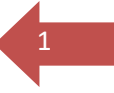




MTSS Semplice (per davvero)

Rovesti Gabriel

Attenzione



Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.

Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione ed Issue Tracking System/ITS.....	3
VCS, tipi di VCS e workflow patterns	6
Laboratorio 1: GitHub come strumento di ITS	8
Git: caratteristiche e confronto con SVN (Git vs SVN).....	9
Laboratorio 2: Git Workflow e GitHub ITS.....	11
Visione degli altri workflow: GitFlow e Fork Workflow; inizio framework SCRUM.....	12
SCRUM: disamina completa	14
Software Testing.....	20
Laboratorio 3 - Maven	23
Test e Unit testing	24
Laboratorio 4: Junit	27
Fine Laboratorio 4 e Analisi Statica	28
Continuous Integration/CI.....	31
Laboratorio 5: GitHub Actions	33
Artifact repository	35
Laboratorio 6 – Jenkins.....	37
Continuous Delivery (CD)	37
Configuration Management (CM)/CM DevOps.....	41
Container e Docker	44
Laboratorio 7: Ansible	47
Caso d’uso per riferimento di Assignment 3: ASPI/Autostrade per l’Italia	48
Caso d’uso: Chili e ItsArt	49
Robot Framework e Laboratorio 8: Robot Framework	50
Esempio di test con Docker-Selenium e Robot Framework	52



Introduzione ed Issue Tracking System/ITS

Il corso tratta principalmente la condivisione e creazione di software gestito in maniera condivisa. In questo senso, il programma deve essere controllato nel suo codice sorgente e poi integrato attraverso la fase di build. Lo sviluppo avviene attraverso una modalità di integrazione continua, al fine di verificare se sia effettivamente funzionante in vari ambienti di sviluppo.

In linea di massima si segue il principio della *continuous delivery pipeline*, che è un'implementazione del paradigma continuo, in cui build, test e distribuzioni automatizzati sono orchestrati come un flusso di lavoro di rilascio.

Tutto ciò fa parte della creazione Agile dei progetti, al fine di gestire vari processi tradizionali e fornire molteplici funzionalità. Tutto ciò viene garantito attraverso vari principi, prendendo ad esempio l'esplorazione continua dei bisogni del mercato/clienti attraverso feedback, integrazione continua di caratteristiche, creazione e consegna di fasi di produzione e rilascio a richiesta, rendendo il prodotto disponibile una volta correttamente pronto.

Gli esempi più classici di ITS sono:

- il bugtracker di Ubuntu, con diverse migliaia di bug giornalmente segnalati
- *Jira*, software molto utilizzato, dove si riportano le segnalazioni per ogni singolo progetto all'interno di un server centrale ospitante, con dei menù e a destra la versione del software in cui un certo bug è stato sistemato.
- GitHub presenta una serie di tag, labels ed assegnatari, mettendo nero su bianco quante più cose possibili, utile per tutti gli attori di un progetto software.

Questi sistemi mantengono una lista di problemi/criticità (issues), permettendo di trovare o localizzare una serie di problemi (*tracking*), lasciando una "traccia" di come siano stati risolti e di tutte le segnalazioni accolte.

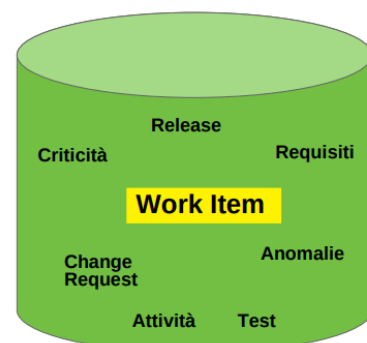
Definiamo ora formalmente un *ITS/Issue Tracking System*, è un pacchetto software che gestisce e mantiene una lista di problemi/issues, come richiesto da un'organizzazione. Essi sono generalmente usati in ambienti collaborativi e usati nella registrazione di un supporto ai clienti per creare, aggiornare e risolvere problemi legati ai clienti, aggiornando e risolvendo segnalazioni presenti nell'organizzazione.



Facilita quindi la gestione del processo di sviluppo e di change management attraverso la gestione di attività diverse (Work item) come: *analisi requisiti, sviluppo, test, bug, release, deploy, change request...*, tutto utile per costruire un buon ALM/Application Lifecycle Management, basato sui principi dell'immagine a lato. Ogni singola "attività minima" (Work item) del progetto è gestita mediante un workflow e mantenuta all'interno di un'unica piattaforma e di un unico repository.

Work Item sono degli strumenti utili per capire come sviluppare e lavorare su un progetto, magari dal punto di vista del cliente facendo capire le sue esigenze e come usarlo. Si pongono quindi come strumenti di confronto tra il team di sviluppo e il project manager che lo individua.

Essi servono a condividere le informazioni con il team di sviluppo, il Project Manager e il cliente (SAL), tramite una repo su cui reperire tutte le info utili.

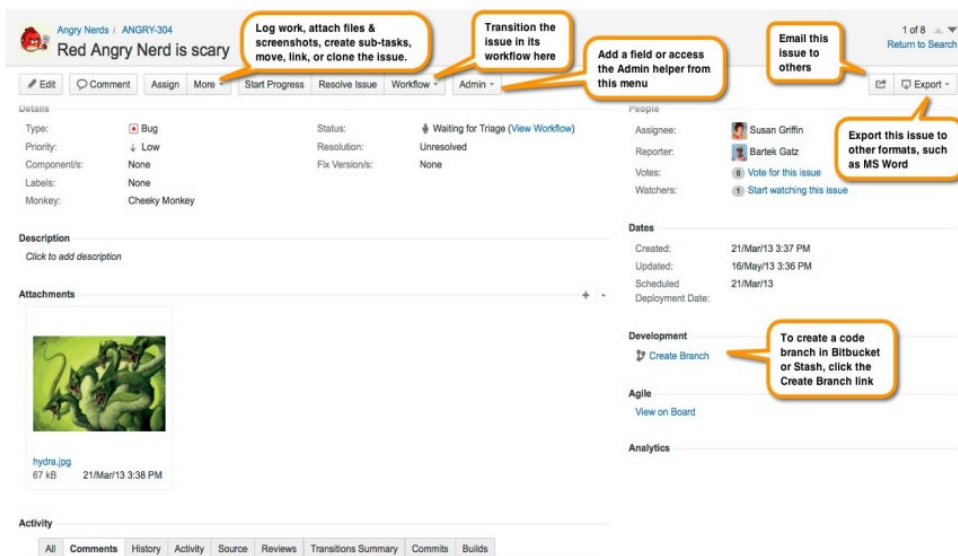


Permettono anche di implementare un processo per misurare la qualità del progetto, avendone un'istantanea e capendo quando e cosa rilasciare, assegnando una proprietà alle attività, quantificando per ciascuna un tempo ed una responsabilità. Per ognuna si traccia una memoria storica, che segue tutti i cambiamenti del progetto. I più usati sono Jira, Github, Gitlab, BugZilla, Trello, ecc.

Il Project Manager all'interno di un ITS:

- 1) crea un nuovo progetto
- 2) definisce il processo da seguire (Tipi di work item, campi custom, work flow, collegamenti)
- 3) aggiunge gli utenti assegnando ruoli e permessi.

Buona comparazione dei vari ITS al link: https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems . Vediamo le caratteristiche di un Work Item in Jira:



Ciascun work item possiede come campi (lista della spesa completa):

- Progetto: progetto a cui si riferisce
- Codice: identificativo univoco
- Riepilogo: descrizione breve dell'attività
- Descrizione: descrizione esaustiva dell'attività
- Tipo: Categoria del Work item p.es: "Task", "Epic", "Bug", "Requisito", "Test execution". Ne determina i campi, gli stati, le schermate e il workflow (ciclo di vita)
- Stato: lo stato all'interno del workflow in cui si trova il work item
- Priorità: Importanza del work item in relazione con gli altri work item del progetto
- Stato di Risoluzione: identifica lo stato di risoluzione del work item p.es. Chiuso, duplicato, work for me
- Versione di riferimento: versione del progetto in cui è richiesta l'attività
- Componente: Componente del progetto a cui si riferisce il Work Item
- Etichette: permettono di classificare i Work Item anche di diversi tipi. Ne facilitano il raggruppamento e la ricerca
- Collegamenti: permettono di collegare tra di loro i Work Item
- Assegnatario: identifica chi è il responsabile per svolgere l'attività
- Segnalante: identifica chi ha segnalato l'attività
- Data di creazione
- Data di ultimo aggiornamento
- Data di risoluzione
- Stima originaria: stima per lo svolgimento dell'attività
- Stima a finire: stima presunta per terminare l'attività
- Tempo speso
- Allegati

Per portare a compimento le attività, l'insieme di stati e transizioni dei Work Item e del loro tempo di vita è il Workflow, insieme di stati e transizioni di un Work item attraversa durante il suo ciclo di vita. Un Workflow viene associato ad un Progetto e può essere associato ad uno o più Tipi, registrando tutte le transizioni e cambi di stato.

Determinate esigenze del cliente sono documentate nei *SLA (Service Level Agreement)*, per esempio la risoluzione di una certa problematica X entro un certo tempo Y o le caratteristiche che il provider si impegna a voler rispettare. Ciascun requisito può essere categorizzato e classificato in *macrorequisito*, *requisito* e *sottorequisito*, suddividendo e specificando l'individuazione e l'utilizzo delle sottoattività. Importanti in questo senso sono i *collegamenti*, definendo le relazioni tra i Work item, solitamente bidirezionali e utilizzati come possibile criterio di ricerca. Questo permette di verificare la presenza o meno di relazione tra i Work Item (p.es. Il requisito è coperto da casi di test).

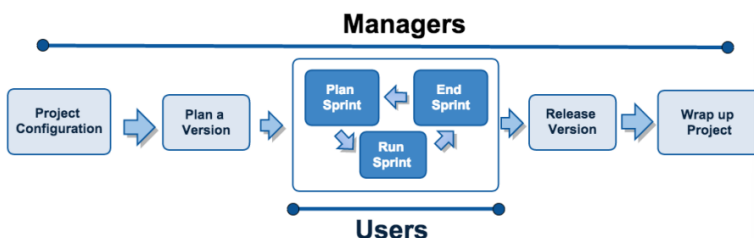
Un ITS offre varie funzionalità, ad esempio la ricerca avanzata dei Work item, salvataggio di ricerche, esportazione, le notifiche (utili a segnalare agli attori l'andamento del progetto, le Bacheche o Boards (che definiscono le attività e la loro organizzazione), Reporting, Dashboard (stato generale), definizione di Road map (fronte strategico da seguire) e Release Notes (note di rilascio), integrazione con il Source code management (gestione diretta del sorgente) ed Integrazione con l'ambiente di sviluppo.

Alle attività vengono possibilmente applicati *filtri*, per capire lo stato (in progress, done, to do, etc.) e permettono di ricercare i work item in base ai campi. I filtri possono essere salvati per facilitare le ricerche più frequenti (p.es. i work items attivi) e i loro risultati possono essere esportati per creare report, board e dashboard.

Le *board/bacheche* permettono di visualizzare i work item di uno o più progetti, offrendo un modo flessibile e interattivo di gestire e visualizzare dei dati di sintesi sulle attività in corso. Esse vengono configurate e visualizzano i work item ricercati con un filtro, interagendovi velocemente. Strumenti di *report* grafici sono ottenibili tramite un ITS. L'utilizzo e la configurazione di un ITS richiedono di:

- 1) Identificare i processi richiesti per la gestione del progetto:
 - Procedure e best practices definiti dai framework di qualità presenti in azienda o richiesti dal Cliente (p.es. CMMI, ISO 9001, ITIL, ...)
 - Vincoli imposti dal cliente (SLA, informazioni richieste da contratto, ...)
 - Modalità di gestione del progetto del Team (Agile Scrum, Agile Kanban, metodologie utili di sviluppo, ecc.)
- 2) Identificare e configurare gli strumenti che permettono di implementare i processi (ITS)
 - Identificazione e definizione dei tipi, dei campi custom, dei work flow e dei collegamenti che ci permettono di tracciare le informazioni richieste dal process

Nel caso d'uso di Jira:



Il Manager (amministratore) dell'ITS:

- Crea un nuovo progetto e definisce le versioni (release)
- Definisce il processo da seguire/lavoro da svolgere (backlog):
 - Tipi di work item, campi custom, workflow, collegamenti
 - Seleziona il modello di stima
 - Differenti Board e Report per processo

- Definisce le componenti del progetto
- Aggiunge gli utenti e assegnazione ruoli/permessi
- Definisce la prima iterazione

Gli utenti (il Team di sviluppo):

- Ricevono le notifiche dei work item assegnati
- Selezionano i work item in base alle priorità
- Avviano e completano la lavorazione
 - Avanzano gli stati del workflow
 - Aggiornano la stima a finire
 - Registrano il tempo impiegato
- Documentano lo stato dell'attività (commenti) e compilano i campi nel work item
- Completano tutte le attività presenti nell'iterazione
- Effettuano il rilascio

Il Manager (e/o il Capo progetto):

- Monitora l'avanzamento e il completamento delle attività (filtri, board, Dashboard, Report)
- Definisce le nuove versioni
- Definisce le nuove iterazioni
- Definisce e aggiorna e monitora le attività (priorità, verifica stima/consuntivo)
- Produce i report richiesti dal cliente (p.es. Calcolo SLA, release log, qualità delle versioni o dei componenti...)

I benefici nell'utilizzo di un ITS sono:

- Implementare un processo e verificarne l'adozione
- Migliorare e misurare la qualità del progetto
- Misurare e aumentare la soddisfazione del cliente
- Migliorare la definizione delle responsabilità
- Migliorare la comunicazione nel team di sviluppo e con il cliente
- Aumentare la produttività del team di sviluppo
- Gestione del tempo e della produttività personale
- Ridurre le spese e gli sprechi
- No mail company

VCS, tipi di VCS e workflow patterns

La repository di deposito del codice sorgente e conseguente evoluzione di un progetto è cosiddetto Version Control System (VCS) gestendo i cambiamenti di documentazione o codice sorgente (ciascuno identificato da un numero o una lettera, che identifica il numero di revisione). Ogni revisione ha un timestamp di riferimento e può essere confrontata con le altre tramite *rollback*, tornando ad una versione precedente, oppure svolgere una attività di *merge* tra versioni presenti, quindi unendo le modifiche presenti. Essi vengono anche definiti come SCM/Source Code Management systems (SCM) e sono dei sistemi software che registrano tutte le modifiche avvenute ad un insieme di file, permettono la condivisione di file e modifiche ed offrono funzionalità di merging e tracciamento delle modifiche. In un team di sviluppo permettono di *collaborare* in modo efficiente, individuando e risolvendo facilmente i conflitti e facilitando la *condivisione* di commenti e documentazione.

Ogni modifica quindi possiede una sua storia completa ed è facile fare un eventuale ripristino ad una versione precedente. Il lavoro viene svolto senza interferenze tra i differenti rami/branch di sviluppo e le modifiche possono poi confluire sul ramo principale (main/master), tale che ogni attività sia registrata dentro l'ITS. Ne esistono di tre tipologie:

- 1) VCS locali/Local VCS; quindi, tra directory dello stesso PC; è l'idea più semplice ma anche la più probabile ad errore. Registrano solo la storia dei cambiamenti e utilizzano un database (*Version Database*) che registra tutta la storia dei file, salvando sul disco una serie di patch e ricreando lo stato di tutti i file in qualsiasi momento. Essi non gestiscono la condivisione.
- 2) Centralized SCM (CVCS), meno vecchi e molto diffusi e gestiscono sia la condivisione che il tracciamento della storia. Qui il version database si trova in un server centrale (unico punto di rottura) e lo sviluppatore è un client che ha nel suo spazio di lavoro solo una versione del codice alla volta. Sono facili da apprendere (es. SVN)
- 3) Distributed VCS/DVCS è distribuito per duplicazione in ogni nodo; quando il nodo centrale non è disponibile, è possibile continuare a lavorare e registrare i cambiamenti. La fase di registrazione avviene attraverso commit e singole push, risolvendo i singoli conflitti anche grazie all'impostazione di diversi tipi di flussi di lavoro che non sono possibili in sistemi centralizzati. L'apprendimento è più complesso rispetto ai CVCS (ad es. Mercurial)

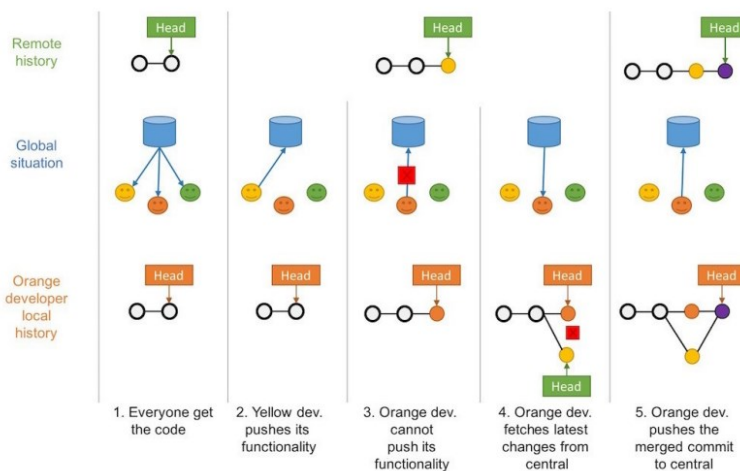
Strumenti più interessanti e presenti oggi sono i Cloud-Based SCM, che sono dei "VCS As a Service": il Version Database è gestito in un servizio cloud, delegando così la gestione ad un servizio esterno, al di fuori dell'infrastruttura aziendale (es. GitHub, BitBucket, Sourceforge, ecc.)

Parliamo quindi di terminologia:

- *commit*, quindi cambiamenti sul database, nuove versioni e differenze varie. Questi permettono di versione capire se una sia più aggiornata o meno. Essi possono essere locali o remoti e l'ultima commit cronologicamente è la HEAD.
- *branch*, puntatore verso un commit specifico, salvando uno *snapshot*, quindi un'istantanea della situazione fino a quel momento. Per integrare un branch si deve eseguire un'operazione di merge.
- *pull request*, un modo di gestire branch e di unire questi al ramo master. Essa può richiedere di inoltrare le modifiche sul server centrale. Prima di fare il merge si ha la possibilità di rivedere le modifiche apportate; a quel punto viene poi eseguita la modifica, eseguendo il merge nel branch di destinazione.

Ora vediamo i singoli workflow patterns:

- 1) centralized, pattern naturale per SVN o CVS ed è facile da capire e da usare. La collaborazione si ferma quando il server centrale è in down o la cronologia è corrotta. L'esempio grafico segue:



- 2) feature branch, inserendo un branch per ogni caratteristica. Qui è ancora più facile tracciare i conflitti, eseguendo merge o risolvendoli.

- Un esempio esplicativo è quello del polipo con gli occhiali e col cappello:
 - su un ramo viene sviluppato la caratteristica “occhiali”
 - su un altro ramo viene sviluppata la caratteristica “cappello”
 - sul ramo principale si esegue un merge, ottenendo un polipo con gli occhiali e col cappello
- 3) GitHub Flow, pattern semplice e collaborativo per implementazione di branch, merge, pull request offerto già graficamente da GitHub nel mentre si sviluppa un progetto similmente a questo ci sta anche la GitLab Flow che implementa questo principio su GitLab (piattaforma simile a GitHub) e permette di avere grafici per verificare lo stato di avanzamento del progetto
 - 4) Gitflow, estendendo le funzionalità del Feature Branch Workflow, operante come target a larga scala. Dal ramo master stacciamo il ramo di *develop*, ramo di sviluppo e gestione del progetto. Ad esso, parallelamente vengono implementati dei *release branches*, qualora si voglia ideare un rilascio
 - 5) fork, pattern ereditato da GitHub e utilizzato spesso nei progetti open source. Il fork intende normalmente la copia di una repository all’interno del proprio spazio GitHub per potervi lavorare come si avesse “fisicamente” la stessa repo. Le modifiche fatte rimangono in mano all’utente. Nel caso d’uso del pattern, però, il fork si ha sulla cartella principale e si offre la possibilità di proporre delle pull request, in maniera decentralizzata per i vari pattern

Un confronto utile tra CVCS/DVCS (si intende per lock il fatto di non poter eseguire modifiche):

CVCS

- + L’apprendimento è più semplice
- Meno recenti
- *Single point of failure*. Se si corrompe il server centrale si perde tutta la storia
- + Permettono il lock dei file
- Favoriscono l’utilizzo del centralized work flow. Non è possibile adottare tutti i work flow precedentemente descritti
- Le operazioni di commit sono più lente. Non possono essere fatte off-line

DVCS

- L’apprendimento è più difficile
- + Più recenti *Standard de facto*
- + Architettura distribuita per replica. Meno possibilità di perdere tutta la storia
- Non permettono il lock dei file
- + È possibile adottare più tipi di work flow
- + Le operazioni di commit sono più veloci perché avvengono in locale e possono avvenire anche off-line

Laboratorio 1: GitHub come strumento di ITS

In generale GitHub risulta essere uno strumento valido di versionamento e di tracking system, offrendo funzionalità di collaborazione e gli account free riescono ad avere collaboratori illimitati e repository con un certo numero di azioni (2000 azioni al minuto al mese). Permette facilmente la suddivisione in compiti, implementazione di singole caratteristiche ed offerta di servizi base in modo gratuito; similmente, i profili presenti sono liberi di ospitare progetti e avere collaboratori illimitatamente.

Il fork è permesso pubblicamente qualora permesso e le cartelle sono pubbliche.

Alle università (anche la nostra) viene offerto il GitHub Student Developer Pack (per avere le funzionalità PRO integrate: <https://education.github.com/pack>).

Ci occupiamo di aprire una *segnalazione* (tramite la sezione *Issue*, cliccando su *New*). Eventualmente ad una segnalazione è possibile associare un commento, nel caso chiudendo anche la issue tramite il commento. È possibile sottoscrivere alle singole segnalazioni, verificando se la segnalazione viene risolta o meno. Ad una segnalazione è possibile associare delle etichette (*labels*, e.g. Bug, Documentation, Enhancement, etc.) oppure crearne facilmente di personalizzate.

Altra sezione importante sono le *Milestones*, equivalente delle versioni. All’atto della creazione, oltre al titolo, si ha anche una data entro cui si intende rilasciarla (*due date*). Ad ogni milestone è associata una percentuale di completamento.

È inoltre possibile associare issues alle milestones, sapendo l'ordine delle attività e organizzando al meglio il lavoro.



Andando nella repository e cliccando *Settings*, nella sezione *General*, scorrendo verso il basso, si ha *Set up templates*, di cui ci sono alcuni esempi come *Bug Report* (contenente già un formato della segnalazione dell'utente, come degli screen, desktop, screen, etc.). i template vengono aggiunti proprio sotto forma di commit, cliccando su *Propose Changes* e poi su *Commit Changes*.

A questo punto viene creata nella repo una cartella chiamata *Issue Templates*, contenente dei file .md (markdown) che possiede i template precedenti, poi eventualmente editandoli e customizzandoli.

A questo punto cliccando su *New Issue*, si vede che si facilita la compilazione della issue, avendo le categorie scelte dal template. Eventualmente si può creare anche un form, magari con un template custom, avendo ancora più campi e rendendo quindi la segnalazione agevola e anche esteticamente carina.

Andando a *Close with comment*, la segnalazione può essere chiusa (passerà in stato *Closed*). Naturalmente le varie segnalazioni possono essere *filtrate*, ad esempio a seconda dello stato.

Tutto questo fa parte del workflow, composto di vari stati (*Open/Closed/Reopen*, l'ultima nel caso si riapra la segnalazione).

La parte di notifiche viene gestita come prima, entrando sulla segnalazione e cliccando *Subscribe*. Nella repo, cliccando su *Settings* e poi *Collaborators*, posso aggiungere delle persone che collaborano ad un certo progetto con *Add People*. A quel punto l'altra persona riceve una notifica con cui deciderà o meno di accettare la collaborazione.

Per poter conoscere lo stato di un progetto, si ha la sezione *Board*, all'interno della sezione *Projects*. Noi vedremo le board di tipo *Agile*.

Creato il progetto, con un template, un nome ed una descrizione, si possono associare segnalazioni in modalità *drag and drop*, dando segnalazioni aperte oppure chiuse, aggiornandone facilmente lo stato.

Questo strumento è un simil report e segnala l'avanzamento delle segnalazioni.

Per sopperire alla mancanza di campi custom, possono creare gerarchie di etichette associando vari stati di avanzamento, negli esempi reali di uso di GitHub come ITS.

Git: caratteristiche e confronto con SVN (Git vs SVN)

Oggi andremo a vedere la possibilità di storicizzazione e lo stesso Git, software di controllo distribuito utilizzabile anche in CLI creato da Linus Torvalds, ispirandosi a strumenti proprietari analoghi per facilitare lo sviluppo del kernel Linux. Le caratteristiche sono:

- Branching and Merging, è incentivato lo sviluppo su Branch diversi (locali o condivisi)
- Piccolo e veloce, con la maggior parte delle operazioni fatte in locale, con velocità e performance come requisiti primari
- Distribuito, anche tramite clone del repository, avendo quindi backup multipli con diversi workflow
- Integrità, poiché ogni commit è identificato da un ID (checksum SHA-1 di 40-caratteri) che ne garantisce la struttura
- Staging Area, dove vengono validati i file modificati che potranno essere versionati da un commit. Qui aggiungiamo tutto ciò che interessa ai fini del commit
- Free ed Open Source, con licenza GNU e codice sorgente pubblico
- Il codice visto come una serie di snapshot, salvando esattamente lo stato del progetto in quel momento ed un link allo stesso file modificato per come era salvato in precedenza.

Segue una strutturazione in copie locali in cui i file possono essere in 3 stati:

- 1) nella Working directory/Modified, con file modificati ma non ancora validati
- 2) nella Staging Area/Staged, validati ma non ancora committati. Il commit salva uno snapshot di tutti i file presenti nella staging area
- 3) nel Repository locale/Committed, in cui appunto il file è preso dalla Staging Area e salvato in locale

Git richiede una configurazione ed è disponibile per tutti gli OS anche con client appositi (es. Gitkraken, Sourcetree). Prima di tutto user ed e-mail:

```
git config --global user.name "Bugs Bunny"
```

```
git config --global user.email bugs@gmail.com
```

Le configurazioni vengono salvate a vari livelli:

- system: per l'intero sistema per tutti gli utenti
- global: per il singolo utente
- local (di default): per singolo repository

I comandi utili sono:

- Creazione del repository locale nella cartella corrente: *git init*
- Clonazione di un repository remoto nella cartella corrente: *git clone url localDirectoryName*

Add and Commit

- Aggiungere i file nella staging area e creazione di una Snapshot: *git add Hello.java Goodbye.java*
- Commit e salvataggio di una nuova versione nel repository locale: *git commit -m "Fixing bug #22"*
- Rimuovere un file dalla staging area senza perdere le modifiche: *git reset HEAD filename*
- Rimuovere un file dalla staging area perdendo le modifiche: *git checkout - filename*

Stato e ripristino modifiche

- Vedere lo stato dei file workspace o nella staging area: *git status* oppure *git status -s*
- Per vedere cos'è stato modificato ma non ancora validato nella staging area: *git diff*
- Per vedere cos'è stato modificato nella staging area: *git diff -cached*
- Per vedere la lista dei cambiamenti (commit) nel repository locale: *git log*
- Per vedere le ultime 2 modifiche: *git log -2*

Branch e merging

- Per creare un nuovo branch: *git branch name*
- Per avere la lista dei branch locali: *git branch*
- Per passare in un branch: *git checkout branchname*
- Per effettuare attività di merge di un branch nel master:
git checkout master git
merge branchname

Repository remoto

- Lista dei repository remoti: *git remote* oppure *git remote -v*
- Fetch: recupero i nuovi branch e le modifiche dal remoto senza aggiornare la working copy (senza fare merge): *git fetch origin*
- Pull: recupero le ultime modifiche dal repository remoto e aggiornano la working copy (fetch and merge): *git pull origin master*
- Push: Per inviare le modifiche al repository remoto: *git push origin master*

SVN VS GIT

	SVN	Git
Controllo di versione	centrale	distribuito
Repository	Repository centrale in cui vengono create le copie di lavoro	Copie locali del repository su cui poter lavorare
Permesso di accesso	Basato sul percorso	Per tutta la directory
Visualizzazione delle modifiche	Registra i file	Registra i contenuti
Cronologia delle modifiche	Completa solo nel repository, le copie di lavoro contengono solo la versione più recente	Repository e copie di lavoro contengono la cronologia completa
Connessione di rete	Ad ogni accesso	Necessaria solo per la sincronizzazione

Laboratorio 2: Git Workflow e GitHub ITS

Si crea una nuova repository; il prof consiglia di dare l'accesso in SSH, dando un'apposita chiave. Ciò che viene fatto è la creazione di un file readme che viene committato.

Importante: occorre aggiungere una chiave SSH all'account per poter lavorare correttamente.

Questo viene fatto:

- tramite terminale (<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>)
Attenzione che Windows presenta il comando ls nella guida; ovviamente non è supportato dal caro Windows. Consiglio nel caso Windows il caso sottostante; per Linux basta il link precedente, trovando chiave SSH e poi aggiungendola.
- tramite GUI (aprendosi Git GUI, cliccando *Show SSH Key* e poi cliccando *Generate SSH Key*. Successivamente si va in Settings dell'account GitHub e poi si aggiungere la chiave in copia/incolla, eventualmente aggiungendo una passphrase). Eventualmente:
- <https://docs.github.com/articles/generating-an-ssh-key/>
- <https://support.automaticsync.com/hc/en-us/articles/202357115-Generating-an-SSH-Key-on-Windows#:~:text=On%20the%20Start%20Menu%20of,and%20select%20Show%20SSH%20Key>

Si vede in questo laboratorio un esempio di conflitto, inviando delle modifiche ad un file di una repo remota e committandolo sul branch master. Viene inserita una issue di riferimento con un ID e l'aggiunta del file viene collegata ad essa.

Segue l'esempio di un Centralized Workflow tra due sviluppatori:

- Il primo effettua un commit e salva sulla repo remota
- Il secondo prova ad inviare le modifiche alla repo remota, aggiorna il suo branch locale e prima di inviare le modifiche deve risolvere i conflitti (modificando il file, eseguendo il commit e poi il push)

In ultimo l'esempio come Feature branch workflow:

- lo sviluppatore 1 si posiziona sul master, recupera le modifiche dal repository remoto e aggiorna la sua copia locale con l'ultima versione scaricata
- lo sviluppatore 1 crea un nuovo branch e si posiziona sul nuovo branch creato
- lo sviluppatore 1 crea il nuovo file ed effettua le modifiche e verifica lo stato dei file presenti nella sua copia locale, aggiunge il nuovo file all'area di staging e lo committa

- lo sviluppatore 1 condivide il nuovo branch nel repository remoto

Per integrare lo sviluppo nel ramo principale lo sviluppatore 1 può:

- effettuare una pull request
- effettuare il merge direttamente nel ramo master

oppure:

- lo sviluppatore 1 si posiziona sul ramo master e lo aggiorna
- effettua il merge con il nuovo ramo
- se sono presenti conflitti, devono essere risolti altrimenti si possono inviare le modifiche al repository remoto

Visione degli altri workflow: GitFlow e Fork Workflow; inizio framework SCRUM

Parliamo del GitFlow ancora, dove si accenna ad uno sviluppo in parallelo rispetto al ramo di develop, bloccando tramite il ramo di release eventuali sviluppi di feature per la milestone principale. Quindi andremo a fare un'attività di merge verso un ramo master, contenente l'ultima versione rilasciata. Taggando poi una nuova versione, è possibile trovare il ramo con il tag corretto e fare una hotfix.

Proseguiamo vedendo il Git Flow (fine lab scorso), seguendo tutto quello che sta al link:

<https://danielkummer.github.io/git-flow-cheatsheet/>

La parte la aggiungo per totale completezza (non essendo presente con tutti i comandi a differenza delle altre nelle guide/pdf del prof).

git flow init

(Mettiamo il ramo per tutte le funzionalità di creazione)

Segue l'inizializzazione tipo così (per caso mio ho i branch diversi, il branch di develop si chiamava *develop*, ramo master è *master*).

```
Branch name for production releases: [next release] first
warning: ignoring ref with broken name refs/heads/next release
Branch name for "next release" development: [branch] second

How to name your supporting branch prefixes?
Feature branches? [feature/] feature
Bugfix branches? [bugfix/] bugfix
Release branches? [release/] release
Hotfix branches? [hotfix/] hotfix
Support branches? [support/] support
Version tag prefix? []
Hooks and filters directory? [C:/Users/tiger/temp/.git/hooks]
```

Implementiamo un branch per una nuova feature:

```
C:\Users\tiger\temp>git flow feature start occhiali
Switched to a new branch 'featureocchiali'

Summary of actions:
- A new branch 'featureocchiali' was created, based on 'second
- You are now on branch 'featureocchiali'

Now, start committing on your feature. When done, use:

    git flow feature finish occhiali
```

```
C:\Users\tiger\temp>echo "occhiali" >> occhiali.txt
C:\Users\tiger\temp>git add occhiali.txt
C:\Users\tiger\temp>git commit -m "creato il file occhiali"
[featureocchiali 757b223] creato il file occhiali
1 file changed, 1 insertion(+)
create mode 100644 occhiali.txt
```

Seguono gli altri comandi:

```
git checkout develop
git flow feature start cappello
git add cappello.txt
git commit -m "aggiunto il file cappello"
git flow feature finish cappello
```

Andiamo quindi a rilasciare la funzionalità presente aprendo un ramo di release (release branch) rilasciando la funzionalità richiesta.

Lo schema delle modifiche si vede con *git log*.

Avviamo poi:

```
git flow release start v.1.0
echo "README V.1.0" >> README.txt
git flow feature finish
```

Sul ramo di "develop" si sta continuando a lavorare, con il ramo master che con conclude mettendo insieme le feature dei rami di develop, ad esempio.

```
git checkout release/v.1.0
git flow release finish cappello
git flow feature publish
```

Il ramo occhiali è solo su develop, invece il ramo master avrà solo cappello il readMe.

Similmente switcha a develop (*git checkout develop*) e incorporare altre funzionalità (*git feature start pipa*)

Noi quindi abbiamo visto la pubblicazione e la storicizzazione; inoltre con *feature publish*, viene pubblicata la release dal branch, eventualmente può essere chiusa e convalidata.

Vediamo quindi meglio fork workflow, completando la visione di tutti i workflow.

Vado quindi a lavorare su un repo pubblico su cui non ho i permessi con il fork, spostandosi nella mia working area e quindi modificarlo, clonarlo o altro.

Ad esempio:

```
git clone (indirizzo di un progetto, che inizia con "git@github"...)
A questo punto un semplicissimo esempio:
```

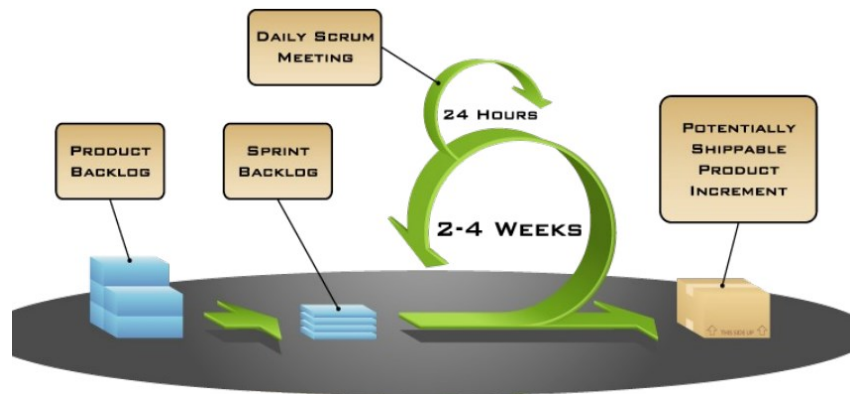
modifica di un file readMe, aggiunta (*git add*), commit (*git commit ""*) e si vede subito la modifica.

Creo quindi una pull request, dove all'altro progetto arriverà una notifica di modifica, che potrà essere integrata nell'originale.

Esempio reale con uso di repo distribuiti sono i fork su repo del cliente (caso consulenza di progetto) e inoltre della push su un branch con alias dell'originale.

SCRUM: disamina completa

Cominciamo a parlare del framework SCRUM, avendo un approccio a staffetta atto allo sviluppo dei prodotti in gruppo, cercando di essere veloci e competitivi. Scrum è un processo agile che nasce per lo sviluppo di progetti complessi (adaptive problems = difficili da definire e da risolvere) e che ci permette di concentrarci sulla consegna del maggior valore business nel più breve tempo.



Ci permette di ispezionare software funzionante rapidamente e ripetutamente (ogni due settimane o ogni mese) in cui il business stabilisce le priorità e i team si organizzano per scegliere la strada migliore per consegnare le funzionalità a priorità più alta. Ogni due settimane o ogni mese, chiunque può vedere il software funzionante e decidere se rilasciarlo così com'è o continuare a migliorarlo per un altro sprint.

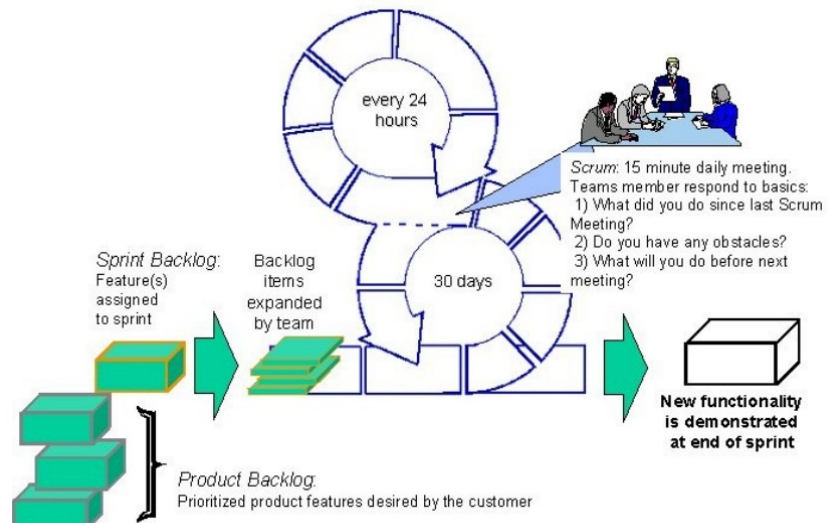
Scrum viene usato da tutte le compagnie, sia per i propri progetti che per terzi e in qualunque campo applicativo. Esso è leggero, facile da capire e difficile da padroneggiare.

I 3 pilastri principali sono:

- 1) Trasparenza (es. linguaggio comune per una conoscenza condivisa, definizione di "Done")
- 2) Controllo (es. ispezioni pianificate per prevenire variazioni non desiderate)
- 3) Adattamento (es. aggiustamenti per minimizzare ulteriori deviazioni tramite feedback continuo)

Dall'immagine vediamo:

- il *product backlog*, che comprende le attività con priorità massima/maggiore per il cliente;
- lo *sprint backlog*, che assegna un certo numero di feature per ogni fase dello sprint;
- *daily scrum meeting*, l'incontro giornaliero di 15 minuti che viene fatto per capire gli ostacoli, cosa è stato fatto nello sviluppo e l'oggetto del successivo incontro;



I gruppi si auto-organizzano e il prodotto evolve attraverso "sprint" mensili. Il processo di creazione è quindi iterativo/incrementale, trattando i requisiti come *product backlog* e non prescrivendo particolari pratiche ingegneristiche, ma basandosi solo sull'esperienza acquisita o sul campo, incrementalmente ottimizzando lo sviluppo ed il controllo sul rischio.

Il focus è quindi di concentrarsi sulle piccole funzionalità in ogni sprint, ricevendo anche i feedback dell'utente e adattando sulla base di esso i propri requisiti. Tutti gli stakeholders decidono quindi se rilasciare il progetto così com'è o se implementare in successivi sprint.

Il principio è: *piuttosto che fare tutto di una singola cosa, i gruppi Scrum fanno un pezzetto di tutto.*

In base alla Definition of Done (cosa deve essere fatto), si capirà se le attività corrispondono alle esigenze espresse in precedenza, definite con formalismi prima della creazione di attività.

Tutti questi principi in linea teorica seguono l'Agile Manifesto, con questi principi:

Gli individui e le interazioni più che i processi e gli strumenti
Il software funzionante più che la documentazione esaustiva
La collaborazione col cliente più che la negoziazione dei contratti
Rispondere al cambiamento più che seguire un piano

Durante lo sprint non si cambia; ne stabiliamo la durata degli sprint sulla base di quanto possiamo mantenere i cambiamenti all'esterno di un singolo sprint.

Lo sprint backlog può essere chiarito e rinegoziato tra il Product Owner e il team di sviluppo non appena si conosce di più. La durata dello sprint assicura che il rischio di scostarsi da quanto chiesto dal Product Owner sia limitato alla durata dello sprint. Uno sprint può essere cancellato se lo Sprint Goal diventa obsoleto; tuttavia, avendo una durata limitata (max 1 mese), raramente ha senso.

Parliamo ora di ruoli: (di questi non fa parte il Project Manager)

Ruoli
Product owner
ScrumMaster
Development Team

Product owner: Definisce le caratteristiche del prodotto, rappresentando il desiderio del committente (cioè il cliente), decidendo date e contenuto del rilascio e capendo la redditività del prodotto (ROI/Return of Investments, rapporto tra risultato operativo e capitale investito). Adegua poi le caratteristiche per ogni iterazione, secondo quanto necessario, accettando o rifiutando i risultati del lavoro.

Scrum Master: Conduce il progetto e fa ragionare il product owner in merito a come le attività funzionano, adottando il tutto a livello di gestione dei valori e delle pratiche Scrum. Rimuove gli ostacoli, assicurandosi poi che il gruppo di lavoro sia pienamente operativo e produttivo. Previene il working team da interferenze esterne, coadiuvando il lavoro tra Product Owner e Team di sviluppo (ruolo di *servant leader*).

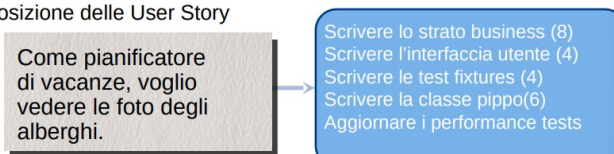
Development Team: Generalmente composto da 5-9 persone, responsabili dell'incremento in conformità alla Definition of Done, lista di obiettivi. Naturalmente questo racchiude una serie di competenze trasversali (programmatori, tester, progettisti di user experience, ecc.). Tutti lavorano full-time, ma possono esserci eccezioni (es. amministratori di database). Generalmente si auto-organizza.

Definiamo poi gli eventi:

Sprint planning: La stima delle attività viene fatta dal development team, in un ambito time boxed (8 ore per Sprint di 1 mese), selezionando dal product backlog gli item che può impegnarsi a completare. Viene creato quindi lo Sprint backlog in maniera collaborativa per tutto il team, definendo i tasks ed eseguendo una stima per ciascuno. Vengono poi decomposte le User Story (cioè le specifiche da condividere con il team di sviluppo capendo cosa sviluppare), tipo in questo modo:

Eventi
Sprint planning
Daily scrum meeting
Sprint review
Sprint retrospective

• Decomposizione delle User Story



Definiamo poi il meeting dello sprint, eseguendo il planning come segue:



La stima dello sprint backlog si stima non in ore ma in storypoint, capendo le singole attività in base alla "pesantezza" (level of effort), presupponendo siano ognuna scomposte in sottoattività.

Daily scrum meeting (stand up meeting): Incontro giornaliero di 15 minuti circa fatto in piedi, perché si pensa venga fatto in fretta (prima della pausa caffè/pausa pranzo). Si cerca di sincronizzarsi capendo lo stato delle attività di tutti, aggiornando la scrumboard. Ad esso può anche partecipare il product owner. Non è un SAL, ma è un impegno assunto tra pari.

Sprint review: Rappresenta un recap e una organizzazione di gruppo su quanto fatto fino a quel momento. È time boxed, 4 ore per sprint di 1 mese, realizzato senza slide, con la regola delle 2 ore per preparazione. Il gruppo di lavoro presenta ciò che ha realizzato durante lo sprint e viene validato/accettato quanto realizzato. Generalmente coinvolge tutto il gruppo e anche esterni.

Sprint retrospective: Fatta dopo la Sprint review e prima del prossimo Sprint Planning, anche qui Time boxed (3 ore per Sprint di 1 mese), valutando ciò che funziona e ciò che non funziona. Si cerca quindi di capire se il progetto qualitativamente è convincente, definendo le attività possibilmente migliorabili dal team. A questo partecipa tutto il gruppo di lavoro. Esistono vari template per poter organizzare le caratteristiche volute, possibili miglioramenti, ecc.



Parliamo poi di **artefatti**:

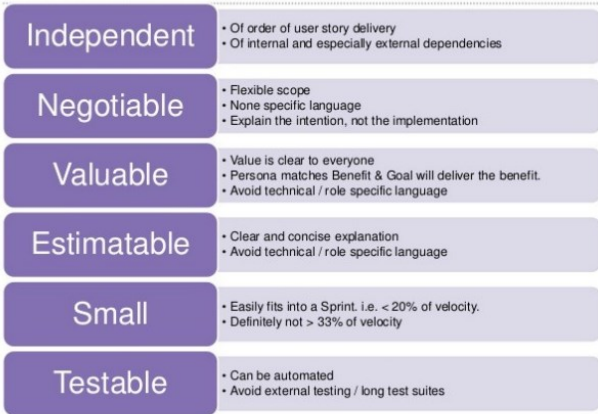
- Artefatti
- Product backlog
- Sprint backlog
- Burndown charts

Product backlog: L'insieme di requisiti e funzionalità, miglioramenti, fix composta dal Product Owner con l'aiuto dello Scrum Master, rivalutando frequentemente le attività con il Development Team. Viene raffinato in maniera dinamica e continuativa. Un esempio di definizione di attività (dx):

Backlog Item	Stima
Permettere ad un ospite di effettuare una prenotazione	3
Come ospite, voglio cancellare una prenotazione.	5
Come ospite, voglio cambiare le date di una prenotazione.	3
Come impiegato dell'hotel, posso lanciare i report RevPAR (Revenue Per Available Room = Fatturato per camera disponibile)	8
Migliorare la gestione delle eccezioni	8

INVEST by Bill Wake

14



Qui (a sx) invece descriviamo le caratteristiche delle User Stories (comunque scomposte da parte del development team sulla base delle esigenze lavorative).

Si cerca poi di dare una breve indicazione dell'obiettivo principale dello sprint, dando tutte le funzionalità all'utente (user stories mica per niente, come si vede quindi) entro la fine di quello specifico sprint goal.

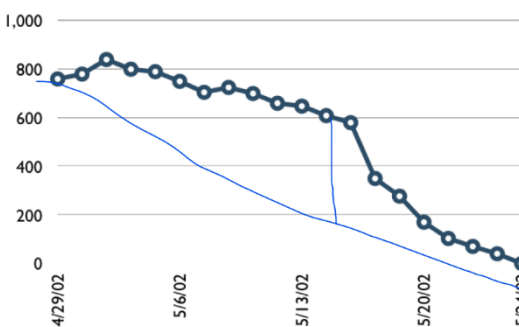
Sprint backlog: Ogni componente del Development Team sceglie cosa fare e la stima delle attività viene gestita durante lo stesso giorno, dato che ogni membro può aggiungere/cancellare/modificare parti dello sprint backlog, facendo "emergere" i lavori da svolgere durante gli sprint. Il lavoro deve essere chiaro, decomponendolo a mano a mano che il progetto avanza. Tutto deve essere massimamente visibile a tutti attraverso la scrumboard. Un esempio (dx):

User Story	Tasks	Day 1	Day 2	Day 3	Day 4	Day 5	...
As a member, I can read profiles of other members so that I can find someone to date.	Code the ...	8	4	8	0		
	Design the ...	16	12	10	4		
	Meet with Mary about ...	8	16	16	11		
	Design the UI	12	6	0	0		
	Automate tests ...	4	4	1	0		
	Code the other ...	8	8	8	8		
As a member, I can update my billing information.	Update security tests	6	6	4	0		
	Design a solution to ...	12	6	0	0		
	Write test plan	8	8	4	0		
	Automate tests ...	12	12	10	6		
	Code the ...	8	8	8	4		

In questo contesto, vengono introdotte due importanti definizioni intermedie:

Definition of Done: Definisce il significato di completamento ("Done") per ogni Sprint Item, definendo il minimo set di attività che definisce l'attività completata. Varia per ogni gruppo di lavoro e deve essere ben chiaro per ogni membro del gruppo di lavoro.

Acceptance criteria: Definisce se una storia sia stata completata come voluto, con frasi semplici condivise tra Product Owner e Development Team. Possono essere incluse con la User Story. Rimuovono l'ambiguità dei requisiti. Un esempio di grafico che capisce le attività (vedendo poi lo scostamento delle attività dall'andamento ipotizzato, come si vede disegnato brutalmente da me).



Burndown charts, in questo contesto si usa lo *Sprint Burndown Chart*, che evidenzia in maniera chiara l'andamento del Team. È una rappresentazione grafica della velocità con cui il lavoro viene completato e di quanto lavoro rimane da fare. La linea segue (esempio fatto dal prof), storicamente, l'evoluzione del progetto in un punto X.

25/03/2022: Presentazione primo assignment e Build Automation

Si parla per l'appunto di build automation, quindi rendere automatica la compilazione del software, compilando codice sorgente e impacchettandolo in codice binario, usando test automatici.

Scritto da Gabriel

Storicamente, questa pratica è sempre stata eseguita nella fase di creazione dei makefile, oggi tuttavia vi sono due tipologie caratteristiche che la implementano esplicitamente:

- *build-automation utility*, con lo scopo di creare un *build artifact* (pacchetti, file di log, report, dunque file immutabili prodotti da una build; esempi: Make, Gradle, Ant, ecc.);
- *build-automation servers*, solitamente strumenti web-based che eseguono utility di build-automation programmate o attivate su base giornaliera

Gli errori più comuni nelle build dei progetti sono principalmente le dipendenze (39%) e la compilazione in Java (22%), generazione di documentazione, configurazione errata, file non trovati e via discorrendo.

È chiaro come la fase di integrazione di un progetto sia fondamentale, che si pone come obiettivo di creare un progetto open-source compilabile, tale da potervi collaborare ed implementare modifiche effettuate, individuando eventuali bug nelle dipendenze (quindi, partendo dal sorgente, crearne un artefatto utilizzabile).

Ci sono vari tipi di tool, per esempio di *scripting*, tramite utilizzo di apposito linguaggio o automatismo (tramite file script, makefile, Gradle), oppure *artifact oriented*, definendo e creando un artefatto (prodotto) tramite lo strumento usato (Apache Maven e NPM).

Il processo di build si compone di un insieme di passi che trasformano gli script di build, il codice sorgente, i file di configurazione, la documentazione e i test in un prodotto software *sempre* distribuibile.



Sempre dal libro *Pragmatic Programmer*, deriva la descrizione delle caratteristiche Agile come CRISP quindi:

- *C/Completo*, indipendente dalle fonti di build
- *R/Ripetibile*, per cui un'esecuzione ripetuta dà sempre lo stesso risultato, accedendo ai file contenuti nel sistema di gestione del codice
- *I/Informativo*, descrivendo lo stato del prodotto
- *S/Schedulabile*, programmato ad una certa ora ed eseguibile automaticamente
- *P/Portabile*, indipendente il più possibile dall'ambiente di esecuzione

Da questo vi sono i repo degli artefatti (*artifact repository*), mantenendo dati e documentazioni. A tale scopo approfondiamo proprio *Maven*, è uno strumento software di gestione e comprensione dei progetti.

Basato sul concetto di un Project Object Model (POM), cioè un file XML che contiene tutti i dettagli di un progetto e della sua configurazione. Maven è lo strumento più usato nel mercato.

Sfrutta il paradigma "*Convention over configuration*" che prevede una configurazione minima (o addirittura assente) per il programmatore che utilizza un framework (come Maven) che lo rispetti, obbligandolo a configurare solo gli aspetti *che si differenziano dalle implementazioni standard* o che non rispettano particolari convenzioni di denominazione o simili.

Alcune delle principali caratteristiche di Maven sono:

- *build tool*, sono definite delle Build Lifecycle che permettono di configurare ed eseguire il processo di build (e altri processi);
- *dependency management*, le dipendenze di progetto vengono specificate nel file di configurazione (pom.xml). Maven si occupa di scaricarle in automatico da dei repository remoti e salvarle in un repository locale.
- *remote repositories*, sono stati definiti dei repository remoti dove sono presenti gran parte delle librerie di progetti opensource e dei plugin utilizzati da maven per implementare ed estendere le fasi dei Build Lifecycle
- *Universal Reuse of Build Logic*, per cui le Build Lifecycle e i plugin Maven permettono di definire in modo riusabile i principali aspetti richiesti per la gestione di progetto. Tra cui: l'esecuzione del processo di build, l'esecuzione di framework di test (p.es. JUnit/TestNG, quest'ultima estensione di Junit che usa l'istanza della classe di test per eseguirli tutti e organizza anche questa i test raggruppati come classi), la creazione di template di progetto (p.es. Applicazioni Java Web o applicazioni costruite con un determinato framework)

Esso è basato sul concetto centrale di Build Lifecycle, quindi costruendo e distribuendo un particolare artefatto chiaramente definito. Per ogni persona che costruisce il progetto è necessario solo un piccolo set di comandi per poter utilizzare Maven, per qualsiasi tipo di progetto.

Vi sono tre tipi di build lifecycles:

- *default*, gestisce la consegna/deployment del progetto
- *clean*, pulizia del progetto da configurazioni precedenti, dunque avvio pulito
- *site lifecycle*, creando il sito di documentazione del progetto

Per il default si strutturano le fasi presenti qui a lato:

- **validate** - validate the project is correct and all necessary information is available
- **compile** - compile the source code of the project
- **test** - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package** - take the compiled code and package it in its distributable format, such as a JAR.
- **verify** - run any checks on results of integration tests to ensure quality criteria are met
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

Normalmente, se un progetto è gestito con Maven (ed è presente il file pom.xml) sarà possibile eseguire uno dei processi (Build Lifecycle) invocando una delle fasi predefinite e normalmente tramite il comando "*mvn install*".

Verrà effettuata la compilazione, eseguiti i test, costruito l'artefatto e copiato nel repository locale (eseguite tutte le fasi del Build Lifecycle default precedenti a install). Le fasi vengono gestite da dei plugin maven (attraverso dei mojo/Maven plain Old Java Object, definito come maven goal, cioè una funzionalità che non sia già presente in maven)

L'idea quindi è la definizione delle fasi appena descritte, implementandole singolarmente con l'utilizzo di mojo. Si ha poi il *POM/Project Object Model*, unità fondamentale di lavoro, con un file XML che contiene informazioni sul progetto e sulle sue configurazioni (ad esempio vengono specificati plugin, profili di build, dipendenze di progetto, sviluppatori, ecc.).

Un esempio qui riportato permette di eseguire la build di un progetto Java:

```

1. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.
5.   <groupId>com.mycompany.app</groupId>
6.   <artifactId>my-app</artifactId>
7.   <version>1.0-SNAPSHOT</version>
8.   <!-- packaging>jar</packaging -->
9.
10.  <properties>
11.    <maven.compiler.source>1.8</maven.compiler.source>
12.    <maven.compiler.target>1.8</maven.compiler.target>
13.  </properties>
14.
15.  <dependencies>
16.    <dependency>
17.      <groupId>junit</groupId>
18.      <artifactId>junit</artifactId>
19.      <version>4.12</version>
20.      <scope>test</scope>
21.    </dependency>
22.  </dependencies>
23. </project>

```



In breve, *Archetype* è un toolkit di modelli di progetto Maven. Un archetipo è definito come un modello o pattern originale da cui generare cose dello stesso tipo (archetipo come parola, infatti, significa “esempio primario”, quindi si intende un punto pronto da cui partire per creare il proprio progetto).

Archetype aiuta gli autori a creare template di progetto Maven per gli utenti e fornisce agli stessi i mezzi per generare versioni parametrizzate dei template creati.

Il comando: `mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false`

Permette di creare un progetto come definito dall’archetipo *maven-archetype-quickstart* (esempio di default di creazione di un archetipo), avendo il riferimento al package “*com.mycompany.app*” e fa vedere il progresso nelle fasi di scaricamento/compilazione (*DinteractiveMode=false*).

Maven stesso quindi si pone come framework collezione di plugin.

Software Testing

Il software testing è uno strumento di investigazione del prodotto, creando informazione sulla qualità del prodotto/progetto, fornendo una vista indipendente sul software e permettendo al business di apprezzare e comprendere i rischi e discuterne correttamente con gli stakeholders. Tutto ciò deve rendere un prodotto ed un software adatto all’uso. Le linee guida/terminologie e casi di test in maniera corretta/oggettiva viene fornita dalla ISTQB (International Software Testing Qualifications Board).

Definiamo il processo di testing, sia statico che dinamico, concerne la preparazione, pianificazione e valutazione dei prodotti software, determinando la soddisfazione di determinati requisiti e trovando eventuali difetti. Ciò parte anche dalla stessa progettazione del manuale utente. L’approccio scettico nei confronti dei test è il non fidarsi se, a seguito delle esecuzioni di test, non si trovino bug o problematiche.

Lo sviluppo software è un processo umano, come tale passibile ad errori di qualsiasi natura. La buona scrittura di test è la rappresentazione di tutte le situazioni di errore/sviluppo del programma. Le figure principali che introducono errori sono:

- il programmatore, (45% di possibilità inserimento difetti) che fa un errore/*mistake* durante lo sviluppo oppure inserisce un difetto/bug, generando un comportamento inatteso che dà luogo ad una *failure*;
- l’analista, (20% analisi requisiti, 25% progettazione) che ascolta il cliente e ne interpreta le volontà, eventualmente immettendo difetti se non hanno una corretta comprensione di quanto richiesto. Naturalmente tutto ciò può pesantemente influenzare la qualità del prodotto.

Altre possibili cause di fallimento: Condizioni ambientali impreviste (temperatura, umidità, inquinamento, magnetismo, radiazione) che alterano il funzionamento dell'hardware sottostante.

Abbiamo poi le seguenti categorie di testing:

- *funzionale*, funzionalità richieste come esigenze reali dall'utente finale, presenti esplicitamente;
- *non funzionale*, cioè non funzionalità ma requisiti pratici (usabilità, facilità di utilizzo, sicurezza, accessibilità). La stessa legge italiana prevede l'erogazione di servizi e progettazione degli stessi si considerino tutte le possibili categorie di utenti (non vedenti, daltonici, ecc.);
- *statico*, che si occupa di test senza esecuzione di codice (dovuti anche in merito alla complessità del codice sorgente scritto, in maniera tale che implementando modifiche il progetto sia solido e stabile);
- *dinamico*, codice in esecuzione da qualche parte;
- *verifica*, un software fatto bene e secondo i canoni giusti, facendo in modo funzioni correttamente;
- *validazione*, rispettando le specifiche dell'utente e i requisiti stabiliti, convalidando il software affinché sia conforme ai suoi obiettivi (fit for purpose)

Definiamo una serie di casi di test, così come descritti dagli standard IEEE/ISTQB:

- *caso di test*, una serie di valori di input, precondizioni, risultati attesi (es. caso d'uso in un sito, scaricando del materiale; precondizione, essere registrati ed avere un profilo), postcondizioni, sviluppate per un particolare obiettivo con uno specifico requisito. Devono essere eseguite più prove per poter scrivere test buoni e la loro scrittura migliora l'ideazione degli stessi requisiti, risolvendo le ambiguità;
- *condizione di test*, aspetto testabile o un oggetto verificato da un sistema con uno o più casi di test.

I requisiti sono espressi nel linguaggio naturale, dunque forieri di ambiguità. I casi di test devono quindi essere parlanti, così da far capire esattamente a quale problema ci si sta riferendo (*testable requirements*), per fare in modo siano progettati secondo le catene CI/CD (continuous integration/continuous development).

Molto spesso infatti i clienti non sono tecnici nell'ambito informatico; bisogna quindi considerare questo nel processo di sviluppo. Esempi utili:

- la funzione di inserimento di un ordine deve essere di facile utilizzo (requisito non chiaro)
- l'operatore deve poter inserire un ordine nel sistema in meno di 3 minuti, senza bisogno di consultare il manuale operatore (requisito espresso in maniera più chiara)
- il sito deve essere veloce (requisito non chiaro)
- il sito deve presentare le sue pagine agli utenti finali con un tempo medio non superiore a 500 ms (requisito espresso in maniera più chiara)

Se i casi di test sono scritti bene, possono essere automatizzati e quindi usati da una macchina. Il processo di test è infatti composto da queste fasi:

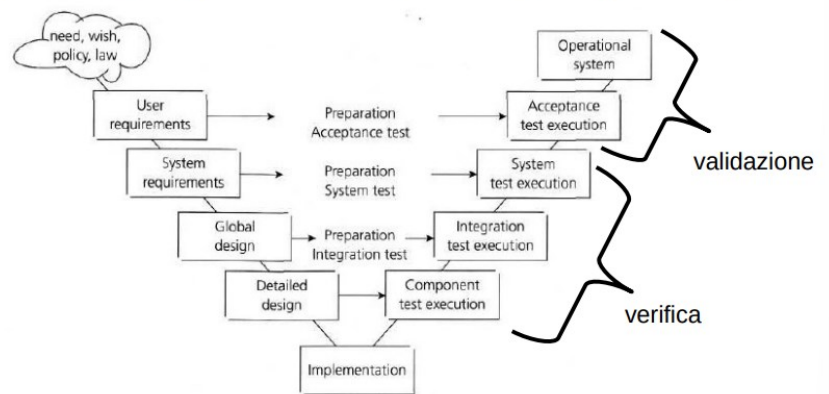
- 1) Test planning → stabilire o aggiornare un piano di test
- 2) Test control → azioni correttive e di controllo se il piano non viene rispettato
- 3) Test analysis → cosa testare
- 4) Test design → come testare
- 5) Test implementation → attività propedeutica all'esecuzione (p.es. definizione dei casi di test)
- 6) Test execution → eseguire il test
- 7) Checking result → verificare i risultati e i dati collezionati dalla test execution per capire se il test è stato superato/fallito
- 8) Evaluating exit criteria → verificare se sono stati raggiunti gli exit criteria definiti nel test plan
- 9) Test results reporting → riportare il progresso rispetto agli exit criteria definiti nel test plan
- 10) Test closure → chiusura del processo e definizione azioni di miglioramento

Si ha quindi una serie di principi (sette, cosiddetti *seven testing principles*):

- 1) *testing show presence of defects*, il test evidenzia e dimostra la presenza di difetti, non ne evidenzia l'assenza;
- 2) *exhaustive testing is impossibile*, testare tutto è impossibile, a meno di avere un'applicazione con input limitato e struttura logica molto semplice. Per questo motivo è importante valutare il rischio di malfunzionamento, capendo cosa serve testare. Ci si può concentrare sui casi più importanti da risolvere (*risk based testing*), oppure partendo dagli stessi requisiti (*requirement based*);
- 3) *early testing*, avviando la fase di test il prima possibile, parallelamente al processo di sviluppo stesso;
- 4) *defect clustering*, capendo che la maggior parte degli errori si estende ad un numero limitato di moduli; statisticamente circa il 20% di questi contiene circa l'80% degli errori;
- 5) *pesticide paradox*, quindi adattando i casi di test allo sviluppo software altrimenti disponiamo di test inefficaci e in grado di risolvere problemi vecchi ma non attuali;
- 6) *testing is context dependent*, ovviamente il testing viene fatto in base alle specifiche richieste ed applicazioni del software (ad esempio i software medici hanno test basati sul rischio, diversi da un'applicazione/sito online popolare in cui si richiedono rigorosi test prestazionali);
- 7) *absence of error fallacy*, cioè l'idea che se non si riscontrano difetti nel software non significa che sia perfetto, anzi, al contrario. Si deve capire se tutto corrisponde alle esigenze del cliente.

Il test è dipendente dal contesto (ad es. dispositivi medici richiedono test basati sul rischio, un sito web popolare richiede rigorosi test di prestazione, ecc.).

Si ha poi la correlazione tra fasi di sviluppo e fasi di test ed esecuzione dagli stessi (*v-model*), come ad esempio:



Altre categorie di testing utili sono:

- *component testing/unit testing*, verificando l'unità (cioè una cosa piccola come una classe o un metodo) testata poi separatamente; è facile e veloce da eseguire ed ogni modifica del codice sorgente dovrebbe scatenare l'esecuzione degli unit test. Essi sono indipendenti tra di loro e non dipendono dall'ordine di esecuzione; un sistema sotto test (SUT – System Under Test) è considerato come white box (cioè, in presenza del codice sorgente).
- *integration testing*, che verifica se sono rispettati i contratti di interfaccia tra moduli e subsystem e verificano l'integrazione tra più subsystem, i quali possono essere interni (verificati dagli unit testing) ed esterni (es. database, file system, ecc.). Essi sono lenti da configurare ed eseguire ed è considerato come white box.
- *system testing*, verificando il comportamento dell'intero sistema rispetto alle specifiche tecniche. Viene considerato sia white box che black box. Un esempio sono gli smoke testing, con l'obiettivo di trovare degli errori il prima possibile (prima della produzione) o eseguire altri test sul SUT, verificandone le funzionalità (*sanity checking*).
- *acceptance testing*, anche conosciuti come "UAT - User Acceptance Testing o End User Testing" e rappresentano una test suite su tutto il SUT, relativi agli use case e ai requisiti concordati con l'utente finale/cliente; viene svolto con questi ultimi in prima persona ed è considerato come black box. Per essi il SUT è considerato black box, appartiene alle categorie Test funzionale e Test dinamico.

Si verifica l'installazione preventiva di Maven e JDK.

Link utili (step by step per entrambi):

<https://toolsqa.com/maven/how-to-install-maven-on-windows/>

<https://www.windows11.dev/ce7in/java-55a9>

Si esegue quindi la creazione del progetto spostandosi in una cartella temporanea:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false
```

(Senza mettere i parametri di groupId ed artifactId dovranno poi essere specificati manualmente).

Ci sarà poi a seguito del comando *tree* la struttura:

```
C:\Users\tiger\temp>tree
Elenco del percorso delle cartelle per il volume Windows-SSD
Numero di serie del volume: 4A1F-425E
C:
├── my-app
│   ├── src
│   │   ├── main
│   │   │   ├── java
│   │   │   │   ├── com
│   │   │   │   │   ├── mycompany
│   │   │   │   │   │   └── app
│   │   └── test
│   │       ├── java
│   │       │   ├── com
│   │       │   │   ├── mycompany
│   │       │   │   │   └── app
```

Andiamo dentro *my-app* e poi verifichiamo la struttura del file *pom.xml*, aprendolo con un editor (code *pom.xml* ad esempio).

Lanciando *mvn install* vengono definiti file jar e POM del progetto creato.

Aprendo un IDE diretto, vediamo tutta l'alberatura, compresi plugin, class (file che non verranno versionati) e test del codice. Maven segnala subito dove si trova un errore nel progetto.

Ogni plugin, a livello di file POM, è configurato con un goal in merito al proprio progetto ed artefatto.

La cartella Target viene cancellata con *mvn clean*, essendo file generati.

Poi con *mvn install* dovrà dare lo stesso risultato.

Vediamo poi l'esempio dell'analisi di codice statica con il plugin *checkstyle* e proviamo ad utilizzarne il goal. Se vogliamo utilizzarlo e basta, non modifichiamo il POM.

Ad esempio avviamo: *mvn checkstyle:check*

Segnala il codice del progetto e analizza tutto un progetto, dando possibili errori/warning.

Andiamo quindi a creare un file *checkstyle.xml* con il contenuto indicato dalla guida, inserendo anche il file *license.txt*. Una volta modificato il *pom.xml* inserendo il plugin, eseguendo *mvn verify* si genera un errore, dato che non è stato configurato Prova nel checker.

Si dovrà inserire come commento *“//prova”* all'interno del file *MyApp.java* del progetto *my-app*.

Ogni artefatto ha una sua dipendenza, ciascuna con dipendenze/licenze, collegate tra loro transitivamente.

Esse possono essere aggiunte sotto forma di pacchetto come import in un file *.java* da cui poi possiamo compilare ed eseguire la verifica delle dipendenze.

Test e Unit testing

Per unit testing (testing d'unità o testing unitario) si intende l'attività di testing (prova, collaudo) di singole unità software. Per unità si intende normalmente il minimo componente di un programma dotato di funzionamento autonomo e una funzionalità atomica; a seconda del paradigma di programmazione o linguaggio di programmazione, questo può corrispondere per esempio a una singola funzione nella programmazione procedurale, o una singola classe/singolo metodo nella programmazione a oggetti.

Come le altre forme di testing, lo unit testing può variare da completamente "manuale" ad automatico. Specialmente nel caso dello unit testing automatico, lo sviluppo dei test case (cioè delle singole procedure di test) può essere considerato parte integrante dell'attività di sviluppo (per esempio, nel caso dello sviluppo guidato da test). Vengono sviluppati dal programmatore che sviluppa le unità, per verificare l'assenza di alcuni errori, e documentare il comportamento dell'unità prodotta.

I test di unità sono in queste categorie: White Box, Verifica, Funzionali e Test Dinamico

Esso deve essere *A TRIP (Automatic Thorough Repeatable Independent Professional)*, eseguendo tutto in maniera automatic/automatica, in modo rapido (test semplici, pochi secondi), senza input umano (precondizione che si abbiano parametri ben definiti), ed in modo autonomo (effettuando l'esecuzione in certi orari del giorno, capendo poi quando e dove i test falliscono).

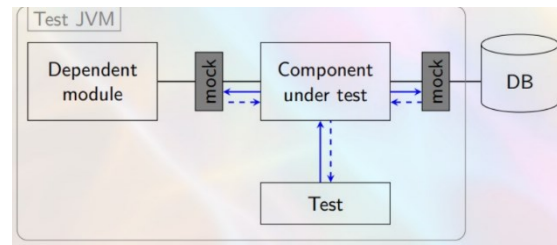
Similmente devono essere esaustivi/thorough, con test accurati, testando il buon comportamento di parti che potrebbero creare errore, calcolando le righe di codice esercitate, varie diramazioni possibili, il numero di eccezioni e dati generici che capiscono se il progetto sia carente di test di unità. Un piccolo esempio qui a fianco:

```

jacoco > com.example.jacoco > Rectangle.java
Rectangle.java
1. package com.example.jacoco;
2.
3. public class Rectangle {
4.     private int x;
5.     private int y;
6.     private int width;
7.     private int height;
8.
9.     public Rectangle(int x, int y, int width, int height) {
10.         if (width <= 0 || height <= 0)
11.             throw new IllegalArgumentException("Dimensions are not positive");
12.
13.         this.x = x;
14.         this.y = y;
15.         this.width = width;
16.         this.height = height;
17.     }
18.
19.     public boolean intersects(Rectangle other) {
20.         if (x + width <= other.x)
21.             return false;
22.         if (x >= other.x + other.width)
23.             return false;
24.         return (y + height > other.y && y < other.y + other.height);
25.     }
26. }
    
```

Gli stessi test devono essere repeatable/ripetibili, indipendenti quindi dall'ordine di esecuzione (ai fini del risultato) e indipendenti dall'ordine di esecuzione e dall'ambiente di esecuzione con la tecnica del *mock object*, oggetti simulati che riproducono il comportamento degli oggetti reali in modo controllato e simulabile. Inoltre, sono independent/indipendenti, dall'ambiente di esecuzione e da elementi esterni al progetto e dall'ordine di esecuzione, verificando il comportamento di ogni singolo aspetto.

Come si vede qui, usiamo un framework con l'uso di una classe, specifici metodi di I/O aspettandosi un certo valore.



L'esempio qui a lato analizza con dei mock i moduli e le dipendenze, nonché le componenti sotto test.

I test devono essere professionali/ragionevoli, scritti e mantenuti con la stessa professionalità del codice di produzione, con i test scritti spesso con più righe di codice del progetto stesso.

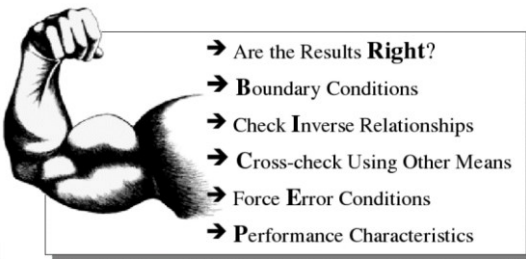
Un framework di test deve quindi avere:

- Un modo per configurare l'ambiente di esecuzione del test
- Un modo per selezionare un test o un insieme di test da eseguire

- Un modo per analizzare i valori attesi, prodotti dalle unità
- Un modo standard per eseguire ed esprimere se il test è stato superato, se è fallito o se sono stati prodotti degli errori

Prendendo come esempio Junit 4, per creare i test di unità sono necessari i seguenti componenti:

- Un modo per configurare l'ambiente di esecuzione del test:
<https://github.com/junit-team/junit4/wiki/Test-fixtures>
- Un modo per selezionare un test o un insieme di test da eseguire:
<https://github.com/junit-team/junit4/wiki/Aggregating-tests-in-suites>
<https://github.com/junit-team/junit4/wiki/Categories>
- Un modo per analizzare i valori aspettati, prodotti dalle unità:
<https://github.com/junit-team/junit4/wiki/Assertions>
- Un modo standard per eseguire ed esprimere se il test è stato superato, se è fallito o se sono stati prodotti degli errori:
<https://github.com/junit-team/junit4/wiki/Getting-started>



Ci chiediamo quindi se i risultati prodotti dal codice siano corretti, cioè se il risultato atteso è uguale al risultato prodotto dall'unità. Capita che i requisiti non siano chiari o possono cambiare nel tempo. In questi casi i test di unità sono un buon punto di partenza per documentare nel codice, come uno sviluppatore ha interpretato i requisiti e descrivere il comportamento delle unità realizzate.

Si verifica infatti che il test è formato da precondizioni/asserzioni/azioni, documentando il codice in maniera utile. Esempio molto semplice: verifichiamo se la somma dei numeri sia effettivamente uguale a 6.

```
public class Calculator {
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
}
```

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void evaluatesExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6, sum);
    }
}
```

Definiamo poi le boundary conditions (**CORRECT**), con errori che di solito accadono in condizioni limite, parte molto importante:

- **Conformance** - I valori sono conformi al formato atteso? [A+1] [1-2]
 - **Ordering** - I valori seguono o non seguono un ordine? [++ 1 2 3]
 - **Range** - I valori sono all'interno di un valore di minimo e massimo appropriato? [2147483647 + 2]
 - **Reference** - I valori possono provenire da codice che si riferisce a dati esterni che non sono sotto il controllo del codice?
 - **Existence** - I valori esistono (non sono nulli, non sono zero, sono presenti in un determinato insieme)? [null]
 - **Cardinality** - I valori sono nella quantità desiderata? [1 + 2 + 3 +]
 - **Time** - I valori rispettano un ordine temporale?
- Con il termine valore si fa riferimento sia ai parametri di input dei metodi di un'unità, che ai dati interni all'unità e ai risultati che questa produce.

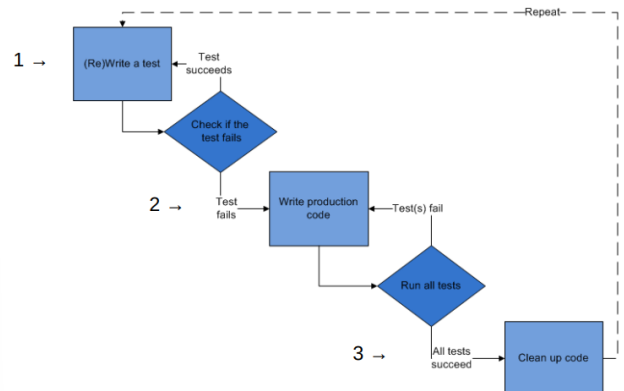
Similmente si ha una verifica dell'applicazione della funzionalità inversa (*check inverse relationship*) (partendo ad esempio dal risultato), come ad esempio la radice quadrata (risultato rispetto al dato di partenza) ed inserimento di un elemento in una pila (verificando se l'inserimento è andato a buon fine rispetto al prelievo). Utilizziamo poi strumenti esistenti per verificare le funzionalità (*cross-check using other means*), utilizzando il vecchio sistema per verificare se la nuova unità abbia lo stesso comportamento (ad es. Migrazione da un vecchio sistema ad uno nuovo appena realizzato).

Similmente nel mondo reale gli errori accadono (*force error conditions*), creando un buon progetto ricreando condizioni di errore con buone caratteristiche di performance (*performance characteristics*), quindi i test di unità devono essere veloci perché devono essere eseguiti molto spesso.

Il *test-driven development* (abbreviato in TDD), è un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test, e che lo sviluppo del software applicativo sia orientato esclusivamente all'obiettivo di passare i test automatici precedentemente predisposti.

Più in dettaglio, il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto "ciclo TDD":

- 1) Nella prima fase (detta "fase rossa"/(Re)Write a Test), il programmatore scrive un test automatico per la nuova funzione da sviluppare, che deve fallire in quanto la funzione non è stata ancora realizzata.
- 2) Nella seconda fase (detta "fase verde"/Write production code), il programmatore sviluppa la quantità minima di codice necessaria per passare il test.
- 3) Nella terza fase (detta "fase grigia" o di refactoring, cioè ristrutturare il codice senza cambiarne la funzionalità originale/Clean up code), il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualità.



Laboratorio 4: Junit

All'interno di un progetto viene generato l'archetipo maven con il package di riferimento e, una volta modificato correttamente il file pom.xml, si ha la generazione del progetto con esecuzione sotto forma di Maven Build. Successivamente ad una classe aggiungiamo un test creandolo sotto forma di *Junit Test Case*.

Annotation	Description
@Test public void method()	The Test annotation indicates that the public void method to which it is attached can be run as a test case.
@Before public void method()	The Before annotation indicates that this method must be executed before each test in the class, so as to execute some preconditions necessary for the test.
@BeforeClass public static void method()	The BeforeClass annotation indicates that the static method to which is attached must be executed once and before all tests in the class. That happens when the test methods share computationally expensive setup (e.g. connect to database).

Un esempio classico di Test è l'esecuzione dei metodi assert e un insieme di annotazioni utili:

@After public void method()	The After annotation indicates that this method gets executed after execution of each test (e.g. reset some variables after execution of every test, delete temporary variables etc)
@AfterClass public static void method()	The AfterClass annotation can be used when a method needs to be executed after executing all the tests in a JUnit Test Case class so as to clean-up the expensive set-up (e.g disconnect from a database). Attention: The method attached with this annotation (similar to BeforeClass) must be defined as static.
@Ignore public static void method()	The Ignore annotation can be used when you want temporarily disable the execution of a specific test. Every method that is annotated with @Ignore won't be executed.

Poi si vedono gli insiemi di test (raggruppati in una sola cartella), per ogni singola classe, per esempio su *StringHelper* che contiene metodi del tipo:

```
public String truncateAInFirst2Positions(String str) {
    if (str.length() <= 2)
        return str.replaceAll("A", "");

    String first2Chars = str.substring(0, 2);
    String stringMinusFirst2Chars = str.substring(2);

    return first2Chars.replaceAll("A", "") + stringMinusFirst2Chars;
}

public boolean areFirstAndLastTwoCharactersTheSame(String str) {
    if (str.length() <= 1)
        return false;
    if (str.length() == 2)
        return true;

    String first2Chars = str.substring(0, 2);
    String last2Chars = str.substring(str.length() - 2);

    return first2Chars.equals(last2Chars);
}
```

Similmente esistono test di unità *parametrici*, come:

```
@Parameters
public static Collection<String[]> testConditions() {
    String expectedOutputs[][] = {
        { "AACD", "CD" },
        { "ACD", "CD" } };
    return Arrays.asList(expectedOutputs);
}

@Test
public void testTruncateAInFirst2Positions() {
    assertEquals(expectedOutput,
        helper.truncateAInFirst2Positions(input));
}
```

```
import static org.junit.Assert.*;

@RunWith(Parameterized.class)
public class StringHelperParameterizedTest {

    // AACD => CD ACD => CD CDEF=>CDEF CDA => CDA

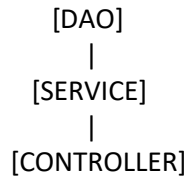
    StringHelper helper = new StringHelper();

    private String input;
    private String expectedOutput;

    public StringHelperParameterizedTest(String input, String expectedOutput) {
        this.input = input;
        this.expectedOutput = expectedOutput;
    }
}
```

Generalmente si eseguono singolarmente come Junit Test su un IDE. Nel nostro caso abbiamo esempi utili nel package *Business*, con la conversione di valute per prodotto, con nomi dei test poco utili e raggruppamento nelle strutture utili, tipo array.

La struttura schematica di un pattern simil MVC (Model View Controller), stando alla nostra creazione di progetto con relativi test è:



Di fatto vediamo gli ultimi tipi di test, riunite un suite (tramite annotazione `@RunWith(Suite.class)`- oppure `@SuiteClasses({ArraysTest.class,StringHelperTest.class})` nel nostro caso e anche le categorie, tipo:

```

@Category({SlowTests.class, FastTests.class})
public class B {
    @Test
    public void c() {
    }
}

@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@SuiteClasses( { A.class, B.class }) // Note that Categories is a kind of Suite
public class SlowTestSuite {
    // Will run A.b and B.c, but not A.a
}

@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@ExcludeCategory(FastTests.class)
@SuiteClasses( { A.class, B.class }) // Note that Categories is a kind of Suite
public class SlowTestSuite {
    // Will run A.b, but not A.a or B.c
}
    
```

Vogliamo quindi un modo per eseguire un test in modo singolo, tramite i mock (istanze di metodi definite nei test), tramite ad esempio il framework *Mockito*. Esso viene aggiunto al file POM come dipendenza. Un esempio completo: <https://www.vogella.com/tutorials/Mockito/article.html>

Parliamo quindi di *analisi statica*, quindi fatta senza effettivamente eseguire il programma fatta da un umano oppure in maniera automatica, verificando bad practices nel codice. Fa parte dei *test non funzionali*, *white box* (con la presenza di codice sorgente quindi) oltre che essere *test statico* (non richiedendo l'esecuzione). Il codice quindi viene esaminato in maniera conforme, non sostituendo la code review manuale, tuttavia risultando efficiente e veloce come lista di warning, incapsulando logiche e semantiche in maniera semplice.

Il software di revisione automatica del codice verifica la conformità del codice sorgente a un insieme predefinito di regole o best practices. L'uso di metodi analitici per ispezionare e rivedere il codice sorgente per rilevare bug è stata una *pratica di sviluppo standard*. Questo processo può essere eseguito sia manualmente che in modo automatizzato. Con l'automazione, gli strumenti software forniscono assistenza con il processo di revisione e ispezione del codice. Il programma o lo strumento di revisione visualizza in genere un elenco di avvisi (violazioni degli standard di programmazione). Un programma di revisione può anche fornire un modo automatizzato o in maniera guidata per correggere i problemi riscontrati.

Questi strumenti incapsulano anche una profonda conoscenza delle regole sottostanti e della semantica del codice, necessaria per eseguire correttamente un test.

Si fa riferimento alla *teoria delle finestre rotte*, dicendo che la risoluzione dei piccoli problemi riduce il rischio dei problemi più grossi; ciò è particolarmente utile come immaginabile in questo ambito. Strumenti come SCM Tools, IDE, Editor, e CI/CD Tools vengono spesso usati per queste pratiche.

Si struttura poi una serie di funzionalità:

- Imporre il rispetto di convenzioni e stili
- Verificare la congruità della documentazione
- Controllare metriche ed indicatori (complessità ciclomatica, grafo delle dipendenze, numerosità delle linee di codice)
- Ricercare codice copiato in più punti
- Ricercare errori comuni nel codice
- Misurare la percentuale di codice testato
- Ricercare indicatori di parti incomplete (p. es. tag)



Vediamo l'esempio dei software usati, ad esempio *CheckStyle*, strumento di sviluppo che cerca di far aderire codice Java ad uno standard di programmazione, forzandolo al suo interno trovando problemi di design nel codice o nei metodi; può anche controllare il layout o la formattazione.

Esempi di uso nel codice Java al link: <https://checkstyle.sourceforge.io/checks.html>

Esempio di utilizzo dentro Maven: <https://maven.apache.org/plugins/maven-checkstyle-plugin/>

Similmente abbiamo *FindBugs*, soprattutto usato nel codice Java come strumento di caccia di bug.

Si hanno delle implementazioni utili con esempio, ad esempio al link:

<https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html>

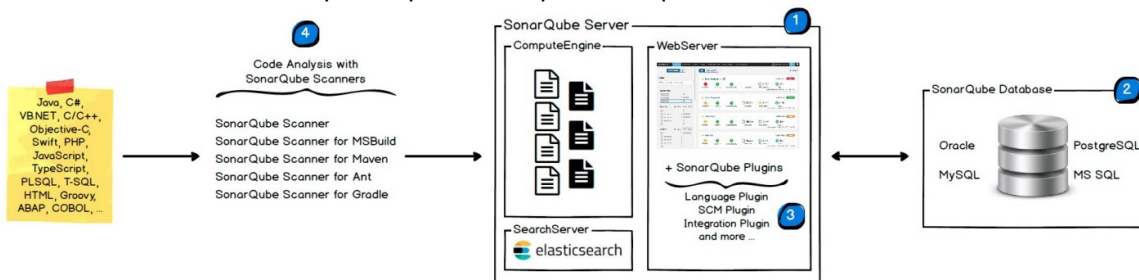
volendo sempre utilizzabile dentro Maven come plugin e con una serie di controlli al link:

<http://findbugs.sourceforge.net/bugDescriptions.html>

Altro plugin utile è *PMD* secondo il CPD (Continuous Professional Development), che trovava difetti comuni di programmazione come variabili inutilizzate, blocchi try-catch vuoti, oggetti non necessari, copia/incolla, ecc. Il seguente link riporta una serie di feature per linguaggio; anche qui esiste un plugin Maven da usare:

https://pmd.github.io/pmd-6.9.0/pmd_rules_java.html

Altro molto utile e il più utilizzato ora è *SonarQube*, abilitando pratiche di CI e in grado di trovare vulnerabilità, code smells (debolezze nel codice) e offrendo la registrazione di metriche e la creazione di grafici che documentano l'evoluzione del progetto. Il suo scopo è integrare tramite l'*ispezione continua del codice* attraverso branch e pull request. Esso presenta questa architettura:



MTSS semplice (per davvero)

Presenta le seguenti funzionalità:

- Storicizza l'andamento della qualità (revisando anche le issues e segnalando falsi positivi)

- Permette di verificare se c'è un miglioramento o un deterioramento del progetto nel tempo

- Permette di stabilire un *quality profile* (un insieme di regole) da applicare al progetto

- Permette di stabilire un *quality gate* per verificare se la qualità del progetto rispetta determinati standard

- Le issue segnalate vengono classificate in base alla gravità (Blocker, Critical, Major, Minor, Info) in:

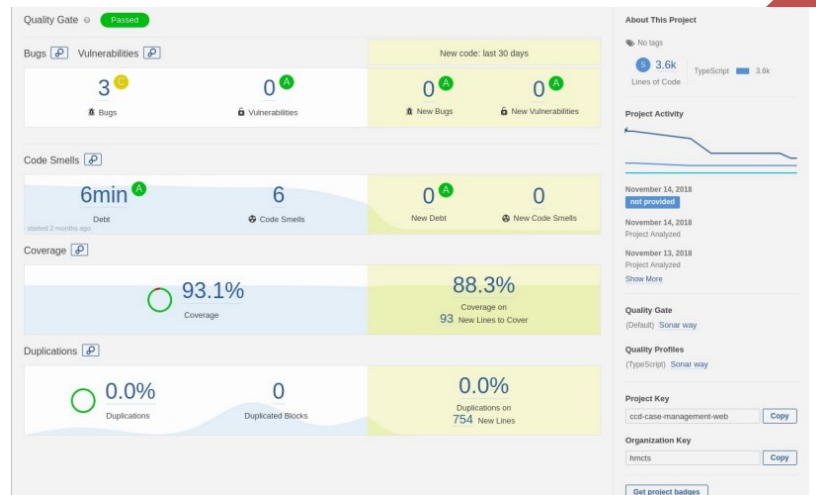
- Vulnerabilità → Permette di valutare il livello di sicurezza del progetto (Security)
- Bug → Permette di valutare l'affidabilità del progetto (Reliability)
- Code Smell → Permette di valutare la mantenibilità del progetto (Maintainability)

Un esempio grafico di Sonarqube è riportato a lato.

La configurazione pratica, ad esempio, è tramite AWS con cui si configura una macchina su un certo IP a cui si può accedere e configurare su questa altri mezzi come Docker e, su questa, lavorare clonando progetti ed eseguendoli direttamente.

Di fatto la tendenza di controllo del codice è cresciuta nel corso del tempo nel contesto reale, con le aziende sempre più interessate alla manutenzione e gestione corretta dello stesso, migliorandolo almeno in parte.

Nell'ordine di utilizzo, il più diffuso è proprio SonarQube, seguito da FindBugs, CheckStyle, nessuno strumento utilizzato e PMD. Esso è uno strumento di Automated code review storicizza l'andamento della qualità e permette di verificare se c'è un miglioramento o un deterioramento del progetto e nel tempo (continuous code integration). Un esempio di code smells comuni è dato qui sopra.

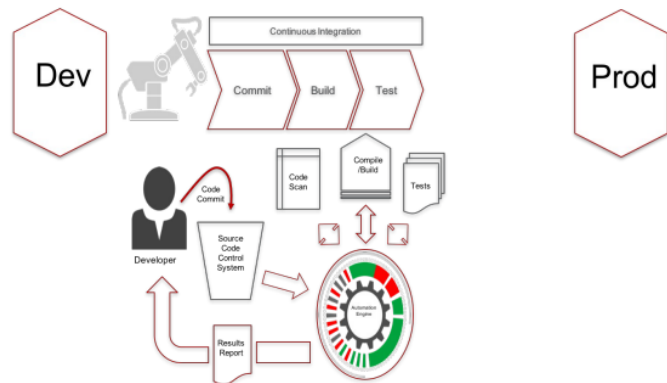


Continuous Integration/CI

Nell'ingegneria del software, l'integrazione continua o continuous integration è una pratica che si applica in contesti in cui lo sviluppo del software avviene attraverso un sistema di versioning.

Consiste nell'allineamento frequente (ovvero "molte volte al giorno") dagli ambienti di lavoro degli sviluppatori verso l'ambiente condiviso (mainline). Il CI è stato originariamente concepito per essere complementare rispetto ad altre pratiche, in particolare legate al Test Driven Development (sviluppo guidato dai test, TDD). Si suppone generalmente che siano stati predisposti test automatici che gli sviluppatori possono eseguire immediatamente prima di rilasciare i loro contributi verso l'ambiente condiviso, in modo da garantire che le modifiche non introducano errori nel software esistente. Per questo motivo, il CI viene spesso applicato in ambienti in cui siano presenti sistemi di *build automatico* e/o esecuzione automatica di test, come *Jenkins*.

In molti progetti, per lunghi periodi di tempo e soprattutto quelli su cui si sviluppa su un solo ramo di sviluppo (*centralized workflow*), spesso non è in uno stato funzionante. Infatti, l'applicazione non si prova fino a quando non è finito il processo di sviluppo. Nei progetti, dunque, spesso viene pianificata la fase di integrazione alla fine del processo di sviluppo.



Si può entrare nella fase di *integration hell*, richiedente molto tempo e nessuno ha modo di prevedere quando terminerà, ritardando il rilascio del prodotto.

La CI non fa in modo di eliminare i bug, ma *cerca di trovarli il prima possibile*, integrando gli sviluppi quanto più spesso. L'integrazione è fatta frequentemente, anche almeno una volta al giorno, portando ad avere integrazioni multiple al giorno.

Al completamento di un'attività viene costruito il prodotto:

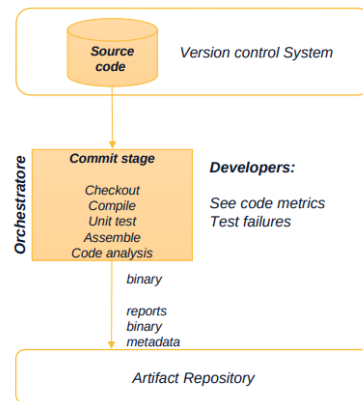
- ogni volta che uno sviluppatore invia un commit al VCS viene eseguito il processo di build (compilazione e test)

Se il processo di costruzione fallisce l'attività non *continua* fino a che il prodotto non viene riparato; se non è possibile riparare il prodotto immediatamente (in pochi minuti) si ritorna all'ultima versione funzionante. In questo modo si assicura la presenza di un prodotto consistente e pronto per essere validato e rilasciato.

Per implementare la pratica di CI è necessario che:

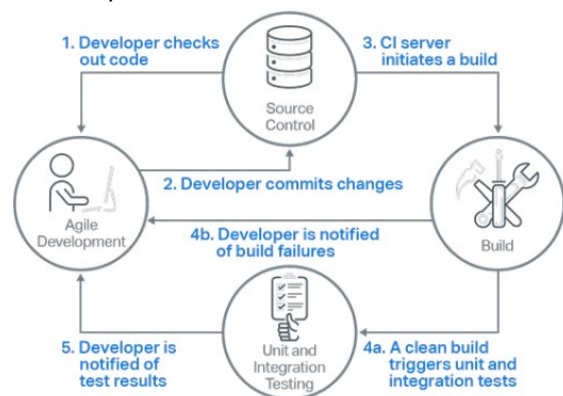
- 1) Il codice del progetto venga gestito in un VCS
- 2) Il processo di build del progetto sia automatico
- 3) Il processo di build esegua delle verifiche automatiche (test di unità, test di integrazione, analisi statica...)
- 4) Il team di sviluppo adotti correttamente questa pratica
- 5) Un sistema automatico dove eseguire il processo di build ad ogni integrazione [Opzionale]

Quindi, il codice sorgente, presente all'interno di un VCS, viene analizzato e testato da parte degli sviluppatori. Nella fase di commit, si eseguono vari checkout, compilazioni, test di unità ed analisi del codice, rendendolo poi gradualmente pubblico sotto forma di artefatto (e poi archiviato in apposita *artifact repository*).



Nel dettaglio, questo processo implementa varie fasi:

- 1) Controllo se il processo di build è in esecuzione nel sistema di CI. Se è in esecuzione aspetto che finisca, se fallisce lavoro con il team in modo da sistemare il problema
- 2) Quando il processo di build ha terminato con successo, aggiorno il codice nel mio workspace con il codice del VCS ed effettuo l'integrazione in locale
- 3) Eseguo il processo di build in locale in modo da verificare che tutto funzioni correttamente
- 4) Se il processo di build termina con successo invio le modifiche al VCS
- 5) Attendo che il sistema di CI esegua il processo di build con i miei cambiamenti
- 6) Se il processo di build fallisce mi fermo con le attività di sviluppo, e lavoro per sistemare il problema in locale e riprendo dal passo 3
- 7) Se il processo di build termina con successo passo allo sviluppo dell'attività successiva



Integrare frequentemente gli sviluppi è la chiave; in questo modo le modifiche sono poche e facili da gestire e se viene segnalato un errore nella build è più facile individuarla.

Il principio segue il *divide et impera*; divisione di un progetto in molte attività brevi. Se non vengono eseguiti dei test automatici, il processo di build può solo verificare se il codice integrato compila correttamente. Utile quindi creare una *suite di test automatici* (automated test suite).

I test che possono essere eseguiti allora sono:

- test di unità
- test di integrazione di subsystem interni
- analisi statica del codice

Dato che il processo di build deve essere eseguito frequentemente, se il processo è lento:

- Gli sviluppatori smetteranno di eseguire il processo di build e i test prima di inviare le modifiche al VCS → Inizieranno a generare più build in errore
- Il processo di CI richiederà così tanto tempo che si verificheranno più commit nel momento in cui è possibile eseguire nuovamente la build → sarà più difficile identificare cosa ha fatto fallire la build
- Si disincentiva l'invio frequente delle modifiche al VCS

Il processo deve essere quindi rapido ed efficace, con la corretta gestione del workspace.

Ogni sviluppatore deve essere in grado di:

- eseguire localmente il processo di build/test/deploy, gestendo codice di produzione/di test/script di configurazione ambiente nel VCS
- essere preparati a tornare alla versione precedente, scaricando le modifiche dal VCS
- avere un corretto versionamento di build, verificando l'esito della compilazione nel CI server
→ Se il processo di CI fallisce lo sviluppatore deve essere in grado o di risolvere il problema o di ripristinare la versione del VCS all'ultimo stato consistente

La priorità numero uno in un progetto è avere una build funzionante; se si ha una *broken build*, essa deve essere corretta in un tempo limitato. Se non è possibile correggere l'errore che ha fatto fallire la build velocemente, ripristinare lo stato del VCS all'ultima versione funzionante. Non è ammesso correggere il problema commentando le verifiche che hanno fatto fallire la build; bisogna agire concretamente.

Laboratorio 5: GitHub Actions

Qui serve per forza Java 8, configurabile nel path JAVA_HOME in quanto si necessita di un particolare framework per far andare l'applicazione.

Il link al download:

<https://www.oracle.com/java/technologies/javase/javase8-archive-downloads.html>

Per Windows 11:

Una volta fatto questo si imposta all'interno di Impostazioni – Informazioni sul Sistema – Impostazioni avanzate – Variabili d'ambiente il percorso all'installazione di Java, del tipo:

C:\Program Files\Java\jdk1.8.0_202

E si è a posto

Maggiori info qui per tutti, Windows compreso, qui:

<https://www.baeldung.com/java-home-on-windows-7-8-10-mac-os-x-linux>

Eseguire il fork del progetto <https://github.com/dduportal/dw-demo-app> e poi clonarselo in locale (git clone (link appena messo qui)

Entrare nella cartella "dw-demo-app" ed eseguire mvn clean install

A quel punto vengono scaricati tutti i plugin e l'installazione di Maven viene eseguita correttamente con JDK 8. Si configura manualmente la repo seguendo il link:

<https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven>

in cui viene impostata il file .yml sulla base della propria configurazione Java, dando dei parametri personalizzati.

Di fatto noi abbiamo impostato il processo di *build* per il processo di esempio (spiegato qui).

Entriamo quindi nel nostro user, nella cartella forkata "dw-demo-app", entriamo su "Actions" e premiamo "Set up a workflow yourself" (nel caso non fosse il primo workflow, la voce non appare. Semplicemente basterà cliccare "New workflow" ed appare l'opzione).

Si crei il file build.yml (che sarà già dentro il percorso come github/workflows/build.yml con questo contenuto (con questa esatta formattazione, quella del prof scombrina un poco):

name: Java CI with Maven

on: [push]

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Set up JDK 1.8

uses: actions/setup-java@v3

with:

```
distribution: 'temurin'  
java-version: 8.0.332+9  
cache: 'maven'  
- name: Build with Maven  
  run: mvn -B install --file pom.xml
```

A questo punto il progetto eseguirà in un sistema Ubuntu versione 20.04 con le caratteristiche dettagliate, in particolare la Java JDK 8.0.332+9, impostando la build con Maven. Si dovrà e potrà ovviamente personalizzare se e dove necessario, ma per questo lab va fatta così.

In questo caso, volendo sfruttare il file *pom.xml* appena creato su Ubuntu 20.04, con quella specifica versione di Java (facendo riferimento in particolare alla distribuzione Temurin), si andranno a seguire questi passi (messi in un'immagine per pura comodità di formattazione e visione):

- **Sistema Operativo:** Ubuntu 20.04
- Vengono eseguiti i seguenti steps:
 - Effettuato il checkout del codice grazie alla action <https://github.com/actions/checkout>
 - Configurata la jdk 8 grazie alla action <https://github.com/actions/setup-java>
 - Eseguito il comando `mvn -B install --file pom.xml`

Ci sarebbe anche un altro modo per eseguire direttamente il build dell'applicazione partendo dal file *.jar* direttamente dentro la cartella target del progetto *dw-demo-app* (quindi nel percorso del tipo `C:\Users\tiger\dw-demo-app\target>`) e da lì eseguire (direi comando per Linux, in quanto su Windows si fa l'escape con `%JAVA_HOME%`):
`$JAVA_HOME/bin/java -jar ./target/demoapp.jar server ./hello-world.yml`
dopodiché si verifica la presenza ed attività del progetto su una porta di rete in una prima finestra di terminale.

Entrando su un altro si verifica che la connessione è attiva per quella porta. Poi viene fermata regolarmente la connessione con il solito CTRL+C (parte che cito in velocità completata dalle slide ma a noi meno importante della parte spiegata sopra).

A questo punto se tutto funziona, rientrando sulla sezione Actions del progetto *dw-demo-app* si vedrà che la build è andata a buon fine (tic verde); nel caso in cui ci sia qualche problema, da buon VCS, GitHub ce lo fa sapere subito, mandando una mail di fallimento. Similmente manderà una mail a seguito di configurazione con successo.

Si può volendo impostare manualmente all'interno del file *readme.md* una configurazione apposita del badge che indica lo stato di successo, seguendo il link <https://docs.github.com/en/actions/monitoring-and-troubleshooting-workflows/adding-a-workflow-status-badge>

Il prof consiglia, anche per il secondo assignment, di seguire il link <https://docs.github.com/en/actions/security-guides/encrypted-secrets> per poter configurare i propri commit in modo tale da non trasmettere le proprie credenziali all'esterno, cosa che normalmente succederebbe per qualunque utente abbia quella specifica cartella.

Artifact repository

Queste repo sono progettate ed ottimizzate per scaricamento ed ottimizzazione nel mantenimento di file binari usati e prodotti nello sviluppo software. La gestione dei pacchetti binari è centralizzata, in modo tale che gli artefatti sono generati ed utilizzati da tutta l'organizzazione, facendo in modo non ne si usino di diversi. Una *binary repository* è una repo software per pacchetti, artefatti e corrispondenti metadati. Può essere usata dai file binari e/o prodotti dall'organizzazione stessa oppure librerie di terze parti, che devono essere trattate diversamente per motivi tecnico/legali.

L'output di questa fase viene riutilizzato in fasi successive dalla pipeline di lavoro e dallo stesso team, lavorando in maniera facile e veloce. Non viene consigliato di mettere questo output all'interno di un VCS, considerando che la maggior parte lavora sul proprio disco.

Infatti, l'*artifact repository* tiene solamente traccia di alcune versioni. Una volta che una release candidata fallisce in qualche fase della pipeline di deployment (consegna), non si è più interessati a mantenerla. È dunque essenziale per il software poter tornare indietro a qualche revisione passata, in merito alle versioni esistenti. Un successivo controllo manuale nella pipeline è ben più complesso rispetto all'introduzione di successive revisioni della stessa pipeline.

Il processo binario deve essere ripetibile, pertanto anche cancellando le librerie e rieseguendo il commit, *si devono riottenere esattamente gli stessi file binari*. Una artifact repository ha queste caratteristiche:

- ambiente di deposito/pubblicazione dei prodotti
- intermediario per scaricare prodotti da depositi esterni
- effettuazione di ricerche e reperimento di informazioni sui vari prodotti
- gestione ed associazione dei permessi d'accesso sui prodotti
- segnalazione dei problemi di vulnerabilità sui prodotti
- verifica dei problemi legati a licenze (dipendenze con prodotti terzi)
- documentazione degli artefatti con metadati (i.e. pom.xml)

Tra gli artifact repository più famosi citiamo *Maven*, *Python*, *Ruby*, *npm*, *Debian* e *Docker*, in particolare quest'ultimo nato per pure esigenze collaborative. Gli artefatti (identificati univocamente da un *GAV (Group ID/Artifact ID/Version ID)*) hanno dei metadati (attraverso ad esempio un file pom.xml), fornendo dettagli in merito alla produzione/sviluppo, come il riferimento al commit del VCS da cui è stato creato l'artefatto, info su licenze/dipendenze ed hash MD5/SHA1 con id univoco per garantirne l'autenticità.

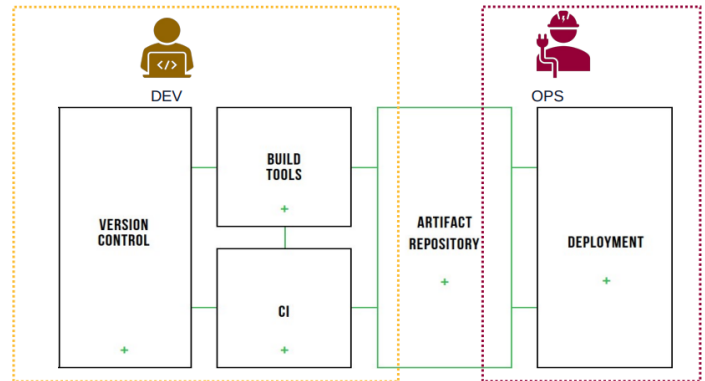
Nel caso di Maven la gestione degli artefatti (*Maven Repository Management*) permette di gestire:

- *proxy remote repositories*, offrendo la possibilità di configurare il repository interno per poter recuperare gli artefatti provenienti da repo esterni, qualora un certo artefatto non sia già presente nel repository interno;
- *hosted internal repositories*, condividendo all'interno di un'organizzazione artefatti di terze parti ma con vincolo di licenza (es. JDBC, connessione al database per Java);
- *release artifacts*, quindi artefatti di tipo Release che possono essere rilasciati solo una volta e vengono mantenuti nel repository;
- *snapshot artifacts*, dunque artefatti in fase di sviluppo che possono essere rilasciati più volte. Essi possono essere eliminati (mantenendo la versione più recente) ed hanno la keyword *SNAPSHOT* ed un loro timestamp.

Usare un Maven Repository Interno garantisce:

- velocità nel processo di Build, le dipendenze/artefatti di tipo plugin vengono scaricati dalla rete interna;
- maggiore stabilità e controllo nella gestione/analisi delle repo, rendendo semplice gestire gli artefatti presenti e stando attenti a quelli di terze parti, per evitare problemi
- facilitano la collaborazione, evitando di far creare gli artefatti dal VCS e identificano le versioni di rilascio certificate

Si cita il funzionamento del *DevOps*, pratica Agile che combina “development” ed “operations” fornendo un giusto bilancio nella velocità di implementazione delle modifiche e adattamento ai cambiamenti ma dall’altra combinando strumenti di priorità e feedback nel processo CI, basata su test continuo.



Graficamente si hanno:

- i *dev (developers)*, che gestiscono costruendo incrementalmente (CI) il codice con strumenti di build come visto fino ad ora (automatici/manuali);
- le *ops (operations)*, definite dal team come lavoro collaborativo e organizzando tutti gli strumenti in maniera standard.

Piccolo link di approfondimento: <https://about.gitlab.com/topics/devops/>

Abbiamo due casi d’uso:

- Docker, che permette di costruire molteplici immagini di pacchetto ed interdipendenze e compatibile con Docker Compose (uguale a Docker ma è configurato da uno YAML, mentre Docker funziona interamente a linea di comando) e Kubernetes (sviluppato da Google, organizza il funzionamento in container tramite unità chiamate pod, risorse di rete ed è più configurabile e tollerante ai guasti di Docker). La collaborazione è semplice e le sue versioni eseguono su varie applicazioni ed ambienti;
- Maven Gitflow, che funziona come il Gitflow classico ed esistono delle repo snapshot, in cui vengono pubblicate più volte versioni del software con appositi timestamp di riferimento e delle repo release, pubblicando una versione una volta sola. Quindi si ragiona esattamente come il Gitflow classico, sia con rami di release che anche rami di develop/hotfix/regular release.

Tra i binary/artifact repository più usati troviamo soprattutto Nexus o Artifactory, usati come repo manager dentro Maven.

Si introduce il discorso di *feedback loop*, che implementa ciclicamente le fasi di sviluppo del software secondo un’idea di creazione (con fasi di checkout, build, test di unità e test di smoke/accettazione), fase di staging (consegna e produzione) e fase di produzione stessa (deployment dell’applicazione). Ricevendo un feedback, sia manuale che automatico, da parte degli utenti finali, si coinvolgono direttamente ingegneri e product managers. Formalmente, il feedback loop è l’insieme di strumenti che:

- forniscono lo stato ad ogni passo/stage della pipeline
- inviano il messaggio giusto alla persona giusta
- permettono di misurare il processo

Lo scopo di questa implementazione è l’efficienza, confidenza (tenendo informati gli attori) ed attendibilità sui messaggi (inviare messaggi corretti è certamente un valore aggiunto). Questo deve essere ben stabilito, soprattutto quali attori notificare, che media utilizzare e con che priorità, frequenza e per quali eventi avvisarli. Implementazione utile al link: <https://www.cloudbees.com/blog/sending-notifications-pipeline>

L'obiettivo della pipeline è fornire un modello che permette di misurare il processo (tramite *kpi/Key Performance Indicator*, valori che misurano il successo in merito ad obiettivi aziendali e metriche):

- Ogni stage può fornire un valore qualitativo (OK/KO)
- Ogni gate permette di registrare il tempo trascorso
- Inviare feedback con questi valori permette di monitorare la qualità del processo
- Aggiungere feedback con metriche relative alla pipeline permette di:
 - Fornire informazioni al Business (p.es. Tempi di rilascio, bug corretti per versione, qualità statica del prodotto)
 - Aiutano a prendere decisioni (tecniche)
 - Permettono di dimensionare il processo in modo ottimale



Laboratorio 6 – Jenkins

Sulla macchina virtuale installata del laboratorio, accedere a <http://localhost:10000/>

Accedendo a WebIDE si scarica il codice dell'applicazione Java, entrando poi su Git per copiare l'indirizzo HTTP del codice. Inserito all'interno del WebIDE risulta visibile come progetto.

Vogliamo configurare Jenkins, cliccando sul link diretto dalla homepage.

Per fare ciò occorre configurare Maven nella sezione "Configurazione strumenti globali" e verificare se presente *maven3*; se non è presente, va cercato e installato con la barra di ricerca presente.

Similmente, è presente anche *Docker* con le variabili d'ambiente presenti nella guida del lab.

Si imposta la build sul progetto *demoapp-build*, presente su GitServer e di cui si vuole copiare lo URL cliccando sul tasto http quando si entra nel progetto.

La sezione Build va impostata sul comando *clean install*, mettendo una Post-build Action / Azioni post compilazione di tipo Archive the artifacts e impostando il File to archive come: `target/**/*.*.jar`

La build viene lanciata ad ogni commit, perché il job è automatico; se è giallo si ha un risultato inatteso (dato nel nostro anno da problemi di compatibilità, purtroppo non risolti).

L'idea sarebbe semplice; usare una VM funzionante in grado, tramite apertura di server Vagrant in ssh, di usare una versione di Jenkins ed un progetto di copia su cui far andare Maven.

Jenkins garantisce un buon utilizzo sia di ITS che di VCS.

Quest'anno il prof aveva condiviso VM e guida sbagliata, quindi la sintesi utile è pressappoco questa.

Continuous Delivery (CD)

La *continuous delivery* è una pratica usata nell'ingegneria del software per cicli corti, assicurandosi che il software sia rilasciato in maniera sicura in ogni momento, in particolare rilasciandolo manualmente.

Si prefigura come approccio utile a testare, costruire e rilasciare software con grande frequenza e velocità riducendo costi e rischi secondo un processo ripetibile e semplice.

Essa si struttura sulla base di alcuni punti:

- Il software è in stato distribuibile (deployable) durante tutto il lifecycle
- Il team dà la priorità a mantenere il software distribuibile lavorando su nuove funzionalità
- Chiunque può ottenere un feedback rapido e automatizzato sulla prontezza di produzione dei propri sistemi ogni volta che qualcuno apporta una modifica
- È possibile eseguire distribuzioni a pulsante (push-button deployments) di qualsiasi versione del software a qualsiasi ambiente su richiesta
- Si ottiene la CD integrando continuamente il software fatto dal team di sviluppo, compilando eseguibili, eseguendo test automatizzati sugli stessi per trovare problemi. Si possono pushare gli eseguibili in maniera incrementale in produzione, assicurandosi che il software funzioni di volta in volta.

Si configura anche l'abilità di integrare tutti i cambiamenti (nuove features, cambi di configurazione, bugfix, esperimenti) nella produzione in maniera sicura e veloce.

L'obiettivo è di costruire delle fasi di deployment a larga scala in maniera predicibile e con delle routine configurabili alla bisogna, eliminando completamente il processo di integrazione tradizionale (ad esempio il *dev complete*, quindi un prodotto creato, finito, testato e pronto alla consegna).

Continuous Delivery	Un insieme di pratiche progettate per assicurare che il codice sia sempre pronto per essere rilasciato rapidamente e in modo sicuro, attraverso tutto il suo ciclo di vita fino all'esercizio, realizzato passando prima gli eseguibili in un ambiente simile a quello di produzione ed effettuando test automatizzati per rilevare problemi.
Continuous Deployment	Una estensione del concetto di Continuous Delivery nella quale tutte le modifiche che superano i test automatizzati vengono automaticamente passate in produzione. Il Continuous Deployment automatizza il passaggio precedentemente effettuato in modo manuale nel Continuous Delivery, e consente rilasci multipli giornalieri.
Continuous Integration	La pratica di mettere insieme le copie di lavoro di tutti gli sviluppatori in un'unica linea principale condivisa (un code repository o blocco principale di codice) durante una giornata. In un processo automatizzato di Continuous Delivery, la Continuous Integration copre principalmente la fase di realizzazione del codice. Tipicamente la Continuous Integration si applica alle attività di integrazione, realizzazione e test del codice nell'ambiente di sviluppo.
Continuous Testing	L'esecuzione di test automatici in ogni fase della Deployment pipeline. Fornisce un feedback immediato in ogni fase, al fine di mitigare i rischi. Un Continuous Testing automatizzato rappresenta un elemento chiave della Continuous Integration e del Continuous Delivery. Assicura che il codice e l'ambiente operino in modo appropriato, e che il codice rimanga in uno stato rilasciabile.

Le motivazioni per una CD sono:

- avere feedback sui problemi introdotti dagli stessi sviluppatori, focalizzandosi quindi sulla parte DEV e assicura che il codice compili e vengano eseguiti test di unità, integrazione ed analisi statica. Questo non è sufficiente per garantire la possibilità di rilasciare il prodotto ad ogni modifica perché le attività che di solito fanno perdere più tempo avvengono nella fase di rilascio e test (e nella comunicazione collaborativa tra *dev* ed *ops*);
- porre un ambiente di risoluzione di problemi comuni durante un progetto software.

Nel concreto, per esempio:

- I sistemisti (OPS) aspettano molto tempo per ricevere la documentazione (procedure di rilascio) per effettuare il rilascio;
- I tester (TEST) attendono molto tempo per effettuare le verifiche e le validazioni nella versione giusta (e di avere un ambiente funzionante);
- Il team di sviluppo (DEV) riceve segnalazioni di bug su funzionalità che sono state rilasciate da settimane
- Ci si rende conto solo alla fine dello sviluppo (troppo tardi) che l'architettura scelta non permette di soddisfare i requisiti non funzionali (DEV e OPS)
- il software dunque non è rilasciabile perché si ha troppo tempo per farlo entrare simile alla produzione e si hanno molti difetti ereditati dal ciclo di feedback tra team di sviluppo e team di testing ed operations

Nell'approccio Agile si *risolve con il confronto diretto*.

L'obiettivo è dunque cercare di migliorare il processo nel rilascio delle modifiche del codice sorgente dato che il vantaggio competitivo si ha nella creazione di tutte le attività richieste nel processo.

Una "Value Chain" è una modellazione del processo che misura il valore globale ed il valore residuo, quindi il miglioramento ottenibile. Una Deployment Pipeline dà gli stessi risultati dell'utilizzo di una "Value Chain", controllando prestazioni ed efficienza secondo il principio di "Fast is cheap", trovando i problemi il prima possibile.

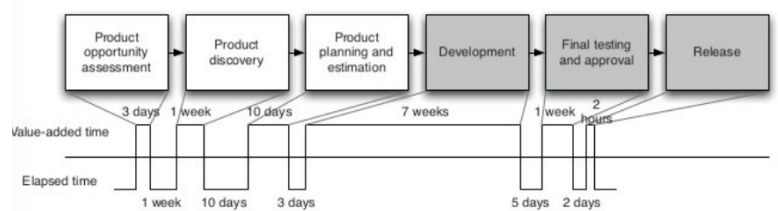
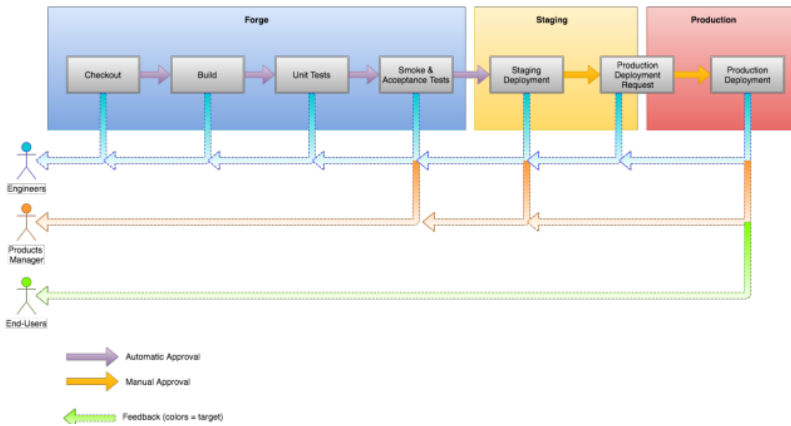


Figure 5.1 A simple value stream map for a product

Una possibile chain di valori e valutazione temporale trascorsa e stimata qui a destra:



Nel concreto, la *Deployment Pipeline* è la modellazione del processo di Deployment tramite una successione di fasi (stages) e verifiche (gates). In generale, il passaggio da una fase all'altra viene certificato tramite il superamento di una verifica, scatenando anche notifiche (anche nel caso dei fallimenti, idea del Fail-Fast).

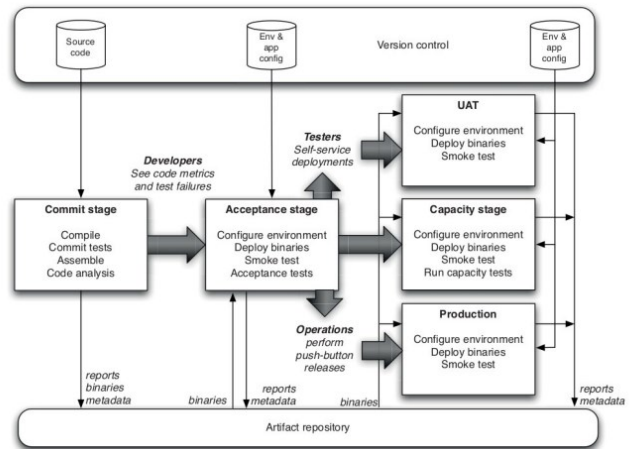
Essa è composta da stages mappate su attività misurabili (es. build/test). La transazione tra due stages viene definita *gate* e può essere automatica o manuale. I gates scatenano il passaggio allo stage successivo e possono essere multidirezionali (quindi ad esempio eseguiti in parallelo e/o in sequenza).

Una deployment pipeline presenta:

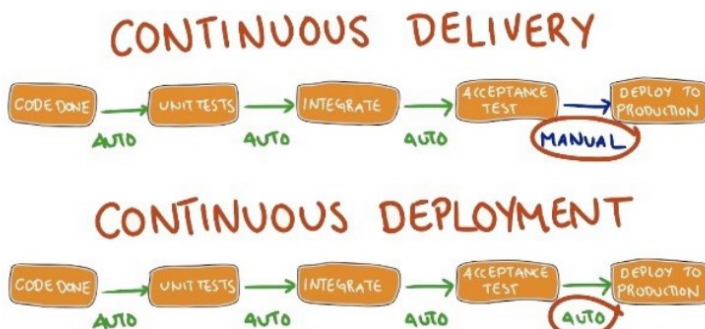
- fasi automatizzate (fase di commit, fase di accettazione/acceptance, capacity testing/quantità di dati o di traffico gestibile)
- fasi di self-service deployment (quindi ogni team ha visione completa delle fasi di deployment senza doversi interfacciare con altri), come exploratory testing (capire il codice ed il software imparandolo incrementalmente, testandolo, scoprendone pregi e difetti), integration testing, staging e produzione

L'idea base quindi è: continuare a lavorare su nuove caratteristiche, in modo tale che ogni cambiamento al sistema può essere rilasciato in produzione in ogni momento *premendo un bottone*, essendo la priorità l'implementazione delle nuove features.

Si deve quindi avere un rapporto di lavoro collaborativo con tutti i partecipanti (dev, test e ops), utilizzando le Deployment Pipelines per definire il processo di build, test e deploy dell'applicazione (e renderlo il più automatico possibile, distribuendo il tutto su più ambienti). La realizzazione della CD viene data qui a destra.




Si noti inoltre che Continuous Delivery != Continuous Deployment, come ben descritto da questa immagine:



I requisiti richiesti quindi sono:

- tutti gli strumenti di CI (VCS, Build Automation, Unit Testing, Artifact Repository di gestione dipendenze e file binari prodotti dal sistema)
- Configuration Management, strumenti che ci permettono, di gestire tramite codice, la configurazione degli ambienti dove dovrà essere rilasciato il nostro software (quindi può farne parte anche la Artifact Repository)
- Continuous Testing, test automatici a livello di sistema funzionali e non funzionali
- Orchestratore, sistema software che modella le esecuzioni delle pipeline (p.es. Jenkins)

Alcuni altri:

- *Only Build Your Binaries Once*, quindi eseguire il processo di build solo una volta garantisce di utilizzare lo stesso artefatto per effettuare tutte le verifiche in ogni ambiente. Questo ci garantisce che:
 - l'artefatto che verrà rilasciato in produzione sia esattamente lo stesso artefatto che è stato verificato e validato nelle Stages della pipeline
 - è una forma di ottimizzazione. Eseguire il processo di build più volte rende la pipeline meno efficiente. Infatti, per realizzare questa pratica è consigliato avere un repository dove rilasciare gli artefatti e da cui recuperare le informazioni del commit nel VCS da cui è stato creato l'artefatto (Artifact Repository). L'artefatto generato deve essere indipendente dall'ambiente di esecuzione: è necessario tenere il codice (che rimane lo stesso tra gli ambienti) separato dalle configurazioni che differiscono tra gli ambienti (p.es: riferimenti al DB, o a sub system esterni).
- *Deploy the Same Way to Every Environment*, poiché è essenziale utilizzare lo stesso script per effettuare il rilascio in differenti ambienti. In questo modo lo script di rilascio sarà più solido perché verrà verificato maggiormente, infatti:
 - gli sviluppatori lo utilizzeranno per rilasciare molto frequentemente negli ambienti di sviluppo
 - i tester e gli analisti lo utilizzeranno per rilasciare negli ambienti di test
 - quando verrà rilasciato in produzione lo script sarà stato eseguito molte volte e sarà più probabile che non fallirà. Gli script sono codice e quindi devono essere gestiti nel VCS, tenendo separati dagli script le configurazioni (che saranno differenti per ogni ambiente). Se gli ambienti di rilascio sono gestiti da altri gruppi (OPS) è necessario collaborare con questi gruppi per definire gli script di rilascio e condividerli con il DEV (VCS).
- *Smoke-Test Your Deployments*, verificando che il rilascio automatico è andato bene, di solito tramite smoke-test. Essi devono verificare il corretto funzionamento dei sub-system esterni (DB, altri web-services, messaging bus). Gli smoke-test sono veloci e semplici da realizzare e permettono di far fallire velocemente la pipeline in caso di problemi.
- *Deploy into a Copy of Production*, dunque prevedere di avere a disposizione un ambiente con le stesse caratteristiche (o quantomeno simili) dell'ambiente di produzione. Per essere sicuri che il deploy funzionerà, è necessario eseguire i test e gli script di rilascio in ambienti il più possibile simili a quello di produzione. L'ambiente dovrà avere:
 - La stessa configurazione di rete (e dei firewall)
 - Lo stesso sistema operativo (versioni e patching)
 - Lo stesso stack applicativo (application server, versione DB etc.)
 - I dati gestiti dall'applicazione devono essere in uno stato consistente
- *Each Change Should Propagate through the Pipeline Instantly*, quindi ogni modifica al codice sorgente deve avviare il processo di Deploy (pipeline). Molto probabilmente la pipeline impiegherà tanto tempo per eseguire l'intero processo, per questo è necessario inserire delle verifiche negli stages che la facciano fallire il prima possibile. Il processo di Continuous integration non eseguirà ad ogni commit e sarà più complicato identificare l'errore. Per questo si consiglia un CI Server che

possa eseguire il processo di CI a partire da uno specifico commit. In questo modo sarà più semplice effettuare attività di debug per identificare la modifica che ha fatto fallire il processo.

- *If Any Part of the Pipeline Fails, Stop the Line*, quindi progettare la pipeline in modo da eseguire per primi i controlli veloci e meno esaustivi. In questo modo sarà possibile far fallire la pipeline e notificare tutto il team (DEV, TEST, OPS) del problema.

I benefici legati alla CD sono:

- Riduzione del rischio legato al deploy: dal momento che si sta distribuendo piccole modifiche, è meno probabile corrompere il sistema ed è più facile risolvere l'errore in caso di problemi.
- Velocizzazione del tempo di rilascio: questa pratica porta ad avere rilasci più frequenti
- Maggiori feedback da parte degli utenti
- Progressi tangibili: molti Team monitorano i progressi in un ITS, che certifica il funzionamento in vari ambienti di produzione
- Minor costo: diretta conseguenza dell'automazione
- Prodotti migliori che soddisfano le aspettative degli utenti
- Team meno stressati e più collaborativi
- Maggiore documentazione implicita

Configuration Management (CM)/CM DevOps



La *configuration management* è un processo collaudato in vari ambiti (partendo dall'ambiente militare) che cerca di stabilire e mantenere la coerenza delle prestazioni, delle funzionalità e degli attributi fisici di un prodotto con i suoi requisiti ed informazioni operative per tutto il suo ciclo di vita.

Il processo di CM è utilizzato anche per la gestione dei servizi IT, come definito da *ITIL (Information Technology Infrastructure Library*, linee guida in questo campo), e per altri modelli di dominio nell'ingegneria civile e in altri settori industriali (la fase di configurazione è volta alle fasi di versionamento e controllo sullo stesso, come si vede dall'immagine).

L'obiettivo è di fornire un modello logico di infrastruttura attraverso controllo, gestione e verifica di tutti i "Configuration Items" (unità di configurazione gestite individualmente, ad es. PC/router/server, ecc.). Un elemento chiave del processo è il Configuration Management Database (CMDB), che viene utilizzato per tracciare tutte le CI e le relazioni tra di loro (tipo: il server A ospita il Servizio B, etc...)

Alcuni benefici dell'implementare il processo di configuration management sono i seguenti:

- Disponibilità di informazioni accurate sull'infrastruttura IT
- Maggiore controllo sulle CI (potenzialmente costose)
- Maggiore aderenza alle leggi (es. numero licenze software)
- Miglior supporto al processo di Incident/Problem Management, soprattutto nella valutazione dell'impatto degli incidenti e nell'analisi della "root cause"

Il tradizionale processo di gestione della configurazione del software (software configuration management/SCM) è considerato dai professionisti come la soluzione migliore per gestire le modifiche nei progetti software. Identifica gli attributi funzionali e fisici del software in vari momenti nel tempo ed esegue il controllo sistematico delle modifiche agli attributi identificati allo scopo di mantenere l'integrità e la tracciabilità del software durante tutto il ciclo di vita dello sviluppo del software.

Il processo SCM definisce ulteriormente la necessità di tracciare le modifiche e la capacità di verificare che il software consegnato finale abbia tutti i miglioramenti pianificati che dovrebbero essere inclusi nella versione.

Identifica quattro procedure che devono essere definite per ogni progetto software per garantire che venga implementato un processo SCM valido:

- *configuration identification*, processo di identificazione degli attributi di un sistema
- *configuration control*, definizione delle fasi richieste per adattare e riformare la configurazione sulla base dei cambiamenti
- *configuration status accounting*, definendo procedure per ottenere e riportare informazioni sullo stato non tecnico di cambiamenti proposti, cambiamenti in attesa di risposta e linee guida
- *configuration audits*, verificando che la configurazione rispetti caratteristiche note secondo test e dimostrazioni

Gli obiettivi principali da raggiungere sono la riproducibilità in ogni ambiente in maniera identica e la capacità di tracciare facilmente le dipendenze usate nelle singole versioni, spesso comparando versioni precedenti e verificando singolarmente i cambiamenti.

(Prendendo ora come riferimento le slide DevOps sullo stesso argomento)

Tradizionalmente, il configuration management era inteso come insieme di attività di tracciamento e controllo usate nei progetti di ingegneria del software finite le operazioni, identificando oggetti correlati, gestendo cambiamenti e variazioni al fine di garantire tracciabilità completa e una buona qualità.

I cambiamenti devono sempre essere ben regolamentati e documentati tra le componenti.

Le variazioni possono essere facilmente recuperate da branch o feature flags e assicurare pratiche CI/CD assicura una qualità e tracciabilità senza avere una control board dedicata.

Il codice di per sé è utile: più utile ancora è la *configurazione*.

Tipici i casi in cui per differenti versioni della JDK si ha un problema legato alle dipendenze delle classi di orari o date. Script di bash possono non essere abbastanza; si preferisce usare una serie di componenti più utili allo scopo della CM (git, package managers, task and build managers, file di Inventory).

Le dipendenze sono importanti, in quanto in futuro potrebbero generare fallimenti nelle build e nella gestione dei progetti, oltre alla compilazione in sé. La gestione delle dipendenze può essere difficile e la configurazione può facilmente fallire: si hanno bisogno quindi di mezzi utili per garantire compatibilità con i tanti *snippet* (frammenti di codice generalmente, spesso sotto forma di *gistable*, frammenti di codice di Git chiamati *Gists* ed *Executable*), che spesso non sono eseguibili. Similmente, anche snippet di codice difficilmente risulta essere funzionante.

La soluzione più semplice è di usare uno script di configurazione di pacchetti o installare solo le dipendenze necessarie tramite Python (*pip*, package manager di Python) o Dockerfiles per creare file utili di configurazione (spesso con vari errori, ad esempio sul nome, normalmente diverso tra pacchetto e modulo), dipendenze transitive, librerie di sistema/di versione. Tuttavia, anche questa soluzione non è il massimo, perché in entrambi i casi si incorre nello stesso problema di creazione di file di configurazione funzionanti (la causa principale: *Module-name != package-name*)

Per ogni piattaforma esistono *package manager* specifici, ad esempio *choco* (Windows), *brew* (Mac), *apt-get/yum* (Linux) oppure anche *library package managers*, come *npm*, *maven* con i pom.xml, *pip*, *nuget*. Essi tuttavia richiedono dipendenze transitive del tipo:

```
Install d3 with bower with grunt with npm with brew.
```

richiedendo una apposita configurazione dei file di sistema, risorse di rete e degli utenti. In questa situazione, si considera di mantenere aggiornato l'inventario di infrastruttura (server, indirizzi IP, ruoli, chiavi SSH, token API, ecc.). La configurazione può essere tanto ampia quanto il codice sorgente e utilizzare script di configurazione e migliaia di cambiamenti al giorno. Ogni cambiamento agli asset deve essere versionato (configuration is code/infrastructure as code).

Un esempio di tool di configurazione: un Playbook Ansible (file di configurazione attraverso un framework che funziona in accoppiata a mezzi come Docker, visto più nel dettaglio in laboratorio apposito):

Ansible - "Playbook" scripts run over ssh

```
---
# The playbook creates a new database test and populates data in the database to test the sharding.
- hosts: $servername
  user: root
  tasks:
    - name: Create a new database and user
      mongodb_user: login_user=admin login_password=${mongo_admin_pass} login_port=${mongos_port} database=test
      user=admin password=${mongo_admin_pass} state=present
    - name: Pause for the user to get created and replicated
      pause: minutes=3
    - name: Execute the collection creation script
      command: /usr/bin/mongo localhost:${mongos_port}/test -u admin -p ${mongo_admin_pass} /tmp/testsharding.js
    - name: Enable sharding on the database and collection
      command: /usr/bin/mongo localhost:${mongos_port}/admin -u admin -p ${mongo_admin_pass} /tmp/enablesharding.js
```

con due esempi di configuration tools:

Puppet - Less hands on, agents run on server.

```
puppet module install thias-mongodb

mongodb::key { ['/etc/mongodb.key']:
  content => 'c9otjehasAvlactocPiphAgC9',
}

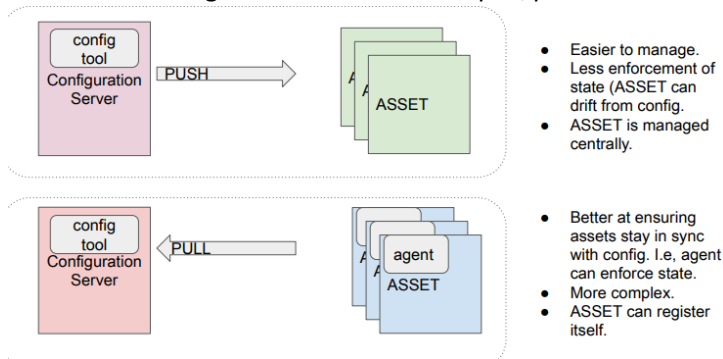
class { '[:mongodb]':
  bind_ip => '0.0.0.0',
  auth => 'true',
  rest => 'true',
  replset => 'rs0',
  keyfile => '/etc/mongodb.key',
  verbose => 'true',
}
```

Chef - "Cookbooks" ruby-based task and build recipes

```
1. knife cookbook site download mongodb

mongodb_user { testuser:
  username => 'testuser',
  ensure => present,
  password_hash => mongodb_password('testuser', 'p@ssw0rd'),
  database => testdb,
  roles => ['readwrite', 'dbAdmin'],
  tries => 10,
  require => Class['mongodb::server'],
}
```

I modelli di configurazione si basano su pull/push:



L'applicazione della stessa operazione più volte dovrebbe portare, a livello di sistema, sempre allo stesso stato, a prescindere dallo stato attuale. Ciò è definito come *idempotenza*. Un esempio di *idempotente/non idempotente*:

NOT IDEMPOTENT

```
ssh-keyscan -H >> ~/.ssh/known_hosts
```

IDEMPOTENT*

```
---
# ansible playbook that adds ssh fingerprints to known_hosts
- hosts: all
  connection: local
  gather_facts: no
  tasks:
    - command: /usr/bin/ssh-keyscan -H {{ ansible_host }}
      register: keyscan
    - lineinfile: name=~/.ssh/known_hosts create=yes line={{ item }}
      with_items: '{{ keyscan.stdout_lines }}'
```

L'aggiornamento dell'infrastruttura segue due pattern:

- *living infrastructure*, mantenendo qualche istanza dedicata. Essa accumula aggiornamenti nel corso del tempo sotto forma di patch, con modifica fisica per venire incontro a nuove esigenze di configurazione (*configuration drift*, ad es. una nuova porta, permessi elevati, archiviazione in cloud). L'infrastruttura è semplice ma potrebbe non essere stabile in termini di resilienza;
- *immutable infrastructure*, istanze a sola lettura costruendo nuove immagini (offline e fornite continuamente). Le immagini sono costruite offline e richiedono apposita configurazione e gli aggiornamenti sono più lenti per questo.

Per costruire immagini si possono usare:

- *snapshot online*, fornendo una nuova istanza, costruendo un filesystem con configurazione ed aggiornamento di pacchetti e salvandone uno snapshot come immagine;
- *roofts construction*, costruendo un filesystem con layer di container e script di configurazione al kernel e serializzazione tramite immagine.

Chiaro quindi come si debba garantire una certa raggiungibilità (in termini di servizi e risorse esterne, come ad esempio un server DB o tramite DNS), disponibilità (l'ambiente fornisce un certo servizio su una certa porta), capacità (l'ambiente supporta operazioni richieste e fornisce supporto di RAM/GPU) e identificabile (oggetti configurati d'ambiente). Oltre a tutto ciò, la password deve essere sicura, evitando inoltre IP non bloccati/controllati nella rete e istanza di admin by default (poche azioni servono per poter prendere il controllo del sistema). In sintesi:

- *Identificare/Identify*: tutte le dipendenze del sistema e dei pacchetti utilizzando i gestori di pacchetti. Tutti i componenti dell'infrastruttura con inventario e variabili negli script di configurazione.
- *Gestire/Manage*: Apportare modifiche agli ambienti informatici con strumenti di configurazione.
- *Variazioni/Variations*: Etichettare le istanze dell'infrastruttura, permettere che le modifiche alla configurazione siano dirette all'implementazione specifica.
- *Qualità/Quality*: Linter (strumenti di tracking di qualità), test in produzione, rollback automatici.
- *Tracciabilità/Traceability*: Tutto il codice dell'infrastruttura viene controllato e tracciato insieme ai codici normali.

I principi della CM (tradotti e lasciati per completezza)

- Versioni aggiunte: tutte le versioni delle dipendenze sono bloccate (o il repository è almeno snapshot?)
- Memorizzare le dipendenze: sei protetto contro l'eliminazione delle dipendenze web? È ancora possibile distribuire quando github è inattivo?
- Nessuna build in produzione: assicurati che non vengano eseguite compilazioni sul sistema remoto
- Usa immagini: usa il cloud dove puoi per una maggiore affidabilità - le build di immagini dove possibile, alcune post-configurazioni fatte da cloud init, anche eseguendo localmente ansible, sono ok.
- Build riproducibili: la nuova build di questo server sarà simile a quella che ho costruito la scorsa settimana (istantanea dei repository, ecc.) - o questo mi può servire o no?
- Memorizza nella cache tutto: dove possibile, gli artefatti della cache nel tuo datacenter (esempio: yum reposync ai vecchi tempi, forse artefatti) e il contenuto arriva da lì, non DDOS di qualcuno .com sito Web con richieste remote di copiare un tarball, poiché questo scala a 1000 server che lo fanno contemporaneamente accenderà le cose o almeno chiamerà gli errori

Abbiamo alcuni esempi di CM in AWS (*Amazon Web Services*):

- *AWS OpsWorks*, gestione delle configurazioni di istanze di Chef e Puppet, piattaforme di automatizzazione delle configurazioni sui server
- *AWS CloudFormation*, linguaggio comune di descrizione delle risorse infrastrutturali usando un file di testo modellando le risorse tra applicazione ed account

Container e Docker

Originariamente, ogni applicazione viveva su un server fisico dedicato. Vi erano grosse limitazioni, date da tempi di deploy molto lenti, con molte risorse sprecate e difficoltà di scalabilità e migrazione applicazioni. Le risorse sono disaccoppiate ed un servizio gestito tramite software è esposto alle API, virtualizzando servizi di calcolo, rete e memorizzazione. In questa era si sviluppano le Virtual machine che permettono ai server fisici di contenere più di un'applicazione. Anche la scalabilità ne ottiene dei benefici, così come la possibilità di gestire al meglio le risorse fisiche riducendo i costi. La nascita dell'IaaS permette di avere Macchine Virtuali nel cloud.

Le macchine virtuali possono avere alcune limitazioni:

- Ogni Virtual Machine necessita di molte risorse: CPU, Spazio disco, RAM ed un intero SO completo
- Un Sistema Operativo completo implica molte risorse sprecate
- La portabilità non è garantita

Si cerca quindi di avere un *pool* di risorse computazionali in cloud, tale da ridurre i costi (sia per il provider che per il consumer) e per allocare dinamicamente le risorse sulla base dei client collegati in maniera elastica. L'idea è di astrarre l'hardware rispetto al sistema operativo, consentendo a più sistemi operativi di girare in simultanea sullo stesso server, con un ambiente isolato (*virtual machine*) per tutti i sistemi, non accedendo a risorse esterne se non tramite interfacce di rete (applicando eventuali patch a tutte le VM).

Si ottiene il disaccoppiamento dell'hardware rispetto al software, tramite uno strato di *HyperVisor*, che si trova tra l'hardware e l'SO della VM, fornendo ambienti di esecuzione (es. VMWare ESX Server, Linux KVM, Hyper-V, XenServer). L'HyperVisor è il responsabile primario di creazione ed esecuzione di VM isolate, partizionando e condividendo dinamicamente le risorse fisiche dell'hardware tra le diverse VM.

Ce ne sono due tipi:

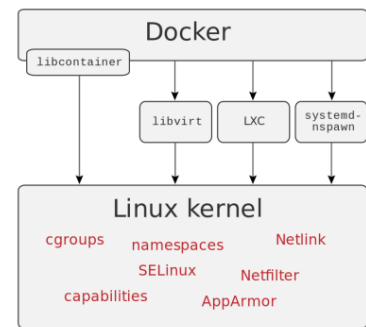
- Bare-metal HV o Type 1, che gira direttamente sull'hardware ed ospita le VM (es. VMWare ESX), non richiedendo un SO installato
- Type 2, gira direttamente sul SO (VMWare, VirtualBox)

La macchina fisica su cui avviene la virtualizzazione è l'*host*, definito anche come *compute node*.

La macchina virtuale stessa è la *guest*, in cui gira un SO completo e le app utente, memorizzata fisicamente come insieme di file (disco virtuale, file di configurazione, di paging, file del BIOS, di log, ecc.).

L'idea di base dei container è quella di utilizzare il kernel del sistema operativo della macchina host per eseguire tanti root file system. Ogni root file system è chiamato *container* e su ogni container abbiamo:

- processi (di cui uno principale)
 - memoria
 - stack di rete



Diamo quindi la definizione formale di *container*:

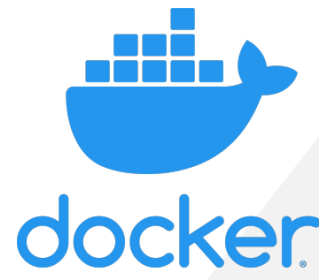
“Applicazione completamente isolata che gira nativamente nel kernel della macchina host e che offre completo isolamento da altri container e dalla macchina host stessa.”

Un container può accedere ai file ed alle porte di rete della macchina host solo se configurato, infatti:

- Esegue applicazioni direttamente nel kernel dell'host. Un container non ha bisogno di un HyperVisor
- I container che girano sull'host condividono il kernel dell'host
- Virtualizzazione del S.O. e non dell'hardware
- Ogni container esegue un singolo componente software (es. un container che esegue Apache HTTP Server)

Essi sono creati a partire da *immagini*, contenenti informazioni circa il codice eseguibile (Python, Java, ecc.) e l'ambiente necessario all'esecuzione (con file di configurazione, variabili d'ambiente, runtime e librerie usate). Di fatto i container sono *leggeri e veloci*, non avendo bisogno di un SO dedicato e necessitano di meno risorse fisiche, girando facilmente in maniera portabile ed efficace.

Introduciamo quindi *Docker*, inteso come una piattaforma aperta per lo sviluppo, la spedizione e l'esecuzione di applicazioni. Docker consente di separare le applicazioni dall'infrastruttura, in modo da poter distribuire rapidamente il software, gestendo l'infrastruttura nello stesso modo in cui si gestiscono le applicazioni. Sfruttando le metodologie di Docker per spedire è possibile ridurre in modo significativo il tempo che intercorre tra la scrittura del codice e la sua esecuzione in produzione.



Docker offre la possibilità di impacchettare ed eseguire un'applicazione in un ambiente isolato, quindi il container. L'isolamento e la sicurezza consentono di eseguire molti container contemporaneamente su un determinato host. I container sono leggeri perché non hanno bisogno del carico supplementare di un Hypervisor, ma vengono eseguiti direttamente nel kernel della macchina host (potendo così eseguire molteplici container in una macchina sola).

Esso fornisce strumenti e una piattaforma per gestire il ciclo di vita dei container:

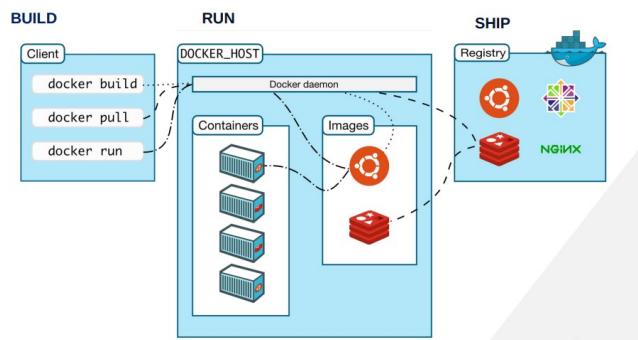
- Sviluppo dell'applicazione e i suoi componenti di supporto utilizzando i container.
- Uso del contenitore come unità di distribuzione e test dell'applicazione.
- Quando si è pronti, distribuzione dell'applicazione nell'ambiente di produzione come contenitore o servizio organizzato all'esecuzione.

Questo funziona allo stesso modo sia che l'ambiente di produzione sia un data center locale, un provider cloud o un ibrido dei due. Il Docker Engine è costituito da un server (con namespaces del Kernel Linux) e API per interfacciarsi con altri server, container e client CLI.

Docker utilizza due tecnologie contenute nel kernel di Linux per isolare i container:

- namespaces, fornendo un ambiente isolato per l'esecuzione dei processi, con accesso per loro limitato ed isolando ogni aspetto usando IP, rete, mount per host-name e domain-name
- Control Groups (cgroups), usati per limitare le risorse disponibili per un container

L'architettura ha questa struttura:

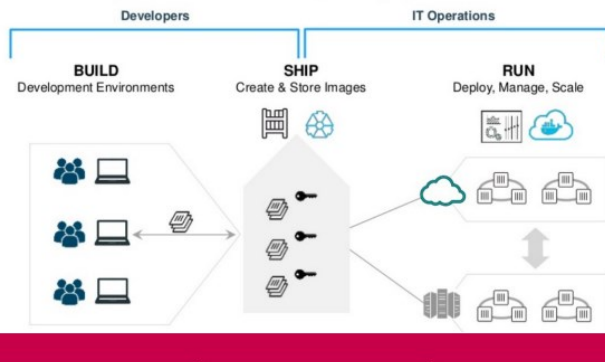


L'intera applicazione nel container è rappresentata da una *Docker Image*, file immutabile che rappresenta l'istantanea di un container; generalmente le immagini sono costituite da strati di altre immagini e vengono create con l'operazione di build per un file descrittore chiamato *DockerFile*, che possono essere condivise e scaricate da un registry.

Un esempio di *DockerFile* (dove ogni riga rappresenta un layer salvato e identificato tramite chiave, usato per eseguire il comando successivo):

```

11 lines (6 sloc) | 231 Bytes
1 FROM tiangolo/meinheld-gunicorn-flask:python3.7
2
3 COPY /dist/flaskr-1.0.0-py2.py3-none-any.whl /app
4
5 RUN pip install /app/flaskr-1.0.0-py2.py3-none-any.whl
6
7 ENV FLASK_APP=flaskr
8
9 RUN flask init-db
10
11 ENV APP_MODULE flaskr:create_app()
    
```



I container partono da un'immagine immutabile in ambienti diversi con nuove risorse.

(Anche su Mega ho segnato l'ultima parte di lezione come laboratorio, il prof ha detto che avrebbe poi condiviso i comandi. Tuttavia, non ha messo nulla e non fa parte ufficialmente di laboratorio. Ai fini della teoria e dei lab non si aggiungono altri dettagli).

Laboratorio 7: Ansible

Prerequisiti

- Vagrant
- VirtualBox (non serve per forza la versione 6 come indicato nel file, a me andava comunque)

Preparazione

- Posizionarsi in una cartella a piacere.
- Eseguire nella powershell (Windows) o shell (linux) il comando `◦ vagrant init ubuntu/trusty64` • Far partire la macchina virtuale appena creata con il comando `◦ vagrant up`

Importante: In caso di errore assicurarsi che la versione di virtualbox sia ≤ 6.0 e che tutti i moduli siano attivi

- Connettersi alla macchina virtuale tramite il comando `◦ vagrant ssh` • inserire la seguente password quando e se richiesto `◦ vagrant`

- Dalla shell della macchina virtuale digitiamo i seguenti due comandi

```
$ sudo apt-get update
```

```
$ sudo apt-get -y install git make vim python-dev python-pip libffi-dev libssl-dev libxml2-dev libxslt1-dev libjpeg8-dev zlib1g-dev
```

- Quindi se tutto é andato bene digitare `sudo apt-get install ansible`

Va quindi creata in un nuovo file *inventory* la chiave privata come:

```
node0 ansible_ssh_host=192.168.56.10 ansible_ssh_user=vagrant
ansible_ssh_private_key_file=./keys/node0.ke
```

modificando anche il file *hosts* e mettendo `192.168.56.10` come indirizzo.

Successivamente si installa *nginx* di Apache come webserver ed avviato:

```
◦ ansible 192.168.33.10 -s -m apt -i inventory -a 'pkg=nginx state=installed
update_cache=true'
```

- Avviamo il webserver sul nodo

```
◦ ansible all -s -m shell -i inventory -a 'nginx'
```

I file *inventory* contengono una lista di tutti gli oggetti utili all'implementazione; il file di lab ne configura alcune buone. Successivamente discutiamo i *playbook*, lista di host o gruppi in formato YAML, tipo di file usato nelle configurazioni serializzabili simile a JSON, in cui si definiscono dei *roles/ruoli* utili alla configurazione di attività nello stesso ambito. Vi sono anche i *facts*, variabili utili a contenere le informazioni dei sistemi target.

Nei moduli va sempre configurato lo stato, come ad es:

es. `ansible host1 -m pkg -a "name=gparted state=present"`

avendo la possibilità di variare dinamicamente gruppi in base all'OS.

Vogliamo idealmente configurare un nodo CentOS, piattaforma Linux di tipo Enterprise.

Ci creiamo una cartella apposita su cui inizializziamo vagrant e anche in SSH. All'interno di questa cloniamo un progetto Tomcat e inseriamo nel file hosts la chiave e l'indirizzo di avvio 192.168.56.10.

```
node1 ansible_ssh_host=192.168.56.11 ansible_ssh_user=vagrant
```

```
ansible_ssh_private_key_file=/home/vagrant/keys/node1.key
```

Eseguendo il playbook con: `ansible-playbook -i hosts site.yml` verranno eseguiti i ruoli selinux e tomcat nel gruppo tomcat-servers, in cui si specifica che il nodo 192.168.56.10 è attivo con i due ruoli utilizzati. Facendo dei ping o dei curl sul link specificato, vediamo quindi che risulta attivo con quella specifica configurazione.

Setting di una repo: https://docs.ansible.com/ansible/latest/user_guide/sample_setup.html#sample-directory-layout

Ansible Built-in Modules: <https://docs.ansible.com/ansible/latest/collections/ansible/builtin/#description>

Ansible Examples: <https://github.com/ansible/ansible-examples>

CM Workshop e Ansible: <https://github.com/CSC-DevOps/Course/blob/Spring2020/Workshops/CM.md>

Caso d'uso per riferimento di Assignment 3: ASPI/Autostrade per l'Italia

Si descrive un processo di automazione attraverso filiere di automazione di sviluppo e rilascio di applicazione per la CI, partendo da tecnologie e linguaggi programmazione comuni. Nei casi di utilizzo infrastrutturali ed IT, tutto lo sviluppo deve essere guidato e testato fase per fase. Il focus principale in merito alle caratteristiche:

ASPI SecDevOps

Lesigenza

- Autostrade per l'Italia (ASPI) ha avviato da tempo un processo di trasformazione digitale
- Dispone di una propria divisione IT
- Sviluppa internamente le proprie soluzioni IT, collaborando anche con altre aziende partner (come Engineering)
- Ogni soluzione IT può essere costituita da più componenti tecnologicamente eterogenee (filiere): app mobile Android/iOS, soluzioni Java, Python, Kotlin, Cobol, Angular, ecc ecc
- Ogni soluzione IT dispone di più ambienti (sviluppo, test, quality, collaudo), a supporto delle attività di test e collaudo che precedono il rilascio definitivo nell'ambiente di produzione
- Ha predisposto un'infrastruttura affinché tutti i gruppi di sviluppo interni ed esterni lavorino secondo i medesimi standard di qualità, grazie all'uso di strumenti comuni, configurati opportunamente
- L'infrastruttura e gli standard di qualità sono definiti da un ufficio apposito, guidato da Lorenzo Carlà, con l'ausilio di personale Engineering
- Il team che si occupa di sviluppare le configurazioni e i servizi a supporto dei team di sviluppo è denominato Team SecDevOps: questo nome pone esplicitamente l'accento sul tema della sicurezza, promuovendo le più moderne pratiche di sviluppo, che vanno sotto il nome di DevOps, in cui la standardizzazione e l'automazione giocano un ruolo centrale (la seconda non sarebbe possibile senza la prima)
- Gli ambiti di standardizzazione e automazione? ITS, VCS, TEST, ARTIFACT REPOSITORY, BUILD AUTOMATION, CI, CD, CM, per citarne alcuni tra i più importanti

ITEXL
toDigital

autostrade per l'Italia

ENGINEERING



Si descrivono nell'ordine:

- ITS, tracciando le attività tramite sprint e in modo bidirezionale (dalle issue al codice e viceversa). Il team DevOps usa strumenti come *Asana* per pianificare le attività e *GitLab Issues* per tracciare issue ed attività stesse
- VCS, che ospita tutto il parco software ASPI e le pipeline di sviluppo con relativi sorgenti secondo le pratiche CI/CD. Si ha quindi un accesso ITS integrato, agendo direttamente sui sorgenti ed operando sulle varie merge-request e sui vari branch. In maniera semplice si vedono anche workflow e altro.
- Ciascun elemento del team progetta ed esegue test secondo livelli e tipi stabiliti, coprendo tutti i livelli di unità, integrazione e collaudo. I test funzionali garantiscono qualità e sicurezza del codice

Nell'esecuzione dei test si calcola direttamente il code coverage e attraverso la piattaforma proprietaria Salesforce si integra la copertura. I test di sistema sono eseguiti in automatico, con strumenti quali Robot Framework, riportando semplicemente i bug in report estraibili.

La sicurezza viene controllata a livello di librerie e vulnerabilità, a livello di sicurezza e del codice sorgente, opponendo il codice a test di qualità con strumenti come PMD/SonarQube, con esecuzione automatizzata

- Artifact Repository, fornita da Nexus OSS, in cui ogni filiera tecnologica ospita artefatti software, basato su soluzioni opensource come Red Hat
- Build Automation, in cui filiere come Maven, Ant, Gradle e Fastlane (su mobile) vanno per la maggiore, usando strumento di container Docker dedicati; anche la produzione di pacchetti software è integrata anche in cloud grazie ad AWS, usabili sia localmente che centralmente
- CI/CD tramite Jenkins, garantisce ulteriormente il rispetto degli standard e delle best practice, imponendo un feedback loop entro cui operano tutti i team applicativi e altre figure coinvolte. Si adotta il linguaggio Groovy in un framework condiviso, notificando modifiche e notizie e creando quality gate fase per fase
- CM, con Dockerfile che descrivono i pacchetti, file YAML che descrivono le configurazioni e template per ambienti AWS

Caso d'uso: Chili e ItsArt

Link di riferimento per i casi d'uso (di cui il prof si è occupato personalmente).

1)

<https://it.chili.com/>

Le attività sono misurate con storypoint e Sprint plan eseguiti attraverso Atlassian e Jira e le riunioni vengono fatte con Google Meet, analizzando il prodotto attraverso tutte le sue fasi. Per ogni sprint si vuole arrivare ad uno story point e l'evoluzione è documentata dal burndown chart. Ogni componente rappresenta con nota tecnica le attività tra oggetti e strumenti presenti, commentando segnalazioni e decidendo a chi assegnare le issue tramite Jira.

Le attività sono collegate tramite branch e rilasciate tramite BitBucket. I branch possono essere creati con lo stesso nome delle attività e per ognuna si ha una Definition of Done.

Tramite BitBucket ci sono dei branch di tutto il codice e si usa il ramo di devel per le attività di completare per rilasciare sulla release sfruttando le feature.

Ogni file è composto da file e Dockerfile e si usa un frontend per raccogliere riferimenti alle attività, loro cambiamenti e relativo processo di build.

L'analisi statica è svolta con SonarQube e si usa anche Jenkins per gestire il codice e relativo build con Gradle.

2)

<https://www.itsart.tv/it/>

Nuovamente come strumento si usa Jira e Jenkins per gestire i rilasci sull'ambiente di stage

Tutti gli ambienti di integrazione sono aggiornati facilmente e riconfigurati con immagini Docker tramite ricreazione da fase di stage o strumenti di backup e restore si recuperano cambiamenti.

L'ambiente di devel viene recuperato facilmente tramite Kubernetes e ArgoCD e conseguente rilascio nella build di stage. Il rilascio avviene con la cartella deployment.

Ogni test con Jenkins capisce se ci sono stati dei problemi (in questo caso a visualizzazioni di film che non vanno bene, notificando tutti gli sviluppatori). I rilasci sono confidenti con centinaia di test con fasi di Scrum normalmente in 2 giorni e integrazioni in settimana.

Robot Framework e Laboratorio 8: Robot Framework

Robot Framework è un *automation framework* open source, usato per test automation e robotic process automation (RPA), usando robot software per eseguire test ripetitivi configurato con appositi file, chiamati *robot*. Ha una sintassi apposita e una API che permette l'estendibilità con Python/Java; è inoltre hostato su GitHub con documentazione, codice sorgente ed issue tracking. La licenza è Apache e usa l'approccio *keyword-driven* test, cioè separa il design dei casi di test dall'esecuzione, usando parole specifiche per test specifici, ciascuno con i suoi passi.

Ha un supporto apposito alle variabili e alle keyword di riferimento, con un buon supporto della community in modo *full stack*, dunque sia frontend che backend.

Per la parte di laboratorio (presente dalla slide 12 in poi del pdf *RFTest*), si offre una sintesi come al solito:

Si crea una cartella temporanea chiamata *robot* in una posizione preferita.

Dentro la cartella Robot, aperta da VS Code e creiamo il file *prova.robot*.

E si incolla come testo:

```
***Test Cases***
```

```
Test hello world
```

```
Log To Console ciao mondo
```

Robot capisce che gli asterischi introducono i casi di test; ogni due spazi nel testo facciamo terminare la keyword e vengono messi nuovi argomenti. Cliccando su Terminal dalle opzioni di VS Code, si apre il terminale da dentro la cartella *robot* e avviamo il comando *pip install robotframework* se non è già stato installato.

Mettiamo il comando *robot prova.robot*

In questo mondo viene avviato il primo test e il colore cambia in verde/rosso a seconda di tutti i test della suite. L'output è in formato log in HTML.

Ora andiamo a clonare, sempre dal terminale in VS Code il progetto:

<https://github.com/robotframework/QuickStartGuide/blob/master/QuickStart.rst>

con il solito *git clone*.

Prendiamo come esempio un'applicazione di login in Python con accesso e creazione di utenti con una password. L'esempio è al link:

<https://github.com/robotframework/QuickStartGuide/blob/master/QuickStart.rst>

Per eseguire correttamente il test occorre installare il modulo *docutils* tramite pip:

```
pip install docutils
```

Ora semplicemente si entra nella cartella QuickStartGuide si esegue:

```
robot QuickStart.rst
```

Vengono eseguiti 5 test per una suite per controllare quanto prima descritto.

Eventualmente può essere creato un *venv*, virtual environment che permette di operare in una cartella come fosse uno spazio isolato dalle altre, con i suoi pacchetti Python.

Similmente, si esegue e viene visto il risultato tramite browser.

Torniamo quindi nella cartella *robot* e cloniamo il progetto:

<https://github.com/robotframework/RobotDemo.git>

MTSS semplice (per davvero)

Per Linux, si attiva il venv così:
`python3 -m venv robotdemo`
`source ./robotdemo/bin/activate`

Per Windows, esiste un file `Activate.Ps1` dentro la cartella `robotdemo/bin`. Ci si sposti lì dentro e si provi ad eseguire lo script con `.\Activate.ps1`
Ora il prompt ha (robotdemo) davanti.
Installiamo quindi le dipendenze del progetto (rispostandosi in RobotDemo) con:
`pip install -r requirements.txt`

Sempre per Windows bellissimo potrebbe dare:
ERROR: Failed building wheel for pynacl
Soluzione utile → Installare libsodium.dll seguendo:
https://py-ipv8.readthedocs.io/en/latest/preliminaries/install_libsodium.html
Dopo un po' di prove, ho trovato il link a cui seguire altri passaggi e anche quello indicato:
<https://stackoverflow.com/questions/65517524/pynacl-building-problems>
Non ho tuttavia risolto il problema purtroppo; come per Postgre, Windows si rivela meraviglioso e lascio perdere, dopo diverso tempo di prove senza soluzioni.

I casi di test sono i calcoli eseguiti con varie operazioni da una calcolatrice, aspettandosi un certo risultato.
Eseguiamo un esempio di test con
`robot keyboard_driven.robot`
Poi il prof esegue altri casi di test.
Eseguito il comando `deactivate` si disattiva il venv e si clona la repo:
<https://franz-see.github.io/Robotframework-Database-Library/>

Similmente a prima si eseguono:
`source robotdb/bin/activate`
`pip install -r requirements.txt`

Andiamo nella cartella test ed eseguiamo `robot SQLite3 DB_Tests.robot`
Successivamente abbiamo bisogno della libreria RequestsLibrary, visibile anche al link:
<https://marketsquare.github.io/robotframework-requests/doc/RequestsLibrary.html>

La installiamo con: `pip install requestslibrary`
Creiamo il file `request.robot` e incolliamo il contenuto (Esso esegue un HTTP a Google e verifica il risultato sotto forma di JSON come `response`):

```
* Settings *
Library          RequestsLibrary

* Test Cases *
Quick Get Request Test
    ${response}= GET https://www.google.com

Quick Get Request With Parameters Test
    ${response}= GET https://www.google.com/search params=query=ciao expected_status=200

Quick Get A JSON Body Test
    ${response}= GET https://jsonplaceholder.typicode.com/posts/1
    Should Be Equal As Strings    1    ${response.json()[id]}
```

Esempio di test con Docker-Selenium e Robot Framework

Prendiamo l'esempio di *dw-demo-app* viste altre volte:

<https://github.com/dduportal/dw-demo-app>

Si avvia come container Docker, tramite *docker container ls*.

Si vede che è in esecuzione e dunque è raggiungibile tramite porta di rete (8080).

Le macchine possiedono un motore grafico, esempio *ChromeWebDriver*.

Il framework adottato è Selenium, in questo caso con Docker:

<https://github.com/SeleniumHQ/docker-selenium>

Selenium è la libreria di test più usata in ambito di applicazioni web:

<https://www.selenium.dev/>

Lo scopo ultimo è di usarlo con Robot Framework:

<https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html>

Vogliamo verificare che il secondo paragrafo dell'applicazione riporti *Hello, Stranger!* come si vede dalla foto.

I comandi per il setup dell'esempio sono:

`git clone https://github.com/dduportal/dw-demo-app`

`cd dw-demo-app`

`mvn package`

`docker run -d -p 8080:8080 dw-demo-app:latest`

The ID is: 7
The content is: Hello, Stranger!



Sfruttando l'interfaccia del browser, con il seguente plugin, legge il DOM e interagendo con l'applicazione.

Chromedriver permette di, una volta settato nella cartella in cui si trova tra le variabili d'ambiente

Il riferimento è al link <https://chromedriver.chromium.org/downloads> (scaricato per la versione Chrome di riferimento). Una volta correttamente configurato come condiviso al link di esempio, si possono settare specifiche proprietà di creazione oggetti, gestione dei cookies e lancio dell'URL.

Nell'esempio si fa riferimento ad Eclipse:

<https://www.browserstack.com/guide/run-selenium-tests-using-selenium-chromedriver>

Ora si crea l'ambiente di riferimento e si usa il file di attivazione:

`python3 -m venv robot`

`source robot/bin/activate`

Creando il file di testo con estensione *.robot* si crea correttamente il file di test.

Esso si compone in questo modo, aspettandosi che la pagina si colleghi all'url corretto e la pagina contenga l'immagine corretta:

```
***Settings***
Library          SeleniumLibrary

Test Setup       Open Browser To Argument  url=${HOME_URL}

Test Teardown    Close Browser

*** Variables ***
${DEFAULT_BROWSER_OPTION}  add_argument("--no-sandbox");add_argument
${BROWSER}                 chrome
${SELENIUM_HUB}            false
${HOME_URL}                http://localhost:8080

*** Test Cases ***

Test Hello
    Wait Until Page Contains Element  locator=//p[@class='ng-binding']
    Element Should Contain  locator=//p[@class='ng-binding']
```

L'esempio di log test è il seguente:

Test Execution Log

SUITE Test	00:00:12.155
Full Name:	Test
Source:	/tmp/robot/test.robot
Start / End / Elapsed:	20220610 11:24:06.609 / 20220610 11:24:18.764 / 00:00:12.155
Status:	1 test total, 1 passed, 0 failed, 0 skipped
<hr/>	
TEST Test Hello	00:00:11.876
Full Name:	Test.Test Hello
Start / End / Elapsed:	20220610 11:24:06.887 / 20220610 11:24:18.763 / 00:00:11.876
Status:	PASS
+ SETUP Open Browser To Argument url=\${HOME_URL}	00:00:01.726
+ KEYWORD SeleniumLibrary.Wait Until Page Contains Element locator=//p[@class='ng-binding']	00:00:00.031
+ KEYWORD SeleniumLibrary.Element Should Contain locator=//p[@class='ng-binding'], expected>Hello, Stranger	00:00:00.031
+ KEYWORD BuiltIn.Sleep 10	00:00:10.001

Provando ad esempio a scrivere un'altra cosa sul parametro *expected*, parte dopo a quello che si vede dall'immagine con `***Settings***` e `***Test Cases***`, il test fallirà; è possibile anche impostare il tempo di avvio e collegamento alla pagina.

L'esempio di avvio, su localhost:4444 e su localhost:7900, avvia un sistema operativo apposito di Chrome che su un container simula un container con una serie di apposite funzionalità.

Il test usa un container apposito, ad un altro indirizzo e verifica sia correttamente raggiungibile da dentro Selenium; al di fuori del container. Su quelle porte è installato noVNC, che sarebbe un ITS e accede all'applicazione testandola sull'ip 8080.

Per avviarlo usa il comando:

```
docker run -d -p 4444:4444 -p 7900:7900 --shm-size="2g" selenium/standalone-chrome:98.0.4758.102-chromedriver-98.0.4758.102-grid-4.1.2-20220217
```

Durante questo test, Chrome informa che l'applicazione è controllata dal driver; il test è eseguito in modalità headless, cioè si esegue il test su una pagina web senza renderizzarla/caricarla, e dunque il browser reagisce ai test più velocemente restituendo comunque il risultato (tramite user scripting). Per i curiosi, un esempio di interazione con i driver:

https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/

L'idea grafica è:

