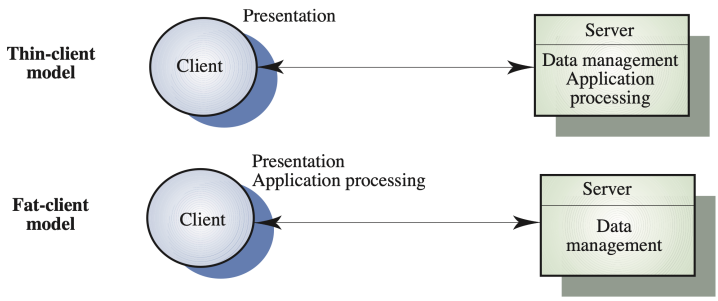
- ***Lezione 22***

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).

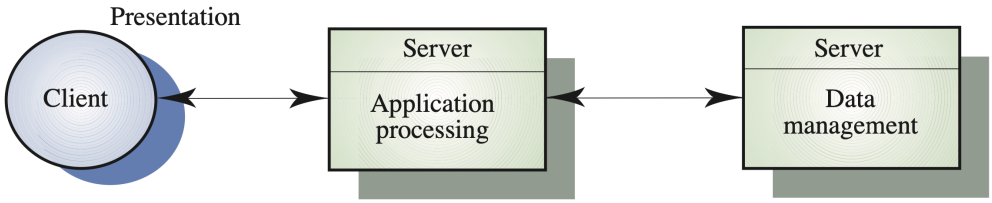
Nasce a questo punto il seguente problema, i vari Application Layers su quale tipo di processi (client o server) devono essere allocati?
Questo ha portato all'introduzione di diversi sottotipi di architetture C/S, tra cui quelle che seguono:

- 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.



- Three-Tier (un architettura Two-Tier evoluta nel tempo), 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.



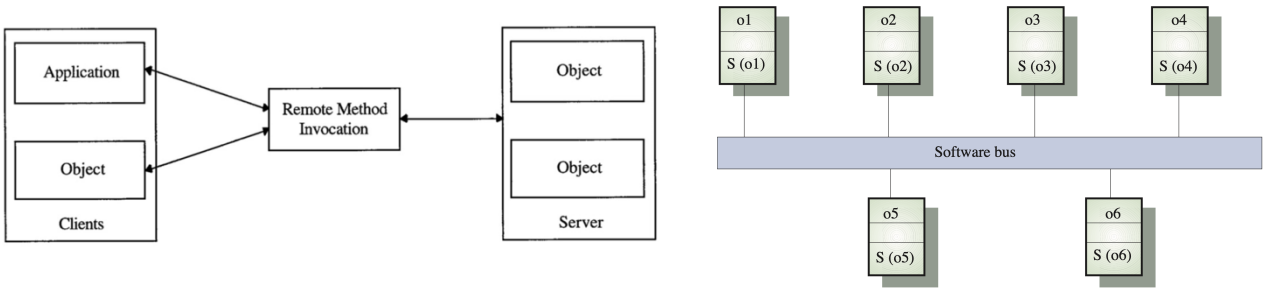
L'uso di questo approccio offre migliori performance rispetto al two-tier thin-client ed è più semplice da gestire rispetto al two-tier fat-client. Inoltre è un'architettura più scalabile della two-tier in quanto se si necessità di maggiori informazioni allora più server possono essere aggiunti.

4. Distributed objects architectures - l'Architettura a Oggetti Distribuiti

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.

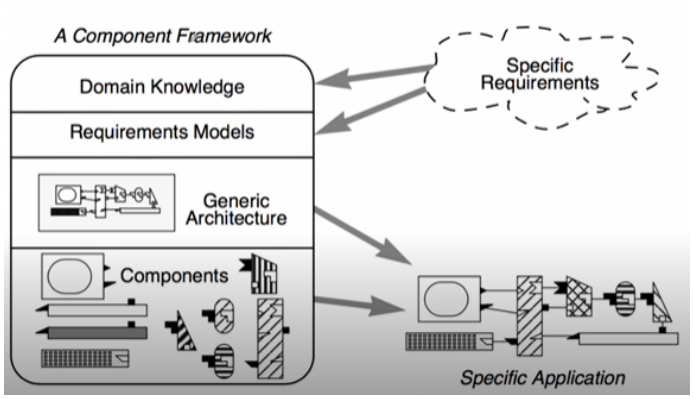


ORB come “software bus” che permette di ampliare il paradigma a oggetti in un ambiente distribuito.
A valle delle architetture a oggetti distribuiti sono state introdotte architetture che cambiano anche il modello di “business” dietro allo sviluppo software (cambierà ulteriormente con le architetture service oriented), stiamo parlando delle architetture

5. Component-based architectures – Architetture basate su componenti

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.

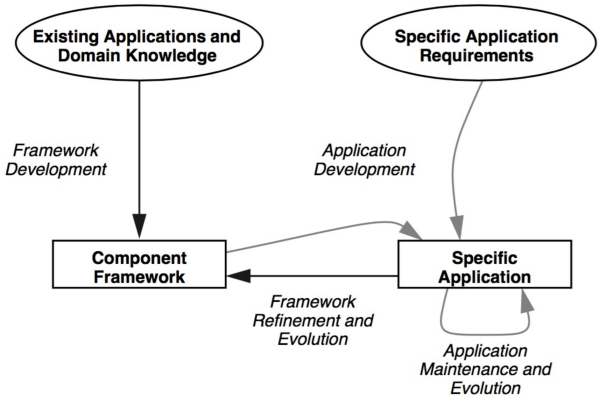
Aspetti importanti delle componenti software sono quindi:
- capacità di incapsulare strutture software in queste componenti astratte (variabilità, cambiano comportamento in base a come sono utilizzate)
- possibilità di “assemblare” queste componenti collegandoli attraverso l’interfaccia e lo scambio 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 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.

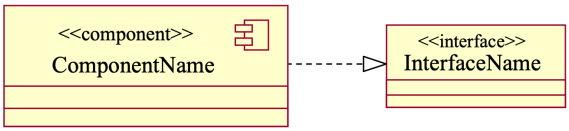
Quindi si fa tipicamente quanto segue: ho dei requisiti specifici per il mio dominio, vedo se vi sono somiglianze con i requisiti generici di un certo framework e se ve ne sono abbastanza allora implemento le componenti mancanti e il framework potrà permettermi di costruire l’applicazione per soddisfare 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: un'implementazione di cui posso fare l'allocazione in una certa piattaforma di esecuzione (es. eseguibile, documento word etc..).
  Tuttavia ciò comportava un problema: non vi era rappresentazione di come si arrivasse a queste componenti a partire ad esempio dalle classi
- 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.



OO Principles: Encapsulation and Modularity

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 inoltre, per superare il gap semantico tra gli elementi a livello di progettazione e livello di esecuzione, il concetto di classe è stato rimpiazzato dal concetto di classe strutturata (per far capire come le classi che progetto sono tradotte in oggetto eseguibile).
Ciò è possibile anche all'introduzione del concetto di sottosistema, per cui il componente è in realtà progettato come sottosistema e i dettagli del sottosistema sono progettati facendo

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

6. Service-oriented architectures

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 model 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 risposta.
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. I principi di questa architettura sono:

- 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

- 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)