Appunti Tecnologie Open-Source

Michele Veronesi

A.A. 2020/2021

Contents

Questi appunti coprono la sola parte di teoria del corso.

1 Issue Tracking System - ITS

Issue: criticità, evento da gestire
Tracking: registrare, lasciare traccia

1.1 Definizione

Un Issue Tracking System (ITS) è un pacchetto software che gestisce e mantiene una lista di criticità (issue), come richiesto da un'organizzazione.
Questo tipo di sistema è spesso usato nei call center di supporto delle aziende per creare, aggiornare e risolvere i problemi segnalati dai clienti o dagli impiegati.
In sostanza, è uno strumento che semplifica 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...
Ogni singola "attività minima" (work item) del progetto è gestita mediante un workflow e mantenuta all'interno di un'unica piattaforma e un'unica repository.

1.2 A cosa serve

- Condividere le informazioni tra team di sviluppo, project manager e cliente, infatti offre un unico repository in cui trovare le informazioni e un sistema di notifica.
- Implementare il processo di misurazione della qualità del progetto
- Avere un'istantanea dello stato del progetto: attività da fare, in corso d'opera, completate.
- Decidere cosa rilasciare e quando rilasciare: i work item possono essere assegnati ad una versione
- Assegnare una priorità alle attività
- Avere una chiara assegnazione delle responsabilità sulle attività: ogni work item riporta incaricato e assegnatario
- Tiene la memoria storica di tutti i cambiamenti

1.3 Workflow

Il workflow è un insieme di stati e transizioni che un work item attraversa durante il suo ciclo di vita. Viene associato ad un progetto e ad uno o più tipi di work item. Permette di registrare tutte le transizioni e i cambi di stato.

1.4 Collegamenti

Permettono di mettere in relazione vari work item, anche di differenti tipi (attività, sotto-attività, requisiti, casi di test, ...), sono bidirezionali e vengono registrati come criterio di ricerca. Questo permette di verificare la presenza o meno di relazione tra i work item (per esempio vedere se un requisito è coperto da casi di test).

1.5 Funzionalità

- Ricerca avanzata dei work item
- Salvataggio di ricerche
- Esportazione
- Notifiche
- Bacheche o board
- Reporting
- Dashboard
- Definizione di Road map e Release Notes
- Integrazione con il Source code management
- Integrazione con l'ambiente di sviluppo

Filtri

I filtri permettono di ricercare i work item in base ai campi, possono essere salvati per facilitare le ricerche più recenti, i risultati possono essere esportati e sono la base per creare report, board e dashboard.

Board o bacheche

Permette di visualizzare i work item di uno o più progetti, offrendo un modo flessibile e interattivo di visualizzazione, gestione e visualizzare dei dati di sintesi sulle attività in corso.
È configurata e visualizza i work item ricercati con un filtro.
Permette di interagire velocemente con i work items (avanzare di stato, modificare alcuni campi).

1.6 Configurazione e utilizzo

1. Identificare i processi richiesti per la gestione del progetto: vincoli imposti dal cliente, procedure e best practices definiti dai framework di qualità presenti in azienda o imposti dal cliente.
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 processo.

- Il manager del ITS:
  - crea un nuovo progetto
  - definisce il processo da seguire (tipi di work item, campi custom, work flow, collegamenti), seleziona il modello di stima, differenti board e report per processo
  - Aggiunge gli utenti e assegna ruoli/permessi
- Il manager (capo progetto):
  - Definisce le versioni (release)
  - Definisce le componenti del progetto
  - Definisce i lavori da svolgere (backlog): priorità, assegnatario e stima
  - Definisce la prima iterazione
  - Monitora l'avanzamento e il completamento delle attività (filtri, board, dashboard, report)
  - Definisce le nuove versioni
  - Definisce le nuove iterazioni
  - Definisce, aggiorna e monitora le attività (priorità, verifica stima/consuntivo)
  - Produce i report richiesti dal cliente (p.es. Calcolo SLA, release log, ...)
- 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
  - Completano tutte le attività presenti nell'iterazione
  - Effettuano il rilascio

1.7 Vantaggi ITS

- 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
- Ridurre le spese e gli sprechi

2 Source Code Management - SCM/VCS

2.1 Definizione

I source code management systems (aka version control systems) sono sistemi software che registrano tutte le modifiche avvenute ad un insieme di file. Inoltre permettono la condivisione di file e modifiche, offrendo funzionalità di merging e tracciamento.

2.2 Vantaggi

In un team di sviluppo, un SCM permette di:

- collaborare in modo efficiente sul codice di un prodotto, facilitando l'individuazione e la correzione dei conflitti e la condivisione di commenti e documentazione
- tracciare ogni modifica (storia del prodotto): fornisce una lista completa dei cambiamenti apportati ad un file, offrendo la possibilità di effettuare un rollback ad una versione precedente
- lavorare senza interferenze in differenti rami di sviluppo (*branching*); le modifiche avvenute in un branch possono confluire nel master con un merge
- tracciabilità: tutte le modifiche possono fare riferimento ad un'attività contenuta nel issue tracking system; in ogni istante è possibile capire che attività sono state effettuate in una specifica versione

2.3 Utilizzo

Con il termine *diff* si indica la differenza tra due versioni di uno stesso file (i.e. cambiamenti tra le righe del file). Un insieme esplicitamente validato di *diff* è detto *commit*. Un *commit* è difatti una nuova versione della codebase, e può esistere localmente o in remoto. L'ultimo *commit* nella history viene chiamato *HEAD*. Questo viene utilizzato per verificare le differenze tra la codebase locale e quella remota.
Un *branch* è un puntatore ad un singolo commit. L'*HEAD* fa parte del *master branch*, ogni altro *branch* può integrarsi con il *master* a seguito di un *merge*.
L'attività di merging può essere gestita attraverso una *Pull Request*, cioè una procedura di discussione e correzione a seguito della quale una branch può essere integrata con il master.
Un *fork* è una copia di una codebase, e permette di lavorare su tale copia liberamente, senza dover richiedere permessi di modifica alla repository originale. Una volta terminate le modifiche è possibile integrarle nella versione originale attraverso una Pull Request.

Gitflow

Il *Gitflow* è un'estensione del pattern *Feature Branch Workflow*. Pensato per progetti di larga scale, il Gitflow prevede la strutturazione dello sviluppo su più branch, assegnando uno specifico ruolo ad ogni branch.

2.4 Tipi di VCS

Locali

Sono i più vecchi, registrano solo la storia dei cambiamenti utilizzando un *version database* dove viene registrata tutta la storia dei file e salvando sul disco una serie di patch, rappresentanti il cambiamento tra una versione e l'altra. Tuttavia non gestiscono la condivisione.

Centralizzati

Più recenti e molto diffusi, aggiungono la condivisone dei file con un version database centralizzato (singolo punto di rottura). Ogni sviluppatore ha solo una versione in locale. Facili da apprendere.

Distribuiti

Simili ai centralizzati, ma il database è distribuito ad ogni noto (anche i client hanno la storia completa dei file). Sono i più diffusi poiché hanno una serie di vantaggi:

- quando il nodo centrale non è disponibile, è possibile continuare a lavorare localmente registrando i cambiamenti
- hanno una migliore risoluzione dei conflitti che favorisce la collaborazione
- permettono di impostare diversi flussi di lavoro (branch)

Tuttavia l'apprendimento è più complesso rispetto ai centralizzati.

3 Il framework Scrum

3.1 Definizione

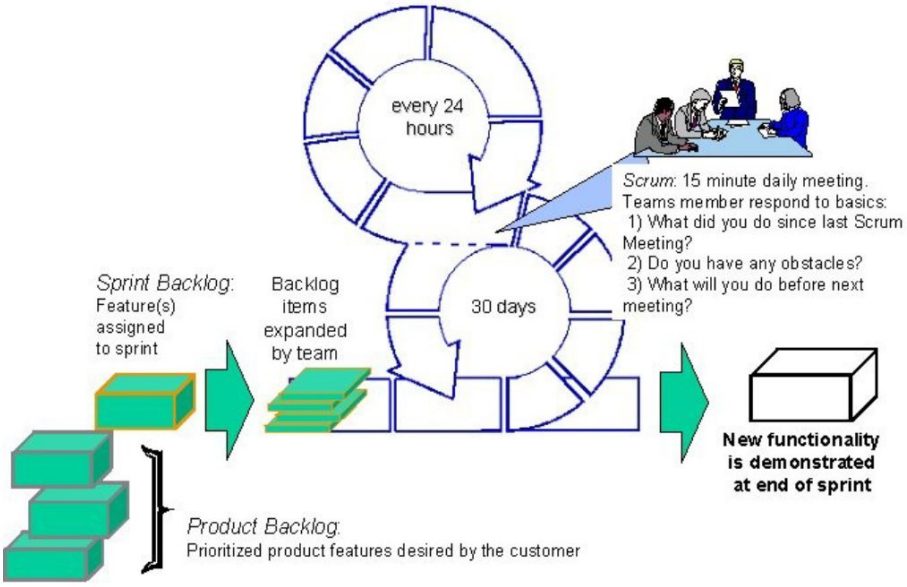
Scrum è un *processo agile* che nasce per lo sviluppo di progetti complessi (difficili da definire e da risolvere) e che permette di concentrarsi sulla consegna del maggior valore business nel minor tempo possibile.
Permette di ispezionare software funzionante rapidamente e ripetutamente (ogni 2-4 settimane).
Il business stabilisce le priorità. I team si organizzano per scegliere la strada migliore per consegnare le funzionalità a priorità più alta.
Ogni due settimane o un mese, chiunque può vedere il software funzionante e decidere se lasciarlo così o se migliorarlo facendo un altro sprint.

3.2 Caratteristiche

- Leggero
- Facile da capire
- Difficile da padroneggiare

Si basa su tre pilastri:

- Trasparenza: linguaggio comune per una conoscenza condivisa, definition of done
- Controllo: ispezioni pianificate per prevenire variazioni indesiderate
- Adattamento: aggiustamenti per minimizzare ulteriori deviazioni tramite feedback continuo



Durante lo sprint c'è un daily meeting della durata di 15 minuti in cui si discute l'avanzamento dei lavori locale (ovvero cosa si è fatto ieri e cosa si farà oggi). Inoltre, se qualcuno ha incontrato problemi svolgendo dei work item deve segnalarlo.

- Gruppi che si auto-organizzano
- Il prodotto evolve attraverso "sprint" mensili (o comunque di durata fissa)
- I requisiti sono trattati come elementi di una lista detta "product backlog"
- Non vengono prescritte particolari pratiche ingegneristiche
- Si basa sull'attività empirica cioè la conoscenza si basa sull'esperienza e le decisioni si basano su ciò che si è conosciuto
- Processo iterativo e incrementale per ottimizzare il controllo dello sviluppo e il controllo del rischio

Sprint

I progetti Scrum progrediscono attraverso una serie di "sprint", analoghi alle iterazioni della extreme programming (altra pratica agile). Hanno una durata tipica di 2-4 settimane costante, poiché favorisce un ritmo migliore.
Durante lo sprint il prodotto viene progettato, realizzato e testato, e durante questo si celebrano tutti gli eventi.
Non si cambia durante lo sprint: è necessario che lo scrum master tenga i cambiamenti fuori dallo sprint. Nonostante lo sprint backlog possa essere modificato e rinegoziato tra team di sviluppo e product owner, la durata dello sprint assicura che il rischio di spostarsi da quanto richiesto sia limitato alla durata dello sprint. Uno sprint può essere cancellato se l'obiettivo dello stesso diventa obsoleto, tuttavia ha poco senso vista la durata limitata.

3.3 Ruoli

Product Owner

- Definisce le caratteristiche del prodotto
- Rappresenta il desiderio del committente
- Decide date e contenuto del rilascio
- È responsabile della redditività del prodotto (ROI)
- Definisce le priorità delle caratteristiche del prodotto in base al valore di mercato che gli attribuisce
- Adegua le caratteristiche e la priorità ad ogni iterazione, secondo quanto necessario
- Responsabile che il Product Backlog sia chiaro e ordinato
- Accetta o rifiuta i risultati del lavoro

Scrum Master

- Rappresenta la conduzione del progetto
- Responsabile dell'adozione dei valori e delle pratiche Scrum
- Rimuove gli ostacoli
- Si assicura che il gruppo di lavoro sia pienamente operativo e produttivo
- Favorisce una stretta cooperazione tra tutti i ruoli e le funzioni
- Protegge il gruppo di lavoro da interferenze esterne
- Servant leader: aiuta Product Owner e Team di sviluppo condividendo la gestione e le decisioni con il team

Development Team

- Tipicamente 5-9 persone
- Responsabili di realizzare l'incremento in conformità con la Definition of Done
- Competenze trasversali (cross functional): programmatori, tester, progettisti di user experience, etc.
- Membri di progetto dovrebbero lavorare full-time
- Il gruppo di lavoro si auto-organizza: idealmente niente titoli, ma in rari casi può essere una possibilità

3.4 Eventi

Sprint Planning

Evento time boxed (circa 8 ore per sprint di un mese). L'oggetto di questo è far selezionare dal team di sviluppo gli item da inserire nello sprint backlog, presi dal product backlog. Ciascun task inserito viene quindi stimato (1-16 ore).

Daily Scrum

Detto anche stand up meeting o mischia quotidiana, deve durare al massimo 15 minuti. Non serve a risolvere problemi, ma per sincronizzarsi su quanto fatto e pianificare la giornata per il raggiungimento dello sprint goal. Si aggiorna quindi la scrumboard. Aiuta ad evitare altre riunioni non necessarie.
I problemi emersi verranno discussi successivamente con i singoli interessati.
NB: non è un SAL (Stato Avanzamento Lavori) per lo scrum master, sono impegni assunti tra pari (team di sviluppo).

Sprint Review

Evento time boxed (circa 4 ore per sprint di un mese). Il gruppo di lavoro presenta ciò che ha realizzato durante lo sprint tipicamente in forma di demo delle nuove caratteristiche o dell'architettura sottostante (niente slide, max 2 ore per la preparazione). Partecipa tutto il gruppo e sono invitati anche gli esterni.

Sprint Retrospective

Evento time boxed (circa 3 ore per sprint di un mese). Si celebra dopo la Sprint Review e prima del prossimo Sprint Planning. Si valuta ciò che sta funzionando e ciò che non sta funzionando:

- come migliorare il prodotto?
- la Definition of Done è appropriata?
- come posso migliorare il prossimo sprint?

Vi partecipa tutto il gruppo.

3.5 Artefatti

Product Backlog

Raccoglie i requisiti, le funzionalità, i miglioramenti e i fix da realizzare nei prossimi rilasci, essenzialmente una lista di tutti i "desiderata" espressa in modo che sia comprensibile da tutti gli utenti del prodotto o per gli utenti. Le priorità vengono assegnate dal product owner, mentre le stime sono fatte dal development team. Le priorità sono rivalutate all'inizio di ogni sprint, è quindi una lista dinamica che evolve con il prodotto, con un raffinamento continuo.

Sprint Backlog

Ogni componente del Development Team si sceglie qualcosa da fare, il lavoro non è mai assegnato. La stima del lavoro rimanente è aggiornata ogni giorno nel daily scrum, infatti ogni membro del gruppo può modificare parti dello sprint backlog. Se il lavoro non è chiaro, definire un elemento dello sprint backlog con una stima temporale più ampia, e decomporlo successivamente, aggiornando il lavoro rimanente man mano che diventa più chiaro.

Sprint Goal

Una breve indicazione dell'obiettivo principale dello Sprint. Alcuni esempi:

- Database Application: Fare girare l'applicazione anche su SQL Server oltre che su Oracle
- Financial Services: Supportare più indicatori tecnici di quanto faccia ABC con dati in tempo reale

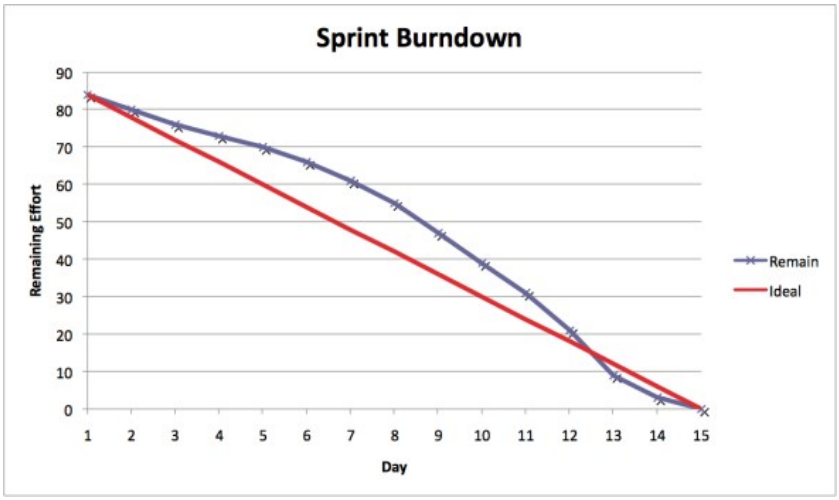
Definition of Done

Definisce il significato di "svolto" per uno sprint item, ovvero il minimo set di attività per definire che un'attività è completa. Può variare per gruppo di lavoro e dev'essere ben chiara a tutti i membri. È utilizzato per verificare se un'attività è da ritenersi completata.

Acceptance Criteria

Permette di confermare se la storia è completa e funziona come voluto. Sono un insieme di frasi semplici condivise da Product Owner e Development Team. Possono essere incluse con la User Story e rimuovono l'ambiguità dei requisiti.
Esempi: A user cannot submit a form without completing all the mandatory fields; Information from the form is stored in the registrations database; An acknowledgment email is sent to the user after submitting the form.

Sprint burndown chart



La linea rossa è la stima, quella blu l'andamento reale. È comune che all'inizio si vada peggio del previsto, tuttavia è facile che poi si vada sotto la stima e si finisca in tempo.

4 Git

Git è un software di controllo versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005.

4.1 Caratteristiche

- Branching and merging: favorisce lo sviluppo isolato
- Piccolo e veloce: la maggior parte delle operazioni viene fatta in locale, solo la condivisione in remoto richiede connessione. È stato realizzato in C per essere leggero e veloce.
- Distribuito: ci sono backup multipli della repository creati tramite la *clone*
- Integrità: ogni commit è identificato da un ID (checksum SHA-1 di 40-caratteri basato sul contenuto di file o della struttura della directory). Non è possibile cambiare un commit senza modificare l'ID del commit stesso e di i commit successivi.
  I commit fanno sempre riferimento al commit padre (il precedente, quello da cui sono cominciate le modifiche).
- Staging area: È stata aggiunta un'area di staging dove vengono validati i file modificati che potranno essere versionati con un commit
- Free and open source: licenza GNU/GPL 2.0, sorgente gestito su github in repo pubblica.

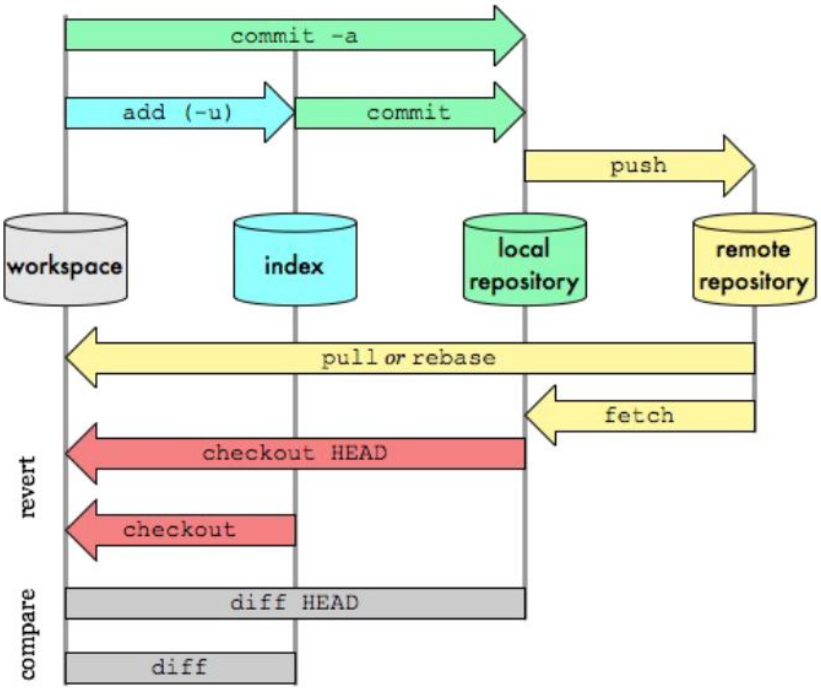
Git salva le variazioni ai file sotto forma di istantanee, non diff. Infatti, ogni volta che si esegue un commit viene fatto uno snapshot dello stato attuale del file system. Inoltre, se un file non subisce modifiche questo sarà un puntatore alla versione precedente, in modo da risparmiare spazio.

4.2 Aree locali - Stato di un file

In GIT i file della copia locale possono essere:

- Nella *Working directory*, checked out, modificati ma non ancora validati (**Modified**)
- Nella *Staging Area*, validati ma non ancora committati. Il commit salva uno snapshot di tutti i file presenti nella staging area (**Staged**)
- Nel *Repository locale* (**Committed**)

Un file appena creato è nello stato di **Untracked**.



4.3 Configurazione

È richiesta una configurazione iniziale dove vengono impostati l'username e l'email da usare per ogni commit:

- `git config --global user.name "Bugs Bunny"`
- `git config --global user.email bugs@gmail.com`

Ci sono tre livelli di visibilità per il config: system (tutti gli utenti), global (utente corrente), local (default, singola repository).
È possibile invocare `git config --list` per avere la lista di tutte le configurazioni.

Per creare una repository locale si usa `git init` dentro la cartella di un progetto esistente. Tutti i file sono quindi untracked. Viene creata una cartella `.git` contenente la repo.
È inoltre possibile clonare una repository remota con `git clone <url>` o locale `git clone <local_directory_path>`.

4.4 Altri comandi utili

- `git reset HEAD <file_name>`: rimuove un file dalla staging area senza perdere le modifiche
- `git checkout -- <file_name>`: rimuove un file dalla staging area perdendo le modifiche
- `git status`: vedere lo stato dei file workspace o nella staging area
- `git diff`: per vedere cos'è stato modificato ma non ancora validato nella staging area
- `git diff --cached`: per vedere cos'è stato modificato nella staging area
- `git log`: mostra la lista dei cambiamenti (commit) nel repository locale
- `git log -2`: mostra ultimi due commit
- `git branch <name>`: crea un nuovo branch indipendente
- `git branch`: lista dei branch
- `git checkout <branch_name>`: cambia branch su cui si sta lavorando
- `git checkout master -> git merge <branch_name>`: effettuare attività di merge di un branch nel master
- `git remote`: lista dei repository remoti collegati (possono essere multipli)
- `git fetch origin`: recupero i nuovi branch e le modifiche dal remoto senza aggiornare la working copy (senza fare merge)
- `git pull origin master`: recupero le ultime modifiche dal repository remoto e aggiorno la working copy (fetch and merge)
- `git push origin master`: inviare le modifiche al repository remoto

Commit keywords

- `git commit -m "close #N"`: questo commit chiude la issue N
- `git commit -m "#N message`: aggiunge questo commit alla issue N, con relativo messaggio

4.5 SVN vs GIT

SVN sta per *subversion*. La principale differenza è che il primo è centralizzato, il secondo distribuito. Inoltre, in SVN una repository può contenere diversi progetti, questo perché il branching avviene sulle singole cartelle, non nella root. In git questo non è possibile, esiste una repo per ogni progetto.
Un'altra importante differenza è che in assenza di connessione, in SVN non è possibile continuare il lavoro di versionamento, visto che il database è singolo e centralizzato.

5 Build Automation

5.1 Definizione

È il processo di automazione della creazione di build. Ci sono 2 categorie:

- **build-automation utility** (make, gradle, ant...), il cui scopo primario è quello di costruire eseguibili a partire dai diversi file sorgenti che compongono il progetto gestendo attività come compiling e linking
- **build-automation servers** (continuos integration), i quali sono dei server che eseguono utility di build-automation in remoto

Il problema iniziale era quello di integrare il codice scritto (è difficile compilare un grande progetto tutto insieme). C'era la necessità di poter scaricare, compilare ed eseguire un progetto open-source per poter collaborare e provare le modifiche apportate.

Vi sono 2 diversi tipi di build-tools:

- **scripting tools:** tramite l'utilizzo di un linguaggio di scripting si definisce un automatismo (script, make, ant, gradle...)
- **artefact oriented tools:** si basa sulla definizione e creazione di un artefatto (prodotto). Viene configurato il processo di build che è definito nello strumento (Apache Maven, NPM, ...)

Definizione processo di build

Il *processo di build* è 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 distribuibile.

5.2 Caratteristiche (CRISP)

- **Completo:** indipendente da fonti non specificate nello script di build. Tutto quello che verrà utilizzato dev'essere specificato.
- **Ripetibile:** accede ai file contenuti nel sistema di gestione del codice sorgente. Un'esecuzione ripetuta dallo stesso risultato. A partire da un commit devo essere in grado di produrre lo stesso artefatto
- **Informativo:** Fornisce informazioni sullo stato del processo (se tutti i passi sono andati a buon fine o meno). Devo accorgermi se ha fallito, come e dove ha fallito
- **Schedulabile:** programmabile ad una certa ora e fatto eseguire automaticamente. I prerequisiti affinché sia schedulabile sono le altre 4 caratteristiche
- **Portabile:** indipendente il più possibile dall'ambiente di esecuzione (eseguibile su un server esterno)

5.3 Apache Maven

5.3.1 Definizione

È uno strumento di gestione di processo basato sul concetto del POM (project - object - model). Può gestire i processi di build, di documentazione, di reportistica a partire dalle informazioni presenti in un file chiamato POM.
Viene usato per gestire progetti Java e sfrutta il paradigma *"convention over configuration"*, il quale prevede che ci sia una configurazione minima che vada a descrivere i metadati del nostro progetto, ovvero dove si trovano i sorgenti. La configurazione completa è integrata nello strumento e va bene per la maggior parte dei progetti.
Nella configurazione minima specifichiamo solo le caratteristiche del nostro progetto basandoci sul workflow e le configurazioni presenti nello strumento.

5.3.2 Caratteristiche di Maven

- **Build Tool:** sono definite delle Build Lifecycle che permettono di configurare ed eseguire il processo di build (e altri processi). È quindi uno strumento di build e il processo di build è standard.
- **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. Tutte le dipendenze vengono identificate da una tripletta detta GAV (group ID - artefact ID - version ID).
- **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 e estendere le fasi dei Build Lifecycle. La più famosa è Maven Central, tuttavia possiamo definire le nostre proprietarie.
- **Universal Reuse of Build Logic:** Le Build Lifecycle, 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), la creazione di template di progetto (p.es. Applicazioni Java Web o applicazioni costriuite con un determinato framework)

5.3.3 Build Lifecycle

Maven si basa sul concetto di Build Lifecycle, ovvero il processo per fare build e distribuire un artefatto è ben definito. Questo significa che lo sviluppatore deve imparare un set ristretto di comandi per fare build di qualsiasi progetto Maven, e il file POM assicura che abbiano il risultato desiderato.
Ci sono tre build lifecycle già integrati:

- **default lifecycle:** ci permette di andare ad eseguire la compilazione ed esecuzione del nostro pacchetto
- **clean lifecycle:** ci permette di andare a pulire la directory dopo il processo di build
- **site lifecycle:** serve per creare la documentazione del nostro progetto sotto forma di sito web

Build Lifecyle (default)

La default Build Lifecycle è composta dalle seguenti fasi (o goals):

- *validate:* controlla che il progetto abbia tutte le informazioni richieste per la fase successiva
- *compile:* attraverso i metadati e le dipendenze definite, Maven capisce il compilatore da usare, che dipendenze scaricare, come creare classpath e come compilare il progetto
- *test:* recupera le dipendenze di test, crea i classpath ed esegue i test di unità
- *package:* parte dai compilati generati precedentemente e crea il pacchetto (es. il file .jar in Java)

- *verify:* esegue i test di integrazione sul pacchetto generato
- *install:* installa il pacchetto generato nel nostro repository locale in modo da usarlo come dipendenza per altri pacchetti
- *deploy:* ci permette di condividere il progetto in un artefact repository e condividere il nostro pacchetto

Se un progetto è gestito con maven (ed è presente il file pom.xml) sarà possibile eseguire il uno dei processi (Build Lifecycle) invocando una delle fasi predefinite.
In un qualsiasi progetto gestito con maven, invocando 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*).

5.3.4 POM

Un Project Object Model è l'unità di lavoro fondmanetale di Maven. È in XML e contiene le informazioni in metadati del progetto e le configurazioni desiderate.
Quando si esegue un task o un goal, Maven va a vedere le configurazioni nel file POM, se non presenti le va a recuperare nel POM di default di Maven, poi esegue il goal.
Le configurazioni che possono essere specificate nel POM sono: le dipendenze del progetto, i plugin o i goal che possono essere eseguiti, i plugin che devono essere usati in determinati goal, dei profili che possono essere attivati ed eseguono determinate operazioni, dei metadati che descrivono la versione del progetto, come identificarlo tra le dipendenze, etc...
Tutti i plugin sono mantenuti nell'artefact repository.

5.3.5 Altre utility di Maven

Project Archetypes
Ci permette di andare a definire delle caratteristiche comuni per creare un progetto. Tipicamente se devo cominciare un progetto Java con Maven mi è scomodo ricordare la struttura delle cartelle standard, come creare il POM e come strutturare il progetto.
Tramite questa feature, Maven ci mette a disposizione un catalogo di template di progetti da cui possiamo partire.

Mavel plugin e Mojo
Maven in realtà è un orchestratore di plugin Mojo, che vengono attribuiti alle varie fasi del processo di build.
I plugin sono dei componenti che vanno a leggere le informazioni presenti nel POM del progetto e hanno la possibilità di andare a fare determinate attività, come compilazione, l'esecuzione dei test, l'analisi statica, la generazione dei javadoc, etc...
È molto semplice creare plugin per Maven, grazie all'architettura convention over configuration.
I plugin sono a loro volta artefatti Java.

6 Software Testing

6.1 Definizione

- È un'attività di investigazione fatta principalmente per trovare informazioni relative alla qualità del software sotto test. Ha lo scopo di verificare se quello che è stato chiesto dal committente è stato implementato e non contenga errori.
- È un processo composto da attività statiche e dinamiche che riguardano la preparazione, la pianificazione e la valutazione delle attività per verificare che un software soddisfi i requisiti e non abbia dei difetti.

Il 45% dei difetti è introdotto dal programmatore, il 20% durante l'analisi dei requisiti e il 25% nella progettazione.

6.2 Categorie di testing

- *Funzionale:* viene condotto per verificare che il prodotto rispetti i requisiti funzionali (richiesti dal committente)
- *Non funzionale:* il test fa riferimento a requisiti non funzionali (impliciti, necessari dal problema, non esplicitati dal committente)
- *Statico:* non richiedono che il codice venga eseguito, comprendono la revisione dei documenti
- *Dinamico:* richiede l'esecuzione del codice o parte di esso
- *Verifica:* il prodotto è stato realizzato secondo le specifiche (tecniche) e funziona correttamente. Cercare difetti per migliorare la qualità del software
- *Validazione:* il prodotto è stato realizzato rispettando le specifiche dell'utente (requisiti). Convalidare che il software sia conforme ai suoi obiettivi (fit for purpose)

6.3 Processo di test

Deve essere parallelo al processo di sviluppo, fin dalla definizione dei documenti. Ecco le sue fasi:

1. *Test planning:* viene definito il text plan, il documento principale in cui vengono definite le operazioni di test da fare durante il progetto.
2. *Test control:* sono le azioni correttive e di controllo da attuare nel caso in cui il piano non venga rispettato
3. *Test analysis:* si decide cosa testare
4. *Test design:* si decide come effettuare i test
5. *Test implementation:* si definiscono i casi e gli script di test
6. *Test execution:* viene effettuata l'attività di test
7. *Checking result:* si controlla se il test è stato superato/fallito
8. *Evaluating exit criteria:* si verifica se sono stati raggiunti i requisiti definiti nel text plan
9. *Test results reporting:* si riporta il progresso rispetto agli exit criteria definiti nel text plan
10. *Test closure:* chiusura del processo e definizione azioni di miglioramento per i test

6.4 Sette principi del testing

Il test mostra la presenza di difetti

Testare un'applicazione può rivelare che uno o più difetti esistono nell'applicazione, ma non può dimostrare che l'applicazione sia priva di errori. È solo possibile progettare casi di test per individuare il maggior numero di errori possibile.

Il test esaustivo è impossibile

A meno che l'applicazione in prova abbia una struttura logica molto semplice e un input limitato, non è possibile testare tutte le possibili combinazioni di dati e scenari.
Per questo motivo, il rischio e le priorità vengono utilizzati per concentrarsi sugli aspetti più importanti da testare.
Le strategie per selezionare i test sono:

- *Risk based testing:* rischio, funzionalità che hanno impatto sul business
- *Requirement based:* controllo i requisiti coperti

Early testing

Avviare la fase di test il prima possibile permette di risparmiare sui costi del progetto. Prima troviamo il bug e meno ci costa.
Il processo di test non deve essere eseguito quando il progetto è al termine, ma deve andare in parallelo con il processo di sviluppo.

Defect clustering

Durante i test, si può osservare che la maggior parte dei difetti segnalati sono legati a un numero ridotto di moduli all'interno di un sistema.
Un piccolo numero di moduli contiene la maggior parte dei difetti nel sistema, circa l'80% dei problemi si trova nel 20% dei moduli.

Paradosso del pesticida

Se continui a eseguire lo stesso set di test più e più volte (ad ogni nuova versione), siamo sicuri che non ci saranno più gli stessi difetti scoperti da quei casi di test.
Poiché il sistema si evolve, molti dei difetti precedentemente segnalati vengono corretti. Ogni volta che viene risolto un errore o è stata aggiunta una nuova funzionalità, è necessario eseguire tutti i test (di non regressione) per assicurarsi che il nuovo software modificato non abbia interrotto vecchi errori.
Tuttavia, anche questi casi di test di non regressione devono essere modificati per riflettere le modifiche apportate nel software per essere applicabili.
Il set di test va modificato con l'evoluzione del nostro progetto.

Il testing dipende dal contesto

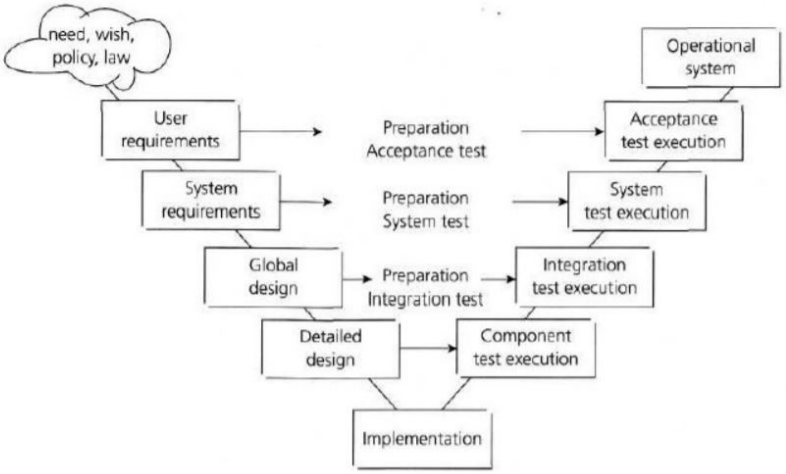
Diverse metodologie, tecniche e tipi di test sono legati al tipo e alla natura dell'applicazione.
Ad esempio, un'applicazione software in un dispositivo medico richiede più test di un software di giochi.
Un software per dispositivi medici richiede test basati sul rischio, essere conformi con i regolatori dell'industria medica e possibilmente con specifiche tecniche di progettazione dei test.
Un sito web molto popolare, deve superare rigorosi test delle prestazioni e test di funzionalità per assicurarsi che le prestazioni non siano influenzate dal carico sui server.

Absence of errors fallacy

Solo perché il test non ha riscontrato alcun difetto nel software, non significa che il software sia perfetto e pronto per essere rilasciato. I test eseguiti sono stati davvero progettati per catturare il maggior numero di difetti? Il software corrispondeva ai requisiti dell'utente?
Bisogna bilanciare costi, benefici e tempo.

6.5 V-model

Il processo di test è parallelo a quello di sviluppo, ogni fase dell'attività di sviluppo corrisponde ad una fase di test.



6.6 Tipi di test

Component testing (unit testing)

Verificano l'unità: il più piccolo sottosistema possibile (in Java: classe o metodo) che può essere testato separatamente, sono veloci da eseguire, non dipendono dall'ordine di esecuzione e ogni modifica del codice sorgente dovrebbe scatenare l'esecuzione degli unit test.
Il sistema sotto test (SUT) è considerato come *white box*, ovvero possiamo vedere il codice sorgente.

Integration test

Verificano se sono rispettati i contratti di interfaccia tra più moduli o sub-system.
I sub-systems possono essere:

- Interni: prodotti da noi, già verificati dagli unit test
- Esterni: altri componenti con cui integriamo il nostro progetto, es. database, filesystem

Questo tipo di test è più lento da configurare e da eseguire. Il SUT è white box.

System testing

Verificano il comportamento dell'intero sistema. Il loro scopo principale è la verifica rispetto alle specifiche tecniche. Il sistema può essere sia white box che black box.
Un tipo di system testing sono gli **smoke test**, il cui obiettivo è quello di trovare gli errori il prima possibile (prima della produzione) o eseguire altri test sul SUT. Verificano le funzionalità del SUT.

Acceptance test

Il sistema è black box e sono eseguiti con il cliente finale. Serve a verificare che i requisiti siano stati implementati correttamente.

Non-regression testing

Verificano che il comportamento del SUT rimanga almeno lo stesso rispetto alla versione precedente (non ci sia un deterioramento qualitativo), si focalizzano su un singolo bug per volta e assicurano che i bug corretti non si presentino ancora.

7 Unit Testing

7.1 Definizione

In ingegneria del software, 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; 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 o un 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).

I test di unità sono del codice, prodotto dallo sviluppatore, che esercitano un'unità del programma. Per unità si intende una funzionalità atomica che può essere verificata in modo isolato, in modo da assicurare che il risultato del test non sia influenzato da altre unità. Nella programmazione ad oggetti un'unità può essere uno o più metodi di una classe, o un'istanza di una classe. Nella programmazione procedurale un'unità corrisponde ad una funzione.
Vengono sviluppati dal programmatore che sviluppa le unità, per verificare l'assenza di alcuni errori, e documentare il comportamento dell'unità prodotta.

7.2 Proprietà desiderabili - A TRIP

Automatic - Thorough - Repeatable - Independent - Professional

Automatic

I test di unità devono essere eseguiti automaticamente. In ogni progetto deve essere disponibile un "automazione a comando" che permetta a tutti di invocare e far eseguire tutti o una parte dei test di unità in modo semplice.
Durante la fase di sviluppo del progetto è importante che i test possano eseguire:

- *In modo rapido:* I test di unità devono essere semplici e la loro esecuzione non deve impiegare più di pochi secondi.
- *Senza richiedere l'interazione umana:* Se un test di unità richiede che alcuni parametri siano inseriti, ogni volta, manualmente da uno sviluppatore, questo non permetterebbe di eseguire tutti i test del progetto in modo automatico a determinate ore del giorno.
- *In modo autonomo:* L'automazione che effettua l'esecuzione dei test di unità deve essere in grado di capire quando e dove i test falliscono ed avvisare gli sviluppatori. In questo modo gli sviluppatori saranno interrotti, dall'attività lavorativa, solo quando uno o più test falliranno. Serve un metodo per far fallire la build se il test fallisce.

Thorough (esaustivi)

Dei buoni test di unità devono essere esaustivi e accurati, devono verificare il comportamento di qualsiasi parte del progetto che potrebbe creare degli errori.
Esistono degli strumenti che permettono di misurare se ogni parte del progetto è stata eseguita durante la fase di test, e possono calcolare:

- La percentuale di righe di codice che vengono esercitate attraverso i test di unità nel progetto
- La percentuale di possibili diramazioni che vengono eseguiti dai test di unità
- Il numero di eccezioni che vengono controllate attraverso i test
- Altri dati che permettono di capire dove il progetto è carente di test di unità

Come si può intuire, non è detto che se in un progetto viene eseguito il 100% del codice dai test di unità questo è privo di errori.

Repeatable (ripetibili)

I test di unità devono produrre sempre lo stesso risultato con lo stesso input. I test non devono seguire un ordine.
Per essere ripetibili, i test di unità devono avere l seguenti caratteristiche:

- *Essere indipendenti dall'ordine di esecuzione:* L'ordine di esecuzione dei test di unità non deve influenzare il risultato. Per questo è necessario che i test siano indipendenti l'uno dall'altro. Un test non può essere precondizione di un altro.
- *Essere indipendenti dall'ambiente di esecuzione:* L'esecuzione dei test non deve dipendere da risorse esterne al progetto o da risorse non gestite nel VCS. Se alcune unità devono utilizzare risorse esterne (p.es. database) è consigliato utilizzare la tecnica Mock Object per simulare il comportamento di queste componenti (oggetti finti che simulano il comportamento dei subsystem esterni, classe di simulazione). Esistono dei framework specifici per questo tipo di oggetti.

Independent (indipendenti)

I test di unità devono essere il più possibile indipendenti dall'ambiente di esecuzione, dagli elementi esterni al progetto e dall'ordine di esecuzione. Quando si scrive un test è consigliato verificare il comportamento di un singolo aspetto del progetto (in questo modo si riesce ad identificare univocamente un'errore). Questo non significa che un test di un'unità deve avere solo una asserzione, ma deve controllare solo un metodo o più metodi che realizzano un aspetto di una funzionalità del progetto.
Se il test è indipendente il suo comportamento sarà ripetibile nel tempo, perché il suo comportamento non dipenderà dalle altre unità del progetto. La ripetibilità del test è un aspetto che permette di capire se il test è indipendente.

Professional

Poiché i test di unità sono codice, devono essere scritti e mantenuti con la stessa professionalità del codice di produzione del progetto.
Visto che i buoni test di unità devono essere esaustivi, è ragionevole che il numero di linee di codice per realizzare i test sia pari o a volte superiore delle linee di codice in produzione.

7.3 Framework per unit testing

Per creare i test di unità sono necessari i seguenti componenti:

- 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 aspettati, 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

Nel corso si vedrà JUnit 4, il quale ha tutte le caratteristiche elencate.

7.4 Come verificare i risultati

Verificare se i risultati che essa produce sono corretti. Per corretto si intende che il risultato atteso sia uguale al risultato prodotto dall'unità. Capita che i requisiti non sono 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.
I test di unità descrivono come le unità che abbiamo prodotto dovrebbero funzionare.

Boundary conditions (CORRECT)

- 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.

Check inverse relationship

Alcune unità possono o devono essere verificati tramite l'applicazione della loro funzionalità inversa.
Esempi:

- Calcolare la radice quadrata di un numero. Per testare se la radice quadrata è corretta è possibile elevare al quadrato il risultato ritornato dall'unità e confrontarlo con il dato di partenza.
- Inserimento di un un elemento in una pila. Il modo più semplice per verificare se l'inserimento è andato a buon fine è quello di effettuare un prelevamento dalla pila e controllare che l'elemento ritornato sia l'elemento di partenza.

Cross-check Using Other Means

Utilizzare uno strumento esistente (oracolo) per verificare se la nuova unità ha lo stesso comportamento.
Esempio: migrazione da un vecchio sistema ad uno nuovo appena realizzato, si utilizza il vecchio sistema per verificare che quello nuovo abbia lo stesso comportamento.

Force error contitions

Nel mondo reale gli errori accadono. Una buona norma, per creare un buon progetto, è quello di ricreare le condizioni di errore e verificare che il progetto funzioni come ci si aspetta in queste condizioni.

Performance caratteristics

I test di unità devono essere veloci perché devono poter essere eseguiti molto spesso, sia nelle workstation degli sviluppatori che negli automation server.

7.5 Test driven development

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"), 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"), il programmatore sviluppa la quantità minima di codice necessaria per passare il test.
3. Nella terza fase (detta "fase grigia" o di refactoring), il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualità.

8 Analisi statica del codice

8.1 Definizione

L'analisi statica del codice viene effettuata senza eseguire il software, a differenza dell'analisi dinamica.
Si va a vedere che vengano rispettati gli standard e si cercano bug che possono essere trovati dallo sviluppatore che legge il codice. Viene effettuata tramite strumenti automatizzati, i quali mostrano una lista di warning riguardo i problemi del codice.
Teoria delle finestre rotte: Si segnalano errori minori e poco rilevanti in modo da evitare cose più gravi, mantenendo ordine su tutto il progetto.

8.2 Tool utilizzati

Sono simili ad un correttore ortografico, permettono di:

- 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): l'indice di complessità ci suggerisce quanti test unitari sono necessari per quella parte di codice
- Ricercare codice copiato in più punti (fare copia incolla è una pratica sbagliata)
- Ricercare errori comuni nel codice
- Misurare la percentuale di codice testato
- Ricercare indicatori di parti incomplete (p. es. tag)

Ci sono diversi tool per questo scopo, il più avanzato è SonarQube.

SonarQube

Può essere integrato nel workflow esistente per garantire un'ispezione del codice continua tra i vari branch del progetto e le pull request. Alcune sue funzionalità sono:

- Storicizza l'andamento della qualità
- 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)
- Le issue vengono classificate 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)
- Permette di revisionare le issue segnalate e segnare i falsi positivi

9 Continuous Integration

9.1 Definizione

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, VCS). Il concetto è stato originariamente proposto nel contesto dell'extreme programming (XP), come contromisura preventiva per il problema dell' "integration hell" (le difficoltà dell'integrazione di porzioni di software sviluppati in modo indipendente su lunghi periodi di tempo e che di conseguenza potrebbero essere significativamente divergenti). Si cerca perciò di fare più volte possibile l'integrazione.
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). In particolare, 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 (Maven) e/o esecuzione automatica di test, come Jenkins.

La CI non rimuove tutti i bug, ma rende molto più facile trovarli e rimuoverli in seguito. Inoltre consente a un team di intensificare l'attività di sviluppo e test, integrando gli sviluppi il più spesso possibile. È una *pratica agile* in cui i membri di un team integrano il loro lavoro frequentemente, di solito giornalmente o più volte al giorno.

9.2 Motivazioni

Prima della nascita della CI, in molti progetti vi erano questi problemi:

- Per lunghi periodi di tempo, durante il processo di sviluppo, il progetto non è in uno stato funzionante o in uno stato utilizzabile. Sopratutto in progetti dove si sviluppa in un singolo ramo di sviluppo (centralized work-flow)
- Nessuno è interessato a provare ad eseguire l'intera applicazione fino a quando non è finito il processo di sviluppo
- In questi progetti spesso viene pianificata la fase di integrazione alla fine del processo di sviluppo. In questa fase gli sviluppatori effettuano attività di merge ed effettuano attività di verifica e validazione

Tutto questo comporta l'*integration hell*: La fase di integrazione può richiedere molto tempo, e nel caso peggiore, nessuno ha modo di prevedere quando terminerà questa fase (solitamente termina quando non abbiamo più tempo, e non quando abbiamo risolto tutti i problemi), di conseguenza quando l'applicazione può essere rilasciata.
In pratica si sposta il problema della verifica in avanti, dilatandone i tempi richiesti.

9.3 Prerequisiti

Per implementare la pratica di CI è necessario che:

- Il codice del progetto venga gestito in un VCS
- Il processo di build del progetto sia automatico
- Il processo di build esegua delle verifiche automatiche (test di unità, test di integrazione, analisi statica del codice e altre verifiche)

- il team di sviluppo adotti correttamente questa pratica
- (Opzionale) Un sistema automatico dove eseguire il processo di build ad ogni integrazione. L'integrazione potrebbe avvenire attraverso le pull request, il responsabile della repository esegue manualmente le verifiche ed accetta o meno la richiesta. Tuttavia è preferibile l'automatismo per questa attività.

9.4 Processo

9.4.1 Visione generale

- 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 automation(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, per evitare che altre persone vadano a lavorare su una versione non funzionante.

In questo modo si assicura la presenza di un prodotto consistente potenzialmente pronto per essere validato e rilasciato

9.4.2 In dettaglio

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 buil 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

Nella pratica è il sistema di CI che ci manda le notifiche (tramite e-mail), in modo da ridurre i tempi morti.

9.5 Best practices

Integrare frequentemente gli sviluppi

Integrare frequentemente gli sviluppi, più di una volta al giorno.
In questo modo:

- le modifiche da integrare saranno poche e più facile da gestire
- Se viene segnalato un errore dall'esecuzione del processo di build sarà più semplice identificare il codice che ha introdotto l'errore (che sarà presente nei commit che hanno fatto scatenare il processo di build)

Per poter integrare frequentemente gli sviluppi è richiesto che il progetto venga scomposto in tante attività brevi (framework scrum).

Creare una suite di test automatici robusti

Se non vengono eseguiti dei test automatici per verificare e validare il progetto, il processo di build può solo verificare se il codice integrato compila correttamente.
I test che possono essere eseguiti nel processo di CI sono:

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

Tenere i processi di build e test corto

Nella CI il processo di Build viene eseguito molto frequentemente (ad ogni integrazione). Se questo è lento:

- Gli sviluppatori smetteranno di eseguire il processo di build e i test prima di inviare le modifiche al VCS, perciò inizieranno a generare più build in errore, quindi perderà di credibilità il processo.
- Il processo di integrazione continua richiederà così tanto tempo che si verificheranno più commit nel momento in cui è possibile eseguire nuovamente la build, perciò sarà più difficile identificare cosa ha fatto fallire la build, perché il delta delle modifiche che avremo sarà troppo grande.
- Si disincentiva l'invio frequente delle modifiche al VCS.

Se abbiamo un processo di compilazione troppo lento significa che o abbiamo una macchina con poche risorse o che il nostro progetto è diventato troppo grande, bisogna quindi suddividerlo in moduli.

Gestire il proprio ambiente di sviluppo e quello condiviso

Ogni sviluppatore deve essere in grado di:

- Eseguire localmente il processo di build e test e deploy, per questo nel VCS devono essere gestiti:
  - codice di produzione
  - codice di test
  - script richiesti per configurare l'ambiente dove eseguire il progetto
- Always Be Prepared to Revert to the Previous Revision: Scaricare le modifiche dal VCS e essere in grado di ripristinare il progetto ad uno stato consistente.
- Never Go Home on a Broken Build: Verificare 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 (per non impattare le altre attività di sviluppo).

Riparare le build rotte immediatamente

"Niente ha più priorità rispetto ad una build che fallisce".
Il tempo per ripristinare lo stato del progetto deve essere 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 commentano le verifiche che hanno fatto fallire la build.

Trasparenza: tutti possono vedere cosa succede

Lo stato della build deve essere pubblicato in un servizio visibile a tutto il team del progetto.
Ogni componente del team deve essere in grado di capire lo stato del progetto e capire qual'è l'ultima versione in cui è stato eseguito il processo di build con successo.
Se il processo di build fallisce deve essere possibile:

- identificare chi ha introdotto l'errore
- avere a disposizione un log per identificare quale parte del processo di build è fallita
- avere a disposizione la lista dei commit che hanno introdotto l'errore (capire qual era l'ultima versione funzionante e quale commit ha fatto uscire l'errore)

Il sistema di continuous integreation deve poter avvisare (con delle notifiche) i componenti del team ad ogni cambio di stato del processo (da successo a fallimento, da fallimento a successo).

10 Artifact Repository

10.1 Definizione

La repository dei prodotti contiene l'artefatto che vogliamo rilasciare, ottimizzato per gestire il mantenimento e il download di questi. Centralizza la gestione di tutti i file binari e memorizza i metadati di questi (branch di provenienza, autori, stato del pacchetto, etc).
Nella pratica c'è la necessità di storicizzare i binari rilasciati dal processo di build, che verranno poi utilizzati nella pipeline, ovvero la modellazione sotto forma di processo di sviluppo e rilascio dell'applicazione.
I VCS sono fatti per gestire file sorgenti e non binari, che andrebbero a rallentare il processo di continuous integration. Per questo si usano gli artifact repository, che trattano i file in modo diverso dai VCS (visto che sono binari e non sorgenti).
Maven Central è un artifact repository pubblico, un'azienda può scegliere di utilizzare solo il suo interno privato.

10.2 Caratteristiche del repository

- Ambiente dove depositare e pubblicare i prodotti
- Agisce da intermediario per scaricare prodotti da depositi esterni
- Permette di effettuare ricerche e reperire informazioni riguardanti i prodotti
- Permette di gestire e associare permessi d'accesso sui prodotti
- Permette di segnalare problemi di vulnerabilità su prodotti (si collegano ai database di vulnerabilità e controllano gli artefatti)
- Permette di verificare problemi legati a licenze (dipendenze con prodotti terzi)
- Permette di documentare gli artefatti con dei metadati (p.es pom.xml)

10.3 Caratteristiche degli artefatti

Possibilità di distinguere univocamente un artefatto (group ID - artifact ID - versione del pacchetto).
Metadati che permettono di capire se (p. es. pom.xml):

- Artefatto di produzione/sviluppo (politiche di gestioni differenti)
- Riferimento del commit nel VCS da cui è stato creato l'artefatto
- Informazioni delle licenze
- Informazioni delle dipendenze
- MD5, SHA1 per garantire l'autenticità dell'artefatto
- Identificativo univoco

10.4 Maven repository management

Tipi di repository

- *Proxy Remote Repositories:* Possibilità di configurare il repository interno in modo da recuperare gli artefatti da repository esterni. Se l'artefatto non è presente nel repository interno viene consultato il repository esterno
- *Hosted Internal Repositories:* Possibilità di condividere all'interno di un'organizzazione artefatti di terze parti che hanno vincoli di licenza o artefatti creati internamente all'azienda e che devono restare interni (p.es. Dirver JDBC)

Tipi di artefatti

- *Release Artifacts:* Gli artefatti di tipo Release solitamente possono essere rilasciati una sola volta e vengono mantenuti nel repository (altrimenti non si può più verificare l'autenticità del pacchetto).
- *Snapshot Artifacts*: Gli artefatti in fase di sviluppo possono essere rilasciati più volte. Possono essere eliminati (e mantenuta la versione più recente). Di solito contengono il la Keyworkd SNAPSHOT e il timestamp.

Perché usare un Maven Repository Interno

- *Velocità del processo di Build:* Le dipendenze e gli artefatti di tipo plugin utilizzati nel processo di build vengono scaricati dalla rete interna
- *Maggiore stabilità e controllo:* I repository gestiti possono essere analizzati e si può decidere di vietare l'utilizzo di determinati artefatti per evitare problemi di sicurezza. È possibile distribuire le versioni ufficiali e certificate degli artefatti di terze parti. È più semplice fare analisi statiche come compatibilità della licenza del prodotto con le dipendenze, analisi legate alla sicurezza, analisi d'impatto, etc...
- *Facilitano la collaborazione:* Si evita di far creare gli artefatti dal VCS ed è più semplice identificare le versioni di rilascio certificate, non bisogna ripete il processo di build ogni volta per ogni pacchetto.

11 Continuous Delivery

11.1 Definizione

Chiamata anche CD, è una pratica di ingegneria del software dove il team produce il software in iterazioni brevi (approccio agile) dove viene assicurato che il prodotto sia rilasciabile in ogni momento. Il rilascio può avvenire in modo automatico e continuo oppure manuale.
C'è la necessità di fare build, test e rilascio velocemente ed in modo automatico; questo approccio riduce costi e fa risparmiare tempo rilasciando più frequentemente e avendo quindi più feedback da parte degli stakeholder.
Si sta facendo continuous delivery quando:

- il team da la priorità a mantenere il prodotto sempre in uno stato rilasciabile piuttosto che aggiungere nuove feature (se il pacchetto non può essere rilasciato per via di alcuni errori, il fix di questi ha la priorità su qualsiasi altra cosa)
- l'attività di rilascio è automatica (anche in produzione)
- si fa continuous integration (build e test automatici)
- il prodotto viene rilasciato in ambienti sempre più simili a quello in produzione, in modo che sia testato efficaciemente

Offre la possibilità di apportare cambiamenti in produzione con minor tempo e costo possibile.
Questi obiettivi sono raggiunti avendo il codice sempre pronto ad essere rilasciato.

La differenza con la continuous deployment sta nel fatto che il processo di rilascio in produzione è manuale in delivery, automatico in deployment.

11.2 Motivazioni

Prinicipali motivi per adottare la CD:

- La Continuous Integration permette di avere feedback su problemi introdotti dagli sviluppatori. Si focalizza sulla parte DEV assicurandosi che il codice compili e che vengano eseguiti i test di unità, integrazione, accettazione e l'analisi statica.
- Ci sono molti problemi durante un progetto software:
  - 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)

  Questi portano a software che non è rilasciabile perché è stato impiegato troppo tempo per farlo entrare in un ambiente simile alla produzione e che contiene molti difetti perché il ciclo di feedback tra il team di sviluppo e il team di testing e operations è troppo lungo.

Questa inoltre favorisce l'interazione fra le vari componenti del progetto.

Misurare per migliorare

L'obiettivo è quello di migliorare il processo che permette di rilasciare una modifica al codice sorgente del progetto in produzione.
Un vantaggio competitivo è dato dalla creazione di valore su tutte le attività richieste dal processo.
Una "Value Chain" è una modellazione del processo per misurare:

- Il Valore Globale
- Il Valore residuo da inserire nella "Value Chain" (il miglioramento)

Usare una "Deployment Pipeline" dà gli stessi risultati di utilizzare una "Value chain" negli altri settori non software:

- Controllare le prestazioni e migliorarne l'efficienza
- "Fast is cheap": le Pipeline permetto di trovare i problemi il prima possibile

Deployment pipeline

È la modellazione del processo di Deployment tramite una successione di fasi (stages) e verifiche (gates).

- Il passaggio da una fase all'altra viene verificato tramite il superamento di una verifica
- Il passaggio di una fase può scatenare una notifica
- È guidato dal concetto di Fail-Fast
- È Composta da stages mappate su attività misurabili (p.es build, test).
- La transazione tra due stages viene definita gate, può essere automatica o manuale.
- Gli Stages possono essere eseguiti in parallelo e/o in sequenza. I gates possono essere multi direzionali (attivare stages parallele).

11.3 Realizzazione e requisiti

È necessario:

- avere un rapporto di lavoro collaborativo con tutti i partecipanti (dev, test e ops)
- utilizzare le Deployment Pipelines per definire il processo di build, test e deploy dell'applicazione (e renderlo il più automatico possibile). Fondamentale che sia documentato e condiviso con tutti i componenti.
- distribuire il processo di build e deploy su più ambienti

Sono propedeutici alla CD tutti i seguenti requisiti:

- Continuous Integration
  - VCS
  - Build automation (Maven, gradle, ...)
  - Unit testing
  - Artifact Repository: sistemi che permettono di gestire le dipendenze e gli artefatti (binari) prodotti dal nostro sistema (p.es. Docker Hub, Maven central)
- Configuration management: strumenti che ci permettono, di gestire tramite codice, la configurazione degli ambienti dove dovrà essere rilasciato il nostro software
- Continuous testing: test automatici a livello di sistema funzionali e non funzionali
- Orchestratore: Sistema software che ci permette di modellare le esecuzioni delle pipeline (p.es. Jenkins)

11.4 Good practices

Only build your binaries Once

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 efficienza della pipeline e che l'artefatto rilasciato in produzione è esattamente lo stesso artefatto che è stato verificato e validato nelle Stages della pipeline.
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 differisce tra gli ambienti (p.es: riferimenti al DB, o a sub system esterni).

Deploy the same way to every environment

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:

- Gli sviluppatore 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, tenendoli separati dalle 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

Per verificare se il rilascio automatico è andato bene prevedere l'esecuzione di smoke-test. Questi devono verificare anche il corretto funzionamento dei sub-system esterni (DB, altri web-services, messageing bus). Sono veloci e semplici da realizzare e permettono di far fallire velocemente la pipeline in caso di problemi.

Deploy into a copy of production

Prevedere di avere a disposizione un ambiente con le stesse caratteristiche (o caratteristiche 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 all'ambiente 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

Negli ambienti cloud è molto più semplice.

Each change should Propagate through the pipeline instantly

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

Progettare la pipeline in moda 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

11.5 Vantaggi

- *Ridurre il 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.
- *Velocizza il time to market:* questa pratica porta ad avere rilasci più frequenti
- *Maggiori feedback da parte degli utenti:* riusciamo a sistemare e soddisfare meglio i requisiti degli utenti.
- *Progressi tangibili:* molti Team monitorano i progressi in un ITS. Se "DONE" significa "gli sviluppatori dichiarano che è stato fatto" è molto meno credibile rispetto a "è fatto, verificato e distribuito in un ambiente di produzione" (o simile alla produzione).
- *Minor costo:* diretta conseguenza dell'automazione, soprattutto nel lungo periodo
- *Prodotti migliori:* che soddisfano le aspettative degli utenti
- *Team meno stressati e più collaborativi*
- *Maggiore documentazione implicita:* conseguenza diretta dell'automazione.
- *Non dipendere da determinate persone:* essendo tutto codificato da automatismi le attività possono essere eseguite da personale senza specifiche competenze.