

- SOFTWARE ENGINEERING: è un approccio sistematico, organizzato e quantificabile a tutti gli aspetti della PRODUZIONE del SOFTWARE:
  - progettazione
  - sviluppo / programmazione
  - manutenzione

Lo sviluppo del software si basa su attività, dette PRACTICES;

### PRACTICES:

- REQUIREMENTS engineering
- System analysis
- High level design / architecture
- Low level design
- Coding
- INTEGRATION
- Design and code reviews
- Testing
- Maintenance (manutenzione)
- Project management
- Configuration management

- BEST PRACTICE: Una ATTIVITÀ, un metodo che si è dimostrato superiore agli altri. Di solito, tale termine è usato per indicare una "Standard" che si fa perché tutti lo fanno e perché sembra ragionevole fare, ma non sempre è la cosa migliore (BEST) da fare!

### PROCESSO DI SVILUPPO SOFTWARE:

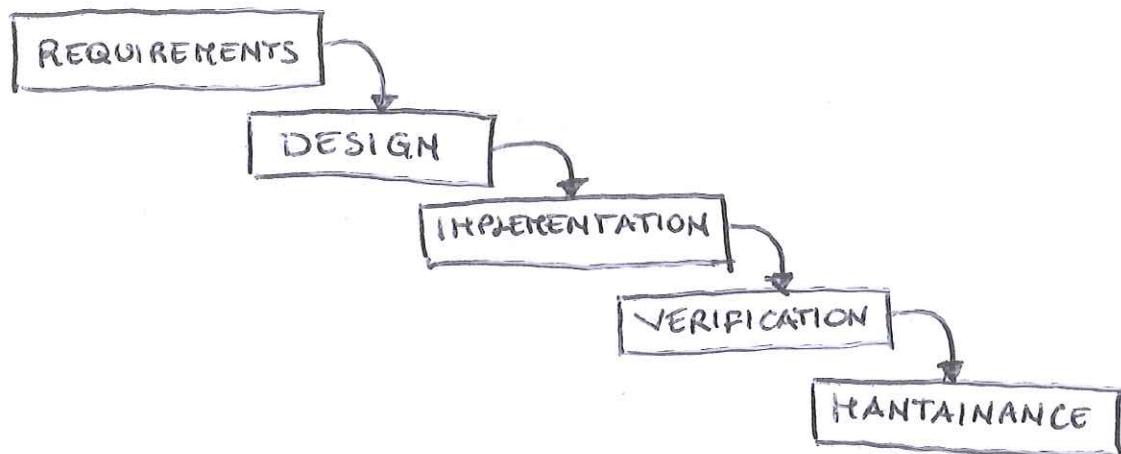
E' l'insieme delle PRACTICES, il loro ORDINE e la loro IMPORTANZA.

Un software development process può essere suddiviso in fasi, nelle quali vengono svolte tutte o solo alcune practices (attività), ponendo in una fase enfasi su alcune practices e in un'altra fase su altre practices.

### MODELLO DI PROCESSO:

E' una semplificazione e ASTRAZIONE di un processo di sviluppo software, ideato per determinare l'ordine delle fasi e i criteri di transizione da una fase all'altra. Essendo un'estrazione, processi di sviluppo software DIVERSI potrebbero condividere lo STESO MODELLO DI PROCESSO software.

## • MODELLO DI PROCESSO A CASCATA (WATERFALL) :



- È il primo modello di processo software delle storie, che ha il merito di aver organizzato le idee sui case fore in che ordine e di avere dato l'ecce<sup>sive</sup> attenzione all'implementazione
- Le fasi coincidono con le PRACTICES! Un'unica practice in ogni fase.
- Ogni practice viene svolta e COMPLETATA, dopodiché si pessa alla fase e alle attività successive, e NON si può tornare indietro  
→ MODELLO UNIDIREZIONALE
- Questo modello è ormai obsoleto, ma è stato un grande passo avanti per l'epoca!
- RUP (Rational Unified Process) :

- È suddiviso in 4 FASI: Inception, Elaboration, Construction, Transition.
- Ogni fase ha delle ITERAZIONI, in cui si può fornire qualcosa agli stakeholders per avere dei FEEDBACK; iterazioni lunghe, di circa 6 mesi.
- Durante le 4 FASI si applicano TUTTE LE PRACTICES insieme, dette "discipline"; ovviamente in alcune fasi c'è più effett su alcune discipline che su altre  
→ "GRAFICO DELLE BAIE"

## • MODELLO DI PROCESSO AGILE :

Sono simili al RUP, ma hanno delle ITERAZIONI molto BREVI (max 2-3 settimane), dette SPRINT.

- Ogni SPRINT è caratterizzato da 4 fasi: Discover, Design, Develop, Test.
- Ogni SPRINT è quello di MINIMIZZARE IL TEMPO di sviluppo, SENZA FEEDBACK del cliente. Chiedo il feedback solo alla fine dello sprint.  
Ci si concentra su qualcosa in 2 settimane alla volta, utile in contesti in cui requisiti non chiari all'inizio e progettazione molto VARIABILE!

## THE AGILE MANIFESTO :

- 1) E' vero che abbiamo bisogno di un processo e dei tool per automatizzare il più possibile la produzione, ma non dimentichiamo che gli sviluppatori sono umani e hanno bisogno di interazione. Deve sentirsi a suo agio per lavorare in maniera migliore, sfruttando il processo ed i tools. Dunque:
  - "Individuals and interactions over process and tools"
- 2) Per sì che il software funzioni bene, piuttosto che avere una documentazione. La documentazione non deve essere fine a sé stessa, ma di SUPPORTO al software.
  - "Working software over comprehensive documentation"
- 3) Collaborare con il compratore del software, piuttosto che negoziare il contratto. Se il customer è contento e soddisfatto ad ha successo, anche noi avremo successo.
  - "Customer collaboration over contract negotiation"
- 4) Rispondere attivamente ai cambiamenti e valutare se adattarsi o meno per trarre vantaggi, piuttosto che seguire pedissequamente un piano. Dunque:
  - "Responding to change over following a plan"

## USER STORIES & REQUIREMENTS

- **REQUISITO:** Un'espressione che descrive il comportamento, le funzionalità di un sistema, al fine di soddisfare un obiettivo.  
Devo specificare COSA il sistema fa, e NON COME lo fa!  
DOMINIO DEL PROBLEMA e non dominio delle soluzioni  
→ USE CASE VIEW
- **VERIFICA DI UN REQUISITO:** Verifica della CORRETTEZZA SINTATTICA del requisito. Se tale requisito è conforme o meno alle "regole" che caratterizzano un buon requisito.
- **VALIDAZIONE DI UN REQUISITO:** Verifica della CORRETTEZZA SEMANTICA rispetto alle richieste del cliente. Tanto più il requisito è conforme alle RICHIESTE del cliente, tanto più è valido!

- CARATTERISTICHE DI UN BUON REQUISITO:
- NECESSARIO: deve riguardare le reali necessità per il sistema, NON i desideri secondari degli utenti. Discernere le funzionalità principali.
- FATTIBILE: deve essere fattibile, rispettando i TEMPI ed i COSTI.
- CORRETTO: la descrizione deve essere accurata e deve essere tecnicamente e legalmente corretto.
- CONCISO: scritto in maniera semplice e col minor numero di parole possibili.
- NON AMBIGUO: deve essere chiaro e deve poter essere interpretato in una sola maniera → ONE WAY INTERPRETATION
- COMPLETO: tutte le condizioni del requisito sono stabilite e il requisito esprime un'idea intera e completa, di senso compiuto.
- CONSISTENTE: non deve essere in contrasto con altri requisiti.
- VERIFICABILE / TESTABILE: un requisito deve poter essere misurato e quantificato, per esempio: "il sistema deve essere veloce", oppure "il sistema deve essere facile da utilizzare" NON VANO BENE!
- TRACCIABILE: deve essere tracciabile alle sue sorgenti (stakeholders che lo richiedono) e tracciabile nel sistema. Essendoci requisiti a diversi livelli, è importante la tracciabilità tra di loro: il cambiamento di un requisito è facilmente propagabile agli altri requisiti.
- IMPLEMENTATION-FREE: deve descrivere il COSA e' richiesto e il sistema deve fare, NON IL COME farlo (che sarebbe dominio delle soluzioni).
- NON RIDONDANTE: non deve essere un duplicato di un altro requisito.
- COSTRUITO STANDARD: deve essere scritto in forme IMPERATIVA, usando le parole "shall" e non deve essere troppo complesso grammaticalmente.
- EVITARE CLAUSOLE DI ESCAPE: evitare di usare le parole "usually", "generally", "often", "normally", "typically".

## USER STORIES

Sono requisiti di tipologie utente; sono formate da 3 elementi:

- l'utente richiedente
- le funzionalità richieste
- i benefici che le funzionalità garantisce all'utente.

Dunque, seguono il seguente TEMPLATE:

As a <> user role <>, I want to <> do something with the application <>, so that <> I get specific benefits <>.

- \* Gli utenti nelle user stories, possono essere (e spesso sono) diversi rispetto agli utenti poi effettivi del sistema: per esempio potremmo avere un utente di tipo STUDENTE, che poi però si loggherà nel sistema come uno degli utenti previsti (per esempio COLUI CHE PRENOTA o COLUI CHE AFFITTA).
- \* Per capire se un requisito è buono o meno, bisogna mettersi dal punto di vista dello SVILUPPATORE e del TESTER e chiedersi se si avrebbe difficoltà ad implementare o a definire un caso di test per tale funzionalità.

## FUNCTIONAL REQUIREMENTS

Sono requisiti PRODUCT LEVEL. Descrivono quello che il SISTEMA FA, per andare incontro alle necessità degli utenti e dei clienti.

Sono molto più vicini al DOMINIO DELLA SOLUZIONE!

Seguono anche loro un template e delle linee guida:

"The system shall ..."

- Usare verbi alla FORMA ATTIVA: "the system shall provide <> some feature<>" e NON "<> some feature<> must be provided..."
- Evitare le NEGAZIONI: dire cose il sistema fa, e non cose non fa;
- Essere CONCISO;
- Utilizzare il più possibile il verbo "PROVIDE".

\* Di solito le User Stories sono scritte prima dei FR. Quindi i FR sono qualitativamente molto migliori.

\* Se esistono, le user stories sono usate per RICAVARNE i Functional Requirements. Può capitare che più FR's siano DERIVATI da 1 US.

## USE CASE

- **USE CASE (caso d'uso)**: Una descrizione di un insieme di possibili interazioni tra un ATTORE e un SISTEMA, che produce un risultato che fornisce valore all'attore.

- Gli use cases spostano le prospettive dello sviluppo dei requisiti su COSA GLI UTENTI DEL SISTEMA VOLGONO HANNO BISOGNO DI RAGGIUNGERE
- L'obiettivo degli use cases è di descrivere i compiti che gli utenti necessitano di raggiungere col sistema.

## USE CASE DIAGRAM

- C'è un box che divide ciò che fa parte del sistema e gli attori all'esterno del sistema. Il box ha il TITOLO del progetto.
- Le frecce perte de chi incomincia l'azione / interazione.
- Gli d'uso sono scritti sempre in maniera succinta e al SIMPLE PRESENT.
- Solitamente, ATTORI primari sulle sinistre, ATTORI SECONDARI con le destre.
- **ATTORE**: E' una persona, un sistema software o un dispositivo hardware che interagisce con il nostro sistema!

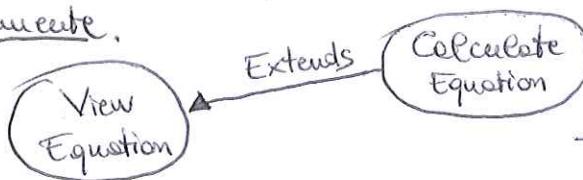
Gli attori definiscono tutto ciò che interagisce con il sistema, ma NON è il sistema.

- **PRIMARIO**: Imitatore di un'interazione; è un attore da cui parte sempre almeno 1 freccia
- **SECONDARIO**: Attore con cui IL SISTEMA HA BISOGNO di interagire; riceve sempre almeno 1 freccia.  
! Attenzione a non confondere attori secondari (esterni al sistema e non controllabili) con hardware/software che fa parte del sistema, dunque non è attore !

### RELAZIONI TRA CASI D'USO:

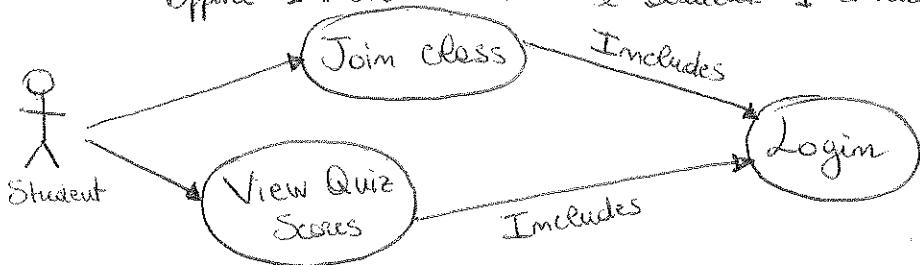
Sono collegati con una frecce che definisce il verso del collegamento + uno stereotipo che definisce il tipo di relazione:

- EXTENDS: il caso d'uso da cui parte le frecce "extends" può essere svolto solo in seguito al caso d'uso cui punte le frecce. Inoltre, vuol dire che il 1° caso d'uso può essere svolto terzate, ma NON necessariamente.



→ OGNI TANTO calcola  
l'equazione - non ogni volta che lo vede.

- **INCLUDES**: indica un caso d'uso NECESSARIO (senza) prima di potere svolgere un altro.  
Tuttavia un caso d'uso che NON ha frecce dirette, ha senso di esistere solo se ha 2 o più frecce "includes" entranti.  
Oppure 1 includes entrate e almeno 1 extends uscente!



\* Direzione freccia: la PUNTA è verso il caso d'uso NECESSARIO, che è anche quello che viene svolto per primo;  
Le frecce PARTE del caso d'uso che HA NECESSITÀ dell'altro prima di poter essere eseguito.

\* ATTENZIONE: Occhio all'esempio di **TICKETONE**; il caso d'uso principale è "Compre Biglietto" e NON "Visualizza Prenzo", che non è un caso d'uso, ma è incluso in "Compre Biglietto".  
Potrebbe essere un caso d'uso (con frecce "includes") se condiviso anche da altri UC!

# SVN and GITHUB

## • VERSION CONTROL SYSTEM (VCS):

E' un sistema che tiene tracce dei cambiamenti applicati ai files; inoltre, permette di tornare indietro ad una versione precedente dei files.

Esistono 2 tipi di VCS:

- CENTRALIZZATO (come SVN):

- I repository condivisi risiedono in 1 UNICO server!
- Gli utenti del repository hanno solo una working copy, che ha bisogno di essere sincronizzata con il server.

- DISTRIBUITO (come GIT):

- Tutto il repository condiviso RISIEDE LOCALMENTE.

\* In realtà, le differenze tra SVN e GIT è minima, e le più delle volte è a scosse dell'utente.

- SVN (Apache SubVersion):

E' il successore del CVS (Concurrent Version System) ed è un VCS centralizzato!

## \* PERCHÉ USARE VCS?

	SVN / GitHub	Dropbox / Google Drive
Synchronization	On demand	Automatic
Merge	Fine grained	Coarse grained
Version History	Yes	Maybe

• REVISIONE: E' ogni STATO possibile del repository. Ha una revisione diversa ogni volta che avviene un cambiamento (commit). Le revisioni sono TRACCiate dal VCS, per poter tornare indietro.

## • CREAZIONE REPOSITORY:

1. Su GitHub, in alto a destra cliccare "+" , "New Repository";
2. Inserire nome del repo, lasciarlo pubblico, aggiungere README per usare SVN;
3. Sul pc, creare una cartella e fare "SVN Checkout", copiando e incollando il link del repository GitHub.
4. Nelle cartelle "Trunk", aggiungere e modificare files; dopo ciò, fare SVN Tortoise > + Add...
5. Andare su Trunk, fare testo destro e fare SVN Commit, inserendo un messaggio.

## • ISSUE TRACKING SYSTEM (ITS):

È un sistema che permette agli sviluppatori di CREARE, ASSEGNARE e TRACCIARE gli ISSUES, anche noti come "TICKETS".

- ISSUE: La traduzione letterale è "problema", ma può essere una delle seguenti cose:

- Una FUNZIONALITÀ da sviluppare;
- Un BUG da risolvere;
- Un compito da completare;
- Qualcosa di simile alle precedenti.

Ogni ISSUE deve avere le seguenti CARATTERISTICHE:

(1) Un NOME e un ID numerico.

(2) Un insieme di CAMPIONI (data di creazione, summary, ...);

(3) Un WORKFLOW del suo STATO: created, assigned, developed, tested, approved;

(4) Un insieme di RESTRIZIONI: chi può creare cose, insieme di regole assegnate agli utenti.

\* Esempi di ITS: Jira, Redmine, GitHub.

\* L'ID degli ISSUE è comodo e professionale che venga usato per riferirsi agli issues nei messaggi utilizzati nei COMMIT. Per esempio: "#1".

Ci si aspetta una relazione 1 ad 1 tra ISSUE e commit!

## • CREARE DELL'ISSUE:

1. Andare su GitHub nel repository di inserimento;

2. Cliccare sulla sezione "Issues" (di default si è in "Code");

3. Cliccare su "New issue";

4. Inserire il titolo e la descrizione (nella sezione "Leave a comment");

sulla destra è possibile assegnarla ad uno sviluppatore e/o mercarle con una label (per esempio, "Bug").

5. Cliccare su "Submit".

\* 6. Si possono anche creare nuove labels.

• VERSIONE: Ha una semantica ambigua; potremmo dire che coincide con REVISIONE.

• RELEASE: È una particolare versione o revisione, che è PROPAGATA ALL'UTENTE;

ha sempre una versione corrispondente,

la release ha un nome "lesco", scelto dallo sviluppatore; invece la versione ha sempre un numero progressivo gestito dal VCS.



## CONTINUOUS INTEGRATION (CI)

Continuous Integration (CI) è una BEST PRACTICE dello sviluppo software che consiste nell'impegno dei membri di un team ad INTEGRARE il loro lavoro FREQUENTEMENTE. Ogni integrazione è verificata tramite una BUILD AUTOMATIZZATA (che include testing) per rilevare errori di integrazione il più velocemente possibile.

Tale best practice RIDUCE ERRORI DI INTEGRAZIONE e consente di sviluppare software PIÙ COESO in maniera più rapida.

### • PRATICHE E REGOLE DI CI:

- (1) Il Repository deve essere UNICO e condiviso;
- (2) Il sistema deve AUTOMATIZZARE le build;
- (3) Fare il self-testing delle proprie build;
- (4) Ognuno fa commit dei cambiamenti OGNI GIORNO, così che si egisca velocemente e le modifiche sono piccole; inoltre ci si può rendere conto subito dei problemi di integrazione;
- (5) Ogni commit dovrebbe comportare una build su una INTEGRATION MACHINE;
- (6) RISOLVERE I PROBLEMI IMMEDIATAMENTE;
- (7) Mantenere le build VELOCE, in modo tale da poter egire subito sui cambiamenti e sugli errori;
- (8) Eseguire i test in un clone del PRODUCTION ENVIRONMENT, cioè in un ambiente che sia il più possibile vicino a quello che utilizzeranno gli utenti; anche il server dovrebbe essere il più vicino possibile al server reale!

## TRAVIS

### • KEYTERMS:

- JOB: Un processo automatizzato che clona il repository in un VIRTUAL ENVIRONMENT ed esegue una serie di PHASES quali: compilare il codice, eseguire test, ... Un job fallisce se il codice di ritorno della fase di script è diverso da 0.
- PHASE: Sono i passi sequenziali di un JOB. Per esempio:  
Install phase → Script phase → Deploy phase ---
- STAGE: È un GRUPPO DI JOBS che eseguono in parallelo come parte di un processo sequenziale di BUILD, che è composto da più stages.
- BUILD: È un GRUPPO DI JOBS (ed eventualmente di stages), Una build termina quando TUTTI i suoi jobs hanno terminato.

## • BROKEN BUILDS:

Una build è considerata BROKEN quando 1 o più dei suoi jobs termina con uno stato che è NOT PASSED e può essere uno dei seguenti:

- **ERRORED:** Un comando nelle fasi `before install`, `install` o `before script` ha ritornato un codice d'uscita diverso da ZERO. Il job termina immediatamente.
- **FAILED:** Un comando nelle **SCRIPT PHASE** ha ritornato un codice di uscita diverso da ZERO. Il job continua ad eseguire fino al completamento.
- **CANCELED:** Un utente CANCELLA il job prima che esso termini.

## • How To:

1. Creare un progetto in Java (Eclipse) ed esportarlo via ANT; ciò dovrebbe produrre un file "`build.xml`" che serve per indicare a Travis come compilare i files.

2. Creare un file chiamato "`.travis.yml`" come il seguente:

```
language: java  
jdk:  
  - openjdk8  
script: mvn build // script:  
      - mvn build
```

3. Creare un repository su GitHub e connetterlo su Travis; creare una working copy (SVN checkout) ed inserire nelle cartelle Trunk i files del progetto più il file "`.travis.yml`".

4. Fare COMMIT e pushare su Travis (.com) nella sezione "Build Status" per verificare se la build fallisce o ha successo.

## THE TECHNICAL DEBT (TD)

- IL TECHNICAL DEBT (TD) è un'applicazione di CONCETTI FINANZIARI al dominio dell'ingegneria del software;

- TD è una METAFORA:

PRO: applicabile in maniera sempile

CONTRO: applicato in maniera sbagliata e inappropriata (diverse volte).

### TERMINOLOGIA:

- QUALITY RULE: Una REGOLA DI QUALITÀ è un principio dell'ingegneria del software, empiricamente validato, che descrive COME IL CODICE DOREBBE ESSERE. Ad esempio, otto densità di commenti, bassa complessità del codice, if-statement brevi, ...
- VIOLATION: Una porzione di codice che NON è COMPLIANT con 1 o più quality rules. Anche chiamate "CODE SMELL" (qualità interna).
- DEFECT: Un problema nel codice che richiede EFFORT per essere risolto. Questi problemi, sotto determinate circostanze, diventano FAILURE (o difese delle Violazioni); una FAILURE è un comportamento del sistema inaspettato, che non è consistente con le specifiche. Un difetto è relativo alle qualità esterne del codice.  
L'autore vede le Failure.

### PERCHE' HO DEBITO TECNICO?

Perché spesso si cerca di ottimizzare gli obiettivi a CORTO-TERMINE (scadute) e ciò porta ad handicapp nel LUNGO-TERMINE. Ad esempio:  
pochi commenti, code smells, codice duplicato, ecc...

- TD può emergere:

- (1) ORGANICAMENTE così come ogni sistema evolve nel tempo, aumentando la complessità;
- (2) SCORSO OPPORTUNISTICALEMENTE, quindi si sceglie (conseguentemente) di voler perseguire un obiettivo a ~~lungo~~ termine, piuttosto che uno a lungo termine.

Per esempio: "Let's release now, we'll deal with it later on".

### CONCETTI CHIAVI:

TD consiste in 2 principali concetti:

- PRINCIPAL: il COSTO necessario per eliminare il DEBITO (costo in risorse)

- INTEREST: le PENALITA' da pagare in futuro se il debito non viene saldato.

Esempio: Un modulo molto complesso potrebbe richiedere un effett significativo per essere ristrutturato (**PRINCIPAL**); comunque, se non ristrutturato, potrebbe rallentare le velocità di sviluppo (**INTEREST**).

## • REFACTORING:

Il termine **REFACTORING** riguarda tutte quelle attività che cambiano il codice, ma non per modificarne le funzionalità, bensì per migliorare le QUALITÀ INTERNA e le leggibilità del codice.

Sostenibilmente, sono le attività effettuate per **RISOLVERE i CODE SMELLS**.

## • TD VS MAINTAINABILITY:

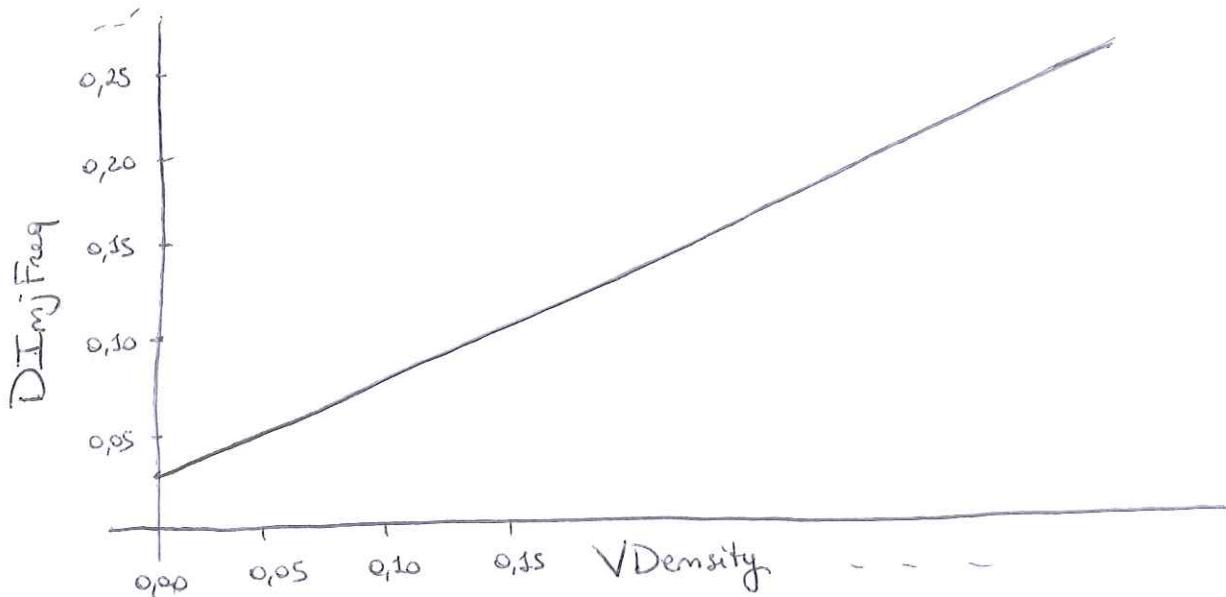
TD è molto vicino al concetto di **MANTENIBILITÀ**; tuttavia, TD aggiunge i criteri di **TEMPO** ed **ECONOMICI** nel prendere decisioni.

Quindi, talvolta è possibile fare scelte a scopo delle manutenibilità del sistema, ma che favoriscono aspetti di BUSINESS, che possono essere prettamente economici o legati al tempo (anticipare una release per battere le concorrenze, ad esempio).

## • CONSIDERAZIONI PRATICHE:

- Il Technical Debt PUO' SCADERE! A differenza del debito finanziario. Infatti, tutti i progetti hanno un **TEMPO DI VITA** e si arriverà ad un punto in cui non esisterà più le "release successive". Inoltre, i contratti potrebbero non includere aspetti di manutenibilità; sarebbe sempre meglio includere dei vincoli di manutenibilità, per esempio nel numero di code smells.
- TUTTI i progetti hanno del TD; in un mondo di risorse finite, l'imperfezione è la norma. Inoltre, il successo di un progetto dipende soprattutto dagli obiettivi di business: costo, time-to-market e soddisfazione degli utenti. Dunque si introduce spesso del TD per venire incontro ai requisiti di business. Ve sempre trovato il giusto COPROMESSO (**TRADE-OFF**).
- Il TD va sempre gestito, monitorato e analizzato, per poi decidere se eliminarlo, quando, come e quanto costerebbe.

## CASO DI STUDIO:



- $V\text{Density} = \text{Violation Density} := \frac{\text{numero di Violazioni}}{\text{LOC}}$ , dove  $\text{LOC} = \text{Lines Of Code}$
- $D\text{InjFreq} = \text{Defect Injection Frequency} := \frac{\text{numero di Difetti inseriti}}{\text{LOC modificate}}$
- ⇒ Più è alto il numero di violazioni (CODE SMELLS), più è alto il numero di difetti che si inseriscono quando si modifica il codice.
- \* Sviluppare il sistema in modo che non abbiate SMELLS e che quindi sia più manutenibile, implica una riduzione di inserimenti di DIFETTI.

## SONAR CLOUD

### VIA TRAVIS:

- 1) Andare su SonarCloud ed eseguire il login via GitHub;
- 2) In alto a destra, premere su "+" ⇒ "Analyze new project" e cerca il repository da monitorare e clicca su "Set up";
- 3) Decidere come gestirlo (in questo caso con Travis CI);
- 4) Cambiare il token premendo sulla "penna", poi se "Generare e token" e inserire qualcosa da tokenizzare. Salvare il token generato in un posto sicuro perché servirà dopo;
- 5) Continuare e utilizzare "Other--"; ora bisogna aggiornare il file ".travis.yml" e generare un file "sonar-project.properties";
- 6) Nel file properties parametrizzato, sostituire Z con il nome dell'account di GitHub, A con il nome del repository ed Y con il token generato in precedenza (senza virgolette). Tale file va inserito nella cartella Trunk.

7) Per il file ".travis.yml" seguire il seguente pattern:

language: java

jdk:

- openjdk8@11

→ deve essere versione 11 o superiore

addons:

sonarcloud:

organization: "<nameAccountGithub>"

token: "<token>"

script:

- mvn build

- sonar-scanner

- \* al posto di <nameAccountGithub> inserire lo stesso di Z per properties;
- al posto di token inserire il token (Y precedentemente). Fare attenzione che a volte lo vuole con le virgolette, altre sì.

Tutto ciò funziona come un TRIGGER: quando viene eseguito un commit GitHub lo segnala a Travis, il quale va a leggere il suo file ".travis.yml" per capire cosa fare e quali comandi eseguire. Quando deve eseguire sonarcloud, si va a leggere il file "sonar-project.properties" per eseguire i comandi giusti, che vanno a scrivere i risultati dell'analisi sul DATABASE di SonarCloud.

## VIA GitHub Actions:

- 1) Andare su SonarCloud e chiedere di creare un nuovo progetto;
  - 2) Scegliere di gestirlo MANUAMENTE;
  - 3) Scegliere SO Windows e Linguaggi "others"
  - 4) Seguire le istruzioni e configurare le variabili di ambiente PATH con la cartella bin dello scanner statico di Sonar, di cui scaricare lo zip ed estrarlo (in un luogo abbastanza sicuro); inoltre aggiungere nuova variabile d'ambiente SONAR\_SCANNER con il valore indicato. (Prospettiva (x86) >> SonarCloud scanner >> bin)  
TOKEN
  - 5) Solvare le istruzioni da eseguire da prompt di comando in un luogo sicuro; inoltre rimuovere i caratteri "\r" e EOL → deve essere tutto su una linea.
  - 6) Per eseguire l'analisi, andare da prompt dei comandi (cmd), entrare nella folder del progetto ( >> cd "<Path>" ) ed eseguire il comando precedentemente solvuto. Per folder si intende quella che include Trunk, NON Trunk!
- \* Alternativamente costruire file properties ed eseguire da cmd comando:  
"sonar-scanner.bat -D<pathFileProperties>" !

## USER INTERFACE PROTOTYPES

Rappresenta un punto di racordo tra dominio del problema e dominio delle soluzioni, in quanto cerca di mostrare all'utente un'INTERFACCIA GRAFICA con la quale interagire col sistema.

- SCOPO: descrivere il MODO in cui l'uomo interagisce col sistema.
- Questo aiuta a guadagnare una REAZIONE DEGLI UTENTI il PRIMA POSSIBILE, potendo così misurare la soddisfazione dell'utente e puo' aiutare l'utente stesso a chiarificare le copie cio' che vorrebbe davvero.
- Il prototipo mostra OGNI finestre, dialogo, menu, bottone, messaggi d'errore, etc... del sistema.
- Annotazioni appropriate descrivono degli aspetti che non sono ovvi.
- Il prototipo deve essere SELF-EXPLANATORY, cioè non deve avere bisogno di spiegazioni aggiuntive.
- CARATTERISTICHE DEL BUON UI Prototype

- 1) VISIBILITÀ DELLO STATO DEL SISTEMA: Gli utenti devono sapere dove si trovano nel sistema e che sta facendo.
- 2) REAL WORLD - SYSTEM MATCH: Usare linguaggi / concetti che sono familiari all'utente. Le icone devono rappresentare le semantica effettive che hanno nel mondo reale. Ordinare i processi e le schermate in modo logico e che abbia significato per l'utente.
- 3) CONTROLLO E LIBERTÀ: NON "INTRAPPOLARE" l'utente. L'utente deve essere sempre in grado di uscire e TORNARE INDIETRO. Non forzare l'utente in une lunghe sequenze lineare di operazioni senza via d'uscita.
- 4) RECOGNITION NOT RECALL: Usare oggetti visivi (ICONE), azioni e opzioni intuiti e di facile intuizione semantica.
- 5) FLESSIBILITÀ ED EFFICIENZA D'USO: "ACCELERATORI" → consentono all'utente di personalizzare il sistema, salvando configurazioni predefinite. Oppure consentire scorciatoie da tastiera.
- 6) ESTETICA E DESIGN MINIMALISTA: Informazioni non strettamente necessarie non dovrebbero essere visibili. Più informazioni ci sono sullo schermo, meno ciascuna informazione è visibile.

f) ONLINE HELP E DOCUMENTAZIONE AGGIUNTIVA: E' bene fornire un manuale d'uso online

8) EFFETTIVA GESTIONE DEGLI ERRORI: Aiutare gli utenti a riconoscere, diagnosticare e risolvere gli errori. Non dire semplicemente che c'è un errore, ma suggerire azioni correttive quando possibile.

9) PREVENZIONE DEGLI ERRORI: Una progettazione che prevede errori è migliore di un buon messaggio d'errore.  
Cercare di evitare che l'utente possa cercare di fare azioni NON fattibili.

10) CONSISTENZA: Il "LOOK AND FEEL" dell'interfaccia deve essere sempre consistente! Stesse colori e posizioni, stesso font, effacements e cose del genere... .

• STORYBOARD: E' una SEQUENZA DI ILLUSTRAZIONI, mostrate appunto in sequenze.  
Describe COME l'utente interagisce col sistema; come il dato viene visualizzato nelle pagine; DOVE sono le informazioni

• WIREFRAME:

E' un prototipo GUI sviluppato per gestire principalmente applicazioni Web;  
sono creati per lo scopo di mettere insieme elementi per raggiungere un obiettivo  
E' maggiormente orientato a COME gli ELEMENTI LAVORANO tra loro e su come il  
DATO viene PROCESSATO nelle pagine e da una pagina all'altra.

# JAVAFX

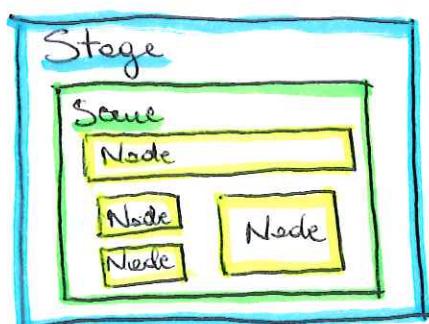
Javafx è un insieme di API grafiche e media packages che consentono di progettare, creare, testare, debuggere e depolare interfacce grafiche di applicazioni consistentemente su diverse piattaforme. Javafx è una LIBRERIA.

## • SCENE GRAPH: E':

- il punto di partenza per costruire un'applicazione Javafx
- un ALBERO GERARCHICO DI NODI, che rappresenta tutti gli elementi visuali della GUI dell'applicazione
- può gestire input e può essere renderizzato

## • NODO: Un singolo elemento in uno Scene graph è chiamato NODO; Ogni nodo ha un ID, una ~~class~~ style class e dei controlli del suo volume.

- Ad eccezione del nodo radice, tutti i nodi hanno ESATTAMENTE 1 NODO GENITORE, mentre possono avere 0 o più figli.
- Un nodo può avere:
  - effetti ( blur, shadow)
  - opacità
  - Transform
  - uno specifico stato dell'applicazione
  - EVENT HANDLER ( mouse, key, input method, ... )



## ACTIVITY DIAGRAM

### • Overview:

- ANALISI DEI REQUISITI: determina cose gli utenti vogliono e ciò di cui hanno bisogno del sistema software.
- SPECIFICHE: definiscono formalmente i requisiti utente.
- PROGETTAZIONE: definisce e organizza i principi COMPONENTI OPERAZIONALI del sistema.
- IMPLEMENTAZIONE: definisce i dettagli operazionali in un linguaggio di programmazione.

### • OBIETTIVI DELLA PROGETTAZIONE:

- 1) TRACCIABILITÀ: elementi della progettazione devono essere tracciati all'indietro, corrispondendo ad elementi delle specifiche.
- 2) MODULARITÀ: gli elementi della progettazione devono essere organizzati in MODULI logicamente COESI !
- 3) PORTABILITÀ: la progettazione deve essere obbligatoriamente generale, tale da non essere legate ad una specifica piattaforma o ad uno specifico linguaggio di programmazione.
- 4) MANTENIBILITÀ: il sistema deve essere progettato in modo tale che possa essere facilmente ripetuto e cambiato → LOW COUPLING, HIGH COHESION e concetto di TECHNICAL DEBT !
- 5) RISUSABILITÀ: quando opportuno, i moduli devono essere progettati così da poter essere riutilizzati e riedetti in futuro

### • ACTIVITY DIAGRAM:

Un'ATTIVITÀ è un INSIEME di AZIONI (boxes), definite IN SEQUENZA dal punto di vista dell'utente.

\* L'Activity Diagram modella il Flusso (diagramma dinamico / comportamentale) di un'ATTIVITÀ, che è determinato dello STATO DEL SISTEMA e dalle azioni dell'UTENTE.

- Di solito, si ha corrispondenza 1 ad 1 tra caso d'uso e activity diagram.  
Dunque:
  - ATTIVITÀ: è UNICA e spesso coincide con il caso d'uso!
  - AZIONI: sono multiple e rappresentano i passi necessari per svolgere l'attività.

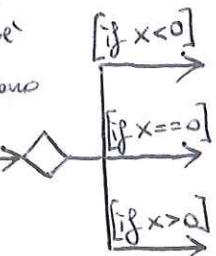
• SIMBOLI:

- DIAMOND: → Può indicare una DECISIONE o un MERGE:

- DECISIONE: C'è 1 freccia entrante e PIÙ frecce uscenti.

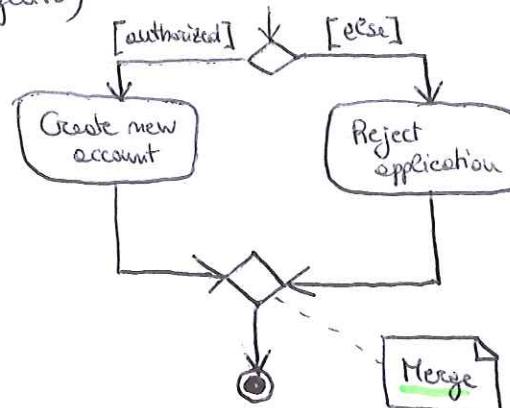
• Sulle frecce uscenti ci sono delle GUARDIE, cioè degli if-statement. Le condizioni delle guardie devono essere COMPLETE ed ESCLUSIVE!

\* È bene usare "else" come guardie.

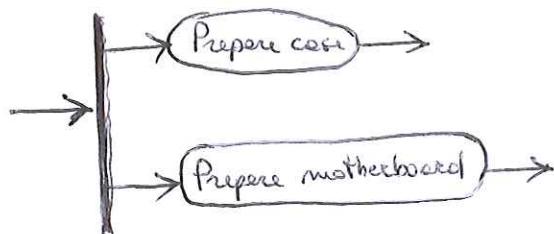


- MERGE: deve esserci 1 sola freccia uscente e PIÙ frecce entranti.

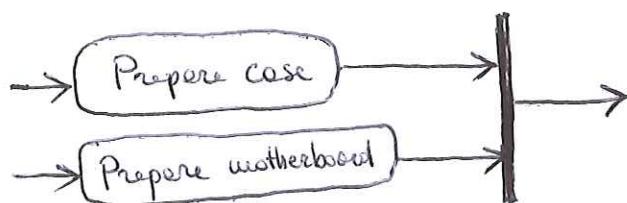
Il flusso nel merge è portato SOLO de 1 delle FRECCE ENTRANTI (si fa il merge quando 1 dei precedenti flussi viene eseguito)



- FORKS: Ha sempre 1 freccia entrante e PIÙ frecce uscenti. Le azioni sono SVOLTE IN PARALLELO → implementabili tramite THREADS.



- JOIN: Si usa sempre in seguito ad una FORK. ha PIÙ frecce entranti e 1 sola freccia uscente, ma, a differenza del MERGE, ASPETTA che arrivino TUTTI I FLUSSI ENTRANTI per poter eseguire il flusso uscente!



\* L'ordine delle attività svolte in parallelo NON IMPORTA! Però devono essere concluse ENTRAMBE per poter andare avanti.

- TIME EVENTS: Rappresentano eventi legati all'attesa di TEMPO, come timeout/timer. Sono però relativi solo al tempo, non ad un contesto rispetto a qualcosa! Wait 3 days



- Se le azioni diventano troppe e troppo complesse, le si possono raggruppare in un'unica azione, contrassegnarle con un asterisco e trasformarle in una ATTIVITÀ, che avrà il suo Activity Diagram a parte.
- Le AZIONI dell'Activity Diagram devono essere viste dal punto di vista del SISTEMA! Gli utenti / lettori sono gestiti quasi unicamente tramite GUARDIE.
- Ogni activity diagram ha esattamente 1 pellino iniziale (pieno) e 1 pellino finale (non completamente pieno).

# HTML

- HTML: HyperText Markup Language → È un linguaggio di MARKUP! Non di programmazione.

MARKUP → Utilizzo di marcatori per aggiungere informazioni (metadati) ad un documento, in modo che sia trattabile in modo automatico. Per esempio:

- Layout (struttura del documento)
- Componenti logici

Vediamo in dettaglio degli elementi di HTML.

## • TAG:

Sono elementi strutturati in maniera **GERARCHICA** secondo una sintassi (schema).

Rappresentano il vero e proprio codice HTML.

- Un tag è racchiuso tra i simboli < >. Generalmente vanno in coppie: tag di apertura + tag di chiusura:

<b> questo testo va in gressetto </b>.

## • ATTRIBUTO:

Un attributo è una caratteristica / proprietà associata ad un tag.

- È formato da coppie (name, valore) : name = "name".

## • TAG DI DOCUMENTO:

- <html> [...] </html> → racchiudono l'intero documento html
- <head> [...] </head> → racchiudono l'intestazione; contengono info NON visualizzate dal browser.
- <title> [...] </title> → Testo che appare come titolo del Browser, in alto a sinistra
- <body> [...] </body> → Tutto il CONTENUTO visualizzato nelle finestre del browser!

\* I tag <head>, <title> e <body> sono tutti **FIGLI** del tag <html>, cioè vi sono contenuti dentro!

\* <title> DEVE essere **FIGLIO** di <head>!

## • TAG DI STRUTTURA

- `<h1>, ..., <h6>` → Intestazioni (con dimensioni variabili); "header".
- `<br>` → Si va a capo e NON ha bisogno di chiusura (cioè il tag `</br>` NON ESISTE!).
- `<p> [...] </p>` → Paragrafo; inoltre va a capo e lascia una linea bianca alla fine.
- `<ol> [...] </ol>` → Ordered List: liste ordinate di elementi; tramite attributi si puo' indicare il tipo di enumerazione
- `<ul> [...] </ul>` → Unordered List
  - `<li> [...] </li>` → List Item: racchiude un elemento di una lista.  
DEVE essere FIGLIO di `<ol>` o di `<ul>`!

## • TAG DI STILE:

`<i>` : italic (corsivo)

`<b>` : bold (grassetto)

`<font>`: permette di specificare il carattere tramite attributo

`<u>` : underlined (sottolineato)

`<center>`: centra il testo rispetto alla pagina

• `<em>`: emphasized (enfatizzato) → di default e' corsivo, ma dipende dal browser

• `<strong>`: in rilievo → di default, grassetto

\* L'uso di questi tag è DEPRECATO!

\* Si consiglia l'utilizzo di fogli di stile CSS !

## • TAG DI ANCORA:

• `<a> [...] </a>` → Utilizzati per creare HYPYERLINKS ad altri documenti o ad una posizione nello stesso file html.

\* Ha l'attributo **Href** (Hyperlink REference) che ha come valore la destinazione del link!

`<a href = "http://www.google.com"> "Clicca qui" </a>`

## • TABELLE:

Una tabella è rappresentata come una SEQUENZA DI RIGHE.

- <table> → per definire una tabella
- <tr> → table row ; inserisce una nuova riga
- <td> → inserisce una colonna all'interno di una riga
- <th> → table header ; è una riga speciale : il testo di ogni colonna di <th> sarà centrato e in grassetto!
- <caption> → consente di inserire una didascalia per la tabella.

## • FORM:

- <form> [...] </form> → permette di costruire un form
- <input> → permette all'utente di interrogare con le pagine , costruendo le richieste da inviare al server.
  - \* Ha come ATTRIBUTO il TIPO, che permette di scegliere tra:
    - Testo
    - Spunte
    - Selezione singola
    - Selezione multiple
    - Bottoni
    - Selettore di file

## • INPUT: TEXT

- "text" → <input type = "text" value = "Sample Text" size = "20" maxLength = "40">  
Testo con valore di default "Sample Text" di dimensione 20 cher in cui è possibile inserire al massimo 40 cher.
- "password" → <input type = "password" name = "pwd">  
Testo non visibile

## • INPUT: Checkbox e Radio Button

- "checkbox" → <input type = "checkbox" name = "name" checked = "true">  
↓  
Valore di default
- "radio" → Insieme di caselle selezionabili , ma MUTUAMENTE ESCLUSIVE  
<input type = "radio" value = "Male" name = "Group1">
- \* 2 Radio Button appartenenti allo stesso gruppo solo se HANNO LO STESSO NAME !

## • INPUT: SELEZIONE SINGOLA

Un MENU' A TENDINA da cui si puo' scegliere una sola opzione. Le varie opzioni sono definite tramite il tag `<option>` + attributo "value":

`<select>`

```
<option value = "Mele"> Mele </option>
<option value = "Pere"> Pere </option>
<option value = "Banane"> Banane </option>
```

`</select>`

## • INPUT: SELEZIONE MULTIPLO

Un MENU' A TENDINA a SELEZIONE MULTIPLO (CTRL + click); con l'attributo "size" si indica il numero di elementi visibili senza scroll:

`<select multiple size = "5">`

```
<option value = "Mele"> Mele </option>
<option value = "Pere"> Pere </option>
```

---

`</select>`

## • INVIO DI DATI AL SERVER

Si puo' far uso di bottoni speciali:

- Type = "reset" → pulisce i valori all'interno del form
- Type = "submit" → Sottomette i dati del form, li invia al Server

Chi e' il SERVER?

Il server ve specificato all'interno del tag `<form>` con l'attributo ACTION:

`<form action = "http://www.nomedelserver.it" method = "get">`

Come vengono inviati li dati?

Con il protocollo HTTP → Il browser costruisce un messaggio HTTP in base ai parametri del form e al method.

I method possibili sono quelli previsti da HTTP:

- GET
- POST
- PUT
- DELETE
- (- HEAD)

## • ALTRO:

- ATTRIBUTI per une tabelle:

<table border="1" width="500px">

- <hr> → HORIZONTAL LINE: disegna une linee orizzontale; NON ha bisogno di chiusure

- Posciore uno spazio vuoto (in questo caso una colonna):

<td> &nbsp; </td> ; "nbsp" ste per non breaking space!

- Uso dell' AMPERSAND (&) come CARATTERE DI ESCAPE:

- &egrave; → è

L' & serve per indicare che dopo c'è il codice, che deve terminare con ";".

Serve per non incappare in problemi dovuti alla codifica.

- CREARE RIFERIMENTO:

<a mouse="InizioDocumento"> </a> → Chiudiamo immediatamente, stendendo un NONE!

Per richiamarlo: <a href="#InizioDocumento"> o capo </a>

→ Come valore dell' attributo href si mette il NONE DELL' ANCORA!



## JSP (Java Server Pages)

JSP è una tecnologia per costruire pagine Web DINAMICHE.

- I files JSP mischiano codice HTML con codice Java;
    - le pagine HTML hanno embedded dei pezzi di codice Java con tag e costrutti speciali
  - JSP è una tecnologia SERVER SIDE (a differenza di JavaScript, lato client):
    - Le risorse JSP non sono elaborate sul client (browser).
    - Le risorse JSP sono deployed lato server ed elaborate dal JSP container, un componente dell'APPLICATION SERVER (e.g. Tomcat, ...).
- \* L'interleccianento tra HTML e Java, per poter essere eseguito dalla JVM, deve essere trasformato solo in Java e tradotto (compilato) in eseguibile (bytecode). Ciò viene fatto nel Server automaticamente tramite SERVLET!

### ESEMPIO:

- <% [...] %> → TAG per inserire del jsp!
- <%@ page contentType = "text/html; charset=UTF-8" language = "java" %>  
→ Serve per istruire l'APPLICATION SERVER sul contenuto del file jsp!

### DIRETTIVE:

- 1) useBean → serve per dichiarare e istanziare VARIABILI; si usa così:  
`<jsp:useBean id = "<name>" scope = "request" class = "<JavaClassFullName>"/>`

- \* Lo Scope è il livello di persistenza delle variabili; può essere di tipo:
- request → variabile locale (dura fino al refresh della pagina)
  - session → variabile quasi globale, accessibile da ogni pagina per 1 UTENTE
  - application → variabile globale, le stesse per TUTTI GLI UTENTI

- 2) setProperty → serve per mappare gli attributi di un oggetto sui campi della form

`<jsp:setProperty name = "<variabile>" property = "*"/>`

Voglio mappare TUTTI gli attributi per nome!

- 3) Per importare file.css: Nell'<head> della parte html, inserire:

`<link href = "css/style.css" rel = "stylesheet">`

#### 4) COME FAR ELABORARE LE RICHIESTE AL SERVER?

<%

if (request.getParameter("Login") != null) { // Se il bottone (submit) chiamato "Login" e' stato premuto

if (LoginBean.Validate()) { %>

<jsp:forward page="Riassumologin.jsp" /> → Direttive per inviare ad un'altra pagina

<%

} else {

%>

<p style="color: red"> Dati errati </p> → Stampa un paragraf

<%

}

%>

}