Project Management: Cosa sperimenteremo?

- **Project Reviews:** Dallo stato attuale dl progetto al quello che ancora dobbiamo fare;
- **Hiring Process:** Procedimento per assumere qualcuno;
- **Project meeting:** Incontri tra chi fa il progetto per chiarire dubbi e fare il punto della situazione;
- **Performance Review:** Si analizza il lavoro della persona a fine anno, e si danno feedback costruttivi;
- **Laboratory presentation:** Si espone lo stato del lavoro, sia in formato tecnico per i manager, sia in formato economico per il cliente;

L' **Obbiettivo** del project Management è gestire un progetto e le persone che ci lavorano. Si usano software per gestire l'andamento del progetto, come **Trello** o **TeamGantt** (Costruzione di diagrammi con task e assegnazioni). In un progetto ognuno ha il suo ruolo, è quindi richiesta presenta e partecipazione. Ma perché serve tutta questa organizzazione? Perché i progetti sono in ritardo nel 75% delle volte, a causa di pianificazione, poco monitoraggio o cattiva comunicazione tra i gruppi. Le soluzioni sono molteplici: riutilizzo del codice e dato che ci sono scadenze simili per progetti simili, riandare a utilizzare cosa già sperimentate in passato. Inoltre, per una buona previsione, è necessario usare gli **Spike**, ovvero delle demo funzionanti della tecnologia che ci serve, da cui poi partirà l'intero progetto (formato da più spike).
In breve, il Project Managment è **Definire, Pianificare, Schedulare e Controllare** spesso ogni 2 settimane. Per capire questo metodo di lavoro, è necessario sapere cosa si deve fare e saper affrontare i cambiamenti.

Come condurre un progetto software? Anticipare i problemi e costruire un piano ben fatto è la base. Un planning ben fatto aiuta ad organizzare, a dirigere e a controllare il lavoro. Utile anche costruire delle **Roadmap**, ovvero stabilire quali funzionalità si avranno nella prossima versione.

Gestione delle persone: La versione americana è quella di pagarti per fare quello in cui sei bravo, facendo carriera anche come sviluppatore. In Europa più sei bravo e più Sali, anche se per salire cambi lavoro (non farai più lo sviluppatore ma gestirai altri sviluppatori).
Integration Testing: Le varie parti si devono parlare tra loro. Bisogna quindi testare tutti i componenti/moduli di tutto il progetto.
Extreme programming: E' la metodologia più agile di tutte.Poca documentazione, subito sul codice. La documentazione prevede le **User Stories,** ovvero una descrizione di ogni tipo di utente che userà il mio sistema. A queste vanno aggiunti gli **User Cases,** ovvero tutti gli scenari che i vari tipi di user potranno affrontare. In questo modo abbiamo i **Requisiti** per ogni tipo di utente. Unendo questi agli spike, abbiamo un project plan, che verrà iterato più volte fino alla versione finale, cioè la prima release (Stories → Cases → Spike → loop) → Prima release.

Alla documentazione vanno aggiunti analisi e design, che hanno significati molti diversi:

- **Analisi:** Alto livello, serve a sgrossare i problemi iniziali, individuando, dai requisiti, le varie parti/classi/componenti, del progetto, in maniera rapida. E' come un UML senza signature più veloce e senza classi di servizio. Solo classi grosse. Si esegue creando le **CRC Cards,** ovvero classi con responsabilità (funzionalità) e collaborazioni (link con altre classi. Se A raggiunge B, collaborazione di A sarà B, ma non vale il viceversa) esplicite. Si creano per rappresentare le user stories e dare un'idea del moke-up iniziale. Ovviamente questa fase è transitoria e non finale, perché sia l'utente che i programmatori possono cambiare idee e piani.
- **Design**: Classi diagram, UML + Sequence diagram + Activity Diagram … → Basso livello.

Action planning model: Post Mortem Meeting: riunione per capire così ha funzionato e cosa no, riguardando da capo tutti i task.

Coordinazione tra i membri del gruppo, frequente, mirata. Tanti report, a tutti, che tutti devono leggere. Se le richieste cambiano, l'obbiettivo cambia. Gruppo: Tutti devono esprimere un'opinione durante le user stories → Brainstorming per far CRC. Capire se una classe può eseguire le proprie responsabilità da sola o ha bisogno di supporto di un'altra classe.

SOA – Service oriented architecture:

Nelle grandi aziende serve un qualche che dia dei legami eterogenei tra le varie parti dell'architettura. Non possiamo buttare però quello che c'era prima: bisogna trovare un approccio che permetta di aggiornare senza distruggere tutto → SOA lo permette, controllando architettura e struttura dei singoli servizi, in modo che siano riusabili singolarmente. Quindi le SOA si usano in sistemi grandi.

(Sistema datawarehouse: un sistema di dati per il management). Prima delle SOA questo sistema era incasinato e difficilmente riusabile. Per questo si crearono dei Web Service, con cui si richiama solo la funzionalità che ci serve, che ci dia il servizio **singolo** di cui abbiamo bisogno (es: pagamento con credit card). → Inoltre, la business logic è più compatta, e da fuori non riusciamo a distinguere le varie parti (che sono composte dai web service). Un **Web Service** è un servizio condiviso tra tante applicazioni, tramite il quale queste app comunicano. Il web service può cambiare ed essere modificato senza influenzare le applicazioni che lo sfruttano. Amazon infatti si fa pagare utilizzando le **API** del web service Visa. Le API sono esposte in linguaggi particolari. Gli standard sono SOAP, XML, UDDL (Ultimo standard SOA) o WSDL (per le richieste).

Ma come si utilizzano i Web Service? Essi girano su http o reti locali e sono scritti in XML. L'**Uddy registry** contiene gli indirizzi delle API e l'indirizzo del web service. Il consuler richiede un servizio tramite linguaggio SOAP.

SOAP: Posso definirne i metodi. REST: Ottengo dei dati senza definire metodi.

Il protocollo SOAP: Simple Object Access Protocoll. Si basa su XML. Permette l'esecuzione di un servizio offerto da un web service e il richiamo di procedure remote attraverso messaggi SOAP. Utilizza il paradigma della programmazione orientata ad oggetti.

WSDL: Web Service Definition Language: E' l'analogo dell'IDL di CORBA. Permette di comprendere dove inviare la richiesta e come costruirla.

UDDI: Universal Descriptio, Discovery e Integration. Registro nel quale i Web Service si iscrivono in modo tale da essere trovati per chi volesse utilizzarli. Dopo la registrazione, fornisce diverse informazioni relative ad un certo web service.

REST: Representational State Transfer. Stile architetturale per i sistemi distribuiti. REST usa esplicitamente i metodi http, senza altri livelli e non prevedono il concetto di sessione (sono quindi **stateless**). Il funzionamento prevede una struttura degli URL ben definita e l'utilizzo di metodi http specifici per il recupero di informazioni (GET) o per la modifica (POST).

Puoi decidere quale contenuto ritornare, ma i più comuni sono JSON e XML.

REST prevede che la scalabilità del web e la crescita siano diretti risultati di pochi principi chiave di progettazione. Infatti lo stato dell'applicazione e le funzionalità sono divisi in risorse web e ogni risorsa è unica e indirizzabile, usando sintatti universale per uso nei link ipertestuali. Le risorse inoltre sono condivise in modo uniforme per il trasferimento di stato tra client e server, per questo sono molto comuni i formati strutturati come JSON.

Enterprise Maven (maven non compila senza rete, ed è fondamentale se si usano più librerie**):** Costruisce una struttura di progetto che ci rende autonomi, creando una repositori locale con tutto il progetto. E' quindi un tool per costruzione di progetti grossi, costruito con java. Le dipendenze sono gestite dal **POM**, il Project Object Model. La struttura delle cartelle è molto rigida. Nel POM devi dichiarare il JAR file (se le dipendenze dei moduli sono runnate con Java) o il WAR file (se le dipendenze sono runnate su Tomcat o Jelly o altro server).

L'accesso al DB viene gestito dall'**entity manager.** Non si utilizzano le query ma utilizziamo un find tramite il metodo get(). Per salvare qualcosa nel DB è uguale: chiediamo all'entity manager di salvarlo.

Dalla categoria persistent creiamo una persistent unit → questo significa che stiamo creando un ponte tra la business logic della mia applicazione e il DB. Nell'XML indichiamo quale database stiamo usando.

Ogni istanza di entità corrisponde ad una riga nella tabella del DB.

Avendo quindi l'oggetto persistente, creiamo l'oggetto fasad(classe astratta) che ha la funzione di contenere i metodi per fare accesso alle istanze del DB (crea, cancella, modifica…).

Servlet: Dentro la parte web abbiamo le parti html, css, jsp.

Chi utilizza il book manager? Nella parte sorgenti abbiamo il servlet controller.

Non si sa una new di un session bean ma si prende il riferimento, chiesto al container, e quindi bisogna fare una injection della notation, dicendo quindi che in quella classe voglio usare un session bean → manager local sarebbe la mia interfaccia del session bean.

Cloud Computing: Una serie di tecnologie informatiche orientate a fornire attraverso la rete dei servizi, Hardware (società che offrono servizi di calcolo) e Software (necessità di connessione alla rete). Non richiedono la conoscenza da parte dell'utente finale della locazione fisica della macchina.

Posso affittare macchine in remoto in cui mettere sistema operativo e tutti i service sopra. Questo si chiama **Infrastructure as a Service.** In questo modo un'azienda può affittare lo spazio senza comprare nulla di fisico. Finchè non sa che le cose stanno andando bene (pensiamo alle start-up), questa è un'ottima soluzione. Anche Instagram scelse questa.

- **Infrastructure as a Service**: Tutto quello che può essere ospitato da altre parti. Si affittano da un provider di servizi cloud (Macchine virtuali, sistemi operativi…);
- **Platform as a Service**: La differenza con l'infrastructure è che le piattaforme sono già forniti, quindi abbiamo già web service. Questo è detto **Load balance,** cioè se io utilizzo una di queste piattaforme cloud automaticamente i servizi vengono offerti in maniera trasparente (solo quelli che ho richiesto, ovviamente). Sono quindi servizi di cloud che offrono un ambiente per sviluppo, gestione test e distribuzione di applicazioni web;
- **Software as Service**: Distribuzione di applicazioni software tramite internet. Gli utenti si connettono all'applicazione tramite internet, in genere con un web browser.

In Realtà tra questi servizi ci sono dei livelli intermedi: **Data as a service e HD as service,** quindi lo schermino è impreciso.

Amazon e google avevano tanti server e hanno bisogno di tanta capacità di calcolo per rispondere a tutti. Molti di questi server sono metti a disposizione degli utenti, a pagamento.

In sostanza, il cloud computing prevede calcolo, software, accesso ai dati, servizi di storage e non richiede che l'utente finale conosca la locazione fisica e la configurazione del sistema che si occupa di fornire tali servizi.

L'altro punto molto importante per cui i cloud computer sono importanti sono i **mobile**. Se accedo molto spesso al cloud, ho bisogno di un mobile. Molti scaricano anche le mail in locale. Il problema di tutto questo è solo avere la banda larga. È ovvio che se la rete non funziona, l'utilizzo del cloud viene compromesso.

In sostanza i server costano molto e averli in affitto risolvono questo problema.

Entrando più nel dettaglio, **cosa c'è in un sistema cloud**? Possiamo avere inizialmente HW e Firmware a disposizione (risorse virtuali di storage, network e capacità di calcolo). Ovviamente si garantisce la sicurezza dei dati, devono essere accessibili 24/7 e devono gestire guasti velocemente. Se ho un'app bancaria che va giù è un macello (problemi transazionali ecc…). Quindi quando scelgo il mio provider devo esaminare anche la sua affidabilità. In ogni caso si paga a consumo.

Ci sono diverse differenza tra i vari tipi di cloud:

- **Ibrida**: Posso avere una parte privata e una parte pubblica.
- **Privato:** La mail universitaria è istanziata solo il server dell'università di Torino, quindi privato, poiché sotto google. (Cloud = modello computazionale di questo genere. Avere il server sotto il proprio controllo)
- **Pubblico**: Pubblico.

Cause – Effetti.

Il web browser diventa il personal PC. Ad esempio non tutti scaricano le mail in locale ma le leggono direttamente sul browser (una volta si scaricava la posta). Un altro effetto è la riservatezza dei dati → i nostri dati sono sul cloud e vengono utilizzati. ES: se usi google photo lui ti mette a posto da solo la

luminosità. Lo Fanno per raffinare i loro algoritmi per il miglioramento delle immagini. ES: se compro un libro, dopo una settimana mi compare la pubblicità di una cosa analoga → questo perché sanno che ho comprato il libro. Morale: molto spesso i miei dati vengono utilizzati, analizzando le parole chiave nelle mie mail, acquisti, e in molti altri modi.

Cause del cloud (Perché il cloud?): la banda larga è stata inventata proprio per il cloud, oltre che per fornire un miglioramento di accesso ai cloud dai mobili.

Nel back-end ci sono tante applicazioni che possono agganciarsi al cloud per molteplici scopi, come le application server.

Quali sono i player del cloud? Data-Center cioè solo delle macchine, software come XEN e vmware che hanno messo a disposizione macchine virtuali per la virtualizzazione del cloud, sulle quali poi vengono offerti i sistemi operativi. Nella parte piattaforma abbiamo i database, come SQL e i noSQL database. Anche Microsoft ha costruito Azure, cioè la loro piattaforma cloud basata su SQL server offerto nel cloud, in parallelo alle altre piattaforme che possono avere altri linguaggi.

Nella parte di sw application abbiamo i vari software che possono essere installati sulle varie piattaforme che utilizziamo

Aspetti nuovi: Ovviamente l'affitto del cloud ha dei picchi, diamo la possibilità di avere più potenza in quei picchi (potenza computazionale a breve termine): ad esempio le aziende del turismo affitteranno più cloud d'estate. Illusione di infinita potenza computazionale poiché pago per avere di più. Eliminazione di impegno per il cliente → all' inizio le start up affittano.

Origine: Amazon e Google hanno molti data center, quindi ad una certa hanno pensato di affittarli (non troppo importanti.) Quando conviene? Non troppo importante ma ad esempio la storia delle aziende turistiche è uno dei motivi. Inizialmente il cloud è stato utilizzato molto da centri di ricerca, poiché c' era bisogno di moltissima capacità di calcolo durante gli esperiementi, ma questa capacità di calcolo non doveva essere costante (se non ci sono esperimenti da fare, non mi serve).

Ovviamente non tutti lo usano: magari una banca non si fida a mettere tutti i dati in cloud, per problemi di sicurezza.

Problemi? Dati bloccati: nella scelta tra intrastructure as a service e platform as a service, se io uso la seconda io sono vincolato alla piattaforma software che ho scelto di utilizzare. Se io uso una infrastructure, cioè metto io il sw sopra, sistema operativo ecc, sono più libero di cambiare il tipo di software. Bisogna quindi fare attenzione a quello che si decide di utilizzare. Posso scegliere funzionalità maggiori e più precise o posso scegliere il fatto di poter far qualcosa online, in modo da poter utilizzare i dati in ogni momento.

Panoramica sulle principali player del cloud: **Amazon** fornisce sia il livello infrastrutturale sia il livello di piattaforma. Può configurare le macchine a disposizione e il suo livello EC2, cioè il livello piattaforma, è molto dinamico nel senso che permette l'installazione di sistema operativo. Difficile gestire la scalabilità dal punto di vista individuale, ma se uno non ha bisogno di gestirla questa piattaforma è perfetta.

Qual è il modello di sviluppo dl software quando uno utilizza il cloud? Maven. L'idea è che io sviluppo in locale e poi fatto il deploy sul cloud e poi faccio il test in locale e ripeto il test sul cloud. Quindi io programmo etesto in locale, faccio il deploy, e rifaccio il test sul cloud.

Google: Abbiamo SQL e noSQL, il gratuito è noSQL. API infiniti per le varie applicazioni google e tutti i vari strumenti che sono molto famosi. Si ha un SDK per sviluppo locale e abbiamo una web application console.

Lifecicle: Io costruisco, faccio il testing in locale, carico su cloud, gestisco l'applicazione e in preparerò gli update. L'App engine SDK è il software che io mi scarico in locale e mi permette di simulare la mia versione

online, mentre l' app engine admin console è il software che mi permette di gestire le funzionalità che offre la mia applicazione.

Dietro le quinte il web server google platform è un python web server, quindi sponsorizza python (python vs c). Il persistent layer abbiamo data store cluster, che possono contenere tantissimi dati in maniera trasparente e un object database noSQL non relazionale. Quindi possibilità di memorizzare tantissimi dati a basso costo, quindi molte app utilizzate per analisi dati vengono messe qua.

EC2 pura platform, non decidiamo SO, non possiamo fare macchine virtuali ma abbiamo la possibilità di avere tanti linguaggi, cloud storage ma la piattaforma è scelta da loro (contrario di amazon).

Descrizione di Engine google di Wikipedia: è un web framework e una piattaforma di cloud computing per sviluppare e hostare delle applicazioni web, utilizzando i data center della google. Queste applicazioni runnano su server multipli e sono a se stanti. Engine offre scalabilità per le applicazioni web in modo automatico: più c'è domanda, più risorse ti offre. Le piattaforme non sono flessibili ma sono espandibili.

Teoria – l'evoluzione delle tecnologie Software: La struttura architetturale su cui un sistema informativo distribuito si compone di tre livelli:

- **Presentazione:** cioè la parte che permette al sistema di comunicare con entità esterne (persone e macchine), presentando informazioni. Implementato come interfaccia grafica per l'utente o come modulo che formatta un'insieme di dati in una nuova rappresentazione, secondo una cerca grammatica. Questo livello non dov'essere scambiato con i client, che sono entità che utilizzano i servizi offerti dal sistema (a volte anche completamente esterne, come nel caso dei web browser);
- **Logica di business:** Livello in cui si racchiudono tutte le operazioni che implementano le richieste dei client e che ne garantiscono la risposta;
- **Dati:** Dati su cui il sistema opera, sorgenti su cui il sistema opera. Qui oltre ai DBMS, sono da includere i sistemi esterni.

Inizialmente, nelle architetture per aziende le soluzioni enterprise erano le **Host Solution,** grossi PC con tutto dentro (era l'unica soluzione esistente). Vi si accedeva tramite schede. Da qui si è passato ai **mini-computer**, scatolette più piccole del mainfram, con cui si accedeva da linea di comando. Successivamente sono nati i **Personal-PC**, strumenti che stavano sulla scrivania utente (anni 80). Per un breve periodo ci sono stati i **PC-stand alone**, PC che avevano tutto ma non condiviso in rete. Su ogni PC andava installato tutto. Questo generava problemi di reti (troppi si connettevano), quindi per risolvere il problema sono stati inventati i **Networking-PC**, PC connessi da una rete locale, che permetteva di risparmiare tempo su manutenzione, in quanto si installava un solo software uguale su tutti i pc con l'uso della rete. Alla fine degli anni 90 era un must avere un **WorkStation** sulla propria scrivania, cioè una stazione di lavoro in rete, con grosso schermo e sistema operativo.

Al tempo del mainframe era tutto unito (business logic, model e altro). Il primo passo per risolvere questo su mettere la presentazione sul client e il resto sul server (con cui si accedeva tramite client).

Classificazione delle soluzioni Cliente/Server:

Inizialmente, nel main frame la client presentation (PRESENTATION) era separata dalla business logic (LOGIC), che era separata dai dati (DATA). Non esisteva il concetto di separazione dei dati. Con il client/server ci sono state delle modifiche sull'allocazione delle varie componenti.

- **Time Sharing:** Server (mainframe) ha tutto. Ci si accedeva da un terminale accedibile da tastiera e la risposta avveniva come stringa di caratteri; Non era ancora chiara la divisione in tre livelli. Eseguivano spesso programmi scritti in cobol.
- **X-terminal**: Sistema a finestre (presentation a finestre). Si interagiva con mouse, e quindi aveva la presentazione sul client, separata da logica e dati che erano sul server.
- **Distributed App:** Anni 80. In cui nascono i personal computer. Il client ha una capacità computazionale più alta e quindi si sposta una parte della logica applicativa sul client. Si tengono dati in locale poiché al tempo la rete non era ottima (sincronizzazione dei dati lenta). Un esempio sono i cellulari: ha una presentazione sul client ma i dati sono tutti sui server, così come la logica (I Database sul client praticamente non esistono!). Il problema è che se ho più sedi, ho problemi di sincronizzazione dei dati, per questo molti preferiscono il Central DB;
- **Central DB (Fat Client):** Si sposta tutta la logica dei dati sul client (con la presentazione), pur mantenendo i dati sul server. Dato che il client ha tutto, si parla di "fat client". Si ha un Database Management System sul Server e sul client solo un'applicazione con driver che permettono di collegarsi al DB sul Server.
  Il problema è che è molto pesante (richieste ai dati molto frequenti). Scalabilità difficile (l'accesso al DB centralizzato crea problemi se tentano di connettersi 1000 user, quindi serve una gestione di accesso contemporaneo, che è molto costosa). Inoltre aggiornare tutti i pc che si devono collegare

al server è costoso. Per risolvere questi problemi (Problemi del "Fat Client") si è creata la **Stored Procedure;**

- **Stored Procedure:** Sono modi per compattare il numero di operazioni SQL dentro ad un DB, in modo da farne più insieme (Ne eseguo più insieme contemporaneamente). Il problema è che non è portabile, poiché l'SQL è diverso tra i vari DBMS.
- **Distributed DB:** Dati sul client e sul remoto. Se sono un'azienda con più sedi, ho dati replicati sulle aziende secondarie. Dunque ho i dati primari nella sede principale. Ho un problema di sincronizzazione dei dati! Da questo problema scaturiscono altri problemi, come quelli delle transazioni e quindi di sicurezza… ecc. Usato molto meno per questo motivo, ma ottimo se si hanno pochi database da sincronizzare.

La distributed App e la central DB si sono alternate per molti anni.

Middleware: sono nati a se stanti ma hanno molte caratteristiche e funzionalità delle tecnologie usate. E'un software di sistema che permette l'interazione a livello applicativo fra programmi in un ambiente distribuito. E' la soluzione architetturale al problema di integrare una collezione di servizi e applicazioni sotto un'interfaccia di servizi comune. Offrono astrazioni a livello di programmazione che nascondono parte della complessità nella costruzione di un'applicazione distribuita. In questo modo, il programmatore non deve occuparsi di tutti gli aspetti di cui occorre tenere conto quando s'implementa un'applicazione distribuita, infatti è il middleware che se ne occupa.

Per **applicazione distribuita** s'intende un'applicazione le cui componenti non risiedono t utte sulla stessa macchina, e possono anche non essere scritte in un linguaggio comune, per cui occorre poterle mettere in comunicazione e gestire tale comunicazione.

TP Monitor: Sono una tipologia di middleware. Aiutano il programmatore a gestire le transazioni da codice. A volte è necessario sapere se tutti i destinatari sono pronti a ricevere i dati per la sincronizzazione. Il TP monitor automatizza la chiusura delle connessioni al DB, infatti riceve le richieste dei client e apre la connessione solo quando il Client che ha bisogno. Lo svantaggio è che i linguaggi sono privati e quindi non vi è uno standard.

Sono stati utili in passato anche per altri motivi: i sistemi che avevano tanti client e poi server avevano bisogno del bilanciamento del carico, che era molto costoso. I TP monitor erano la soluzione.

In sostanza, i TP-monitor offrono vari servizi:

- Gestione dei processi server: attivazione, funneling, monitoraggio e bilanciamento del carico;
- Gestione delle transazioni: Il middleware gestire le transazioni rispettando le proprietà ACID (atomicità, coerenza, isolamento, durabilità).
- Gestione della comunicazione: Grazie al protocollo 2PC, si ha una gestione delle transazioni in ambiente distribuito.

Funneling: l'idea è che tanti Client si collegano ad un server. Se permetto a tutte le connessioni di essere aperte, il sistema operativo non riesce a gestirle e va in crash. Il TP monitor apre solo le connessioni dei Client che devono effettivamente eseguire. Ovviamente, nel web reale, quando un Server ha troppe richieste chiede aiuto ad un altro server (per il load balancing), ma anche questo ha un costo, oltre ad un altro server.

Quando più utenti accedono ai sistemi, si devono affrontare i problemi di tutti. Inizialmente, le aziende avevano problemi di connessione (troppi utenti). Nel tempo si sono evolute in grandi web application. Questo ha ridotto i problemi di connessione, ma ne ha creato altri: sicurezza, shared data… e questi problemi cambiano continuamente!

ORB: Middleware che permette di creare sistemi distribuiti ad oggetti. Sono gli **Object Request Broker**, che estendono i middleware verso il paradigma ad oggetti, semplificando lo sviluppo di applicazioni distribuite. Supportano inoltre l'interoperabilità tra oggetti remoti che usano processi diversi, scritti in linguaggi diversi. L'importante è che tutti i diversi linguaggi seguano il paradigma ad oggetti.

CORBA: Common Object Request Broker Architecture. E' uno standard che definisce l'architettura e la specifica per la creazione e gestione di applicazioni che seguono il paradigma ad oggetti o per applicazioni distribuite in rete. Corba è una soluzione middleware che consente uno scambio d'informazioni indipendente dalla piattaforma hardware, dai linguaggi di programmazione o dal sistema operativo su cui i processi girano. CORBA offre una specifica standardizzata per ORB, dove ORB fornisce il meccanismo per permettere agli oggetti distribuiti di comunicare tra loro, indipendentemente dal loro linguaggio. E' usato ancora oggi ma trasparente al programmatore, in quanto è nascosto sotto i grossi ambienti come .Net o J2EE. *Come funziona?* Il client invia la richiesta di esecuzione di un metodo di un oggetto che offre il servizio richiesto e che vive in un diverso processo, vivo su un'altra macchina, implementato in un linguaggio diverso. Vi è un'interfaccia (Interface Definition Language) che permette la comunicazione in un linguaggio comune. Successivamente il linguaggio del processo viene tradotto al linguaggio del client. Tra i due vi è il Broker, che collega il client proxy al server proxy. Gli attori quindi sono:

- Il service: implementa il servizio che non è accessibile direttamente o in remoto;
- Il proxy: rappresenta il Service e ne assicura l'accesso in maniera corretta;
- Il client: usa il proxy per accedere al service;

Broker: Quando un Client ha bisogno di parametri per mandare un messaggio al server, deve sapere dove si trova (oltre ad avere i parametri). Il broker serve a questo.

MOM: Message Oriented Middleware. I MOM nascono con l'obbiettivo di integrare nei middleware la g estione dei messaggi asincroni: tra applicazioni distribuite si vuole poter scambiare e gestire messaggi asincroni, messaggi per cui chi manca il messaggio non resta in attesa della risposta di chi riceve il messaggio. Per la gestione delle richieste si utilizzano le code ed è il middleware che si occupa della loro gestione. 2 tipi di paradigma:

- **Fire and Forget**: Il Client non attende il messaggio (non ha bisogno di ACK) ma viene avvertito se qualcosa va male. Nel frattempo, il Client può fare altro. Qui la comunicazione è 1 a 1. (Client ←→ Message processor).
- **Publish and subscribe**: Il Server non sa a chi inviare i messaggi. Li mette quindi in una coda. Il Client, per recuperare il messaggio, dovrà essersi precedentemente registrato. (Publisher ←→ Event channel ←→ Subscriber)

OTM: Object Transaction Monitor: Nelle infrastrutture a oggetti distribuiti, le tematiche di gestione di transazioni sono affrontate con gli OTM, che integrano le lofiche di gestione di oggetti proprie degli object broker con i meccanismi di gestione transazioni proprie dei monitor.

Ma perché questo approccio ad oggetti? Rende automatici alcuni principi di buona programmazione, come modularità e incapsulamento. Favorisce il riuso del Software. Semplifica la manutenzione del software. Possiede una buona base metodologica (Use case, UML…). Favorisce lo sviluppo a componenti.

Microservizi: Dagli anni '90 il modello multi-strato (multi-tier architecture) è stato considerato un pattern architetturale fondamentale per costruire un sistema software. Secondo tale modello le varie funzionalità software sono logicamente separate su più strati che comunicano tra di loro. Ogni strato comunica con gli strati adiacenti in modo diretto richiedendo ed offrendo servizi. In effetti in questa architettura il sistema software, sia pure se logicamente suddiviso in strati, risulta essere un unico sistema monolitico.
Il cloud computing e l'organizzazione agile delle aziende in team di sviluppo piccoli ed autonomi (3-7 persone) sono il contesto in cui è emerso il modello dell'architettura a microservizi.

Che cosa sono i microservizi? In breve i microservizi sono dei servizi "piccoli" ed autonomi che interagiscono tra di loro e che hanno come finalità quella di fare una cosa e di farla bene; sono a tutti gli effetti dei sistemi distribuiti. Per dare una definizione più precisa: *Lo stile architetturale a microservizi è un approccio allo sviluppo di una singola applicazione come insieme di piccoli servizi, ciascuno dei quali viene eseguito da un proprio processo e comunica con un meccanismo snello, spesso una HTTP API.*

Ma quanto devono essere piccoli i microservizi? Un microservizio deve essere tale da poter essere riscritto in due settimane. Teniamo sempre in mente che man mano che la dimensione di un servizio decresce, aumentano i benefici relativi all'indipendenza tra le parti, ma cresce anche la complessità di gestire un numero elevato di parti.

Autonomia: Ogni microservizio si propone all'esterno come una black-box, infatti espone solo un Application Programming Interface (API), astraendo rispetto al dettaglio di come le funzionalità sono implementate e dallo specifico linguaggio o tecnologia utilizzati. Ciò mira a far si che il cambiamento di ciascun microservizio non abbia impatto sugli altri.

Vantaggi:

- Velocizzare i tempi di rilascio del software e reagire velocemente alle esigenze del mercato, infatti ogni singolo servizio è autonomo rispetto agli altri, di coseguenza può raggiungere l'ambiente di produzione in modo indipendente dagli altri, senza che tale attività abbia effetti drammatici sul resto del sistema. Disporre di un processo di deployment snello e veloce consente di poter aggiungere o modificare funzionalità di un sistema software in modo efficace ed efficiente, rispondendo alle necessità di mercato e utenti sempre più esigenti.
- Sperimentare più facilmente nuove tecnologie, infatti la principale barriera per adottare una nuova tecnologia risiede nel rischio associato all'utilizzo di qualcosa di nuovo e con il quale si ha poca esperienza. Confinando questo rischio ad una piccola porzione di un sistema software, che è possibile riscrivere in appena due settimane di lavoro, il rischio risulta molto contenuto e quindi è una sfida da accettare.
- Migliori performance grazie all'utilizzo di tecnologie ad hoc, infatti l'utilizzo di linguaggi e tecnologie eterogenee consente di poter utilizzare gli stack più performanti per implementare specifiche funzionalità: ad esempio è possibile introdurre una particolare tipologia di base dati che risulta naturale per mappare un determinato dato, oppure eseguire un calcolo in un modo particolarmente efficiente.
- Resilienza, infatti nn un'architettura a microservizi, quando una componente non funziona non è automatico che tutto il sistema software smetta di funzionare. In molti casi è possibile isolare il problema ed intervenire mentre il resto del sistema continua a funzionare, cosa non possibile in un'architettura monolitica. Va però sottolineato che l'architettura a microservizi, essendo un insieme di sistemi distribuiti, espone ad una nuova fonte di problemi legati ai disservizi di rete.
- Scalabilità, infatti risulta molto più semplice ed economico scalare un microservizio rispetto ad un sistema software monolitico di grandi dimensioni. Il modello a microservizi consente di poter effettuare provisioning delle parti del sistema software in modo dinamico ed intelligente.
- Facilità di deployment, infatti modificare poche righe di codice su un sistema software monolitico di grandi dimensioni ed effettuarne il deploy è generalmente un'attività non banale, che espone a rischi significativi considerando anche l'impatto che tali modifiche possono avere. Questa paura generalmente porta a raccogliere un certo numero di modifiche prima di avviare un'attività così onerosa e rischiosa.
  Con l'approccio a microservizi ogni singolo servizio può raggiungere l'ambiente di produzione in modo

indipendente, sicché se si verifica un problema esso è facilmente isolato e possono essere intraprese azioni di rollback più velocemente.

- Componibilità, infatti tra le opportunità più interessanti dell'architettura a microservizi vi è la possibilità di riusare le funzionalità. Infatti è possibile che uno stesso servizio venga utilizzato in modi differenti e per scopi diversi. Si pensi ad esempio ad un sistema software che deve poter dialogare non solo col mondo web ma anche con applicazioni mobile, dispositivi wearable, etc.
- Sostituibilità: Quando un sistema software è organizzato a microservizi, il costo di sostituire un servizio con un altro più efficiente e migliore è limitato a circa due settimane di sviluppo, così come banale è il costo di rimuovere un servizio inutile.

Svantaggi: Utilizzare tanti servizi che dialogano tra di loro attraverso la rete, vuol dire di fatto avere a che fare con un sistema distribuito, con tutti i problemi dal caso.
Si pensi, ad esempio, a cosa vuol dire autenticare un utente su un'architettura distribuita volendo garantire il single sign-on, oppure a come gestire la mutua autenticazione tra i servizi che compongono il sistema software. E ancora: cosa vuol dire **testare** un sistema software di questo tipo?