APPUNTI DEL CORSO DI SVILUPPO SOFTWARE PER COMPONENTI E SERVIZI WEB

Prof.ssa Giovanna Petrone

Università degli Studi di Torino
Dipartimento di Informatica
Corso di laurea magistrale in Informatica
A.A. 2015/2016

Stefano Tedeschi

PROJECT MANAGEMENT

INTRODUZIONE

Perché il project management?

Per evitare due problematiche molto comuni:

1. I progetti software sono al 75% in ritardo
    - Cattiva pianificazione
    - Poco monitoraggio
    - Cattiva comunicazione tra gruppi e persone
2. Molti sprechi nell'organizzazione
    - Troppi livelli gerarchici
    - Poca motivazione
    - Attività duplicate

Comunicazione fondamentale

È necessario allo stesso tempo:

- Gestire un progetto
- Gestire un gruppo di sviluppatori

Che cos'è il project management?

Insieme di attività che servono a:

- Definire → Capire cosa si deve sviluppare (requisiti)
- Pianificare → Stimare i tempi di realizzazione
- Schedulare → Sapere quando si completeranno i task, attraverso i **diagrammi di Gantt**, una volta lo scheduling veniva fatto tutto all'inizio dello sviluppo, ora schedulazioni più ridotte
- Controllare → Monitorare e aggiustare i piani perché le cose non funzionano mai esattamente come pianificate

i **task** necessari per raggiungere gli obiettivi del progetto

Progetto → Sforzo delimitato nel tempo (ha una data di inizio e una di fine) volto a crerare prodotti/servizi che portano valore aggiunto al cliente

Raggiungere gli obiettivi del progetto rimanendo nei vincoli definiti dal committente (costo, tempo e qualità)

Ottimizzare l'allocazione delle risorse

L'attività di project management include:

- Distribuzione delle risorse sulle attività
- Capacità di delegare e motivare
- Valutazione dei costi

PIANIFICAZIONE

Pianificare i task è impostante perché:

- Permette di anticipare i problemi
- Permette di stabilire obiettivi concreti e senso di mission
- Requisito per organizzare, dirigere e controllare
- Caratteristica di un buon manager

Fasi della pianificazione

1. Definizione degli obiettivi e priorità
2. Definizione dei task → Importante trovare la giusta granularità
3. Calcolo delle risorse umane → Dipende dalla metodologia di sviluppo scelta
4. Calcolo dei materiali e delle risorse

5. Procedure per l'implementazione → Come verrà messo in atto il piano
    - **Fallpack plan** → Alternative da mettere in atto se qualcosa va storto
6. Schedule → Introduzione di limiti temporali ai vari task
7. Stima dei costi → Sia monetari che temporali
8. Follow-up e review → Si definiscono i meccanismi di controllo e review

Barriere al planning

- Spesso considerato noioso
- Dipende dall'attitudine e dall'ego del manager
- Richiede tempo e sforzo
- È un'attività intellettuale e di astrazione
- Richiede una chiarezza degli obiettivi
- Spesso ll'informazione è inadeguata
- Forze esterne

Fallimento del planning

Il planning può fallire per diverse ragioni:

- Non tutte le persone coinvolte sono considerate
- Non vengono offerte alternative in caso il piano non venga adottato
- Non si anticipano i cambiamenti
- Non si accettano i rischi calcolati
- Priorità errate sul lungo periodo
- Piano non comunicato adeguatamente
- Mancanza di misure per verificare l'andamento del piano

Regole per un buon planning

- Definire priorità
- Prepararsi a risolvere conflitti
- Scrivere i piani
- Modificare i piani attraverso un meccanismo di controllo e revisione

GESTIRE UN PROGETTO

Che cos'è un progetto?

Un insieme di task eseguiti in una certa sequenza per raggiungere un goal

Con una data di inizio e una di fine

Con una quantità limitata di risorse

Il successo di un progetto dipende da molti fattori! → Task, risorse, tempo, costi

Il modello delle quattro fasi

Sono le fasi per la realizzazione di un progetto software

1. Definizione del progetto → Breve descrizione degli obiettivi e dei requisiti necessari per il completamento del progetto (personale, competenze, hardware, ...), si fanno delle previsioni a grandi linee basate sull analisi che si sta facendo (**educated guesses**), vengono identificati i fattori limitanti (tempo, budget, ...)
2. Creazione e definizione di un **piano di progetto** (project plan) → **Task schedule**, ovvero un insieme di task con delle tempistiche e messi in una linea temporale → cosa – quando - come
3. Controllo del progetto e aggiornamento del piano → Aggiornare tempi e costi dei task con dati reali per confrontarli con le previsioni, modificare budget e allocazione risorse se un task è in ritardo

4. Chiusura del progetto → Comunicazione di fine progetto e analisi per capire problemi e migliorare in futuro

Per creare un buon project plan l'esperienza è fondamentale → Imparare dai propri errori!

Il piano di progetto

Il piano è viene utilizzato per diversi scopi:

- Comunicazione all'interno dell'organizzazione
- Ottenere l'approvazione del progetto giustificando necessità e tempistiche
- Mostrare al cliente come un servizio o un prodotto verrà consegnato
- Dimostrare la necessità, la disponibilità e l'utilizzo di staff e risorse
- Ordinare il materiale
- Determinare il **cash flow**
- Ottenere il supporto del team
- Predire il carico di lavoro
- Comunicare progresso ed eventuali problemi
- Avere un punto di partenza per la verifica
- Permettere una valutazione successiva e il riutilizzo dell'esperienza in progetti futuri

Cause di fallimento di un progetto

- Definizione incompleta degli obiettivi
- Planning incompleto
- Project manager incompetente LOL
- Risorse inadeguate nel team
- Poca cooperazione tra aree funzionali
- Troppa burocrazia e gerarchia
- Controllo inadeguato del team
    - Task troppo generali → Non si sa cosa si deve fare
    - Task troppo restrittivi → Se i piani sono troppo dettagliati si spreca tempo a controllare che vengano rispettati perfettamente
    - Cattiva tempistica
    - Poca collaborazione
- Controllo inadeguato del manager
    - Trascura i reports
    - Troppo restrittivo → Mette ansia
    - Cattiva tempistica
    - Non si adatta ai cambiamenti → Se l'obiettivo cambia bisogna cambiare adeguatamente il plan
    - Non assicura la cooperazione

CREARE E RAFFINARE UN PROGETTO

Creare e strutturare i task

- Creare una lista di task
- Stimare il tempo per i task
- Struttura breakdown
- Codici di outline personalizzati (Cosa sono i codici di outline? Boh)

Siccome non si tratta di un'analisi dei requisiti, i task devono essere ben dettagliati

Due possibili approcci alla creazione della lista dei task:

- Creare prima le fasi principali e poi suddividere i task in sub-task → Metodo top-down, <u>delegabile</u>
- Individuare tutti i singoli task dettagliati e poi raggrupparli → Brainstorming

Uno dei metodi migliori per specificare i task e le milestones è:

1. Chiedere ad ogni persona (o gruppo) di definire una lista di task e milestones

2. Inserirli in un project tool chiarendo i termini e rendendo consistente il livello di dettaglio

La lista dei task deve essere facile da leggere

Utile collassare o espandere la lista per analizzare i diversi livelli

I task possono essere raggruppati in fasi

Linee guida per specificare i task

- Identificare i task il più precisamente possibile
- Solo task significativi
- Livello di dettaglio adeguato alla quantità di pianificazione e controllo
- Completezza → Includere rapporti, review, attività di coordinazione
- Task identificati con nome + verbo
- Se due task con lo stesso nome compaiono in fasi diverse, specificare il nome della fase in quello del task
- Sfruttare i task usati nel passato, ma con attenzione

Linee guida per specificare le milestones

Milestone → evento entro il quale si deve portare a termine uno o più task, checkpoint

- Mettere in relazione milestone con task → Per sapere quando la milestone viene raggiunta
- Includere milestone che rappresentano eventi fuori dal controllo (ma influenzano lo schedule)
- Milestone identificate con nome + verbo participio passato
- Oltre alle milestone esterne (imposte dal capo/committente), darsi anche delle milestone interne

Stimare il tempo per un task

Bisogna considerare due aspetti:

- Durata del task → Confrontando con il passato, stimando in base alla propria esperienza e chiedendo alla risorsa interessata
- Dipendenze del task → Nove donne non fanno un bambino in un mese...

Analisi di Pert → I task e le tempistiche possono essere stimati secondo tre approcci:

- Stima realistica
- Stima ottimistica
- Stima pessimistica

La scelta di un approccio piuttosto che un altro dipende dalle persone con cui si sta lavorando

Come far accadere i task al momento giusto

- Sequenzializzare i task
- Chiarire le dipendenze tra task e sfruttarle nello scheduling → Quali task devono iniziare o terminare prima di questo task? Ecc ec
- Raffinare lo schedule
- Pianificare i task per date specifiche

Dipendenze tra task

Dipendenza → Connessione logica (link) tra due task (successore e predecessore)

Diversi tipi di dipendenze:

- Finish-to-start
- Start-to-start
- Finish-to-finish
- Start-to-finish

I task possono essere rappresentati tramite:

- **Gantt chart** → Più incentrati sui task stessi, ma vengono rappresentate graficamente le dipendenze e la durata di ogni task
- **Pert chart** → Più incentrati sulle dipendenze

Disponibili tool per il disegno e la validazione

I task possono avere **lag time** oppure **overlap**

Se non sai come siano fatti cerca un esempio su Google!

Unità di misura

Unità di misura → Tempo + effort

Tempo → Da quando inizia il progetto a quando finisce

Effort → Forza lavoro cumulativa per unità di tempo necessaria per portare a termine il progetto

La distribuzione degli effort non sempre è omogenea

LIMITAZIONI DEL PLANNING TRADIZIONALE

Le tre bestie

Incertezza rispetto alle aspettative del cliente e all'evoluzione tecnologica futura

Necessità di separare le situazioni per livelli di certezza:

- Poca incertezza → Planning dettagliato
- Più incertezza → Preparare diverse possibili opzioni

Complessità → Planning importante, ma difficile

PEOPLE MANAGEMENT

L'obiettivo è raggiungere un'alta produttività nelle organizzazioni

Per fare ciò sono necessari:

- Energia e "commitment" tra i collaboratori
- Un team unito
- Riunioni rapide per risolvere i problemi
- Competizione "sana"
- Fiducia nei subordinati

ALCUNI ESEMPI DI MANAGEMENT

Activity-based management

Modo di gestire le attività simile alla gestione tradizionale delle case automobilistiche (es. catena di montaggio)

L'eliminazione del lavoro è l'unico modo permanente per ridurre i costi

MA questo non significa eliminare posti di lavoro → Così meno persone devono fare più lavoro e si riduce la qualità

Lean management

L'obiettivo è eliminare gli sprechi

Cinque principi:

- Value → Il valore è definito dal punto di vista del cliente
- Value stream
- Flow → Le attività devono fluire tra loro senza interruzioni
- Pull → Fare qualcosa solo quando è richiesto da un processo a valle
- Perfection → Continui miglioramenti

IL PROJECT MANAGER

Deve essere un leader (e non solo un capo)

Deve avere alcune caratteristiche fondamentali:

- Challenging the process
  - Cercare sempre opportunità per innovare, crescere, cambiare, migliorare
  - Sperimentare, accettare il rischio, imparare dagli errori
- Ispirare una visione condivisa
  - Visualizzare un futuro migliore
  - Coinvolgere gli altri in visioni comuni (facendo leva su sogni, valori, ...)
- Rendere gli altri capaci di agire
  - Incoraggiare la collaborazione costruendo fiducia
  - Rinforzare gli altri condividendo informazioni, potere e accrescendo la visibilità dei singoli
- Modellare la strada
  - Dare l'esempio
  - Pianificare piccole vittorie che promuovono l'entusiasmo e costruiscono commitment
- Incoraggiare lo spirito
  - Riconoscere i contributi altrui
  - Celebrare i successi del team

LINEE GUIDA PER IL PEOPLE MANAGEMENT

Gli obiettivi devono essere:

- Specifici

- Realistici
- Misurabili
- Con deadline
- Con commitment

Cercare di essere buoni ascoltatori

Active listening

Tecnica di comunicazione utilizzata in ogni aspetto del people management

Spesso verbalizzare i problemi li alleggerisce → Effetto confortante

Migliora i rapporti tra dipendente e responsabile

Aiuta le persone a pensare autonomamente

No gergo e parole giudicanti

Essere diretti, pensando prima di parlare → Argomenti specifici

Come misurare un ascoltatore:

- Permette a chi parla di esprimere i suoi pensieri senza interrompere?
- Cerca di ricordare i fatti importanti?
- Scrive i dettagli più importanti?
- Evita di smontare l'interlocutore perché il discorso è noioso?
- Evita di diventare ostile se le opinioni sono diverse dalle sue?
- Evita distrazioni durante la conversazione?
- Esprime interesse genuino nelle conversazioni degli altri?

Motivare ≠ manipolare

Manipolare → Direzionare in modo scaltro, sleale, per i propri propositi

Motivare → Direzionare il comportamento verso obiettivi comuni attraverso riconoscimento del lavoro, opportunità di crescita, ...

Il manager deve riconoscere e sviluppare le buone caratteristiche presenti nelle persone

Alcuni principi generali di motivazione sono:

- Mantenere e migliorare l'autostima
- Facolizzarsi su comportamenti e non personalità
- Tecniche di reinforcement per modellare il comportamento
- Ascoltare attivamente
- Stabilire obiettivi
- Mantenere la comunicazione

MANAGING FOR EXCELLENCE

Esistono diversi stili di management:

- Manager as a technician
- Manager as a conductor
- Manager as a developer

Manager as a technician

Eredità del XIXesimo secolo

I manager sono i migliori del team dal punto di vista tecnico

Rapporto basato sull'individuo → No riunioni di gruppo, ma one-one

Utile quando:

- Il manager ha più conoscenza tecnica del team
- Pochi requisiti interpersonali e manageriali
- Emergenze nell'area di esperienza del manager

Problemi:

- Il manager dirige troppo da vicino
- No planning formale
- Focus sulla tecnologia e non sui fattori umani
- Focus sui problemi tecnici e non sulla gestione delle persone

Manager as a conductor

Responsabilità unica, non condivisa con il resto del team → Senso di centralità, charme e carisma

Controllo ampio e costante

Il manager non esegue il lavoro personalmente, ma dirige il team

Utilizzo di strumenti per la gestione dei progetti → Staff meetings

Si chiede consiglio al team per rafforzare la partecipazione

Manager solitamente rispettato dai dipendenti

Utile quando:

- Membri del team non molto esperti
- Task semplici
- Non è necessaria l'eeccellenza per raggiungere gli obiettivi
- Membri del team difficili da gestire

Problemi:

- Troppa coordinazione dei subordinati
- Interdipendenza dei task dei subordinati
- No incoraggiamento del team al problem-solving e alla responsabilità
- Team troppo controllato
- Frustrazione nei membri esperti del team → Meno motivazione

Manager as a developer

Nuova definizione di leadership → Responsabilità e controllo condivisi

Il manager è un coach più che un conduttore

Ha impatto senza esercitare controllo, è coinvolto senza essere centrale, ...

Come ogni problema può essere risolto in modo da sviluppare ulteriormente le capacità del team?

MA coinvolgere le persone nella risoluzione dei problemi può allungare i tempi

Vantaggi:

- Aumentano le probabilità che i task vengano completati con un miglior livello di qualità
- Problemi e difficoltà scoperti prima di diventare crisi
- Conoscenza ed esperienza condivisa
- Creazione di "commitment" → Più motivazione
- Le persone possono assumersi responsabilità

Metodo per creare eccellenza nel lavoro

Gestione del team

L'obiettivo è costruire un team con responsabilità condivisa

Le sfide più comuni nella gestione di un team sono:

- Bilanciare interessi in competizione
- Vedere problemi in un contesto vasto
- Lavorare in collaborazione
- Sostenere buone relazioni di lavoro con persone che non concordano su tutto
- Lottare per i propri punti di vista

Importanza dello sviluppo continuo delle capacità individuali (anche delle figure meno formate del team)

Ogni problema di comportamento come occasione di miglioramento → Effetto motivante

Bilanciamento tra necessità individuali e dell'organizzazione

Costruzione di una visione comune

Task necessari, ma non interessanti potrebbero avere impostanza in un quadro più ampio → Chiarire il goal

OVERARCHING GOAL

Obiettivo generale che fornisce motivazione ai task quotidiani (la retribuzione economica non è sufficiente)

Riflette il nucleo della missione del gruppo

Realistico, ma anche challenging

Importante perché:

- Veicolo di cambiamento
- Visione comune in ambienti con valori eterogenei
- Migliore risoluzione dei confilitti
- Focus su obiettivi ampi
- Il team non si focalizza sul leader

Sviluppo

Processo dinamico → Scambio di idee tra i membri del team

Quale servizio fornisce il gruppo?

Quali obiettivi sono eccitanti sia per il manager che per il gruppo?

Problemi:

- Spesso strati di routine noiosa seppelliscono le funzioni del gruppo
- Non sempre c'è corrispondenza tra le richieste esterne e le aspirazioni degli individui
- Il procedimento può durare circa un anno
- L'ignoto genera paura e quindi spesso viene respinto
- Difficile arrivare in gruppo a concordare su un singolo goal → L'individuazione alla fine spetta al leader

Creazione di commitment

Il goal viene applicato quotidianamente e si crea consenso

Sforzo a lungo termine

Problemi:

- Tutti i gruppi possono avere un overarching goal?
- Quanto tempo occorre per generare il consenso?
- Se necessario, è possibile cambiare l'overarching goal?

Ottenere il massimo da tutti

L'obiettivo è migliorare la capacità del team mentre si esegue un task

Spesso molte opportunità di crescita vengono trascurate

Feedback costruttivi → Rendere la persona più competente

La maggior parte delle persone in un team vuole:

- Imparare
- Migliorare → Ed è in grado di migliorare!
- Prendersi responsabilità

Comunicazione diretta

Per poter dare un buon feedback è necessaria una comunicazione aperta tra manager e membri del team

Spesso il feedback viene dato in modo sbagliato → O troppo "zuccherato" o in modo troppo aggressivo

Il subordinato deve poter influenzare il manager

MEETINGS

Weekly group meeting

1. Stabilire l'agenda
2. Comunicare ciò che avviene nella ditta
3. Sentire da ciascuno l'andamento del progetto
4. Discutere i possibili problemi (ma senza cercare di risolverli sul momento)

Weekly one-one

1. Discutere i possibili problemi
2. Ascoltare!

PERFORMANCE REVIEW

Valutazioni molto dettagliate e puntuali

Focalizzare l'attenzione sul problema di performance, non sulla persona

Parlare con la persona per risolvere il problema, discutere le idee e arrivare a un accordo

Esprimere fiducia nell'abilità a risolvere il problema ed elogiare la persona se la performance migliora

Migliorare una performance media

Identificare un aspetto della performance che è sopra la media e spiegare perché merita speciale riconoscimento

Esprimere il proprio apprezzamento

Comunicare fiducia nelle possibilità di miglioramento

Individui e gruppi con bassa autostima hanno meno facilità a raggiungere gli obiettivi

Performance appraisals

Per migliorare la produttività, è necessario:

- Fornire feedback
- Coaching e counseling
- Sviluppare commitment
- Motivare i dipendenti
- Stringere relazioni tra manager e dipendenti
- Diagnosticare problemi individuali e di organizzazione

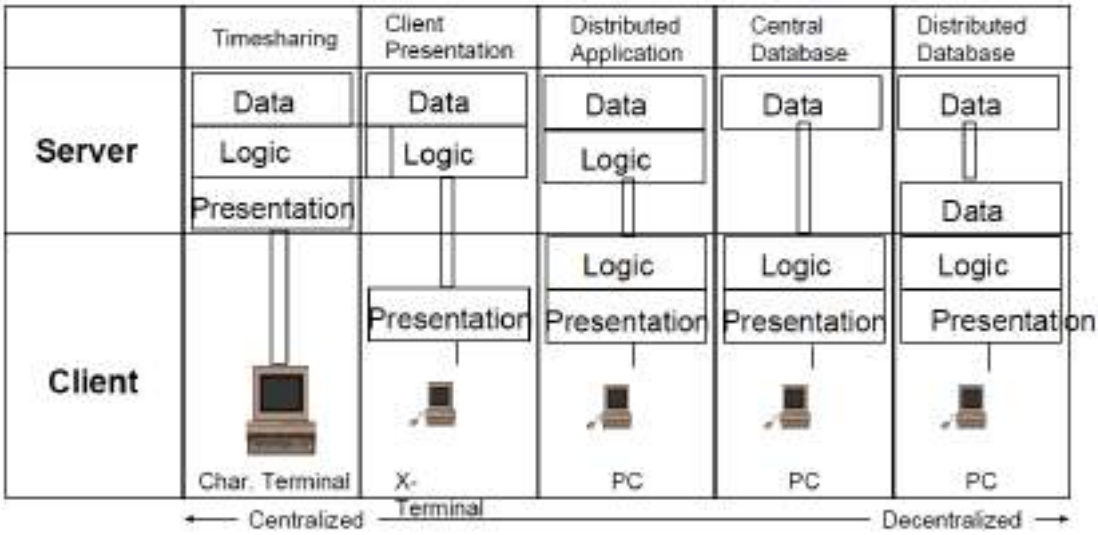
ARCHITETTURE CLIENT/SERVER

Architettura di rete nella quale genericamente un computer **client** si connette ad un **server** per la fruizione di un certo servizio

Tre aspetti principali:

- Dove si trovano i dati?
- Dove si trova la logica di controllo?
- Dove si trova il livello di presentazione?

CLASSIFICAZIONE STORICA DELLE SOLUZIONI CLIENT/SERVER



Thin client:

- Timesharing
- Client presentation

Fat client:

- Central database
- Distributed database

Timesharing

Dati, logica di controllo e presentazione tutti sul server

Char. Terminal → Terminale stupido che invia solamente comandi al mainframe e visualizza l'output prodotto dal server

No parallelismo, viene servito un solo terminale alla volta

Livello di presentazione quasi assente

Client presentation

Livello della presentazione spostato nel client

X-Terminal → Leggermente più avanzato del Char. Terminal, nascono le prime interfacce grafiche che permettono ai client di prendere in carico l'aspetto della presentazione

Distributed application

Logica applicativa suddivisa tra client e server

Soluzione resa possibile dall'evoluzione e dall'aumento di potenza dei dispositivi

Es. I browser e gli smartphones ora sono così evoluti che è possibile eseguire molto codice sul client (controlli dell'input in Javascript)

Se cambia qualcosa nel server, potrebbe essere necessario modificare anche il software presente su ogni client

Central database

Tutta la logica di controllo è spostata sui client

I server diventano grandi database privi di logica di controllo

Approfonditi più in dettaglio in seguito

Distributed databases

Dati (parzialmente) duplicati

Potenziali problemi di sincronizzazione dei dati → Inconsistenza

Soluzione adottata quando grandi aziende hanno più sedi distribuite nel mondo (magari non connesse a Internet)

CENTRAL DATABASE (FAT CLIENT)

Sono presenti degli standard → SQL, ODBC, DRDA

Two Phase Commit (2PC) → Gestione delle transazioni in ambiente distribuito

- I commit avvengono in due fasi:
  - PREPARE TO COMMIT → Il coordinatore della transazione manda un messaggio a tutti gli agenti interessati dalla transazione
  - COMMIT se tutti rispondono positivamente entro un timeout, ROLLBACK altrimenti
- Controllo completo delle tranzazioni

Applicazione nel Data Warehouse

Vantaggi

- Alta produttività nei linguaggi 4GL
- Prodotti maturi ed efficienti
- Buona standardizzazione

Svantaggi

- Espressività limitata al modello ER
- Scalabilità e tuning limitati per grandi sistemi → Essendoci un solo database centrale, si possono gestire un numero limitato di connessioni simultanee
- Forte sollecitazione della rete
- Forte uso delle risorse del client
- Manutenzione costosa
- L'accesso non protetto al database può provocare errori che rendono i dati inconsistenti
- In Internet downloaed degli applet lento

Stored procedures

Procedure effettuate all'interno del database come pre-operazioni sui dati, mantenute nel DB stesso

Per ridurre il numero di accessi al database (utilizzando comandi SQL l'interazione client/server è elevata) e la quantità di dati scambiati

Evitano al client di riscrivere query complesse

Solitamente compilate dal DBMS che le ottimizza quando vengono inserite la prima volta nel DB

MA non portabili da un RDBMS a un altro

Parte della logica viene spostata sul server

Replicazioni asincrone

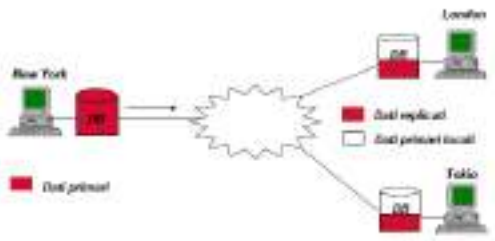
Per garantire una maggiore scalabilità è possibile utilizzare delle architetture distribuite, ma bisogna gestire il problema della replicazione dei dati

Le architetture passate prevedevano la replicazione di alcuni dati in locale con un database centrale contenente tutti i dati

Problema di gestire la sincronizzazione sia in locale, sia nel database centralizzato

Tempi morti derivanti dai diversi fusi orari per effettuare le operazioni di sincronizzazione

In ogni caso enormi problemi di inconsistenza nei dati



MIDDLEWARE

Middleware → Software di sistema che permette l'interazione a livello applicativo fra programmi in un ambiente distribuito

Ne esistono diversi tipi:

- TP monitor
- Message Oriented
- Publish/Subscribe
- Object Request Broker
- Object Transaction Monitor

TP MONITOR (OLTP)

Middleware che forniscono supporto alle transazioni → Permettono che un insieme di operazioni venga eseguito completamente o annullato (proprietà ACID)

Gestione di processi server, transazioni e comunicazione client/server

Sincronizzazione in 2PC

I client interagiscono con servizi remoti che possono essere allocati e duplicati su processi e macchine fisiche differenti

Permettono il bilanciamento del carico tra server equivalenti e la gestione di pool di risorse

High availability in caso di caduta di un server

Richieste instradate da parte del middleware

Funnelling → Se i client richiedono una risorsa, ma non la utilizzano tutti assieme, il middleware può alternare l'assegnazione ai soli client che la utilizzano realmente (in modo trapsarente al client)

Riuso delle risorse da un client a un altro

Es. CICS, IMS, Tuxedo, ...

Vantaggi

- Scalabilità e tuning per grandi sistemi
- Adatti ad applicazioni mission critical → Situazioni in cui ci sono importanti ripercussioni di business se le transazioni hanno problemi (es. Home Banking)

Svantaggi

- Pochi standard → Minore produttività
- Uso limitato di linguaggi 4GL

Scalabilità dei sistemi

La scalabilità doi un sistema dipende sia da risorse hardware che da risorse software

A seconda della dimensione del sistema che si vuole creare sono necessarie diverse funzionalità

OBJECT REQUEST BROKER

Programmazione a oggetti

Rende automatici alcuni principi di buona programmazione (es. incapsulamento, modularità, ...)

Favorisce il riuso del software e lo sviluppo a componenti

Semplifica la manutenzione

Possiede una buona base metodologica → UML

MA inizialmente linguaggi non interoperabili

Oggetti locali al processo che li ha generati → Non persistenti

OODBMS per molto tempo privi di standard e legati ai linguaggi di progammazione

Le grandi aziende utilizzavano già un vasto assortimento di software scritti in modo procedurale e la loro integrazione con i linguaggi di programmazione ad oggetti non è semplice

ORB → Frammento di software middleware che permette di effettuare chiamate di programma tra computer diversi in una rete gestendo le trasformazioni delle strutture dati dei processi in byte e viceversa (marshalling o serializzazione) e la trasmissione in rete

Es. OMG CORBA, Microsoft COM/DCOM, ...

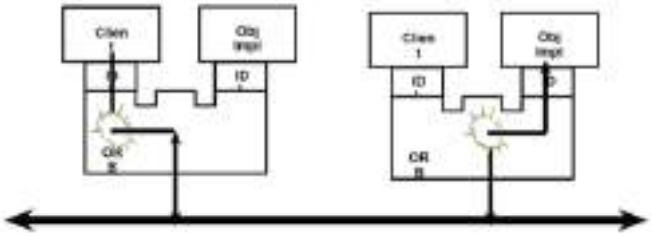
CORBA (Common Object Request Broker Architecture)

Standard per architetture distribuite ad oggetti definito dall'Object Management Group (OMG)

OMG raggruppa numerosi produttori di software, ma non Microsoft

Rende possibile la comunicazione di software scritti con linguaggi ad oggetti diversi e distribuiti in rete

Simile a RMI (un oggetto su un client vuole utilizzare un metodo su un oggetto remoto), ma non è necessario che entrambi i programmi siano scritti in Java



Interface Description Language (IDL) → Astrazione utilizzata in CORBA per separare le interfacce degli oggetti dalle loro implementazioni

L'IDL ermette di specificare:

- Moduli
- Interfacce
- Operazioni
- Tipi di dati

Skeleton → Oggetto che contiene le informazioni relative a un metodo remoto (signature, ...) in modo che il middleware possa tradurre le richieste dei client → Contenuto nel server

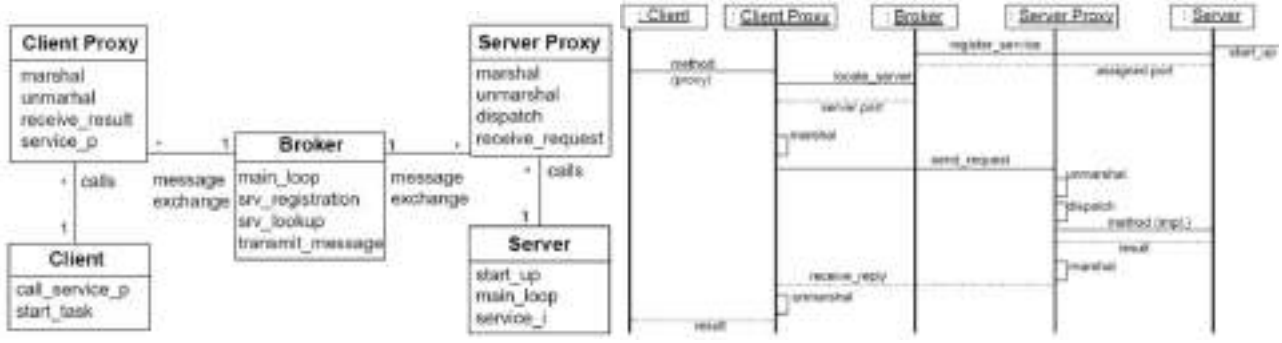
Stub → Permette di simulare la presenza di un metodo che in realtà è remoto → Contenuto nel client

Come in RMI, viene effettuato il **marshal** e l'**unmarshal** dei parametri (sia di input che di output)

La serializzazione consente di "appiattire" gli oggetti in modo tale da rendere più semplice l'invio

A differenza di RMI, il marshal appiattisce un oggetto scritto in un linguaggio e l'unmarshal lo trasforma in un oggetto scritto in un altro linguaggio

Viene utilizzato il **pattern broker**:

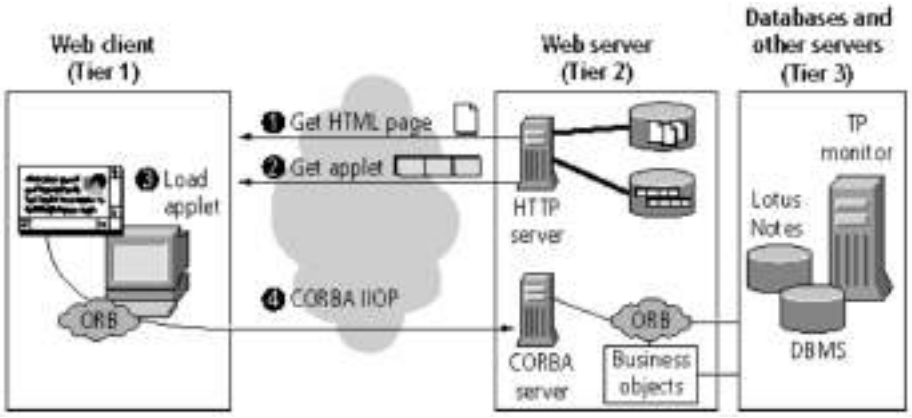


Dynamic binding → Permette di invocare un metodo senza sapere in anticipo qual è il server che gestirà la richiesta, né l'oggetto specifico che la soddisferà

Gli oggetti che forniscono il servizio si registrano presso il middleware in modo tale che sia quest'ultimo a decidere chi sarà a soddisfare una richiesta → Load balancing

Si possono scrivere i programmi client senza molte informazioni riguardanti i server, l'unica cosa che i client devono sapere è qual è il servizio che richiedono, il middleware provvederà poi a cercare qualche oggetto che soddisfi la richiesta

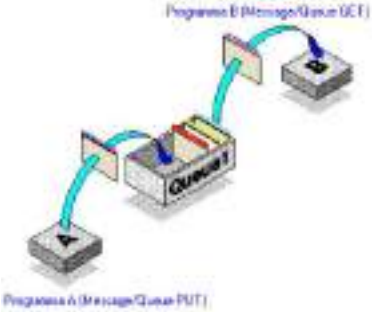
Se necessario CORBA è utilizzabile anche per l'interazione via Internet



Interface repository → Contiene tutte lle interfacce dei metodi invocabili da remoto

MESSAGE ORIENTED MIDDLEWARE

Infrastruttura client/server che, distribuendo un'applicazione tra più piattaforme eterogenee, ne incrementa interoperabilità, portabilità e flessiblità



One-to-one

Es. IBM MQSeries, MS Message Queue Server, ...

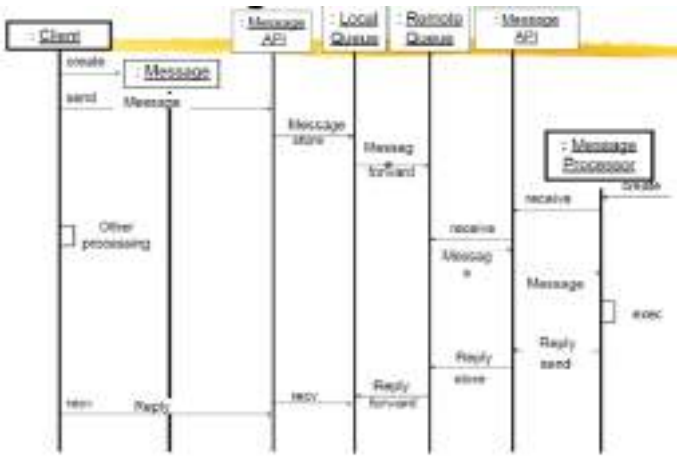
Il programmatore può ignorare i dettagli dei protocolli di rete, sistemi operativi, ...

Disaccoppiamento tra mittente e destinatario

Il middleware risiede sia sul client che sul server e consente l'esecuzione di **chiamate asincrone** tra i due

I messaggi inviati verso programmi non disponibili vengono memorizzati in apposite code che li conservano finché il programma non torna disponibile per la consegna → Consegna garantita

Il mittente può inviare il messaggio senza dover controllare se il server è pronto → **Fire & Forget**



Dopo aver inviato il messaggio, il client esegue altre operazioni, non rimane in attesa della risposta

La coda locale permette al client di continuare l'esecuzione anche nel caso in cui sia il server, sia il broker siano down

Quando il broker torna raggiungibile i messaggi vengono inviati dalla coda locale alla coda remota

PUBLISH/SUBSCRIBE

Mittenti e destinatari dialogano tramite un mediatore detto **dispatcher** o broker

One-to-many (multicasting)

Il mittente del messaggio (**publisher**) non è consapevole dell'identità dei destinatari (**subscriber**), ma "pubblica" il messaggio al dispatcher

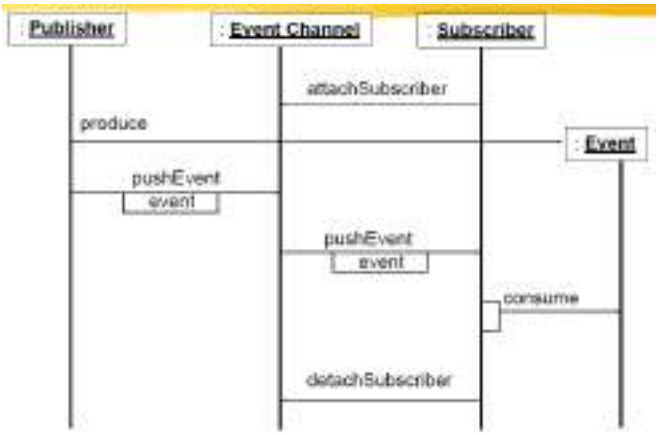
I destinatari si rivolgono anche loro al dispatcher "abbonandosi" alla ricezione di messaggi

Il dispatcher inoltra ogni messaggio inviato da un publisher a tutti i subscriber interessati ad esso → **Push**

I subscriber possono precisare in modo specifico a quali messaggi sono interessati e richiedere esplicitamente i messaggi

Applicazione del paradigma "event-driven"

Es. TIBCO Rendezvous, IBM WebSphere MQ, ...



CONCLUSIONI

Modelli di comunicazione fra programmi

Applicazioni dei middleware

Applicazioni interattive, sincrone "request-reply":

- TP Monitor (OLTP)
- Object Request Broker (ORB)
- Object Transaction Monitor (OTM) e Application Server (AS)

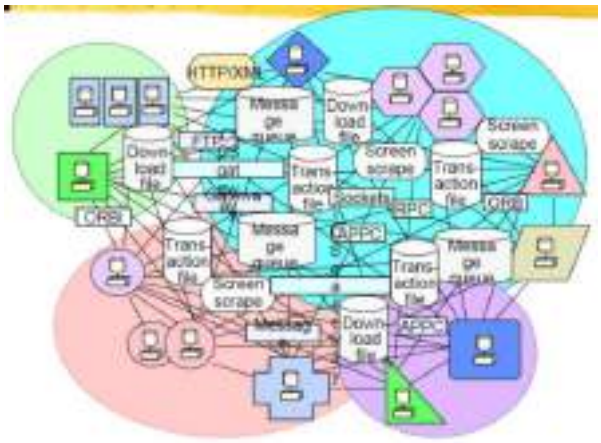
Comunicazioni unidirezionali, asincrone, "Fire & Forget":

- Message Oriented Middleware (MOM)

- Publish/Subscribe

La complessità dei sistemi informativi aziendali nasce dalla sovrapposizione nel tempo delle soluzioni

Es. I sistemi informativi delle banche sono molto complessi e necessitano quindi di un'infrastruttura di intercomunicazione per dialogare tra loro



È necessaria un'infrastruttura aziendale di intercomunicazione tra le applicazioni

Workflow nascosto all'utilizzatore

Sono possibili accessi differenziati e controllati

Struttura a cipolla che fornisce visioni diverse dell'infrastruttura

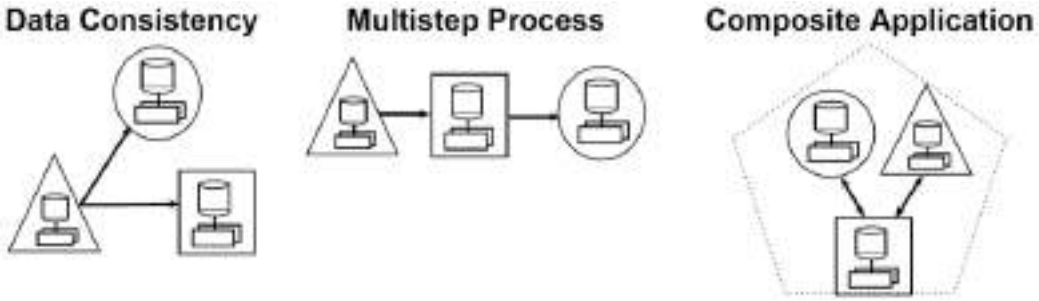
Integration broker

È necessario integrare i sistemi informativi delle aziende, ma anche gestire il **workflow** (coordinazione di attività)

Workflow rappresentabili in UML tramite **activity diagrams**

Intelligent routing → I client non necessitano di sapere l'indirizzo esatto del server che soddisferà la loro richiesta

Integration patterns



Data consistency → Sistemi fisicamente e logicamente indipensenti, interazione acincrona a senso unico, trasferimenti batch, più business process (Es. Accorpamento di due aziende)

Multistep process → Sistemi fisicamente indipendenti, ma logicamente dipendenti, interazione asincrona a senso unico, un solo business process

Composite application → Sistemi fisicamente e logicamente dipendenti, interazione asincrona a doppio senso, un solo business process

SOFTWARE ENGINEERING

Alcune attività sono necessarie per la produzione di un sistema software:

1. Analisi dei requisiti
2. Design
3. Codifica
4. Testing
5. Manutenzione

Con ovvie dipendenze tra le fasi (il design non può iniziare prima dell'analisi dei requisiti, ...)

Ogni tecnica di programmazione definisce una dipendenza reciproca tra le fasi

METODOLOGIE TRADIZIONALI

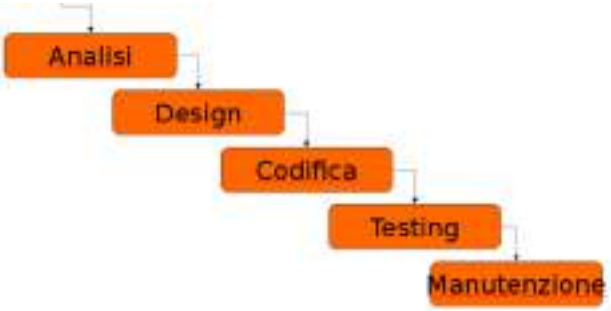
Modello a cascata classico

Rigida implementazione end-to-start (sequenziale) delle dipendenze

Primo modello di processo pubblicato per il software development → Royce, 1970

Derivato da altri processi ingegneristici

Semplice, MA la sequenzialità è difficile da rispettare, non è prevista incertezza e modifica dei requisiti e si consegna solo alla fine



Modello incrementale

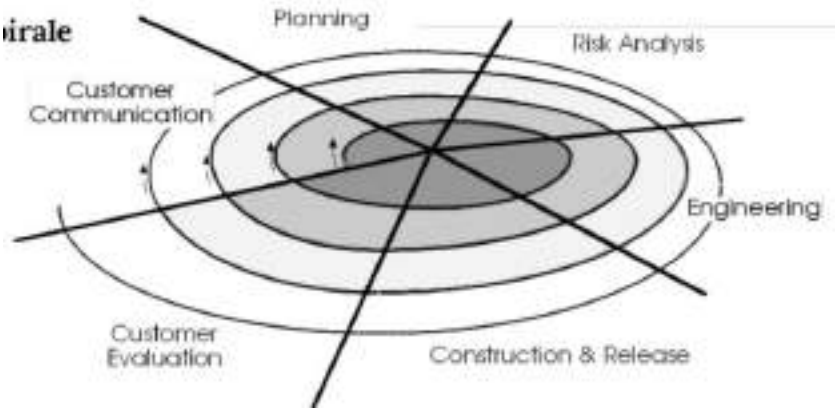
Gilb, 1988

Sistema consegnato in pezzi, alta priorità prima

Incrementi iniziali formano requisiti per futuri incrementi

Dimensione degli incrementi variabile (originalmente un massimo di alcune settimane)

Modello a spirale



Modello iterativo la cui caratteristica principale è l'analisi del rischio, che consente di ridurre i rischi derivanti dalla creazione di software complessi

No parallelismo

Il modello **WinWin** è un'evoluzione del modello a spirale che introduce l'identificazione delle priorità degli **stakeholders**, ovvero gli attori che partecipano allo sviluppo del software (anche marketing, clienti, ecc)

Non è detto che le priorità del cliente siano accettabili, alcune volte occorre prima implementare delle porzioni di software non esplicitamente richieste, ma necessarie per la realizzazione del software nel complesso (**reconcile win conditions**)

EXTREME PROGRAMMING (XP)

"Listening, Testing, Coding, Designing. That's all there is to software. Anyone who tells you different is selling something." Kent Beck

Processo software evolutivo e leggero

La caratteristica principale è ascoltare il cliente, si pone molta attenzione sul capire i requisiti del software da realizzare

No documentazione inutile e poco aggiornata → Solo se davvero necessario per capire e commenti nel codice

Quattro valori:

- Semplicità
- Comunicazione
- Feedback
- Coraggio, abbraccia il cambiamento

Principi base:

- Feedback rapido
- Assume semplicità
- Cambiamenti incrementali
- Abbraccia il cambiamento
- Lavoro di qualità

XP practices

Pair programming → Maggior controllo sul codice sviluppato

Proprietà collettiva → Tutti devono sapere tutto sul codice

Releases corte → Più facile effettuare testing e modifiche, il cliente non rimane spaesato

Major release → Introduce cambiamenti, nuove funzionalità

Minor release → Correzioni, bugfix

Integrazione continua → Le modifiche apportate al software vengono integrate quotidianamente (es. nightly build) con il software principale, possibilmente aiutandosi con software di controllo di versione

Refactoring fondamentale perché non essendoci un'accurata fase di progettazione le cose cambieranno durante lo sviluppo

Sviluppando un pezzo alla volta velocemente, senza troppa analisi, sarà più facile apportare delle modifiche

Se si riescono a sviluppare componenti semplici e indipendenti tra loro, le modifiche avranno ripercussioni solo su una porzione limitata dell'architettura complessiva

Settimana di 40 ore → No stanchezza dei programmatori nel codice

Client on-site → Confronto continuo con il cliente

Standard di codifica → Codice leggibile, commentato, semplice

Metafore → L'architettura e l'interazione con l'utente non devono essere strampalate (vedi IUM)

Planning game → Tutte le persone che sviluppano contribuiscono alla costruzione del piano e delle sue stime

Fasi di XP

Durante il planning si deve tenere conto di ciò che ha richiesto il cliente per poter dare la giusta priorità alle diverse funzionalità da implementare

User Stories → Casi d'uso, ma meno dettagliati di quelli di UP

Architectural spike → Abbozzo di architettura per ottenere una metafora del sistema e una stima ragionevole dei tempi, opzionale se si sa già come strutturare l'architettura

Spike → Prove di codifica per prendere confidenza con le tecnologie da utilizzare utili per capire le tempistiche di realizzazione delle varie componenti (non sono prototipi)

Release planning → A partire da user stories e spikes di genera un piano completo delle tempistiche (**release plan**)

Iteration → Non è una fase vera e propria, ma il momento in cui avviene coding, testing e integration, tutto assieme

Acceptance test → Il cliente deve controllare che il prodotto sia conforme alle user stories, è possibile tornare indietro se ci sono delle cose da modificare o se bisogna ancora implementare delle user stories

Small release → Piccola parte di software, approvata dal cliente

"Analizzare un poco, design un poco, codificare un poco, testare un poco"

Funzionalità principali prima

AGILE PLANNING

La pianificazione tradizionale è una buona scelta se si ha la certezza di cosa andare a fare nel minimo dettaglio

Se il progetto è complesso bisogna gestire l'incertezza → **Agile planning**!

Si pianifica a breve termine e in più fasi → Planning e processo software sempre più vicini

Time of commitment → Frequenza con la quale si mantengono le promesse fatte al cliente (es. con waterfall si fa all'inizio una promessa molto corposa e si consegna alla fine, mentre con XP si fanno poche promesse mantenute velocemente)

Focus sul breve termine per non prendere decisioni irreversibili

Si possono mostrare i progressi al cliente

Non adatto a progetti a lungo termine (si perde di vista la "big picture"), ma ottimo per situazioni complesse

Committente fortemente coinvolto per rendere espliciti subito i problemi

Focus prima su ciò che è più importante/urgente → Customer-set priorities

MA il committente non dovrebbe essere coinvolto nel planning

Stand-up meeting → Incontri di coordinazione tra gli sviluppatori in cui tutti stanno in piedi (per evitare che diventino troppo lunghi), si discute di eventuali problemi e si organizza lo sviluppo giornaliero informando gli altri dei progressi del giorno prima

Il codice è di tutti, per mantenere il sincronismo

Test sviluppati prima della codifica

Tutti i test precedenti dovrebbero essere eseguiti di nuovo quando una nuova funzionalità viene aggiunta (per evitare side-effects)

No misurazione del progresso di ogni singola attività, ma rispetto al goal

OO ANALYSIS & DESIGN

Prima della definizione vera e propria degli oggetti (**fase di design**), c'è una fase di progettazione in cui non si ha ancora idea delle entità che comporranno il dominio (**fase di analisi**) in cui si individuano le entità principali della realtà in cui si opera

I diagrammi UML fanno parte della fase di design

ELICITAZIONE DEI REQUISITI AGILE

Verificare le software requirements specifications (SRS) è importante perché:

- Guidano lo sviluppo del software
- Possono funzionare da contratto tra committente e fornitore

Ma se i progetti cambiano frequentemente è complesso produrre SRS verificabili → Approccio AGILE!

I requisiti devono essere descritti ad alto livello, con frasi brevi

Interazione continua con il cliente

Consegna rapida di piccole parti di software al cliente

User stories

Scritte dal cliente, ma completate da comunicazione durante l'analisi per migliorare la comprensione

Per ognuna viene stimato il tempo ideale di sviluppo (senza distrazioni, ...)

Costituite da titolo, acceptance test, priorità, story points, descrizione, ...

Devono essere comprensibili dal cliente

Di dimensione tale che diverse user stories possano venir completate in un'iterazione

Il più possibile indipendenti tra loro

Devono essere testabili

Approccio goal-oriented → Le user stories sono costituite da fasi per raggiungere un goal

Approccio scattergun → User stories generate dalle aspettative che compaiono nella conversazione

Acceptance tests fondamentali per determinare se un'implementazione corrisponde a quanto specificato

Più test per ogni user story

User stories spesso arrichite da requisiti non funzionali (sicurezza, privacy, usabilità, ...)

Analysis → I concetti del sistema tratti dalle user stories vengono esaminati, tramite le **CRC cards** si definisce come l'utente interagirà con il sistema

Design → L'output dell'analisi viene utilizzato per definire un insieme preciso di classi e oggetti, la loro interazione e i loro attributi

OBJECT ORIENTED ANALYSIS (OOA)

L'approccio object oriented permette di concentrarsi sui concetti importanti del progetto invece che sulle funzionalità tecniche

Combinare dati e funzioni → Modularità, incapsulamento, riuso

Lo stesso modello (OO) guida tutte le fasi (analisi, design, programmazione)¨

OOA model → Descrive gli oggetti principali del dominio e le operazioni da effettuare su di essi

Indipendente dalla UI o dal DB o dal linguaggio di programmazione!

Gli sketch di UI ed eventuali prototipi orizzontali vengono fatti invece quando si definiscono le user stories

Oggetti che appartengono a classi e sono caratterizzati da:

- Stato → Cosa sono
- Comportamento → Cosa fanno

Classi organizzate in:

- Gerarchie
- Tassonomie

Gli oggetti sono istanze di classi

Come individuare le classi e gli oggetti?

Descrizione testuale

A partire dagli use cases si estraggono nomi e verbi

Nomi → Oggetti

Verbi → Metodi

Attenzione a sinonimi → **Glossario dei termini**

Solo sostantivi rilevanti

Categorizzazione dei nomi identificati

Identificazione di attori e obiettivi

CRC cards

Class Responsibility Collaboration

Beck, Cunningham, 1989

Class → Entità che compongono il dominio

Responsibility → Attributi e metodi delle classi

Collaboration → Relazioni (Scambio di informazioni) tra le classi, lista di classi cooperanti, una classe ha bisogno di un'altra classe per portare a termine i suoi compiti?

Dalle user stories si identificano le classi interessanti utili per descrivere il sistema e si compila una CRC card per ognuna

Template su dei cartoncini contenenti il nome della classe, la responsabilità principale, le responsabilità e le classi con le quali collabora

Class Name	
Main Responsibility	
Responsibilities	Collaborators
. . .	. . .

Eventualmente si possono mettere anche le proprietà della classe, che non sono da intendere come proprietà delle classi software, ma le caratteristiche del concetto stesso

Tali classi saranno approfondite maggiormente nella fase di design

Buon collegamento tra use cases e codice

Essendo semplicemente dei pezzi di carta, è immediato modificarle, aggiungerne di nuove, metterne da parte

Trovare nomi adatti alle classi è molto importante

Non devono essere troppo dettagliate, perché nell'iterazione seguente possono sparire o cambiare radicalmente

Attenzione alle gerarchie → Generalizzaizoni e specializzazioni devono riflettere le relazioni degli oggetti nel dominio

Alcuni approcci estremi suggeriscono di programmare direttamente a partire dalle CRC cards

1+5 steps

0. (Brainstorming) → Tutte le idee sono potenzialmente buone
1. Identificazione delle classi candidate (iterativamente)
2. Selezione di un insieme coerenti di use case, scenari, user stories (esaminando prima quelli più semplici)
3. Per ogni scenario, identificazione dei nomi delle CRC cards e delle responsabilità (iterativamente)
4. Modifica delle situazioni (assunzioni iniziali) per testare le cards
5. Aggiunta di cards, rimozione di altre per far evolvere il design

Valutazione del design

Sei modi per valutare il design:

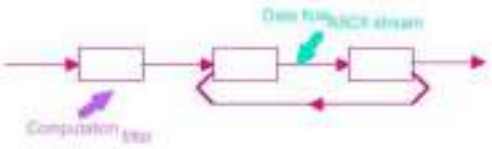
- **Connessione dei dati** → Si possono ottenere tutte le informazioni necessarie attraversando la rete delle collaborazioni?
- **Astrazione** → I nomi degli oggetti identificano le astrazioni del dominio a cui si riferiscono?
- **Allineamento delle responsabilità** → Nomi responsabilità e funzioni sono coerenti?
- **Variazione dei dati** → Il design si adatta facilmente ad un cambiamento nei dati?
- **Evoluzione** → Quante classi devono venire modificate se si introduce un cambiamento?
- **Pattern di comunicazione** → Sono presenti dei pattern di comunicazione strani (es.cicli)?

PATTERN ARCHITETTURALI

Pipe and filters

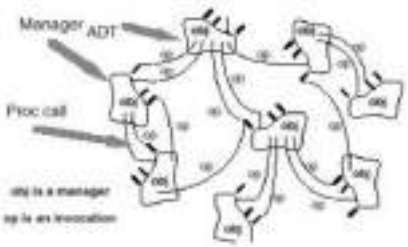
Architettura per programmi batch, senza user interface

Mandare in **pipe** due programmi significa mandare in esecuzione il primo per poi eseguire il secondo passandogli il risultato del primo in input



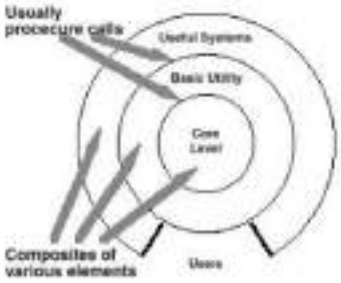
Filtri → Programmi che prendono in input dei dati, ad esempio da un file

Data abstractions and OO organizations



Layered systems

Architettura tipica dei sistemi operativi (anche se è possibile sviluppare anche altri programmi con questa architettura)



Repositories

Sono architetture tipiche dell'intelligenza artificiale

Memoria condivisa detta **blackboard**

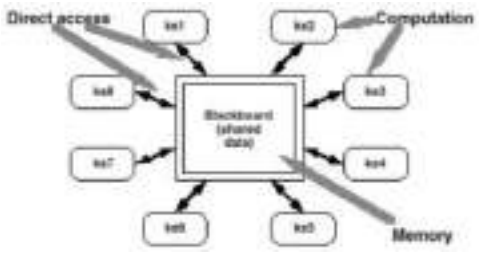
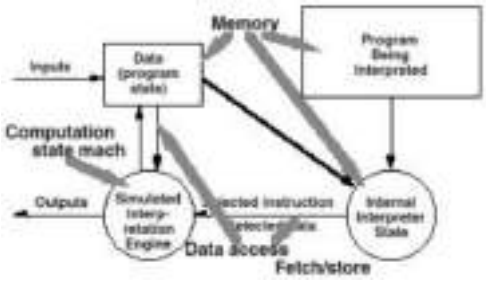


Table driven interpreters

Architettura tipica di compilatori, interpereti e parser



OBJECT ORIENTED DESIGN (OOD)

Dopo la fase di analisi ci si concentra sui dettagli relativi all'implementazione → UI, DB, ...

Modelli rappresentati con diagrammi UML

Nel progetto l'UML serve per:

- Descrivere ad alto livello quello che stiamo progettando
- Comunicazione tra i gruppi
- Capire meglio come sviluppare il sistema

Modelli funzionali → Descrivono le trasformazioni dei dati nel sistema

Modelli statici → Rappresentano la struttura del progetto

I **class diagram** sono simili alle CRC Cards, ma sono più dettagliati

Modelli dinamici → Rappresentano una fase di esecuzione (runtime)

Gli **state diagram** di solito sono inutili, vanno fatti solo quando lo stato del programma è "complicato"

I **collaboration diagram** descrivono sostanzialmente la stessa cosa dei **sequence diagram**, ma con una maggiore attenzione sulla relazione tra gli oggetti (le collaborazioni runtime), mentre i sequence diagram pongono l'accento sulla sequenza delle operazioni (nel tempo)

I collaboration diagram sono anche meno dettagliati per quanto riguarda ciò che restituiscono le chiamate

Gli **activity diagram** rappresentano un'esecuzione, ma c'è un maggior focus sul workflow

Le attività racchiuse tra linee parallele rappresentano le attività che possono essere svolte parallelamente

Gli activity diagram vanno fatti solo se il livello di parallelismo di alcuni task è importante, se tutti i task sono sequenziali non è rilevante

Spesso i task rappresentano invocazioni di servizi esterni ed è quindi possibile proseguire con altre elaborazioni mentre si aspetta la risposta, in questo caso è anche utile la stesura degli activity diagrams

I **component diagram** servono solo per applicazioni piuttosto corpose, le componenti possono essere sia classi, sia sistemi complessi

I **deployment diagram** sono utili nel caso di sistemi distribuiti

Il committente non ha alcun interesse nel vedere i diagrammi UML, è più importante mostrargli i mock-up

Linee guida per OOD AGILE

Evolutionary design → Il designe deve evolversi man mano che il sistema viene implementato e si evolve

Semplicità → YAGNI "You Aren't Going to Need It"

Se un determinato software non è strettamente necessario non va implementato

Non è consigliabile tentare di sviluppare subito tutto il sistema, occorre fare le cose in modo graduale

Non bisogna neanche pensare subito all'architettura (infrastruttura) complessiva già da subito, verrà definita in fase di sviluppo

Bisogna pensare ad un'architettura di base (es. 3 livelli), ma senza scendere troppo nel dettaglio fin da subito

No logica applicativa duplicata

Metafore → Utilizzare metafore che forniscono una visione e un vocabolario condiviso

Refactoring → Miglioramento graduale della struttura interna del sistema attraverso la riorganizzazione del codice

Il refactoring è un elemento fondamentale dell'extreme programming

Dato che non è pensabile di sviluppare fin da subito del codice perfetto, sarà necessario modificarlo per migliorarlo

Uno dei motivi per cui è difficile sviluppare subito del buon codice è la necessità di consegnare il prodotto con scadenze strette

Migliora il design controllando l'entropia dovuta alle modifiche del sistema

Aumenta la leggibilità e la semplicità

Aiuta a trovare bug

Aumenta la profuttività

Quando è necessario fare refactoring:

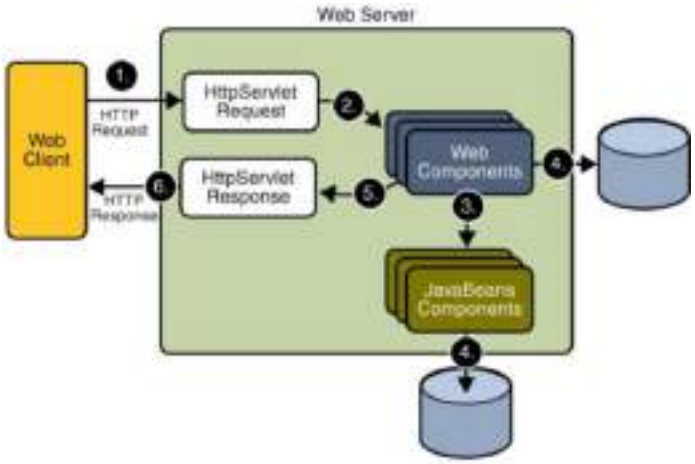
- Presenza di bug
- È difficile aggiungere nuove funzionalità al sistema
- Il codice è oscuro
- Bisogna aumentare la performance
- "Bad smells" nel codice (Troppo codice, troppo poco codice, codice inutilizzato, nomi che generano confuzione, ...)

J2EE

JAVA 2 PLATFORM ENTERPRISE EDITION

Framework per lo sviluppo di applicazioni server-side complesse

Ha delle proprietà di middleware e permette di realizzare dei sistemi aziendali di grosse dimensioni, con architetture multi-livello, bilanciamento di carico, scalabilità, ...



Il framework .NET offre funzionalità molto simili a J2EE, ma è meno scalabile

Da utilizzare quando:

- L'applicazione deve essere scalabile
- Transazioni devono assicurare data integrity
- L'applicazione avrà più clients

Applicazioni "enterprise"

Enterprise software applications → Applicazioni SW che facilitano la gestione delle attività di impresa

Caratterizzate da:

- Necessità informative
- Complessità dei processi di business
- Eterogeneità delle applicazioni

Devono fornire:

- Velocizzazione del processo di sviluppo delle applicazioni
- Affidabilità e disponibilità
- Sicurezza
- Scalabilità
- Integrazione

Diverse soluzioni per affrontare problemi diversi → J2EE permette di integrare tali soluzioni

Caratteristiche di J2EE

Modello di programmazione con approccio alla costruzione di applicazioni basato su API

Infrastruttura che permette di eseguire applicazioni in modo efficiente e scalabile

Java Connectors → API che consentono di creare dei wrapper per facilitare la comunicazione di sistemi legacy con il nuovo software sviluppato utilizzando il framework

Java Messaging Service (JMS) → API per la gestione di messaggi asincroni point-to-point (con coda) o publish & subscribe per i broadcast → **Message-Driven Java Beans**

Java Transaction Service (JTS) → API per la gestione delle transazioni che supportano XA e CORBA

Enterprise Java Beans (EJB) → Modello delle componenti lato server, aiutano il programmatore a gestire la sessione e la logica di business

Application clients → Veri e propri programmi Java che si interfacciano con la business logic contenuta nel server per fornire dei servizi agli utenti. Differiscono dalle web applications perché non vengono eseguite su un browser (non esistono quasi più perché vengono sempre più sfruttati i web clients)

Session Beans → Utilizzati per gestire delle particolari interazioni con l'utente

Se si definisce un Session Bean **stateful**, le variabili sono legate all'utente che ha generato la sessione e durano per tutta la durata della sessione

Se si utilizza un Session Bean **stateless** le variabili non sono dedicate al singolo utente ed è quindi necessario gestire tale interazione attraverso altri meccanismi, come la persistenza o i cookies

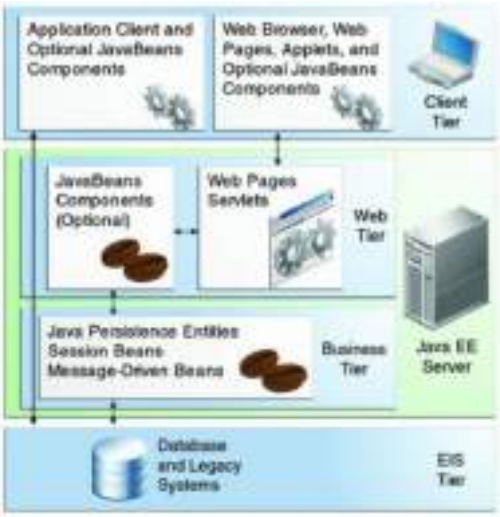
Singleton Java Beans → Comuni a tutto il livello applicativo e supportano due metodi di accesso concorrente: gestito dal web container o gestito dal bean stesso

Persistenza → È possibile definire oggetti persistenti quando determinati Beans rappresentano business entities o quando il loro stato deve essere persistente

Inoltre viene offerto il supporto per:

- Java Naming and Directory Interface (JNDI)
- Remote Method Invocation (RMI)
- Servlets
- Java Server Pages (JSP)
- Java DataBase Connection (JDBC)
- Java Authentication and Authorization Service
- JavaMail

Livelli di J2EE



EJB SERVER E CONTAINER

EJB Server

Fornisce la Java Virtual Machine e le classi di supporto agli EJB

Funzioni di base di ORB e TP Monitor

Accesso ai DB

Bilanciamento del carico & high availability

EJB Container

Fornisce l'ambiente in cui gli AJB di una classe vengono eseguiti

Mantiene pools di istanze di beans pronti per gestire le richieste, gestendo il passaggio da attivi a inattivi e gestendo il threading

Realizza assieme all'EJB Server i servizi sopra descritti

Sincronizza variabili di istanza degli Entity Beans con il DB

PERSISTENZA

Annotazioni per la persistenza definite in **javax.persistence**

Entità

Oggetto persistente del dominio

Rappresenta una tabella in un DB relazionale

- Ogni istanza è una riga nella tabella
- Lo stato persistente è rappresentato da campi e proprietà
- Object-relational mapping effettuato tramite annotazioni

Annotazione **@Entity**

Deve essere presente un costruttore senza parametri, ma possono essere presenti altri costruttori

NON deve essere dichiarata final

Le variabili di istanza persistenti devono essere dichiarate private, protected o package-private e definite con le apposite annotazioni

Annotazione **@Transient** per i campi che non devono essere resi persistenti

I client devono accedere allo stato dell'entity attraverso metodi accessori o business (che devono seguire le convenzioni per i Java Beans)

Ogni entità deve avere come campo una chiave primaria univoca che può essere semplice (un solo campo) o composita (più di un campo, definite da apposite classi) definito con lánnotazione **@Id**

Supporto per ereditarietà e polimorfismo (classi astratte definite come Entity)

EJB Object remoto

Rappresenta l'interfacci dell'EJB verso l'esterno

Filtra ed integra i messaggi verso le istanze di EJB

Coordina le transazioni

EntityManager

Gestore delle entità → Inserimento, rimozione, recupero di istanze dal DB

Ogni istanza di EntityManager è associata con un **persistent context** che definisce l'ambito nel quale un'entità viene creata e persiste

Container-Managed Entity manager → Il contesto è propagato dal container a tutte le componenti dell'applicazione

Annotazione **@PersistentContext**

Application-Managed Entity manager → Il contesto non è propagato a tutte le componenti, si usa quando una applicazione deve accedere al persistent context attraverso multiple istanze di EntityManager in una particolare persistence unit

Annotazione **@PersistenceUnit**

Trovare le entità

```
@PersistentContext
EntityManager em;
public void enterOrder (int custId, Order newOrder) {
   Customer cust = em.find(Customer.class, custId);
   Cust.getOrders().add(newOrder);
```

```
   newOrder.setCustomer(cust);
}
```

Memorizzare istanze di entità

```
@PersistentContext
EntityManager em;
public LineItem createLineItem (int quantity, Order order, Product product) {
   LineItem li = new LineItem (order, product, quantity);
   order.getLineItems().add(li);
   em.persist(li);
   return li;
}
```

Rimuovere un'istanza di entità

```
@PersistentContext
EntityManager em;
public void remove (int orderId) {
   try {
      Order order = em.find(Order.class, orderId);
      em.remove(order);
   }
}
```

Persistence unit

Definisce l'insieme di tutte le entità gestite dalle istanze di EntityManager

L'insieme di entità rappresenta i dati contenuti in un singolo data store

Definite nel file **persistence.xml**

Relazioni

Possono essere:

- One-To-One
- One-To-Many
- Many-To-One
- Many-To-Many

Le relazioni possono essere unidirezionali o bidirezionali

Definite con annotazioni es. **@OneToOne**

Enterprise Java Beans Query Language

Sottoinsieme di SQL92

Estensioni che permettono la navigazione su relazioni definite nello schema astratto del JAR

EJB QL queries definite nel deployment descriptor ejb-jar.xml

Le clausole supportate sono:

- SELECT
- FROM
- WHERE
- ORDER BY

Java Persistence API (JPA)

API per la persistenza di oggetti Java su database relazionali (object-relational mapping)

A partire da Java EE 5

Persistence Provider → Implementazione delle JPA (es. Hibernate, EclipseLink, OpenJPA, ...)

Java Transaction API (JTA)

Incluse in J2EE

Permettono di eseguire transazioni distribuite → Un'applicazione può eseguire transazioni su più DB contemporaneamente

Transaction manager → Schedula in modo efficiente le transazioni (potenzialmente molte) nel server

La data source deve essere registrata presso il server utilizzando il JNDI name

Le transazioni possono essere:

- Container Managed
- Application Managed

JAVA MESSAGING SERVICE (JMS)

Middleware per lo scambio di messaggi asincroni

Permette comunicazione distribuita loosely coupled

Mittente e ricevente non devono essere disponibili nello stesso momento per comunicare

Mittente e ricevente devono solo conoscere il fomrato del messaggio e che destinazione utilizzare

Comunicazione:

- Asincrona → Un provider JMS può consegnare messaggi ad un client quando arrivano; il client non ha bisogno di richiederli
- Reliable → L'API JMS può assicurare che il messaggio sia consegnato solo una volta

Utilizzabile quando:

- Le componenti non dipendono dalle interfacce di altre componenti
- L'applicazione gira indipendentemente dalle altre componenti
- Una componente può inviare informazioni ad altre continuando ad operare senza ricevere risposte immediate

In J2EE gli application clients (EJB o web components) possono inviare o ricevere in modo sincrono e ricevere in modo asincrono

Message-driven beans → EJB in grado di ricevere messaggi asincroni, permettono ai JMS provider di implementare il concurrent processing dei messaggi

Le operazioni **send** e **receive** possono partecipare a transazioni distribuite

Architettura JMS

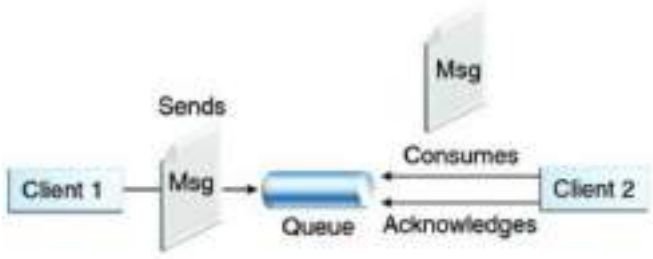
JMS provider → Sistema per la gestione dei messaggi che implementa le interfacce JMS e fornisce funzionalità di amministrazione e controllo

JMS client → Programma o componente scritto in Java che produce e consuma messaggi

Messaggi → Oggetti che comunicano informazioni tra JMS clients

Administered objects → Oggetti JMS preconfigurati creati da un amministratore per l'uso dei clients → destinations e connection factories

Point-to-point



Ogni messaggio ha un solo consumatore

No dipendenze temporali tra mittente e destinatario

Il destinatario invia un acknowledgement per segnalare che il messaggio è stato processato correttamente

Ormai sostituito dall'uso di web services asincroni

Publish/Subscribe

Ogni messaggio può avere più consumatori

Un client che sottoscrive un argomento può consumare solo messaggi pubblicati dopo la sottoscrizione

Il subscriber deve essere attivo per consumare i messaggi

Ancora utilizzato perché non esiste una modalità broadcast nei web services

Topic → Meccanismo di distribuzione per la pubblicazione di messaggi inviati a più destinatari

Destinatari conosciuti solo dal topic e non da chi invia i messaggi

Message-Driven Bean

Deve implementare le interfacce **MessageDriveBean** e **MessageListener**

La classe deve essere definita public e non abstract né final

Deve implementare i metodi **onMessage**, **ejbCreate** ed **ejbRemove**

Deve contenere un costruttore pubblico con nessun argomento

NON deve definire il metodo **finalize**

onMessage

Il tipo di ritorno deve essere void

Un solo parametro di tipo javax.jms.Message

Invocato nello scope di una transazione determinata dall'attributo specificato nel deployment descriptor

JAVA SERVER FACES

Tecnologia Java server-side, basata sul design pattern architetturale Model-View-Controller (MVC), il cui scopo è quello di semplificare lo sviluppo dell'interfaccia utente (UI) di una applicazione web

Composto da:

- API per:
  - Rappresentare componenti UI e gestire lo stato
  - Gestire eventi
  - Validare server-side
  - Definire la navigazione tra pagine
- Due librerie per esprimere componenti UI all'interno di una JSP e per connettere le componenti agli oggetti server-side

Separazione tra logica e presentazione

Utilizzo di componenti UI senza legarsi a particolari tecnologie di scripting o linguaggi di markup

APACHE STRUTS

Framework open source per lo sviluppo di applicazioni web su piattaforma Java EE

Estende le Java Servlet, incoraggiando gli sviluppatori all'utilizzo del design pattern Model-View-Controller (MVC) permettendo lo sviluppo di web application di notevoli dimensioni

SCM

SOFTWARE CONFIGURATION MANAGEMENT

Perché il Software Configuration Management?

Diversi problemi durante la produzione di software:

- Più persone lavorano sullo stesso software che è in continuo cambiamento
- Più di una versione del software deve essere supportata
- Il software deve essere eseguito su macchine e sistemi operativi diversi

Necessità di coordinazione!

Il Software Configuration Management aiuta a gestire l'evoluzione del software e a controllare i costi legati alle modifiche del sistema

Parte di un processo più generale di **quality management**

Promotion → Lo stato interno dello sviluppo di un software viene modificato

Release → Un set di modifiche viene distribuito fuori dal team di sviluppo

Ruoli nel SCM

Configuration manager → Identifica i **configuration items** e definisce le procedure per creare promotions e releases

Change control board member → Approva o rifiuta le richieste di modifica tenendo in considerazione l'impatto della modifica sulle altre componenti e sulla stabilità del sistema

Developer → Crea promotions attraverso richieste di modifica o normale attività di sviluppo, controlla le modifiche e risolve i conflitti

Auditor → Esperto esterno che effettua una review del codice, non fa parte del team, seleziona e valuta le promotions da rilasciare e assicura la consistenza e la completezza della release

Identificazione degli item

Grandi progetti tipicamente producono migliaia di documenti che devono essere identificati univocamente

Alcuni sono necessari per tutto il ciclo di vita del software

È necessario definire un **sistema di denominazione dei documenti** in modo tale che documenti correlati abbiano nomi correlati → Schema gerarchico

Change management

Gestione delle richieste di modifica

Richiesta di modifica → Nuova release

In generale una modifica viene gestita come segue:

1. Richiesta di modifica
2. La richiesta di modifica viene confrontata con il goal del progetto
3. La richiesta viene accettata o rifiutata
4. Se la richiesta è stata accettata, viene assegnata a uno sviluppatore e implementata
5. La modifica implementata viene valutata

La complessità della gestione delle modifiche varia da progetto a progetto:

- Progetti piccoli → Richieste veloci e informali
- Progetti complessi → Richieste dettagliate e formali, approvazioni ufficiali

Change policies

Per ogni nuova promotion o release viene applicata una change policy

Servono per garantire che ogni versione, release, ... sia conforme a determinati criteri

Es. *"No developer is allowed to promote source code which cannot be compiled without errors and warnings."*

Change request form

Serve per specificare:

- Modifica necessaria
- Propositore della modifica
- Ragioni della modifica
- Urgenza della modifica

La risposta è composta da:

- Valutazione della modifica
- Analisi dell'impatto
- Costo della modifica e raccomandazioni

Derivation history

È necessario tenere traccia dello status modifiche → **Derivation history**

Dovrebbe registrare:

- Modifiche effettuate
- Ragioni delle modifiche
- Chi ha effettuato la modifica
- Quando è stata effettuata la modifica

Configuration management plan

Definisce chi è responsabile per le procedure di CM, i documenti da gestire, lo schema di denominazione dei documenti e le policies

Descrive gli strumenti da utilizzare e le limitazioni al loro uso

Da definire prima di tutte le modifiche!

VERSION CONTROL

"The giant UNDO button"

Strumenti per tenere traccia delle modifiche, identificare release e unire modifiche concorrenti

Es. CVS, Subversion, **git**, REPO, Mercurial, Clearcase, ...

Subversion

Due obiettivi:

- Record keeping → È possibile ritornare indietro a version precedenti
- Collaboration → Più sviluppatori possono lavorare contemporaneamente allo stesso progetto

Revision → Modifica committed di un file o di un set di file, "snapshot" del progetto in un determinato istante

Repository → Master copy dove il sistema memorizza l'intera revision history del progetto

Working copy → Copia locale del progetto su cui uno sviluppatore sta effettuando delle modifiche, generalmente sono presenti più working copy (una per sviluppatore)

Checkout → Richiesta di una working copy dal repository (corrispondente allo stato attuale del repository)

Commit → Invio di modifiche dalla working copy al repository

Log message → Commento allegato al commit che descrive le modifiche

Update → Aggiornamento della propria working copy con le modifiche committed su repository da altri sviluppatori

Conflict → Situazione in cui due sviluppatori cercano di fare commit della stessa porzione dello stesso file, deve essere risolto dagli sviluppatori

WEB SERVICES

SERVICE ORIENTED ARCHITECTURE (SOA)

Web services nati principalmente per facilitare l'integrazione di componenti software

Componente software accessibile dalla rete attraverso dei protocolli standard

Il web è passato dall'essere **user centric**, ovvero nel quale è sempre l'utente a scatenare l'interazione tra le componenti, ad **application centric**, nel quale le applicazioni dispongono di tanti servizi che comunicano tra loro a prescindere dall'utente

Interfaccia pubblicamente accessibile (possibilmente scritta in XML)

Un web service è:

- Accessibile in rete
- Indipendente da OS e linguaggi di programmazione delle applicazioni
- Facilmente trovabile → Attraverso dei **registry**, divisi per argomenti
- Descritto via XML
- Utilizza messaggi standardizzati in XML

"Encapsulated, loosely coupled, contracted software objects offered via standard protocols"

Encapsulated → L'implementazione è invisibile alle entità fuori dal servizio che vedono solo l'interfaccia

Loosely coupled → Il servizio e gli utenti che lo usano possono essere disegnati in modo indipendente

Contracted → Una descrizione del servizio è disponibile all'utilizzatore

Standards

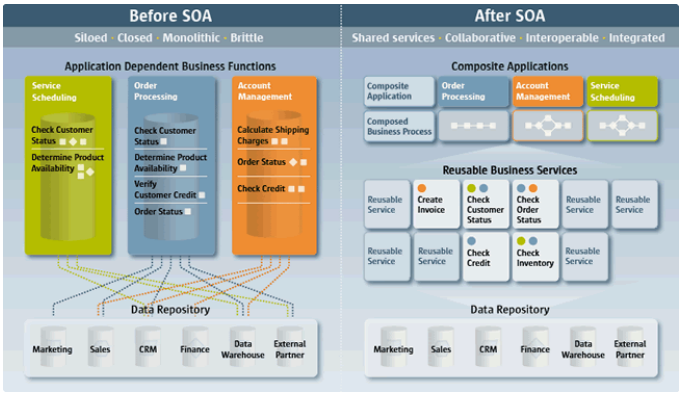
Simple Object Access Protocol (SOAP) → Protocollo di richiamo di procedure remote come web services

Web Service Description Language (WSDL) → Linguaggio di definizione dei web services

Universal Description, Discovery and Integration (UDDI) → Protocollo per ricercare i web services

I protocolli utilizzati per implementare i web services sono comunemente TCP/IP, HTTP, XML, SOAP, WSDL, UDDI

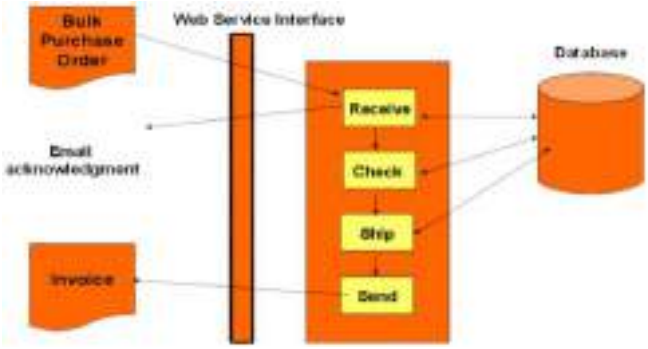
Prima e dopo



Tipologie di web services

RPC-style WS → Interazione sincrona, data una richiesta si ottiene una risposta immediata

Document-style WS → Interazione asincrona

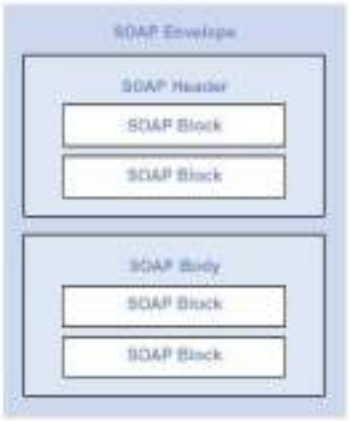


WEB SERVICES SOAP

Formato dei messaggi SOAP

SOAP Header → Dati opzionali sulla chiamata stessa usati dai nodi intermediari

SOAP Body → Dati delle chiamate e/o risultati di ritorno



WEB SERVICES DEFINITION LANGUAGE (WSDL)

Permette a un web service di descrivere se stesso tramite XML

Elementi che descrivono:

- Funzioni disponibili
- Data types di richieste e risposte
- Informazioni sul protocollo di trasporto
- Informazioni in merito all'indirizzo dove è localizzato il servizio

Elementi principali

Types → Contenitore per definizione di tipi

Message → Definizione astratta dei dati che vengono comunicati

Operation → Descrizione astratta di un'operazione supportata dal web service

Port type → Set astratto di operazioni supportate

Binding → Protocollo e formato dei dati per un determinato port type

Port → Endpoint definito da un insieme di binding e indirizzi di rete

Service → Collezione di endpoint correlati

Esempio di WSDL

Parte astratta

Parte concreta

Svantaggi

Nessuna informazione su chi fornisce il servizio

Il servizio è a pagamento o gratuito?

Quality of Service?

Quali altri servizi sono forniti dallo stesso web service?

UNIVERSAL DESCRIPTION, DISCOVERY AND INTEGRATION

Permette di trovare i WS, identificati in modo univoco

Dati salvati in XML

UDDI può essere utilizzato per pubblicare web services o per cercare web services

Architettura tecnica

UDDI data model → XML schema che descrive i web services

Sono definite quattro tipologie base di informazioni:

- Business information → Nomi, informazioni sul contatto
- Business service information → Descrizione tecnica del WS
- Binding information → Informazioni necessarie per l'invocazione del servizio
- Service specification information → Associa le binding informationn con le business service information che il servizio implementa

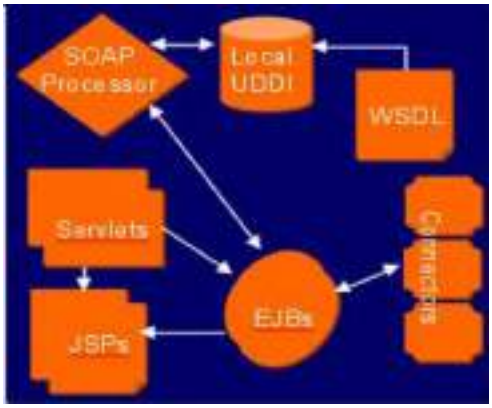
UDDI API → SOAP-based API per pubblicare e cercare dati UDDI

UDDI cloud services → Siti che implementano i registri e vengono aggiornati regolarmente

Esempio di UDDI

```
<businessEntity businessKey="35AF7F00-1419-11D6-A3DC-000C0E00ACDD"
authorizedName="0100002CAL"
 operator="www-3.ibm.com/services/uddi">
  <name>BooksToGo</name>
  <description xml:lang="en">
   The source for all professional books
  </description>
  <contacts>
   <contact>
        <personName>Ramesh Mandava</personName>
        <phone>(877)1111111</phone>
   </contact>
  </contacts>
</businessEntity>
```

WEB SERVICES IN JAVA



In Java si sceglie sostanzialmente di "esporre" una parte degli EJB al pubblico mediante un'interfaccia WSDL

È a discrezione del programmatore decidere cosa esporre

RESTFUL WEB SERVICES

REpresentational State Transfer

Insieme di principi e architetture di rete per sistemi distribuiti

Spesso il termine è utilizzato per descrivere ogni semplice interfaccia che trasmette dati con HTTP senza l'introduzione di un livello aggiuntivo (come in SOAP)

Utilizzo di metodi POST, GET, PUT e DELETE

Il tipo di ritorno può essere scelto dal programmatore → Es. JSON, XML, ma anche tipi complessi

Es. RSS, Twitter, Flickr, ... espongono dati utilizzando REST

Esempio di human centric web, nel quale l'utente si collega al browser e ottiene dei servizi

Sebbene i webservices RESTFul siano molto più leggeri di quelli SOAP, non sono adatti qualora sia necessaria un'integrazione con i sistemi legacy

Il loro scopo principale è quella di fornire informazioni "semplici", senza il processamento di metodi complessi (come RMI)

Principi di design

Client-server → Un'insieme di interfacce separa client e server in modo chiaro, i due possono essere sviluppati in modo indipendente se l'interfaccia non cambia

Stateless → Il contesto del client non deve venire memorizzato sul server, ogni richiesta contiene tutte le informazioni per richiedere il servizio, se necessario lo stato della comunicazione può essere mantenuto sul server attraverso un altro servizio che utilizza, ad esempio, un DB

Cacheable → I client possono fare caching delle risposte (se definite cacheable)

Layered system → Server intermedi (invisibili al client) possono migliorare la scalabilità e la sicurezza, ma il client deve conoscere solo l'indirizzo per la richiesta e il formato della risposta

Code on demand (opzionale)→ I server possono temporaneamente estendere le funzionalità del client trasferendo codice eseguibile (es. applet)

Uniform interface → Interfaccia di comunicazione omogenea tra client e server

Linee guide per gli URI

Le risorse vengono identificate attraverso degli URI

Gli URI dovrebbero essere intuitivi e dovrebbero nascondere il tipo di tecnologia server-side utilizzata

Scritti tutti in minuscolo

Spazi sostituiti da _

Nascondere i parametri dalla stringa di richiesta

Se l'URI non conduce a niente, fornire una pagina di default (no 404 Not Found)

REST vs SOAP

REST	SOAP
Uno stile	Uno standard
Trasporto solo con HTTP/HTTPS	Trasporto solitamente con HTTP/HTTPS, ma anche altro
Risposte solitamente come XML, ma anche altro	Risposte solo come XML
Richiesta trasmessa come URI (leggera, ma limiti di lunghezza)	Richiesta trasmessa come XML
L'analisi del metodo e dell'URI mostra l'obiettivo della richiesta	Bisogna analizzare il payload del messaggio per capire l'obiettivo della richiesta
Facilmente chiamabili da JavaScript	Difficilmente chiamabili da JavaScript
Offre risorse (dati) mediante un formato codificato (JSON, XML, ...)	Offre un vero e proprio servizio (potenzialmente complesso)

JSON

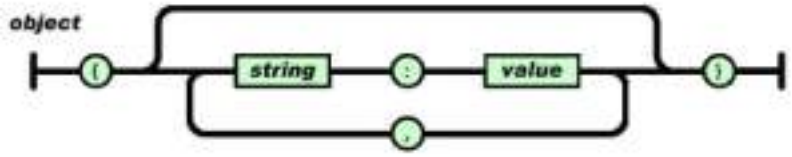
Formato che consente, attraverso l'uso di un'apposita libreria, di trasformare un oggetto in una stringa che può essere inviata, per poi essere trasformata nuovamente in un oggetto

Uno degli usi più frequenti di JSON è quello di sfruttarlo in un web service REST per lo scambio di informazioni con le applicazioni mobili

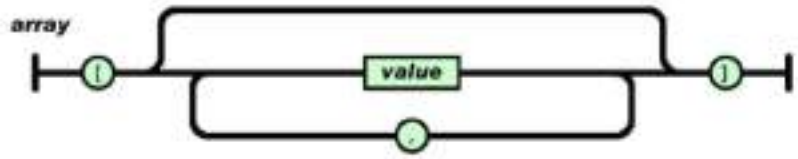
Strutture principali

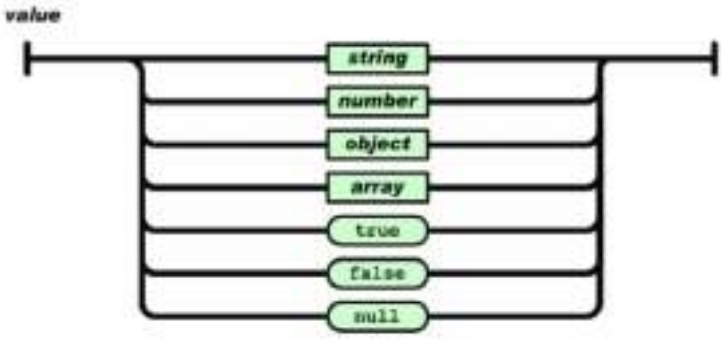
Basato su due strutture:

- Collezione di coppie nome/valore



- Lista ordinata di valori





Rispetto a XML no validazioni, né namespaces

CLOUD COMPUTING

DEFINIZIONE NIST

Fornitura di risorse computazionali, software, di accesso ai dati e di memorizzazione distribuite in rete che non richiedono all'utente finale una conoscenza della locazione fisica dell'hardware utilizzato e della configurazione del sistema che fornisce i servizi

"Una serie di tecnologie informatiche orientate a fornire servizi(hardware e software) attraverso la rete"

Nati in seguito allo sviluppo di banda larga, Wi-Fi e device mobili → Il browser diventa il personal computer

Illusione di infinita potenza computazionale

Eliminazione del grosso impegno iniziale da parte del cliente

Possibilità di pagare potenza computazionale a breve termine in base alle esigenze

Software scalabile in alto e in basso

Licenze "pay per use"

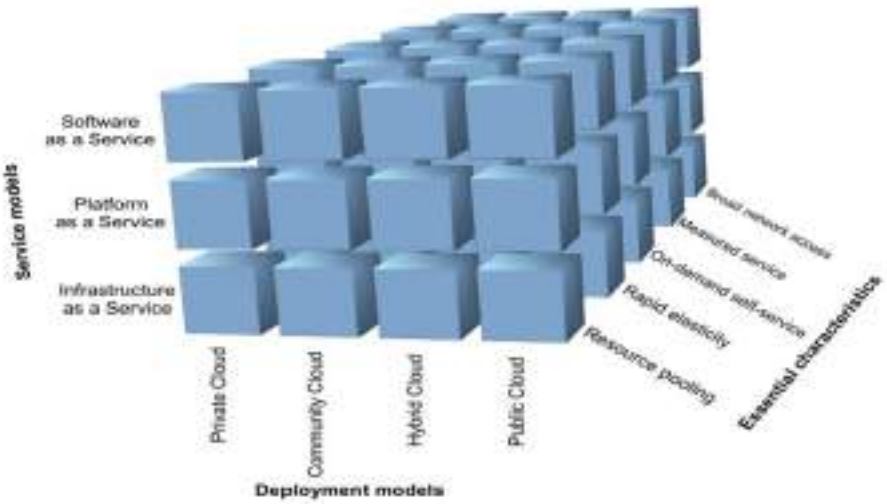
Software di infrastruttura con contabilità preinserita

Hardware in unità maggiori, con consumo di energia proporzionale all'uso

Origine

Costruzione di enormi **Data Center** per ridurre i costi fissi di hardware, elettricità, ...

Sfruttamento ottimale dei server → Statistical multiplexing



Modelli di servizio

Infrastructure as a Service (IaaS) → Vengono fornite risorse computazionali, spazio di memorizzazione, rete fisicamente o attraverso VM, il consumatore non ha il controllo dell'infrastruttura fisica, ma può eseguire software liberamente (inclusi sistemi operativi)

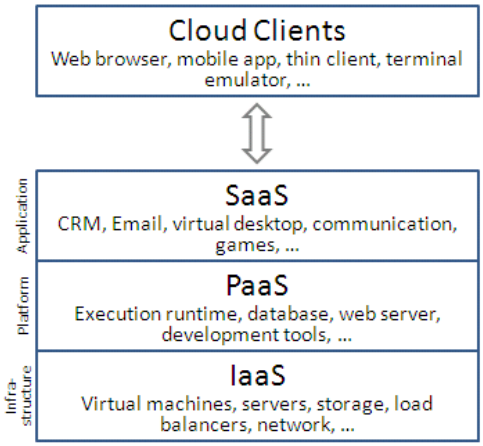
Es. Microsoft Azure, Google Compute Engine

Platform as a Service (PaaS) → Oltre all'infrastruttura, viene fornito un ambien te di deployment, l'utente non ha il controllo sull'infrastruttura e sul sistema operativo, ma sul software distribuito e sulla configurazione dell'ambiente in cui il software viene distribuito

Es. OpenShift, Google App Engine, Amazon EC2

Software as a Service (SaaS) → Viene fornito al cliente un software che viene eseguito su una infrastruttura cloud, l'applicazione è accessibile da diversi devices, solitamente attraverso un client web

Es. Google Drive, Dropbox, iCloud



Esistono due tipologie aggiuntive di servizi

Data as a Service → Vengono messi a disposizione dati a cui gli utenti possono accedere come se fossero residenti sul disco locale

Hardware as a Service → I dati vengono inviati al cloud, elaborate e poi restituiti all'utente, risorse assegnate a prescindere dall'utilizzo effettivo e non su richiesta

Modelli di distribuzione

Private cloud → Infrastruttura cloud a disposizione di una sola azienda, può essere gestita dall'azienda stessa o da terzi (es. cloud delle banche)

Community cloud → Infrastruttura condivisa da più organizzazioni che hanno le stesse esigenze (es. cloud che permette di accedere a sistemi operativi obsoleti, in questo modo le aziende possono eseguire programmi legacy senza dover mantenere macchine obsolete)

Hybrid cloud → Composizione di due o più cloud, alcuni dati vengono mantenuti privati, altri pubblici, in questo modo si aumenta la portabilità di dati e applicaizoni

Public cloud → L'infrastruttura è accessibile al pubblico che però non sa dove vengono ospitati i dati

Caratteristiche principali

Broad network access → Le risorse sono disponibili in rete e l'utente può accedervi attraverso meccanismi standard da dispositivi eterogenei

Measured service → Le risorse vengono automaticamente controllate e ottimizzate, l'utilizzo delle risorse può essere monitorato e riportato sia all'utente che al fornitore del servizio

On demand self-service → L'utente può richiedere le risorse automaticamente in base alle esigenze senza bisogno di interazione umana con il fornitore del servizio

Resource pooling → Le risorse sono organizzate in pool che servono più utenti e vengono assegnate in modo dinamico, l'utente in generale non ha controllo su quali risorse vengono fisicamente utilizzate, ma in alcuni casi può porre vincoli ad alto livello (es. paese o stato)

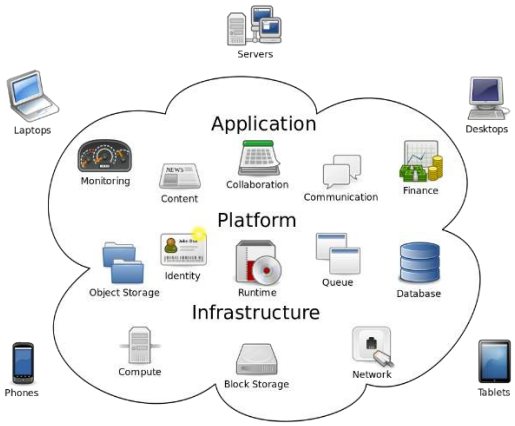
Rapid elasticity → La capacità di calcolo, memorizzazione, … può essere aumentata o diminuita velocemente in base alle esigenze

Architettura

Due livelli interconnessi attraverso la rete

Front-end → Devices e applicazioni con cui si accede al servizio

Back-end → insieme di Data Center



Cloud computing

Problemi

- Disponibilità e continuità del servizio
- Dati "bloccati
- Privatezza e verificabilità dei dati
- Rallentamenti nel trasferimento dei dati
- Imprevidibilità delle prestazioni
- Scalabilità dello storage
- Errori nel software
- Problemi internazionali di tipo economico e politico → Non si sa dove risiedano i dati
- Pagamento licenze software
- Difficoltà di migrazione in caso di cambio di gestore dei servizi

Quali applicazioni su cloud?

- Mobili interattive
- Elaborazione batch parallela di molti dati
- Analisi dei dati
- Applicazioni scientifiche CPU intensive
- Applicazioni grafiche
- Apps di office online

GOOGLE APP ENGINE (GAE)

Piattaforma per sviluppare applicazioni web su cloud

Indirizzato alle applicazioni web, inadatto ad applicazioni generiche

Separazione tra logica applicativa e storage

Scalabilità automatica ed alta disponibilità

Dynamic Web Server con supporto alle comuni tecnologie web

Integrazione con gli acocunt Google attraverso opportune API

SDK per sviluppo locale

Distribuzione rapide e semplice

Console di amministrazione web-based

Data store

Object database non relazionale

Supporto per query e transazioni

Tecnologia basata su Google File System → **Big Table**

Java Persistence API (JPA) e Java Data Object (JDA) implementati con DataNucleus