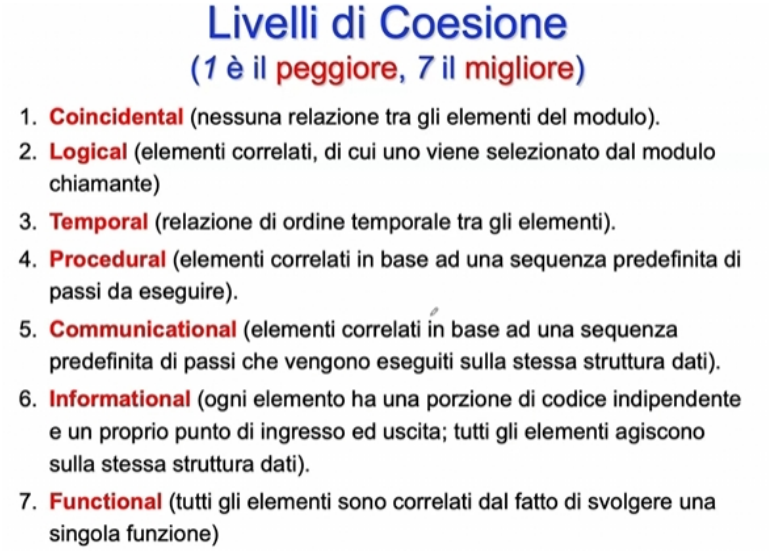


Si torna al grafico visto per il modello incrementale nel cercare di capire il numero adatto di build: fissato il numero di moduli si devono considerare due elementi; il **costo per modulo** (costo per sviluppare la singola componente) e il **costo di integrazione**. Sappiamo bene che il costo è proporzionale 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 quanto le operazioni all'interno di un modulo sono logicamente connesse tra loro

La coesione si misura utilizzando una scala di valori, per un totale di 7:

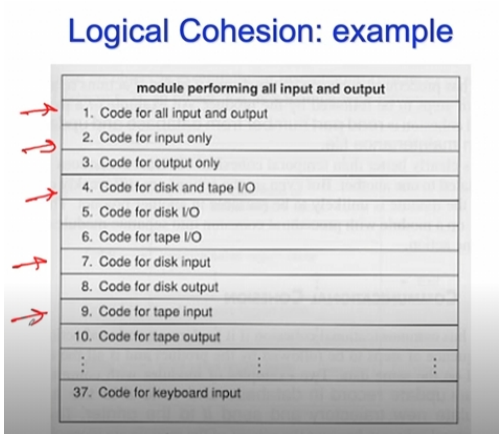


Coincidentale == *nessuna relazione tra azioni nel modulo*. Questo può succedere in casi specifici che vedremo.
Logical == *elementi correlati, ma solo uno di essi viene utilizzato dal modulo chiamante*
Temporal == *relazione temporale tra gli elementi*
Procedurale == *gli elementi sono correlati in base a una sequenza predefinita di passi*
Communicational == leggermente migliore della procedurale, uguale ad essa solo *le azioni sono svolte sulla stessa struttura dati*
Da qui i due livelli migliori:
6-**Informational** == *ogni elemento ha una porzione di codice indipendente e un proprio punto 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.*

es. di **coesione coincidentale**: stampa la prossima riga, inverti i char della seconda stringa parametro, aggiungi 7 al quinto parametro etc…
sono tutte azioni scorrelate, un modulo del genere non è riusabile o mantenibile.

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.



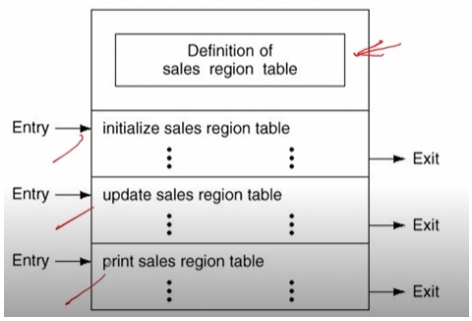
Qui invece un esempio di **coesione logica***: il modulo è diviso in tanti pezzi di codice e so che però se mi servisse ne chiamerò soltanto uno. Problemi di manutenibilità perché le porzioni di codice potrebbero essere interallacciate con altro fuori, per questo ancora solo livello 2 di coesione.*

Poi la **coesione temporale** ad esempio open old_master_file, open new_master_file, open transactionfile etc… azioni correlate a livello temporale (avvengono tutte all'inizio).
Ancora solo livello 3 di coesione perché i file aperti in questo modulo potrebbero essere aggiornati da altri moduli -> sempre problemi manutenibilità.

Coesione procedurale: read part_number from database, use part_number to update… *azioni avvengono in sequenza*.

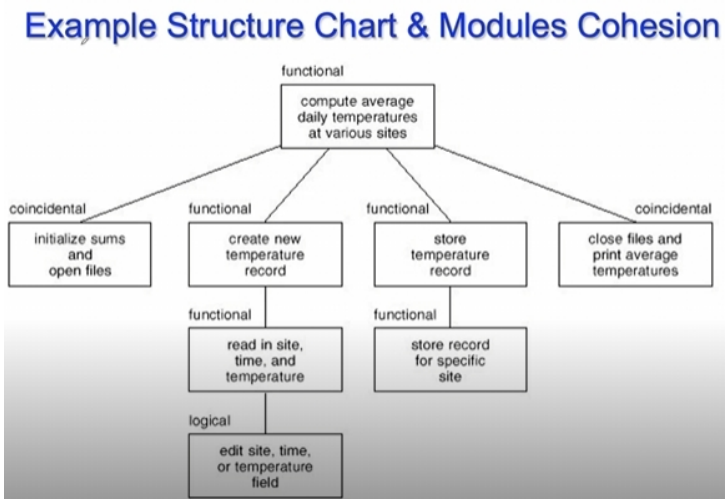
Communicational aggiunge il fatto che si lavora su una singola variabile/struttura dati.
In alcune scale di valori vista la loro somiglianza temporal, procedural e communicational sono combinate in un singolo livello.

Infine coesione **Informational**: es. *classe dove si ha una struttura dati e una serie di funzioni che lavorano su di essa, ognuna utilizzabile in modo separato*



Lez 22 (6/03)

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 dal peggiore al migliore:

- Livelli di *coupling* (*1* è il peggiore, *5* il migliore):
  1. **Content** (un modulo fa diretto riferimento al contenuto di un altro modulo).
  2. **Common** (due moduli che accedono alla stessa struttura dati)
  3. **Control** (un modulo controlla esplicitamente l'esecuzione di un altro modulo).
  4. **Stamp** (due moduli che si passano come argomento una struttura dati, della quale si usano solo alcuni elementi).
  5. **Data** (due moduli che si passano argomenti omogenei, ovvero argomenti semplici o strutture dati delle quali si usano tutti gli elementi).

Content == *un modulo fa riferimento diretto al contenuto di un altro modulo, modificandolo o semplicemente accedendovi* (livello peggiore, forte dipendenza tra moduli)
Common == *due moduli accedono in modalità read e write alla stessa struttura dati*
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.

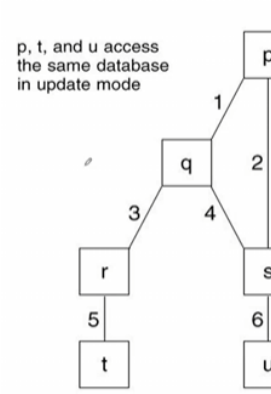
Degli esempi di **content coupling** è, dati due moduli p e q, *se p modifica un'istruzione di q*, se *p fa riferimento a dati locali di q* in termini di qualche "displacement" numerico (in linguaggi a basso livello come assembly, dove si utilizzano i displacement tra moduli per passarsi informazioni), se p utilizza un'etichetta locale del modulo q (caso del go to, se p fa goto su un'etichetta interna di q, sta saltando direttamente nel suo flusso di controllo interno).
Questo tipo di coupling è molto difficile da trovare in codici attuali, si tratta del livello peggiore di coupling perché ogni cambiamento a q richiede una modifica al modulo p.

Riguardo il **common coupling** un esempio sono *due moduli cca e ccb che accedono alla stessa variabile globale in modalità lettura e scrittura*. Si tratta di un grado comunque non buono in quanto capire come si comporta la parte di codice dedicata alla variabile è più difficile se vi sono più moduli che vi accedono e possono modificarla (*problemi di sicurezza e integrità dei dati della variabile*)

Un esempio per il **control coupling** è *se un modulo p chiama il modulo q chiedendogli di fare qualcosa, dopodiché q invia un flag di ritorno a p che permette a p di svolgere una certa azione* (in base a quanto detto da q p si comporta di conseguenza). *In questo senso q esercita controllo su p in quanto in base alla sua risposta p si comporta di conseguenza*.

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*.

Anche per il coupling facciamo un esempio di architettura software per capire come misurare i livelli di coupling tra moduli.
NB. **Structure Chart == termine che si utilizzava storicamente per definire ciò che oggi si chiama architettura software**



parameters

number	In	Out
1	aircraft type	status flag
2	—	list of aircraft parts
3	function code	—
4	—	list of aircraft parts
5	part number	part manufacturer
6	part number	part name

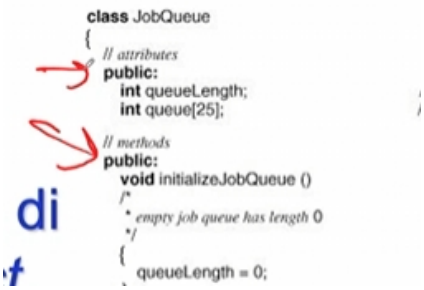
coupling

	q	r	s	t	u
p	Data	—	Data or Stamp	Common	Common
q		Control	Data or Stamp	—	—
r			—	Data	—
s					Data

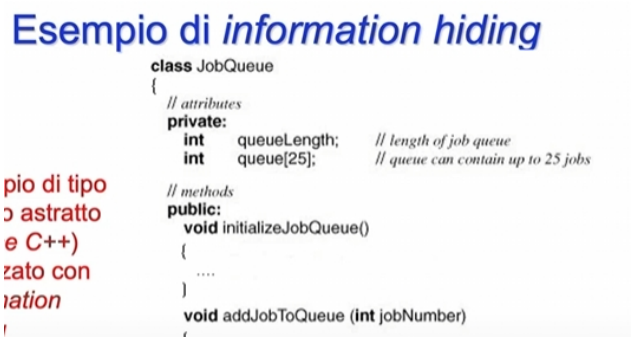
L’interazione tra moduli è evidenziata da degli archi numerati.
I moduli si scambiano dei messaggi, e questi messaggi veicolano dei parametri che possono essere di input o di output.
Es. p e q interagiscono scambiandosi il messaggio 1 che ha come parametro di input tipo di aircraft e output status flag.
Nella tabella sotto invece vediamo per ogni interazione che esiste tra moduli il relativo livello di coupling.
Es. tra p e q viene definito il livello data, si assume cioè che sia aircraft type che status flag siano parametri utilizzati dai rispettivi moduli.
Un altro esempio è l’interazione 4, dove i due moduli si scambiano parametri di output lista di parti di aircraft. Il relativo coupling è definito Data or Stamp poiché evidentemente non si hanno informazioni sull’effettivo utilizzo di questa lista di parti (se ne uso solo alcune allora stamp else data), vale lo stesso per l’interazione 2.
Per l’interazione 3 invece come parametro di input codice di funzione, coupling di controllo in quanto chi manda il messaggio chiede al modulo di eseguire la funzione.
Dopodiché anche 5 e 6 Data Coupling, mentre per l’accoppiamento tra p e t e p e u livello di coupling Common. Nella figura non vi è l’arco che identifica l’interazione in quanto p e t e p e u non si scambiano effettivamente dei messaggi ma accedono allo stesso database in update mode (NB se fosse stato scritto in read mode allora non sarebbe stato accoppiamento di tipo common).

Prima di passare al nostro specifico modello di progettazione approfondiamo gli ultimi principi che abbiamo citato nella scorsa lezione.

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



Qua sopra un esempio di dato astratto (classe in C++), in questo caso sia l'interfaccia della classe (la parte relativa alle operazioni) che la struttura dati sono qualificate con accesso pubblico -> chiunque può leggere e modificare, esempio di data incapsulation senza information hiding.



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.
NB **Data Encapsulation != Information Hiding**, si può avere Data Encapsulation senza Information Hiding come visto sopra. (Data Encapsulation == Meccanismo tecnico per raggruppare dati e operazioni che li manipolano in una singola entità (es. una classe)).

L'ultimo principio che abbiamo citato è quello di **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** (insieme strutturato di componenti software riutilizzabili che forniscono l'infrastruttura logica e tecnica di base per costruire applicazioni)
- **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).

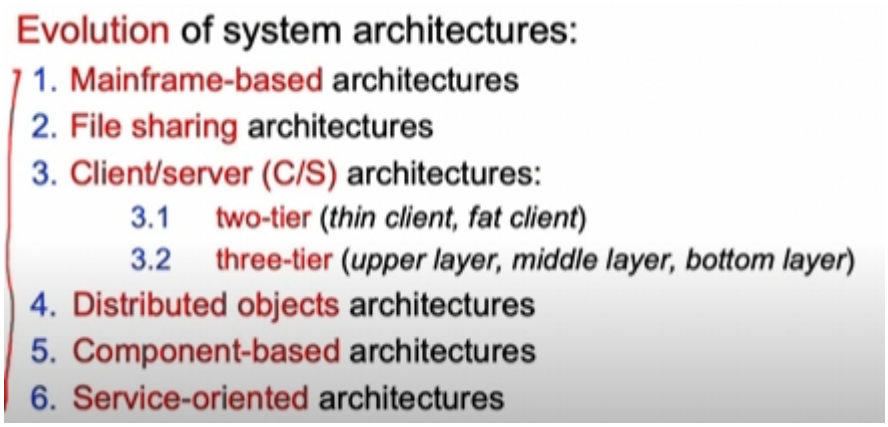
Ora vedremo specificatamente un metodo di progettazione orientato agli oggetti.

Object Oriented Design OOD

Anche qui vale lo stesso discorso visto per **l'OOA**: *esistono vari metodi che utilizzano l'approccio object oriented, ne vedremo uno a livello didattico che prende spunto dai principali metodi utilizzati.*
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).



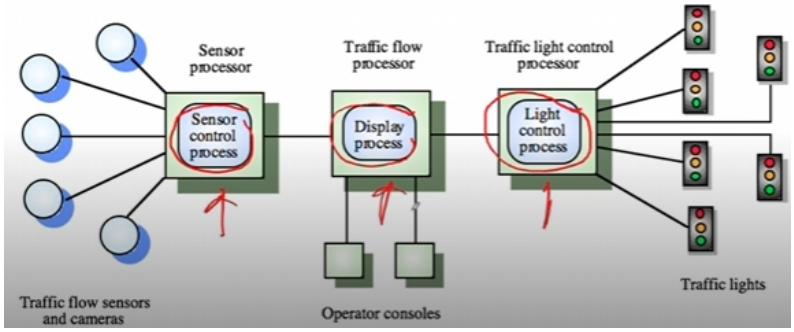
1- **Mainframe-based architectures**: *un mainframe rappresenta un computer molto potente che esegue sistemi operativi che sono multiutente* (a differenza dei classici personal computer. Per realizzare questa multiutenze queste macchine hanno sulla

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).

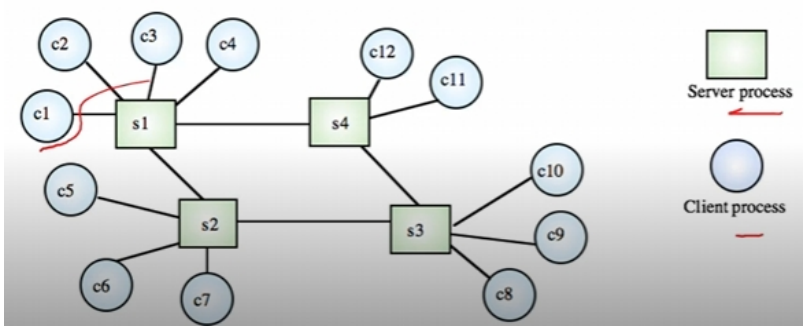


Qui un esempio di sistema di controllo del traffico che è basato su un sistema software distribuito: diversi nodi in esecuizione che rappresentano i processi, a dx il processo che gestisce i semafori, a sx i sensori di traffico, al centro il processo che visualizza agli operatori le condizioni attuali 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)

3- **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.

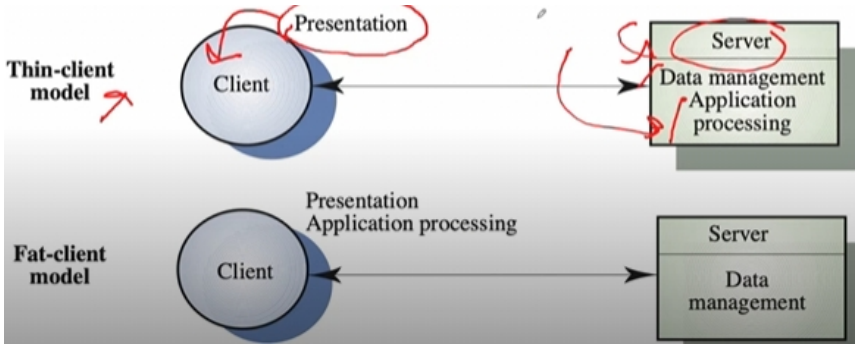
Lez 23 (11/03) (se non capisci chiedi a chat che ha spiegato di merda)

Abbiamo già visto nella prima parte del corso come viene stratificato un software (approccio **BCE Boundary == interfaccia, Control == logica dell'applicazione, Entity == dati**).
Questo vale in generale, e *dal punto di vista applicativo nella rete dei calcolatori si parla* **Application Layers**: **Presentation Layer** == *aspetti di presentazione, interazione con l'utente* (corrisponde a B), **Application Processing Layer** == *fornisce la logica di esecuzione dell'applicazione* (C), **Data Management Layer** == *gestione dati* (E).

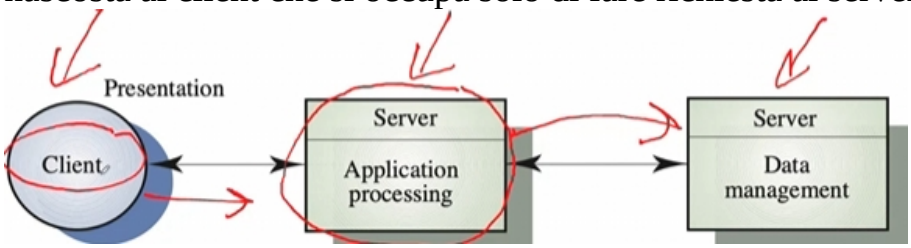
Nasce a questo punto il seguente problema, **i vari Application Layers su quale tipo di processi (client o server) devono essere allocati**?
Questo ha portato all'introduzione di diversi sottotipi di architetture C/S, tra cui quelle che seguono.

Two-Tier C/S architectures: *si chiama in questo modo in quanto l'architettura si esaurisce solo in due livelli (tier); una parte client e una server dove possiamo allocare i layer applicativi*.
Di questo tipo di architettura esistono **due principali modelli**:
- **Thin-Client Model** *dove solo il layer di Presentazione viene lasciato al client, mentre l'application processing e il data management è lasciato al server "appesantendolo"*
- **Fat-Client Model** *invece prevede il client con Presentazione e Application processing, mentre il Data Management al server*. Ciò significa che quando un utente fa una richiesta al client questo riesce a gestire anche quanto deve essere fatto in seguito a questa richiesta, e nel caso sia necessario accedere a dei dati si rivolgerà al server.

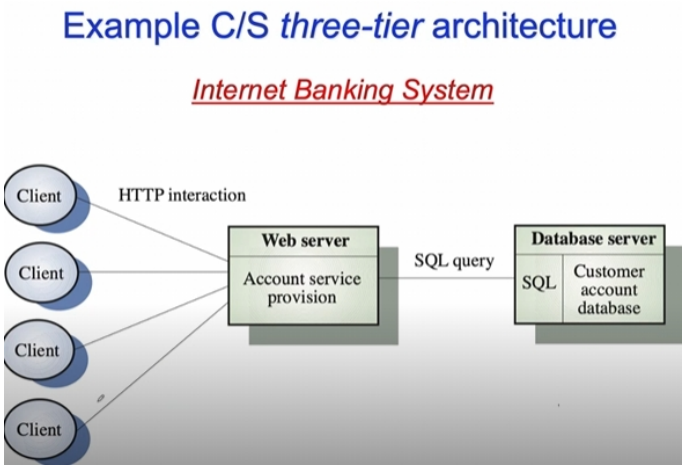
Si tratta chiaramente di modelli agli estremi, esistono infatti modelli intermedi dove la parte di Application Processing viene partizionato nel Client e nel Server.



Nel tempo l'architettura Two-Tier è evoluta nella Three-Tier, dove sostanzialmente si individua uno specifico livello per ogni application layer. Quindi il *Presentation Layer è allocato sul tier client*, *l'application processing su un server intermedio* e *il data management su un server di backend*. **Si osserva chiaramente come il server intermedio agisca sia come client che come server**, questa complessità inoltre viene nascosta al client che si occupa solo di fare richiesta al server intermedio.



L'uso di questo approccio offre migliori performance rispetto al two-tier thin-client ed è più semplice da gestire rispetto al two-tier fat-client. Inoltre è un'architettura più scalabile della two-tier in quanto se si necessità di maggiori informazioni allora più server possono essere aggiunti.



Un esempio di three tier architecture è questo per la gestione di un sito di una banca. L'application Layer è lasciato al client in quanto eseguendo il browser questo permette all'utente di fare le richieste una volta collegato al sito della banca.
Le richieste sono inoltrate al server Web che è server intermediario e che quindi provvede a gestire il servizio per gli account bancari (ossia gestisce la logica dietro

ogni possibile richiesta).
Il server Web a cui si collega il browser ha a sua volta un server di backend (database) al quale accede, nel quale sono memorizzati i dati relativi a tutti gli utenti di cui potrebbe necessitare per rispondere alle richieste.

Le architetture three-tier possono anche espandersi ulteriormente diventando n-tier, poiché facilmente si aggiungono ulteriori strati di backend (es. utilizzo di authenticatuon servers).

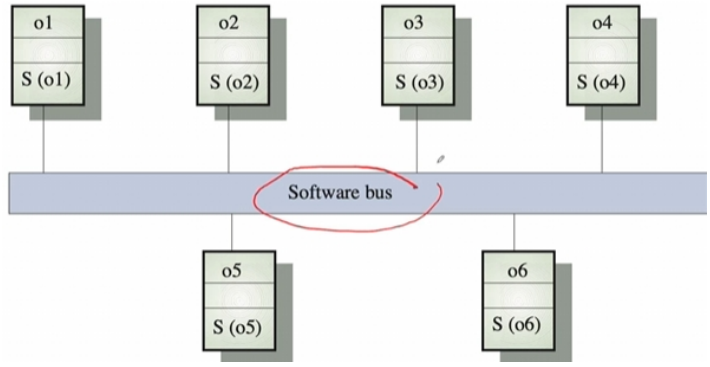
Le architetture client-server sono tutt'ora utilizzate ampliamente, e nel tempo sono state utilizzate come base per implementare architetture software che facciano uso di paradigmi differenti come 4- **l'Architettura a Oggetti Distribuiti** (**Distributed Object Architecture**).
Sappiamo che un software in esecuzione che fa uso di un paradigma object oriented non è altro che un insieme di oggetti creati a partire da classi e che si scambiano messaggi (un oggetto richiede a un altro oggetto di eseguire un certo metodo).
Nel paradigma object oriented se pensiamo quindi al singolo oggetto non vi è distinzione netta tra client e server, svolge entrambi i ruoli (client se richiede a un altro oggetto di eseguire il metodo, server se il contrario).
Nell'architettura a Oggetti Distribuiti l'idea è di ampliare l'uso del paradigma in un ambiente distribuito (infatti tipicamente un programma oo è confinato all'esecuzione in un singolo sistema operativo) ***garantendo comunque un certo livello di interoperabilità*** (non voglio essere vincolato dall'uso di uno specifico linguaggio di programmazione o sistema operativo).

Qui entra quindi molto in gioco il middleware, *si vuole che agli occhi del programmatore invocare ad esempio un metodo non comporti differenze rispetto all'approccio centralizzato*.
Per raggiungere questo obiettivo è stato realizzato un middleware che è chiamato **ORB Object Request Broker**, *egli funge da agente (broker) per cui porta al destinatario la richiesta mittente e riporta al mittente la risposta del destinatario* (si parla in questo senso di software bus).
Anche per ORB ***netta separazione tra interfaccia e implementazione***, infatti i servizi offerti da ORB sono specificati (viene definita quindi solo l'interfaccia) in un **abstract bus**, per poi procedere all'implementazione concreta con la **bus implementation**.
Tale implementazione deve essere presente nei vari linguaggi e sistemi operativi per garantire interoperabilità.

L'esempio più famoso di **abstract bus** si chiama **CORBA**, questo rappresenta *uno standard pubblicato da* **OMG** *per la specifica dell'interfaccia dei servizi offerti da un ORB* (quindi no implementazione, solo standard di specifica). *CORBA è stata implementata da due aziende diverse in due modi diversi*: **Visibroker** e **Orbix**.
Tuttavia ciò comportò problemi in quanto l'interoperabilità non era più garantita: se dispositivi eterogenei interagivano e l'implementazione di ORB era una in orbix e l'altra in visibroker non funzionava.

A questo punto l'OMG ha dovuto rilasciare una versione successiva di CORBA in cui venne introdotto un protocollo chiamato **IIOP che permette di garantire interoperabilità anche tra ORB eterogenei**.

Negli anni '90 le architetture a oggetti distribuiti sono state ad uso molto comune, ma vi era un problema: tutti questi protocolli comunicavano attraverso porte proprietarie. Quindi per mettere in piedi una piattaforma distribuita basata su CORBA si dovevano configurare opportunamente i firewall intermedi per permettere il traffico su queste porte. **Questo problema è stato superato con le architetture Service Oriented in cui ci si è limitati all'uso di protocolli internet standard**.



ORB come "software bus" che permette di ampliare il paradigma a oggetti in un ambiente distribuito.

A valle delle architetture a oggetti distribuiti sono state introdotte architetture che cambiano anche il modello di "business" dietro allo sviluppo software (cambierà ulteriormente con le architetture service oriented), stiamo parlando delle architetture
5- **Component Based**.
Si è pensato nel mondo software di adottare un approccio simile a quello adottato nel mondo hardware per assemblare un dispositivo elettronico, dove si partiva da componenti preconfezionate (i circuiti integrati) da collegare in modi specifici per realizzare determinate funzionalità.
L'idea quindi è stata di sviluppare il software partendo da componenti preconfezionate che già implementano certe funzionalità limitandosi quindi ad assemblare queste componenti tra loro.

In questo senso una componente software è un'astrazione che può essere implementata in modi diversi (oo, approccio strutturale etc…).
Anche per quanto riguarda le componenti software quindi si fa uso del principio di netta separazione tra interfaccia e implementazione -> una componente è un'entità che realizza una certa interfaccia (mettendo quindi a disposizione dei servizi) e gli utenti potranno far uso di questi servizi senza conoscere i dettagli implementativi.

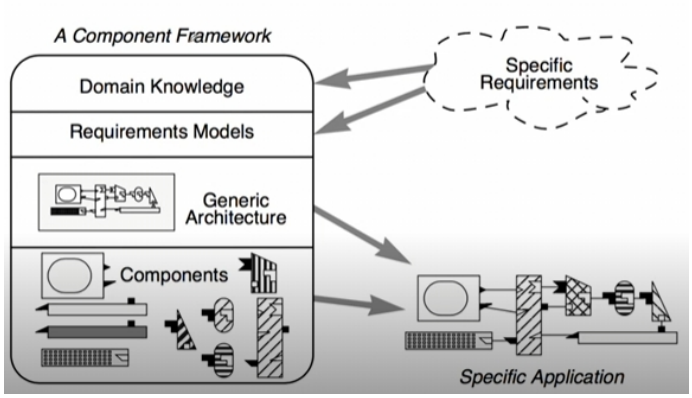
Si parla in questo caso quindi di un **riuso black box**, ***in quanto la componente viene riusata semplicemente perché realizza quell'interfaccia*** *-> posso rimpiazzare una componente con un'altra a patto che realizzi la stessa interfaccia*!
L'idea è quindi come detto di avere componenti software che in modo simile ai plug

(piedini) di un circuito integrato possano essere organizzati, configurati e modificati per soddisfare le esigenze e sviluppare il software

Aspetti importanti delle componenti software in questa architettura sono quindi:
- *capacità di incapsulare strutture software complesse* (codice) *in queste componenti astratte* (**variabilità**, cambiano comportamento in base a come sono utilizzate, le posso usare in domini diversi in base alla necessità)
- *possibilità di "assemblare" queste componenti collegandole attraverso l'interfaccia e lo scambio di messaggi* (**adattabilità**)

Vediamo di seguito le principali differenze tra oggetto e componente:
- *un oggetto incapsula i servizi secondo il paradigma object oriented,* mentre *le componenti sono astrazioni software più ampie che possono a loro volta essere usate per costruire sistemi object oriented*
- *un oggetto ha* ***granularità*** *ben specifica mentre una componente ha granularità molto variabile* (dall'essere un'intera applicazione a incapsulare un singolo oggetto)
- *mentre un oggetto presenta una propria identità, stato e comportamento, la componente è semplicemente un entità software statica a cui chiedere qualcosa e dal quale ottenere qualcosa*

Si è cercato di capire quindi come utilizzare le componenti software per realizzare applicazioni, e da questo punto di vista gioca un ruolo fondamentale il **Component Framework**.

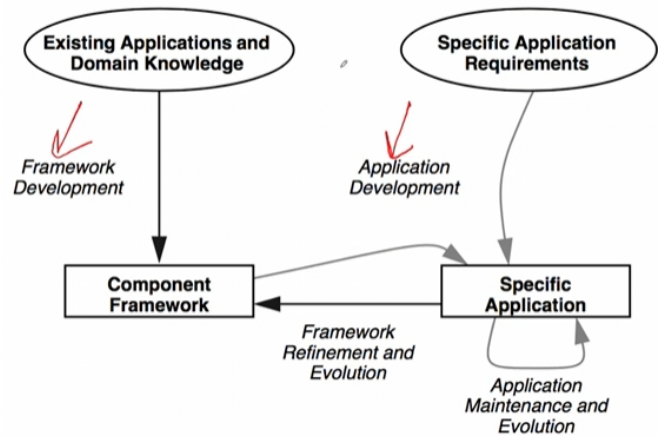


Il component framework è l'elemento di cui si fa uso per realizzare applicazioni facendo uso di architetture component based.
Come si osserva in figura, **il component framework non mette semplicemente a disposizione una libreria di componenti, ma fornisce molto di più in ottica di sviluppo software.**
(dal basso verso l'alto) ***A partire dalla libreria di componenti*** (*che fanno riferimento a uno specifico dominio applicativo es. dominio sanitario allora componenti utili come gestire la cartella clinica*) ***il component framework fornisce un insieme di architetture software generiche che già danno modo di assemblare i componenti per fornire funzionalità di alto livello***, che soddisfano un certo numero di Requisiti generici che fanno riferimento alla conoscenza del dominio.
Quindi il component framework cattura requisiti generici per un particolare

dominio, fornisce poi la relativa architettura software che in base a una lista di componenti permette di soddisfare quei requisiti.

Quindi si fa tipicamente quanto segue: *ho dei requisiti specifici per il mio dominio, vedo se vi sono somiglianze con i requisiti generici di un certo framework e se ve ne sono abbastanza allora implemento le componenti mancanti e il framework potrà permettermi di costruire l'applicazione per soddisfare i miei requisiti specifici*. Avendo aggiunto componenti al framework, se qualcuno dovrà realizzare in futuro quanto ho realizzato potrà partire dal framework aggiornato.

Quindi il framework è costantemente aggiornato, al punto che si distingue la figura del programmatore di applicazione specifica da quello del framework.



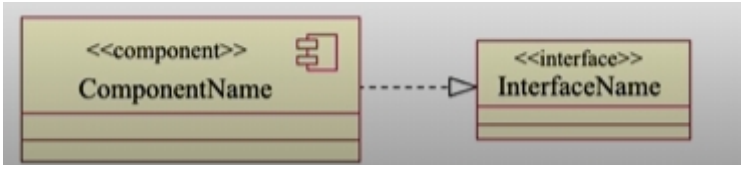
Tuttavia l'architettura model based non ha avuto successo in quanto a livello commerciale/industriale non ha supportato la realizzazione di framework sufficienti.

Stiamo introducendo tutte queste architetture per capire come definire l'architettura software in fase di progettazione, ossia dopo la definizione e specifica dei requisiti. Nel nostro caso stiamo usando un approccio semiformale basato su UML come linguaggio di modellazione.

Come supporta UML l'idea di componenti software?

In UML 1 il concetto di componente era legato a un'entità fisica: componente come qualcosa di concreto e già pronto per essere eseguito/distribuito (es. file .exe o una libreria). Tuttavia non si spiegava come si arrivava a questa componente fisica, ossia quali classi e elementi del progetto la componevano.

Con UML 2 si supera il problema introducendo una modifica sostanziale al concetto di componente: esso rappresenta un elemento che esiste già a livello di progetto.



Il componente viene quindi rappresentato come una sorta di classe con lo stereotipo **<<component>> che in modalità blackbox realizza una certa interfaccia**, questa

realizzazione blackbox è visualizzata con la freccia a triangolo vuoto (simbolo ereditarietà) ma tratteggiata.

In **UML 2**, per colmare il divario concettuale tra la **progettazione** (modelli astratti) e l'**esecuzione** (sistemi reali), è stato introdotto il concetto di **classe strutturata**.
A differenza della classe tradizionale, la classe strutturata permette di rappresentare ***la struttura interna*** *di una classe*, mostrando come essa sia composta da parti, connessioni e punti di interazione (porte). Questo consente di ***modellare più accuratamente il comportamento reale delle classi*** *una volta implementate ed eseguite*. Inoltre, in UML 2, viene introdotto anche il concetto di **sottosistema**: *un componente software può essere progettato come un sottosistema, ovvero un insieme organizzato di classi, oggetti e interazioni che collaborano per offrire una funzionalità complessa, ma che viene gestito come un'unica unità*.
I **dettagli interni del sottosistema** (cioè come funziona al suo interno) possono essere modellati proprio grazie all'uso delle **classi strutturate**, che ne descrivono l'architettura interna in modo preciso.

Vantaggi componenti: **riusabilità**, **affidabilità** (la gente l'ha già usato prima quindi so che funziona), **manutenibilità**.

6- **Architetture Service Oriented**: ***anche in questo caso l'idea è di sviluppare applicazioni facendo uso di componenti preconfezionate***.
Quella che prima si chiamava componente ora si chiama servizio, *ma la differenza sostanziale sta proprio sul modello di business*.
Mentre nel caso dell'architettura component based le componenti di mio interesse sono portate nel mio ambiente di sviluppo per assemblare l'applicazione di mio interesse, ***nelle architetture service oriented ciò non succede dal momento che si riutilizzano componenti senza importarle, riusandoli là dove sono messi a disposizione.***

L'idea è che quindi ci sia qualcuno che crei la componente e la metta a disposizione come servizio, e ora non compro più la componente per farci ciò che voglio ma semplicemente utilizzo il servizio quando mi serve.
Si passa a una modalità di utilizzo del software (nuovo modello di business) "pay per use".

L'altro elemento fondamentale è che dietro a queste architetture service oriented c'è **l'idea di utilizzare la piattaforma internet per veicolare il servizio**, *superando così anche il problema di utilizzo di protocolli proprietari visto prima in quanto si dovranno solo utilizzare protocolli standard internet*.
Si identificano quindi le figure di **Service Provider** e **Service Consumer**.
Il primo, una volta fornito il servizio, deve fornirne anche la descrizione di modo che il consumatore abbia solo le informazioni necessarie per poter utilizzare il servizio (*SOLO perché si nascondono i dettagli implementativi*).

Poiché anche in questo caso siamo in ambiente distribuito sono importanti i Service Broker, del tutto simili all'ORB visto nell'architettura ad oggetti distribuiti.

In quel caso il broker prendeva l'invocazione del metodo per portarla all'oggetto remoto, *in questo caso porta la richiesta dal service consumer al provider e viceversa*

La nascita di queste architetture deriva dalla maggiore facilità di sviluppo in un ambiente standard come Internet e stessi vantaggi delle componenti.

Per creare un'applicazione basata su servizi, devo utilizzare ulteriori servizi detti **servizi di coordinazione** per gestire più servizi necessari alla mia applicazione. **Quindi anche dal punto di vista di "come assemblare" questi servizi devo far riferimento ad altri servizi**.

Di seguito i principi di questa architettura

Loose Coupling == *si vogliono servizi il più possibile indipendenti dagli altri* (simile per quanto visto parlando di coesione e coupling)
Service Contract == *contratto tra consumatore e provider per cui il provider promette al consumatore di rispettare la descrizione mentre il consumatore di usare il servizio secondo le modalità previste*
Autonomia == ancora indipendenza da altri servizi
Astrazione == *il consumatore deve sapere solo quanto necessario*
Riusabilità, Composability e Statelessness == già principi utilizzati nell'architettura basata su componenti (composability come possibilità di combinare servizi diversi, statelessness per dire che il servizio non mantiene informazioni tra una richiesta e un'altra)
Discoverability == **principio esemplare per questa architettura**, *rappresenta la capacità di un servizio di essere trovato e identificato facilmente da altri servizi o da chi li utilizza* (i servizi devono essere descritti chiaramente e deve esistere un meccanismo che permetta ai consumatori di scoprire facilmente le capacità del servizio)

Lez 24 (13/03)

*Illustreremo adesso dei **pattern/protocolli** di interazione tra le varie entità che giocano un ruolo nelle architetture service oriented* usati a livello di progettazione *per capire come tali protocolli possano esser concretamente utilizzati facendo uso di tecnologie implementative.*

Partiamo dai pattern di interazione di base. Abbiamo detto che nell'architettura service oriented ritroviamo i due ruoli introdotti con il client/server: il **consumatore di servizio (client)** e **il service provider (server)**.
Tra le due entità si interpone il **service broker** che regola e permette la comunicazione tra i due.
Si parla quindi di **Software Architectural Broker Patterns**.
Per poter utilizzare il broker il service provider, una volta realizzato un servizio, deve registrarsi presso il broker al fine di rendere il servizio disponibile (*per soddisfare il principio di Discoverability!*) **ed il service consumer utilizzerà il broker (registro di servizi) per capire se esiste un servizio che fa al caso suo**.

Dopo questa fase iniziale esistono ***diverse modalità di interazione tra i due***: una **modalità diretta** *per cui il consumer comunica direttamente con il provider una volta acquisite le informazioni necessarie dal broker* oppure *una* **interazione mediata** *per cui ogni richiesta da parte del consumer passa attraverso il broker*.

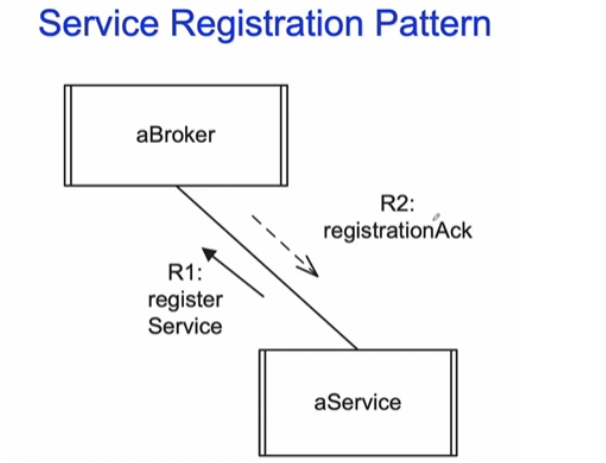
In tutto ciò gioca un ruolo fondamentale il concetto di **trasparenza**, due tipologie:
- **Platform Transparency**: *l'utilizzo del servizio deve essere possibile per qualsiasi piattaforma (sistema operativo etc..) e non si necessita di conoscere i dettagli implementativi che permettono l'esecuzione del servizio sul suo ambiente*
- **Location Transparency**: *se il provider decide si spostare il servizio su una **porta/interfaccia di rete differente**, i consumatori non necessitano di essere informati **ma soltanto il broker***

Si parla in questo senso di **brokered communication** **in quanto al fine di garantire tale trasparenza deve essere il broker a gestire il tutto** (es. se cambia interfaccia di rete sarà informato il broker che provvederà affinché il consumer possa continuare a fare richieste normalmente). Ma come anticipato, per utilizzare la brokered communication, è necessario introdurre il pattern di base che è il **Service Registration Pattern** (*il provider deve registrare le informazioni sul servizio presso il broker*).

Il **Service Registration Pattern** rappresenta semplicemente una richiesta che il service provider fa al broker per registrare il proprio servizio tramite un messaggio

dove **il provider comunica nome del servizio, una descrizione e l'interfaccia di rete presso cui è disponibile**.

Ciascun pattern sarà illustrato dal punto di vista visuale tramite UML 2.
Si ricordi che quando abbiamo introdotto i diagrammi di interazione UML per descrivere le interazioni tra oggetti abbiamo detto che esistono due tipi di diagrammi che dal punto di vista espressivo sono del tutto equivalenti: **sequence diagram** e **collaboration diagram**. *Entrambi sappiamo avere potere espressivo equivalente, ma il collaboration diagram è tipicamente usato in fase di progettazione in quanto è una rappresentazione di interazione tra oggetti più compatta* (e in fase di progettazione molti più oggetti rispetto alla fase di requisiti).



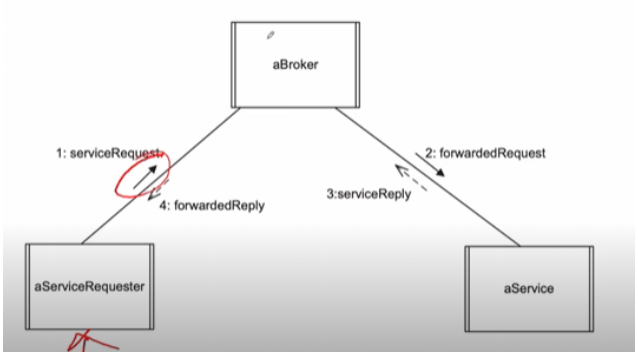
Qui un esempio di collaboration diagram, dove vengono visualizzati solo gli oggetti coinvolti nell'interazione e i messaggi che si scambiano. **Manca una componente fondamentale che invece è presente nel sequence diagram: l'ordine temporale** (nei sequence diagram messaggi scambiati dall'alto verso il basso).
Sappiamo però che sono rappresentazioni equivalenti, quindi è immediato passare da sequence a collaboration ma più difficile fare il contrario se manca ordine temporale.
Quindi alla specifica di ogni messaggio scambiato è associato un **sequence number** (R1 e R2 in figura) ***per ricostruire anche l'ordine temporale e passare al sequence***.

Ma perché usiamo UML 2? Gli oggetti sono rappresentati normalmente da rettangoli dove tramite sintassi si può anche risalire alla classe che ha creato l'oggetto (nomeoggetto riferimentooggetto:nomeclasse). Qui manca la specifica della classe in quanto facciamo riferimento a istanze di un servizio e **i rettangoli hanno barre laterali**: *ciò sta a significare che i due elementi* (broker e service) *vengono eseguiti in modo concorrente* (*in UML 1 non era possibile rappresentare ciò*)
Inoltre in UML 2 il collaboration diagram è stato rinominato ***comunication diagram***.

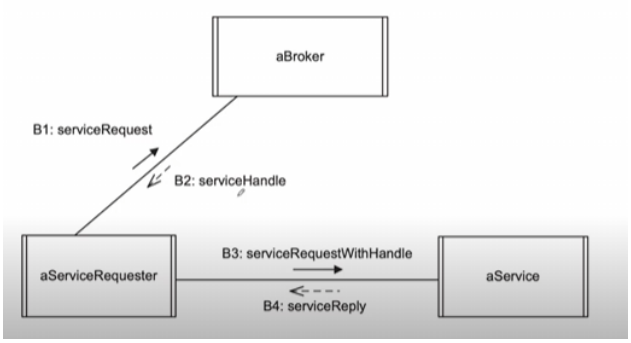
Nel caso in figura il servizio viene registrato presso il broker e una volta ricevuto l'ACK dal broker allora il servizio è stato aggiunto all' "archivio" del broker.

Una volta registrato il servizio i potenziali consumatori possono interagire con il broker e capire dalla descrizione se il servizio è di loro interesse o meno.
A questo punto allora vi sono le due possibilità di **comunicazione diretta** o **intermediata**.

Broker Forwarding Pattern (**pagine bianche): il broker si interpone tra provider e consumer anche dopo la fase iniziale per l'utilizzo del servizio**.
Si parla di pagine bianche perché il meccanismo alla base è simile al meccanismo che si utilizzava con gli elenchi telefonici dove si poteva recuperare il numero di una persona a partire da dettagli come nome e cognome.
Lo stesso avviene per questo protocollo: si assume che il consumer conosca il nome del service provider ma non sappia l'interfaccia di rete per usare il servizio, quindi lo chiede al provider.
In particolare il client manda un messaggio identificando il servizio richiesto, il broker riceve la richiesta e recupera (*in base alle informazioni presenti nel suo archivio dove sono registrati i servizi*) **l'interfaccia di rete presso cui il servizio è reso disponibile, inoltra quindi la richiesta al servizio, prende la risposta e la inoltra infine al consumer.**



Un altro tipo di pattern basato sempre sullo scenario di pagine bianche è il **Broker Handle Pattern**. In questo caso ancora il broker è inizialmente coinvolto per ottenere le informazioni della location del service provider, *ma invece di inoltrare il tutto manda le informazioni al consumer affinché la successiva comunicazione sia diretta con il provider*



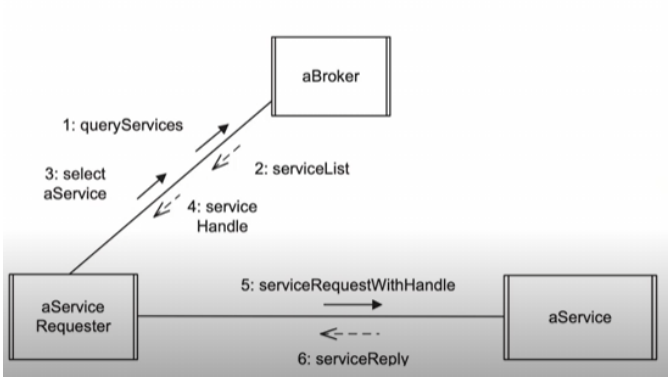
Il vantaggio nell'utilizzo del primo è che è *garantita la location trasparency*: *se cambia interfaccia di rete lo viene a sapere il broker che aggiornando il registro di*

servizi nasconderà automaticamente il tutto al consumer.
Ciò non funziona più con il broker handle pattern, se si cambia l'interfaccia le successive richieste dal consumer che non lo saprà andranno a vuoto e sarà necessaria una nuova comunicazione con il broker. ***Il vantaggio del secondo però è che si scambiano meno messaggi, maggiore efficienza*** (ogni interazione due messaggi uno richiesta uno risposta contro i quattro del Broker Forwarding Pattern)

Se utilizzo il servizio poche volte conviene usare il primo approccio, ma se lo uso molto frequentemente conviene il secondo.

Diverso lo scenario del tipo pagine gialle. Mentre nelle pagine bianche ad es. ordine alfabetico per nome, cognome e indirizzo, *con le pagine gialle si introducono le categorie di servizi* (es. idraulici, elettricisti etc… scelgo quello che fa al caso mio magari perché vicino a casa mia e prendo il numero di telefono).
Quindi la differenza è che non so esattamente il servizio specifico che mi serve ma il tipo di servizio, dalle pagine gialle ottengo un elenco di quella tipologia e potrò scegliere quello che voglio.

Si parla di **Service Discovery Pattern** **(pagine gialle), dove la differenza sta nell'introduzione di una serie ulteriore di messaggi**.
Il consumer richiede al broker una tipologia di servizio (queryServices), ***il broker restituisce una lista di servizi che soddisfano la richiesta*** (sempre tramite l'archivio interno dove i servizi sono registrati), i***l service requester sceglie il servizio da utilizzare***. Da questo punto in poi si può interagire nelle due modalità viste prima: in figura Broker Handle in quanto il broker manda l'interfaccia di rete e il client potrà interagire direttamente con il provider, altrimenti si può utilizzare broker forwarding.



Ma in base a cosa potrei scegliere uno specifico servizio da una lista generica?
In base a vari criteri al di là della funzionalità, come **QoS** (**quality of service**).

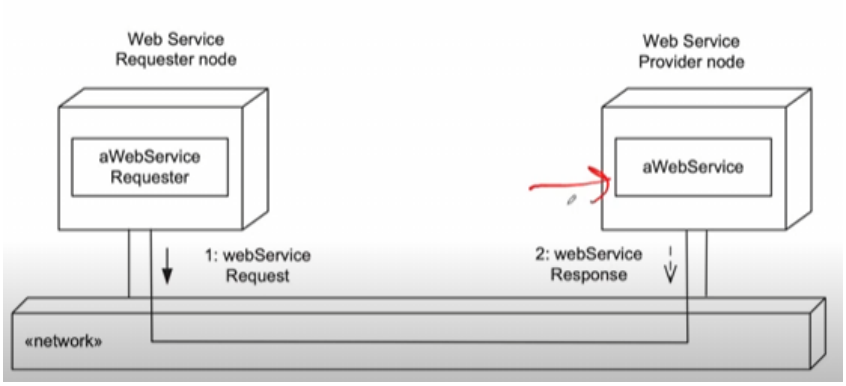
Vediamo ora *quale tecnologia può rappresentare un valido supporto* dal punto di vista implementativo per la realizzazione di applicazioni basate su *service oriented*.
Si fa riferimento alla tecnologia **Web Services**.
Un **Web Service** consente l'interazione tra diversi servizi attraverso l'utilizzo di **protocolli standard di Internet**, e lo **scambio di dati** (come richieste e risposte) avviene tipicamente tramite **linguaggi basati su XML**, che garantiscono interoperabilità tra piattaforme eterogenee.

Una tecnologia implementativa per architettura service oriented deve garantire tutte le funzionalità viste a livello architetturale come broker, descrizione del servizio…
Si introducono quindi degli standard (come **SOAP**) per mettere a disposizione queste funzionalità di base, che come detto fanno uso di XML come formato di serializzazione e di protocolli standard internet, in particolare *tutte le richieste mediate dal broker tra consumer e provider fanno uso dello standard HTTP e la relativa porta 80.*

Si evitano quindi quei problemi evidenziati con tecnologie come CORBA che fanno uso di protocolli proprietari (dove in ambiente distribuito facendo uso di firewall era difficile permettere la comunicazione).
In particolare viene citato il protocollo **SOAP Simple Object Access Protocol**, utilizzato per permettere al consumer di inviare un messaggio al provider come richiesta di esecuzione di una certa operazione.

SOAP è un **protocollo basato su XML** e si compone di tre parti principali**:**
1- La **busta: *indica cosa contiene il messaggio e come deve essere elaborato*.**
2- Le **regole di codifica: *spiegano come rappresentare i dati che vengono scambiati tra chi offre il servizio (provider) e chi lo usa (consumer).***
3- Le **chiamate remote: *definiscono il modo in cui un'applicazione può eseguire funzioni a distanza, come se fossero locali.***

L'uso del Web Services ha come idea di base di rendere disponibili i servizi tramite interfaccia di rete raggiungibile tramite una piattaforma web (non solo usando HTTP come vedremo ma nella maggior parte dei casi si).



Qui un esempio di utilizzo della tecnologia Web Service. La richiesta viene veicolata al service provider e allo specifico servizio messo a disposizione tramite l'uso del protocollo SOAP.
Il diagramma in figura fa riferimento a un diagramma di implementazione che ancora non avevamo visto, in particolare un **deployment diagram** *dove si combina la descrizione della piattaforma di esecuzione e l'allocazione di componenti software su questi elementi della piattaforma.*
In particolare tre nodi (i parallelogrammi) che rappresentano nodi di esecuzione il primo quello che esegue il consumer (contenente l'oggetto in esecuzione, la componente awebservice requester), il secondo del provider (contenente l'oggetto …

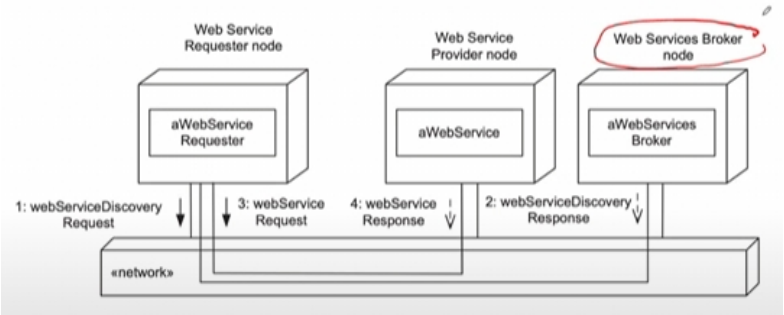
aWebService) e quello sotto quello che rappresenta l'interfaccia di rete che collega i due (*basata ovviamente sul web e quindi protocolli internet, in particolare HTTP*).

Quando il consumer ha le informazioni necessarie per identificare il servizio che necessita (oggetto aWebService Requester) grazie al protocollo SOAP posso confezionare la richiesta da inviare al servizio, che potrà elaborare la richiesta e rispondere ancora usando SOAP.
Ma come ottenere queste informazioni necessarie (la sintassi affinché possa avvenire la richiesta)? *Grazie alla descrizione del servizio, che sappiamo essere inviata al broker dal service provider all'atto di registrazione del servizio*.

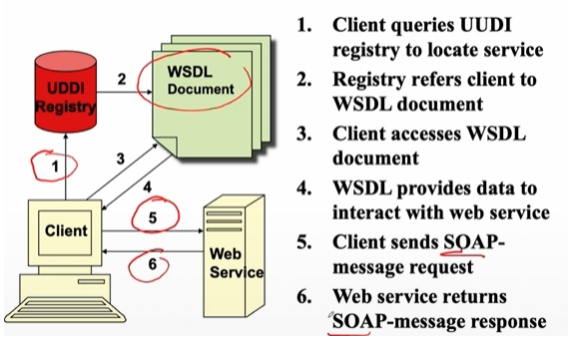
Si introduce quindi nell'ambito della tecnologia WebService un ulteriore standard per scrivere la descrizione del servizio.
Questo standard si chiama **WSDL Web Service Description Language**, *linguaggio ancora basato su XML che permette di fornire SOLO le informazioni necessarie del servizio tramite un documento WSDL al broker e quindi al potenziale consumatore* (**principio di discovery**).
Il documento WSDL oltre alla descrizione contiene ovviamente anche il broker handle, il riferimento che il consumer potrà utilizzare per interagire direttamente con il provider.

Quindi la tecnologia WebService mette a disposizione un protocollo per definire i messaggi da scambiare (**SOAP**) e un **linguaggio per descrivere i servizi** (**WSDL**).
Ci manca il terzo elemento, ***come realizzare il concetto di service registry*** (broker)?
Viene introdotto *un particolare framework*, **UDDI Universal Description Discovery and Integration.**
Tale framework realizza quindi tutte le funzioni base che deve svolgere il broker.

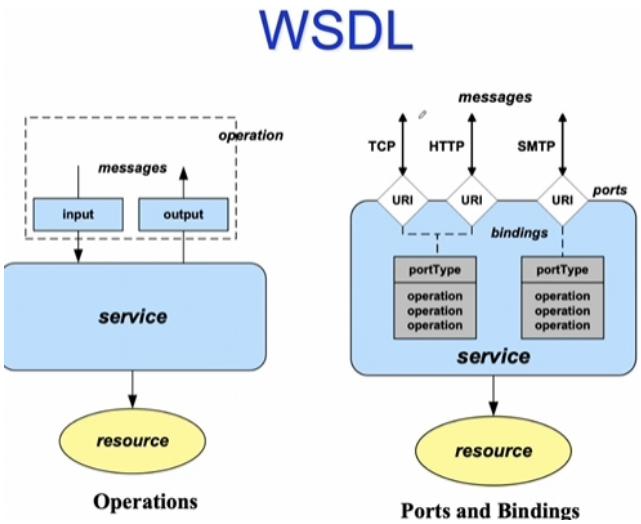
Quindi in generale quando parliamo di tecnologia implementativa a supporto di architetture service oriented si fa riferimento a **come realizzare il broker**, **come descrivere il servizio** e **come interagire con il servizio**, **nel caso web services UDDI, WSDL e SOAP.**



Qui un esempio di Web Service in cui interviene anche il broker su un altro nodo remoto, il consumatore chiede al broker informazioni sul servizio per ottenere eventualmente il documento WSDL e interagire quindi col provider.
Tutto il processo riassunto in questa slide:



WSDL mette a disposizione solo gli elementi di base per utilizzare il servizio (quindi niente roba al di là dell'interfaccia tipo QoS), ossia l'insieme di operazioni che possono essere invocate (del tutto simile all'interfaccia pubblica di una classe che mi dice i metodi che posso invocare, la differenza sta nel fatto che in un'applicazione object oriented la richiesta è veicolata nello stesso spazio di memoria mentre nel secondo caso la richiesta è remota), la network location presso cui raggiungere il servizio ed inviare la relativa richiesta SOAP.



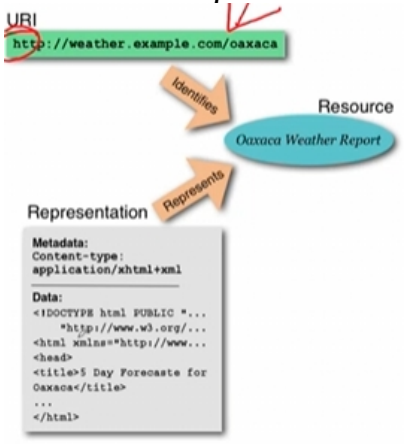
Un documento WSDL come si vede è diviso in due parti separate (livello di astrazione): *a sinistra livello di dettaglio relativo alle operazioni*. WSDL mette a disposizione le operazioni in termini di messaggi input e output (esiste in realtà anche il messaggio di fault oltre a input e output se errore). Questa parte descritta in XML.
A destra invece si mostra come le operazioni sono concretamente messe a disposizione sull'interfaccia di rete: es. la richiesta è inviata tramite HTTP e raggiunge uno specifico indirizzo di rete URI e lo strumento utilizzato per mettere a disposizione le informazioni a livello di interfaccia di rete è il portType.
Un portType è null'altro che un insieme di operazioni messe a disposizione su uno o più URI (endpoint). Come detto si utilizza maggiormente HTTP (sincrono) per condividere questi messaggi ma si potrebbero utilizzare anche altri protocolli come TCP o SMTP (asincrona).

Nel tempo sono stati introdotti anche ulteriori contributi che non sono alternativi alla tecnologia Web Service ma che si affiancano ad essa, uno di questi è **REST** **REpresentational State Transfer**.

Questo più che essere una tecnologia implementativa (come WebService) è uno stile architetturale (come gestire l'architettura di un sistema software distribuito usando sempre protocolli standard internet, in particolare HTTP).
REST mette a disposizione un interfaccia di rete (API) per accedere ad un insieme di risorse messe a disposizione sulla piattaforma web (queste risorse sono di diversi tipi e possono anche essere servizi web).

La particolarità di REST è che HTTP non viene usato solo come canale di comunicazione, ma anche come modello di interazione: si usano esclusivamente le operazioni standard di HTTP (i cosiddetti verbi) per accedere e manipolare le risorse..
Caratteristiche: *ancora una volta* ***client/server*** *quindi un server mette a disposizione una risorsa con cui si può interagire facendo uso solo di HTTP*, ambiente ***Stateless*** *in quanto non si salva lo stato nell'interazione c/s*, vengono però messe a disposizione delle ***cache*** per migliorare l'efficienza in una rete basata su REST, ***interfaccia uniforme*** in quanto le interazioni sono basate solo e soltanto sui 4 verbi di base HTTP ossia GET PUT POST (aggiornare) DELETE (**CRUD**!), *REST si fonda sul concetto di risorsa che deve essere identificata con URL specifico* (o URI), inoltre ***le rappresentazioni delle risorse sono interconnesse***, ossia le risorse possono essere collegate tra loro tramite URL, creando una rete navigabile di informazioni.

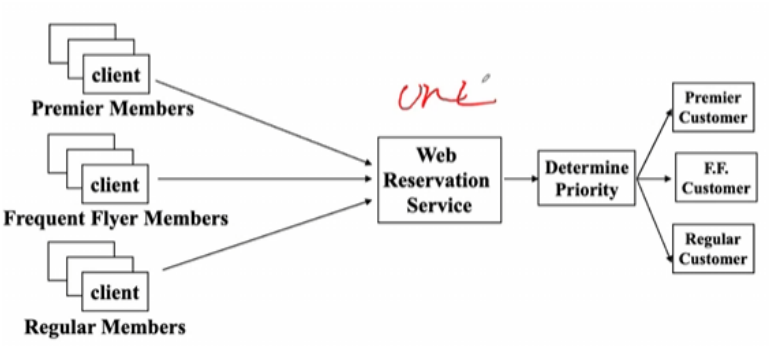
Tutto si basa sulle risorse web e ogni entità distinguibile è risorsa (sito web, pagina html, documento xml, web service, etc…) **e sono identificate da un URL**.
La risorsa è tipicamente rappresentata facendo uso di un documento XML.



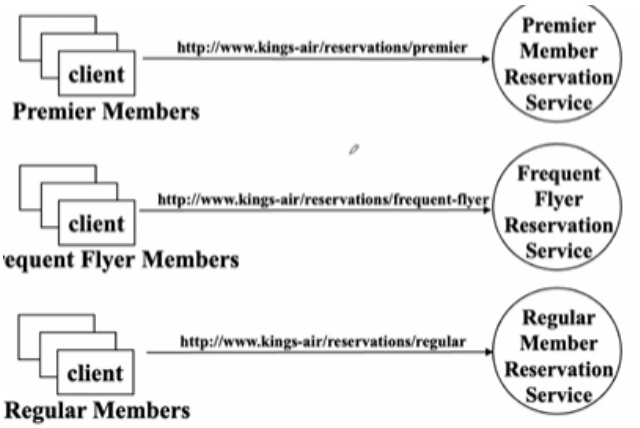
Per questo abbiamo detto che REST non rappresenta un'alternativa a Web Service (egli stesso li fornisce attraverso non più un'interfaccia proprietaria ma un meccanismo diverso).
Per interagire con le risorse si utilizzano le RESTful API che usano le 4 operazioni CRUD HTTP in base al "contesto": se si deve prendere una Collection di risorse (es. /users nell'URL) o un singolo Item (es. /users/{id}). *In particolare Post può essere usato solo nelle collezioni per aggiungervi un item mentre Get sia sulle collezioni che sui singoli item (nel primo caso per ottenere la lista di elementi e nel secondo il singolo elemento), invece PUT e DELETE utilizzabili solo negli item*.

<u>Ma cosa mi cambia se uso il design convenzionale per Web Services o se ci aggiungo REST</u>? Vediamolo con un esempio concreto dove confrontiamo una progettazione basata su una e sull'altra.
Si vuole realizzare un servizio di prenotazione biglietti aerei online, e si vuole fornire un servizio di qualità differente in base al tipo di utenti (premier member servizio immediato, frequent flyer members servizio spedito, e gli altri servizio regolare).

<u>Nell'approccio convenzionale si fa uso di un singolo URL per mettere a disposizione il servizio che offra le tre tipologie di accesso</u>.
<u>L'approccio basato su REST invece fa uso di più URL</u>.



Nel primo caso regular, frequent e premier accedono tutti allo stesso URL e in base al tipo si determina la priorità della richiesta e inoltra quindi la richiesta alle parti di servizio che gestiscono la priorità.
Il problema di ciò è che si sta assumendo che usare tre URL diversi è più costoso, e l'affidabilità del servizio non è supportata visto che a questo punto quel singolo URL è un central point of failure e bottleneck.



Nel secondo caso invece non c'è l'esigenza di porre attenzione sul numero di URL che si utilizza -> si evitano i problemi.

Lez 25 (18/03)

(Riassuntino) Siamo ancora nell'argomento riguardante le tipologie di architetture che possiamo utilizzare nella sottofase di progettazione preliminare.
Abbiamo introdotto gli elementi alla base di queste architetture e alcuni pattern di interazione tra i due attori principali: service provider e consumer tra cui si interpone il broker (che può essere più o meno coinvolto).
Dopodiché abbiamo parlato delle tecnologie implementative (come realizzare concretamente una architettura service oriented) facendo uso di tecnologie che mettano a disposizione i tre elementi alla base di una architettura service oriented: protocollo per scambiare i messaggi tra consumer e provider (*nel caso dei WebServices SOAP, basato sulla serializzazione XML dei messaggi scambiati), descrivere il servizio offerto per garantire discoverability (linguaggio WSDL anch'esso basato su XML per descrivere i servizi solo dal punto di vista d'interfaccia, non dettagli tipo QoS), e infine come mettere a disposizione (implementare) i servizi offerti dal broker (UDDI).*)
Abbiamo parlato di WebServices perché è stato l'approccio che ha preso maggiormente piede nell'implementare l'architettura service oriented e abbiamo anche parlato di REST, che non rappresenta un'alternativa ai WebServices quanto qualcosa che può essere usato insieme ad essi, uno stile architetturale (REST fondamentalmente mette a disposizione risorse di rete permetto di far uso e interagire con esse facendo uso solo ed esclusivamente di protocolli standard internet come soprattutto HTTP).
REST non è alternativo ai Web Services perché tra queste risorse possiamo includere anche i servizi web stessi.
Abbiamo infine visto come l'uso di REST offra comunque vantaggi in termini di scalabilità e affidabilità (si evita il single point of failure).

Si torna ora al livello di astrazione di progettazione preliminare, scendendo più in dettaglio, per vedere pattern/protocolli che si usano in applicazioni service oriented per garantire certe proprietà.
Un aspetto fondamentale che deve essere garantito anche dalle applicazioni service oriented è quello legato alle proprietà di una transazione (studieremo quindi i **software architectural transaction patterns**).
Una Transazione rappresenta una richiesta effettuata da un client che contenga due o più operazioni, che però svolgono una singola funzione logica e devono

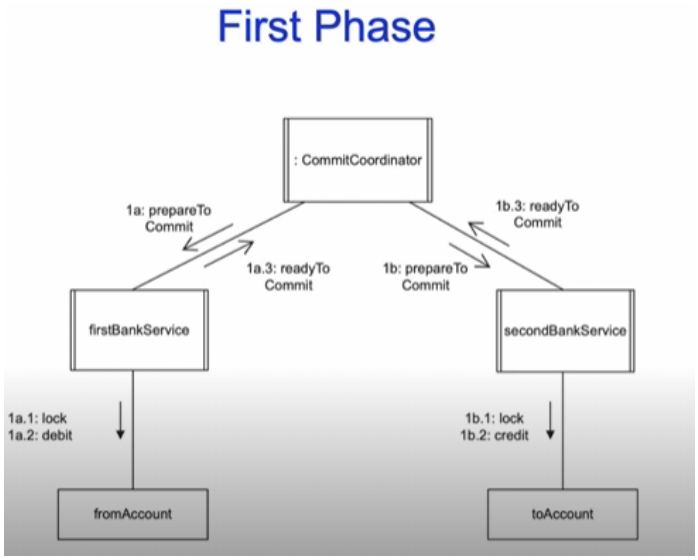
essere completate interamente o per nulla.
(*es. si vogliono trasferire i soldi al conto di un amico: le operazioni sono togliere i soldi dal mio e inviarli al conto dell'amico, entrambe costituiscono la stessa transazione in quanto hanno lo stesso scopo logico di trasferire i soldi e o entrambe riescono o entrambe falliscono, non posso svolgerne una e lasciare l'altra fallita*).

Le **proprietà di una transazione** sono racchiuse nell'acronimo **ACID** **Atomicity**, **Consistency**, **Isolation** e **Durability**.
Atomicità == *la transazione pur essendo insieme di operazioni è vista come unità indivisibile di lavoro, o tutto (committed) o niente (rolled back)*

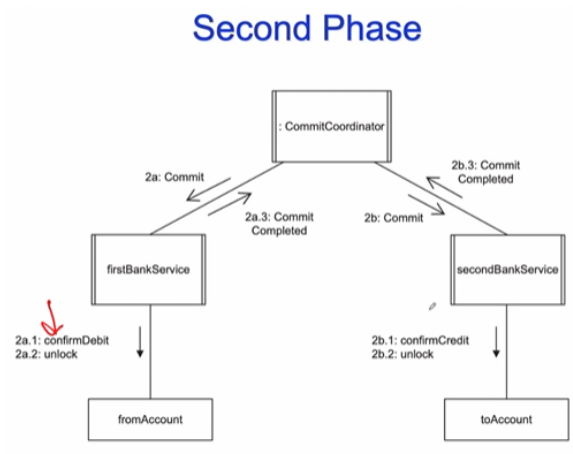
Consistenza == *quando si esegue una transazione* (sia di successo che non) *il sistema deve rimanere in uno stato consistente*
Isolation == *ogni transazione deve essere eseguita in modo isolato e quindi non essere compromessa da altre transazioni*
Durability == *gli effetti prodotti dalla transazione sono permanenti e devono quindi sopravvivere anche ad eventuali system failures*.

Come garantire queste proprietà in un ambiente distribuito basato su servizi?
Usiamo dei casi di studio per capire come e dove applicare questi pattern.
L'esempio tipico di una transazione è la transazione bancaria.
Quando avviene un trasferimento di denaro (es. bonifico da conto corrente a conto corrente) ciò che deve avvenire è che i soldi del conto corrente di origine vengano scalati e quelli di arrivo aggiornati. Ciò che non può e non deve succedere è che vengano scalati i soldi ma non aggiunti all'altro conto corrente e viceversa.
Nell'esempio specifico il primo conto è savings account (soldi per investimenti) e il secondo checking account (conto utilizzato normalmente per fare bonifici).
La transazione consisterà nelle due operazioni debit sul primo e credit sul secondo e deve essere o committed (i soldi sono trasferiti con successo) o aborted (nessuna delle due operazioni avviene mantenendo lo stato del sistema consistente).

Per garantire le proprietà di questa transazione si utilizza il protocollo (anche in basi di dati) del **Two Phase Commit Protocol**.
Vi sono **due servizi nella transazione di trasferimento bancario** ognuno su un'interfaccia di rete differente: ***firstBankService*** **per prelevare i soldi** dal saving account e ***secondBankService*** **per permettere il deposito del denaro sul secondo conto**.
Per coordinare correttamente le attività svolte dai due servizi dobbiamo coinvolgere anche un altro oggetto che svolga il ruolo di coordinatore, il ***CommitCoordinator***.
Vediamo come funzionano le **due fasi**.



La transazione è gestita in modo centralizzato per cui sarà il commitcoordinator a gestire l'ordine della transazione.
In figura si vede il comunication diagram (ossia il collaboration diagram come è stato rinominato in UML 2).
Il commitcoordinator invia un primo messaggio 1a e 1b ai due servizi coinvolti in cui gli comunica di prepararsi al commit. A fronte di questa richiesta i due servizi effettuano 1a.1 e 1b.1, ***lock per garantire l'Isolation e quindi la Consistency*** *(non si vuole che altre transazioni operino mentre i due servizi effettuano l'operazione sui due conti). A questo punto il firstBankService farà l'operazione di debito e l'altro di accredito*. Se le operazioni vanno a buon fine i servizi inviano al coordinator un readyToCommit, in assenza di questo messaggio la transazione è abortita.
Se invece a buon fine si passa alla **seconda fase**.

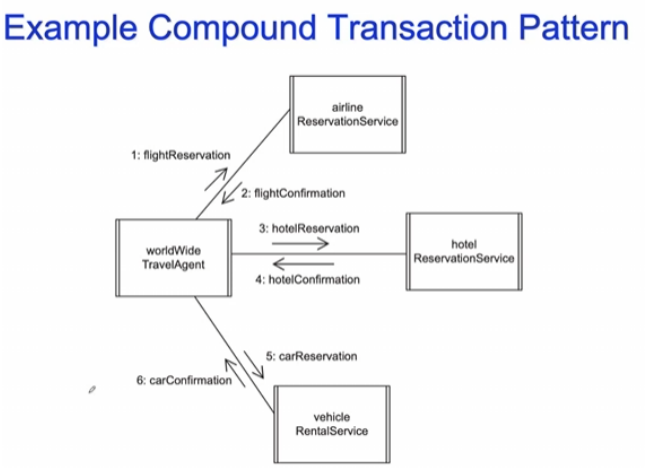


Viene inviato commit dal coordinatore, confermata l'operazione di addebito e di accredito (2a.1, 2b.1), si libera il lock e viene completata la transazione.

Vediamo ora esempi di transazioni più complesse, partendo dalla **Compound Transaction** per poi descriverne il Pattern da adottare.
La compound transaction è una transazione composita, ossia costituita da un insieme di sottotransazioni. *Mentre una transazione singola (flat) come quella della*

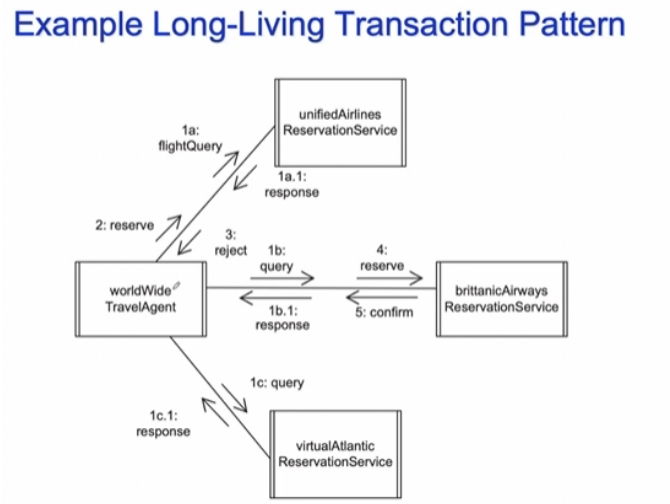
banca si basava sulla regola "o tutto o niente", nel caso di una compound si parla di una transazione costituita da singole sottotransazioni -> **in caso qualcosa vada storto si cerca di salvare il salvabile, ossia se la seconda transazione non va a buon fine ma la prima si si farà un rollback "parziale", ossia rollback solo sulla seconda transazione.**

Per illustrare questo tipo di transazione facciamo un esempio concreto: un agente di viaggio deve pianificare il viaggio per un cliente: è necessario prenotare il biglietto aereo, poi prenotare l'albergo e la macchina a noleggio. Piuttosto che vedere questa compound transaction come operazione indivisibile per cui o tutto o niente posso vederla come costituita di tre sottotransazioni in modo che se riesco a fare la prima ma non la seconda mi salvo la prima.



Qui il caso in cui tutte e tre le sottotransazioni hanno esito positivo. Nel caso in cui una singola sottotransazione non abbia esito positivo, si salvano le prenotazioni che hanno avuto successo.

Questa transazione diventa ancora più complessa quando si mette in mezzo anche l'elemento umano, si parla di **Long Living Transaction**.
In queste transazioni si ha il così detto human in the loop per cui si deve tener conto di possibili ritardi dovuti alla decisione di persone coinvolte nelle operazioni.
Il pattern in questo caso prevede di organizzare la transazione distinguendo in essa due o più sottotransazioni separate proprio dall'elemento umano.
Un esempio concreto può essere il considerare una prenotazione aerea in cui c'è il coinvolgimento di un essere umano, dove l'utente fa una ricerca per capire i posti disponibili in un certo volo, scegliere un posto disponibile e confermare la prenotazione. Per effettuare la scelta però l'utente umano impiega del tempo, e in questo periodo può succedere che il posto scelto dall'utente venga prenotato da qualcun altro. Quindi prima di riconfermare la transazione quindi è necessario un **recheck**



Nell'esempio in figura il servizio che funge da intermediario tra utente e compagnia aerea (TravelAgent) chiede a 3 compagnie aeree (1a, 1b, 1c) di mostrare i posti disponibili in un certo volo.

Ottenuta la risposta l'utente decide di prenotare il posto in unifiedAirlines, (2. reserve), il servizio fa un recheck che ha esito negativo e quindi invia 3. reject all'utente che dovrà rivolgersi quindi ad un'altra compagnia aerea (in questo caso britannic) prenotando e in questo caso il recheck ha esito positivo e quindi la transazione ha successo.

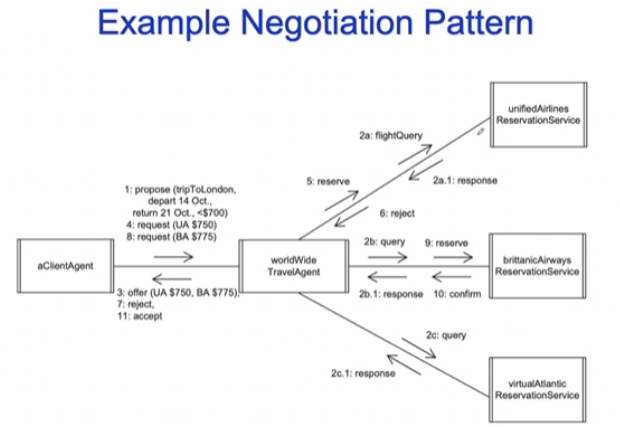
Ultimo caso che consideriamo è quello del **Pattern di Negoziazione**.
Anche chiamato **Agent Based Negotiation** in quanto *subentra la figura di un servizio che lavora per conto dell'utente* (l'utente si affida al servizio chiedendogli di far qualcosa al posto suo).
Es. *riguardo la transazione per prenotazione posti vista prima si immagina un servizio che permetta, sapendo che devo andare a new york tot data, di trovare le offerte disponibili (senza che io utente vada a vedere il sito di ogni compagnia aerea).*
*Si ha quindi un **client agent** che lavora per conto dell'utente, egli interagirà con il **service agent** per soddisfare le esigenze del cliente*.
Il service agent mostra una lista di offerte al client agent che si avvicinano a soddisfare la sua richiesta. A questo punto il client agent lavorando per conto dell'utente può decidere se rifiutare/accettare una richiesta etc… (da qui negoziazione)

In particolare il **client agent** svolge tre tipi di operazioni:
- **Proposta di servizio** (es. cerca volo roma new york a meno di 1000 euro)
- **Richiesta servizio** *se a fronte dell'offerta (risposta alla proposta) del service agent è soddisfatto*
- **Rifiutare il servizio altrimenti**

Invece dal punto di vista del **service agent**:
- **Offrire un servizio** di fronte alla proposta di servizio del client agent

- **<u>Rifiutare una richiesta/proposta del client agent in base alle disponibilità</u>**
- **<u>Accettare la richiesta/proposta del client agent</u>**.

<u>La principale differenza tra richiesta e proposta è che la seconda è negoziabile</u> (es. cerco il volo da roma a ny a meno di 1000 euro ma sono disposto a vedere anche offerte di prezzo lievemente superiori), **<u>mentre la richiesta è imperativa</u>**.



In questo esempio 1. Il clientagent invia una proposta (negoziabile!) per viaggio a londra il 14 ottobre e ritorno il 21 che costi meno di 700 euro.
A fronte di questa richiesta il serviceagent si confronta con diversi servizi offerti da compagnie aeree inviando in modo concorrente le richieste (2a, 2b, 2c).
Il messaggio che torna al clientagent è una offer con le soluzioni migliori trovate dal serviceagent (unifiedairlines a 750 euro o britannic a 775).
Il client invia quindi un messaggio di richiesta (non negoziabile!) (4) per prenotare UA, ma nel frattempo biglietto già acquistato da qualcun altro quindi reject dalla compagnia al serviceagent e reject dal serviceagent al clientagent (7).
Allora il clientagent richiede quello a 775 di BA, in questo caso la reserve da parte del serviceagent ha successo -> il serviceagent risponde con accettazione finale alla richiesta non negoziabile del clientagent.

I servizi sono quindi elementi che interagiscono tramite scambio di messaggi, ed è quindi fondamentale come visto saper gestire la <u>logica di controllo</u> delle applicazioni.
Un altro problema fondamentale è saper **progettare correttamente <u>l'interfaccia dei servizi</u>** (*stesso problema anche per le classi nell'analisi dei requisiti object oriented, ma qua ancora più forte perché mentre la classe è un elemento ben identificato sappiamo come la granularità dei servizi sia molto variabile e non nota all'utente del servizio*).
Possiamo sfruttare UML 2 e in particolare il concetto di <u>classi strutturate</u> per progettare un servizio come se fosse una classe avente un'interfaccia, che però non è necessariamente monolitica ma composita e quindi costituita eventualmente da ulteriori servizi al suo interno.
Quindi da una parte **progettazione del servizio a <u>livello strutturale</u>**, **dall'altra a <u>livello comportamentale</u>** (dinamica attraverso cui questi servizi interagiscono), dove una parte fondamentale è <u>come coordinare questi servizi</u> (*infatti a differenza della*

architettura component based dove i framework permettevano di guidare anche l'integrazione tra le varie componenti, nelle service oriented i servizi sono indipendenti e se me ne servono diversi devo saperli coordinare).

In questo contesto di **Service Coordination** **in SOA** (service oriented architecture) si utilizzano due termini per indicare due tipi di coordinamento tra servizi: **orchestrazione e coreografia**.
Orchestrazione == ***si ha un elemento centralizzato che coordina l'esecuzione dei servizi*** (simile a quanto visto nel caso dei two phase commit protocol)
Coreografia == ***gli aspetti di coordinamento sono decentralizzati*** (distribuiti)

Nella progettazione di applicazioni service oriented tuttavia è difficile che venga usata piena orchestrazione o piena coreografia, tipicamente si usano entrambi gli approcci. Per questa ragione in generale con il termine coordinazione si vuole intendere il **controllo e la sequenza delle azioni tra i servizi**, indipendentemente dal fatto che siano centralizzati o distribuiti.
I pattern delle transazioni visti in precedenza possono essere usati per la coordinazione tra servizi.

Con ciò la sottofase di progettazione preliminare è conclusa.

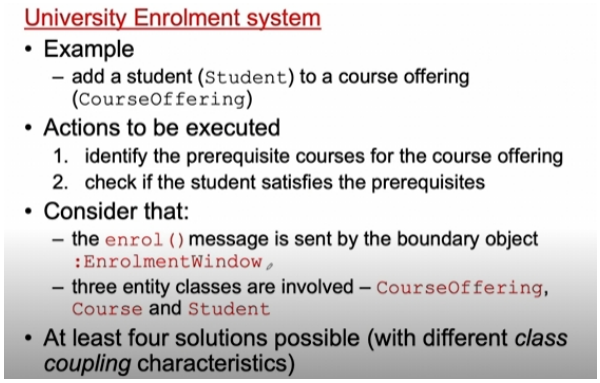
Tornando all'inizio della fase di progettazione avevamo detto come sia costituita da due sottofasi: la prima in cui definire l'architettura software (di cui ci siamo già occupati) e la seconda di progettazione dettagliata (*tutti gli elementi da progettare in modo dettagliato sia dal punto di vista strutturale che comportamentale*).
Nel nostro caso avendo usato un approccio OO fin dall'analisi dei requisiti parleremo di una sottofase di progettazione dettagliata che farà uso di un approccio OO -> **Detailed OOD**.

OOD rappresenta una diretta continuazione di OOA, dove secondo il principio di stepwise refinement raffineremo sempre più in dettaglio quanto trovato nella prima parte del corso, *in particolare occupandoci di come far collaborare gli oggetti affinché vengano resi disponibili i servizi* che l'applicazione dovrebbe fornire.
Mentre l'OOA era guidata dai casi d'uso quindi l'OOD è guidata dalla collaborazione degli oggetti.

La collaboration è primitiva UML (per questo in UML 2 il collaboration diagram rinominato in comunication diagram) **ed è costituita da due parti fondamentali** (qui torniamo a quanto visto nella fase di analisi orientata agli oggetti, dove il modello del sistema software era costituito da vari sottomodelli: dati, comportamentale e dinamico):
- ***una parte comportamentale***: **rappresenta la dinamica che mostra come gli elementi collaborano tra loro**. Si definirà facendo uso dei **comunication diagrams**.
- **una parte strutturale**: **rappresenta gli aspetti statici della collaborazione ed è definita facendo uso del class diagram**, *facendo però anche uso dell'estensione*

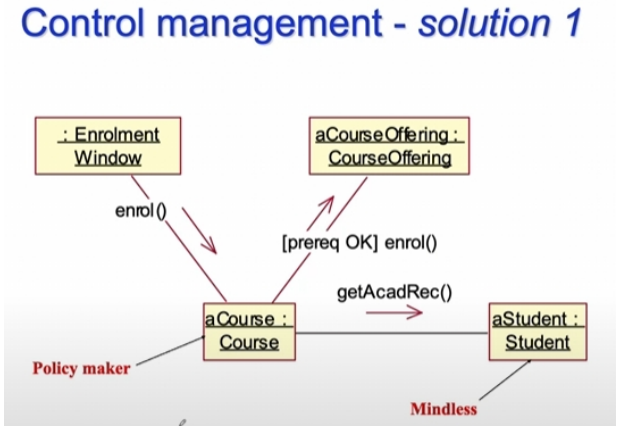
fornita da UML 2 che permette di progettare la struttura della classe in modo gerarchico (quindi useremo i **composite structure diagram**)

Uno degli aspetti più importanti di cui tener conto in fase di progettazione a livello comportamentale è la **gestione del controllo** (la logica che l'applicazione utilizza per eseguire le funzioni che mette a disposizione).
Questa parte ricade nel **Control Management**.
Iniziamo con un esempio tornando al sistema di iscrizione all'università.



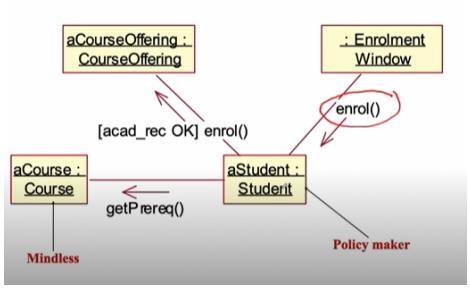
Vediamo come gestire il controllo nel caso in cui si voglia aggiungere uno studente a una specifica courseoffering. Nel sistema essendo OO avrò ad es. l'oggetto mario rossi studente e l'oggetto ISW courseoffering.
Prima di poter iscrivere lo studente al corso si devono identificare gli esami propedeutici e verificare che lo studente li abbia superati.
Gli oggetti entity di nostro interesse per questa interazione sono classe Studente, Courseoffering e Course (course memorizza le informazioni legate a un corso, courseoffering riguardo un corso istanziato in uno specifico semestre ed anno, e gli esami propedeutici stanno in Course).
Lato interfaccia utente invece si ha l'oggetto boundary che riceve la richiesta dell'utente (segreteria studenti) di iscrivere lo studente al corso.

Diverse soluzioni per il control management:

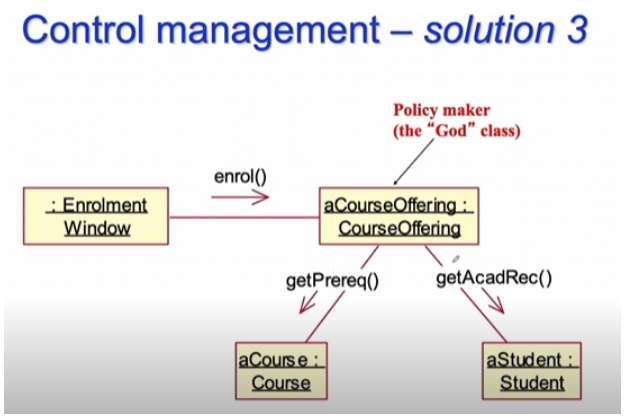


Comunication diagram dove l'oggetto enrolment window (boundary) inoltra la richiesta di iscrizione all'oggetto corso, che chiede all'oggetto studente qual è il suo

curriculum. Se si verifica che lo studente ha superato i corsi propedeutici allora conferma l'iscrizione inoltrando il messaggio all'oggetto courseoffering, che iscriverà lo studente alla specifica istanza di corso.
*Qui chi gestisce il controllo è l'oggetto di corso (**Policy Maker**) mentre student è **mindless** perché risponde semplicemente alla richiesta di corso*.

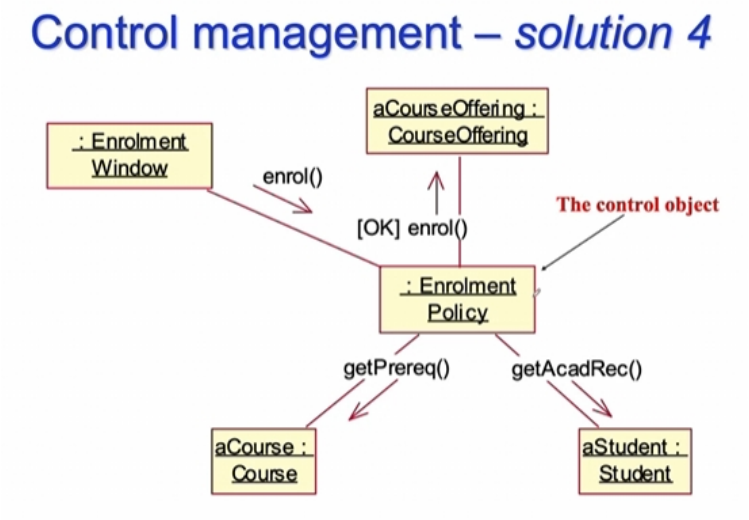


La seconda soluzione prevede invece che la richiesta sia inanzitutto inoltrata allo studente, che chiede al corso quali sono gli esami propedeutici che deve aver passato e se tutto ok (student fa controllo interno per vedere se li ha superati) inoltra a courseoffering. (***si invertono i ruoli di controllo***)



Si centra il tutto sull'oggetto CourseOffering che *diventa **classe God*** **in quanto decide tutto**. È infatti courseoffering a chiedere a course gli esami propedeuitici e a student i risultati per capire se li ha passati o meno.

Queste soluzioni rispettano l'approccio BCE, definito alla fine della parte di analisi dei requisiti? (boundary control entity)
In una stratificazione corretta delle classi devono esserci oggetti boundary che si devono occupare solo di catturare le richieste dell'utente e inoltrarle agli oggetti di controllo, che conoscono la logica applicativa e interagiranno con gli oggetti entity (oggetti che mantengono le informazioni) al fine di soddisfare la richiesta.
Ma in questo caso sto dando responsabilità di controllo a un oggetto entity -> non rispettano la stratificazione BCE.



Si introduce quindi un oggetto di controllo che si occupi della logica di esecuzione.
Corso fornisce gli esami propedeutici, student la carriera universitaria e courseoffering interpellato solo eventualmente per iscrivere lo studente, si garantisce che le classi entity si limitino ad avere le informazioni senza controllare.
Isolando la logica di controllo in una classe apposita molti vantaggi tra cui a livello di **manutenibilità** (eventuali modifiche sul come gestire le iscrizioni sono limitate a questa classe senza dover cercare una classe che si occupasse anche di quello)

NB RISPETTA SEMPRE BCE ANCHE NEL PROGETTO.

In questo caso (così come visto nelle applicazioni service oriented) **gioca un ruolo fondamentale il coupling** (grado di accoppiamento). *Il più desiderabile è* **l'Inter-Layer Coupling**: ***si vogliono evitare delle dipendenze tra elementi appartenenti a diversi layer applicativi*** (*con layer applicativi si intende BCE*).
Si vuole favorire quindi **l'Intra-Layer coupling** *ossia l'interazione all'interno di uno stesso strato piuttosto che l'interazione tra strati differenti*.
Si vuole quindi minimizzare **l'Inter-Layer Coupling**, **e uno strumento utile per farlo è la Legge di Demeter.**

Lez 26 (25/03)

La **Legge di Demeter** (*anche nota come "don't talk to strangers" in quanto si basa sull'idea di non "comunicare" con oggetti non noti*) **è una legge che aiuta a mantenere basso l'accoppiamento inter-classi migliorando la manutenibilità del codice.**
Essa afferma che **un metodo può inviare messaggi (cioè invocare metodi) solo ai seguenti oggetti:**
1) **L'oggetto del metodo stesso** (un metodo deve poter invocare i metodi su se stesso, es. usando this in Java e C++)
2) **Oggetti passati come argomenti nel metodo** (perché essendo l'oggetto formalmente presente nei parametri dell'operazione e quindi necessario per la sua

esecuzione è un oggetto noto, si conosce la classe dalla quale questo oggetto è creato)
3) **Oggetti appartenenti a uno degli attributi dell'oggetto corrente** (ossia se tra gli attributi dell'oggetto che ha invocato me metodo vi sono altri oggetti io metodo posso invocarne i metodi, tuttavia in questo caso vale la strong law, ossia non posso invocare i metodi degli oggetti che sono a loro volta loro attributi (annidati))
4) **Un oggetto creato dal metodo**
5) **Un oggetto che fa riferimento a una variabile globale**

Come detto la Legge di Demeter favorisce quindi manutenibilità ma anche comprensione del codice, *infatti per ad es. ridurre il numero di righe si potrebbe pensare di fare invocazioni a partire dal mio metodo ad un oggetto noto a per poi invocare altri metodi non noti a.b().c().d(), tuttavia ciò non favorisce la comprensibilità e quindi la manutenibilità del codice in quanto un programmatore diverso potrebbe non capire il senso di tale scelta*.

Ora quello che ci resta per completare questa parte di progettazione OO è capire come utilizzare UML per produrre i diagrammi di implementazione, che ancora non abbiamo considerato. *Essi permettono di definire in modo preciso come gli oggetti che progettiamo vengono tradotti in componenti eseguibili.*

Come abbiamo già accennato una delle novità più interessanti di UML 2 è l'introduzione delle **structured classes**. *Mentre in UML 1 la classe è intesa come semplice aggregato di dati e operazioni*, **in UML 2 si mantiene lo stesso simbolo** (forma rettangolare) ***ma diventa "classe strutturata" in quanto contiene elementi detti roles o parti che formano la sua struttura e ne descrivono il comportamento***. Questo meccanismo è **gerarchico** in quanto un ruolo/parte di una classe strutturata può essere a sua volta classe strutturata, ciò permette di gestire la complessità in modo stratificato lavorando a diversi livelli di astrazione (un po' come avevamo fatto con i DFD data flow diagram che permettevano di rappresentare il comportamento del nostro software a diversi livelli di astrazione).

Ogni ruolo/parte rappresenta un elemento partecipante nella realizzazione della struttura interna della classe, e questi sono interconnessi attraverso il concetto di *connettore* (come il ruolo anche il connettore rappresenta quindi una primitiva in UML 2, che permette appunto di descrivere come queste parti interagiscono tra loro)

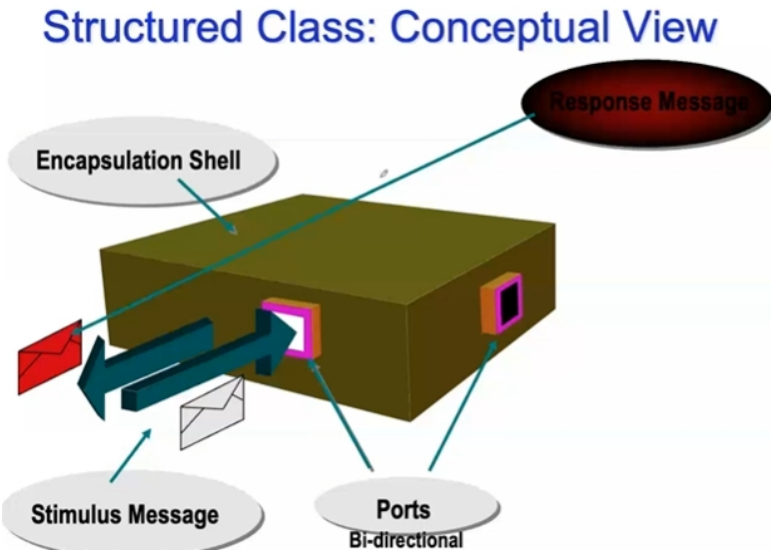
Esiste una differenza tra ruoli e parti anche se molto sottile: ***i ruoli seguono una semantica "per riferimento"*** mentre ***le parti "per valore"*** (quindi i ruoli rappresentano degli elementi presenti nella classe per riferimento mentre le parti elementi sono proprio contenuti nella classe strutturata) (simile al ragionamento fatto introducendo i costrutti per aggregation e composition: la prima semantica per

riferimento perché l'oggetto contenuto non è fisicamente contenuto ma utilizzato attraverso riferimento esterno, mentre nella composition l'oggetto è proprio fisicamente contenuto -> **existence dependency** etc…)

Le structured classes sono molto utili in fase di progettazione. Mentre in fase di analisi dei requisiti ci focalizziamo sul COSA identificando le classi che definiscono gli elementi fondamentali e dichiarandone le operazioni solo con la loro firma e le relative associazioni, in fase di progettazione dobbiamo definire la struttura interna della classe, vedendo ad es. ciascuna operazione COME viene realizzata.
Con UML 1 dovremmo limitarci a progettare ogni operazione dal punto di vista algoritmico, mentre con UML 2 possiamo delegare a dei ruoli interni la realizzazione di queste operazioni.

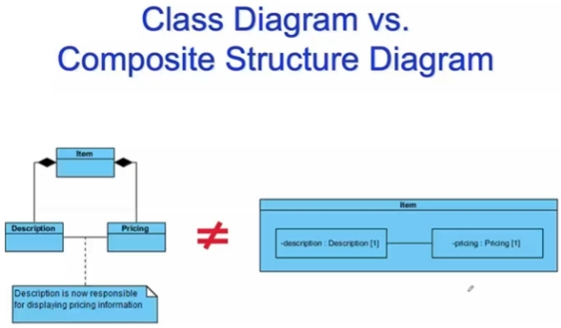
In questo senso si utilizza la classe strutturata per definire i building blocks di un'applicazione, nascondendo i dettagli implementativi.
In particolare vi è un incapsulamento molto stretto del comportamento, per cui ogni interazione tra elementi di classi diverse deve avvenire attraverso un meccanismo basato su messaggi (qui si torna al ragionamento fatto sul Coupling: il livello migliore è Data/Stamp dove l'interazione tra moduli avviene tramite scambio di messaggi)

Si vede quindi concettualmente la classe strutturata come una scatola nera che mette a disposizione servizi (operazioni elencate nell'interfaccia pubblica della classe) nascondendo l'implementazione delle operazioni agli utilizzatori della classe.
Per utilizzare i metodi messi a disposizione dalla classe si utilizzano delle **Porte**, *attraverso le quali inviare messaggi e riceverne eventualmente indietro*.
Inoltre all'interno della classe strutturata possono esservi altre classi strutturate, altre classi semplici etc…



Internamente poi deve essere definito come viene realizzata la classe strutturata sia in termini strutturali che comportamentali. ***Per la parte comportamentale si utilizzano spesso macchine a stati finiti*** (come la classe evolve durante il suo funzionamento transitando da uno stato iniziale a uno finale).
In questo senso quindi ogni classe può essere vista come un elemento di progettazione autonomo, molto utile in fase di progettazione perché quindi posso suddividere il team per lavorare su classi diverse partendo dall'interfaccia definita in fase di analisi dei requisiti, utile anche a livello di testing (**si testa a livello di unità Unit-Testing fino alla verifica del funzionamento dell'applicazione quando queste unità sono integrate, integration testing**).

Ma che differenza c'è nell'utilizzo dell'approccio structured classes (in termini di ruoli e parti) rispetto ad un approccio che faccia uso dei costrutti di aggregation e composition? Vediamolo con un esempio pratico.



Si vuole rappresentare il legame tra la classe item e le classi descrizione e prezzo.
Con il class diagram e composition: si rappresenta come item sia costituito da due oggetti interni, uno che descrive l'elemento e l'altro che fornisce informazioni sul prezzo -> se cancello l'oggetto item cancello anche gli oggetti description e pricing in esso contenuti.
Riguardo la classe strutturata invece metto nella classe item due parti: una che è istanza di description e una che è istanza di pricing.
Perché sono diversi? *Il significato in sé è lo stesso, però la classe strutturata mi fornisce un'informazione ancora più precisa in particolare legata all'associazione tra Description e Pricing: con la classe strutturata sto dicendo che anche l'associazione fa parte della classe strutturata* (linea tra description e pricing), *mentre con il class diagram non è chiaro come viene gestita* (se cancello l'oggetto item l'associazione resta o no?)

Quindi la classe strutturata rappresenta un meccanismo migliore per rappresentare informazioni tipiche di sistemi software di grandi dimensioni, dove gli elementi vengono rappresentati in modo gerarchico. Si garantisce una precisione e quindi pulizia di progettazione maggiore rispetto all'uso di UML 1.

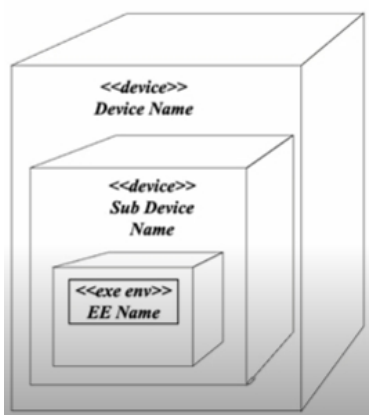
Il passaggio che ci manca e che non era possibile con UML 1 è: ***una volta progettata in dettaglio la mia applicazione via classi strutturate, communication diagrams etc… come si traducono le classi dal punto di vista di componenti fisiche che poi saranno effettivamente mandate in esecuzione su uno dei nodi di esecuzione***?
Come passare dall'insieme di classi strutturate all'eseguibile che contiene l'applicazione che funziona grazie a quelle classi?
In UML 1 il concetto di "componente" descrive un componente fisico, e mancava totalmente il passaggio da classi a livello di progettazione a eseguibile.
Con UML 2 invece, dato che le classi sono descritte in dettaglio tramite classi strutturate e dato che il concetto di "componente" ha subito un cambiamento sostanziale (da descrivere un singolo componente fisico è passato a descrivere un elemento a livello di progettazione) è possibile descrivere il passaggio da progettazione a software effettivo da eseguire.

Questo passaggio viene fatto anzitutto descrivendo la piattaforma di esecuzione sottostante. *La configurazione di piattaforma definisce come le funzionalità del software possono essere distribuite sui vari nodi fisici dove viene eseguito il sistema software distribuito.*
Questo viene ottenuto attraverso **due passi**:
- **si definisce la piattaforma hardware sottostante attraverso il Deployment Diagram in UML**
- **si descrive come sui nodi fisici di esecuzione vengono allocati i componenti software ricavati durante la fase di progettazione**.

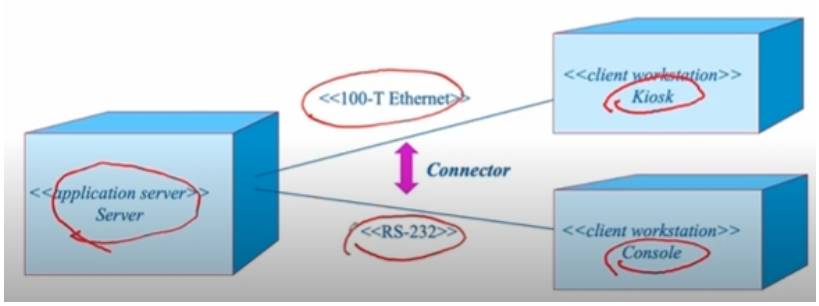
Questi componenti software non si chiamano più "components" come in UML 1, ma **artefatti**.

Per quel che riguarda il deployment diagram ***ogni nodo di esecuzione è rappresentato da un parallelogramma***. Le *connessioni sono rappresentate mediante archi che collegano i nodi*, anche in questo caso ampia libertà sul significato: ***meccanismo di comunicazione, mezzo fisico o protocollo software***.

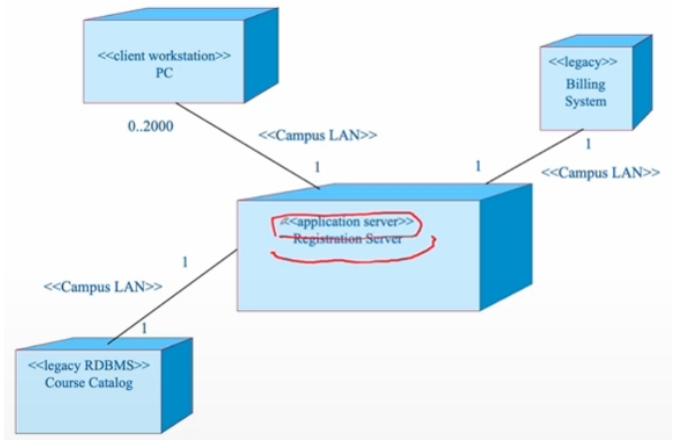
Come detto quindi un nodo rappresenta una risorsa computazionale che ha tipicamente capacità di elaborazione e memoria:
Ne esistono **due tipologie principali**: **Nodi Device** *se rappresentano proprio la risorsa fisica con capacità di processing* o **Execution environment** *se rappresenta particolari piattaforme di esecuzione allocate nel nodo.*



Rigurado i Connector invece rappresenta una connessione tra due nodi delle tipologie citate sopra. Nell'esempio qui sotto a sinistra l'application server rappresenta un execution environment, collegato a una workstation (chiosco) via ethernet (mezzo fisico) mentre console (terminale) collegato direttamente al server attraverso connessione seriale di tipo RS-232.



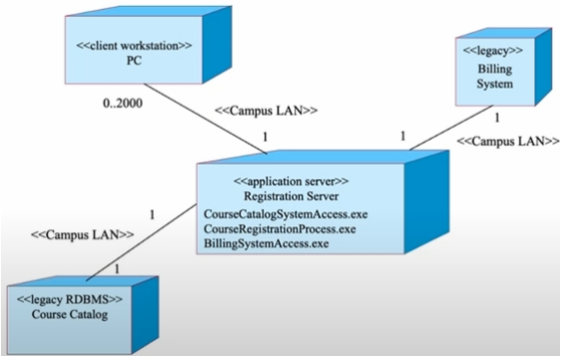
Essendovi nel deployment diagram grande libertà di rappresentazione, è fondamentale come si vede dall'immagine utilizzare gli stereotipi (es. <<100 T Ethernet>>) che permettono di annotare sia nodi che connettori descrivendo le informazioni necessarie a comprenderne il significato.



Esempio di deployement diagram: al centro un application server (lo raggiungo collegandomi a un indirizzo web) il cui accesso può avvenire da parte degli studenti tramite il proprio PC usando ad esempio come mezzo la LAN d'ateneo.
Il Registration Server è utile agli studenti per registrarsi al nuovo anno di uni, quindi si connette sempre via Campus LAN al server un altro nodo di elaborazione che

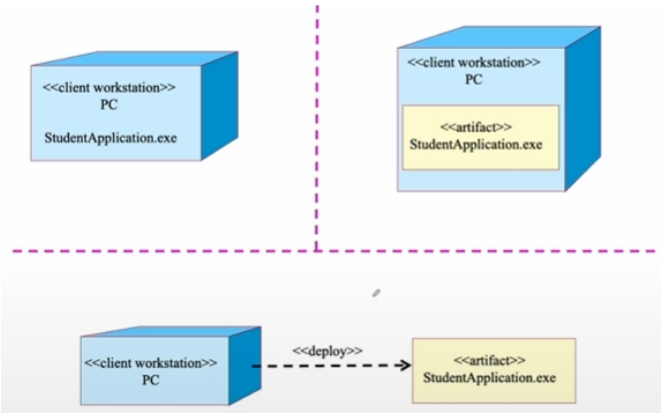
rappresenta il Billing System, ossia il sistema contabile che l'università usa per gestire i pagamenti. Vale lo stesso per il Course Catalog che contiene l'insieme di corsi di modo che lo studente possa fare il piano di studi, entrambi i nodi sono legacy (preesistenti nell'università e collegati tramite rete locale al sistema utilizzato dagli studenti per iscriversi).
In questo caso oltre alle connessioni (che in questo caso no danno molte informazioni in termini di connessione specifica) si utilizzano anche le molteplicità per descrivere quanti nodi partecipano a una connessione nel deployment diagram (fino a 2k studenti contemporaneamente).

L'esempio descrive la situazione di un deployment diagram una volta sviluppata l'applicazione (il Registration Server) e come dovrà relazionarsi ad altri nodi nella rete distribuita.
Quello che ci manca è capire come collegare il Registration Server a quanto ho progettato in fase di progettazione, ossia come inserire le classi strutturate e i vari meccanismi affinché l'applicazione possa funzionare? (questo mancava in UML 1)
Ciò fa riferimento al **Process-to-Node Allocation**, ossia come assegnare i vari processi ai dispositivi hardware in esecuzione. Per farlo si tiene conto di vari aspetti:
- **Pattern di Distribuzione** (il carico deve essere distribuito adeguatamente di modo da evitare colli di bottiglia)
- **Si vuole trovare un'allocazione che minimizzi i tempi di risposta e aumenti throughput**
- **Minimizzazione del traffico attraverso la rete** (si vorrebbe che processi comunicanti spesso tra loro siano sullo stesso nodo o su nodi vicini per minimizzare traffico)
- **In base alla capacità dei nodi** (CPU, RAM, spazio)
- **In base alla larghezza di banda del mezzo di comunicazione**
- **In base all'availability dei nodi e connessioni** (se connessioni instabili ne devo tener conto per l'allocazione dei processi)
- **In base ai Rerouting Requirements** (se un nodo fallisce o una connessione si interrompe, il sistema deve poter riallocare i processi)



Si inseriscono quindi i processi nei vari nodi di esecuzione (es. il processo che permette la registrazione e l'accesso al catalogo corsi e al sistema di billing garantendo interoperabilità)

Ma cosa metto nei vari processi (che sono entità logiche-funzionali) rispetto a quanto realizzato in fase di progettazione?
Di questi aspetti se ne occupa l'attività di **Deployment**.
Rappresenta ciò che mi permette di assegnare/mappare gli artefatti software sui nodi fisici durante l'esecuzione.
L'artefatto software rappresenta quindi l'entità su cui può essere fatto il deployment verso il nodo fisico, ***e questi artefatti modellano le entità fisiche che in UML 1 erano rappresentate attraverso il costrutto componente*** (quindi componente rimpiazzata in UML 2 da artifact). **Tra gli artefatti si hanno quindi file, eseguibili, tabelle database, pagine web etc…**

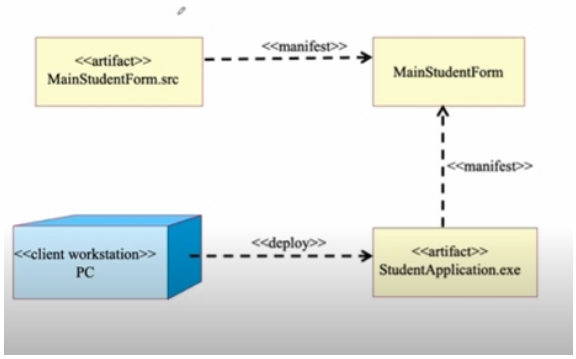


NB quindi gli artefatti == componenti in UML 1 (eseguibili, componenti fisiche)

Qui sopra un esempio di come gli artefatti possono essere disposti (deployment) sui nodi, tre modi.
Nel primo, come visto nell'esempio prima, scrivo il nome dell'artefatto nel nodo.

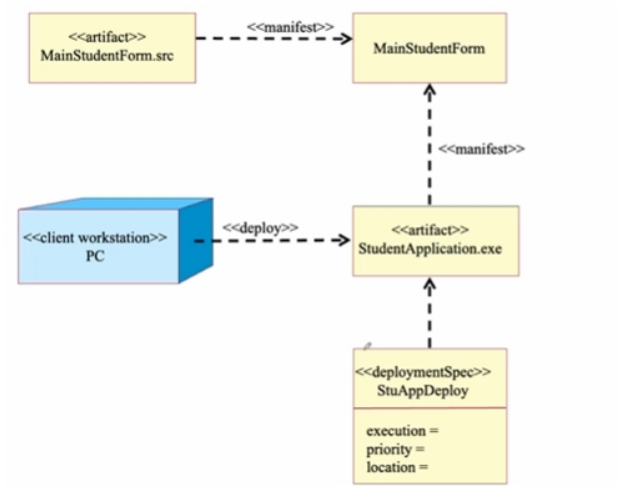
Nell'altro si usa il rettangolo con stereotipo <<artifact>> e nell'ultimo, in modo ancora più esplicito, usando la freccia tratteggiata (associazione generica UML) che assume significato grazie allo stereotipo <<deploy>> per cui l'artefatto deve essere disposto nel nodo di esecuzione PC.

Ma quanto definito in fase di progettazione in quali artefatti finisce?
UML 2 permette di definire quest'informazione tramite il concetto di **Manifestazione**.
La Manifestazione è una relazione tra un elemento di un modello e il relativo artifact che implementa il modello.



L'artefatto MainStudentForm.src è codice sorgente manifestazione della classe (elemento di modello) MainStudenForm, ciò viene ancora una volta descritto tramite associazione generica con relativo stereotipo.
Una volta compilato il codice sorgente diventa eseguibile, che a sua volta è manifestazione di MainStudentForm. Infine il deployment avviene per quel che riguarda proprio questo eseguibile.
(in UML 1 non era possibile definire il passaggio da elemento di modello a corrispettiva implementazione!)

Questi legami di deployment possono anche essere arricchiti di informazioni per specificarne meglio il significato attraverso il **Deployment Specification**.



Oggetto <<DeploymentSpec>> che permette di specificare informazioni del tipo **come eseguire l'artefatto**, **dove eseguirlo** e **la priorità**.

L'utilizzo della deployment specification e di questi parametri è utile oltre che per arricchire il modello di informazioni utili anche per sistemi automatici che permettono automaticamente di effettuare il deployment su nodi di elaborazione.

Con ciò abbiamo concluso la parte legata alla fase di progettazione Object Oriented.