

▼ Lezione 4 11/03/2024

Riprendiamo a parlare dell'approccio **waterfall** e le sue attività chiave.

Per molto tempo si è pensato che queste attività dovessero essere fatte come nelle altre ingegnerie più mature, cioè seguire l'approccio a progetto anche per il software.

In un approccio **waterfall** la **comunicazione** tra le attività avviene in maniera **formale** e **idealmente senza feedback**. Con il tempo ci si è accorti che quasi tutti i costi venivano spesi per l'ultimo step, cioè **mantainance**.

Lehman ha affermato che per il software **NON** si parla di **manutenzione** ma di **evoluzione**.

L'entropia dei programmi cresce inevitabilmente con il passare del tempo.

Discipline basate su progetto

Quando si progetta un ospedale o un ponte, la prima fase è identificare un insieme di requisiti.

Questi requisiti possono includere:

- il numero di malati da ospitare
- la quantità di mense necessarie
- il numero di stanze richieste
- altri aspetti fondamentali che vanno dai requisiti funzionali a quelli non funzionali, come l'approvvigionamento di acqua ed elettricità.

Una volta raccolti questi requisiti, vengono documentati in un report che costituirà la base del contratto tra il cliente e il team di progettazione o costruzione.

Da qui, molte metodologie seguono un approccio a cascata, noto anche come **waterfall**. Questo modello **organizza le attività in una sequenza lineare**, dove ognuna si basa sui risultati della fase precedente. È importante notare che la cascata non è solo una metodologia organizzativa, ma rappresenta anche una comunicazione formale senza feedback diretto tra il cliente e il team di progettazione o sviluppo.

Per anni, il lavoro ha utilizzato l'approccio a progetto per il software, ma negli ultimi decenni si è fatta avanti l'idea di software orientato al prodotto.

Oggi, nella maggior parte dei casi, un software viene distribuito come se fosse una piattaforma che implementa processi e offre punti dove si possono iniettare dei behaviour personali ⇒ **product based software engineer**

Le due ingegnerie convivono.

Esempio: SAP ERP (un sistema enterprise resource planning prodotto da SAP AG)

SAP ERP lo si compra attraverso un fornitore di servizi che lo configura.

In generale, ci sono due categorie di approccio di tipo **product-based**:

- di tipo **consumer**: ad **esempio** compro un videogioco e lo uso così com'è

- di tipo **enterprise**: aziende che delegano altre attività ad altre aziende (out sourcing), un **esempio** è la logistica

La differenza tra **product** e **project** sta nella **figura** che ha i requisiti

Nel **project-based** i requisiti sono controllati dal **customer**

Nel **product-based** no, ma è lo **sviluppatore** che cattura conoscenza e decide cosa potrebbe essere utile per i clienti

L'approccio **product-based** è usato dagli **startupper** e dalle aziende che fanno analisi di dominio e progettano software rispetto ad quel dominio applicativo.

Con quest'approccio, sono le **features** che danno vita ad un prodotto

Si è sviluppato anche il concetto di **product lines**, cioè i prodotti condividono lo stesso core, ma possono essere personalizzati in base alle esigenze di ogni cliente

Product Vision

È il primo step per l'approccio a prodotto

Describe ad altissimo livello l'area di business dove si vuole inserire la propria attività.

In una vision devono esserci le risposte a 3 domande fondamentali:

- **Qual è il prodotto che voglio sviluppare?**
- **Per chi lo voglio fare? (qual è il target?)**
- **Cosa offro in più rispetto ai competitor? (in cosa sono unico)**

Quando si parte da una vision, è difficile da prevedere sul lungo termine la direzione che prenderà il sistema, mentre se si parte dai requisiti di un customer è diverso.

Il punto di partenza per un **product-based** software è la **vision**, mentre per il **project-based** è un **requirement**

Come si definisce una vision?

Le sorgenti di informazioni da analizzare sono:

- Intuito o Fantasia
- Domain Experience
- Product Experience
- Customer Experience
- Prototyping and "playing around"

Oltre a cambiare il punto di partenza, **cambia** anche l'**organizzazione** e il **processo** da seguire

Nelle aziende **product-oriented** nasce una figura nuova, cioè il **product manager** cioè colui che tiene insieme :

- il business
- la tecnologia
- i feedback dei clienti

È una figura nuova, a breve seminario con Severini.

La figura del product manager è un profilo molto skillato (key responsibilities)

Molte aziende software sono organizzate intorno ad un product manager

Questa è una figura che sta diventando sempre più importante

Deve gestire la vision del prodotto, il backlog, i feedback degli utenti i quali diventeranno input del processo di sviluppo

Back-log: insieme di features **NON** ancora implementate (magari nemmeno immaginate), ogni ciclo di sviluppo dà priorità al back-log, cioè cosa deve esserci nella prossima release

Non sempre ogni ciclo di sviluppo verso una nuova release implementa delle nuove feature del back-log.

Nasce l'idea di un sistema che è in continuo sviluppo (agile development)

Prototipazione

È fondamentale per product-based SE

Esempio: la maggior parte dei tool che usiamo è in beta

Tipicamente la prototipazione serve a **dimostrare la fattibilità e come funzionerà** il software al cliente

Esempio: una delle cose più difficili da progettare quando si parla di app mobile è il flusso delle interfacce

meme lettore su meme pc gaetano

Approccio Agile (piccola introduzione)

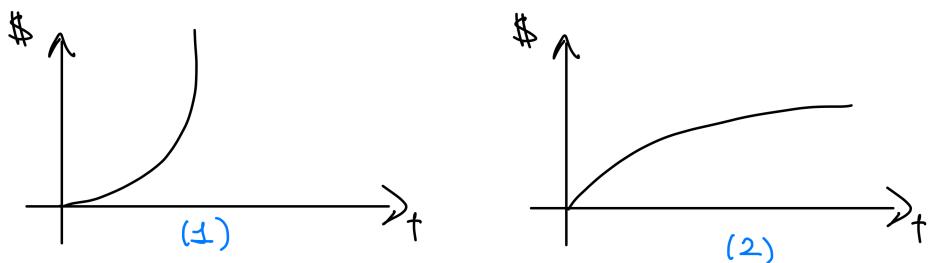
Lo sviluppo del prodotto viene fatto in ambiente aperto, cioè che consente di cambiare la direzione più volte.

Elapsed time: è il tempo che trascorre tra la sedimentazione di un errore e la scoperta dello stesso.

Questa è stata la rovina di generazioni di sviluppatori, in quanto se l'errore viene scoperto in tempi brevi allora si avranno dei costi di riparazione minori

Prima si parlava di manutenzione preventiva, cioè con il passare del tempo si pensava che la **curva costo-tempo potesse convergere** ad un valore, ma era necessario fare un buon progetto e avere una base architettonica stabile che permetta di tenere separate le varie funzionalità così da tener sotto controllo lo sviluppo.

L'idea era che se l'errore **NON** veniva scoperto in tempo, allora il grafico sarebbe stato del tipo **(1)**, altrimenti **(2)**



Il processo che useremo è figlio del cambiamento \Rightarrow da modelli a cascata allo sviluppo per spike

▼ Lezione 5 15/03/2024 (Accenno di Agile SE, Version Control)

Abbiamo fatto una breve disamina dei concetti fondamentali dell'ingegneria del software.

Oggi vediamo **come organizzare il lavoro del team**.

Dobbiamo darci una struttura organizzativa, abbiamo già differenziato tra project based e product based.

C'è differenza nell'**organizzazione** per questi due approcci?

Storicamente sì, adesso non più.

Come già detto l'ingegneria del software ha cercato di organizzarsi come le ingegnerie più mature, ma con il passare del tempo ha cambiato strada.

La **comunicazione** era **formale**, fatta con **documenti**.

Ciò ha portato ad una **struttura monolitica (monumentale)** e ingessata, dove l'idea era che se si ha a disposizione abbastanza tempo, risorse e denaro per fare analisi dei requisiti e progettazione, allora il sistema sarà migliore (ciò è proprio quello che succede nelle ingegnerie tradizionali)

Questo approccio aveva come obiettivo il **plan-driven**, cioè si lavorava seguendo un **piano organizzativo**

In particolare, tutto ciò era fatto per prevenire il cambiamento perché c'era come riferimento il grafico **"elapsed time"** (in alcuni casi si ha ancora oggi perché esistono delle organizzazioni rigide)

L'idea era "se si **accetta il cambiamento** faccio **esplodere i costi**" ⇒ l'**approccio preventivo NON funziona**

Agile software Engineering

Metafora **rafting**: per poter arrivare fino in fondo nella discesa di un torrente in piena c'è un unico modo, ovvero quello di avere un team affiatato, competente e capace di reagire ai cambiamenti delle correnti

L'approccio **agile** quindi prevede di avere un **team affiatato**, che si comprende, **competente e capace di reagire ai cambiamenti continui** e **NON** che **previene** e cura l'ipotesi del **cambiamento** ma che vive con esso, cioè capace di reagire al cambiamento.

Ancora una volta non parliamo di tecnologia, ma si è potuto andare in questa direzione perché essa si è evoluta, cioè si è costruita una grande conoscenza dell'architettura.

Oggi siamo **agili** o tendiamo all'agilità perché l'ingegneria del software **NON** è più **monolitica**, non c'è più la necessità di partire dal foglio bianco ogni volta che si inizia un progetto ma si parte da componenti, API, ecc...

Qual è dal punto di vista storico il passaggio che ci ha portato al modello agile?

Abbiamo già detto che il modello in passato era di tipo waterfall (anche se formato da blocchi diversi, sempre quello era) con ultimo blocco la maintenance.

Il modello **waterfall** è ideale per chi deve fare pianificazione perché è lineare (consente di avere semplici relazioni additive)

Questo modello, nell'ambito dell'ingegneria del software, **NON funziona perché NON spiega la realtà**, un modello è utile se appunto spiega la realtà e se è uno strumento che supporta chi deve prendere decisioni.

Se un modello spiega la realtà deve tener conto di come effettivamente evolvono le cose, ci aspettiamo che i blocchetti abbiano un ruolo nel pezzo di realtà che devono spiegare e che abbiano lo stesso peso.

I **chaos report** sono la fotografia dello stato di salute del software

Esistono **3 famiglie di progetto**:

- **di successo** ⇒ vengono completati nel tempo stimato e con il budget stimato
- **project challenged** ⇒ arrivano in fondo ma sono over budget e over time
- **progetti impaired** ⇒ progetti che dopo un certo numero di anni e milioni di euro vengono abbandonati

Circa l'80% dei progetti di ingegneria del software rientra nelle categorie challenged e impaired

Lo studio ha notato che in media la metà dei progetti ha sbagliato la pianificazione, mentre il 30% non viene portato a compimento

Che maturità ha un settore che per ogni 10 progetti, 5 sono pianificati male e 3 non arrivano mai in porto?

Non è un problema di tecnologie (**no silver bullet**), ma è il concetto del **modello** che possa **prevedere tutto** che **NON funziona**

Infatti in passato si pensava che la prima fase di progettazione doveva essere indipendente dalla tecnologia (dato che il software doveva andare su un mainframe), **oggi invece si sceglie prima il modello architetturale.**

Il passaggio è stato:

Monumentale ⇒ Incrementale ⇒ Iterativo (Barry Boehm) ⇒ Agile o Reattivo

Negli approcci **monumentale e incrementale** c'è **molta fede nell'ingegnere**, cioè si pensa che se l'ingegnere *ingegna* bene allora il sistema funzionerà.

Nell'approccio **incrementale** nasce la figura dell'ingegnere *di domino* in grado di creare modelli dei processi e requisiti per un dominio.

Vantaggi del modello incrementale

Avere un modello di un dominio di riferimento portava a poter fare analisi di requisiti anche in un secondo momento senza generare feature/requirements in contraddizione tra loro.

Nasce così il concetto di **release** e di **release planning**.

Il **modello incrementale** differisce da quello a cascata perché **permette una migliore gestione delle risorse umane** in quanto si riesce a parallelizzare il lavoro di piccoli gruppi di persone

Approccio iterativo

L'idea è che un sistema può nascere prima che sia completata la fase di progettazione di tutte le funzionalità.

Un progetto software è **knowledge discovery**, man mano che sviluppo acquisisco conoscenza che poi diventerà software

Modello a spirale

Lo sviluppo di un progetto software **NON** ha una **fine**, si lavora con cicli e una volta che si arriva alla fine di un ciclo ne parte un altro.

Un sistema evolve seguendo delle fasi:

- Determinare gli obiettivi
- Valutazione delle alternative
- Sviluppo e verifica del next-level product
- Pianificazione nuova fase

Ogni volta che completiamo una spirale si termina una fase.

Il progetto quindi **NON finisce** ma **evolve**, i primi cicli della spirale si intendono come **knowledge analysis**

Il **modello agile** invece prevede la **pianificazione a brevi cicli e il coinvolgimento degli utenti**

Adesso l'invariante è la qualità, non più le features

Quando c'è il rilascio di una **release**, in **pipe** ci sono **già altre feature**.

Quindi non cambia solo il processo di realizzazione ma anche quello di analisi dei requisiti.

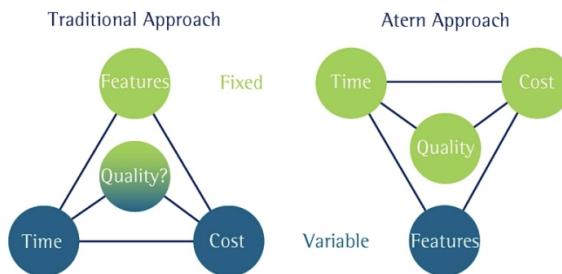
Come abbiamo detto le **aziende agili** usano solo un **backlog**

Prima si pensava che avere un backlog voleva dire essere in ritardo, mentre adesso si vive di backlog.

Dato che non abbiamo un cliente che ci commissiona il software, la sua parte la prende il **product owner** che effettua il **mining dei feedback**.

Spesso **alcune feature presenti nel backlog NON vengono più sviluppate** perché ci si accorge che non servono più dato che il sistema si è evoluto verso altre strade

Di conseguenza, è cambiato anche l'iron triangle



SOFTWARE VERSIONING

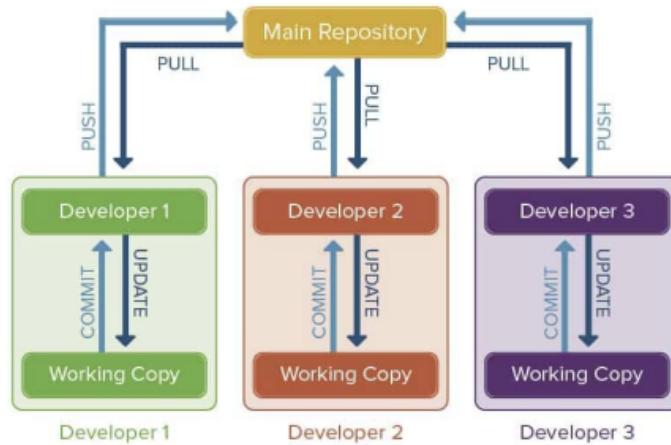
Questi concetti si applicano anche alla scrittura di testo semplice, **NON** solo al codice.

Gestire la versione dei progetti è importante

Il **version control** nell'ingegneria del software è un sistema **responsabile del controllo** delle **versione dei codici sorgente**, in pratica effettua una "fotografia" ad un determinato livello di sviluppo e consente di ritornarci in un secondo momento.

Molto spesso si utilizzano strumenti **remotizzati**, cioè più sviluppatori insistono sulla **stessa repository** e ognuno lavora con la propria copia.

Si **scaricano** le **modifiche** dalla repository remota, si effettuano **modifiche** e quando si è pronti si **pubblica**.



Anche google docs funziona con gli stessi concetti

Questo sistema dà molti vantaggi, ad esempio:

- cambiamenti reversibili (**damage mitigation**)
- sviluppo semplificato
- miglioramento della collaborazione e comunicazione del team

Terminologia:

- **repository**: è una struttura dati che memorizza metadati per un insieme di file o directory. I **metadati** includono uno storico dei cambiamenti nella repository, referenze ai **commit objects** (detti **heads**)
- **branch**: ramificazioni che si possono creare a partire da un ramo già esistente. Dopo aver effettuato un branch, le due linee di sviluppo create sono indipendenti, quindi possono essere sviluppate anche in maniera differente
- **trunk (o baseline, master, mainline)**: è la linea unica di sviluppo che **NON** è un branch
- **commit**: l'azione che permette di scrivere o fare il merge dei cambiamenti fatti dalla working copy alla repository locale
- **clone**: l'azione che permette di creare una repository a partire da un'altra già esistente
- **push**: l'azione che consente la pubblicazione delle modifiche sulla repository remota
- **pull**: l'azione che permette di scaricare i cambiamenti dalla repository remota alla propria locale
- **merge**: l'azione che permette di combinare i cambiamenti da branch differenti in uno unico, il sistema gestisce automaticamente l'unione
- **conflict**: si verifica quando sono stati effettuati diversi cambiamenti di uno stesso documento e il sistema non è in grado di riconciliarli automaticamente

- **head:** è l'ultimo commit fatto sul trunk o su un branch
- **tag:** è possibile taggare un determinato commit, ad esempio assegnando un nome oppure il numero della versione
- **blame:** effettua una ricerca per individuare l'autore dell'ultima modifica di una particolare linea di codice

Sviluppo ad Albero

Si parte da un ramo comune e man mano si creano rami dove ogni sviluppatore lavora in maniera indipendente e poi alla fine si riuniscono i rami.

Quindi, per fissare i punti nella storia si effettuano:

- **commit:** si salva la "fotografia" del codice in un determinato momento
- **revert:** si torna indietro se ci si accorge di un errore in un commit

Quando si effettua un **commit** è buona norma accompagnarlo con un **commento** dove si spiega ciò che si è stato fatto

Ogni nodo dell'albero è un commit

Il **cloning** è il **primo evento** che avviene quando si inizia a lavorare con una repository perché permette di fare una prima copia locale di essa, di solito è molto pesante.

Push e pull: ho la mia working copy ed effettuo un commit, quando faccio **push** chiedo alla repository remota di **pubblicare** queste modifiche, mentre con **pull** **scarico le modifiche** (proprie o di altri).

Il **merge** è un **commit** particolare, può generare conflitti

Esempio conflitto: due sviluppatori modificano la stessa riga di codice

Pull Merge Request: uno sviluppatore che effettua un pull merge request chiede al gestore della repository di approvare le sue modifiche. Quando ciò si effettua sul **main branch** si apre un thread perché le modifiche devono essere approvate.

GIT

È lo strumento per il **versioning** più famoso di tutti, inventato da Linus Torvalds per gestire lo sviluppo del codice.

Git **gestisce solo il version control**, cioè la working copy, è un client

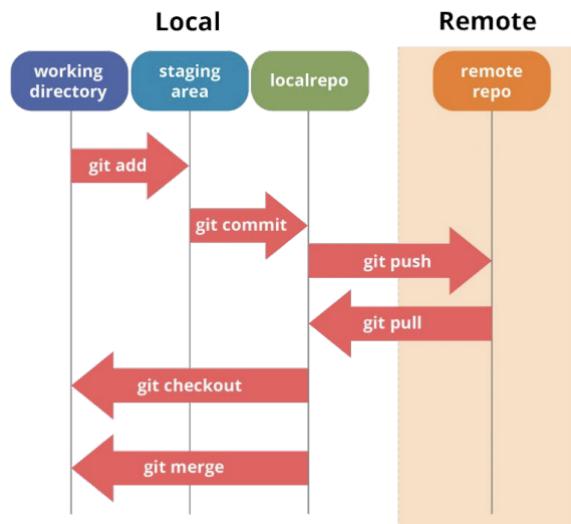
WORKFLOW

I file vengono modificati nella propria **working directory**

Dopo le modifiche, si possono scegliere quali devono essere oggetto del prossimo commit e si aggiungono alla **staging area**

Staging area: area di intermezzo tra working directory e local repository, in questa zona troviamo i file che sono stati modificati e che sono pronti per il commit sulla local repo.

Dopo aver fatto il commit, è possibile effettuare il push verso la remote repo



▼ Lezione 6 18/03/2024

Git è quindi uno strumento sia di version control sia di collaborazione

Soltanente, tendiamo a mettere sotto collaborazione tutti i file che contengono codice sorgente.

Dietro ad un commit potrebbero esserci varie organizzazioni, quindi non per forza una singola persona.

Per "strumento di version control" si intende uno strumento che permette di gestire l'**evoluzione** degli **item (file)**

Invece con "collaborare" si intende che si ha una **baseline** e questa viene divisa in **task**

A questo punto, si creano tanti branch su cui lavorano team differenti e alla fine si avrà la necessità di unire le modifiche (merge)

Git è anche uno strumento di **gestione del processo**

In pratica, Git permette di decidere cosa deve succedere quando avviene un push

Esempio:

1. si crea una macchina virtuale
2. il codice viene caricato e viene fatto il build (compilazione ed esecuzione dei casi di test)

3. se vengono superati i casi di test, allora si effettua il deploy sulla macchina che deve utilizzare il codice

Ritorniamo ai modelli di sviluppo

Abbiamo visto che in passato l'obiettivo principale erano le features, anche andando a discapito della qualità

Invece, oggi è la **qualità** che deve essere **invariante**

AGILE SOFTWARE DEVELOPMENT

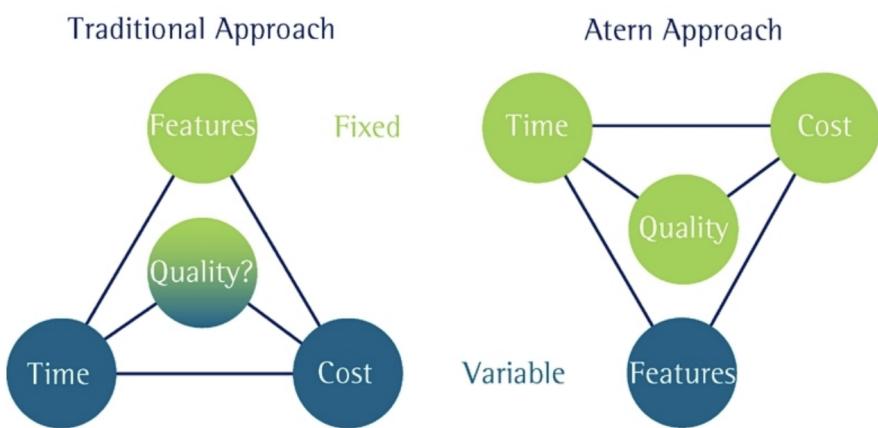
Il principio che viene messo in discussione è che un sufficiente planning con un approccio lineare può portare ad un software "buono" al primo tentativo.

Agile vuole costruire un approccio ingegneristico che **accetti il cambiamento** e che **sia capace di prendere decisioni basandosi sui dati attuali**, di **abbracciare il cambiamento**

Rivisitiamo quello che nella lezione 0 avevamo chiamato iron triangle (qualità-tempo-costo)

Ora mettiamo le feature nel gioco della pianificazione (cioè variano) e quindi l'approccio è capovolto.

Nell'approccio da effettuare l'**unica grandezza variabile** sono le **features**.



Cosa vuol dire avere "ferme" le altre due grandezze?

Un **modello agile** non fa altro che **ripetere la tradizione** (waterfall) **su un insieme ridotto di funzionalità** e attraverso degli spike di cambiamento.

Un **modello agile si evolve attraverso degli spike di cambiamento**, dove ognuno di questi è un prodotto ma non necessariamente destinato al cliente, infatti può essere sia un potentially production product (sul mercato) che una **milestone** (versione interna del sistema).

I processi si sviluppano su archi temporali di 4 settimane (6 max), quindi **cicli brevi**

Le prime release potrebbero essere tutte interne mentre le successive potrebbero alternarsi in interne e esterne.

Filosofia AGILE

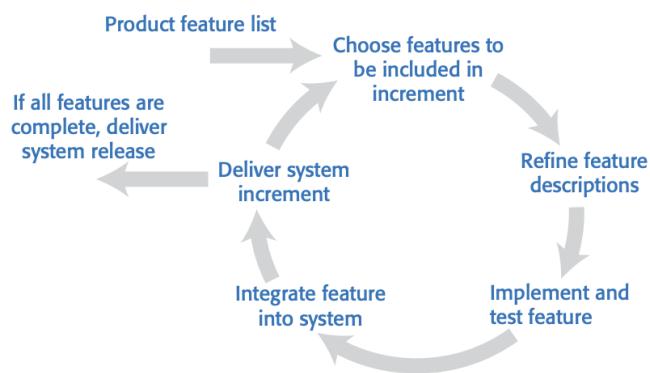
- Coinvolgere gli utenti del software il prima possibile
- Costruire un'organizzazione che faccia del cambiamento il suo punto di forza
- Utilizzare un'architettura che permetta lo sviluppo incrementale
- Mantenere la semplicità
- Avere il focus sulle persone (developer), non sulle cose

Perché l'ingegneria del software aveva abbandonato questi principi e adesso li inizia a sposare?

Perché non c'erano né le conoscenze né le tecnologie, il cambiamento era visto in maniera deleteria.

Processo AGILE

1. Scelta delle features da implementare
2. Raffinamento della descrizione delle features scelte
3. Implementazione e testing
4. Integrazione nel sistema
5. Delivery



Le features sono i **punti di ingresso** del ciclo di sviluppo

I modelli agili sono molti, i più famosi sono:

- **Extreme programming (XP)**, orientato all'**implementazione**
- **Scrum**, forse quello più applicato al momento, orientato alla **gestione** (management)

EXTREME PROGRAMMING (XP)

Modello ideato da Kent Beck

XP si focalizza su 12 tecniche di programmazione che permettono di sviluppare, cambiare e distribuire software in maniera rapida

Alcune di queste sono diventate molto utilizzate per lo sviluppo del software, altre un po' meno

Le 12 tecniche sono:

- **the planning game:** è alla base di tutti i metodi agili, se alla base del plan driven è il management che effettua la pianificazione, **con questo schema la pianificazione è il risultato della coordinazione di 3 figure che sono**
CLIENT - PRODUCT OWNER - TECNICI

L'idea è che esistono **user stories, use case, scenarios** (in approccio tradizionale ⇒ requisiti) che dicono cosa deve fare il sistema (**feature**)

A fronte di questa situazione devo fare determinate cose

~~Diventa un game, il cliente decide cosa vuole compatibilmente alle risorse messe nel progetto~~

Extreme programming non è un nome a caso, è stato scelto perché ha come filosofia il fatto che tutto ruoti intorno alla programmazione

- **small release:** i tempi del ciclo devono essere brevi, 4/6 settimane
- **metaphor:** utilizzare una metafora, altrove viene chiamato use case
- **simple design:** non fare preventive design, cioè non aggiungere cose non chieste, mantenere il focus su ciò che serve.
- **testing:** prima di scrivere il codice si scrivono i casi di test, in genere si scrivono dopo aver letto la metafora
- **pair programming:** programmare sempre in 2 e non da solo, vantaggio di fondo ⇒ keeping the knowledge
- **coding standard:** si hanno regole che definiscono i nomi delle variabili, i commenti, lo stile di indentazione e la lunghezza massima ragionevole di un metodo.
- **refactoring:** si intende intervenire su un pezzo di codice per cambiare la struttura lasciando invariato il funzionamento

Vengono fatti degli stand up meeting

Stand up perché deve essere scomodo avendo l'obiettivo che deve durare poco

I **requisiti diventano cards**, cioè delle **user stories**.

Gli sviluppatori scompongono le storie in task

User stories

Il **punto di partenza** del ciclo sono le user stories, possono essere scritte da:

- customer
- product owner (o suoi rappresentanti)

L'idea è di avere uno spacchettamento dei comportamenti degli utenti in singole unità elementari.

All'inizio del ciclo di sviluppo vengono scelte le storie da includere nella prossima release in base alle priorità dei customer

Refactoring

Tutti gli sviluppatori sono tenuti a **rifactorizzare continuamente** il codice non **appena vengono individuati possibili miglioramenti**.

In questo modo si mantiene il codice **semplice e manutenibile**.

Il refactoring è il processo dove si ristruttura il codice però non effettuando modifiche alle sue funzionalità originali e ai suoi comportamenti esterni.

Test Driven Development (TDD)

Il modello richiede di **scrivere i codici di test prima** di scrivere il codice applicativo

Dato che i test vengono scritti a partire dalle stories, diventano essi stessi il **driver per completare lo sviluppo**

Quindi, il testing NON è effettuato con prove a caso

Bisogna capire come definire il numero di test che servono

Quando si finisce lo sviluppo di una storia, si deve **testare l'intero sistema** e non solo le feature introdotte con essa, ciò perché potrebbero presentarsi dei bug sul codice già esistente.

Pair Programming



Serve ad ottenere la **condivisione della conoscenza**

Vari studi hanno notato che si scoprono più errori leggendo il codice che eseguendolo

Quindi, la **code inspection** è una tecnica che consente di scoprire gli errori nel codice con una **percentuale di successo maggiore rispetto alla code execution**.

Negli approcci **plan-driven**, la code inspection era un processo, cioè un vero e proprio meeting dove il team leggeva il codice e rispondeva ad un questionario

Invece, nel **pair programming** la **code inspection** è **intrinseca** dato che uno scrive il codice e un altro, leggendo, dà consigli continui

XP guarda agli **aspetti tecnici**, ma **NON** fornisce un'idea dell'**organizzazione del team**

Fornisce un catalogo di buone pratiche, usate anche al di fuori di XP come refactoring e TDD (diventato behaviour development)

SCRUM (mischia nel rugby)

È un metodo agile che si focalizza sulla gestione del team di sviluppo

Il termine deriva dal rugby e indica forza e agilità

Un **team** (o processo) **Scrum organizza il lavoro secondo 3 fasi:**

- **Piano di massima:** la fase iniziale è un piano di massima (outline planning) dove si **stabiliscono** gli **obiettivi** generali del progetto e l'**architettura** del sistema da costruire
- **Fase di sviluppo:** la seconda fase è composta da una serie di cicli di sprint (**sprint cycles**) dove **alla fine di ogni ciclo si ottiene una nuova versione del sistema** con features aggiuntive
- **Closure:** l'ultima fase, viene fatto un assessment post mortem per capire cosa si è imparato dal progetto

SCRUM **NON** ci dice cosa fare, ma **come si deve organizzare il team** di lavoro e il tempo a disposizione

Organizzato secondo 3 principi:

- **Ruoli:** sono product owner, ScrumMaster e team vero e proprio.

Lo ScrumMaster è il responsabile della corretta applicazione delle pratiche.

- **Meetings:**

- sprint planning, cioè ogni sprint avrà un meeting per pianificare
- sprint review
- retrospective
- daily scrum meeting (quello che era lo stand up meeting in XP)

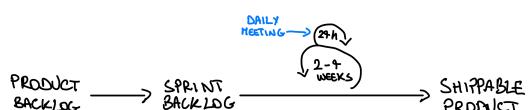
- **Componenti**

- **product backlog:** tutto ciò che è il futuro del prodotto, il prodotto prima o poi dovrà avere delle feature che sono nel backlog.

NON sono elementi di **pianificazione**, fanno parte della visione e entrano nella pianificazione quando entrano nello sprint backlog

- **sprint backlog:** una quota che fa parte del product backlog entra nello sprint backlog e viene programmata, decido cosa entra nello sprint planning e lo misuro nel burndown chart
- burndown chart: quanta "energia" ho ancora nello sprint per raggiungere gli obiettivi prefissati.

Scrum può essere visto come **due cicli interlacciati, uno che ruota intorno al product backlog e l'altro intorno al Sprint backlog**



Fra product e sprint backlog c'è un continuo interscambio

In generale, **si pianifica in base alle esigenze correnti**, quindi sullo sprint, mentre il **product serve per avere una visione a lungo termine**.

Il **product backlog** viene spesso **rivisto**, cioè può accadere che delle storie che stavano per entrare nello sprint backlog potrebbero smettere di interessare e quindi si evita di farle entrare in produzione

Il **product owner** deve assicurare che il team di sviluppo sia sempre concentrato sul prodotto che stanno sviluppando

Lo **Scrum Master** invece deve guidare il team affinché utilizzi correttamente il metodo Scrum, quindi deve anche garantire il rispetto delle pianificazioni

PBI estimation (Product BackLog Item)

La stima del lavoro si può fare in vari modi

Effort requirement: è il metodo tradizionale, in cui il lavoro si misura con i giorni-uomo, ore-uomo

Invece, Scrum tende ad utilizzare una grandezza **adimensionale**, cioè gli **story point**

Gli story point possono essere grandezze composte e/o una stima arbitraria dello sforzo che serve per implementare il backlog tenendo conto di determinati fattori.

Queste stime si costruiscono comparando le storie tra loro

Esempio: una storia che è sul mercato ha un certo livello di difficoltà, allora se la storia che devo sviluppare è simile a quella presente sul mercato, di cui conosco il livello di difficoltà, allora assegnerò lo stesso livello di difficoltà.

Per effettuare le stime spesso vengono usate le sigle delle taglie (S, M, L, XL), oppure i numeri della sequenza di Fibonacci.

Una delle tecniche di **PBI Estimation** è il **“Planning Poker”**

L'**obiettivo** è ottenere una stima condivisa del team sullo **sforzo** richiesto per completare un PBI.

Fasi del planning poker:

- **Preparazione:** ogni stimatore ha a disposizione un mazzo di carte del Planning Poker con valori numerici specifici, solitamente basati sulla sequenza di Fibonacci modificate (1, 2, 3, 5, 8, 13, 20, 40, 100).

Questi numeri rappresentano gli **story point**, un'unità di misura astratta per lo sforzo

- **Discussione:** il team discute il PBI da stimare, assicurandosi di avere una comprensione comune dei requisiti
- **Stima individuale:** ogni stimatore seleziona privatamente una carta dal proprio mazzo che rappresenta la propria stima dello sforzo per quel PBI.
- **Rivelazione simultanea:** tutti gli estimatori rivelano le proprie carte contemporaneamente. Questo aiuta a evitare che le stime siano influenzate dalle opinioni degli altri.
- **Convergenza:** se tutti gli estimatori hanno selezionato lo stesso valore, quello diventa la stima finale per il PBI.
- **Discussione e iterazione:** se le stime divergono, gli estimatori motivano le proprie scelte, in particolare chi ha fornito la stima più alta e quella più bassa.

Dopo la discussione, il processo di stima viene ripetuto finché il team non raggiunge un consenso sulla stima finale.

La discussione deve essere pubblica, mentre la votazione segreta, l'approccio utilizzato è il DELPHI, cioè un approccio di indagine di tipo iterativo

XP invece utilizza come base il tempo, ma lo corregge con la velocità

In pratica, se il team di sviluppo dichiara che per implementare una feature occorrono 4 settimane, ma poi ne impiega 6, allora al prossimo ciclo di sviluppo se il team afferma di necessitare di 4 settimane, si può già prevedere che in realtà ce ne vorranno 6

Sprint activity

- **Sprint planning:** si scelgono le storie da sviluppare, il numero dipende dalla velocità del team
- **Sprint execution:** si implementano le storie scelte e se ci si accorge che è impossibile completare tutte quelle scelte, il ciclo non viene esteso, ma quelle incomplete tornano nel backlog
- **Sprint reviewing:** il lavoro viene revisionato dal team e possibilmente anche dagli stakeholders

Alla fine di ogni short meeting viene aggiornata la **board**, cioè una lavagna dove vengono affissi dei post-in, che rappresentano le storie da sviluppare

L'aggiornamento della board prevede che le storie si "muovano" su di essa, cioè passano da uno stato all'altro

Il nome dello stato si trova sulle colonne e viene scelto dal team

La board viene tradotta nella burndown chart dal product owner

In pratica, quest'ultimo produce un grafico per avere un'idea sull'andamento del ciclo

Continuos integration

Alla fine di ogni sprint c'è un'attività di review, si vede se lo sprint ha raggiunto i propri goal, si individuano i problemi e si propongono soluzioni.

▼ Lezione 7 22/03/2024

Personas, Scenarios & Stories

Nel mondo tradizionale project-oriented, dopo la pubblicazione del bando o costruzione della vision inizia la fase di analisi dei requisiti.

Di fronte si avranno una serie di stakeholder a cui si possono chiedere informazioni per raccogliere requisiti

Nel **product based** questo **NON** è previsto, dobbiamo trovare un percorso alternativo.

I percorsi proposti sono tanti e portano più o meno allo stesso punto, cioè bisogna identificare un insieme di features che definiscono il back-log su cui faremo avanzare il progetto.

Esempio Vision: tutoraggio per studenti in una rete universitaria

Come si sviluppano le funzioni (cioè quello che il sistema deve fare) se **NON** c'è un cliente?

In generale, dato che **NON** c'è un **cliente commissionatore** da cui ottenere feedback, **come si decide cosa sta nel sistema e cosa no?**

Normalmente è il **team**, in particolare il **product manager che deve guidare il processo** a partire da 3 fattori principali, cioè si va a vedere:

- prodotti che non soddisfano i bisogni dei consumatori e dei business
- insoddisfazione dei business presenti o insoddisfazione del prodotto software
- cambiamenti nella tecnologia che permettono la nascita di nuovi prodotti software (taxi radio e uber)

Nelle prime fasi del progetto, il punto di arrivo deve essere un insieme di features, anche chiamate FUNCTIONALITY

Nel caso dei software product, **molti viene ricavato dalle analisi del livello di soddisfazione dei clienti**

Tutto ciò serve a distillare le **features**, questo termine se lo cerco online ha significati diversi, ma essenzialmente **è una cosa che il sistema deve fare per soddisfare la vision**

Le aziende usano questo termine nel modo più variegato, infatti spesso viene utilizzata per indicare i **NON-FUNCTIONAL REQUIREMENTS** (*sistema deve essere user friendly*)

NON-FUNCTIONAL REQUIREMENTS: tutto ciò che è importante ma **NON descrivibile attraverso un behaviour** del sistema

Il ricavare le feature a partire dalle esigenze degli utenti è una cosa che ha messo in difficoltà gli ingegneri del software per tanto tempo, perché gli utenti mediamente non sanno ciò che vogliono e quindi il mito che chiedendo ad un utente le features che vuole è stato demolito, dato che nel migliore dei casi l'utente ci dirà di voler mantenere la sua posizione di controllo e di poter mantenere la sua immaginazione del mondo.

L'idea di chiedere e ottenere una risposta è un'idea che ci ha accompagnato per un sacco di tempo, è un'idea ingenua e che **NON funziona**.

Tuttavia, non si può prescindere dall'utente per ottenere info riguardo un dominio specifico

Si devono capire le esigenze dei clienti senza interferire con essi.

Etnografia: studio degli usi e costumi che hanno portato all'evoluzione di una società

From personas to features

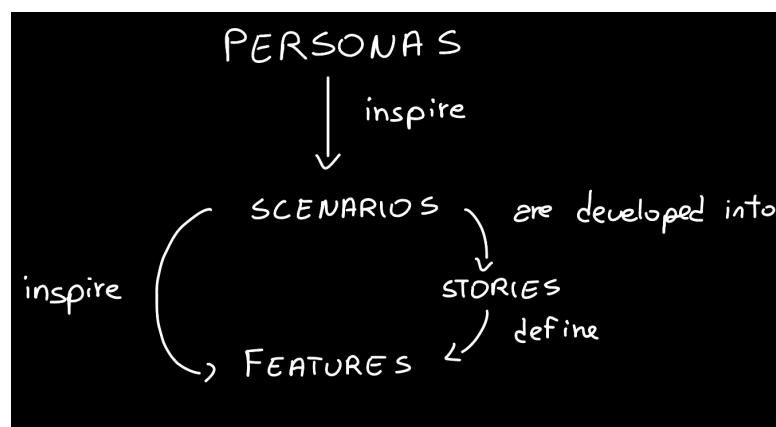
Personas

È un modo per rappresentare **utenti tipici** di un sistema

Le personas sono "utenti immaginari" eventualmente il risultato di un'indagine di tipo etnografica

È importante capire la differenza tra persona e ruolo

Il ruolo dice cosa l'utente farà, mentre la **personas dice come** compierà determinate azioni.



Le personas quindi sono particolari tipologie di possibili utenti

Le personas si usano per definire gli scenari, cioè come quell'utente interagirà con il sistema

Lo **scenario** tipicamente è complesso e **viene dettagliato in storie** ed è **la storia** che tipicamente **definisce la funzionalità** del sistema.

La storia definisce una funzionalità del sistema.

Cosa c'è in una features?

Una feature è ciò che ci consentirà di lavorare

Una feature ha un nome, una descrizione (trattare degli input sulla base di un trigger e specificare un'azione)

Esempio:

Panel di personas per svecchiamento sistema Cineca

A real example (in italian)

The image shows a persona profile for 'Giovanni'. At the top, there are two tabs: 'PERSONAS' (highlighted in blue) and 'DOCENTI' (highlighted in green). Below the tabs is a circular icon containing a cartoon illustration of a man with glasses and a mustache. To the right of the icon, the name 'Giovanni' is written in bold black text, followed by the subtitle 'Il docente ordinario veterano' and the age '64 anni'. A green quote 'QUOTE " Ai miei tempi..."' is displayed below his name. On the left side of the profile, there are three sections with horizontal scales: 'PROFILO / BACKGROUND' (describing Giovanni as a professor who uses digital tools), 'CULTURA DIGITALE' (describing his level of comfort with digital tools), and 'FREQUENZA D'USO' (describing how often he uses university services). On the right side, there are three sections with horizontal scales: 'USO DEL SERVIZIO / ATTIVITÀ PRINCIPALI' (describing his use of services and main activities), 'BISOGNI' (listing needs like support for tracking and managing tasks), and 'OSTACOLI' (listing obstacles like difficulty delegating authority). Each section includes a brief description and a bulleted list of specific requirements or challenges.

Da queste vengono fuori dei punti di partenza, delle opportunità che non sono ancora features
Si passa dalle opportunità al catalogo delle idee.

Sulla base del lavoro fatto (personas, ruoli, ecc) si traccia una prima distinzione delle opportunità.

Le grafichiamo su un piano i cui assi sono Fattibilità Tecnologica e Valore per utenti ⇒ abbiamo così creato una roadmap delle cose da fare, ma non bisogna svilupparle tutte per entrare sul mercato.

Come descriviamo le personas?

La letteratura si apre, il prof consiglia il libro

- Minimo di caratteristiche che le identificano come persone
- Ruoli che ricopre e che lo porteranno ad utilizzare il sistema, anche il background culturale
- Education
- Rilevanza

Descrivere le personas è un modo che abbiamo per immedesimarcì negli utenti, le personas si possono ricavare dagli utenti di prodotti simili.

Eventualmente, si può indicare quale feature del sistema interessa particolarmente ad una persona

Le personas sono un modo per immedesimarsi negli utenti dell'app

Le personas si possono ricavare dalla propria esperienza oppure analizzando gli utenti di altri prodotti simili

Scenarios

Describe come un gruppo di utenti userà il sistema, il singolo scenario non deve essere inteso come requisito.

È il modo in cui una o più persone interagiscono con il sistema, **all'interno dello scenario possiamo trovare un insieme di STORIES**

Si tratta di una descrizione di una situazione in cui un utente utilizza le funzionalità del prodotto per fare qualcosa che vuole fare.

Quando si descrive un sistema, è importante assicurarsi di coprire tutti i ruoli potenziali degli utenti.

Come descriviamo uno scenario?

1. **Nome:** Assegnamo un titolo distintivo allo scenario.
2. **Obiettivo:** Definiamo l'obiettivo che gli attori intendono raggiungere attraverso l'interazione con il sistema.
3. **Attori:** Identifichiamo i partecipanti o le persone coinvolte nello scenario.
4. **Problemi:** Indichiamo le sfide o le difficoltà che gli attori possono incontrare durante lo scenario.
5. **Soluzioni:** Descriviamo il modo in cui il sistema affronta i problemi identificati, senza entrare nei dettagli implementativi.

Essi devono essere descritti dal punto di vista dell'utente, focalizzandosi su ciò che il sistema deve fare e non su come deve farlo.

La descrizione degli scenari deve mantenere una neutralità rispetto all'implementazione, sebbene talvolta un dettaglio di implementazione possa contribuire a una migliore comprensione dello scenario.

Quindi, possiamo dire che gli scenarios sono ancora troppo complessi per ricavare le features

User involvement

Gli scenari sono progettati per essere facilmente comprensibili da chiunque, e quindi consentendo il coinvolgimento degli utenti nello sviluppo del sistema.

- **Sviluppo di uno scenario immaginario:** L'approccio migliore consiste nel creare uno scenario immaginario sulla base della nostra comprensione di come il sistema potrebbe essere utilizzato.
- **Coinvolgimento degli utenti:** Successivamente, chiediamo agli utenti di identificare eventuali discrepanze o punti poco chiari. Questo ci permette di ottenere feedback su ciò che non è stato compreso e suggerire miglioramenti per rendere lo scenario più realistico ed esaustivo.
- **Competenze degli utenti nella scrittura degli scenari:** La storia ha dimostrato che gli utenti spesso non sono esperti nella scrittura degli scenari.

Le loro descrizioni tendono a essere troppo dettagliate e specifiche, rendendo difficile per gli altri utenti generalizzare l'esperienza.

Gli scenari creati dagli utenti possono riflettere il modo in cui operano attualmente, risultando eccessivamente dettagliati e specifici. Ciò può limitare la capacità degli altri utenti di identificarsi con lo scenario e di applicarlo alle proprie situazioni.

In sintesi, coinvolgere gli utenti nello sviluppo degli scenari è cruciale per garantire la comprensibilità delle interazioni del sistema.

Tuttavia, è importante guidare questo processo per garantire che gli scenari siano chiari, concisi e facilmente adattabili a una varietà di contesti utente.

User Stories

Le **user stories** rappresentano interazioni dettagliate tra il sistema e le personas.

Sono descritte in modo strutturato e mirano a catturare le necessità specifiche dell'utente da parte del sistema software.

Formato Standard:

"In qualità di <ruolo>, io <voglio | ho bisogno> di <fare qualcosa>"

Formato con Giustificazione:

"In qualità di <ruolo>, io <voglio | ho bisogno> di <fare qualcosa> in modo che <ragione>"

Le **user stories sono il punto di partenza per lo sviluppo di sistemi software** e vengono utilizzate ampiamente nella pianificazione

Spesso il **backlog** del prodotto viene rappresentato come un **insieme di user stories**.

Le **user stories dovrebbero essere focalizzate su una singola feature chiara e definita** o su un aspetto di essa che può essere implementato in una singola iterazione di sviluppo, nota come sprint.

Quando una user story riguarda una feature più complessa che richiede più sprint per essere completamente implementata, viene chiamata "epic".

Le user stories, quindi, offrono una visione dettagliata delle interazioni sistema-persona e guidano lo sviluppo del software in modo iterativo e focalizzato sulle necessità dell'utente.

Importanza degli scenari

Dal momento che è possibile esprimere tutte le funzionalità descritte in uno scenario come storie utente, abbiamo davvero bisogno di scenari?

Gli scenari sono più naturali e sono utili per i seguenti motivi:

- **Leggibilità Naturale:** Poiché descrivono le interazioni utente-sistema in modo più naturale e narrativo, sono più facili da comprendere per gli utenti reali rispetto alle user stories.
- **Accessibilità agli Utenti:** Durante interviste o sessioni di verifica con utenti reali, gli scenari consentono una comunicazione più diretta e intuitiva, poiché riflettono il linguaggio e le esperienze degli utenti stessi.

- **Contesto Ampio:** Gli scenari forniscono un contesto più ampio sulle attività e le modalità di lavoro dell'utente, aiutando a comprendere meglio le sue esigenze e obiettivi nel contesto del sistema.

In breve, mentre le user story sono preziose per la pianificazione e la gestione del backlog del prodotto, gli scenari sono fondamentali per comprendere appieno le esigenze degli utenti e garantire che il sistema soddisfi le loro aspettative nel contesto delle loro attività quotidiane.

Story Development

Si prende un certo numero di user stories da uno scenario, questo processo è **user-centrico**.

Gli scenari, scomposti in storie, ci consentono di estrapolare le features.

Potremmo perderci delle richieste degli utenti che sono di natura NON FUNZIONALE (tempi di risposta, funzionamento con e senza rete ...)

Sono cose che non riusciamo a catturare.

Feature Identification

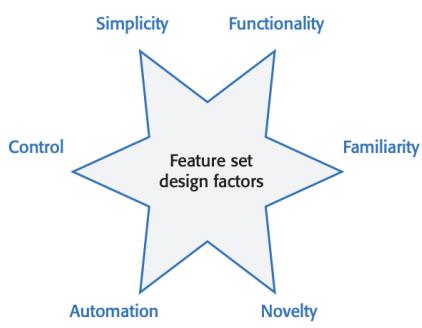
Si parte dalle storie, ma oltre a tener conto della conoscenza del prodotto bisogna avere anche una buona conoscenza della tecnologia.

Le features devono essere:

- **Indipendenti**
- **Rilevanti per gli utenti**
- **Coerenti**

Tutte le caratteristiche di qualità vanno in trade-off tra loro.

Feature trade-offs



- Simplicity and functionality
 - You need to find a balance between providing a simple, easy-to-use system and including enough functionality to attract users with a variety of needs.
- Familiarity and novelty
 - Users prefer that new software should support the familiar everyday tasks that are part of their work or life. To encourage them to adopt your system, you need to find a balance between familiar features and new features that convince users that your product can do more than its competitors.
- Automation and control
 - Some users like automation, where the software does things for them. Others prefer to have control. You have to think carefully about what can be automated, how it is automated and how users can configure the automation so that the system can be tailored to their preferences.

Le features sono il risultato di un processo ingegneristico che applica un trade off sulla qualità.

Feature Creep

Feature Creep è un fenomeno che si verifica quando vengono aggiunte nuove funzionalità in risposta alle richieste degli utenti senza considerare se queste funzionalità siano o meno utili in generale o se possano essere implementate in altro modo.

Esempio: Office

Feature Derivation

Le feature possono essere derivate direttamente dalla product vision o dagli scenario.

Se serve si possono dare dei livelli di descrizione aggiuntivi direttamente alle features.

Le idee più innovative nascono dalla DESTRUCTION e non dall'UNDERSTANDING di ciò che un cliente richiede

Esempio: nessun utente della metro di Tokyo avrebbe mai detto che era arrivata l'ora di avere la musica in tasca (Sony Walkman)

▼ Lezione 8 08/04/2024

Che cos'è un'architettura?

In generale l'architettura si occupa di selezionare gli elementi architettonici, della loro interazione e dei vincoli che questi elementi hanno. Questa selezione viene effettuata per dare un'idea dei requisiti da soddisfare e per avere una base di progettazione.

CHE COS'È UN'ARCHITETTURA SOFTWARE?

I sistemi diventano sempre più grandi e complessi, di conseguenza **diventa cruciale gestire** una vasta gamma di **questioni di progettazione** che vanno oltre la mera logica computazionale.

Alcuni **aspetti** da tenere in considerazione sono:

- **protocolli** per la **comunicazione**
- **sincronizzazione** e **accesso ai dati**
- **distribuzione fisica**

L'**architettura software** è un'**organizzazione** di determinate parti, le quali seguono uno stile che risponde a concetti di **funzione** o **parametri** di **qualità** che sono **dipendenti dal tempo** in cui l'architettura è stata ideata, infatti riflette le idee e le conoscenze degli ingegneri di quell'epoca.

Solo recentemente lo studio delle architetture software ha assunto il peso e l'importanza che deve avere all'interno di un progetto software.

La **raccomandazione** che veniva fatta in **passato** era quella di **focalizzare** la fase iniziale sull'**analisi** delle cose che il **sistema avrebbe dovuto fare senza lasciarsi influenzare** dalla **forma** (architettura/tecniche) che il sistema doveva assumere

Il ragionamento era così **radicale** perché all'epoca l'**architettura** era considerata **un'invariante** e quindi il **modello architettonico** era **predefinito** e andava solo riempito di contenuti applicativi, cioè i sistemi erano **monolitici** e tutto girava su mainframe

Con "**architettura**" si intende **come** le varie **parti** che compongono un sistema **collaborano** tra loro

Ci sono voluti un po' di anni a trovare una common understanding, cioè una definizione che va bene un po' per tutte le considerazioni che facciamo

L'**architettura software** ha assunto sempre maggiore importanza, infatti sono stati inventati **linguaggi** proprio per definire l'architettura.

Definizioni della letteratura

Perry e Wolf nel '92 hanno introdotto la distinzione tra gli elementi di elaborazione, di dati e di connessione

Dopo aver definito le interazioni, vincoli ed elementi si vanno a dettagliare le singole parti, ad implementare le interfacce, algoritmi e strutture dati.

Garlan e Shaw sono le due personalità che hanno spinto il concetto di architettura software ad avere un approccio ingegneristico.

Secondo loro il problema è come organizzare la struttura più che l'implementazione degli algoritmi veri e propri.

Si è iniziato a capire che in base all'organizzazione delle parti il sistema rispondeva in modo diverso, non tanto da un punto di vista funzionale ma di scalabilità, resistenza ai failure, ecc.

Tutto ciò è stato possibile perché con il passare del tempo si è andati **oltre l'architettura monolitica**.

Alcune definizioni

- **Componente:** è un sottosistema che offre un servizio alle altre parti del sistema, quindi può essere un elemento computazionale o un data store. Prendiamo ad **esempio** la programmazione OO, un componente è la classe
- **Connettori:** gestiscono le interazioni tra le componenti, quindi indicano come sono connesse. Una delle connessioni più facili da fare è quella con variabili condivise, altri modi possono essere i messaggi, eventi, pipe.
- **Interfacce:** sono il punto di accesso dall'esterno ad una componente o connettore e descrivono come interagire con esso

In passato avevamo architetture basate sul concetto di bus, un canale neutro sul quale le componenti potevano esporre le loro interfacce e altre componenti potevano richiamarle.

Questa idea ha avuto tanta attenzione prima del web, cioè prima che le architetture fossero basate sul web

Oggi si è passati ad utilizzare il livello di trasporto del web.

Il bus che oggi si usa è HTTP, le interfacce le definiamo con un linguaggio come IDL oppure REST.

Un modo per far parlare le componenti è far partire dei messaggi etichettati con topic a cui le componenti possono registrarsi

Esempio: Android che chiede con quale browser aprire un link

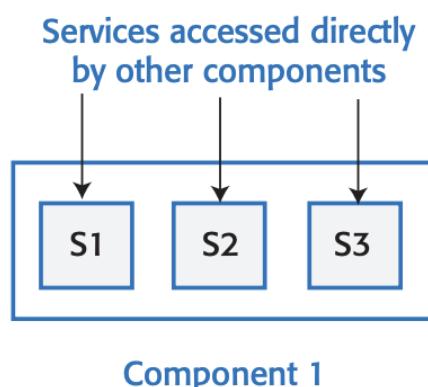
Il bus concettualmente c'è sempre

Accedere ai servizi forniti da componenti software

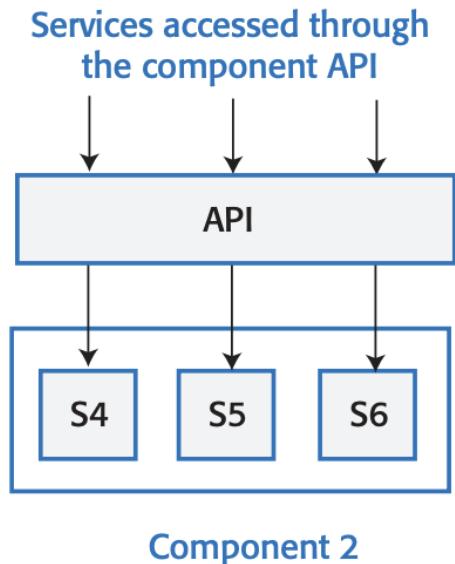
Il miglior modo per immaginare una componente software è pensarla come una collezione di uno o più servizi che possono essere usati da altre componenti.

Abbiamo due tipologie di accessi:

- Servizi a cui si accede direttamente da altri componenti



- Servizi a cui si accede tramite API



I parametri da portare in conto quando progettiamo la nostra architettura sono:

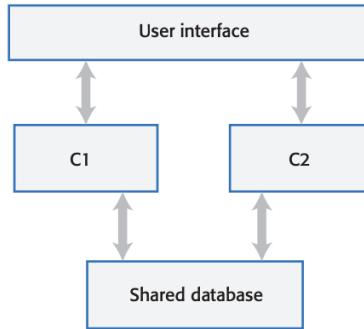
- **responsitività** ⇒ Il sistema risponde in un tempo ragionevole?
- **reliability** ⇒ Le funzionalità del sistema si comportano come previsto dagli sviluppatori e dagli utenti?
- **availability** ⇒ Il sistema è in grado di fornire i servizi richiesti dagli utenti?
- **security** ⇒ Il sistema protegge se stesso e gli utenti da eventuali attacchi e intrusioni?
- **usability** ⇒ Il sistema garantisce l'accesso alle feature che gli utenti vogliono in tempi brevi e senza errori?
- **Maintainability** ⇒ Il sistema può essere aggiornato senza costi inaspettati?
- **resilience** ⇒ Il sistema può continuare a fornire servizi agli utenti in caso di guasto parziale o di attacco esterno?

Per **accedere** ad una **libreria** creiamo oggetti che sono PROXY, cioè sanno dove recuperare una determinata **informazione**. Così creiamo uno strato di **ridirezione**

Non c'è problema di eleganza che non possa essere risolto aggiungendo un layer, non c'è un problema di performance che non possa essere ritolto togliendo un layer

Trade off maintainability and performance

Supponiamo che C1 funzioni lentamente perché ha bisogno di riorganizzare le informazioni nel database prima di essere usato, l'unico modo che abbiamo di rendere C1 più veloce è quello di cambiare il database, ciò vuol dire che anche C2 ha bisogno di cambiamenti che potrebbero eventualmente cambiare il suo tempo di risposta.

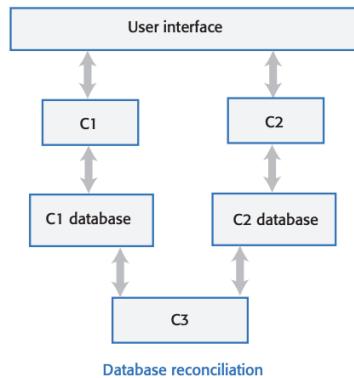


Altro approccio

Supponiamo che una componente ha bisogno di cambiare l'organizzazione del database e che questo non influenzi le altri componenti

Tuttavia un'architettura a multi database potrebbe girare molto più lentamente e costare tanto per essere implementata e cambiata.

Un'architettura multi database ha bisogno di meccanismi per assicurare che i dati siano consistenti quando vengono modificati.



Domande da porsi quando si sviluppa un'architettura

Quali componenti devo usare? Dove le metto?

Cosa faccio fare al client e cosa al server?

Come le distribuisco e come le faccio comunicare?

L'obiettivo è avere le proprietà:

- Security
- Reliability
- Performance

Principi per organizzare il sistema in parti

- **Abstraction e information hiding** ⇒ offriamo una vista pubblica cioè un insieme di operazioni pubbliche nascondendo come sono organizzati i dati e le operazioni

- A noi piacerebbe tanto che ogni parte si esponesse attraverso un'interfaccia pubblica

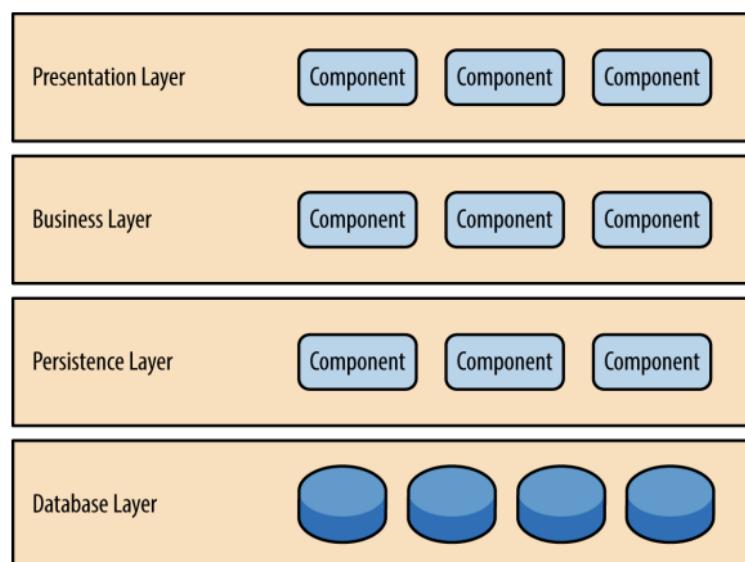
LAYERED ARCHITECTURE

Utilizzato per modellare le interfacce dei sottosistemi.

- Organizza il sistema in una serie di livelli, ciascuno dei quali fornisce una serie di servizi.
- Supporta lo sviluppo incrementale di sottosistemi in diversi livelli. Quando l'interfaccia di un livello cambia, viene influenzato solo il livello adiacente.
- Tuttavia, spesso è difficile strutturare i sistemi in questo modo.

Layer fondamentali:

- persistenza
- logica business
- presentazione



Esempio: Android



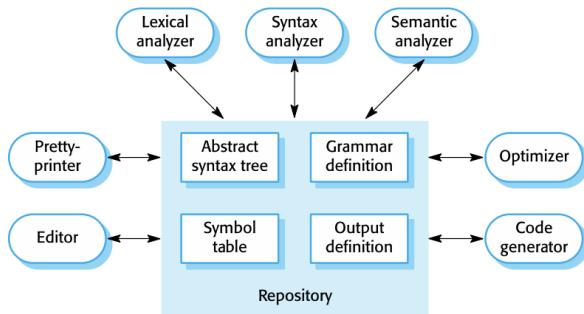
Ogni **funzione** di un determinato **layer** deve usare solo le **risorse** del layer **sottostante**

Ogni **componente** vede il layer **sottostante** come una **macchina virtuale**

MODELLO A REPOSITORY

Questo modello vive intorno ad un modello di dati

La slide descrive come è fatto un compilatore



Anche SAP (prodotto) è fatto così, è un sistema di gestione risorse enterprise

Un ERP (Enterprise Resource Planning) è un insieme di risorse condivise intorno a cui si possono attaccare (plug) dei moduli

I **sottosistemi** che compongono il software **accedono** e **modificano** una singola struttura dati chiamata appunto **repository**. I vari sottosistemi sono fra loro relativamente **indipendenti**, in

quanto interagiscono solo mediante il repository

ARCHITECTURAL PATTERN

Un pattern architettonico è una descrizione stilizzata di buone pratiche di design che sono state provate e testate in diversi ambienti.

I pattern dovrebbero avere informazioni che specificano quando il loro impiego è utile o meno.

MODEL-VIEW-CONTROLLER

È il modello architettonico con cui è realizzato il file manager ed è molto utilizzato in ambito web

Model ⇒ è la parte di **DATA** e **PERSISTENZA**

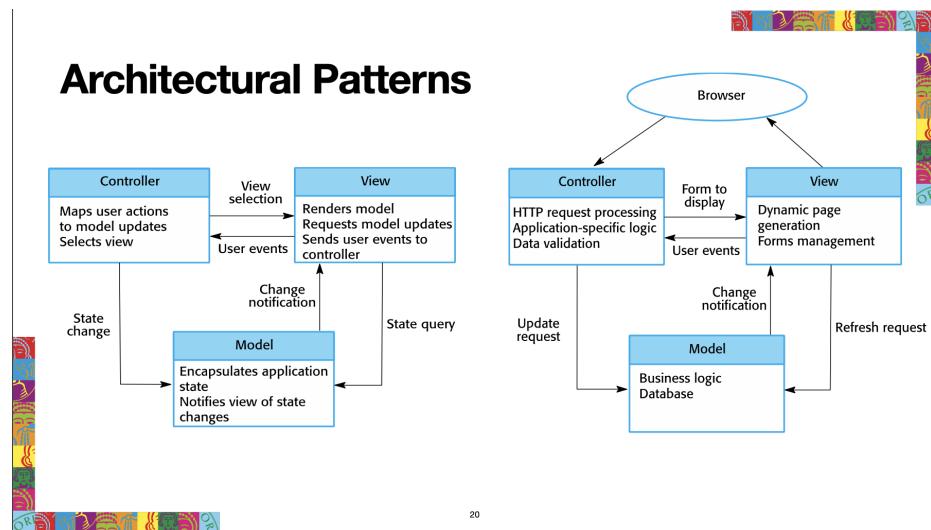
Controller ⇒ **logica** che c'è dietro il modello, permette di propagare le interazioni dell'utente dalla parte **View** al **Model**

View ⇒ parte di **presentazione**, quindi definisce e gestisce come i dati sono presentati all'utente

Questo pattern separa la presentazione e l'interazione dai dati del sistema

È composto da una serie di view che "guardano" una directory del file system.

Se tramite una view modifco il nome di un file allora ci sarà una **componente** che **propagherà** questa **modifica** dello **stato** nel modello



Vantaggi: permette ai dati di cambiare indipendentemente dalla loro rappresentazione e viceversa

Svantaggi: potrebbe richiedere codice in più o più complesso

Parametri che guideranno la scelta dell'architettura

Ogni architettura è il risultato dell'applicazione di un pattern, la scelta del pattern dipende da:

- **Caratteristiche del prodotto non funzionali:** sicurezza, prestazioni, usabilità influenzano la soddisfazione degli utenti e il successo commerciale.
- **Durata del prodotto:** sistemi con lunga durata richiedono architetture evolutive per adattarsi a nuove funzionalità e tecnologie.
- **Riutilizzo del software:** il riutilizzo di componenti esterni può vincolare le scelte architettoniche, ma può risparmiare tempo e fatica.
- **Numero di utenti:** i sistemi con molti utenti richiedono architetture scalabili per gestire le variazioni di carico.
- **Compatibilità del software:** la compatibilità con altri software può limitare le scelte architettoniche, come il database utilizzato.

Terminologia again again again

- **Servizio** ⇒ È un'**unità** coerente di **funzionalità**. (Insieme di funzionalità che ci azzeccano l'una con l'altra)
Questo può significare cose diverse a diversi livelli del sistema.
Ad
esempio, un sistema può offrire un servizio di posta elettronica e questo può includere servizi per creare, inviare, leggere e archiviare posta elettronica.
- **Componente** ⇒ È un'**unità software** con un nome che **offre** uno o più **servizi** ad altre **componenti** software o agli utenti **finali**.
Quando viene utilizzato da altre componenti l'accesso a questi servizi avviene tramite un'API.
~~Le componenti possono utilizzare più componenti per implementare i propri servizi.~~
- **Modulo** ⇒ È un **insieme** di **componenti**, le componenti in un modulo devono avere qualcosa in comune, per **esempio** dovrebbero offrire un insieme di servizi correlati.

In sintesi: **il servizio è la funzionalità mentre la componente è la struttura.**

Collezione componenti in dei moduli per offrire servizi attraverso un'interfaccia.

Il modo con cui organizziamo le componenti fa cambiare il modo in cui vengono soddisfatte le proprietà desiderate.

Architectural design guidelines

Posso organizzare i trade off con 3 principi:

- **separazione dei problemi**, cioè si vuole organizzare l'architettura in **componenti** che si **focalizzano su un determinato problema**
- **interfaccia stabile**, si vogliono progettare interfacce per le componenti che siano **coerenti** e che **cambino lentamente**
- **implementare solo una volta**, si vuole **evitare di duplicare** le **funzionalità** in diversi posti dell'architettura

Cross cutting concerns

Si tratta di features che riguardano tutti gli oggetti del sistema

Esempio: undo in un editor di testo

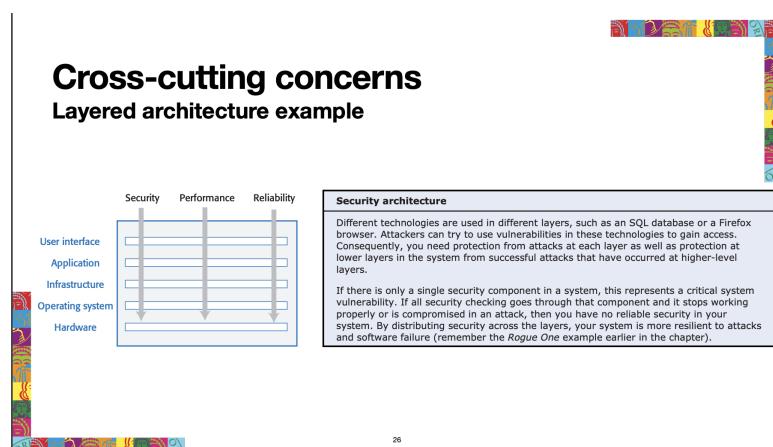
Tutti gli oggetti dell'editor oltre a svolgere la propria funzione devono sempre permettere il ripristino dell'operazione

I problemi trasversali sono problemi sistematici, cioè riguardano tutto il sistema.

Per esempio in un'architettura a layer i problemi trasversali affliggono tutti i layer del sistema così come le persone che lo usano.

I problemi trasversali sono completamente diversi da quelli funzionali.

A causa dei problemi trasversali spesso è difficile modificare un sistema dopo che è stato progettato per migliorarne la sicurezza.



Esempio di architettura a layer per un'applicazione web

- **Interfaccia utente basata su browser o mobile:** Interfaccia di sistema basata su browser Web dove spesso vengono utilizzati moduli HTML per raccogliere input dall'utente. I componenti Javascript per azioni locali, come la convalida degli input, dovrebbero essere inclusi anche a questo livello. In alternativa, può essere implementata un'interfaccia mobile come un'app.
- **Gestione dell'autenticazione e dell'interfaccia utente:** Livello di gestione dell'interfaccia utente che può includere componenti per l'autenticazione dell'utente e la generazione di pagine Web.
- **Funzionalità specifiche dell'applicazione**
- **Servizi condivisi di base**
- **Gestione del database e delle transazioni:** Livello di database che fornisce servizi come la gestione e il ripristino delle transazioni.

Distribution Architecture

Vediamo come distribuire e far parlare tra loro le componenti

La distribuzione di un sistema software definisce i server nel sistema e l'allocazione dei componenti a questi server.

Architettura client-server

Le architetture **client-server** sono un tipo di distribuzione **adatta alle applicazioni** in cui i **client accedono** a un **database** condiviso e ad **operazioni di logica** su tali dati.

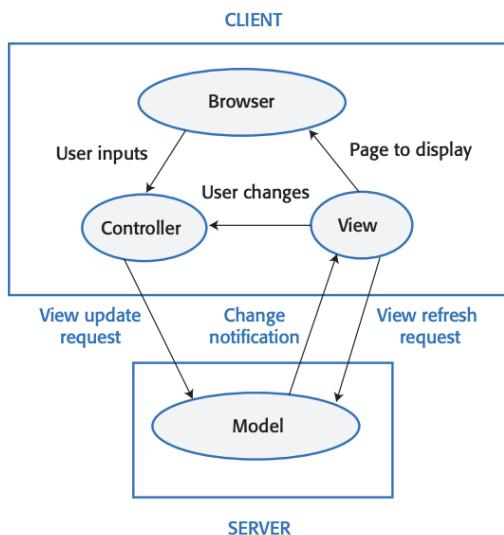
In questa architettura, l'**interfaccia utente** viene **implementata sul computer** o dispositivo mobile dell'**utente**. La funzionalità è distribuita tra il client e uno o più computer server.

Definiamo:

- **client pesante**: contiene tutta la logica applicativa
- **client leggero**: la logica applicativa è tutta sul server

Bisogna decidere dove "tagliare", cioè se mettere tutto su client, su server, 50 e 50.

Solitamente un'architettura di tipo client-server è fatta con **pattern MVC**



COMUNICAZIONE CLIENT-SERVER

Solitamente client e server comunicano tramite il protocollo HTTP

Il client invia un messaggio al server che include un metodo HTTP con l'identificativo della risorsa (URI)

HTTP è un protocollo text-only, quindi anche le strutture dati vengono rappresentate come testo

Le rappresentazioni più utilizzate sono XML e JSON

Architettura multi-livello client-server

Un'architettura multi-livello è una variazione dell'architettura client-server in cui i componenti di un'applicazione sono distribuiti su più livelli.

I livelli sono in genere **organizzati gerarchicamente**, con i livelli superiori che forniscono servizi a quelli inferiori.

Questa architettura è spesso utilizzata per applicazioni web di grandi dimensioni in cui è necessaria una separazione delle responsabilità tra i diversi componenti dell'applicazione

Architettura orientata ai servizi (SOA)

Un'**architettura orientata ai servizi** è uno stile architettonico in cui le **funzionalità** di un'**applicazione** sono **esposte come servizi indipendenti**

Questi servizi possono essere combinati e riutilizzati per creare nuove applicazioni o funzionalità.

Questo stile architettonico è spesso utilizzato per **applicazioni distribuite su larga scala** in cui sono necessarie flessibilità, riutilizzabilità e resistenza ai failure elevate

Queste caratteristiche sono garantite perché le varie **componenti** sono **stateless**, quindi possono essere **replicate** e **migrate** da un computer ad un altro senza problemi

Problemi nella scelta dell'architettura

Quando si sceglie un'architettura di distribuzione, è necessario considerare numerosi fattori, tra cui:

- **Tipo e aggiornamenti dei dati:** se si utilizzano principalmente **dati strutturati** che possono essere **aggiornati da diverse funzionalità** del sistema, allora sarebbe meglio avere un **singolo database condiviso** che gestisca le transazioni. Invece, se i **dati** sono **distribuiti** tra più servizi, è necessario un **sistema** che **garantisca la consistenza** e ciò aggiunge overhead al sistema
- **Frequenza di modifica:** se si prevede che i **componenti** del sistema verranno **regolarmente modificati** o sostituiti, allora è opportuno **isolare** questi componenti, quindi considerarli come servizi separati in modo da semplificare tali modifiche
- **Piattaforma di esecuzione del sistema:** se si prevede di eseguire il proprio sistema sul cloud e gli utenti vi accederanno tramite Internet, allora è meglio implementare un'architettura orientata ai servizi perché la scalabilità del sistema è più semplice.
Invece se si ha a che fare con un sistema aziendale che viene eseguito su server locali, allora un'architettura multi-livello potrebbe essere più appropriata.

Quali tecnologie si dovrebbero usare?

- **DATABASE**

Ci sono due tipi di database che sono comunemente utilizzati:

- **relazionali**, nei quali i dati sono organizzati in **tabelle** strutturate.

I database relazionali, come **MySQL**, sono particolarmente **adatti** a situazioni in cui è necessaria la gestione delle transazioni e le **strutture dati** sono **prevedibili** e abbastanza **semplici**

- **NoSQL**, nei quali i **dati** hanno un'**organizzazione più flessibile** e definita dall'utente.

I database NoSQL, come **MongoDB**, sono più flessibili e potenzialmente più efficienti rispetto ai database relazionali per l'analisi dei dati.

I database NoSQL permettono di organizzare i dati gerarchicamente piuttosto che come tabelle piatte e ciò consente un'elaborazione concorrente più efficiente dei "Big Data".

- **PIATTAFORMA DI DISTRIBUZIONE**

La distribuzione può avvenire tramite web, tramite un prodotto mobile oppure tramite entrambi.

Soltamente è necessario avere versioni separate, una per i browser e una per i dispositivi mobili, del front-end del prodotto

In particolare, i problemi riguardano principalmente i sistemi mobile:

- **Connettività intermittente:** è necessario fornire un servizio, seppur limitato, anche in assenza di connettività di rete.
- **Potenza del processore:** i dispositivi mobili hanno processori meno potenti, quindi è necessario ridurre al minimo le operazioni computazionalmente intensive.
- **Gestione dell'alimentazione:** la durata della batteria dei dispositivi mobili è limitata, quindi è necessario provare a ridurre al minimo l'alimentazione utilizzata dall'applicazione.
- **Tastiera su schermo:** le tastiere su schermo sono lente e soggette a errori, per cui si dovrebbe ridurre al minimo l'input tramite la tastiera su schermo per evitare la frustrazione dell'utente.

- **SERVER**

Una decisione chiave è progettare il sistema per eseguirlo su server dei clienti o eseguirlo sul cloud.

Per i prodotti customer che non sono semplicemente app mobili, ha quasi sempre senso svilupparli per il cloud.

Per i prodotti aziendali, è una decisione più difficile.

Alcune aziende sono preoccupate per la sicurezza del cloud e preferiscono eseguire i propri sistemi su server interni.

Una scelta importante che si deve fare se si sceglie di eseguire il software sul cloud è quale provider di servizi cloud utilizzare

- **OPEN SOURCE**

Un software open source è un software che è disponibile gratuitamente e che può essere cambiato a piacimento.

La decisione sull'utilizzo del software open source dipende anche dalla disponibilità, dalla maturità e dal supporto continuo dei componenti open source.

I problemi di licenza open source possono imporre dei vincoli al modo in cui si vuole utilizzare il software.

La scelta riguardante software open source dovrebbe dipendere dal tipo di prodotto che si sta sviluppando, dal mercato di riferimento e dalle competenze del team di sviluppo.

Vantaggi: riutilizzare piuttosto che implementare un nuovo software, il che riduce i costi di sviluppo e il time to market.

Svantaggi: si è vincolati a quel software e non si ha alcun controllo sulla sua evoluzione.

- **STRUMENTI DI SVILUPPO**

I tool di sviluppo, come un toolkit di sviluppo di applicazioni mobili o un framework per applicazioni web, influenzano l'architettura del software.

Queste tecnologie hanno assunzioni incorporate sulle architetture di sistema e ci si deve conformare a queste per utilizzare il sistema di sviluppo.

Le tecnologie di sviluppo che si utilizzano potrebbero anche avere un'influenza indiretta sull'architettura del sistema.

Di solito gli sviluppatori favoriscono le scelte architettoniche che utilizzano tecnologie a loro familiari. Ad **esempio**, se il tuo team ha molta esperienza di database relazionali, potrebbero proporre questi ultimi al posto di un database NoSQL.

▼ Lezione 9 12/04/2024 (Carmine)

What is Maven?

Maven è un command-line tool di **gestione** e comprensione di **progetti** che assiste gli ingegneri del software durante la manutenzione del software.

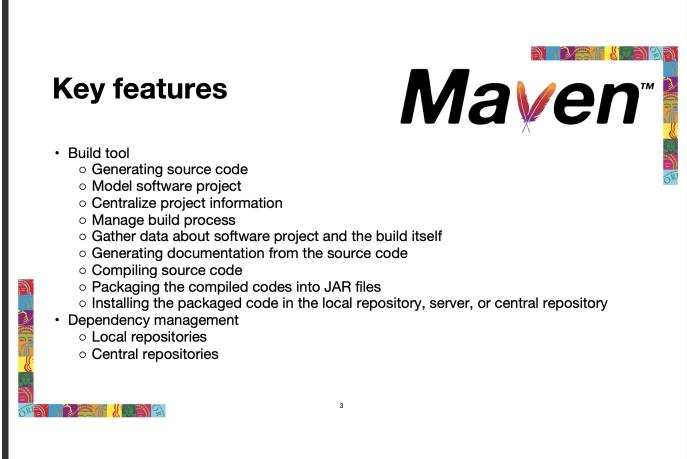
È possibile usare Maven senza configurare il sistema operativo, alcuni ide come eclipse lo hanno incorporato.

Fornisce agli sviluppatori un framework di build lifecycle completo.

Si basa sul **concepto** di **project object model (POM)**, Maven gestisce la build di un progetto, la segnalazione e documentazione di un progetto da un unico posto di "informazione" cioè il file pom.

Maven ha diversi obiettivi/aree di interesse:

- Rendere il processo di build più facile
- Fornire un sistema di sviluppo uniforme, infatti maven builda un progetto grazie al POM e un set di plugins.
- Fornisce informazioni riguardo la qualità del progetto
- Incoraggia migliori pratiche di sviluppo, per esempio
 - mantiene il codice di test in un albero separato ma parallelo
 - utilizzo nomi convenzionali per i codici di test
 - preferire convenzioni rispetto alle configurazioni



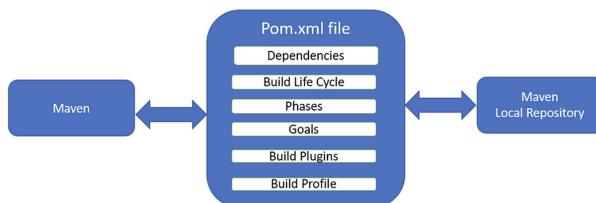
Key features

- Build tool
 - Generating source code
 - Model software project
 - Centralize project information
 - Manage build process
 - Gather data about software project and the build itself
 - Generating documentation from the source code
 - Compiling source code
 - Packaging the compiled codes into JAR files
 - Installing the packaged code in the local repository, server, or central repository
- Dependency management
 - Local repositories
 - Central repositories

Project Object Model

Il file **POM** è una rappresentazione **XML** di un **progetto Maven**, tutto contenuto in un file pom.xml

Contiene tutte le informazioni del progetto



Il POM è un oggetto contenitore di diverse entità

È gerarchico

Permette di definire **dipendenze**, **plugin**, **goal**

Il tag `<project>` è un tag obbligatorio

Maven Coordinates GAV

- **G ⇒ Group id**, identifica noi o la nostra azienda, dovrebbe essere unico per non creare conflitti. Può essere inteso come una firma. Tipicamente segue il formato di un **reverse domain name**
- **A ⇒ Artifact id**, identificatore univoco o nome assegnato ad un modulo all'interno di un gruppo.
- **V ⇒ Version**, identificatore per la versione o il numero di build di un progetto.

Questi parametri sono gli unici parametri **obbligatori** del pom (oltre a project)

Convention over Configuration

Anche conosciuto come coding by convention, è un software design paradigm usato da framework che cercano di diminuire il numero di decisioni che uno sviluppatore deve effettuare quando usa un framework senza perdere flessibilità.

In pratica, Maven **obbliga** ad avere una **struttura fissa** delle directory

Il POM è alla radice del progetto

Poi ci sono due directory:

- **src**: contiene il sorgente
- **target**: contiene i file generati alla fine del processo di compilazione/build

Avendo a disposizione il **POM** e la directory **src**, chiunque può essere in grado di ricostruire la directory **target**

Ciclo di vita di una build

Maven si basa sul concetto di **ciclo di vita della build**, che definisce i passaggi da effettuare per la creazione e la distribuzione di un progetto.

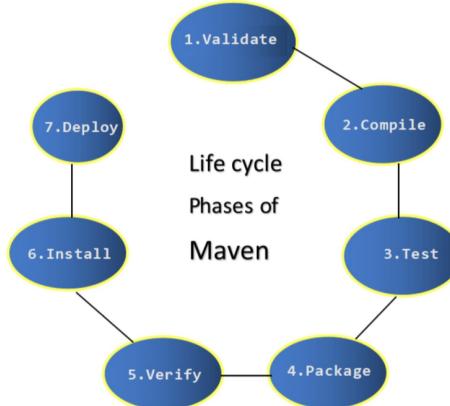
Maven fornisce 3 cicli di vita predefiniti:

- **default**: gestisce il deploy del progetto
- **clean**: cancella tutto il contenuto della directory target, riporta il codice allo stato originale
- **site**: gestisce la creazione del sito web del progetto

Ognuna di queste build è composta da una lista di fasi differenti

Fasi del ciclo di vita di DEFAULT

- **Validate**: verifica la correttezza e se sono presenti tutte le informazioni necessarie del progetto
- **Compile**: compila il codice sorgente
- **Test**: esegue il test del codice compilato utilizzando un framework di test
- **Package**: impacchetta il codice compilato in un formato distribuibile, ad esempio JAR
- **Verify**: esegue controlli sui risultati dei test per assicurarsi che la qualità che si desidera viene raggiunta.
- **Install**: installa i package esterni nella repository locale
- **Deploy**: viene effettuato nell'ambiente di sviluppo, copia il package finale nella repository remota per condividere con altri sviluppatori e progetti



Goal

Un goal rappresenta uno specifico task che contribuisce al build e alla gestione del progetto.

Può essere legato a zero o più fasi di build, quando non è legato a nessuna build può essere eseguito al di fuori del build lifecycle tramite un'invocazione diretta

Esempio: consideriamo `mvn clean assembly:single package`

- `clean` invoca il ciclo di vita
- `assembly:single` è un goal di un plugin
- `package` è una fase di un ciclo di vita

Plugins

Sono la feature principale di Maven perché permettono di riutilizzare una logica comune tra vari progetti.

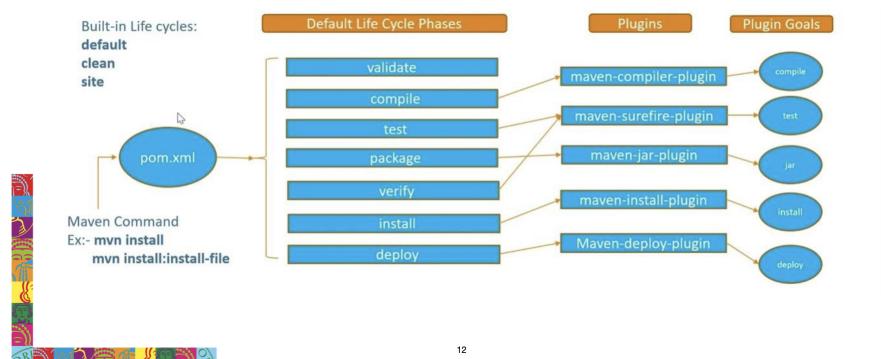
Ciò è possibile grazie alle "action" (per **esempio** creare un file WAR) nel contesto della descrizione di un progetto (cioè si indica nel pom).

Il comportamento del plugin può essere personalizzato tramite un set di parametri univoci esposti da una descrizione di ciascun goal del plugin.

Fornisce un insieme di goal che possono essere usati per eseguire un'attività.

Uno dei plugin più semplici di Maven è il **Clean Plugin** che è responsabile della rimozione della directory **target** di un progetto Maven. Quando si esegue `mvn clean`, Maven esegue il goal **clean** e la directory target viene rimossa.

Lifecycle - Plugin - Goals overview



Archetipo

Un archetipo è definito come un modello o uno schema originale da cui vengono ricavati tutti gli altri elementi dello stesso tipo.

È un modo per organizzare il progetto.

Un archetipo Maven è un'astrazione di un tipo di progetto che può essere istanziato in un progetto concreto e personalizzato.

È un modello di progetto da cui vengono creati altri progetti

Dipendenze

Esistono due tipologie di dipendenze:

- **dirette**: le dipendenze dirette sono quelle che in modo esplicito includiamo nel progetto con il tag `<dependency>` nel pom
- **transitive**: le dipendenze transitive sono richieste dalle dipendenze dirette, Maven include in modo automatico le dipendenze transitive richieste

Per listare tutte le dipendenze, incluse quelle transitive usiamo `mvn dependency:tree`
`dependency:tree` è un goal che non è associato a nessuna build lifecycle.

Dependency Scope

- **Compile** ⇒ disponibili nel classpath del progetto in tutti i build task
- **Provided** ⇒ dipendenze che dovrebbero essere fornite a runtime dalla JDK o da un container, un **esempio** sono le web application sviluppate in dei container dove il container fornisce già delle librerie.

È l'ambiente di sviluppo che fornisce la libreria.

- **Runtime** ⇒ librerie che non servono per la compilazione del progetto. Per questo motivo sono presenti nei classpath runtime e test ma non sono presenti in compile.
Scarico un jar e faccio istanziare una classe di quel jar al volo.
- **Test** ⇒ dipendenze che non sono richieste a runtime ma sono usate solo per scopi di test.
- **System** (deprecated) ⇒ molto simili a quelle provided con la differenza che loro puntano direttamente ad uno specifico jar nel file system

Per distribuire il codice si impacchetta nel jar

Per creare il **jar** si deve modificare il **manifest** per indicare la classe principale, cioè nel POM bisogna inserire il tag `<manifest>`

Documentazione

Maven può generare anche documentazione tramite il plugin **javadoc** e con il **goal javadoc**: `mvn javadoc:javadoc`

Maven supporta la documentazione dinamica

Multi-Mode

Un **multi-mode project** viene costruito **aggregando tanti file POM** che gestiscono un **gruppo di sottomoduli**

L'**aggregatore** è situato nella directory **root** del progetto e dovrebbe avere un **packaging** di tipo **pom**.

I sottomoduli sono regolari progetti Maven e possono essere buildati separatamente o attraverso l'aggregatore POM

Costruendo il progetto attraverso il POM aggregatore, ogni progetto che ha un tipo di packaging diverso da pom risulterà in un file build di archivio

Repository

Una **repository** in Maven contiene **artefatti** e dipendenze di vario tipo

Possono essere **locali** o **remote**:

- **locali**: tutte le librerie che il progetto usa le ha in locale e se non le ha effettua il **fetch** da remoto in modo automatico.
- **remote**: repository a cui si accede tramite una varietà di protocolli come `file://` e `https://`
Questi repository potrebbero essere:
 - **Repository veramente remote** create da terzi per fornire i loro artefatti per il download (ad esempio, `repo.maven.apache.org`).
 - **Repository interne** creati su un file o server HTTP all'interno di un'azienda, utilizzati per condividere artefatti privati tra team di sviluppo e per rilasci.

Le repo remote e locali sono strutturate nello stesso modo così il layout è completamente trasparente a Maven

▼ Lezione 10 15/04/2024

The Cloud

Il passaggio al Cloud è avvenuto circa 20 anni fa.

Il Cloud è composto da un grande numero di server remoti che sono offerti a noleggio dalle aziende che li posseggono.

I server cloud-based sono server virtuali, ciò significa che sono implementati nel software invece che nell'hardware.

Qualsiasi persona può noleggiare quanti server vuole, farci girare del software sopra e renderlo disponibile ai propri clienti che potranno accedere ai servizi offerti dai loro computer o altri dispositivi connessi in rete.

Questi server possono essere creati e spenti in base all'andamento del mercato.

Caratteristiche

Le caratteristiche fondamentali del cloud sono 3:

- **Scalabilità:** è la capacità del software di **gestire** un **numero crescente di utenti**.
- **Elasticità:** **adattare** le **configurazioni** dei server in **base** alla **variazione** della **domanda** da parte del mercato, è una feature correlata alla scalabilità.
- **Resilienza:** bisogna progettare l'architettura software in modo da **tollerare** eventuali **failures** dei server.

Vantaggi di utilizzare il cloud per lo sviluppo software

Utilizzare il cloud porta alcuni benefici:

- **costi:** si evitano i costi iniziali per l'acquisto dell'hardware.
- **startup time:** si può iniziare a lavorare fin da subito, non c'è bisogno di aspettare i tempi di consegna dell'hardware. Con il cloud, si possono avere server attivi e funzionanti in pochi minuti.
- **scelta del server:** se si nota che i server che affittati non sono abbastanza potenti, si può facilmente passare a sistemi più potenti. Inoltre, si possono aggiungere server per esigenze a breve termine, come i test di carico.
- **sviluppo distribuito:** se si ha un team di sviluppo distribuito che lavora da diverse località, tutti i membri del team hanno lo stesso ambiente di sviluppo e possono condividere tutte le informazioni in modo trasparente.

Virtual Cloud Servers

Grazie all'hardware sempre più potente è stata resa possibile la **virtualizzazione**

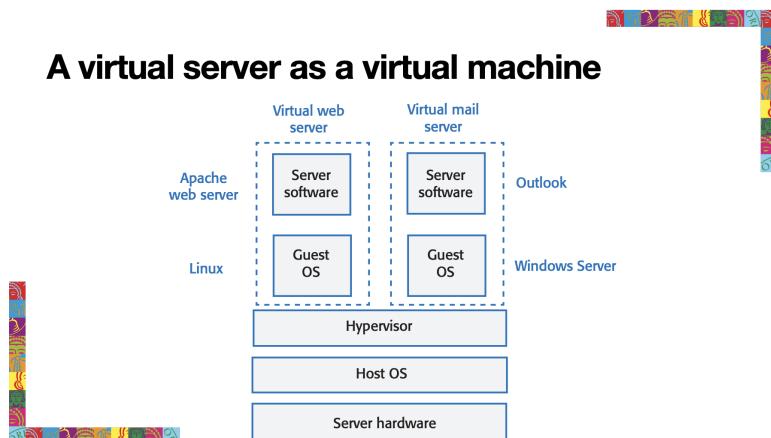
In pratica, a partire da un hardware è possibile istanziare **tanti server virtuali indipendenti ed in sicurezza** (grazie ad un **hypervisor**).

In questo modo, il **server** che **eroga il servizio** **NON** è **più fisico ma logico**.

Un Server Virtuale è un server che grazie ad un layer software riesce ad organizzare una serie di macchine virtuali.

Le macchine virtuali che sono installate sul server fisico possono essere usate per implementare server virtuali.

Il compito dell'**hypervisor** è quello di **emulare l'hardware sottostante**.



Questa viene anche chiamata **Virtualizzazione Pesante**, cioè si costruisce un **insieme di server ognuno con il proprio sistema operativo**

Virtualizzazione basata su container

Un'altra tipologia di virtualizzazione è quella basata su container (si può chiamare anche **leggera**).

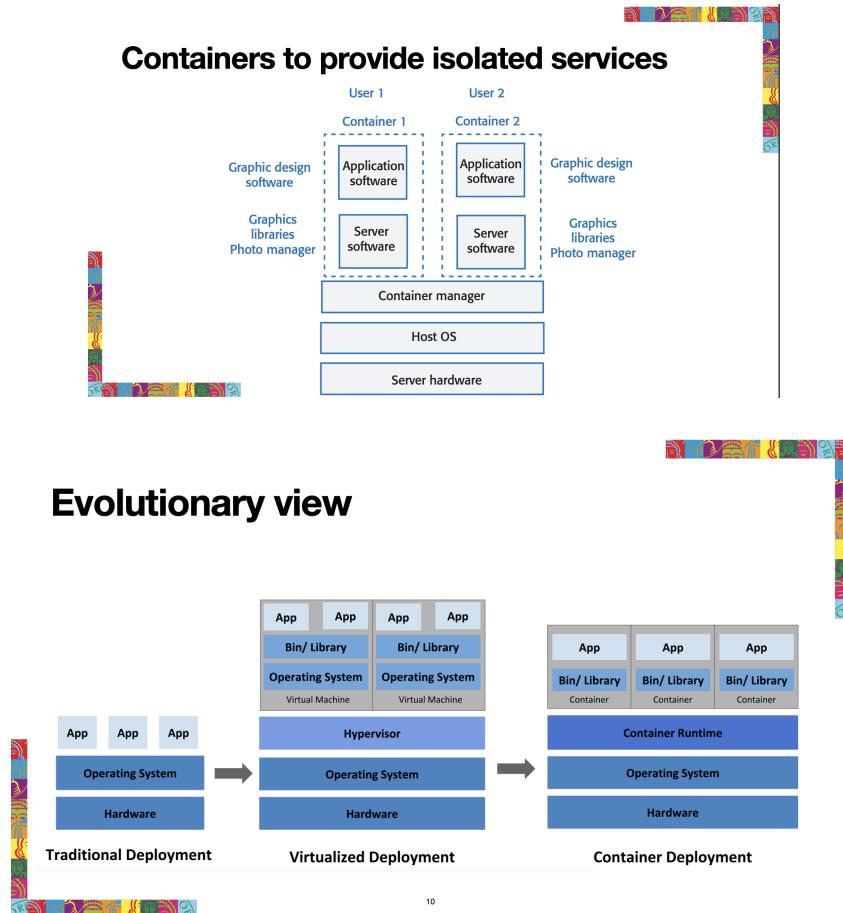
Se si esegue un sistema basato su cloud con molte istanze di applicazioni o servizi che utilizzano tutte lo stesso sistema operativo, è possibile utilizzare una tecnologia di virtualizzazione più semplice chiamata **container**.

Vantaggi dei container:

- **accelerano il processo di distribuzione** di server virtuali sul cloud.
- i **container** sono in genere di dimensioni in **megabyte**, mentre le VM sono in gigabyte.
- i **container** possono essere **avviati** e arrestati in **pochi secondi** rispetto ai pochi minuti necessari per una VM.

I **container** sono una tecnologia di virtualizzazione di un OS che **permette a server indipendenti di condividere un singolo sistema operativo**

Sono particolarmente utili per fornire servizi applicativi isolati in cui ogni utente vede la propria versione di un'applicazione.



Docker

I container sono stati inventati da Google intorno al 2007 ma sono diventati di uso comune nel 2015

Docker fornisce degli **strumenti** che **facilitano l'utilizzo e la manutenzione** dei **container**, inoltre è facile e veloce da usare.

È un **sistema di gestione** dei **container** che permette agli utenti di utilizzare il software che deve essere incluso in un container come immagine Docker.

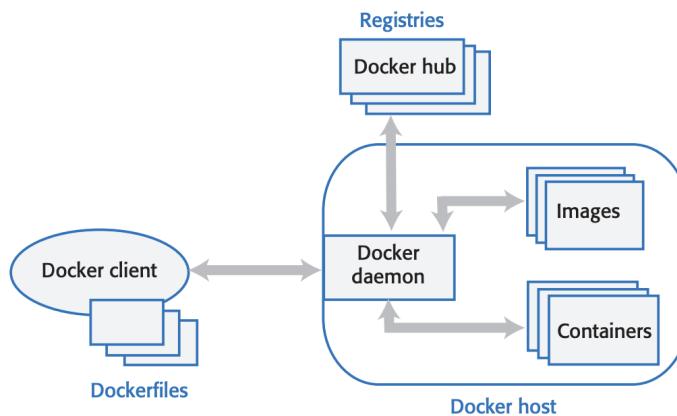
Include anche un sistema a **run-time** che permette di creare e gestire le immagini.

DOCKER CONTAINER SYSTEM

È composto da una serie di elementi:

- **Docker daemon:** è un processo che è eseguito su un server host ed è responsabile del set up, avvio, stop e monitoraggio dei container, oltre a permettere il build e la gestione delle immagini locali

- **Docker client:** è il software utilizzato dagli sviluppatori per definire e controllare i container
- **Dockerfile:** definisce le immagini con una serie di comandi che permettono di specificare che il software deve essere incluso in un container. Ogni container deve avere un Dockerfile associato
- **Image:** un Dockerfile è interpretato per creare un'immagine Docker, la quale sarà un insieme di directory con lo specifico software e dati installati
- **Docker hub:** è il registro che contiene le immagini che sono state create
- **Containers:** sono le immagini in esecuzione



Le docker images sono delle directory che possono essere archiviate, condivise, e eseguite su diversi Docker Host, tutto ciò che serve per eseguire un sistema software (file binari, librerie, tool di sistema ecc...) sono inclusi nella directory.

Un'immagine Docker è un livello di base, di solito preso dal registro Docker, con il proprio software e i propri dati aggiunti come livello superiore.

Il modello a layer permette di aggiornare un'applicazione Docker in modo facile e veloce, ogni aggiornamento viene considerato come layer aggiuntivo al sistema già esistente.

Per apportare delle modifiche ad un'applicazione tutto ciò che si deve fare è spedire i cambiamenti all'immagine che spesso sono solo un paio di file.

Kubernetes

Mentre Docker è una piattaforma di containerizzazione, Kubernetes è un tool di **orchestrazione** utilizzato per gestire più container

Docker offre un modo semplice ed efficiente per creare e distribuire i container, mentre Kubernetes offre funzionalità più complesse per la gestione dei container su larga scala

Le feature di Kubernetes sono:

- scaling

- self-healing: sostituisce automaticamente i container che hanno avuti errori con dei nuovi
- load balancing

Benefici dei Containers

I container **risolvono** il problema delle **dipendenze dei software**, non ci si deve preoccupare che le librerie e altri software sul server siano diversi da quelli sul server di sviluppo

Forniscono la **portabilità** del software su cloud diversi, i sistemi **Docker** possono **funzionare su qualsiasi sistema** o cloud provider dove è presente il Docker Daemon

Forniscono un meccanismo efficiente per l'implementazione di servizi software e **supportano lo sviluppo di architetture orientate ai servizi**.

Semplificano l'adozione dei **DevOps**, cioè di un approccio al software in cui lo stesso team è responsabile sia dello sviluppo che del supporto del software operativo.

Everything as a service

L'idea di affittare un servizio piuttosto che comprarlo è fondamentale per il cloud computing

Infrastrutture as a service (IaaS)

Un fornitore di servizi informatici permette ai clienti di creare piani personalizzati per la creazione di un'infrastruttura cloud scalabile

È possibile perché i cloud providers offrono diverse tipologie di servizi come

- servizi di computazione (risorse di calcolo)
- servizi di rete
- servizi di storage

Tutti questi possono essere usati per implementare i **virtual servers**.

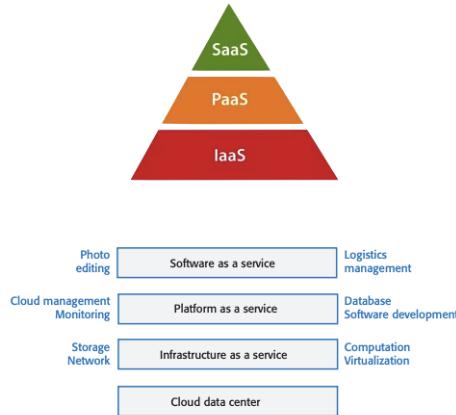
Platform as a service (PaaS)

Livello intermedio dove si utilizzano le **librerie** e i **framework forniti** dai cloud provider per **implementare il proprio software**.

I provider forniscono l'accesso a una serie di funzioni, compresi i database SQL e NoSQL

Software as a service (SaaS)

Il software gira sul cloud ed è utilizzato da utenti attraverso mobile app o web app



Software as a service (SaaS)

Esempio: abbonamento a office 365

Il **Software as a Service (SaaS)** è un modello di distribuzione del software in cui il **software viene ospitato su server remoti** ed è **accessibile agli utenti tramite un browser web o un'applicazione mobile**

I **clienti NON** devono **installare il software** e accedono al sistema remoto tramite un browser web o un'applicazione mobile dedicata

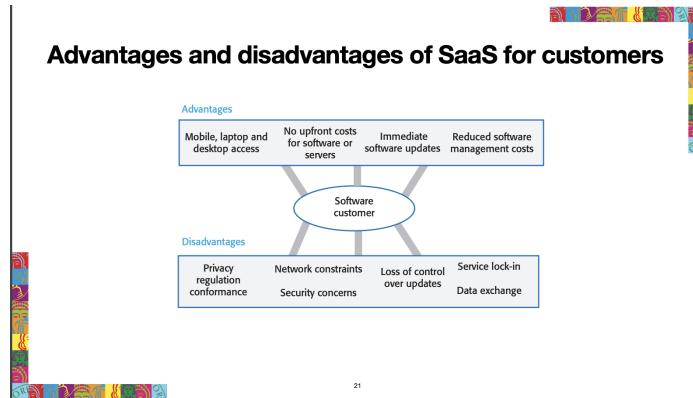
Il **modello di pagamento** per il SaaS è **soltamente** un modello di **abbonamento**. Gli utenti pagano una tariffa mensile per utilizzare il software anziché acquistarlo in blocco

Vantaggi del SaaS per i fornitori di prodotti software

- **Cash flow:** i clienti pagano un abbonamento regolare o in base all'utilizzo del software. Ciò significa che si dispone di un flusso di cassa regolare, con pagamenti durante tutto l'anno. Non si ha una situazione in cui si ha una grande entrata di denaro al momento dell'acquisto dei prodotti, ma di entrate molto basse tra le release dei prodotti.
- **Gestione degli aggiornamenti:** si ha il controllo sugli aggiornamenti del prodotto e tutti i clienti ricevono l'aggiornamento contemporaneamente. In questo modo si evita il problema di avere più versioni utilizzate e mantenute contemporaneamente. Ciò riduce i costi e semplifica la manutenzione di una base di codice software coerente.
- **Continous deployment:** è possibile distribuire nuove versioni del software non appena sono state apportate e testate le modifiche. Ciò significa che è possibile correggere i bug rapidamente in modo che l'affidabilità del software possa migliorare continuamente.
- **Flessibilità di pagamento:** è possibile avere diverse opzioni di pagamento in modo da poter attrarre una gamma più ampia di clienti. Le piccole aziende o gli individui non devono essere scoraggiati dal dover pagare ingenti costi iniziali per il software.
- **Prova prima di acquistare:** è possibile rendere disponibili rapidamente versioni gratuite o a basso costo del software con l'obiettivo di ottenere feedback dai clienti su bug e su come il

prodotto potrebbe essere approvato.

- **Raccolta dati:** è possibile raccogliere facilmente dati su come viene utilizzato il prodotto e quindi identificare aree di miglioramento. È inoltre possibile raccogliere dati sui clienti che consentono di commercializzare altri prodotti a questi clienti.



Problemi di archiviazione e gestione dei dati per SaaS

- **Regolamentazione:** alcuni paesi, come i paesi dell'UE, hanno leggi severe sull'archiviazione delle informazioni personali. Queste leggi possono essere incompatibili con le leggi e i regolamenti del paese in cui ha sede il provider SaaS. Se un provider SaaS non può garantire che i propri luoghi di archiviazione siano conformi alle leggi del paese del cliente, le aziende potrebbero essere riluttanti a utilizzare il proprio prodotto.
- **Trasferimento dati:** se l'utilizzo del software comporta un elevato trasferimento di dati, il tempo di risposta del software potrebbe essere limitato dalla velocità della rete. Questo è un problema per individui e piccole aziende che non possono permettersi di avere connessioni di rete ad altissima velocità.
- **Sicurezza dei dati:** le aziende che trattano informazioni riservate potrebbero non essere disposte a cedere il controllo dei propri dati a un provider di software esterno. Come abbiamo visto da una serie di casi di alto profilo, anche i grandi fornitori di cloud hanno avuto violazioni della sicurezza. Non si può pensare che i SaaS provider forniscano sempre una sicurezza migliore rispetto ai server propri del cliente.
- **Scambio dati:** se è necessario scambiare dati tra un servizio cloud e altri servizi o applicazioni software locali, ciò può essere difficile a meno che il servizio cloud non fornisca un'API accessibile per uso esterno.

Problemi di progettazione SaaS

- **Elaborazione locale/remota**

Un prodotto software può essere progettato in modo tale che alcune funzionalità vengano eseguite localmente nel browser o nell'app mobile dell'utente e altre su un server remoto.

L'esecuzione locale riduce il traffico di rete e quindi aumenta la velocità di risposta dell'utente. Ciò è utile quando gli utenti hanno una connessione di rete lenta. Tuttavia l'elaborazione locale aumenta l'energia elettrica necessaria per eseguire il sistema.

- **Autenticazione**

Se si implementa un proprio sistema di autenticazione, gli utenti devono ricordare altre credenziali di autenticazione.

Molti sistemi consentono l'autenticazione utilizzando le credenziali Google, Facebook o LinkedIn dell'utente.

Per i prodotti aziendali, potrebbe essere necessario impostare un sistema di autenticazione federato, che delega l'autenticazione all'azienda per cui lavora l'utente.

- **Leakage di informazioni**

Se si hanno più utenti provenienti da più organizzazioni, un rischio potrebbe essere che le informazioni vengano trasmesse da un'organizzazione all'altra.

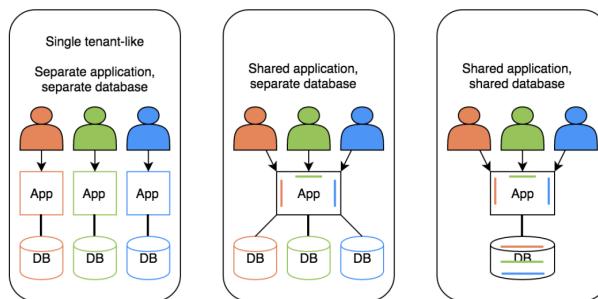
Ci sono diversi modi in cui ciò può accadere, quindi bisogna essere molto attenti nella progettazione del sistema di sicurezza per evitare che ciò accada.

- **Sistemi multi-tenant e multi-istanza**

In un sistema multi-tenant, tutti i clienti vengono serviti da un'unica istanza del sistema e da un database multi-tenant.

In un sistema multi-istanza, viene messa a disposizione una copia separata del sistema e del database per ciascun utente.

Sistemi multi-tenant



Un database multi-tenant è partizionato in modo tale che le società clienti abbiano il proprio spazio e possano archiviare e accedere ai propri dati.

Esiste un **singolo schema di database**, definito dal provider SaaS, che viene condiviso da tutti gli utenti del sistema.

Gli elementi nel database sono contrassegnati con un **identificatore tenant**, che **rappresenta un'azienda** che ha archiviato dati nel sistema

Il software di accesso al database utilizza questo identificatore tenant per fornire **"isolamento logico"**, ovvero dà la sensazione agli utenti di lavorare con il proprio database.

Vantaggi

- **Utilizzo efficiente delle risorse:** Il provider SaaS ha il controllo di tutte le risorse utilizzate dal software e può ottimizzare il software per un utilizzo efficace di queste risorse.
- **Gestione degli aggiornamenti semplificata:** È più facile aggiornare un'unica istanza di software piuttosto che più istanze. Gli aggiornamenti vengono distribuiti a tutti i clienti contemporaneamente, quindi tutti utilizzano la versione più recente del software.
- **Costi contenuti:** I sistemi multi-tenant sono solitamente più convenienti dei sistemi multi-istanza perché non richiedono risorse dedicate per ciascun tenant.

Svantaggi

- **Inesistenza di flessibilità:** Tutti i clienti devono utilizzare lo stesso schema di database con possibilità limitate di adattare questo schema alle esigenze individuali.
- **Problemi di sicurezza:** Poiché i dati di tutti i clienti sono archiviati nello stesso database, c'è una possibilità che i dati vengano trafugati da un cliente all'altro. Inoltre se c'è un security breach nel database, la vulnerabilità coinvolgerà tutti gli utenti
- **Complessità:** I sistemi multi-tenant sono generalmente più complessi dei sistemi multi-istanza a causa della necessità di gestire molti utenti.

I sistemi Multi-istance si realizzano con VMs e containers, i vantaggi e svantaggi di questo sistema sono simmetrici ai sistemi multi-tenant.

Possibili personalizzazioni per sistemi SaaS

- **Autenticazione:** le imprese vorrebbero che gli utenti si autenticassero con le proprie credenziali e non con quelle fornite dal software provider
- **Branding:** le imprese vorrebbero che l'interfaccia utente sia brandizzata in modo da rispecchiare la propria organizzazione
- **Business rules:** le imprese vorrebbero che sia possibile definire il proprio workflow sui propri dati
- **Data schemas:** le imprese vorrebbero che sia possibile estendere il modello dei dati già esistente in modo ad adattarlo alle proprie necessità aziendali
- **Access control:** le imprese vorrebbero che sia possibile definire il proprio modello di controllo degli accessi sui dati in modo da avere diverse categorie di utenti, ognuno con determinati permessi

Come estendere un database

Ci sono due modalità

Estensione del database tramite campi aggiuntivi

Vengono **aggiunte** alcune **colonne extra** a ciascuna tabella del database e **viene definito** un **profilo** del **cliente** che mappa i nomi delle colonne desiderati dal cliente a queste colonne extra.

È difficile determinare quante colonne extra includere. Se ne vengono incluse poche, i clienti potrebbero non averne abbastanza per le loro esigenze.

Se invece si includono molte colonne extra per soddisfare le esigenze di alcuni clienti, la maggior parte dei clienti non le utilizzerà, lasciando molto spazio sprecato nel database.

È probabile che clienti diversi abbiano bisogno di diversi tipi di colonne, ad **esempio** alcuni clienti potrebbero desiderare colonne con elementi di tipo stringa, mentre altri potrebbero desiderare colonne con numeri interi.

Questo problema può essere risolto mantenendo tutto in formato stringa. Tuttavia, in questo modo il provider dovrebbe fornire oppure il cliente deve già possedere un software di conversione per creare elementi del tipo corretto.

Estensione del database tramite tabelle

Un altro approccio è **consentire ai clienti di aggiungere un numero qualsiasi di campi aggiuntivi e di definirne i nomi, i tipi e i valori.**

I nomi e i tipi di questi valori sono **archiviati** in una **tabella separata**, a cui si **accede** tramite l'**ID del tenant**.

Sfortunatamente, utilizzare le tabelle in questo modo aumenta la complessità del software di gestione del database.

Devono essere gestite tabelle extra e le informazioni da esse integrate nel database.

Database Security

Quando si utilizzano **sistemi multi-tenant**, tutte le **informazioni** dei clienti vengono **archiviate** nello **stesso database**, quindi un bug del software o un attacco potrebbero portare all'esposizione dei dati di alcuni o tutti i clienti.

Problemi chiave di sicurezza:

- **Controllo accessi multilivello:**
 - Accesso ai dati da controllare sia a livello organizzativo che individuale.
 - Controllo accessi a livello organizzativo per garantire che le operazioni sul database agiscano solo sui dati di quell'organizzazione.
 - Autorizzazioni di accesso individuali per gli utenti che accedono ai dati.
- **Crittografia dei dati:**
 - Rassicura gli utenti del fatto che i loro dati non possono essere visualizzati da persone di altre aziende in caso di errori del sistema.

Database multi-istanza

I **database multi-istanza** sono sistemi SaaS in cui **ogni cliente** ha il **proprio sistema adattato** alle **sue esigenze**, incluso il proprio database e i propri controlli di sicurezza.

Esistono **due tipi di sistemi multi-istanza**:

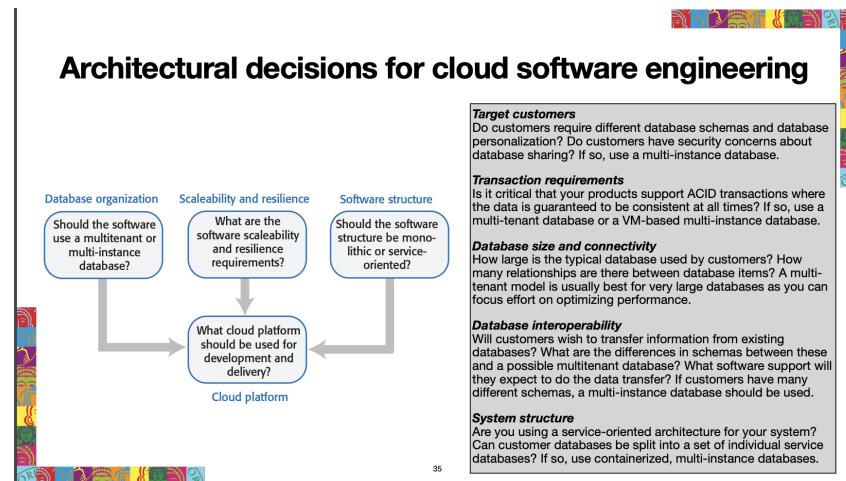
- **basati su VM:**
 - Le istanze del software e del database per ciascun cliente vengono eseguite in una propria macchina virtuale.
 - Tutti gli utenti dello stesso cliente possono accedere al database di sistema condiviso.
- **basati su container:**
 - Ogni utente ha una versione isolata del software e del database in esecuzione in una serie di container.
 - Questo approccio è adatto a prodotti in cui gli utenti lavorano principalmente in modo indipendente, con una condivisione dei dati relativamente ridotta.

Vantaggi:

- **Flessibilità:** ogni istanza del software può essere personalizzata e adattata alle esigenze del cliente. I clienti possono utilizzare schemi di database completamente diversi ed è semplice trasferire dati da un database del cliente al database del prodotto.
- **Sicurezza:** ogni cliente ha il proprio database, quindi non vi è alcuna possibilità di perdita di dati da un cliente all'altro.
- **Scalabilità:** le istanze del sistema possono essere scalate in base alle esigenze dei singoli clienti. Ad **esempio**, alcuni clienti potrebbero richiedere server più potenti di altri.
- **Resilienza:** in caso di errore del software, probabilmente questo influenzerà solo un singolo cliente. Gli altri clienti possono continuare a lavorare normalmente.

Svantaggi:

- **Costo:** è più costoso utilizzare sistemi multi-istanza a causa dei costi del noleggio di molte VM nel cloud e dei costi di gestione di sistemi multipli. Poiché il tempo di avvio è lento, le VM potrebbero dover essere noleggiate e mantenute in esecuzione continuamente, anche se la domanda del servizio è molto bassa.
- **Gestione degli aggiornamenti:** molte istanze devono essere aggiornate, quindi gli aggiornamenti sono più complessi, soprattutto se le istanze sono state adattate alle esigenze specifiche del cliente.



Resilienza e Scalabilità

La **scalabilità** di un sistema riflette la sua **capacità di adattarsi automaticamente ai cambiamenti di carico** su quel sistema.

La **resilienza** di un sistema riflette la sua **capacità di continuare a fornire servizi critici in caso di guasto** del sistema o di utilizzo dannoso del sistema.

Per ottenere la **scalabilità** in un sistema, è necessario renderlo **compatibile all'aggiunta di nuovi server virtuali (scaling-out)** o **all'aumento della potenza di un server di sistema (scaling-up)** in risposta all'aumento di carico.

Nei sistemi basati su cloud, l'**approccio più utilizzato** è lo **scaling-out** piuttosto che lo scaling-up. Il **software deve essere organizzato** in modo che i **singoli componenti** software possano essere **replicati ed eseguiti in parallelo**.

Per ottenere **resilienza**, è **necessario essere in grado di riavviare rapidamente il software dopo un guasto** hardware o software.

Utilizzo di un sistema stand-by per resilienza

La **resilienza** si basa sulla **ridondanza**:

- Vengono mantenute **repliche** del **software** e dei **dati** in posizioni diverse.
- Gli **aggiornamenti** del **database** vengono eseguiti in **mirroring** in modo che il **database di stand-by sia una copia funzionante del database operativo**.
- Un **sistema di monitoraggio** controlla continuamente lo stato del sistema e permette il **passaggio automatico al sistema stand-by in caso di guasto** del sistema operativo.

Idealmente, si dovrebbero utilizzare server virtuali ridondanti che non sono ospitati sullo stesso computer fisico e posizionare i server in posizioni diverse. Idealmente, questi server dovrebbero

trovarsi in data center diversi. In caso di guasto di un server fisico o di un guasto più ampio del data center, l'operazione può essere commutata automaticamente sulle copie del software altrove.

System Structure

L'ingegneria del software ha usato in modo estensivo l'**approccio object oriented** per lo **sviluppo di sistemi client-server costruiti intorno ad un database condiviso**.

Un sistema fatto in questo modo è, logicamente, **monolitico** con una distribuzione su diversi server su cui girano grandi componenti software. La tradizionale architettura client-server multi-tier si basa su questo modello di sistema distribuito.

L'alternativa alla struttura monolitica è l'**approccio service-oriented** dove il **sistema** è **decomposto a grana fine**, in cui i **servizi** sono **stateless**

Dato che sono stateless, **ogni servizio è indipendente** e può essere **replicato, distribuito e migrato** da un server all'altro.

Questo approccio è ideale per software cloud-based, in particolare per lo sviluppo di servizi in containers.

▼ Lezione 11 19/04/2024

Recap lezione precedente

Nella scorsa lezione abbiamo visto il cloud, in particolare cloud pubblico.

Con cloud pubblico intendiamo che le risorse che una persona o un'azienda acquista sono messe a disposizione da vendor noti come Amazon, Microsoft e Google.

Nell'ultimo periodo si sta sviluppando un grosso movimento verso il **cloud privato**, cioè cloud sotto il controllo di autorità e non di aziende multinazionali.

In particolare è noto che le PA stanno cercando di andare verso queste tipologie di soluzioni, cioè di avere del cloud che sia di dominio della PA anche se non lo gestisce a livello tecnico.

Il dibattito tra cloud pubblico, privato o dominio amministrato è aperto e c'è una grande diversità di opinioni.

Esempio di non integrazione di servizi

Oggi siamo costretti ad andare al comune di residenza per prendere il certificato di nascita e portarlo ad un altro ente che lo metterà a sua volta in un pc.

Purtroppo, la PA si basa sull'autonomia decisionale e di conseguenza non c'è uniformità, ogni ente ha fatto le sue scelte tecnologiche e contrattuali, per cui i software non sempre riescono a comunicare e di conseguenza c'è un problema che viene portato avanti da tempo.

L'Emilia Romagna ha fatto da apri pista per i CUP (centri unici prenotazione) per cercare di avere un unico punto dove è possibile vedere la disponibilità di tutte le strutture ospedaliere, ASL e centri privati convenzionati.

Una volta che ho messo insieme le strutture posso fare cose complesse, per **esempio** sviluppare una strategia di prenotazione di varie visite in un unico giorno.

Cooperazione applicativa: ogni ente ha le proprie tecnologie indipendenti, ma tutti espongono un livello di visibilità minimale che consente di organizzare dei servizi trasversali.

Architettura a servizi/microservizi

L'idea nel passato era quella di una condivisione a monte dell'informazione

Esempio PA: una serie di comuni ha la propria base di dati e a monte doveva essere creata un'integrazione che ognuno deve implementare, cioè doveva esistere un agente che integrava dati e informazioni in un modello progettuale e tecnico condiviso.

Invece, il **modello a microservizi** ha alla base l'idea di **avere degli standard e di rendere noto ciò che si fa agli altri player**.

Abbiamo la libertà degli attori in quanto **utilizzando degli standard noti è possibile la comunicazione tra altri software**.

In particolare, la prima cosa da fare quando si adotta questo modello è quello di **mettere a disposizione delle API** con cui altri attori possano creare i loro servizi

Perché ci siamo innamorati dei servizi?

Perché sono **stateless** e quindi **riallocabili**, li posso spostare sul cloud

I servizi al mondo esterno si presentano con API a cui si può accedere da remoto

Servizi Software

Un **servizio software** è un componente software che può essere **acceduto da computer remoti tramite Internet**. Dato un input, un servizio produce un output corrispondente, senza effetti collaterali.

Si accede al servizio tramite la sua **interfaccia pubblica** e tutti i dettagli dell'implementazione del servizio sono nascosti.

I servizi **NON mantengono alcuno stato interno**. Le informazioni sullo **stato vengono archiviate in un database** o gestite dal richiedente del servizio.

Quando viene effettuata una richiesta, le **informazioni sullo stato possono essere incluse come parte della richiesta** e vengono restituite come parte del risultato del servizio.

Poiché non esiste uno stato locale, i servizi possono essere **riassegnati dinamicamente da un server virtuale all'altro e replicati su più server**.

All'esterno i servizi si presentano con la propria API

Il ruolo dell'API diventa **centrale**

Modern Web Services

L'idea dei servizi Web è emersa nei primi anni 2000.

All'inizio, si basavano su protocolli e standard XML-based come SOAP per l'interazione del servizio e WSDL per la descrizione dell'interfaccia.

Invece, i **moderni sistemi** orientati ai servizi **utilizzano protocolli di interazione più semplici e "leggeri"** che hanno costi generali inferiori e, di conseguenza, un'esecuzione più rapida.

SOAP (Simple Object Access Protocol)

Poiché XML può definire i token, **SOAP è un sottoinsieme di token che consente di fare delle chiamate ad una procedura remota**

Quindi, permette l'**invocazione del servizio**

WSDL (Web Service Description Level)

Describe i servizi, se si vuole interrogare un carrier si deve scaricare il WSDL che ci consente di interagire con il servizio.

In particolare **descrive** i servizi, cioè che **interfaccia** hanno, come chiamarli, ecc

WSDL e SOAP sono due protocolli XML Based

Perché i web services hanno fatto fatica ad affermarsi?

La visione che c'era dietro era ambiziosa, ma **pesante**

Molti **servizi moderni** sono **LIGHT**, cioè determinate operazioni vengono effettuate **offline** e quindi **NON** sono parte del **workflow della gestione dei servizi**.

I web services erano il futuro negli anni 2000

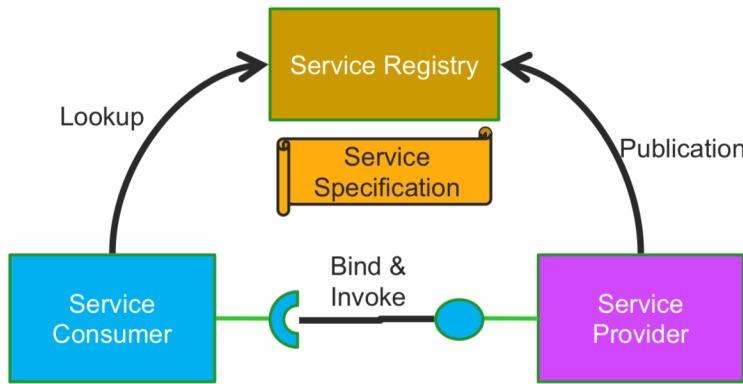
In quegli anni 2000 eravamo tutti innamorati della globalizzazione, si volevano realizzare sistemi distribuiti entreprise, cioè sotto il dominio dell'azienda.

Successivamente si è passati ad un modello inter-enterprise e poi a modelli globali.

L'idea era quella di fare un ultra late binding del servizio.

Così come succede su Android quando si vuole aprire un link, anche a livello globale ci sarà un manifesto dove qualcuno dichiarava di saper fare qualcosa.

Tutte le **aziende** che vogliono fare **servizi**, devono **pubblicare** questo **manifesto** in cui dichiarano cosa fanno, come lo fanno, come interagire con il servizio.



WSDL contiene tutte le informazioni per costruire la chiamata che verrà fatta con SOAP

Ci sono 3 livelli:

- **Descrivo** il servizio con XML (WSDL)
- **Invoco** il servizio con XML (SOAP)
- **Discovery** del servizio perché sono i servizi sono descritti con UDDI (Universal Description Discovery and Integration)

La Visione

I servizi devono essere descritti, l'interfaccia è il livello minimo di descrizione.

La visione è: ho un requisito, voglio mettere a disposizione dei servizi (anche esterni) che soddisfano il requisito, inoltre voglio una situazione **dinamicamente aperta**, cioè se cambia il requisito si possono aggiungere servizi.

Se a cambiare non è il requisito, ma le condizioni tecnologiche, l'adattamento deve essere fatto in maniera automatica.

Dietro al servizio software c'è un servizio.

Per avere l'**adattamento automatico** ho bisogno di descrivere anche la semantica, si fa attraverso l'antologia.

Quindi l'ingegnere del software scrive del software, altri lo prendono dal mercato e poi si scrive del software per linkare dinamicamente i servizi solo quando servono. Tipicamente ciò si faceva con richieste POST HTTP

SOA (Service Oriented Architecture)

Il **servizio** è un **contratto** che può avere più interfacce e soprattutto più implementazioni (anche 0).

O Implementazioni si intende che non è detto che sia già pronto per essere offerto al mercato infatti può essere la descrizione di un servizio che ancora non c'è, ma che si vorrebbe

realizzare.

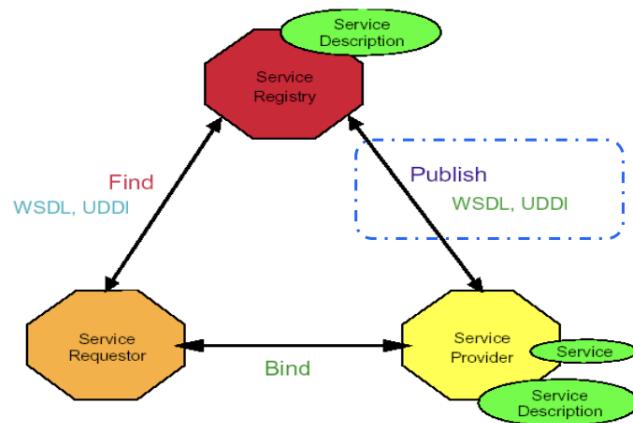
Il modo in cui si sviluppò quest'idea era un gioco a 3 attori:

- **Registro di servizi:** qui vengono conservate le **descrizioni di servizio** con o senza implementazioni
Il
registro a sua volta si offre al mondo con un'interfaccia web service.

Lo standard era **UDDI** e aveva un'architettura distribuita, è un searchable register.

- **Service Provider:** colui che esporta il servizio, utilizzando UDDI carica le descrizioni (WSDL e semantica). Quest'operazione è detta **PUBLISH**
- **Service Requestor:** colui che vuole chiamare il servizio, per farlo utilizza UDDI e ciò che ottiene è il WSDL pubblicato dal provider. L'operazione è detta **FIND**.

Il bind avviene tra REQUESTOR e PROVIDER

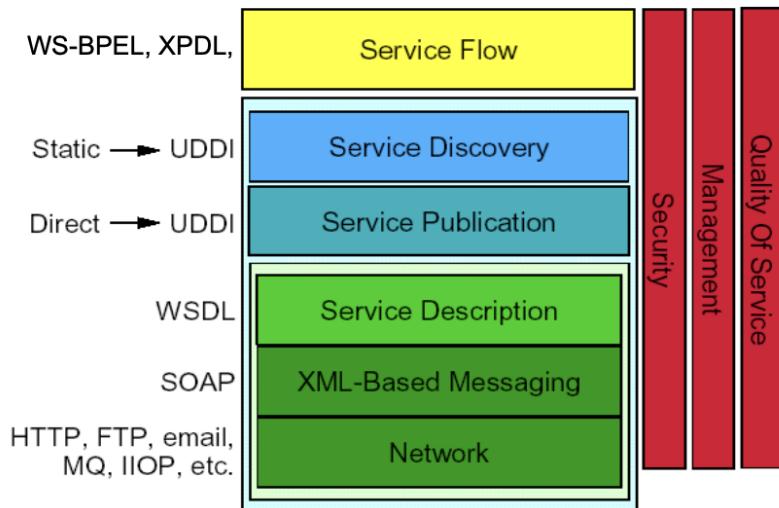


Questo modello così generale, permette di avere un'**architettura STATICA** perché il **bind** può essere risolto a **DESIGN TIME**

Web Service Stack

Web Service Stack

The Conceptual Web Services Stack



Microservices

La **differenza fondamentale** che c'è tra **microservizi e servizi** è che con questi ultimi si lavorava in un **mondo completamente aperto**, chiunque poteva andare sul registro e **pubblicare il servizio**.

L'altra differenza che possiamo notare è che i **microservizi si sviluppano in verticale** (singola responsabilità), mentre i servizi in orizzontale.

I **microservizi** sono servizi su piccola scala, **senza stato** e con una **singola responsabilità**. Vengono combinati per creare applicazioni.

Sono **completamente indipendenti**, infatti **ognuno ha il proprio codice di gestione del database e dell'interfaccia utente**.

I prodotti software che utilizzano i microservizi hanno un'architettura a microservizi.

Se è necessario creare **prodotti software basati su cloud** che siano adattabili, scalabili e resilienti, è consigliato progettarli attorno a un'**architettura a microservizi**.

Caratteristiche dei microservizi

- **Self Contained**: i microservizi non hanno dipendenze esterne, gestiscono da soli i loro dati e implementano da soli la loro UI
- **Lightweight**: comunicano attraverso protocolli leggeri, così facendo gli overhead di comunicazione sono bassi
- **Implementation independent**: possono essere implementati tramite diversi linguaggi di programmazione e possono usare diverse tipologie di tecnologie nella loro implementazione
- **Independently deployable**: ogni microservizio ha il proprio processo

Comunicazione tra microservizi

Essi **comunicano attraverso lo scambio di messaggi**, un messaggio che viene scambiato tra due microservizi include delle informazioni amministrative, una richiesta al servizio e i dati necessari per inviare la richiesta.

Alla richiesta corrisponde una risposta, per **esempio**:

un servizio di autenticazione invia un messaggio ad un servizio di login contenente il nome dell'utente che si vuole autenticare ⇒ la risposta potrebbe essere un token associato all'username oppure un errore.

▼ Lezione 12 22/04/2024

Nella lezione precedente abbiamo visto l'**architettura a servizi**, argomento articolato che permetteva di dare vita ad un mondo di applicazioni software.

Si **definiva** un flusso di esecuzione, un **workflow** che poteva essere anche assegnato **staticamente**.

Altri modi per assegnare un workflow era tramite **descrizioni semantiche** per cui **a run-time** si cercava di effettuare un binding tra i servizi che permettevano di risolvere il problema specificato (WDSL e UDDI)

L'intero stack **NON** ha mai avuto grande applicazione pratica

Ciò che è rimasto di questa architettura è **la visione di un sistema come un insieme di parti indipendenti che messe insieme possono risolvere un problema**.

Per cui, nasce l'idea che un sistema può essere costruito utilizzando componenti che fanno parte di altri domini amministrativi (sviluppati da enti diversi dal nostro)

Tra i vari punti della conferenza del 1968 (lezione 0), Mc Hillory aveva già ipotizzato l'idea di ingegneria del software fatta per componenti, che oggi è lo standard, infatti l'industria è andata da tempo verso un'ipotesi di **componentizzazione**.

(I ponti si fanno in fabbrica e poi si assemblano sul posto non si fanno colate di cemento sul posto)

Nasce il concetto di System Integration, con la novità che non si paga più l'algoritmo, ma esso viene sviluppato, messo online e pagato solo quando deve essere usato.

Microservizi

Li definiamo micro perché cambia la logica, sono su scala più piccola, **sviluppati in forma verticale**.

Quando sono nati i servizi si pensava ad una forma orizzontale, ma questo ha senso solo nell'ambito di un workflow.

I **microservizi** sono **componenti autonome, leggere** che **fanno una sola cosa**, replaceable

Infatti, si sposano molto bene con il cloud, la componentizzazione e containerizzazione

L'idea di fondo è che i servizi esistevano in un ecosistema, c'era un modo per farli comunicare attraverso una base di dati condivisa, mentre i microservizi hanno il proprio modello di dati e gestiscono la loro UI in modo indipendente

Esempio

Ipotizziamo un sistema di autenticazione che prevede:

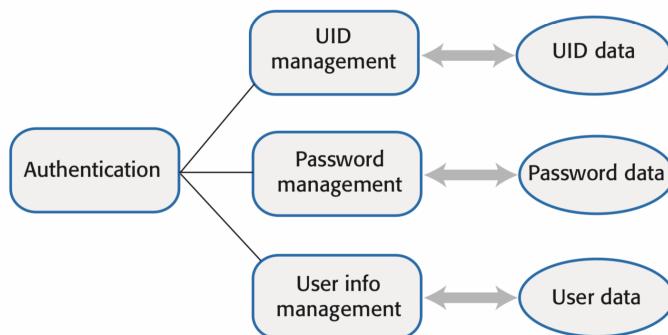
- user registration
- autenticazione ⇒ possono essere autenticazione con user id e password
- livelli di autenticazione ⇒ autenticazione a due fattori
- management delle info dell'utente
- reset password dimenticata

Queste sono le feature che ci aspettiamo

Ognuna può essere realizzata come un servizio separato, ma che usano un database condiviso

Questa è la scelta che se torniamo al quadro di prima fa riferimento ai web services

Se cerchiamo una soluzione a microservizi dobbiamo spacchettare ciascun feature in un insieme di operazioni di livello più basso



I microservizi comunicano tra loro scambiando messaggi, quindi per poter realizzare qualche funzionalità li devo coordinare.

Oltre al payload spesso vengono inviate anche informazioni di tipo amministrativo

Information hiding, abstraction, cohesion and coupling

Un microservizio deve essere:

- **coesio**: deve essere disaccoppiato dagli altri e infatti abbiamo tolto con prudenza l'idea che condividessero dati
Ogni microservizio deve avere una **singola responsabilità**.

La coesione è una metrica che misura la forza con cui le parti che ho messo in un contenitore stanno bene insieme.

- **coupling**: numero di dipendenze che un microservizio ha con gli altri

Cos'è una architettura a microservizi?

Un'architettura a microservizi è uno **stile architettonico**, quindi un modo collaudato e testato per implementare un'architettura software logica.

Questo stile **affronta due problemi tipici** delle applicazioni **monolitiche**:

- la necessità di **ricostruire, testare** nuovamente e **ridistribuire l'intero sistema in caso di qualsiasi modifica**.
Ciò può essere un processo lento, poiché le modifiche a una parte del sistema possono influire negativamente su altri componenti.
- la necessità di **scalare** il sistema in caso di una crescita della domanda.

I microservizi sono **autosufficienti** e vengono eseguiti in **processi separati**.

Ciò significa che possono essere **interrotti** e **riavviati senza influire** su altre parti del sistema.

Se aumenta la domanda di un particolare microservizio, è possibile creare e distribuire rapidamente delle **repliche** di quel microservizio.

In genere, questo è molto più economico del "ridimensionamento" mediante l'utilizzo di un server più potente.

Decomposition guidelines

- **Bilanciare la funzionalità e le prestazioni del sistema**: avere servizi verticali significa che i cambiamenti sono limitati a un numero minore di servizi, ma richiedono comunicazioni per implementare le chiamate che effettua l'utente.
Ciò rallenta un sistema a causa della necessità per ciascun servizio di impacchettare e decomprimere i messaggi inviati da altri servizi.
- **Seguire il principio di "chiusura comune" (common closure)**: gli elementi di un sistema che probabilmente verranno modificati contemporaneamente dovrebbero essere posizionati all'interno dello stesso servizio. La maggior parte dei requisiti nuovi e modificati dovrebbe quindi influire solo su un singolo servizio.
- **Associare i servizi alle capability aziendali**: una capability aziendale è un'area distinta di funzionalità aziendale che è responsabilità di un individuo o di un gruppo.
È necessario identificare i servizi necessari per supportare ciascuna capability aziendale.

- **Progettare i servizi in modo che abbiano accesso solo ai dati di cui hanno bisogno:** se esiste una sovrapposizione tra i dati utilizzati da diversi servizi, è necessario un meccanismo per propagare le modifiche ai dati a tutti i servizi che utilizzano gli stessi dati.

▼ Lezione 13 29/04/2024

Comunicazione tra servizi

I microservizi comunicano tramite lo **scambio di messaggi**, che includono informazioni sul mittente del messaggio, nonché i dati che sono l'input o l'output della richiesta.

Durante la progettazione di un'architettura a microservizi, è necessario stabilire uno **standard per le comunicazioni** che tutti i microservizi devono seguire. Alcune delle decisioni chiave da prendere sono:

- L'interazione del servizio deve essere sincrona o asincrona?
- I servizi devono comunicare direttamente o tramite un middleware di messaggistica?
- Quale protocollo deve essere utilizzato per i messaggi scambiati tra i servizi?

Interazione sincrona e asincrona

- **Sincrona:** un servizio attende che un altro servizio risponda prima di continuare.
- **Asincrona:** i servizi eseguono le attività contemporaneamente.

Comunicazione diretta e indiretta

- **Diretta:** i servizi si inviano messaggi direttamente, quindi possono invocarsi uno con l'altro, ma necessitano di una **referenza** dell'altro.
- **Indiretta:** i servizi comunicano tramite un broker di messaggi che gestisce l'instradamento e la consegna dei messaggi.

SOAP è un modo per effettuare comunicazione **diretta**, remote procedure call sul web

I servizi **REST** che costruiremo noi adotteranno una **comunicazione diretta e stateless** dove la **referenza** che A deve possedere per poter chiamare B è un **end point**, abbiamo bisogno di una **URI**

L'alternativa può essere la **comunicazione attraverso bus**, che si può implementare attraverso un **message broker**, posso mettere in comunicazione due servizi partendo dall'idea di **topic** e **keywords**

Esempio: RabbitMQ è molto utilizzato per la comunicazione tramite message broker, ha un bus che implementa il concetto di coda per mettere servizi in comunicazione tra di loro

Design dei dati per i microservizi

Quando si parla di dati il problema principale che si riscontra è la loro **consistenza**.

Le **transazioni ACID** (Atomic, Consistence, Isolated, Durability) **NON** funzionano con i microservizi perché raggruppano una serie di aggiornamenti dei dati in un'unica unità in modo che tutti gli aggiornamenti vengano completati o che nessuno venga completato.

Eventually Consistency

Al fine di avere una **verticalità spinta si accetta di avere un livello di inconsistenza** che però prima o poi verrà sanato

Si parla di **eventually consistent** dove con "eventually" si intende che prima o poi sarà assicurata la consistenza

Deve quindi esserci un meccanismo di propagazione o allineamento.

Seguendo questa logica, le transazioni vengono fatte **localmente** al servizio e il risultato viene messo in uno spazio comune in modo che possa essere **propagato** in tutti gli altri database

È possibile implementare la consistenza "eventuale" mantenendo un **log delle transazioni**.

Quando viene apportata una modifica al database, questa viene registrata in un log degli "aggiornamenti in sospeso".

Altre istanze del servizio esaminano questo registro, aggiornano i propri database e indicano di aver apportato la modifica.

Service Coordination

Idealmente, le architetture a microservizi sono **fully verticali** (anche il blocco di GUI è indipendente)

Nella pratica però questo **NON** succede sempre

Esistono meccanismi di **workflow**

Dato che la maggior parte delle interazioni di un utente prevedono una serie di interazioni eseguite con un ordine specifico, allora viene definito un workflow.

L'integrazione avviene attraverso i workflow.

I due meccanismi per coordinare i servizi sono:

- **service orchestration:** è presente un meccanismo centralizzato (**motore**), il quale possiede la descrizione del workflow, possibilmente serve una sua interfaccia ed **esegue la descrizione del workflow** chiamando i servizi passo per passo (lo abbiamo visto nell'architettura SOA)
- **coreografia dei servizi:** **NON** c'è un meccanismo di workflow centralizzato, ma di **caratterizzazione e propagazione** degli **eventi** a cui i servizi rispondono

Quando avviene un evento alcuni servizi che hanno sottoscritto o dichiarato interesse per determinati eventi vengono notificati e poi ci sarà un meccanismo per permettere la risoluzione della richiesta

RESTful Services

Il concetto fondamentale è la **risorsa**, i **microservizi** sono **stateless** e non sono altro che lo **scambio di rappresentazione di una risorsa**.

Nel nostro caso specifico la risorsa avrà una rappresentazione XML SOAP o attraverso una notazione nested, cioè JSON.

La **comunicazione** tra microservizi è un **continuo scambio di risorse**, per la maggior parte in rappresentazioni JSON, **ognuna identificata da un URI**.

I principi REST

I servizi REST sono esattamente coerenti con il web

Su ogni end-point, che è una risorsa, le operazioni che si effettuano sono quelle HTTP (GET, POST, PUT, DELETE), mentre prima nel payload XML c'era la semantica delle operazioni.

Operazioni

- Create
- Read
- Update
- Delete

Put deve essere utilizzato solo per aggiornare mentre POST per creare

Inoltre put è idempotente.

Continuos Deployment

Oggi si ha un'**automazione completa del processo di deployment**

Continuos deployment pipeline: quando si effettua un commit o resolution di un branch parte in automatico una batteria di unit test e se questi vengono superati, viene eseguita la parte di containerizzazione fino ad arrivare al rilascio della nuova istanza del servizio.

Esempio Servizio RESTful con Carmine

SOA afferma che tutti i componenti che compongono l'app vengono chiamati servizi e possono essere coordinati per creare un'applicazione

Basata su concetti che non ci spiega, bus, tecnologia di comunicazione, **definizione di contratti**

Ogni servizio va implementato come entità a sé ed esposto sulla rete

Ogni componente deve essere incentrato sul proprio dominio

Architettura a microservizi

L'idea di programmare a servizi non è nuova, ma si parla di microservizi dal 2014 cioè da quanto sono stati introdotti i **DevOps**, dove il deployment deve essere leggero e il sistema deve essere in grado di funzionare anche quando si modifica qualcosa

- grana fine, specializzati in un task o solo uno
- loosely coupled
- comunicano tra di loro con meccanismi leggeri
- business driven development, intercettare una serie di identità specifiche
- polyglot programming and persistence
- lightweight container deployment

Logica monolitica e a microservizi

Monolite: qualsiasi cosa dove le funzionalità sono intrecciate, non si riescono a spezzare i domini applicativi

Ogni **microservizio può essere visto come un piccolo monolite** che gestisce un dominio estremamente ristretto.

Programmazione in C: spaghetti

Programmazione Zimeo: lasagna

Programmazione Canfora: ravioli

▼ Lezione 14 03/05/2024

Abbiamo visto finora l'ingegneria del software come modo per organizzare il lavoro, architetture, tool di lavoro per automatizzare i processi, Maven per gestire le dipendenze nel processo di build

Ora vedremo due fasi importanti del life cycle di un sistema software:

- software design
- software testing

Software Design

Approccio agile, non sequenziale

Se avessimo usato approccio waterfall oggi dovremmo seguire uno schema di sviluppo che parte con il no level design, poi si decide cosa il sistema deve fare per rispondere ai requisiti, successivamente progettazione di alto livello e infine livello di progettazione delle parti (una volta che conosco quali sono i pezzi devo andare a svilupparli)

In altri testi troveremo le due fasi come low e high level

Vedremo che forma dare al codice

The Twelve Days of Christmas

Martin Robillard inizia il libro con una poesia

Ha una struttura particolare, in qualche modo generativa (incrementale)

Una poesia del genere può essere generata facilmente tramite un programma per generare una sequenza del genere dato che è una regola abbastanza semplice

Che cosa significa scrivere del codice di qualità?

Innanzitutto, bisogna interrogarsi su chi è il nostro interlocutore, quali sono gli obiettivi che ci poniamo nello scrivere un codice di qualità?

L'obiettivo è scrivere codice che un interlocutore umano possa leggere, comprendere e modificare

Ian Philipps vinse una gara per il miglior codice offuscato, cioè codice pensato per NON essere leggibile da un umano

Il codice è:

```
main(t,_,a ) char* a;{return!0<t?t<3?main(-79,-13,a+main(-87,
1-_ ,main(-86, 0,a+1 )+a)):1,t<_?main( t+1, _, a ):3,main(-94,
-27+t, a )&&t == 2 ?_<13 ? main ( 2, _+1,"%s %d %d\n" ):9:16:
t<0?t<-72?main( _, t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,*{*+\
,/w{ %+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'\
3,{w+K w'K:'+'e#';dq#'1 q#' +d'K#!/+k#;q#' r}eKK#}w'r}eKK{nl]\\
'/#;#q#n'){})w'){}){nl]'/+#n';d}rw' i;# ){nl]!/n{n#'; r{#w'r\
nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' iwk{KK{nl]!/w{\
%'1##w#' i; :{nl]/*{q#'1d;r'}{nlwb!/*de}'c ;;{nl'-{}rw]'/+,\
}##'*#nc,' ,#nw]'/+kd'+e}+;#'rdq#w! nr' /' ) }+}{rl#' {n' '}# \
}'+}##(!!/) :t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main
((*a == '/')+t, _,a+1):0<t?main ( 2, 2 , "%s") :*a=='/'||main
(0,main(-61,*a,"!ek;dc i@bK' (q)-[w]*%n+r3#l,{}:\nuuwloca-O;m\
.vpbks,fxntdCeghiry"),a+1);}
```

Questo ha un problema, cioè se si chiede di modificarlo sarà molto difficile in quanto è offuscato al massimo

Bisogna fare in modo che l'interlocutore umano sia il nostro focus, dobbiamo scrivere codice che non solo il compilatore, ma un altro essere umano possa leggere, comprendere, modificare e migliorare

Software Quality

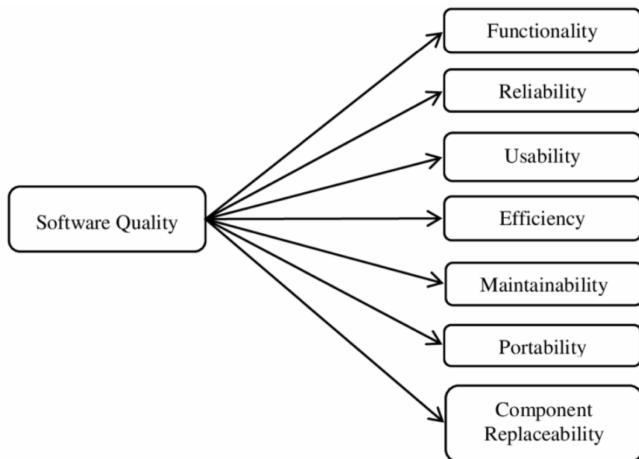
Abbiamo capito che il nostro interlocutore è anche l'umano

Ogni scelta ha la sua **rilevanza**

La software quality è un concetto difficile da definire

Quando si parla della qualità del software si va a finire con modelli complicati e quasi sicuramente gerarchici

Un modello di qualità è:



Il più delle volte i livelli mostrati sono a loro volta difficili da definire e si finisce con dei **modelli gerarchici su più livelli**

Se prendessimo questa strada dovremmo discutere con modelli **complessi** e che richiederebbero metriche per confrontare vari modelli

Software design

Cosa si intende per software design?

È il processo con il quale costruiamo le astrazioni dei dati, gli algoritmi e organizziamo queste astrazioni in applicazioni software funzionali

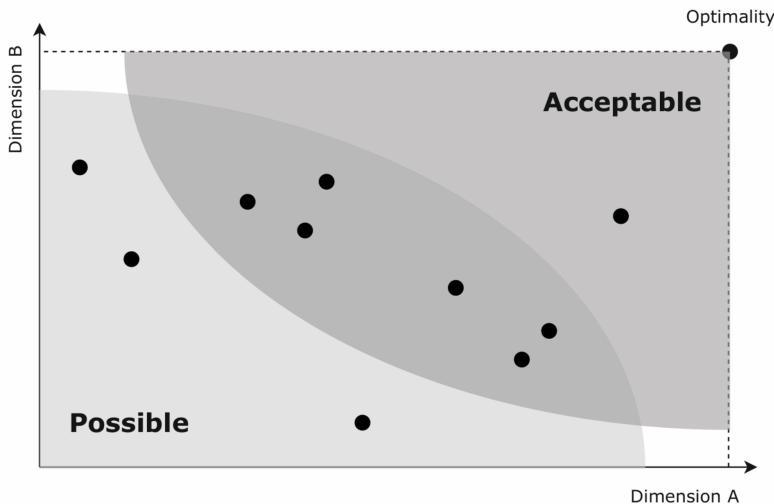
È un processo di **decision making** ed è altamente euristico

Design Space

Quando si parla di software design spesso viene mostrato un grafico bidimensionale, ma che in realtà si sviluppa in uno spazio a n-dimensioni

Sostanzialmente quello che si vuole dire è che **fare software design significa delimitare lo spazio di tutte le soluzioni**

In particolare, ci si concentra sullo spazio delle soluzioni accettabili e possibili da realizzare



Anche qui, viene fatto un tradeoff

Capturing Design Knowledge

Per **delimitare lo spazio**, ci viene in aiuto il fatto che spesso le soluzioni ingegneristiche possono essere descritte mediante **modelli**

L'ingegneria da sempre ha cercato di fare tesoro delle soluzioni che funzionano

La letteratura partendo da un'esigenza e da una conoscenza pigna describe come si comporterebbero diversi approcci

Nel software questa cosa è stata portata all'attenzione da un libro che ha introdotto il concetto di **Design Patterns**, noto come The Gang of Four book, quattro cristiani che hanno avuto voce in capitolo nel movimento agile ecc...

Hanno fatto un **catalogo di soluzioni** tipiche

In pratica, hanno notato che esistono dei **problem**i che si **presentano** nello sviluppo del software in maniera abbastanza **frequente**, quindi è **inutile inventare soluzioni** fantasiose perché **per questi problemi esiste una configurazione del software** (classi interfacce metodi) **che ha dimostrato di essere efficace**

All'inizio, il catalogo era formato da 23 pattern, oggi ce ne sono centinaia categorizzati in famiglie

I pattern sono un modo concreto di costruire una classe o grafi di classi

Esempio: voglio assicurarmi che una classe sia istanziata in singola istanza. Esiste un pattern, **singleton**, che mi indica come fare

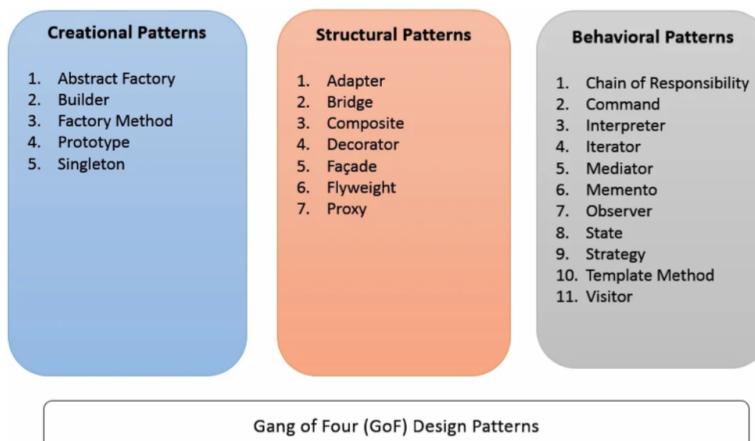
GoF patterns

Pattern: soluzioni che hanno dimostrato di funzionare a classi di problemi che si ripetono frequentemente

L'approccio in altri libri è molto più sistematico

Ogni pattern ha:

- nome
- descrizione astratta del problema
- soluzioni
- conseguenze



Antipatterns

Esistono anche gli antipattern, cioè **bad ideas**

In pratica, sono state codificate anche soluzioni che **NON funzionano**, per cui è meglio evitarle

Quando sono stati definiti gli antipattern sono stati definiti con l'idea di **refactoring**

Cioè Fowler voleva avere un catalogo di bad smells su cui fare il triggering del refactoring

Ciò che viene fatto è **definire una serie di antipattern** e una volta che ho trovato questo bad smell allora **applico il refactoring**.

Design pattern e antipattern sono strumenti operativi per la realizzazione dei principi di:

- **encapsulation**
- **types and interfaces**
- **object state**
- **composition**
- **inheritance**

- **inversion of control:** avremo un main che effettua accoppiamenti tra sorgenti di eventi e codice che è interessato all'evento, il main sarà quindi un **gestore di eventi**
- **functional design**

Design Context: gioco di carte

Implementiamo la classe `Card`

Il rettore alla nostra età avrebbe scritto:

```
int carta;
carta = 37;
```

L'ordine delle carte scelto è il seguente:



Per capire di che carta si tratta si fanno le operazioni di **modulo** e **resto**

Con il **modulo** otteniamo il **seme**

Con il **resto** otteniamo l'**offset** all'interno del seme

I pattern ci aiutano ad implementare codice, a patto di avere conoscenza dei principi

L'obiettivo in questa prima fase è di fare **ENCAPSULATION E INFORMATION HIDING**

Un altro modo per implementare il mazzo di carte è utilizzare due `int`

In particolare, uno per il seme e uno per il valore

Questo metodo è proprio un **antipattern**, detto **PRIMITIVE OBSESSION**

La stragrande maggioranza dei programmati quando si trovano di fronte ad un problema tipicamente provano a mapparlo su tipi primitivi

Per implementare il **seme** esiste l'**ENUMERATION**, dato che sarà un tipo che deve assumere 4 valori ordinati tra loro

In Java, esiste `enum`

ENUMERATION

È un modo per definire un **tipo che può assumere un insieme limitato di valori**

Definendo `enum Seme = {CUORI, QUADRI, FIORI, PICCHE}` allora da quel momento in poi `Seme` potrà assumere solo uno di quei 4 valori

Enum inoltre dà anche il vantaggio di avere un **ordinamento**

Cosa succede se devo implementare il mazzo di carte?

Scriviamo: `List<Card> deck = new ArrayList();`

Non è una buona soluzione perché per pescare faremo un `get` di `deck[0]` e facendo così il codice NON sarà intelligibile

Il deck nel costruttore richiamerà il metodo shuffle, il quale creerà la collezione

Reference Escape

Che significa che una variabile, una classe è privata?

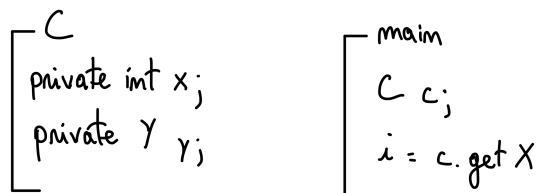
Supponiamo di avere una classe C, supponiamo che ha un'unica variabile di istanza che è privata, è un'intero e si chiama x

Che vuol dire che è privata?

Vuol dire che se un qualunque client ha un'istanza di C a sua disposizione, quindi ha una referenza a C, allora il **main NON può fare** `c.x = 0` a meno che la classe C non lo **consenta** con dei **metodi ad hoc** (che sono i setter e i getter).

Questa cosa però **NON** è sempre vera

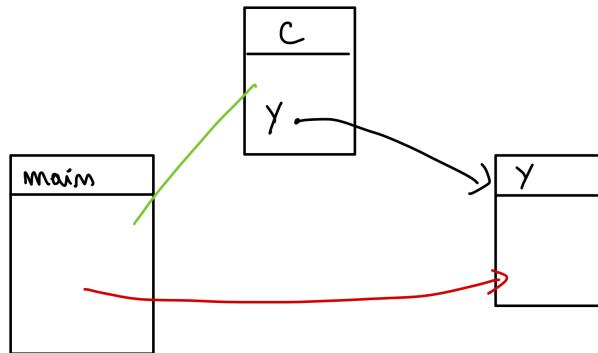
Supponiamo di avere una classe C che ha come variabile d'istanza un'altra classe Y



A questo punto, se nel *main* facciamo: `y = c.getY` otteniamo un **riferimento**

Se utilizzassimo il riferimento per cambiare qualcosa all'interno di Y nel main, allora il cambiamento sarà visto anche da `c`

Abbiamo causato una **fuga di referenza**



Supponiamo di avere una classe C che dichiara un `int x` e una riferimento ad una classe Y e che questa abbiamo un metodo `setStato`

Abbiamo un client di questa situazione che fa:

```
a = c.getX
b = c.getY
```

Il destino di `a` e di `x` di C è diverso

Invece se uso il riferimento memorizzato in `b` posso cambiare lo stato

Una **soluzione all'escape di referenza** è permettere questa situazione **solo a classi immutabili**

Si ha una **fuga di referenza** se:

- mi faccio sfuggire verso un client una referenza
- accetto ad un client una referenza (**esempio**: un costruttore che prende come parametro un array)

Nel nostro esempio delle carte, la **soluzione è rendere immutabile** il mazzo

Ciò ci fa capire che **NON** bastano i **modificatori di accesso (public, private, protected)** perché l'**esecuzione dinamica è complessa**

È importante che la **radice** della nostra **gerarchia dei tipi** sia **immutabile**, in questo modo gli oggetti sono **safe**

Enum è immutable

L'escape reference può capitare in **3 occasioni**:

- se si **ritorna** una **referenza ad un oggetto interno**
- se si ha come **stato** una **referenza**, spesso accade quando si ha uno stato che è una collezione e ciò si può evitare restituendo una copia
- **condividendo strutture dati**

Esempio: LISTA

Una lista è composta da un insieme di coppie che sono formate da un valore e un riferimento alla coppia successiva

Supponiamo di voler creare la classe lista e di avere la classe Coppia

La classe Coppia è fatta in questo modo:

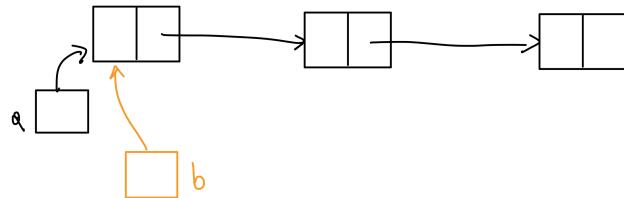
```
Object val;  
Coppia next;
```

La classe Lista invece:

```
Coppia testa;
```

Supponiamo di avere una Lista a e poi fare `Lista b = a`

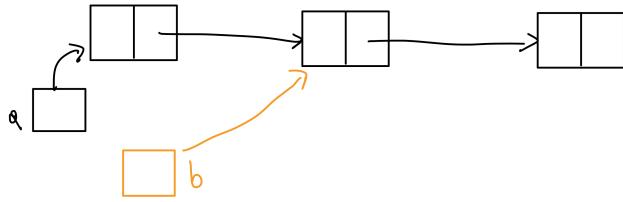
Ciò che abbiamo fatto **NON** è una copia della lista, bensì abbiamo creato un **alias**



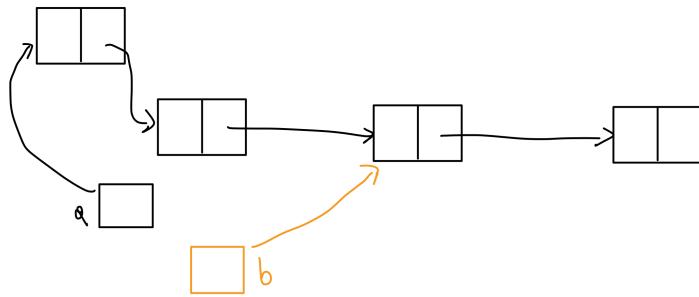
Quindi se cambio il terzo elemento di a, sarà cambiato anche quello di b

Questo fenomeno è detto **interferenza** ed è molto simile al problema di escape reference

Supponiamo di voler rimuovere il primo elemento da b, ciò che succede è:



Ora, se aggiungiamo un elemento in testa ad a, la situazione sarà



Quindi, lavorando **in testa** **NON** ci sono problemi di **interferenza**

Una **strategia** può essere quella di **accettare la sovrapposizione** fin quando **NON** ci sono **problem**i, salvo poi fare la **copia** per avere due **liste indipendenti**

La **duplicazione** può essere fatta con un criterio **selettivo**, cioè duplicare solo da un certo indice in poi

Quindi o rendiamo le cose **immutabili** oppure **restituiamo delle copie**

Che succede se per creare una carta passo "Cuori" e "null"?

Questa situazione si può trattare in 3 modi:

- **bocciato con picchiatura:** si mappa su un valore speciale dell'output un'**operazione** che **NON** può **avvenire** perché i dati in **input** **NON** sono **corretti**. Codifico l'errore in un messaggio di restituzione, ma è da evitare perché il sistema si inghiomba se non si fa un controllo
- **promosso a patto che mi convincete:** si può risolvere con le **eccezioni**, le quali sono la stessa cosa del punto precedente, ma **costringono** a verificare. Hanno grande eleganza, ma vanno cercate sempre
- **promosso con soddisfazione:** usare le **asserzioni**, cioè un meccanismo che permette di realizzare il concetto del **design by contract**, cioè si garantisce la correttezza del funzionamento come metodo a patto che venga rispettato il contratto pubblico esposto

Design Contract

L'asserzione fa il controllo

In Java: `assert`

In questo modo si controlla se viene rispettato il **contratto pubblico**, quindi se l'utente fornisce in input valori **NON** corretti, allora il metodo **NON** garantisce il funzionamento

L'assert verifica se i valori sono leciti

Assomiglia molto ad un costrutto `if` o `throw`

Quando si compila un file Java, si possono aggiungere i tag *ea (enable assertion)* i quali permettono di utilizzare le **asserezioni** come una sorta di eccezioni perché vengono **testate a run-time**

Quando il sistema è a regime e siamo sicuri che sia tutto a posto, allora si ricompila il codice senza asserzioni e queste rimangono nel codice sotto forma di commenti

Per abilitarle:

```
javac -ea Classe.java
```

Per descrivere gli assert nella documentazione si utilizzano i tag **pre** e **post**

▼ Lezione 15 06/05/2024

Oggi continuiamo a parlare di Design, vedremo una serie di feature che abbiamo visto in parte in quanto programmatore Java e vedremo che sono delle implementazioni di pattern

4 pattern e un paio di antipattern e come si collocano nell'API di Java

Types and interface

Abbiamo parlato di antipattern e abbiamo accennato il concetto di primitive type obsession e di rendere esplicito ciò che è l'assunzione che come codice faccio rispetto a chi mi andrà ad invocare (assert)

Oggi parleremo di tipizzazione ed interfacce e ci interrogheremo sullo stato, introducendo dei pattern e antipattern

Che cos'è un'interfaccia?

In programmazione abbiamo usato il termine con significati diversi

Tutte le **interazioni** che avvengono **tra gli oggetti** che compongono un sistema software
avvengono attraverso scambi di messaggi che rispondono a delle **interfacce pubbliche**

L'**interfaccia** di una classe è l'**insieme delle risorse che la classe mette a disposizione degli altri.**

Per buona abitudine, noi diciamo che un'interfaccia pubblica è un **insieme di metodi** visibili alle altre classi

All'interno della programmazione OO assumere un'interfaccia permette di **gestire il polimorfismo**

L'eventuale client che cosa può sapere di una classe? Tutto quello che c'è scritto nell'interfaccia

Decoupling behavior from implementation

Riprendiamo l'**esempio** del card deck

Nell'interfaccia che abbiamo definito c'è un **accoppiamento forte** tra quale deve essere l'**interfaccia pubblica dell'oggetto e l'idea di implementazione** da parte di uno sviluppatore.

Tuttavia, possono esistere delle situazioni dove si vuole definire prima l'interfaccia rispetto all'effettiva realizzazione della classe

Si vuole **disaccoppiare la definizione dell'interfaccia dalla relativa implementazione**

Perché è importante?

Per **esempio** per il gioco delle carte che siamo vedendo, potrebbero esistere giochi diversi che hanno regole diverse per gli stessi behaviors, come può accadere che il drawing di una carta avvenga da un mazzo o da un insieme di mazzi diversi o in altri giochi il metodo per pescare dipende dallo stato del gioco

Allora nasce l'esigenza di **disaccoppiare il behavior dalla relativa implementazione**

Java consente di **definire esplicitamente** l'interfaccia pubblica di un insieme di risorse ognuna delle quali può implementare il comportamento a modo suo

Il **vantaggio** è che l'interfaccia permette di **definire un tipo**, anche se **NON** si **possono istanziare oggetti** proprio perché è la **definizione di un behaviour senza implementazione**

Se una classe decide di implementare l'interfaccia, allora ha l'**obbligo** di implementarne **tutti i metodi**

L'**interfaccia** e la **classe** che la implementa creano una relazione **polimorfa**, cioè **tipo-sottotipo**

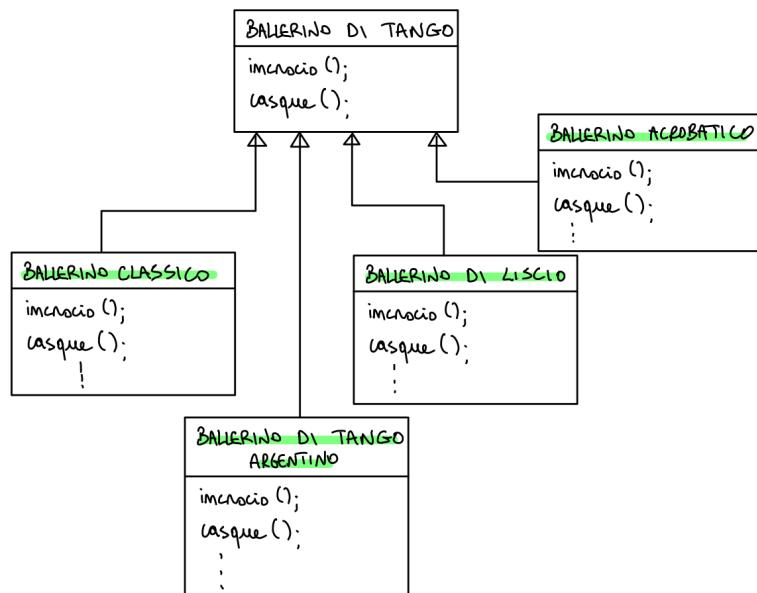
Quindi, se ho la classe Deck che implementa l'interfaccia CardSource, si il Deck deve implementare i metodi `draw` e `isEmpty`, ma c'è il grande vantaggio della **relazione polimorfa**

Infatti, in questo modo si può definire una reference di tipo CardSource e assegnargli un oggetto Deck

Il che vuol dire che ogni oggetto di tipo Deck è un valore del tipo CardSource

Quindi CardSource è un **tipo** e i suoi valori sono istanze di altre classi che hanno deciso di accoppiarsi a quell'interfaccia definita.

Esempio: ballerino di tango



Se istanzio un ballerino classico, allora ne ho uno classico e uno di tango, c'è una **relazione di tipo**

NON posso istanziare **direttamente** un **ballerino di tango**, ma potrei avere una classe di ballerino di tango argentino

Fare: `BallerinoDiTango x = new _____` è legit

C'è polimorfismo perché quando faccio `x.incrocio()` il **codice** che verrà eseguito viene scoperto a runtime

L'interfaccia ha anche un altro effetto

Se ho da qualche parte una procedura "P" che prima o poi avrà bisogno di un ballerino di tango, cioè `P(BallerinoDiTango bt)`

All'interno del metodo si potrà sicuramente fare `bt.incrocio()`, invece `bt.rock&rol()` no

Questo è possibile perché c'è una **relazione di tipo e di maschera**, cioè il metodo **NON** sa quale ballerino verrà passato, l'importante però è che implementi l'interfaccia

Si ottiene in questo modo la **segmentazione** dell'interfaccia e l'**estensibilità**

In pratica, ad un metodo si può passare un oggetto complesso a piacere (che implementa l'interfaccia), ma tanto vedrà solo una parte che è l'implementazione dell'interfaccia (**segmentazione**)

Esempio: il sort funziona se riceve oggetti di tipo Comparable, cioè che aderiscono al **contratto** che è l'interfaccia Comparable

L'interfaccia quindi è un modo per

- creare una relazione polimorfa
- consegnare a client diversi lo stesso oggetto partizionato
- realizzare operazioni complesse con contratti di delega

Interfaccia come mezzo per specificare un comportamento

Abbiamo un deck che è fatto in un determinato modo e usiamo il nostro shuffle

Con il `compareTo` definiamo il **natural order**, dicendo che gli elementi aderiscono al contratto Comparable

La domanda che ci poniamo ora è "posso avere un meccanismo per cui il **concetto** di **ordinamento** non è una proprietà dell'oggetto ma è una cosa che **inietto** nell'oggetto?"

Un sort non deve sapere la struttura degli oggetti, ma deve sapere solo qual è l'ordinamento da fare

Quindi il sort sfrutta solo il `compareTo` e ciò è proprio il **principio di segregazione dell'interfaccia**

Anche questo si fa con le interfacce, in particolare con una famiglia di interfacce, le **Functional Interfaces (lambda, stream)**

In Java un **esempio** è un'interfaccia che fa ciò che già fa Comparable, ma rende la **proprietà di comparazione un behaviour indipendente che può essere iniettato** e NON una proprietà dell'oggetto

Quest'interfaccia è Comparator

Il Comparator consente di confrontare due oggetti

Il sort quindi accetterà come parametro sia la collezione che un'istanza di una classe che implementa Comparator

Questo è il meccanismo di **iniezione di behaviour**

In pratica, oltre a passare la struttura dati da ordinare, passiamo anche la **strategia** per fare l'**ordinamento**

Le istanze di Comparator non sono altro che **funzioni**, infatti saranno delle classi **stateless** e implementeranno **solo** il metodo `compare`

Invece, le istanze di Comparable sono **oggetti** veri e propri, quindi avranno stato

Il Comparator è un **esempio di strategy pattern**

Function Object

Nel caso di Comparator possiamo parlare di function object, cioè **oggetti** che non hanno stato e sono di fatto delle funzioni

Quindi per fare un Comparator devo istanziare una classe stateless con un solo metodo e poi passare l'oggetto come parametro

Tutto ciò può essere evitato con:

- **nested class**, cioè definisco la classe di comparazione nello scope di un'altra
- **anonymous class**: se devo implementare una classe di cui mi serve una istanza che magari ha un solo metodo, allora Java mi dà la possibilità di **creare il comparator on the fly**
- **lambda expression**: il compilatore genera una classe al volo del tipo che il metodo si aspetta

Abbiamo visto quindi che le classi si possono utilizzare come **meccanismo per iniettare behaviour**

Esempi:

```

//NESTED CLASS
public class Card {
    static class CompareBySuitFirst implements Comparator<Card> {
        public int compare(Card pCard1, Card pCard2)
            { /* Comparison code */ }
    }
}

//ANONYMOUS CLASS
public class Deck {
    public void sort() {
        Collections.sort(aCards, new Comparator<Card>() {
            public int compare(Card pCard1, Card pCard2)
                {/* Comparison code */ }
        });
    }
}

//LAMBDA EXPRESSION
public class Deck {
    public void sort() {
        Collections.sort(aCards, (card1, card2) -> card1.getRank().compareTo(card2.getRank()));}

```

Static factory method

Static Factory: generatore di istanze di comparatore

In pratica, all'interno della classe Card implementiamo un metodo che restituisce un riferimento ad un Comparator di Card

La Card per l'esterno diventa un generatore di Comparator

In pratica, la classe oltre ad avere i get delle variabili d'istanza, ha anche dei get che permettono di generare i comparatori

```

public class Card {
    public static Comparator<Card> createByRankComparator() {
        return new Comparator<Card>() {
            public int compare(Card pCard1, Card pCard2)
                { /* Comparison code */ }
        };
    }
}

```

```
}
```

In questo modo **NON** c'è un **problema di visibilità**

Possiamo generalizzare queste cose nel concetto di **strategia**

L'idea dell'interfaccia Comparable non è altro che l'idea di un'operazione complessa di cui definisco il workflow delegando la strategia, i passi specifici, a determinati oggetti

Iterator

È un'interfaccia che ci permette di risolvere il problema della **navigazione su una collezione di oggetti**

Se passo il riferimento faccio **reference escaping**, quindi devo fare un **deep copy**

Oppure posso pensare di fornire all'esterno **NON** la **struttura**, ma **una copia navigabile della struttura dati**, rispetto alla quale **definisco anche il modello di navigazione**

Quando abbiamo una lista in generale di Card e la dobbiamo restituire all'esterno, restituendo un iterator otterremo una **struttura navigabile con due primitive**

- `hasNext()` : è booleano
- `next()` : ottiene il prossimo oggetto

In questo modo, non dovremo fare altro che chiedere l'iterator e impostare un ciclo per recuperare gli oggetti della collezione

L'iterator è proprio un pattern che ci permette di fornire un metodo per navigare una struttura dati rispettando un certo schema ed evitando di violare i principi di information hiding e encapsulation

L'interfaccia che permette di restituire un iterator è **Iterable**

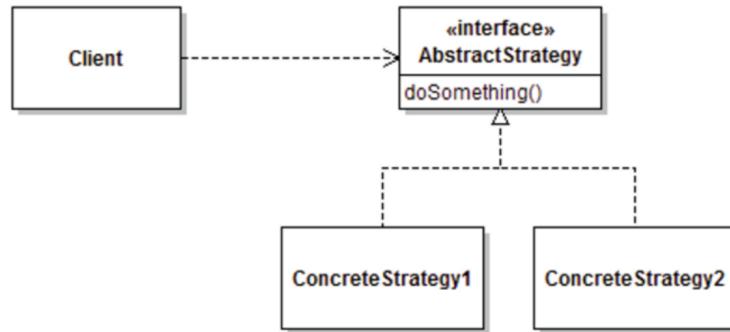
La collezione deve implementare Iterable

La singola classe, invece, Iterator

Strategy Pattern

Sia nel caso dell'iterator, sia nel caso del comparator, potrei avere la necessità di implementare strategie diverse di iterazione o comparazione. Ad esempio per gli iterator di un Binary Tree, potrei voler scorrere gli elementi in maniera pre-order, post-order

In questi casi entra in gioco il design pattern **Strategy**, il quale permette di definire una famiglia di algoritmi intercambiabili dai client che li usano



Struttura:

- <<interface>> **AbstractStrategy** che **dichiara** un metodo `doSomething()`
- Una o più classi **ConcreteStrategy** che **implementano** tale strategia implementando il metodo `doSomething()`

Un **esempio** di applicazione di questo design pattern è, come già visto, nell'utilizzo dei **Comparator**:

l'interface

`Comparator<T>` rappresenta l'interface di **AbstractStrategy**, mentre le varie classi che la implementano sono le **ConcreteStrategy**.

Interface Segregation Principle

Abbiamo analizzato i benefici della definizione di tante piccole interfacce specializzate per fornire ai client l'accesso soltanto ai behavior di cui hanno bisogno.

Questo permette anche la minimizzazione del coupling tra le classi, ma anche alcuni svantaggi, perché talvolta i **client** potrebbero essere **interessati a più behavior**.

In certi casi può tornare utile il concetto di **estensione** di un'**interfaccia**, ossia all'atto della definizione di una interface dichiarare la keyword `extends` che permette di estenderne un'altra.

In questo modo l'interface appena creata dichiarerà implicitamente anche i metodi dell'interface estesa e di conseguenza tutte le classi che la implementano dovranno transitivamente implementare i metodi di entrambe.

Così facendo, facciamo in modo che un oggetto si presenti in contesti diversi con risorse differenti

Object State

Sappiamo che ciò che succede a **run-time** fa parte della visione **dinamica**

È la struttura **statica** che regola la visione dinamica

Che cos'è lo **stato** di un **oggetto**?

Se ho una classe con un'unica variabile `int`, la sua cardinalità è 2^{32}

Se aggiungo una `String`, allora l'unico limite è la memoria fisica dell'elaboratore

Lo **stato** degli oggetti serve a fare molti ragionamenti, soprattutto è molto importante per il **testing**

Esempio: un mazzo di carte ha tanti stati quante sono le permutazioni delle carte

In pratica, quando facciamo testing, utilizziamo lo stato come **driver** per decidere quali test ha senso effettuare

Molto spesso ragioneremo sulle transizioni tra stati **astratti**, cioè si parte da uno stato astratto A e tramite un **evento** (un metodo), si passa ad un altro stato astratto B

Ragionare su **stati astratti** ci permette di passare da **cardinalità enormi** ad altre **gestibili**

I metodi servono per gestire le transizioni tra stati astratti

In generale possiamo definire lo **stato** come uno spazio **multidimensionale** (una dimensione per ogni variabile), dove ogni punto rappresenta l'insieme dei valori assunti da tutte le sue variabili di stato.

Distinguiamo però due tipi di stato:

- **Stato concreto:** lo **stato** è l'insieme dei valori che assumono le sue variabili d'istanza in un determinato istante
- **Stato astratto:** sottoinsieme significativo dello spazio degli stati concreti

In particolare gli stati astratti sono utili per catturare quelle partizioni dello spazio degli stati maggiormente significative.

Ad

esempio per un Deck di Card, lo stato astratto *Empty* è significativo perché comporta l'eliminazione di uno scenario d'uso per l'oggetto (impossibilità di chiamare il metodo *draw*).

Esistono inoltre casi speciali di oggetti che non hanno alcuno stato, come le classi che implementano una singola funzione (*Comparator*), in tal caso parliamo di **stateless object**,

mentre gli oggetti che lo posseggono prendono il nome di **stateful object**.

Nel caso di oggetti immutabili tale distinzione non ha senso, in quanto essi possono avere un singolo stato.

Un altro **esempio** è uno Stack di oggetti: esso ha virtualmente infiniti stati, ma posso caratterizzarli considerando solo quelli fondamentali:

- **Stack vuoto:** posso fare solo operazioni *push*
- **Stack non vuoto:** posso fare operazioni di *push* e *pop*

Inoltre, posso distinguere uno stato in cui lo stack contiene un solo elemento: in questo stato, se eseguo *pop* si torna nello stato *Empty*

Ho definito quindi degli stati astratti, delle vere e proprie **classi di equivalenza** tra i vari stati.

Ad

esempio due Stack non vuoti, anche se con oggetti diversi, sono considerati in due stati equivalenti perché sono entrambi non vuoti.

In generale, possiamo dire che aumentare il numero di stati spesso porta a problemi di **inconsistenza**, quindi è proprio un **antipattern**

Designing Object Life Cycle

Gli State Diagram sono molto utili per rappresentare le transizioni di stato di un oggetto di una classe.

Tali transizioni sono costituite dai metodi della classe. L'assenza di una transizione implica che essa non è possibile (invalida) per un certo stato.

Il modello a stati può essere considerato anche come un **ciclo di vita** di un oggetto, perché descrive la vita a partire dall'inizializzazione fino all'eventuale distruzione da parte del garbage collector.

Nullability

Molto spesso ci sono problemi quando si usa **null**

Con **null** si possono esprimere vari concetti non sempre chiari

Può voler indicare:

- una variabile è inizializzata in modo non corretto
- un'assenza di valore nel ciclo di vita dell'oggetto
- non è stata trovata una specifica istanza in una collezione
- un valore da interpretare in un modo speciale

Il null fu inventato da Tony Hoare

Una best practice sarebbe progettare classi che non usino le null reference, e quindi nel progettare classi, evitare, se possibile, stati astratti di assenza di valore.

Per fare questo si possono utilizzare come sempre gli approcci di input validation o design by contract.

Java fornisce l'implementazione di un pattern, l'**optional** che permette di risolvere questo problema

Optional <T>

Un modo per rendere esplicito il problema delle null reference è l'utilizzo del tipo `Optional<T>`, un tipo *Generic* che funge da **wrapper** per istanze del tipo T che possono essere prive di valore

È una classe **parametrica**

Optional mette a disposizione metodi come `empty()`, `get()`, `isPresent()`

Per **rappresentare l'assenza** di valore della variabile possiamo usare il valore di ritorno di `Optional.empty()`

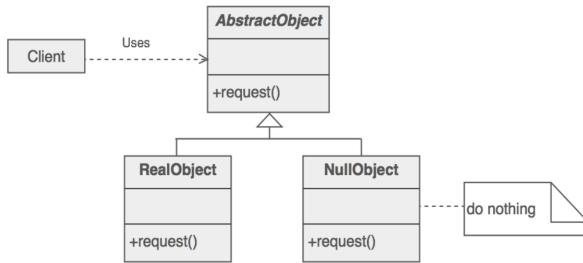
Per **assegnare un valore reale** ad un'istanza Optional usiamo invece `Optional.of(value)`

Se non ci si aspetta che value sia null, `Optional.ofNullable(value)`

Esempio: `private Optional<Book> aBook;`

Con gli Optional si può realizzare il **null object design pattern**, anche se molto spesso lo mappiamo con un **factory method**

Esempio:



▼ Lezione 16 10/05/2024 (Carmine)

L'obiettivo del project work è realizzare un server che espone un'interfaccia **REST** con un payload json

Ora siamo in grado di realizzare un servizio che ascolta su una o più URI e accetta metodi GET, PUT, POST, ecc e che si aspetta e distribuisce payload JSON

Se realizziamo tutto in un unico servizio, allora l'architettura è monolitica

Abbiamo capito che l'aggregazione può avvenire a vari livelli

Non abbiamo visto interfaccia grafica infatti abbiamo testato con client che effettuano il testing di applicativi come POSTMAN e INSOMNIA

Abbiamo visto la differenza tra architettura monolitica e a microservizi

Ciò che manca è la **persistenza**, cioè il fatto che questi microservizi debbano gestire dei dati

Per essere **persistenti** è necessario inserire un **livello di persistenza**

Lo inseriamo tramite un **database relazionale** e/o **NON relazionale**, cioè MySQL e MongoDB

Perché li vediamo entrambi?

Conosciamo i **database relationali**, hanno una **potenza unica** permettono di fare query complesse, ma il loro limite si ha rispetto al mondo delle moderne applicazioni le quali spesso stanno dietro alle **API** e che lavorano bene con **dati strutturati**

Se si vuole progettare un layer di persistenza con un database relazionale, la prima cosa da fare è pensare allo schema, alla struttura dati, che hanno una serie di connessioni (chiavi esterne riferimenti) e se il dato è ben strutturato allora funziona bene

Però, che succede se **poco strutturato**? Nei **database relationali** siamo costretti a trovare dei turn around

Pensiamo ad una **struttura dati** che ha un **campo** che **può assumere tanti valori diversi** che deve essere accompagnato da una label diversa, pensiamo ad una rubrica telefonica

Se mappiamo questo su un **database relazionale** abbiamo varie opzioni:

- **campi dummy**: colonne in più
- **ricorrere ad una chiave esterna**

Quindi, per usare dati non strutturati siamo costretti ad usare una sovrastruttura per poterla gestire

Invece, nei **database NON relazionali** il **concetto di schema** **NON** esiste, cioè non è definito a priori il concetto di struttura con la quale il dato si presenterà

Infatti anche nella nomenclatura parleremo di **documento** e non di tabella

L'assenza di un campo non è l'eccezione perché ogni pezzo di documento e informazione è composto da informazioni che possono esserci o meno

Questo rende i database non relazionali particolarmente adatti ad un contesto dove la comunicazione è per documento e non per dato strutturato

Dato questo legame concettuale, anche lo sviluppo tecnologico si è molto legato alla parentela

Vedremo anche la messa in sicurezza, cioè l'**autenticazione** per dimostrare di essere chi diciamo di essere e l'**autorizzazione** in modo che una volta che il sistema ha riconosciuto l'utente allora gli consentirà di effettuare determinate cose in base al ruolo

In sintesi:

- **autenticazione**: posso accedere al sistema
- **autorizzazione**: gestire i permessi di effettuare determinate operazioni su delle risorse del sistema in base a chi sono

Introduciamo anche il processo di containerizzazione

DATA PERSISTENCE

CRUD ⇒ Create Read Update Delete

ce ne sono molte di più ma queste sono le fondamentali

le API di un servizio o microservizio sono basate su queste 4 operazioni

NoSQL databases (Not only SQL)

Sono database non tabellari e memorizzano dati in maniera diversa rispetto ai relazionali

Con i NoSQL è possibile fare tutte le operazioni SQL e non solo, infatti la possibilità di inserire campi dinamici e query complesse restano ma vengono aggiunte delle funzionalità

I dati possono essere **strutturati**, **NON strutturati** o, addirittura, **polimorfici**

I tipi fondamentali di database NoSQL sono basati su:

- **documenti**
- **grafi**: elemento fondamentale è il nodo e le relazioni che legano quei nodi
- **chiave-valore**: database che gestisce un hashmap, super usato per fare ricerca su chiavi
- **column oriented**: wide column, cioè il dato sta sulle colonne

Un database NoSQL rilascia le ACID per far sì che siano vere le proprietà **BASE**:

- **BA = Basically Available**: lettura e scrittura sono operazioni sempre disponibili, anche a costo di sacrificare la consistenza, cioè non è detto che le letture ritornino sempre il dato aggiornato più recentemente
- **S = Soft state**: c'è una certa probabilità che nel tempo il database sia consistente, ma non si ha la certezza del quando e come
- **E = Eventual consistency**: è una proprietà tipica di questo tipo di database, in pratica se il sistema è funzionante, i dati eventualmente saranno in uno stato consistente

Un'altra proprietà importante è lo **scaling**

Con i database tradizionali non si possono replicare i dati, mentre qui si possono sparpagliare

MongoDB

È un database NoSQL orientato ai documenti, schema-less

Schema-less: né tipo né dati definiti a priori

Se non mettiamo nulla come id, allora mongo autogenera `_id` che sarà univoco per ogni documento

Per memorizzare i dati utilizza la rappresentazione BSON, cioè binary representation of json

Gli oggetti sono organizzati in una **scala gerarchica**:

1. un'istanza di MongoDB può avere zero o più **databases**
2. un database può avere zero o più **collections**
3. una collection può avere zero o più **documents**
4. un document può avere uno o più **campi**

Dato che **NON** c'è uno **schema predefinito** dei dati, ogni **documento** all'interno della **stessa collezione** può avere anche **campi differenti**

Tutto ciò che viene memorizzato è, quindi, a cura del programmatore, il database non gestisce nulla se non pochi vincoli sui dati

RDBMS		MongoDB
Database	→	Database
Table, View	→	Collection
Row	→	Document (BSON)
Column	→	Field
Index	→	Index
Join	→	Embedded Document
Foreign Key	→	Reference
Partition	→	Shard

CRUD OPERATION

- **Create**
 - db.collection.insertOne(<document>)
 - db.collection.insertMany(<document>)
 - db.collection.save(<document>)
 - db.collection.update(<query>, <update>, { upsert: true })
- **Read**
 - db.collection.find(<query>, <projection>)
 - db.collection.findOne(<query>, <projection>)
- **Update**
 - db.collection.update(<query>, <update>, <options>)
- **Delete**
 - db.collection.remove(<query>, <justOne>)

▼ Lezione 17 - 13/05/2024

Continuiamo sugli approcci di software design sempre basandoci sull'**esempio** del gioco delle carte

Oggi vediamo i **Pattern Strutturali**

Null Object Design Pattern

Durante l'evoluzione dello stato dell'oggetto si possono verificare (anche lecitamente) dei momenti in cui gli oggetti non sono definiti

Quindi abbiamo analizzato il concetto della **nullability** degli oggetti, di cui lo stesso Tony Hoare col senno di poi ha definito l'invenzione della nullability come il billion dollar mistake

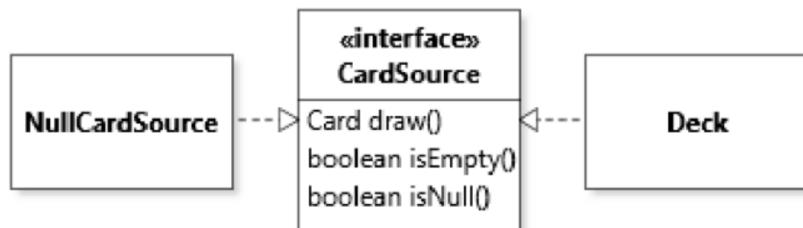
Bisogna evitare la nullability e gestire questa situazione, cioè eliminare l'idea che uno stato non sia definito

Uno dei vantaggi dell'utilizzo di un database NoSQL è l'assenza di uno schema predefinito che consente di avere oggetti con forma diversa uno dall'altro ([esempio](#) del libro Jolly nelle carte)

Esiste una soluzione per evitare l'utilizzo delle null reference per la rappresentazione di valori assenti che evita l'utilizzo di wrapper e rappresenta un vero e proprio **design pattern** chiamato **Null Object**.

L'idea di base è creare un oggetto speciale che rappresenti un **valore assente e testare l'assenza** usando una chiamata ad un metodo polimorfico.

Esempio:



In questo design si aggiunge un metodo `isNull()` ad un'interfaccia per **determinare la presenza** del valore **null** e si aggiunge una classe per rappresentare tale valore (`NullCardSource`).

Il metodo

`isNull()` restituisce `false` per qualsiasi oggetto di tipo **CardSource** ad eccezione di **NullCardSource**, `true` per un'istanza di **NullCardSource**.

Per l'implementazione di questo pattern si può usare una classe anonima come costante dell'interface che implementa il metodo `isNull()` restituendo true e poi usare un metodo default per restituire false

Esempio:

```
public interface CardSource {

    public interface CardSource {
        public static CardSource NULL = new CardSource {
            public boolean isEmpty() { return true; }
            public Card draw() { assert !isEmpty(); return null; }
            public boolean isNull() { return true; }
        }
    }
}
```

```
Card draw();
boolean isEmpty();
default boolean isNull() { return false; }
}
```

Identity, Equality and Uniqueness

NON bisogna **confondere** il concetto di **identità** con quello di **uguaglianza**

L'**identità** di un oggetto si **riferisce** quindi alla sua **locazione di memoria**, o al suo riferimento. Negli IDE, come Eclipse, questa identità è rappresentata da un object id.

In Java l'operatore

`==` **restituisce true** se due operandi hanno lo stesso valore; nel caso di tipi reference per "stesso valore" si intende stesso object id; pertanto **due variabili che referenziano oggetti uguali nello stato, ma distinti nell'identità, restituiranno false** in un predicato di uguaglianza con tale operatore.

L'**uguaglianza** è un concetto più complesso perché dipende dal design della classe. In generale **potremmo dire che due oggetti sono uguali se tutti i campi assumono gli stessi valori**; talvolta, in classi molto complesse, potremmo voler considerare uguali anche oggetti che presentano soltanto alcuni campi uguali, o ancora due Set possono essere considerati uguali se contengono gli stessi elementi, anche se memorizzati in ordine diverso.

Java fornisce un meccanismo per specificare l'uguaglianza mediante l'overriding del metodo `equals` dalla classe *Object*.

L'

uguaglianza deve essere:

- **simmetrica**
- **transitiva**
- **riflessiva**
- **consistente**

Esempio:

```
RealCustomer x = new RealCustomer("Gerardo");
RealCustomer y = new RealCustomer("Gerardo");

System.out.print(x == y) //si ottiene FALSE
```

Ottengo false, perché l'uguaglianza è sui riferimenti e non sullo stato degli oggetti

In questo caso, il **controllo di uguaglianza si traduce in un controllo di identità** sugli oggetti

Solitamente il concetto di uguaglianza coincide con il concetto di identità, ma come facciamo a distinguere? Tutti gli oggetti hanno predefinito il metodo equals, quello che dobbiamo fare è ridefinire il concetto di uguaglianza

Dobbiamo fare attenzione perché **ogni classe che sovrascrive il metodo `equals` deve inoltre sovrscrivere il metodo `hashCode()`** in modo da mantenere il vincolo secondo cui due oggetti sono considerati uguali se l'invocazione del metodo hashCode su entrambi produce lo stesso risultato intero.

Infatti se due oggetti sono uguali devono avere lo stesso hash code

Inoltre, se l'oggetto è anche comparable ci deve essere **consistenza** tra la definizione della **relazione d'ordine** e la definizione della relazione di **uguaglianza**

Questi concetti ci servono per gestire la **uniqueness** degli oggetti

Ci sono oggetti che godono della proprietà di dover vivere in **istanza unica** o **in istanze che hanno valori diversi**

I due esempi che danno origine a due pattern diversi

- **flyweight:** garantisce che ogni istanza sia unica
- **singleton:** garantisce che ogni classe ha un'unica istanza

Design Pattern: Flyweight

Si tratta di un pattern strutturale che fornisce un modo per **gestire collezioni di oggetti immutabili** e consente di assicurare l'unicità degli oggetti di una classe.

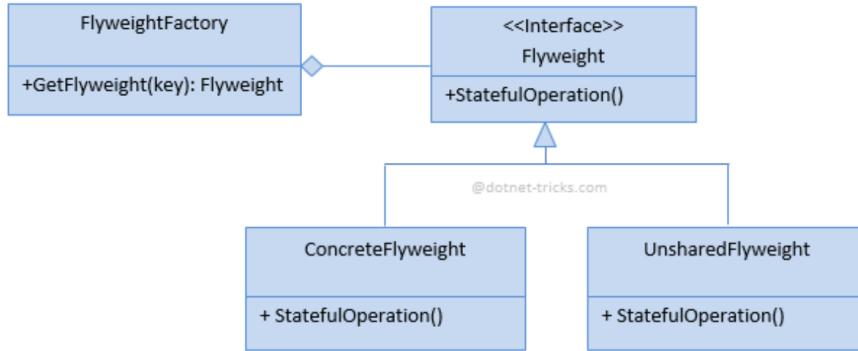
Il contesto di utilizzo è la presenza di classi fortemente condivise in un sistema software

L'idea di base è di **controllare la creazione di oggetti** di una certa classe chiamata *flyweight class* attraverso un **metodo di accesso che assicuri l'impossibilità di creare oggetti duplicati** (distinti ma uguali).

Per realizzare questo pattern c'è bisogno di **3 principali componenti**:

- un **costruttore privato** per la flyweight class, per impedire che i client possano creare oggetti
- uno static **flyweight store** che mantiene una collezione di istanze della classe flyweight

- un **metodo static di accesso** che restituisce un oggetto flyweight unico; il metodo tipicamente **effettua un check per capire se l'oggetto richiesto esiste già nello store o, in alternativa, crea e restituisce l'oggetto**



Questo pattern viene utilizzato in particolare per ottimizzare l'utilizzo delle risorse, infatti il suo utilizzo può migliorare le prestazioni di una applicazione

N.B: gli oggetti flyweight dovrebbero essere **immutabili** per assicurare la loro unicità.

Supponiamo di gestire una **collezione** di **oggetti immutabili** (le carte)

Creiamo un valore unico per ogni carta e quando chiedo una carta mi viene restituita sempre quella

Costruisco un costruttore privato che genera tutti i valori e li memorizza, da questo momento in poi uso questa classe come se fosse un **factory**

In pratica, per ottenere un valore lo chiedo a questa classe

In base ad una chiave e ottengo il riferimento all'oggetto che già esiste

Questo lo posso fare nel caso in cui i valori sono finiti, come per le carte perché sono enumerabili

Esempio: quando istanzio questa classe creo 52 carte e quando dico mi dai il 4 di fiori, mi da un riferimento al 4 fiori

Però, alcune volte ci si può trovare in situazioni in cui non si sa **quanti** valori né **quali** verranno ad essere generati

Posso utilizzare lo stesso principio, ma **NON creo tutti i valori né li memorizzo**

Ogni qualvolta che uso il metodo di accesso vado a vedere nella struttura se l'oggetto esiste e restituisco il riferimento oppure lo creo e restituisco il riferimento

Design Pattern: Singleton

Se il flyweight si presenta in molte varianti a seconda se i valori siano numerabili o meno, il caso del **singleton** è più semplice

È una classe che ha una singola istanza

Bisogna rendere il **costruttore** un metodo **privato** e implementare all'interno della classe che si vuole comportare da singleton un metodo `getInstance`

La **prima** volta che viene chiamato **crea e restituisce il riferimento**, mentre le **volte successive restituisce solo il riferimento**

Deve esserci una variabile statica che mantiene un riferimento ad un'istanza di se stesso

Ogni volta che viene invocato il `getInstance`, questo controlla se `instance == null`:

- se è **true**, allora crea l'istanza e la restituisce
- se è **false**, restituisce il riferimento

A differenza del pattern *Flyweight* gli oggetti *Singleton* sono tipicamente *stateful* e mutabili, mentre gli oggetti *Flyweight* dovrebbero essere immutabili

Nested class

Le classi innestate possono essere divise in due grandi categorie:

- **Inner class**: classi dichiarate all'interno di altre classi che di solito forniscono behavior addizionali che coinvolgono un'istanza della classe esterna ma che per qualche motivo non vogliamo integrare in essa
- **Static nested class**: si distinguono dalle inner class perché non sono collegate ad istanze della classe in cui sono innestate. Sono usate principalmente per encapsulamento e organizzazione del codice.

Tra le *inner class* distinguiamo altre due categorie speciali:

- **Anonymous class**: spesso utilizzate per implementare function object (classi che implementano semplicemente una singola funzione)
- **Local class**: simili, ma più raramente utilizzate

Composition

Come è possibile gestire in maniera uniforme varianti di una stessa struttura complessa?

Il concetto alla base è **divide et impera**

Dividere significa **delegare**, cioè chiedere a qualcuno di fare un pezzo di lavoro per voi

In OO significa che chiedo ad un altro oggetto di cui ho un riferimento di svolgere parte delle mie responsabilità

In pratica vediamo come gestire oggetti che sono una collezione di altri oggetti, i quali portano intrinsecamente con sé altre astrazioni

Esempio: se sto modellando il libretto universitario, oltre ad avere dati relativi alla scheda anagrafica, questo avrà *in pancia* qualcosa che è un oggetto che è naturalmente parte di un libretto universitario ma è concettualmente diverso, ovvero l'esame

La rappresentazione è dettata dal dominio, l'oggetto è per sua natura composto

Il concetto di **delega** nasce perché vogliamo avere classi semplici, cioè vogliamo evitare l'antipattern del **god class**

Delega: se assegnassi tutte le responsabilità ad una classe, questa risulterebbe troppo grande e quindi decido di **delocalizzare** alcune responsabilità in un'altra classe, ma devo fare in modo che la prima classe abbia un **riferimento** alla seconda

Mentre se ciò accade per **natura del dominio** allora si parla di **aggregazione**

UML per rappresentare queste situazioni utilizza un diamante:

- **pieno** per l'**aggregazione**
- **vuoto** per la **composizione**

In realtà, quando si passa all'implementazione del codice tutto ciò si **appiattisce**

Il pattern **composite** gestisce l'**aggregazione**

Il pattern **decorator** ci fa modellare la **delega** in maniera **dinamica**

Design pattern: Composite

Supponiamo di gestire un esempio di Gioco di Carte

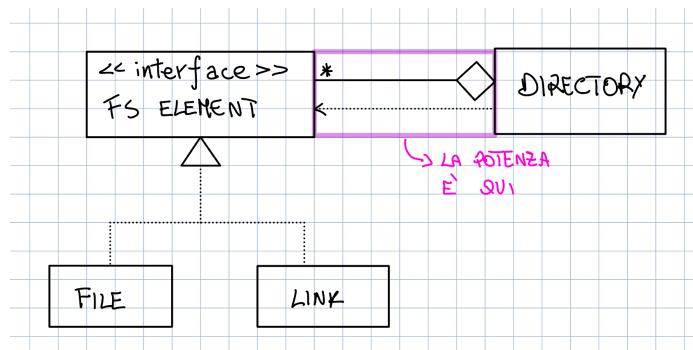
In generale, i diversi GameModel possono avere esigenze diverse nel gestire i mazzi

Stiamo cercando di rappresentare una situazione in cui si vuole operare sugli **oggetti una volta considerandoli come aggregati e un'altra come oggetti differenti**

Sappiamo che un qualunque mazzo di carte dovrà implementare l'interface CardSource
 In questo modo però dobbiamo definire **staticamente** l'insieme delle possibili implementazioni
 Inoltre, il **numero** delle possibili **classi** interessate all'implementazione dell'interfaccia potrebbe essere molto **elevato**
 In questo modo è anche **complicato cambiare scelte a runtime**

Ci viene in aiuto il pattern

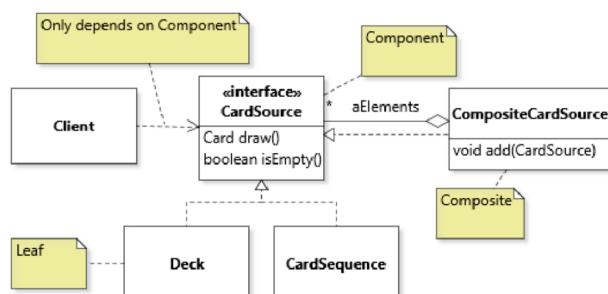
Esempio: file system



Se nell'interfaccia mettiamo un **print** allora:

- **File** e **Link** faranno il loro
- **Directory** invece farà un iterazione (pattern iterator) per stampare tutti gli elementi all'interno, i quali possono essere file, link o altri aggregati, quindi farà delle **deleghe**

Altro esempio:



3 ruoli principali:

- **component**: interfaccia o classe astratta che dichiara i metodi resi disponibili al client
- **composite**: classe che aggredisi compone di component
- **leaf**: l'oggetto semplice che implementa l'interfaccia component

Dove sta la **potenza di questo pattern?**

Si trova nella **relazione IS-A** e in pancia ci saranno * elementi di aElements

In questo modo, posso trattare un singolo elemento in modi diversi a seconda di ciò che mi serve

A runtime, il client vede un CompositeCardSource che in pancia avrà elementi semplici e altri che saranno a loro volta composti

La classe Directory è un file system element e nello stesso tempo ha in pancia una collezione di file system element

Quando si applica questo pattern un problema importante di implementazione è il considerare come aggiungere al composite delle istanze del component che lo compone, ossia un modo per specificare quali sottotipi dell'interface compongono il composite.

Due modi:

- fornire **un metodo per aggiungere elementi del composite**, permettendo in questo modo la modifica a run-time del composite
- **inizializzare gli oggetti composite attraverso i loro costruttori**, usando ad esempio costruttori copia per evitare il leaking delle referenze ad una struttura dati privata

Decorator Pattern

Cambiando solo la cardinalità della relazione IS-A risolviamo un altro problema, cioè aggiungere comportamenti ad una classe esistente in modo dinamico

Il modello principale utilizzato nel paradigma OO per estendere la funzionalità dell'oggetto è l'**ereditarietà**

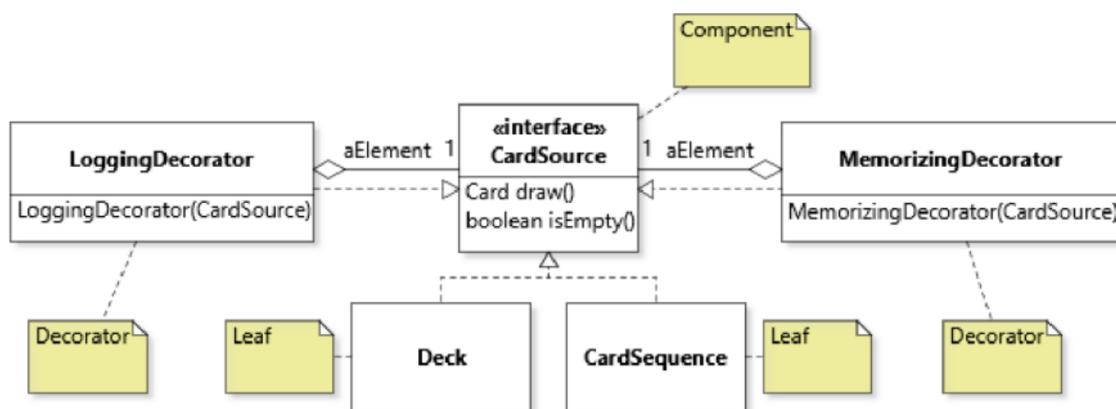
Non sempre però è l'approccio giusto, infatti è stato dimostrato che estendere gli oggetti con l'ereditarietà spesso si traduce in una gerarchia di classi che finisce per esplodere

Il **Decorator** fornisce un'alternativa flessibile all'ereditarietà, poiché consente di **arricchire dinamicamente a run-time** un oggetto con nuove funzionalità

È possibile anche impilare uno o più decorator uno sopra l'altro, dove ognuno aggiunge nuove funzionalità

Principali differenze tra *Decorator* e ereditarietà:

- Un **Decorator agisce a runtime** a differenza dell'ereditarietà che estende con le sottoclassi il comportamento della classe padre in fase di compilazione
- Un **Decorator può operare su qualsiasi implementazione di una determinata interfaccia**, eliminando la necessità di creare sottoclassi di un'intera gerarchia di classi
- La **sottoclasse** aggiunge comportamento a tempo di compilazione e la **modifica interessa tutte le istanze della classe originale**; il decorator pattern può fornire nuovi comportamenti in fase di esecuzione per i singoli oggetti
- L'uso del **Decorator porta a un codice più pulito** e testabile, mentre i servizi creati con l'ereditarietà non possono essere testati separatamente dalla sua classe padre perché non esiste un meccanismo per sostituire una classe padre con uno stub



La struttura del pattern Decorator, molto simile a quella del pattern Composite, consta di 4 elementi principali:

- **Component** che rappresenta l'interfaccia dell'oggetto che deve essere decorato dinamicamente
- **ConcreteComponent**: la classe di oggetti *Component* a cui si vuole aggiungere nuove funzionalità
- **Decorator**: interfaccia tra *Component* e *ConcreteDecorator* che possiede un riferimento a un *Component* (aggredisce un oggetto *Component*) e ne estende l'interfaccia
- **ConcreteDecorator**: classe che implementa il *Decorator* aggiungendo nuove funzionalità rispetto al *ConcreteComponent*

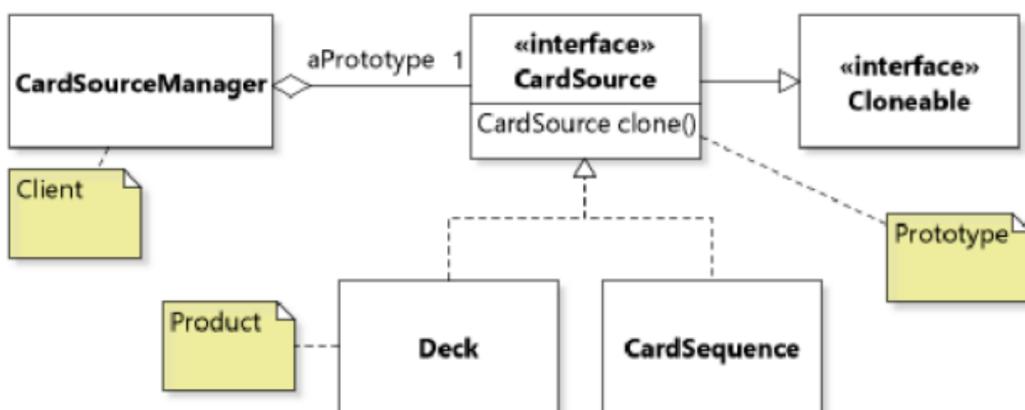
Design Pattern: Prototype (bonus)

Immaginiamo un contesto in cui abbiamo una collezione di oggetti referenziati da un tipo interface: in questo caso, se volessimo effettuare una copia, dovremmo conoscere il tipo concreto degli oggetti che compongono la collezione per poterne fare delle copie, invocandone i rispettivi costruttori

Per evitare di ricadere nell'antipattern *switch statement* nell'implementazione della copia rompendo così i benefici del polimorfismo, vorremmo un meccanismo che ci permetta di copiare tali oggetti polimorfici, ossia poter creare copie di oggetti senza conoscere il loro tipo concreto

In Java tale funzionalità è supportata mediante un meccanismo chiamato **cloning**

Il contesto di utilizzo del pattern *creazionale Prototype* è proprio la creazione di oggetti, i cui tipi non sono conosciuti a tempo di compilazione



La struttura è composta da 3 partecipanti:

- **Prototype**: classe astratta/interfaccia per gli oggetti da clonare che estende *Cloneable* e dichiara il metodo *clone*
- **ConcretePrototype** (o *Product*): classe che implementa l'operazione di clonazione di sé stessa
- **Client**: invoca la clonazione dell'oggetto usando un riferimento all'interface *Prototype*

Questo pattern permette quindi di utilizzare il polimorfismo per creare istanze di classi senza alcuno statement di controllo

Esempio di implementazione: classe *HashTable* che implementa *Cloneable* e viene implementata da classi concrete come *HashMap*, *LinkedHashMap*

L'implementazione di base del metodo *clone* permette una copia superficiale dell'istanza, ma è possibile ridefinire il comportamento prevedendo una **copia deep**, andando a copiare anche ogni elemento contenuto nella struttura dati.

▼ Lezione 18 - 17/05/2024

Recap lezione scorsa

Abbiamo visto soluzioni pattern che abbiamo chiamato **strutturali**

Abbiamo imparato a gestire strutture gerarchiche complesse in maniera uniforme con il composite e abbiamo imparato a gestire la possibilità di aggiungere caratteristiche dinamiche con il decorator

Tutti e due i pattern si assomigliano dal punto di vista strutturale la differenza sta nella molteplicità

Il problema che vogliamo risolvere è trasformare un'azione, che poi in un approccio OO il più delle volte vuol dire l'invocazione di un metodo, in una sorta di first class object

Lo affronteremo nella penultima lezione quando parleremo di progettazione orientata agli stream

La nostra metafora di progettazione e implementazione vede gli oggetti che nascono da una classe di appartenenza (blue print) che detta com'è lo stato di un oggetto (variabili di istanza) e quali sono le operazioni/azioni che dall'esterno possono essere fatte sullo stato di questo oggetto (i metodi)

La nostra idea è quella di far sì che queste **azioni** possano a loro volta diventare un **oggetto manipolabile**

Quindi non solo vogliamo usare il metodo per cambiare lo stato dello studente, ma che poi il **comando/metodo possa a sua volta essere oggetto di azioni**

Invece, ad oggi sui metodi abbiamo una sola azione implicita, cioè l'attivazione

Perché potremmo essere interessati ad avere la possibilità di trattare i metodi come se fossero degli oggetti?

Ad esempio l'**undo** può essere utile in molti contesti

Se voglio correttamente gestire l'undo una cosa che potrei fare è quella di dire che l'azione di **eseguire** un metodo e quella di **disfare l'esecuzione** del metodo sono in realtà due **metodi** di un **oggetto che vede l'azione come stato**

Per fare un'azione su una classe C costruisco un nuovo oggetto che ha in pancia m, ma che la vede come un pezzo di un suo stato

Design Pattern: Command

È l'ultimo pattern strutturale

Concettualmente un **command** è un pezzo di codice che esegue un certo task, ad esempio l'esecuzione di un metodo o di una funzione

Esempio: vogliamo che il draw venga messo come oggetto in uno stack di azioni

In generale, vogliamo far diventare un comando una **manageable units of functionality**

Per implementare questo pattern realizzeremo un'interfaccia **Command** che avrà i metodi **execute** e **undo**

Esempio: abbiamo un contocorrente `cc`

Facciamo `cc.versa(300)`

Vorremmo poter dire "**costrisci un comando .versa e poi più tardi diseseguilo e metti ogni operazione in uno stack di comandi**"

Trattiamo un metodo come un first class object

Definiamo un interfaccia command che ha come metodi (qualunque)

- **execute**
- **undo**

Voglio che alcune operazioni diventino Command

Definiamo la **radice** del Command, un'**interfaccia che ammette il fare e disfare** (execute e undo)

Come implementiamo il comando? Due modi

- creiamo una classe del tipo DepositCommand che molto spesso viene definita implicitamente direttamente all'interno della classe ContoCorrente
Questa classe implementerà execute e undo (e eventualmente anche altri metodi)
Che cosa fa il costruttore di questa classe? Vuole in ingresso il ContoCorrente e il parametro del comando, cioè la somma da depositare (amount)
A questo punto, DepositCommand ha uno stato che è fatto dal target e dai parametri per eseguire

- introduciamo un **AbstractCommand** che ci permette di **fattorizzare lo stato**

In questo modo, le classi che implementeremo saranno una per il comando deposit e una per il withdraw

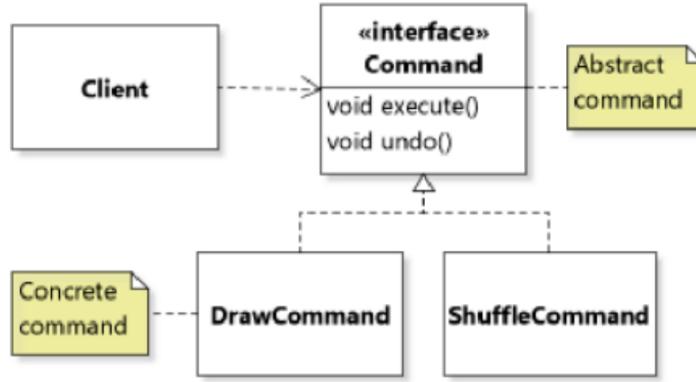
Così facendo, possiamo aggiungere della logica di gestione nella classe astratta, ad esempio aggiungendo il push del comando stesso su uno stack per creare dei log

Le due classi concrete dovranno solo implementare dei metodi

In realtà il pattern Command si realizza con un **invoker**

L'**obiettivo** del pattern è quello di **disaccoppiare** l'invocazione di un comando dai dettagli implementativi, quindi si vuole separare chi invoca il comando da chi esegue l'operazione

Di solito, ciò si realizza con la catena: **Client → Invoker → Receiver**



Il **client** chiama il metodo dell'invocatore senza conoscere i dettagli del comando ed imposta il **receiver**

L'**invoker** ha l'obiettivo di incapsulare e nascondere i dettagli della chiamata ed effettua l'invocazione del comando

Il **Command** fa da interfaccia generica per l'implementazione del comando

Il **ConcreteCommand** è l'implementazione specifica che collega l'invoker con il receiver

Il **receiver** è colui che riceve il comando e sa come eseguirlo

Legge di Demeter

Nel progettare pezzi di software usando l'aggregazione si rischia di finire in una lunga catena di delegazioni tra oggetti sfociando nell'antipattern che prende il nome di **message chain**

Per evitare questo antipattern il principio di Demeter offre delle linee guida affermando che un metodo deve avere accesso soltanto a:

- Variabili di istanza del suo parametro隐式
- Argomenti passati al metodo
- Qualsiasi nuovo oggetto creato all'interno del metodo
- Eventuali oggetti disponibili globalmente (se necessario)

In sintesi, ogni oggetto dovrebbe conoscere il minimo indispensabile di un insieme di oggetti che sia più piccolo possibile

Ad **esempio** in una catena di invocazioni, le classi che occupano posizioni intermedie **NON** devono restituire referenze a strutture interne ma soltanto chiamare altri servizi per completare la richiesta del client

Testing

Che cosa significa fare testing?

Durante lo sviluppo di un prodotto, è fondamentale dimostrare a noi stessi e ai nostri clienti che il sistema funziona come previsto.

Tuttavia, il **testing NON serve a dimostrare la correttezza assoluta** di un programma, poiché è impossibile dimostrare teoricamente che un programma sia completamente privo di errori.

L'approccio che usiamo è quello ingegneristico, si basa sulla ripetizione dei test: se un oggetto viene testato più volte e fornisce risultati positivi, possiamo presumere che funzioni correttamente. **Provandolo in un numero sufficiente di situazioni significative, possiamo avere una ragionevole certezza del suo comportamento.**

Esempio: quando si collauda un ponte, si effettuano diverse prove e, se queste vanno a buon fine, il ponte può essere aperto al traffico con la sicurezza che non crollerà.

L'ingegnere capo, prima di testare il ponte, deve decidere i casi di test.

Per esempio, potrebbe chiedere a una persona di peso noto di attraversare il ponte più volte. Se tutte le prove sono superate con successo, possiamo estendere i risultati ottenuti e avere la certezza che il ponte funzionerà correttamente in situazioni simili.

Fare testing significa:

- **Decidere gli ingressi**
- **Avere un risultato atteso per ogni ingresso (l'oracolo)**

Il processo **NON è deterministico** e non prova la correttezza assoluta del programma. Se in un caso di test l'output non rispetta l'oracolo, possiamo individuare un errore e capire cosa aggiustare. Tuttavia, se l'output coincide con l'oracolo, non possiamo affermare con certezza che il programma sia privo di errori.

Obiettivo del testing

Il testing rileva errori o non fa nulla. Se eseguiamo molti test ben progettati e nessuno rileva errori, **possiamo affermare che il programma si comporta come previsto.**

Progettazione dei casi di test

L'aspetto **cruciale** del testing è la **definizione dei casi di prova**.

L'esecuzione dei test è un'attività secondaria: i programmatore spesso eseguono i test durante la notte tramite un processo automatizzato, che crea un ambiente virtuale, esegue i test e fornisce i report al mattino.

Un ingegnere del software, quando si parla di testing, progetta i casi di prova con una strategia e un criterio ben definiti. Gli input devono essere rappresentativi delle situazioni reali in cui il programma sarà utilizzato, per garantire che il programma funzioni correttamente in tali contesti.

Key terminology

Failure: un comportamento **NON** accettabile di qualsiasi tipo.

Una failure si verifica quando un sistema **NON rispetta i contratti previsti.**

Questo può derivare da problemi infrastrutturali o da bug nel codice.

Defect (o Bug): una **decisione inappropriata** dello sviluppatore che si riflette nel codice. **È una caratteristica statica del codice.**

Errore: la causa che porta all'insorgere di un difetto. È una decisione inappropriata del progettista, spesso dovuta a incomprensioni o errori di logica.

Relazione tra Failure, Defect ed Errore

- **Failure:** è il modo dinamico con cui si manifesta una deficienza del codice (defect) durante l'esecuzione.
- **Defect (o Bug):** è una caratteristica statica del codice causata da una decisione sbagliata.
- **Errore:** è ciò che ha portato al defect, spesso una decisione inappropriata o una comprensione errata da parte del progettista.

Il tester ha il compito di trovare i bug che possono causare failure.

Un tester bravo o fortunato riesce a individuare questi difetti che compromettono il funzionamento del sistema.

Il testing non si concentra principalmente sugli errori di programmazione, ma sui **problematiche logiche o di comprensione**.

È possibile avere un programma dove tutti i metodi funzionano correttamente, ma il sistema nel suo complesso non funziona a causa di assunzioni non concordate o non condivise (**esempio:** non si rispettano le interfacce)

Il testing è fondamentale per garantire che il sistema funzioni correttamente in contesti reali. Un programma può avere tutti i suoi metodi funzionanti, ma se le assunzioni alla base del sistema sono errate, l'intero sistema può fallire.

Come gestiamo queste cose?

Per affrontare i problemi nel software adottiamo tre strategie principali:

- **Fault detection:** la faremo in questo corso, **scoprire quanti più errori possibile**

- **Fault avoidance:** usare una serie di metodologie per **evitare** l'insorgere di **errori**
- **Fault tolerance:** programmare sapendo che ci sono errori, ma **minimizzando i bug**

Il testing è una sottoclasse di Fault Detection.

Miti

- **È impossibile testare tutto:** Questo è un problema sia pratico (il numero di prove sarebbe infinito) sia teorico.
- Il testing può mostrare la presenza di errori, non la loro assenza (Dijkstra)

Testing is not for free, vanno progettati così come abbiamo progettato il codice

Come definire le strategie

Definire la strategie e venire da gerardo con i casi di test

Le strategie possono essere di 2 tipi:

- **Strutturali (White Box):** si basa sulla conoscenza della struttura interna del software.
- **Behavioral (Black Box):** si basa sul comportamento del software senza considerare la struttura interna.

Ho un programma su cui devo fare il testing, devo decidere i casi di prova

Supponiamo che il programma faccia una cosa del genere

```
int m(x,y)
if(x>y) A
else B
```

Per un **approccio strutturale (white box)**, dobbiamo trovare tanti **input** che **consentano** di **eseguire tutti gli statement del programma**, assicurandoci che il programma funzioni correttamente in ogni caso.

In un caso devo avere il codice, nell'altro caso devo avere il sistema

Types of testing

Ci concentreremo principalmente sul **functional testing** (il sistema fa ciò che ci si aspetta che faccia). Altri tipi di testing includono:

- **Performance Testing:** verifica se il sistema funziona sotto stress.
- **User Testing:** fallisce se l'interfaccia non comunica bene con l'utente.
- **Security Testing:** assicura che il sistema sia sicuro contro minacce.

- **Scalability Testing:** valuta la capacità del sistema di crescere e gestire carichi crescenti.

Livelli di testing

- **Unit Testing:** verifica che oggetti e metodi si comportino correttamente
- **Integration Testing:** dopo aver verificato che le singole classi funzionano bene, si verifica se funzionano bene insieme
- **Acceptance Testing:** valuta se il sistema soddisfa le esigenze e le aspettative dell'utente

Quando scrivere i casi di prova?

Idealmente, i **casi di prova dovrebbero essere definiti all'inizio del processo di sviluppo**, utilizzando elementi come le user stories e gli scenari. Nei modelli a cascata, i test si scrivono alla fine.

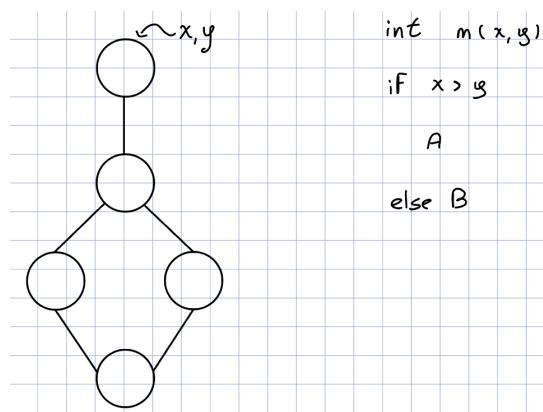
White Box

Vediamo il ruolo dei **Grafi** nei processi di test

Abbiamo visto che con un approccio **white box** scegliamo i casi di test con l'obiettivo di coprire **tutte le caratteristiche strutturali**

Le caratteristiche strutturali vengono spesso mappate con dei grafi.

Esempio:



Fare testing vuol dire "**toccare tutti i nodi, archi del grafo**"

Si crea quindi un **modello del sistema** con un **grafo** e il testing si riduce a voler eccitare certi statement del sistema

Un grafo è un insieme di nodi e di archi

C'è un set di nodi **iniziali** e **finali** **NON vuoti**

Gli **archi** sono **coppie di nodi**, sostanzialmente sono **relazioni di precedenza**

Un **cammino (path)** è un **insieme di nodi** dove questi a coppia a coppia definiscono un **edge**

Le **lunghezza** del **path** è data dal **numero di nodi**

Da un path si può estrarre un **sottopath**

Test path: un cammino che inizia in un nodo iniziale e termina in un nodo finale

I grafi che noi costruiremo sono **SESE (Single Entry-Single Exit)**

Visit: un test path visita un nodo se questo è compreso nel test path, lo stesso per un arco

Tour: un test path visita un sottopath se questo è compreso nel test path

path(t): il test path eseguito dal test t

path(T): l'insieme di test path eseguito dall'insieme di test T

Ogni test esegue uno e uno solo test path

Una location del grafo (nodo o arco) può essere raggiunta da un'altra se esiste una sequenza di archi dalla prima alla seconda

Definiamo:

- **sintatticamente raggiungibile** un subpath che esiste nel grafo
- **semanticamente raggiungibile** un test che quando eseguito copra il subpath

Altre definizioni:

- **test requirement (TR):** descrive l'obiettivo del test
- **test criterion:** regole che definiscono i requisiti del test
- **satisfaction:** dato un insieme TR per un criterio C, diciamo che un insieme di test T soddisfa C su un grafo se e solo se per ogni test requirement di TR esiste un test path in path(T) che soddisfa il requirement di tr
- **structural coverage criteria:** definito su un grafo solo in termini di nodi e archi
- **data flow coverage criteria:** richiede un grafo annotato con i riferimenti alle variabili

Node e Edge Coverage

Questi due criteri richiedono che **ogni nodo e arco** nel grafo sia **eseguito**

Node Coverage (NC): un insieme di test T soddisfa il criterio su un grafo G se per ogni nodo n sintatticamente raggiungibile esiste un percorso p in $\text{path}(T)$ tale che p visita n

Edge Coverage (EC): il test requirement TR contiene ogni path raggiungibile di lunghezza massima l presente nel grafo G . Questo è un criterio più complesso

In generale, fare statement (node) coverage è più debole di edge coverage

Infatti **edge coverage incorpora** anche **statement coverage** dato che se si coprono tutti gli archi, allora sono stati coperti anche tutti i nodi

NC e EC hanno una sola differenza, cioè quando da un nodo si diramano più percorsi

Esempio: if-else



Dobbiamo dare due ingressi per fare edge coverage:

- uno per fare [1, 2, 3]
- uno per [1, 3]

Se un grafo contiene i **loop** allora idealmente ci sono **infiniti path**

Negli anni '70 l'approccio utilizzato è stato quello di considerare i cicli come degli if, cioè eseguiti solo una volta

Successivamente, si è pensato che ci fossero due possibilità: entro nel ciclo o non entro

Oggi, l'approccio che si utilizza è **prime path**, cioè path che non compaiono in nessun altro path

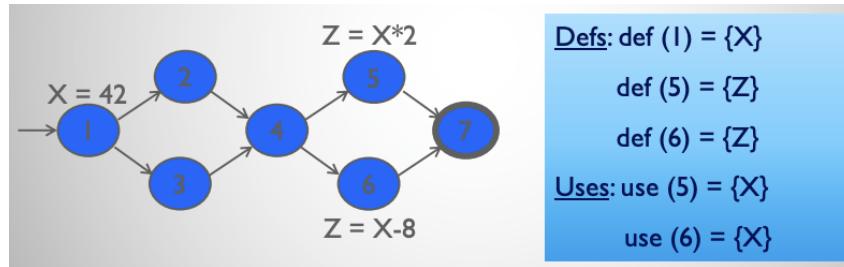
Data Flow Criteria

Obiettivo: provare ad assicurare che i **valori** siano **utilizzati correttamente**

Definiamo:

- **definition (def):** una location del grafo in cui viene memorizzato il valore di una variabile

- **use:** una location dove il valore della variabile viene utilizzato



DU Pairs: una coppia di location (i, j) dove una variabile viene definita nella prima e utilizzata nella seconda

Def-clear: un path è def-clear rispetto ad una variabile v , se nel path alla variabile v **NON** viene assegnato un nuovo valore in nessuna location

DU Paths: un semplice sottopath che è def-clear rispetto a v **dalla definizione di v fino al suo utilizzo**

Decidere di focalizzarsi su DU Paths è importante perché in questo modo mi assicuro che tutte le **parti del grafo** che useranno una variabile definita da qualche parte, la **useranno** in uno **stato consistente**

All-defs coverage (ADC): si parla di ADC se TR contiene almeno un path d in S, dove S è un insieme di du-path che iniziano in un nodo n

All-uses coverage (AUC): si parla di AUC se TR contiene almeno un path d in S, dove S è un insieme di du-path che iniziano in un nodo n_i e finiscono in un nodo n_j

All-du-path coverage (ADUPC): TR contiene tutti i du-path considerati

Gli approcci data flow tendono ad essere più forti

White box: prendo il sistema, lo mappo su un grafo e decido il criterio di copertura

▼ Lezione 19 - 20/05/2024

Code coverage

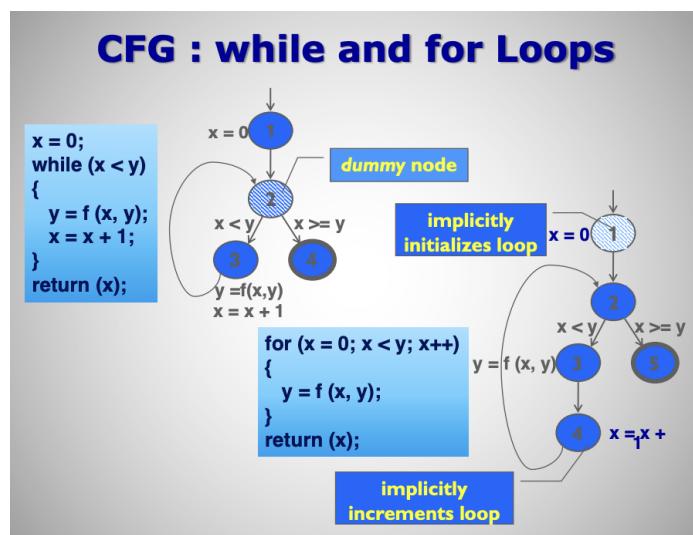
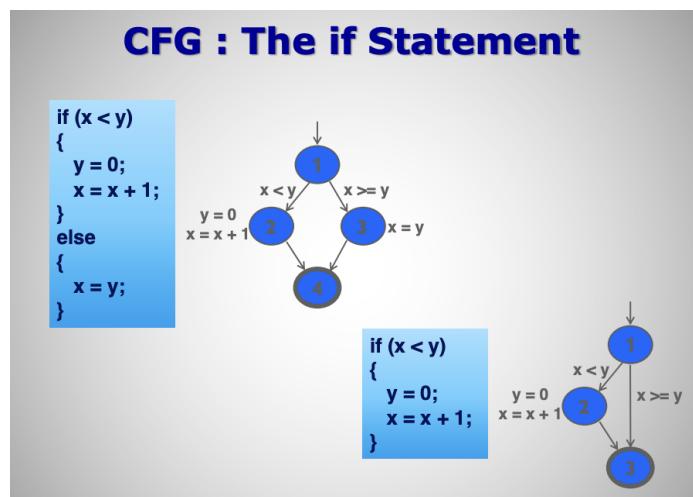
Edge coverage: eseguire ogni branch

Node coverage: eseguire ogni statement

Se entriamo in un if e vado a destra (valore vero) e ci sono altre istruzioni, allora tutte queste verranno necessariamente eseguite in blocco, per cui le **collassiamo in un solo nodo**

Ci piace ragionare a livello di **basic block**, cioè sequenze di statement dove una volta che si entra non si può uscire

Esempi di Control Flow Graph



Logic Coverage

Esiste anche la copertura logica - semantica

Con ciò vogliamo intendere che non è sempre sufficiente coprire entrambi rami di un if

```
while(x[i] != n && i < n)
    fai qualcosa
```

Supponiamo di cercare un numero in un array, fare edge coverage vuol dire uscire una volta a destra e una a sinistra, **NON** mi garantisce tutti i **possibili motivi per cui si verifica una situazione di uscita**, usciamo e basta

Un approccio logico semantico richiede che dopo aver fatto il grafo e individuato i modi per andare a sinistra o a destra, dobbiamo controllare quali sono le espressioni che ci fanno andare in una direzione

Si devono esplorare almeno le combinazioni principali

Ogni qualvolta che ci muoviamo sul grafo, con le condizioni di stato dobbiamo andare a vedere quali sono le possibili famiglie di decisioni e testarle tutte

Possiamo avere che da qualche parte abbiamo un programma che fa `if a < b`

Otteniamo un vero o un falso, ma basta andare a sinistra e destra? In realtà per quanto abbiamo visto fino ad adesso **si**, ma bisogna osservare lo stesso le condizioni che mi portano a quel risultato

Definiamo:

- **predicate coverage:** si suppone che un predicato sia vero o falso, si esprime con un'espressione booleana
- **clause coverage:** è un predicato senza operatori logici

La strategia è uno strumento di progettazione, il risultato è tremendamente **concreto** perché restituiamo a chi deve effettuare il testing un insieme di casi di prova

UNIT TESTING

Una **code unit** è qualsiasi cosa che ha una **responsabilità chiaramente definita**

Tipicamente si tratta di una funzione o una classe metodo

L'**unit testing** si basa sul principio di:

se un'unità di programma si comporta come previsto per un insieme di input con caratteristiche comuni, allora si comporterà allo stesso modo per un insieme più grande di input tutti con caratteristiche comuni

L'unit testing ricerca fault nelle componenti del sistema in relazione agli use case.

Ogni componente è testata in isolamento rispetto alle altre.

In generale, è possibile applicare qualsiasi strategia di testing

Nella pratica quelle di **coverage** si applicano bene a **livello di unità**, mentre altre di tipo **black box** che si basano su scenario e relazioni d'uso vanno meglio a livelli più alti

Come troviamo gli errori? Mettendoli in evidenza, facendo sì che ci sia un mismatch tra oracolo e risultato ottenuto

Copertura accidentale: nonostante abbiamo copertura totale, non abbiamo riscontrato l'errore

Se per decidere il caso di prova **guardo il codice**, parliamo di **glass box testing**

Black Box

Utilizza un approccio logico

Andiamo a guardare la **documentazione** che abbiamo e rispetto ad essa, **senza vedere il codice** e la **struttura interna dei dati**, possiamo definire i nostri **test case**

Qual è in questo caso un approccio tipico per definire i casi di test?

Dobbiamo partizionare i possibili scenari d'uso del sistema in **classi di equivalenza**

In pratica, seguendo dei criteri, partizioniamo tutti i possibili casi di utilizzo del nostro sistema in famiglie delle quali abbiamo la ragionevole certezza che avendone eseguito uno o pochi casi, allora l'intera famiglia risulta per noi comportarsi correttamente

Di solito lo facciamo rispetto ai dati di ingresso, cioè consideriamo **input** e **stato** del sistema

Esempio: dobbiamo scrivere un sistema che manda un messaggio ai maggiorenni che non hanno votato, senza sapere il codice

Possiamo iniziare a ragionare su un insieme di casi:

- query al database per ottenere tutti quelli che hanno votato
- portiamo il database in uno stato in cui tutti hanno votato, eccetto un aente diritto

Possiamo quindi suddividere l'insieme di dati in sottoinsiemi per vedere se vengono allertati

L'obiettivo è **partizionare in classi significative**

Esempio di partizione sui mesi

I casi per testare se il mese ha il numero corretto sono:

- $(-\infty, 0]$
- $[1, 12]$
- $[13, \infty)$

Quindi ciò che viene fatto spesso è lavorare a **boundary**

Per gli approcci white siamo stati sistematici, invece in questo caso stiamo andando avanti per esempi perché dobbiamo trovare un modo significativo di partizionare i casi d'uso vedendo gli scenari

Esempio: cercare l'elemento in un array

- non c'è
- c'è e si trova in ultima o prima posizione

Ci sono alcune dimensioni su cui possiamo categorizzare il testing:

- white
- black
- **Dimensione della grana, grana fine:** andiamo a vedere i singoli metodi, ad [esempio](#) UNIT TESTING dove si va a vedere come si può comprendere se un metodo è scritto bene o male

In generale, **NON basta che le parti siano corrette, c'è bisogno che i rapporti tra le parti siano progettati e rispettati in maniera corretta** (abbiamo introdotto il concetto di contratto e precondizione forzata con le asserzioni)

Integration Testing

Possono esistere situazioni in cui nei programmi ci sono degli errori che **NON** derivano dagli errori di una parte, ma dalla loro **integrazione perché si fanno delle assunzioni reciproche sbagliate**

L'integration testing è un primo livello per andare a testare e scovare questi problemi

L'intero sistema è visto come un insieme di parti, ad [esempio](#) possono essere classi e moduli

Nel nostro caso sono classi ad un primo livello, mentre un servizio è a livello più alto

L'obiettivo dell'integration testing è vedere se le interfacce attraverso cui avviene la comunicazione funzionano correttamente

Qualche volta l'errore si annida in situazioni molto subdole

Esempio: noi facciamo una prenotazione attraverso un microservizio con cui mandiamo al database l'orario di arrivo e il numero di commensali e si effettua la prenotazione

Assunzione: locale romantico, se non mi mandi il numero di commensali allora si aspetta per 2
Può accadere però che chi ha progettato il sistema si aspetta 4 perché suppone formato famiglia

C'è un **mismatch sulle assunzioni**, l'integration testing ha a che fare esattamente con queste situazioni, vuole testare le **interrelazioni** tra le parti

Qual è il problema che si pone quando si fa testing di integrazione?

A seconda della strategia ci troviamo nella situazione di dover isolare un pezzo di codice rispetto agli altri componenti

Nelle strategie di testing di integrazioni spesso si costruiscono dei **falsi moduli (stab) che simulano una dipendenza di sistema**

In pratica, per lavorare sul tested unit faccio in modo che l'eventuale chiamata al database NON sia reale, bensì sia un modulo che restituisce esattamente quello che voglio io in modo da non avere problemi ad isolare i problemi che ci sono da driver e da stub

Chiameremo:

- **driver** un modulo che **chiama** il modulo **undertest** per i casi di prova
- **stub** un modulo che **isola ed emula** dei **fake values**

Questo è particolarmente utilizzato negli stili architetturali a layer

A seconda della strategia, dobbiamo vedere se ci conviene partire dal basso o dall'alto per poter testare in modo isolato

In un approccio bottom-up conviene utilizzare solo **driver**

Mentre con un top-down abbiamo bisogno di **stub**

In un mondo reale non si fa ne top ne bottom, ma si va per necessità e quindi nella maggior parte dei casi avremo un approccio ibrido

Potremmo anche avere degli **stub hardware**

Se siamo in un **project oriented** quello che succede è che abbiamo bisogno di fare il **system testing**, cioè vogliamo essere sicuri che il **sistema funzioni sulle condizioni date dall'utente** con il suo hardware, sua rete

Se siamo in un **product oriented**, tipicamente si fa **release testing**, cioè **verifichiamo** che il **servizio sia rilasciabile** al pubblico

Performance testing

Abbiamo già visto che esistono dei requisiti che sono i **NON functional requirements**

L'obiettivo è quello di provare a violare i requisiti non funzionali

Acceptance testing

In un mondo di servizi e prodotti è molto utilizzato

Rilasciamo le feature a livello incrementale ad un ristretto gruppo di utenti per vedere la loro reazione

Esempi di questo tipo di testing sono i rilasci in alpha e beta testing di alcuni servizi

Scenario based testing

È il testing che utilizzeremo per il project work

Trasformeremo le user stories in casi di test

Bisogna fare attenzione a portare nei casi di test tutte le possibili evoluzioni dello scenario, anche le più banali

▼ Lezione 20 - 24/05/2024

Oggi vediamo il testing con Carmine

JUnit

È il framework che utilizziamo per fare test

Ha gli stessi concetti di Maven per la configurazione, cioè i file devono essere in una certa gerarchia

È facile da usare, basta aggiungere annotazioni e asserzioni

Le asserzioni vengono messe all'interno del codice per verificare che determinate condizioni siano rispettate

Con `assertEquals` viene chiamato il metodo `equals`

Ad assertEquals passiamo come parametri l'**oggetto atteso (expected)** e quello **generato dal codice (actual)**

Ogni funzione che deve essere testata va inserita in un metodo annotato con `@Test`

Tutto ciò che è annotato con `@Test` viene eseguito da JUnit quando lanciamo maven test

Con `@BeforeAll` viene annotato un metodo che deve essere eseguito prima di eseguire i metodi di test

Ad **esempio** può essere utile per istanziare un oggetto e poi utilizzarlo per tutti i casi di test

`@AfterAll` viene annotato il metodo che deve essere eseguito dopo tutti i casi di test

Ad **esempio** può essere utile se in fase di test abbiamo sporcato il database e vogliamo ripulirlo

In realtà, non useremo Junit vero e proprio, ma `spring-boot-starter-test` che contiene JUnit

`<scope>test</scope>` indica che la libreria sarà linkata solo in fase di test, non viene portata nel jar

▼ Lezione 21 - 27/05/2024

Security

La sicurezza va garantita a livello di **servizio**

È il server che deve garantire in qualche modo la sicurezza

La fetta piccola del mondo delle problematiche di sicurezza su cui ci soffermiamo è quella dell'**autenticazione** e dell'**autorizzazione**

Definiamo due fasi:

- **autenticazione**: dimostriamo chi diciamo di essere
- **autorizzazione**: diritti e responsabilità che si hanno in base a che ruolo si ha nel sistema

Una volta che ci siamo loggati dobbiamo definire cosa possiamo fare e cosa no

Oggi quando si progetta software la **sicurezza è un requisito fondamentale**, è un first class requirement, perché siamo interconnessi

I tipici problemi di sicurezza legati ai sistemi in rete sono stati classificati in vari modi

Li distinguiamo in:

- **availability threats**: attacco che tenta di impedire l'accesso al sistema, quello classico è il DDOS

- **confidentiality threats:** si cerca di rubare informazioni personali in modo da effettuare l'accesso come utente valido
- **integrity threats:** attacco ransomware che critta il contenuto dati e chiede il riscatto

Alcuni pattern di progettazione che abbiamo visto sono essenziali per diminuire queste esposizioni, abbiamo parlato di eliminare dati in classi che non ci appartengono (reference escape)

Esistono una serie di organismi in giro per il mondo che emettono CVE, cioè degli avvisi riguardo attacchi noti e vulnerabilità noti (**Common Vulnerabilities and Exposures**)

Questo può voler dire dover fare aggiornamenti incrociati a causa delle dipendenze, cioè se aggiorno A e devo aggiornare anche B

Vediamo alcune tecniche per mantenere il sistema in sicurezza

Attack monitoring: area che effettua il monitoraggio continuo dell'infrastruttura di rete e possiede impianto applicativo per scoprire eventuali attacchi.

Ci sono tecniche molteplici per farlo:

- approcci statistici che considerano le anomalie, cioè se noto un'addensarsi di query che non è corrispondente ad un profilo statistico, allora può essere che qualcuno sta cercando di sfondare il mio sistema, utilizzano le time series

Backup: bisognerebbe avere una politica di backup affidabile

Autenticazione e Autorizzazione

Autenticazione: processo che cerca di garantire la veridicità dell'identità

Autorizzazione: insieme di regole che permettono determinate operazioni in dei luoghi del sistema

Con la prima rispondiamo alla domanda "who you are", con la seconda "what you are allowed to do"

Oggi molto spesso si parla di autenticazione facendo riferimento a tre grandi famiglie che fanno capo al possedere qualcosa, conoscere qualcosa e mostrare uno o più dei propri attributi:

- **knowledge:** **password** (**esempio:** io sono Gerardo canfora se sono in grado di dare la password per Gerardo canfora)
- il **secondo livello è quello di possedere qualcosa**, per effettuare l'operazione online le banche rilasciavano un otp fisico

- attribute based, cioè si dimostra chi si dice di essere mediante un attributo biometrico (impronta digitale, riconoscimento facciale)

HTTP Basic

Vediamo tutto in relazione all'applicazione che dobbiamo sviluppare che prevede un colloquio con HTTP

HTTP offre un meccanismo base, detto **basic authentication scheme**

Questo richiede la **collaborazione** del **client**

Per tutte le risorse lato server a cui vogliamo restringere l'accesso, dobbiamo gestire una lista di utenti autorizzati ad accedere

Tuttavia, **passeremo in chiaro** nell'header request sia **username** sia **password** e verranno passate ad **ogni interazione** per rispettare la natura stateless del colloquio

Se la request fa riferimento ad una risorsa ad accesso ristretto, allora si va a controllare se nell'header ci siano username e password

Se dovesse andare storto qualcosa ci viene restituito una pagina di autenticazione e quindi il browser aprirà un pop-up che ci chiederà le credenziali

In realtà, questo modo di fare sicurezza è tutt'altro che sicuro perché le credenziali camminano in chiaro

Browser cooperation

Durante la sessione, il browser memorizza username e password e automaticamente le invia nell'header delle richieste successive

Lo svantaggio è che non è possibile effettuare un logout, ma bisognerebbe terminare il processo in esecuzione

Accesso federato

Esempio: Spid è un modo di delegare la gestione della fase di autenticazione, cioè anziché avere la fase di autenticazione implementata in ogni sistema (responsabilità di dover gestire dagli attacchi la base dei propri utenti) c'è la base di delegare questo ad un authority esterno

Il servizio gestisce un colloquio con un **trusted authority** per fare l'autenticazione

La trusted authority colloquia con l'utente e restituisce un token di autenticazione al servizio

Vediamo ora l'**authorization**

Come possiamo definire chi può fare cosa? **Normalmente lo facciamo tramite policy di accesso o di controllo degli accessi**

Esistono dei meccanismi che ci dicono chi può accedere a quali risorse del sistema per fare cosa

Come facciamo poi a gestire a runtime questi livelli di policy?

Ci sono molti modelli

Il più semplice è la **access control matrix**, una matrice dove si definiscono quali sono i **diritti** di ogni **oggetto** rispetto agli altri oggetti che il sistema gestisce

È un modello concettuale, quello che c'è dentro lo decidiamo noi nella policy

In realtà si tende ad utilizzare modelli che anziché utilizzare la matrice tendono a focalizzarsi sui **soggetti** o sulle **risorse**, cioè **capability list** e **access control list**

In pratica si tratta della stessa cosa, la differenza sta in cosa viene rappresentato sulle righe e cosa sulle colonne:

- **capability: utenti sulle righe, risorse sulle colonne**
- **access control list: risorse sulle righe, utenti sulle colonne**

Le access control list vengono usate sia dai file system che dai database per implementare la policy

Infatti, implementano dei filtri che vanno a verificare se l'operazione che si vuole fare è compatibile con la policy o con la ACL che è stata definita

Lega ogni utente a quello che può fare

L'alternativa alle ACL è la **role based access control (RBAC)**

È un concetto che è molto utilizzato al giorno d'oggi

Il concetto di base è quello di **ruolo**

Ruolo: ognuno di noi ha un particolare ruolo e i permessi vengono associati ai ruoli e non agli utenti

Quindi l'associazione viene gestita su due livelli e ciò dà il vantaggio di poter cambiare immediatamente lo status

La gestione per ruoli è molto diffusa oggi

Perché **NON** risolve tutto?

Se dico che sono un system administrator di un sistema qualunque devo creare e cancellare utenti e messaggi (immaginiamo di moderare)

il **concetto di owner** **NON** è un ruolo di per se perché io sono il proprietario di tutti e soli i miei messaggi

Ci sono anche altre tecniche per proteggere le info, come i metodi di crittografia

Carmine

Quando un client va a contattare un endpoint, abbiamo un server in ascolto su quella porta e ad ogni porta è associata una o più servlet

Il comportamento che si scatena quando viene matchato una richiesta e tipo di richiesta

Viene eseguito il codice e generata la risposta che sarà fornita al client

In realtà, prima di arrivare alla servlet bisogna passare all'interno di una serie di filtri che può servire a diverse cose

Esempio: intercettore di una chiamata, prende una richiesta (filtro bidirezionale)

Spring fa molto uso di questi filtri, infatti qualsiasi componente aggiunto di Spring è un filtro sopra la servlet

Un filtro va marcato con `@Component`

Siamo obbligati quindi a implementare il metodo `doFilter`

Si tratta di una **catena** perché il **filtro è bidirezionale, a valle la servlet e a monte il client**

▼ Lezione 22 - 31/05/2024

Vedremo due stili di progettazione

Vedremo un approccio che è fondato sull'idea di servizi che cooperano tra di loro, mentre dal punto di vista della progettazione sull'object orientation

L'idea di fondo della progettazione OO è quella di creare un bundle concettuale tra dati e strumenti per la manipolazione degli stessi

Cioè i dati e il codice che manipola questi non sono più separati tra di loro

Abbiamo visto come quest'idea si presta bene a realizzare situazioni stereotipate (design pattern) che sfruttano a pieno l'idea dell'OO, in particolare l'idea fondamentale che i sistemi debbano e possano esibire comportamenti polimorfici

Abbiamo visto come l'idea del polimorfismo ci consente di realizzare quasi tutti i pattern

Oggi vedremo un meccanismo per gestire polimorfismo, in particolare due stili diversi:

- uno basato sull'[inversion of control](#), cioè ad eventi
- mondo della [programmazione](#) / progettazione di tipo [funzionale](#) orientata alle funzioni

Recap di ereditarietà

La relazione fondamentale è che una classe è un tipo e allo stesso tempo è un sottotipo della classe padre

In generale, **una classe che eredita da un'altra classe**

La parola ereditare ci fa pensare più un rapporto meccanico che ad uno di tipo semantico

Un'istanza figlio è anche un'istanza padre, ma si tratta di un'**istanza specializzata**

Esempio: un'istanza della classe studente del corso di laurea in ingegneria civile è anche un'istanza della classe studente UniSannio

Tuttavia, a priori non so a quale classe appartiene

Il rapporto è prima di tutto **semantico**, poi concretamente comporta delle **implicazioni in base al linguaggio**

Lo stesso oggetto in base al lato da cui lo guardo può avere dei comportamenti diversi, infatti lo studente di ingegneria informatica mi sa dire qualcosa di testing, mentre quello di giurispinderà qualcosa di diritto pubblico, però entrambi sanno quanto pagano di tasse

Ad ogni oggetto vengono implicitamente associati un tipo statico e un tipo dinamico

Se ogni istanza di B è anche un'istanza di A, allora un riferimento `y` di tipo A, **potrà in qualsiasi momento del suo ciclo di vita riferirsi sia a un'istanza di A sia a un'istanza di B.**

Cosa succede se in B c'è un metodo m che non è presente in A? In questo caso, il sistema deve impedire di invocare il metodo m su un'istanza che non è di tipo B.

Il polimorfismo ha a che fare con questa doppia natura, B è anche un oggetto di tipo A, ma se **referenzio B tramite riferimento ad A ne vedo una fetta**

NON posso mai fare `B y = new A` a meno che non vado a fare un down sizing

La **dinamicità** sta nell'**override**

Possono esistere casi in cui il concetto che è al di sopra delle singole classi in realtà non esiste, ciò viene trattato con le **classi astratte**

Esiste quindi una **gerarchia concettuale**

La risoluzione dell'appartenenza ad un tipo viene risolta a run-time

In Android invece il late binding avviene in un mondo aperto perché non si conosce a priori quale procedura verrà attivata

La **gerarchia ereditaria** può servire a realizzare **meccanismi di delega**

L'effetto finale è lo stesso sia se estendiamo classi astratte sia se implementiamo interfacce

L'ereditarietà permette di evitare la ridichiarazione dei campi in comune, i quali vengono automaticamente ereditati dalle istanze delle sottoclassi.

Perché usare classi astratte e NON interfacce? Le classi astratte possono avere delle variabili di stato.

Per i pattern che abbiamo visto in cui la radice era una interfaccia, se le implementazioni complete hanno bisogno di una stessa struttura dati, sarò costretto ad implementare più volte lo stesso codice degenerando nell'antipattern **duplicated code**.

Classi astratte e interfacce hanno quindi due principi di design diversi:

- **l'interfaccia è un contratto:** obbliga a implementare dei behaviour
- **La classe astratta è un factoring:** fattorizzo in una classe tutto ciò che è comune ai sottotipi, mentre nei sottotipi implemento tutto ciò che è specifico

Design Pattern: Template method

L'utilizzo delle classi astratte porta alla nascita di un vero e proprio **pattern comportamentale** che si basa su questi principi fondamentali:

- fattorizzare tutto ciò che è comune alle sottoclassi in una superclasse astratta
- implementare un metodo di **Template** per i comportamenti comuni delle sottoclassi e dichiararlo **final** per impedirne l'overriding
- dichiarare abstract i **metodi primitivi** specifici che dovranno essere implementati dalle sottoclassi
- creare un **metodo concreto hook** che può essere eventualmente sovrascritto implementando logica aggiuntiva

Principio di sostituzione di Liskov

Afferma che gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del software che li utilizza

In pratica, se S è un sottotipo di T , allora oggetti dichiarati in un programma di tipo T possono essere sostituiti con oggetti di tipo S senza alterare la correttezza dei risultati del programma.

Inversione del controllo

Esempio: crea file da shell e sulle varie viste desktop compare il file

Normalmente siamo abituati a pensare al nostro codice come un padrone assoluto che cede e riprende il controllo in maniera selettiva

Ciò però non va bene perché se voglio fare il padrone del mondo devo avere il riferimento a tutto

L'inversione del controllo ragiona con il "le faremo sapere", cioè non si ha il riferimento, ma sarà questo a richiedere l'intervento del programma

L'**inversione del controllo** è un'idea del software design che consente di invertire il flusso di controllo che di solito va dal chiamante al chiamato per ottenere maggiore separazione delle responsabilità e diminuzione del coupling.

Un tipico contesto di utilizzo dell'inversione del controllo è nello sviluppo di interfacce grafiche

Per implementare l'inversione del controllo viene usato principalmente lo stile architettonicale **Model View Controller (MVC)** che fornisce tre tipi di astrazioni:

- **Model**: astrazione che mantiene la copia unica del dato di interesse
- **View**: astrazione che rappresenta una vista del dato
- **Controller**: astrazione che implementa la funzionalità necessaria per cambiare quel dato

Design Pattern: Observer

L'idea del MVC viene concretizzata in questo pattern

Consente di memorizzare dei dati di interesse in un oggetto specializzato e permette ad altri oggetti di **osservare** quel dato

L'**oggetto incaricato di mantenere il dato** è un'istanza di **Model** (o **Subject**), mentre gli **oggetti interessati ad essere aggiornati sui cambiamenti** sono istanze di classi che implementano un'**interface Observer**.

Il **Model** deve fornire un **metodo** di interfaccia per **aggiungere (o rimuovere)** degli **Observer** dalla sua collezione: questa operazione prende il nome di **registrazione degli observer** (può anche essere fatta a run-time)

Per venire a conoscenza del cambiamento di un dato gli **Observer** hanno un metodo di interfaccia `update()`, detto metodo di **callback**, proprio per il meccanismo di inversione del controllo: infatti **NON** sono gli observer a chiamare il model, ma **essi attendono che sia il model a chiamarli**

Il model non dice agli observer cosa fare, ma li informa sull'avvenimento di un dato cambiamento

In generale ci sono due modi per notificare gli observer:

- **inserire un metodo per la notifica in qualsiasi altro metodo che è in grado di cambiare lo stato**
- **chi modifica il model può decidere se una modifica è da propagare agli observer**

Come fanno gli observer ad avere accesso alle informazioni di cui hanno bisogno dal model?

Due possibili strategie:

- **push data-flow strategy:** rendere l'informazione di interesse disponibile attraverso uno o più parametri della callback; questa strategia però richiede che il *Model* conosca il tipo di dato di cui l'*Observer* è interessato
- **pull data-flow strategy:** gli *Observer* effettuano delle query definite dal *Model*; in questo caso gli *Observer* devono possedere un riferimento al *model*

Programmazione orientata agli eventi

Paradigma di programmazione che si basa proprio sul meccanismo dell'inversione del controllo: il *model* rappresenta la sorgente degli **eventi** e gli observer sono gli **Handler** che gestiscono tali eventi.

Un evento è ad esempio un cambiamento di stato di interesse per degli handler che vi si registrano.

Un particolare tipo di observer è l'**Adapter**, il quale implementa dei comportamenti di tipo *doNothing*, ossia non esegue alcuna operazione quando avvengono certi eventi.

▼ Lezione 23 - 03/06/2024

Cappello che non fa capire niente

Abbiamo sempre utilizzato la programmazione OO, oggi vediamo un approccio diverso

La volta scorsa abbiamo detto che un vero approccio OO si nutre anche di polimorfismo e abbiamo discusso riguardo le relazioni di tipo.

Ci sono dei problemi (che abbiamo anche incontrato) che sono difficili da implementare nella logica della programmazione e la progettazione OO pura

Esempio: interface ActionListener

Abbiamo introdotto il concetto di programmazione ad eventi e **programmazione reattiva** che sono alla base dello sviluppo di sistemi GUI

Serve anche per progettare **sistemi reattivi**, come il monitoraggio di un malato, di un ambiente industriale, domestico ecc...

Il tutto si risolve in una serie di contratti dove il **bottone consentiva agli altri di registrare un proprio callback**, per cui quando si faceva un'azione sul bottone la reazione a questo non era a carico del bottone in quanto lui si deve limitare a fare il **dispatch**, cioè mandare un evento agli interessati

Il **listener** è l'istanza di una classe che sottoscrive l'interfaccia `ActionListener`.

Vedendo l'interfaccia ci accorgiamo che obbliga ad implementare un solo metodo che è `actionPerformed`

Stiamo costruendo un'impalcatura che fa in modo che quando premo il bottone, allora chiama in gioco chi deve fare l'azione successiva

Affronteremo la progettazione con un orientamento alle funzioni che diventerà un orientamento allo stream

Tutti ricordiamo cos'è uno stack, i suoi quattro metodi hanno una **signature** che è:

- $s \Rightarrow s'$ (s in s') per quanto riguarda il `push` e il `pop`, i quali non fanno altro che prendere un elemento in ingresso e costruire un altro stack (con un elemento in più o in meno)
- $s \rightarrow t$ per il `top`
- $s \rightarrow \text{boolean}$ per `isEmpty` (ci dice se lo stack è vuoto o meno)

Queste singature sono direttamente mappate in una progettazione con side effect, infatti abbiamo assunto che il push e pop cambiassero lo stato dell'oggetto

È possibile progettare in maniera conservativa rispetto allo stato

Abbiamo parlato di classi immutabili, potrei progettare quella classe rispettando la signature di principio senza side effect, cioè se ho uno stack s e ci faccio un push ottengo il vecchio con nuovo elemento

Quindi dal punto di vista Java il nostro push da `void push(T t)` il nuovo push diventa `Stack push(T t)`

Avere un approccio orientato alla funzione non è in contraddizione rispetto ad avere un approccio orientato agli oggetti

Oggi vedremo cosa ci offre Java per integrare delle aree che si comportano in maniera funzionale

First class functions

In Java, abbiamo esaminato il concetto di first class function attraverso l'uso del metodo per il sorting di una collezione.

In particolare, abbiamo considerato un **esempio** in cui siamo chiamati a fornire un **comparatore** per effettuare l'**ordinamento**. Questo evidenzia un aspetto fondamentale delle first class function: la capacità di passare comportamenti (funzioni) come argomenti.

Nel caso specifico, **la collezione può essere ordinata se viene fornito un comparatore**.

Per fornire questo **comparatore**, utilizziamo un'interfaccia `Comparator` generica, che ci obbliga a implementare un unico metodo. Questo metodo prende due oggetti (nel nostro esempio, due carte) e restituisce un intero. In altre parole, **stiamo implementando una funzione pura, poiché la comparazione non richiede uno stato**.

Tradizionalmente, in Java, la creazione di una funzione richiede la creazione di un oggetto. Ad esempio, avremmo potuto creare un oggetto che implementa il metodo di comparazione.

Tuttavia, se avevamo già definito un comportamento simile in un'altra parte del codice, come un metodo `compareByRank` nella classe `Card`, non potevamo facilmente estrarre e riutilizzare questo comportamento fino a Java 8.

Java 8 ha introdotto un cambiamento significativo in questo ambito.

Ora, **possiamo ottenere un riferimento a un metodo di una classe e utilizzarlo altrove senza dover creare nuovi oggetti**. Questo processo avviene attraverso i "**Method References**".

Dal punto di vista del runtime, non è cambiato nulla nel funzionamento sottostante: stiamo semplicemente dicendo al linguaggio di ignorare la necessità di creare un nuovo oggetto e di utilizzare il comportamento già definito altrove.

In programmazione, una funzione è un'entità che produce un output $f(x)$ dato un input x , senza dipendere dallo stato. Se una funzione dipende dallo stato, non è considerata una pura funzione ma un altro tipo di costrutto.

In molti linguaggi di programmazione, inclusi quelli orientati agli oggetti come Java, trattiamo gli oggetti come first class entities.

Ad **esempio**, possiamo avere un oggetto con un metodo che accetta come parametro un'istanza di un'altra classe.

Allo stesso modo, **desideriamo trattare le funzioni come entità di prima classe: funzioni che possono essere passate come argomenti ad altre funzioni senza doverle necessariamente incapsulare in oggetti.**

Il passaggio di funzioni come argomenti è un concetto fondamentale della programmazione funzionale. In Java, questo è stato reso possibile con l'introduzione delle **functional interfaces** e dei **method references**. Questo approccio ci consente di avere funzioni che diventano argomenti per altre funzioni

Functional Interfaces

Una **functional interface** è un tipo di interfaccia che dichiara un singolo metodo astratto.

A partire da questa interface possiamo dichiarare classi (anche anonime) che la implementano.

Le

functional interface possono essere viste come **un tipo di funzione**, in quanto **NON hanno stato e implementano un singolo metodo**.

A partire da Java 8 è possibile definire all'interno delle interface dei metodi **static** (già definiti e non sovrascrivibili nelle classi che implementano l'interface) e **default** (già definiti ma con possibilità di overriding). Per definizione questi metodi NON sono astratti.

Lambda expression

Le lambda expressions in Java permettono di definire funzioni anonime in modo conciso. Sono particolarmente utili quando abbiamo bisogno di un comportamento che non richiede uno stato, rendendo il codice più leggibile e riducendo la necessità di classi anonime o implementazioni verbose.

Una lambda expression in Java è composta da tre parti principali:

1. **Lista dei parametri**: uno o più parametri, optionalmente con tipi esplicitati.
2. **Operatore freccia (→)**: che separa i parametri dal corpo della lambda.
3. **Corpo**: che può essere una singola espressione o un blocco di codice (statement Java). Se è un blocco di codice, è necessario usare `{}` e `return` per restituire un valore.

```
# Questo esempio verifica se il colore del seme di una carta è nero.
```

```
(Card card) -> card.getSuit().getColor() == Suit.Color.BLACK
```

Java permette di omettere il tipo dei parametri nella lambda expression, purché il tipo possa essere dedotto dal contesto. Ad **esempio**:

```
Predicate<Card> blackCardFilter = card -> card.getSuit().getColor() == Suit.Color.BLACK;
```

Quando abbiamo una collezione navigabile, possiamo usare le lambda expressions direttamente all'interno della navigazione per filtrare gli elementi.

Ad **esempio**, se abbiamo una collezione di carte, possiamo filtrare e ottenere solo le carte nere in un'unica operazione.

Le lambda expressions possono essere utilizzate per implementare o fornire comportamenti ad oggetti che fanno parte delle librerie standard di Java. Questo significa che possiamo applicare meccanismi della libreria a collezioni personalizzate o eseguire operazioni complesse in modo conciso e leggibile.

Tuttavia, è importante considerare se il metodo di riferimento è statico o non statico.

Quando utilizziamo method references, è importante capire se il metodo è statico o di istanza:

- **Metodo di istanza:** è un metodo che deve essere chiamato su un'istanza di un oggetto. Ad esempio, `Card::isBlack` è un method reference a un metodo di istanza della classe `Card`.
- **Metodo statico:** è un metodo che può essere chiamato senza creare un'istanza della classe. Ad esempio, `Utilities::compareBySuit` è un method reference a un metodo statico.

Method Reference

Permettono di riutilizzare i metodi di una classe per realizzare una funzione: si isola così un metodo dal contesto e lo si utilizza per avvalorare una funzione.

La sintassi di una method reference è composta dal nome della classe seguita da `::` e infine il nome del metodo referenziato.

Composing behavior

In Java, l'introduzione delle lambda expressions e delle interfacce funzionali nel package `java.util.function` ha reso possibile la creazione dinamica e la composizione dei comparatori in modo estremamente versatile.

Questo approccio è particolarmente utile quando si desidera confrontare oggetti su più criteri e ordinare in modi diversi, come ascending (crescente) e descending (decrescente), senza la necessità di scrivere comparatori statici per ogni combinazione possibile.

Partiamo dall'utilizzo di `Comparator.comparing`, che consente di creare un comparatore basato su una funzione di estrazione di chiavi (`keyExtractor`):

```
Comparator<Card> byRank = Comparator.comparing(Card::getRank);
```

Questo comparatore ordinerà le carte in base al loro valore

Il metodo `thenComparing` permette di comporre comparatori per eseguire confronti aggiuntivi in cascata in caso di parità. Ad esempio, possiamo comporre il comparatore `byRank` con uno per il seme (`suit`):

```
Comparator<Card> bySuitThenRank = Comparator.comparing(Card::getSuit)
                                         .thenComparing(Card::getRank);
                                         k);
```

Questo comparatore ordinerà le carte prima per seme e, in caso di parità di seme, per valore.

Java 8 e successive permettono di invertire l'ordine di un comparatore utilizzando il metodo `reversed`:

```
Comparator<Card> byRankDesc = Comparator.comparing(Card::getRank).reversed();
```

Questo comparatore ordinerà le carte per valore in ordine discendente.

L'uso di `Comparator.comparing` e `thenComparing` insieme a funzioni lambda permette di creare comparatori dinamici al volo.

Questi concetti trovano il massimo delle applicazioni nel contesto dello stream dei dati. Vedremo come la first class object (lambda), primitive di composizione e stream di dati danno vita al data processing che è molto diffuso oggi.

Streams

Lo stream è un metodo per elaborare dati a partire da un flusso di dati potenzialmente infinito, che può essere più o meno strutturato.

Differenza tra Flussi e Collezioni

Le collezioni hanno metodi di accesso spaziali, mentre negli stream l'accesso è sequenziale. Il flusso rappresenta l'elemento corrente, con limitati elementi circostanti (al massimo posso accedere a qualche elemento precedenza).

Caratteristiche dello Stream:

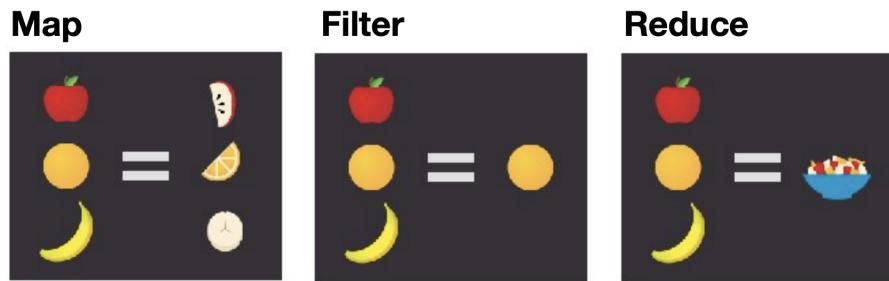
- Il flusso si consuma durante l'elaborazione, mentre la collezione no.

- Gli stream possono essere attraversati una sola volta.

Per costruire uno stream in Java, si parte da una collezione esistente convertendola con il metodo

`stream()`.

Operazioni sugli Stream:



- **Map:** trasforma ogni oggetto in un altro oggetto, ad esempio mappando ogni studente alla sua media.
- **Filter:** applica un filtro per selezionare solo gli elementi che soddisfano un dato criterio.
- **Reduce:** combina gli elementi dello stream in un unico risultato applicando un'operazione di aggregazione.

Queste operazioni possono essere applicate in vari ordini secondo le necessità dell'elaborazione.

Reflection

Strumento da utilizzare con cautela, **è un modo per trattare i nostri programmi come dati**

È il processo mediante il quale un programma può esaminare e modificare il proprio comportamento.

Un programma che può analizzare e modificare i propri parametri in fase di esecuzione è considerato *riflessivo*.

In altre parole, la **reflection è un modo per un programma** o un'applicazione **di conoscere se stesso e persino modificarsi**.

La Java Virtual Machine (JVM) mantiene sempre una definizione di tipo in fase di esecuzione per tutti gli oggetti, che determina i metodi corretti da chiamare. La classe `java.lang.Class` viene utilizzata per questo scopo.

Esistono tre modi per ottenere la classe `Class` di un oggetto:

- Utilizzare `(qualsiasi Oggetto).getClass()`. Questo funziona per gli oggetti che esistono già. Ad esempio, se si dispone di uno `Scanner scan = new Scanner(System.in)`, è possibile ottenere la classe `Class` con `Class c = scan.getClass()`.
 - Utilizzare `(qualsiasi tipo).class`. Questo funziona quando si conosce solo il tipo. Funziona anche per i tipi primitivi come `int.class` o `Scanner.class`.
 - Utilizzare `Class.forName("percorso completo")`. Questo funziona quando si conosce il nome della classe che si desidera ottenere.
- Ad esempio, è possibile ottenere la classe `Class` per `Scanner` con `Class c1 = Class.forName("java.util.Scanner")`.

La classe `Class` ha diversi metodi utili:

- `getName()` restituisce il nome della classe come `String`.
- `getConstructors()` restituisce un array di costruttori per la classe.
- `getMethods()` restituisce un array di metodi per la classe.
- `getFields()` restituisce un array di campi per la classe.
- `getInterfaces()` restituisce un array di interfacce implementate dalla classe.
- `newInstance()` restituisce una nuova istanza della classe. Questo funziona solo in determinati casi.

La classe `java.lang.reflect.Constructor` rappresenta un costruttore per un oggetto classe. Alcuni metodi importanti in `Constructor` includono:

- `getDeclaringClass()` restituisce la classe `Class` a cui appartiene il costruttore.
- `getModifiers()` restituisce i modificatori del costruttore come un `int`. I modificatori indicano se il costruttore è pubblico, privato, protetto, ecc.
- `getParameterTypes()` restituisce un array di classi `Class` che rappresenta i tipi di parametro del costruttore.
- `newInstance(Object[] args)` crea una nuova istanza della classe, passando gli argomenti forniti al costruttore.

La classe `java.lang.reflect.Method` rappresenta un metodo in una classe.

Alcuni metodi utili in `Method` includono:

- `getExceptionTypes()` restituisce un array di classi `Class` che rappresenta i tipi di eccezione che il metodo può generare.
- `getModifiers()` restituisce i modificatori del metodo come un `int`.
- `getName()` restituisce il nome del metodo come `String`.
- `getParameterTypes()` restituisce un array di classi `Class` che rappresenta i tipi di parametro del metodo.
- `getReturnType()` restituisce la classe `Class` che rappresenta il tipo restituito dal metodo.

- `invoke(Object obj, Object[] args)` richiama il metodo sull'oggetto specificato, passando gli argomenti forniti. È importante gestire le eccezioni quando si utilizza `invoke()`.

La classe `java.lang.reflect.Field` rappresenta un campo all'interno di una classe. Alcuni metodi utili in `Field` includono:

- `get(Type)(Object obj)` restituisce il valore del campo nell'oggetto specificato. `Type` è il tipo di dati del campo.
- `set(Type)(Object obj, Type value)` imposta il valore del campo nell'oggetto specificato. `Type` è il tipo di dati del campo e `value` è il nuovo valore da impostare.

▼ Lezione 24 - 07/06/2024

DevOps

Oggi approfondiamo il tema dello sviluppo software, esplorando il confine tra il dominio tecnico e la gestione aziendale, concentrandoci su **DevOps**.

Questo argomento interessa sia il mondo **project-based** che **product-based**.

Il tema centrale è: come gestiamo la vita di un prodotto dopo il suo rilascio?

Come arriviamo al rilascio e come gestiamo la vita del prodotto una volta rilasciato?

Tradizionalmente, questo processo era caratterizzato da una frattura all'interno dell'organizzazione.

Le **aziende** avevano e spesso hanno ancora un'**area di sviluppo composta da progettisti, analisti, formatori**, ecc., che lavorano sulle esigenze e sviluppano software.

Questo software viene quindi **invia**to a un **team di quality assurance (QA) indipendente**, che può essere interno o esterno all'azienda, per la verifica.

Il team **QA decide** se il **software è pronto** per il rilascio.

A questo punto, il team di sviluppo ha completato il suo lavoro, ma il **software deve essere mantenuto**.

Per questo viene istituita un'organizzazione che riceve feedback e svolge un monitoraggio attivo per identificare eventuali colli di bottiglia, svolgendo le attività operative necessarie.

In genere, il team di quality assurance è separato dal team di sviluppo.

Il software torna poi in un ambiente di sviluppo che può essere lo stesso dell'inizio, un sottoinsieme di esso o un team diverso.

Il flusso tipico è il seguente:

- Team di sviluppo
- Stage di integrazione e quality assurance
- Team di gestione

La gestione del software può essere attiva o passiva:

- **Attiva:** comporta il monitoraggio continuo del sistema per identificare e risolvere proattivamente i problemi.

- **Passiva:** prevede l'uso di un call center o di una pagina web per l'apertura di ticket da parte degli utenti, reagendo ai problemi segnalati.

Queste modalità di gestione influenzano il team di sviluppo e sono alla base del software support.

In passato, il rilascio di software avveniva ogni 5 anni ed era considerato un evento epocale che poteva fermare l'attività dell'azienda. Un fallimento in queste condizioni metteva a rischio la sopravvivenza dell'azienda.

Oggi, invece, il fallimento di un prodotto può essere trascurabile. La **riduzione dei tempi** di rilascio ha aumentato la capacità di risposta rispetto al processo controllato. L'**agilità** non si è limitata allo sviluppo, ma si è **estesa anche al processo di gestione**.

DevOps, abbreviazione di Development + Operations, è un approccio nato per velocizzare i processi di rilascio e supporto del software.

Tra i fattori che hanno portato allo sviluppo e all'adozione diffusa di DevOps c'è l'ingegneria del software Agile. Questo tipo di approccio come sappiamo ha ridotto i tempi di sviluppo, ma il processo di rilascio tradizionale ha introdotto un collo di bottiglia tra sviluppo e distribuzione. DevOps ha migliorato questo aspetto.

Nel contesto competitivo odierno, le aziende che non adattano una corretta gestione del software rischiano di affrontare gravi problemi di mercato.

Questo processo può essere descritto attraverso una spirale in tre fasi:

1. Pressioni Esterne e Gestione Reattiva

- Il mercato si muove alla velocità di ciò che è percepito dall'esterno.
Le aziende devono gestire le aspettative degli utenti e rispondere ai loro feedback. Ad esempio, se un'azienda non presta attenzione ai commenti lasciati sul marketplace della propria app, rischia di perdere contatto con le esigenze degli utenti.
Questa pressione esterna si riversa sul team di sviluppo, spingendolo a fare aggiornamenti rapidi e superficiali.

2. Debito Tecnico e Accumulo di Problemi

- Sotto pressione, il team di sviluppo spesso adotta scorciatoie, introducendo soluzioni temporanee e non ottimali. Questo porta all'accumulo di debito tecnico: problemi strutturali che rendono il codice meno leggibile, testabile e mantenibile.
Man mano che il debito tecnico cresce, il software diventa più difficile da gestire e migliorare, rendendo ogni aggiornamento successivo sempre più complesso e problematico.

3. Accelerazione della Spirale e Rischio di Mercato

- Con l'accumulo di debito tecnico, la qualità del software diminuisce, rendendo più difficile rispondere efficacemente alle richieste del mercato.
Ogni tentativo di rilascio accelerato amplifica i problemi, portando l'azienda in una spirale negativa che riduce la sua competitività.

Se non interviene con una strategia adeguata, l'azienda rischia di vedere compromessa la sua posizione competitiva.

Ciò che oggi chiamiamo **DevOps** rappresenta l'agilità dell'azienda nel rispondere alle sfide del mercato.

DevOps funziona perché si basa sull'automazione dei processi di sviluppo e rilascio, riducendo i tempi e migliorando la qualità del software. Grazie all'automazione, le aziende possono rilasciare aggiornamenti frequenti e di alta qualità, riducendo il debito tecnico e mantenendo una posizione competitiva nel mercato.

Principi DevOps

- **Ognuno è responsabile di tutto:** tutti i membri del team hanno responsabilità comuni per lo sviluppo, la consegna e il supporto del software
- **Tutto quello che può essere automatizzato deve essere automatizzato:** tutte le attività di test, distribuzione e supporto devono essere automatizzate, se possibile. Il coinvolgimento manuale nella distribuzione del software deve essere minimo
- **Prima si misura e poi si effettuano modifiche:** DevOps dovrebbe essere guidato da un programma di misurazione in cui si raccolgono dati sul sistema e sul suo funzionamento. I dati raccolti vengono poi utilizzati per prendere decisioni sulla modifica dei processi e degli strumenti DevOps.

Il concetto di DevOps può essere spinto all'interno dell'azienda a diversi livelli.

Continuous integration

Ogni volta che uno sviluppatore effettua una modifica al branch principale del progetto viene costruita e testata una versione eseguibile del sistema.

Continuous delivery

Viene creata una simulazione dell'ambiente operativo del prodotto e viene testata la versione eseguibile del software.

Continuous deployment

Una nuova versione del sistema viene resa disponibile agli utenti ogni volta che viene apportata una modifica al branch principale del software.

Infrastructure as code

Le infrastrutture come server, reti e router su cui viene eseguito il software vengono usati dagli strumenti di gestione per costruire la piattaforma di esecuzione del software.

Il software da installare, come compilatori e librerie e un DBMS, sono inclusi nel modello di infrastruttura.

Il livello più semplice di DevOps è l'integrazione continua (Continuous Integration, CI).

A questo livello i commit vengono verificati regolarmente e, se tutto va a buon fine, vengono integrati nel codice principale. L'evento che scatena questo processo può essere configurato secondo le esigenze del team.

Ma cosa significa in pratica? Nel nostro caso, abbiamo un server che guidato da uno script `.yml` esegue una serie di test. Questo script restituisce un elenco dei test che sono passati o meno.

Se tutti i test vengono superati, i nuovi commit vengono integrati nel codice principale, garantendo che il software rimanga stabile e funzionante.