Riassunto MTSS

Riccardo Graziani

A.A 2024/2025

Indice

1 Issue Tracking System

(Def.) Un **Issue Tracking System** (o ITS) é un software, usato prevalentemente in ambiti di sviluppo **collaborativi**, che gestisce una lista di **issue** (ossia criticitá/eventi da gestire) secondo le necessitá dell'organizzazione che lo usa.

1.1 Utilizzo di ITS

A cosa serve un ITS:

- condividere le informazioni con il **team di sviluppo**, il **Project Manager** e il **cliente**;
- avere un'unica **repository** dove trovare le informazioni, un sistema di notifica e dashboard;
- implementare un processo per misurare la qualitá del progetto;
- avere un'**istantanea** del progetto (attivitá da fare, in corso, completate);
- decidere **quando** e **cosa** rilasciare;
- **assegnare** e dare **prioritá** alle attivitá;

- consultare il tempo impiegato;
- avere una **chiara** assegnazione delle attivitá;
- avere una memoria **storica** di ogni cambiamento del progetto;

1.2 Work Item

(Def.) Un **Work Item** é una singola attivitá minima del progetto, viene gestita tramite un workflow e mantenuta all' interno di un'unica piattaforma e di un unico repository.

I principali campi di un work item sono:

- **Progetto**: progetto a cui si riferisce;
- **Codice**: codice identificativo univoco;
- **Descrizione**: descrizione breve dell'attivitá;
- **Tipo**: categoria del work item (ne determina i campi, gli stati, le schermate e il workflow);
- **Stato**: lo stato all'interno del workflow;
- **Prioritá**: importanza del work item in relazione con gli altri work item del progetto;
- **Stato di risoluzione**: identifica lo stato di risoluzione del work item (es. chiuso, duplicato, etc.);
- **Versione di riferimento**: versione del progetto a cui é richiesta l'attivitá;
- **Componente**: componente del progetto a cui si riferisce il work item;
- **Tag**: classificatori dei work item;
- **Collegamenti**: permettono di collegare diversi work item;
- **Assegnatario**: identifica chi é responsabile per lo svolgimento dell'attivitá;

- **Segnalante**: identifica chi ha segnalato l'attivitá;
- **Data**: indica la data di creazione, ultimo aggiornamento e risoluzione;
- **Allegati**: indica i file allegati;

1.3 Workflow

(Def.) Il **workflow** é un insieme di **stati** e **transizioni** che un work item attraversa durante il suo ciclo di vita. In genere permette di implementare il **processo** da seguire per completare l'attivitá, viene associato ad un **progetto** e ad uno o piú **tipi** e permette di **registrare** tutte le transizioni e cambi di stato.

1.4 Collegamenti

(Def.) I **collegamenti** permettono di definire **relazioni** tra i work item anche di differenti tipi. I collegamenti sono **bidirezionali** (da/a work item) ed essi vengono **registrati** e possono essere usati come criterio di ricerca. Questo permette di verificare la presenza o meno di **relazioni** tra work item.

1.5 Funzionalitá di un ITS

Tre principali funzionalitá:

- **Gestione**: ricerca avanzata dei work item, salvataggio delle ricerche, esportazione e reporting;
- **Integrazione**: integrazione con il source code management e integrazione con l'ambiente di sviluppo;
- **Condivisione**: notifiche, bacheche/board, dashboard e definizione di road map e release notes;

1.6 Filtri

(Def.) I **filtri** permettono di **ricercare** work item in base ai campi, possono essere **salvati** per facilitare le ricerche piú frequenti ed i risultati dei filtri possono essere **esportati**. I filtri sono la base per creare report, board e dashboard.

1.7 Bacheche e Board

(Def.) Le **Board/Bacheche** permettono di **visualizzare** i work item di uno o piú progetti, offrendo un modo flessibile e interattivo di **visualizzazione**, **gestione** e **visualizzazione dei dati** di sintesi sulle attivitá in corso. Permettono di configurare e visualizzare i work item ricercati con un **filtro** e permettono di interagire **velocemente** con i work items (es. avanzare lo stato o modificare alcuni campi).

1.8 Report

(Def.) I **report** hanno lo scopo di monitorare ed avere una visione d'insieme del progetto.

1.9 Obiettivi ITS

I due obiettivi per configurare un ITS: identificare i **processi** richiesti per la gestione del progetto (procedure e best practises definiti dai framework di qualitá), i vincoli imposti dal cliente e la modalitá di gestione del progetto (waterfall, agile scrum, etc.); identificare e configurare i processi nell'ITS tramite la definizione di **tipi**, dei **campi custom**, dei **workflow** e dei collegamenti;

1.10 Caso d'uso per configurazione

1.10.1 Admin ITS

L'**admin** dell'ITS effettua le seguenti operazioni:

- crea un nuovo progetto;
- definisce il **processo** da seguire (tipi di work item, campi, workflow, collegamenti), seleziona il **modello** di stima e crea differenti **board** e **report** per processo;
- aggiunge gli **utenti** e **assegnazioni** ruoli/permessi;

1.10.2 Capo progetto

Il **capo progetto** effettua le seguenti operazioni:

- definisce le versioni (**release**);
- definisce le **componenti** del progetto;
- definisce il lavoro da svolgere (**backlog**) tramite prioritá, assegnatario e stima;
- definisce la prima iterazione;

1.11 Caso d'uso per utilizzo

1.11.1 Team di sviluppo

Il **team di sviluppo**:

- riceve le notifiche degli item assegnati;
- li seleziona in base alla prioritá;
- avviano e completano la lavorazione;
- avanzano gli stati del workflow,
- aggiornano la stima a finire;
- registrano il tempo impiegato;
- documentano lo stato delle attivitá e compilano i campi del work item;
- completano le attivitá dell'iterazione ed effettuano il rilascio;

1.11.2 Capo progetto

Il **capo progetto**:

- monitora l'avanzamento e il completamento delle attivitá (filtri, board, dashboard, report);
- definisce le nuove versioni;

- definisce le nuove iterazioni;
- definisce e aggiorna e monitora le attivitá (prioritá, verifica stima);
- produce i report richiesti dal cliente;

2 Version Control System

(Def.) Un **Version Control System** (VCS) o **Source Code Management systems** (SCM) é un software che permette di gestire **modifiche** a documenti, programmi, siti web e altre collezioni di informazioni.

Le principali caratteristiche dei VCS sono:

- sistemi software;
- registrano tutte le **modifiche** avvenute ad un insieme di file;
- permettono la **condivisione** di file e modifiche;
- offrono funzionalitá di **merge** e **tracciamento** delle modifiche;

I VCS possono essere categorizzati come VCS **locali**, VCS **centralizzati** o VCS **distribuiti**.

2.1 Local VCS

I tool piú vecchi, permettono di registrare soltanto la storia dei cambiamenti tramite un **Version Database** che registra la storia di tutti i file. Salva sul disco una serie di **patch** (ossia le differenze tra i file) e permette di ricreare lo stato di qualsiasi file in qualsiasi momento. Infine **non gestisce** la condivisione. Alcuni esempi sono **SCCS** e diversi **IDE**.

2.2 Centralized VCS

Meno vecchi e molto diffusi, gestiscono sia la **condivisione** sia il **tracciamento** della storia come nel local VCS, ma qui il version database é gestito da un **server centrale** (singolo punto di rottura). Ogni sviluppatore é un client che ha nel suo spazio di lavoro una sola versione alla volta del codice. Alcuni esempi sono **CVS**, **SVN**, **Perforce**.

2.3 Distributed VCS

Simili ai centralized VCS ma il version database é **distribuito** per duplicazione in ogni nodo: quando il nodo centrale non é disponibile si puó continuare a **lavorare** e **registrare** cambiamenti. Hanno una migliore risoluzione dei **conflitti** che favorisce la collaborazione, e permettono di impostare **diversi tipi di flussi** di lavoro. Alcuni esempi sono **Git**, **Mercurial**, **Bazaar** o **Darcs**.

2.4 Cloud-based DVCS

Sono **VCS as a Service** in cui il version database é gestito nel **cloud**: si delega la gestione e installazione ad un servizio esterno e forniscono servizi aggiuntivi come text editor, strumenti visuali, ITS etc. Alcuni esempi sono **GitHub**, **BitBucket**, **GitLab**, etc.

2.5 Nozioni sui VCS

Di seguito un vocabolario con le principali nozioni comuni an VCS:

- **DIFF**: insieme di righe modificate su un singolo file;
- **COMMIT**: insieme di DIFF che sono stati esplicitamente convalidati, possono esistere in locale o in remoto e rappresentano una nuova versione del codice;
- **HEAD**: ultimo commit presente sulla storia;
- **BRANCH**: un puntatore ad un singolo commit, anche qui abbiamo una HEAD ossia il puntatore all'ultima commit nel **master branch**;
- **MERGE**: operazione che permette di integrare un branch;
- **PULL-REQUEST**: un modo per gestire una merge di un branch verso il **master**, chiede di eseguire il merge nel branch master, da la possibilitá di rivedere le modifiche e si chiude la pull quando viene chiusa o si termina la merge;

2.6 Tipologie di workflow

(Def.) Il **workflow** é il processo che utilizziamo per gestire le modifiche che sviluppiamo nel nostro codice. Dipendono dal sistema di VCS.

2.6.1 Centralized workflow

Questo pattern viene usato nei CVCS, é facile da capire e da usare ma la collaborazione viene bloccata quando il server centrale é inattivo. I file su cui si lavora sono posti in **checkout** bloccando gli altri utenti nella merge, che é effettuata nel server centrale.

2.6.2 Feature branch workflow

Obiettivo di questo pattern é quello di usare un **branch per feature** (DVCS). L'incapsulamento consente di lavorare senza disturbare la codebase principale, consente una collaborazione piú semplice e in conflitti in merge mappano i conflitti concettuali.

2.6.3 Gitflow model

Mantiene il **master branch** come ramo del codice rilasciato e usa il **develop branch** come uno snapshot corrente del codice che vorremo portare in rilascio. Genera un **feature branch** di sviluppo per ogni feature e fará la merge quando sará pronta, effettuandola sul master e inserendo un tag per la versione.

2.6.4 GitHub flow

Un approccio piú veloce di sviluppo che si focalizza sulle caratteristiche per unire i nuovi rami con il ramo master: ideale flusso di lavoro per i piccoli team e progetti. Il ramo master é sempre pronto per il deploy e ció assicura di poter ripristinare rapidamente lo stato se qualcosa va storto.

2.6.5 GitLab flow

Approccio di sviluppo attento all'affidabilitá, in cui tutte le funzionalitá e le correzioni vanno al master prima di passare in **pre-production** e **production**. Ció implica un processo di test a piú fasi in cui una singola revisione del codice sulla richiesta di merge non é sufficiente.

2.6.6 Forking workflow

Modello ereditato da GitHub che spinge i concetti di file system condivisi, in cui ogni utente effetua una **fork** del master e propone delle pull request tra repository. Questo modello migliora la gestione delle autorizzazioni e l'autonomia per migliore collaborazione. Infine permette la decentralizzazione per utilizzare diversi pattern.

3 GIT

(Def.) **GIT** é un software di controllo versione distribuito (DVCS) utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005. Nacque per essere un semplice strumento per facilitare lo sviluppo del kernel Linux ed é diventato uno degli strumenti di controllo versione piú diffusi.

Le sue principali caratteristiche sono:

- **Branching & Merging**: incetiva lo sviluppo su branch diversi (locali o condivisi);
- **Piccolo e veloce**: sviluppato in C, esegue la maggior parte delle operazioni in locale;
- **Distribuito**: possibilitá di backup multipli e di adottare diversi workflow;
- **Integritá**: ogni commit é identificata da un ID con checksum SHA-1, non é possibile modificare una commit senza cambiarne l'ID stesso e dei commit successivi;
- **Staging area**: é presente un'area di staging dove vengono validati i file modificati che potranno essere versionati con una commit;
- **Free & Open Source**: rilasciato con licenza pubblica GNU ed il codice sorgente é rilasciato pubblicamente;

3.1 Aree locali

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 ancora non committati (il comando *git add* salva uno snapshot di tutti i file nella staging area);
- nel **repository locale** ossia committed;

3.2 Stato di un file

Un file in GIT puó essere in uno dei seguenti tre stati:

- **Untracked**: file nuovo non ancora versionato;
- **Unmodified**: file non modificato rispetto alla versione precedente;
- **Modified**: modificato nella working directory;
- **Staged**: salvata una snapshot nella staging area;
- **Committed**: preso dallo staging area e salvato nel repository locale;

4 Framework SCRUM

(Def.) **SCRUM** é un processo agile che nasce per lo sviluppo di problemi complessi e che ci permette di concentrarci sulla consegna del maggior valore di business nel piú breve tempo. Permette di ispezionare il software funzionante rapidamente e ripetutamente (ogni due settimane o ogni mese).
Il business stabilisce le prioritá e i team si organizzano per scegliere la strada migliore per consegnare le funzionalitá ad alta prioritá. Ogni due settimane o ogni mese chiunque puó vedere il software funzionante e decidere se rilasciarlo o migliorarlo.

4.1 Pilastri dello SCRUM

Il framework SCRUM si basa su tre pilastri fondamentali:

- **Trasparenza**: ad esempio l'utilizzo di un linguaggio comune per una conoscenza condivisa;

- **Controllo**: ad esempio ispezioni pianificate per prevenire variazioni non desiderate;
- **Adattamento**: ad esempio effettuare aggiustamenti per minimizzare ulteriori deviazioni tramite feedback continuo;

4.2 Caratteristiche dello SCRUM

Il framework SCRUM presenta diverse caratteristiche quali:

- Gruppi che si **auto-organizzano**;
- Il prodotto evolve attraverso **sprint** mensili (o comunque di durata fissa);
- I requisiti sono tratta come elementi di una lista della **product backlog**;
- Non vengono prescritte particolari pratiche ingegneristiche;
- Si basa sull'attivitá **empirica**, cioé la conoscenza si basa sull'esperienza e le decisioni si basano su ció che é conosciuto;
- Processo **iterativo** e **incrementale** per ottimizzare il controllo dello sviluppo e il controllo del rischio;

4.3 Sprint SCRUM

I progetti che adottano SCRUM progrediscono attraverso una serie di **sprint**:

- **Durata costante**: ogni sprint ha una durata costante (2-4 settimane solitamente) e ció favorisce un ritmo migliore;
- **Prodotto**: il prodotto é progettato, realizzato e testato durante lo sprint

4.4 Ruoli nel framework SCRUM

4.4.1 Product Owner

(Def.) Il **product owner** é colui che definisce le caratteristiche del prodotto e rappresenta il desiderio del committente.

Chi ricopre questo ruolo ha i seguenti compiti:

- **Date**: decide date e contenuto del rilascio;
- **ROI**: é responsabile della redditivitá del prodotto (ROI);
- **Prioritá**: definisce le prioritá delle caratteristiche del prodotto in base al valore che il mercato gli attribuisce;
- **Adattabilitá**: adegua le caratteristiche e le prioritá ad ogni iterazione secondo quanto necessario;
- **Responsabilitá**: é il responsabile del product backlog
- **Decisione**: accetta o rifiuta i risultati del lavoro;

4.4.2 Scrum Master

(Def.) Lo **SCRUM master** é colui che rappresenta la conduzione del progetto ed é il responsabile dell'adozione dei valori e delle pratiche SCRUM. prodotto e rappresenta il desiderio del committente.

Chi ricopre questo ruolo ha i seguenti compiti:

- **Ostacoli**: colui che rimuove gli ostacoli;
- **Osservazione**: si assicura che il gruppo di lavoro sia pienamente operativo e produttivo;
- **Cooperazione**: favorisce la cooperazione tra tutti i ruoli e le funzioni;
- **Protezione**: protegge il gruppo di lavoro da interferenze esterne;
- **Aiutante**: aiuta il product owner e il team di sviluppo condividendo la gestione e le decisioni del team;

4.4.3 Development Team

(Def.) Il **development team** rappresenta il gruppo di persone responsabili per la realizzazione dell'incremento in conformitá alla definizione di **Done** (il pilastro della trasparenza). Si compone di 5-9 persone con competenze trasversali (programmatori, tester, UX, etc.) e lavorano in full time (con eccezioni come una DB admin). Il gruppo infine si auto-organizza.

4.5 Eventi nel framework SCRUM

4.5.1 Sprint planning

(Def.) Lo **sprint planning** é un evento (di circa 8 ore per sprint di 1 mese) in cui il team seleziona gli item dal product backlog che puó impegnarsi a completare. Viene creato uno **sprint backlog** collaborativamente da tutto il team, vengono identificati **tasks** e ne vengono stimati i tempi di completamento.

4.5.2 Daily SCRUM meeting

(Def.) Anche detto **stand-up meeting** é un incontro giornaliero in piedi, della durata di 15 min, fatto per sincronizzarsi su quanto fatto e pianificare la giornata per il raggiungimento dello sprint goal. Viene aggiornata la SCRUM board, aiuta ad evitare ulteriori riunioni non necessarie e vi puó partecipare anche il product owner.

4.5.3 Sprint review

(Def.) La **sprint review** é un evento (di circa 4 ore per sprint di 1 mese) in cui il gruppo di lavoro presenta quanto realizzato durante lo sprint. Viene accettato e validato il lavoro, tipicamente sotto forma di **demo** delle nuove caratteristiche o della nuova architettura. É un incontro informale a cui partecipa tutto il gruppo (e anche gli esterni).

4.5.4 Sprint retrospective

(Def.) La **sprint retrospective** viene celebrata dopo la sprint review e prima del prossimo sprint planning. É un evento (di circa 3 ore per sprint di 1 mese) in cui si valuta ció che sta e non sta funzionando (miglioramenti al

prodotto, applicazione della definizione di Done, miglioramenti al prossimo sprint) e vi partecipano tutti i ruoli.

4.6 Artifatti nel framework SCRUM

4.6.1 Product backlog

(Def.) Il **product backlog** é una lista (dinamica, che evolve assieme al prodotto) di tutti i *desiderata*, ossia tutti i requisiti, funzionalitá, miglioramenti e fix da realizzare nei prossimi rilasci. Viene idealmente espressa in modo che ciascun elemento abbia valore per gli utenti o i clienti del prodotto.

Le prioritá sono assegnate dal product owner mentre il dev team stima ogni item. Le prioritá sono rivalutate all'inizio di ogni sprint con il development team. I componenti principali del product backlog sono le **user stories**, le quali andranno scomposte in task.

4.6.2 Sprint backlog

(Def.) Lo **sprint backlog** rappresenta la lista di tutti i desiderata da svolgere durante lo sprint. Ogni componente del gruppo sceglie le proprie task da svolgere, e puó modificare, aggiungere o cancellare parti del backlog. La stima del lavoro rimanente é aggiornata ogni giorno.

4.6.3 Definition of Done

(Def.) Definisce il significato di **fatto** per uno sprint item, ed é il minimo srt di attivitá per definire che un attivitá é completata. Puó variare per gruppo di lavoro e deve essere chiaro per tutti i membri. Viene utilizzato per verificare se un attivitá é da ritenersi completata.

4.6.4 Acceptance Criteria

(Def.) Permette di confermare se la storia é completa e funziona come voluto. Rappresentati da frasi semplici condivise tra product owner e development team, possono essere incluse con la user story e rimuovono l'ambiguitá dei requisiti.

5 Build Automation

(Def.) La **build automation** é il processo che automatizza la creazione di una software build e dei suoi processi associati come compilazione del codice sorgente, packaging dei file binari ed esecuzione dei test automatizzati.

Appartengono a questa definzione due categorie generali di strumenti:

- **Build automation utility**: software come **Make**, **Maven**, **MS build**, etc., il cui scopo é quello di generare artefatti attraverso attivitá come compilazione e linking di codice sorgente;
- **Build automation servers**: sono strumenti **web based** che eseguono build automation su base programmata o in base a certi eventi (**trigger**);

5.1 Processo di Build

(Def.) Il **processo di build** é un insieme di passi che trasformano gli script di build, codice sorgente, file di configurazione, documentazione e test in un software distribuibile.

5.1.1 Caratteristiche CRISP

L'acronimo **CRISP** riflette le seguenti caratteristiche:

- **Completo**: indipendente da fonti non specificate nello script di build;
- **Ripetibile**: accede ai file contenuti nel sistema di gestione del codice sorgente (una esecuzione ripetuta dá lo stesso risultato);
- **Informativo**: fornisce informazioni sullo stato del prodotto;
- **Schedulabile**: puó essere programmato ad una certa ora e fatto eseguire automaticamente;
- **Portabile**: indipendente il piú possibile dall'ambiente di esecuzione;

5.2 Maven

(Def.) **Maven** é uno strumento di gestione e comprensione dei progetti software, basato sul concetto di **Project Object Model** (POM), Maven gestisce la compilazione, la reportistica e la documentazione di un progetto da un'informazione centrale.

Sfrutta il paradigma **convention over configuration** che prevede una configurazione minima (o assente) per il programmatore che utilizza un framework, obbligandolo a configurare solo gli aspetti che si **differenziano** dalle implementazioni standard.

5.2.1 Caratteristiche

Le 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 del progetto vengono specificate nel file di configurazione (pom.xml), Maven si occupa di scaricarle in automatico da dei repository remoti e salvarli in locale;
- **Remote repositories**: sono stati definiti dei repository remoti dove sono presenti gran parte delle librerie di progetti open source e dei plugin utilizzati da Maven per implementare ed estendere le fasi del build cycle;
- **Universial reuse of build logic**: i plugin permettono di definire in modo riusabile i principali aspetti richiesti per la gestione del progetto (esecuzione del processo di build, esecuzione del framework di test, creazione di template di progetto);

5.2.2 Build Lifecycle

Il processo per la **costruzione** e **distribuzione** di un particolare progetto é chiaro e definito. Per chi esegue la build di un progetto é sufficiente imparare pochi comandi per costruire qualsiasi progetto Maven (il POM si assicurerá di ottenere i risultati attesi).

Esistono tre tipi di build lifecycle predefiniti:

- **Default**: gestisce la distribuzione del progetto;
- **Clean**: gestisce la pulizia del progetto;
- **Site**: gestisce la creazione della documentazione del sito del progetto;

5.2.3 Default build lifecycle

In particolare il processo default compie i seguenti passi:

- **Validate**: convalidare la correttezza del progetto e la disponibilitá di tutte le info necessarie;
- **Compile**: compilare il codice sorgente del progetto;
- **Test**: testare il codice sorgente compilato usando un framework di unit testing;
- **Package**: confezionare il codice compilato nel suo formato distribuibile;
- **Verify**: esegue eventuali controlli sui risultati di test e di integrazione per garantire i criteri di qualitá;
- **Install**: installa il pacchetto nel repository locale, per utilizzarlo come dipendenza per altri progetti a livello locale;
- **Deploy**: eseguito nell'ambiente di compilazione, copia il pacchetto finale nel repository remoto per condividerlo con altri sviluppatori e progettisti;

5.3 POM

(Def.) Un **Project Object Model** é l'unitá fondamentale di lavoro in Maven. Si tratta di un file XML che contiene informazioni sul progetto e sui dettagli di configurazione utilizzati da Maven per creare il progetto.

Alcune delle informazioni reperibili nel POM sono:

- **Project ID**: ID gruppo, ID artefatto e versione;
- **Dependencies**: le dipendenze del progetto;

- **Plugin**: i plugin o gli obiettivi che possono essere eseguiti;
- **Profiles**: profili di configurazione;
- **Other**: altre info come versione, descrizione, sviluppatori, etc.;

5.3.1 Project Archetypes

(Def.) Un **archetype** é un toolkit per modelli di progetto Maven, ed é definito come un modello da cui vengono realizzate tutte le altre cose dello stesso tipo.

6 Software testing

(Def.) Il **software testing** é un'indagine condotta per fornire alle parti interessate informazioni sulla qualitá del software, prodotto o servizio. Puó anche fornire una visione obiettiva ed indipendente del software per consentire ad un'azienda di apprendere e comprendere i rischi dell'implementazione software.

6.1 Testing

(Def.) Definiamo come **testing** il processo costituito da tutte le attivitá (statiche o dinamiche) sulla pianificazione, la preparazione e la valutazione dei prodotti software e dei prodotti di lavoro correlati per determinare che soddisfino i requisiti specificati, per dimostrare che sono adatti allo scopo e per rilevare difetti.

6.1.1 Difetti nel software

I difetti nel software possono essere inseriti sia dal programmatore sia dall'analista:

- **Programmatore 45%**: errori in fase di sviluppo che comporta un comportamento inatteso durante l'esecuzione;
- **Analista 20% requisiti/25% progettazione**: interpretando male un requisito la progettazione e la codifica del programma sono influenzate da questo errore;

6.1.2 Categorie di testing

Di seguito si elencano le principali categorie di testing:

- **Funzionale**: test condotti per valutare la conformitá di un componente o un sistema ai **requisiti funzionali** (cosa fa la nostra applicazione);
- **Non funzionale**: test condotti per valutare la conformitá di un componente o un sistema ai **requisiti non funzionali** (come la nostra applicazione risponde alle esigenze);
- **Statico**: testare un prodotto di lavoro **senza eseguire** il codice (analisi statica, analisi dei documenti e analisi dei requisiti);
- **Dinamico**: collaudo eseguito durante l'**esecuzione** del programma o del sistema;
- **Verifica**: il prodotto é stato realizzato secondo le specifiche tecniche e funziona correttamente (conforme ai suoi obiettivi);
- **Validazione**: il prodotto é stato realizzato rispettando le specifiche dell'utente (requisiti);

6.2 Processo di test

Il **processo di test** si articola nelle seguenti fasi:

- **Test planning**: definizione o aggiornamento di un piano di test;
- **Test control**: azioni correttive e di controllo se il piano non viene rispettato;
- **Test analysis**: cosa testare;
- **Test design**: come testare;
- **Test implementation**: attivitá propedeutica all'esecuzione (es. definizione casi di test);
- **Test execution**: esecuzione del test;
- **Checking result**: verificare i risultati e i dati collezionati dalla test execution per capire l'esito del test;

- **Evaluating exit criteria**: verificare se sono stati raggiunti gli exit criteria deifniti nel test plan;
- **Test result reporting**: riportare il progresso rispetto agli exit criteria definiti nel test plan;
- **Test closure**: chiudere il processo e definire azioni di miglioramento;

6.3 Principi di test

6.3.1 Test deve rilevare i difetti

Testare un'applicazione puó solo rivelare che **uno o piú difetti** esistono nell'applicazione: il test non puó mostrare che l'applicazione sia priva di errori. Pertanto é importante progettare casi di test per trovare il maggior numero possibile di difetti.

6.3.2 Test esaustivi non esistono

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 base sono:

- **Risk based testing**: sulle funzionalitá che hanno un impatto sul business;
- **Requirement based**: basato sui requisiti;

6.3.3 Test il prima possibile

Avviare la fase di test il prima possibile permette di risparmiare sui costi del progetto: il processo di test non deve essere eseguito quando il progetto è al termine, ma deve andare in parallelo con il processo di sviluppo.

6.3.4 Clustering dei difetti

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. Questa è l'applicazione del principio di Pareto ai test del software: circa l'80% dei problemi si trova nel 20% dei moduli.

6.3.5 Il 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 introdotto vecchi errori. Tuttavia, anche questi casi di test di non regressione devono essere modificati per riflettere le modifiche apportate nel software per essere applicabili.

6.3.6 I test dipendono dal contesto

Diverse metodologie, tecniche e tipi di test sono legati al tipo e alla natura dell'applicazione.

6.3.7 Assenza di errori non é garanzia

Solo perché il test non ha riscontrato alcun difetto nel software, non significa che il software sia perfetto e pronto per essere rilasciato.

6.4 V-Model

(Def.) Il **V-Model** é un modello sequenziale del ciclo di vita dello sviluppo che descrive una relazione uno-a-uno tra le principali fasi dello sviluppo del software, dalla specifica dei requisiti aziendali alla consegna, e i livelli di test corrispondenti, dal test di accettazione al test dei componenti.

6.4.1 Unit testing

Verificano il piú piccolo sottosistema possibile che puó essere testato separatamente. Sono veloci da eseguire e ogni modifica del codice sorgente dovrebbe

scatenare uno unit test. Sono indipendenti tra di loro e non dipendono dall'ordine di esecuzione. Il **System Under Test** (SUT) é considerato come una white box.

6.4.2 Integration testing

Verificano l'integrazione tra piú sub-systems (**interni**, ossia giá verificati dagli unit testing, o **esterni**, come database, file system, etc.). Sono piú lenti da configurare ed eseguire, ed anche in questo caso il SUT é una white box.

6.4.3 System testing

Verificano il comportamento dell'intero sistema: lo scopo principale é la verifica rispetto alle specifiche tecniche. Il SUT puó essere una white box o black box. Un esempio di system testing sono gli **smoke test**:

- mirano ad individuare gli errori il prima possibile;
- verificano le funzionalitá del SUT;

6.4.4 Acceptance testing

Anche conosciuti come **User Acceptance Testing** (UAT) sono una suite di test su tutto il SUT, relativo agli use cases e ai requisiti concordati con l'utente/cliente. Sono svolti assieme a loro e il SUT é considerato una black box.

7 Unit testing

(Def.) Lo **unit testing** é il collaudo delle singole unitá software. Per unità si intende normalmente il minimo componente di un programma dotato di funzionamento autonomo. Vengono sviluppati dal programmatore che sviluppa le unità, per verificare l'assenza di alcuni errori e documentare il comportamento dell'unità prodotta.

7.1 Proprietá A-TRIP

7.1.1 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 essere eseguiti:

- **In modo rapido**: I test di unità devono essere semplici e la loro esecuzione non deve impiegare più di pochi secondi;
- **Senza 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.

7.1.2 Thorough

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:

- **Percentuale di righe** che vengono esercitate attraverso unit test;
- **Percentuale di possibili** diramazioni che vengono eseguiti dagli unit test;
- **Numero di eccezioni** che vengono controllare attraverso i test;

7.1.3 Repeatable

I test di unità devono produrre sempre lo stesso risultato. Per essere ripetibili, i test di unità devono avere le seguenti caratteristiche:

- **Essere indipendenti dall'ordine di esecuzione**: l'ordine di esecuzione dei test non deve influenzare il risultato;
- **Essere indipendente dall'ambiente di esecuzione**: l'esecuzione dei test non deve dipendere da risorse esterne al progetto o da risorse non gesiste nel VCS.

7.1.4 Independent

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 un'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.

7.1.5 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.

7.2 Framework

Per creare i test di unità si sfruttano dei framework, con le seguenti caratteristiche:

- Configurazione dell'**ambiente di esecuzione** del test;
- **Selezionare** un test o un'insieme di test da eseguire;
- Analizzare i **valori attesi** prodotti dalle unitá;
- Eseguire ed esprimere un esito, ossia se il test é **superato**, **fallito** o se sono stati prodotti **errori**;

7.3 Right BICEP

7.3.1 Are The Results Right?

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 siano chiari, o possano 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.

7.3.2 Boundary Conditions

Solitamente gli errori accadono in condizioni limite. Identificare le condizioni limite è una delle parti più importanti per creare delle buone unità:

- **Conformance**: i valori sono conformi al formato atteso?
- **Ordering**: i valori seguono o non seguono un ordine?
- **Range**: i valori sono all'interno di una valore di minimo e massimo appropriato?
- **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, etc.)?
- **Cardinality**: i valori sono nella quantitá desiderata?
- **Time**: i valori rispettano un ordine temporale?

7.3.3 Check Inverse Relationships

Alcune unità possono o devono essere verificate tramite l'applicazione della loro funzionalità inversa. Ad esempio 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.

7.3.4 Cross-check Using Other Means

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

7.3.5 Force Error Conditions

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.

7.3.6 Performance Characteristics

I test di unità devono essere veloci perché devono poter essere eseguiti molto spesso.

7.4 TDD

(Def.) Il **Test Driven Development** (TDD) é un modello di sviluppo 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 predisposti.

7.4.1 Fase rossa

La prima fase del TDD é detta **fase rossa**, in cui il programmatore scrive un test automatico per la nuova funzione da sviluppare, che deve fallire in quanto la funzione non é stata ancora realizzata.

7.4.2 Fase verde

La seconda fase del TDD é detta **fase verda**, in cui il programmatore sviluppa la quantitá minima di codice necessaria per passare il test.

7.4.3 Fase grigia

La terza fase del TDD é detta **fase grigia**, in cui il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualitá.

8 Analisi statica

(Def.) L'analisi statica del codice è l'analisi del software che viene eseguita **senza effettivamente** eseguire i programmi. Nella maggior parte dei casi l'analisi viene eseguita su una versione del codice sorgente. Appartengono a questa categoria di test: **test statico**, **white box**, **test non funzionale**.

8.1 Automated Code Review

(Def.) La revisione automatica del codice verifica la **conformità** del codice sorgente a un set predefinito di regole o **best practice**. Questo processo può essere eseguito sia **manualmente** che in modo **automatizzato**. Con l'automazione, gli strumenti software forniscono assistenza nel processo di revisione e ispezione del codice. Un programma di revisione può anche fornire un modo automatizzato o assistito da un programmatore per correggere i problemi riscontrati.

8.2 Teoria delle Finestre Rotte

(Def.) La teoria delle finestre rotte è una teoria criminologica sulla capacità del disordine urbano e del vandalismo di generare criminalità aggiuntiva e comportamenti antisociali. La teoria afferma che mantenere e controllare ambienti urbani reprimendo i piccoli reati, gli atti vandalici, la deturpazione dei luoghi, il bere in pubblico, la sosta selvaggia o l'evasione nel pagamento di parcheggi, mezzi pubblici o pedaggi, contribuisce a creare un clima di ordine e legalità e riduce il rischio di crimini più gravi.

8.3 Static Analysis Tools

(Def.) Un tool per l'analisi statica del codice permette, in modo simile ad un correttore ortografico, di: imporre il **rispetto** di stili, verificare **congruitá** con la **documentazione**, ricercare codice **copiato** in piú punti, ricercare gli **errori** comuni, misurare la **percentuale** di codice **testato**, etc.

8.3.1 Checkstyle

(Def.) **Checkstyle** è uno strumento di sviluppo per aiutare i programmatori a scrivere codice Java che aderisce a uno standard di codifica. Automatizza il processo di controllo del codice Java per risparmiare agli sviluppatori questo noioso (ma importante) compito.

Checkstyle può controllare molti aspetti del tuo codice sorgente. Può trovare problemi di progettazione di **classi**, problemi di progettazione di **metodi** ed ha anche la capacità di controllare i problemi di **layout** e **formattazione** del codice.

8.3.2 SpotBugs

(Def.) **SpotBugs**, successore di FindBugs, è un programma che utilizza l'analisi statica per cercare bug nel codice Java.

8.3.3 PMD

(Def.) **PMD** è un analizzatore di codice sorgente statico. Trova difetti di programmazione comuni come **variabili inutilizzate**, blocchi catch **vuoti**, creazione di oggetti **non necessari**, copia e incolla e così via. Si occupa principalmente di Java e Apex, ma supporta altri linguaggi.

8.3.4 SonarQube

(Def.) **SonarQube** è uno strumento di revisione automatica del codice per rilevare bug, vulnerabilità. Può integrarsi con il flusso di lavoro esistente per consentire l'**ispezione continua** del codice tra **branch** del progetto e **pull** request.

SonarQube è in grado di registrare la cronologia delle metriche e di fornire grafici di evoluzione. SonarQube fornisce analisi e integrazione completamente automatizzate. SonarQube puó:

- Storicizzare l'andamento della qualità, permettendo di verificare se il progetto sta migliorando o meno;
- PErmette di stabilire un'insieme di regole al progetto (**quality profile**) e verifica se la qualitá del progetto le rispetta (**quality gate**);

- Classifica le issue in base alla gravitá (Blocker, Critical, Major, Minor, Info);
- Classifica le issue in **vulnerabilitá** (sicurezza), **bug** (affidabilitá), **code smell** (mantenibilitá);

9 Artifact Repository

(Def.) Un gestore di **repository binari** è uno strumento software progettato per ottimizzare il **download** e l'**archiviazione** dei file binari utilizzati e prodotti nello sviluppo del software. **Centralizza** la gestione di tutti gli **artefatti binari** generati e utilizzati dall'organizzazione per superare la complessità derivante dalla diversità dei tipi di artefatti binari, dalla loro posizione nel flusso di lavoro complessivo e dalle dipendenze tra di essi.

9.1 Caratteristiche

Le caratteristiche di un artifact repository sono:

- Permette di effettuare **ricerche** e reperire **informazioni** riguardanti i prodotti;
- Permette di gestire e associare **permessi** di accesso sui prodotti;
- Permette di segnalare **vulnerabilitá** sui prodotti;
- Permette di verificare **problemi** legati a **licenze**;
- Permette di **documentare** gli artefatti con dei **metadati**;

9.2 Maven Repository Management

Mette a disposizione le seguenti funzionalitá:

- **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;
- **Release Artifacts**: gli artefatti di tipo Release solitamente possono essere rilasciati una sola volta e vengono mantenuti nel repository;
- **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;

9.2.1 Caratteristiche di Maven Repository

I principali vantaggi dell'uso di **Maven Repository** sono:

- **Velocitá di build**: le dipendenze e gli artefatti di tipo plugin utilizzati nel processo di build vengono scaricati dalla rete interna;
- **Stabilitá 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;
- **Collaborazione**: si evita di far creare gli artefatti dal VCS ed è più semplice identificare le versioni di rilascio certificate;

10 Continous Integration

(Def.) Nell'ingegneria del software, la **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).

Le principali motivazioni per l'uso della CI é:

- Per lunghi periodi di tempo, durante il processo di sviluppo, il progetto non è in uno stato funzionante o in uno stato utilizzabile. Soprattutto 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;

10.1 Integration Hell

(Def.) La fase di integrazione può richiedere **molto tempo**, e nel caso peggiore, nessuno ha modo di prevedere quando terminerà questa fase e, di conseguenza, quando l'applicazione può essere rilasciata.

10.2 Visione generale del processo

Al completamento di un'attivitá viene **costruito il prodotto** (ad ogni commit nel **VCS** viene eseguito il processo di build). Se il processo di costruzione fallisce l'**attivitá non continua** fino a che il prodotto non viene riparato. Se **non é possibile** riparare il prodotto immediatamente si ritorna all'**ultima versione funzionante**. In questo modo si assicura la presenza di un **prodotto consistente** potenzialmente pronto per essere validato e rilasciato.

10.2.1 Prerequisiti

Per implementare la pratica di continuous integration é necessario che:

- Il codice del progetto venga gestisto in un **VCS**;
- Il processo di build sia **automatico**;
- Il processo di build esegua delle **verifiche automatiche** (unit test, integration test, static code analysis);
- Il team di sviluppo adotti corretamente questa pratica;

10.3 Processo di CI

Il processo di CI in dettaglio:

- 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;
- 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;
- Eseguo il processo di **build** in **locale** in modo da verificare che tutto funzioni correttamente;
- Se il processo di build termina con **successo** invio le modifiche al **VCS**;
- Attendo che il sistema di CI esegua il processo di build con i miei cambiamenti;
- Se il processo di buil **fallisce** mi **fermo** con le attività di sviluppo, e lavoro per **sistemare** il problema in locale e riprendo dal processo di build in locale;
- Se il processo di build termina con **successo** passo allo sviluppo dell'**attività successiva**;

10.4 Best Practices

10.4.1 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.

10.4.2 Creare una Automated Test Suite

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:

- Gli **Unit Test**;
- Gli **Integration Test** di subsystem interni;
- **Static code analysis**;

Devono avere le caratteristiche del modello CRISP.

10.4.3 Avere un processo di Build & Test breve

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

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

10.4.4 Gestire il nostro ambiente di sviluppo

Ogni sviluppatore deve essere in grado di:

- Eseguire localmente il processo di build, test e deploy, quindi per questo nel VCS devono essere gestiti: **codice di produzione**, **codice di test**, **script di configurazione**;
- **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;

10.4.5 Fix una build in errore subito

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 **commentando** le verifiche che hanno fatto fallire la build.

10.4.6 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;

Il sistema di continuous integration deve poter **avvisare** (con delle notifiche) i componenti del team ad ogni cambio di stato del processo.

10.5 Jenkins

(Def.) Jenkins è uno strumento **open source** di supporto allo **sviluppo software** scritto in linguaggio Java. Fornisce dei servizi di **integrazione continua** per lo sviluppo del software. Viene eseguito lato **server** all'interno di un **server web** che supporta la tecnologia Servlet e quindi può essere utilizzato da remoto all'interno di un Web browser.

Le principali caratteristiche di Jenkins sono:

- **Automazione delle Build**: Automatizza la creazione del software con script per build consistenti;
- **Trigger Automatici**: Avvia build automaticamente in risposta a eventi (es. commit, modifiche, orari);
- **Automazione dei Test**: Esegue test automatici (unitari, integrazione, ecc.) ad ogni build;
- **Integrazione con VCS**: Compatibile con sistemi di controllo versione per monitorare e reagire alle modifiche;
- **Notifiche e Report**: Invia notifiche e genera report sui risultati delle build e dei test;
- **Pipeline come Codice**: Consente la configurazione delle pipeline con Jenkinsfile versionabili;
- **Integrazione con Altri Strumenti**: Supporta strumenti come Docker, Kubernetes, Ansible, Terraform;
- **Storico delle Build**: Registra tutte le esecuzioni con log e risultati dettagliati;
- **Scalabilità e Flessibilità**: Supporta build distribuite e pipeline personalizzabili;
- **Sicurezza**: Gestione avanzata di ruoli e permessi per utenti e gruppi;

10.5.1 Jenkins Pipeline

(Def.) Una **pipeline** in Jenkins è una serie di **fasi** o passaggi che rappresentano il processo di **continous integration** (CI) e **continous delivery** (CD) per un progetto software. È uno strumento potente e flessibile che consente di automatizzare il flusso di lavoro di sviluppo, test e distribuzione del software.

Esistono due tipi di pipeline:

- **Pipeline scriptate**: scritte in Groovy, offrono una grande flessibilità e possono essere utilizzate per pipeline complesse e dinamiche;

- **Pipeline dichiarative**: forniscono una sintassi più strutturata, ideale per pipeline più semplici e per chi preferisce un approccio più dichiarativo;

10.6 GitHub Actions

(Def.) GitHub Actions è una funzionalità di GitHub che consente di **automatizzare** i flussi di lavoro per la CI e CD.

Le sue principali caratteristiche sono:

- **Integrazione nativa con GitHub**: è progettato per funzionare all'interno di GitHub, consentendo automazioni basate su eventi GitHub come commit, pull request, e rilasci;
- **Azioni personalizzate e da terzi**: é possibile creare azioni personalizzate o utilizzare una vasta gamma di azioni predefinite da GitHub o da terzi, rendendo semplice l'integrazione con altri strumenti;
- **Integrazione con piattaforme cloud e strumenti DevOps**: può integrarsi con servizi cloud (come AWS, Azure, Google Cloud) e strumenti DevOps (come Docker, Kubernetes, Terraform);
- **Integrazione con Notifiche e Servizi Esterni**: offre integrazioni con servizi di notifica (ad esempio, Slack, Microsoft Teams) e può eseguire chiamate API a servizi esterni;

11 Continuous Delivery

(Def.) **Continuous Delivery** (CD) è un approccio di ingegneria del software in cui i team producono software in cicli **brevi**, garantendo che il software possa essere rilasciato in modo **affidabile** in qualsiasi momento e, quando si rilascia il software, **manualmente**. Mira a **creare**, **testare** e **rilasciare** software con maggiore **velocità** e frequenza. L'approccio consente di ridurre i **costi**, i **tempi** e i **rischi** legati alla distribuzione delle modifiche, consentendo un maggior numero di **aggiornamenti** incrementali per le applicazioni in produzione.

Si sta eseguendo Continuous Delivery quando:

- Il tuo software è rilasciabile per tutto il suo ciclo di vita;
- Il tuo team dà la priorità a mantenere il software distribuibile piuttosto che lavorare su nuove funzionalità;
- Chiunque può ottenere un feedback rapido e automatizzato sulla disponibilità di produzione dei propri sistemi ogni volta che qualcuno apporta una modifica ad essi;
- È possibile eseguire distribuzioni rapide di qualsiasi versione del software in qualsiasi ambiente su richiesta;
- Si ottiene la Continuous Delivery integrando continuamente il software eseguito dal team di sviluppo, creando eseguibili ed eseguendo test automatizzati su tali eseguibili per rilevare i problemi. Inoltre, si rilascia il software in ambienti sempre più simili alla produzione per garantire che il software funzioni in produzione;

11.1 Misurare per migliorare

(Def.) 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: controllare le prestazioni e migliorarne l'efficienza.

11.2 Deployment pipeline

(Def.) La **Deployment Pipeline** é 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;
- E' guidato dal concetto di Fail-Fast;

Una pipeline é composta da **stages** mappate su attivitá misurabili e la transizione tra due stage avviene attraverso un **gate** automatico o manuale. Gli stages possono essere eseguiti in **parallelo** o in **sequenza** e possono essere multidirezionali.

11.3 Requisiti della CD

I requisiti principali per integrare una CD pipeline sono:

- **VCS**, **Build automation**, **Unit Testing**, **Artifact Repository**;
- **Continuous Testing**: test automatici a livello di sistema funzionali e non;
- **Orchestratore**: sistema che ci permette di modellare le esecuzioni della pipeline (es. Jankins);
- **Configuration Management**: strumenti che permettono di gestire, tramite codice, la configurazione degli ambienti dove dovrá essere rilasciato il software;

11.4 Deployment Pipeline Practices

11.4.1 Only Build Your Binaries Once

(Def.) 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 è **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**;

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. Infine tale artefatto deve essere indipendente dall'ambiente di esecuzione.

11.4.2 Deploy the Same Way to Every Environment

(Def.) 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 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 e vanno tenuti separati dalle configurazioni.

11.4.3 Smoke-Test Your Deployments

(Def.) Per **verificare** se il rilascio automatico è andato **bene** prevedere l'**esecuzione** di smoke-test. Gli smoke-test devono verificare anche il corretto funzionamento dei **sub-system** esterni. Gli smoke-test sono **veloci** e **semplici** da realizzare e permettono di far fallire velocemente la pipeline in caso di problemi.

11.4.4 Deploy into a Copy of Production

(Def.) 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.
Tale ambiente dovrá 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;

11.4.5 Each Change Should Propagate through the Pipeline Instantly

(Def.) Ogni **modifica** al codice sorgente deve avviare il processo di **Deploy**. Molto probabilmente la pipeline impiegherà **tanto tempo** per eseguire l'in-

tero processo. Per questo è necessario inserire delle **verifiche** negli stages che la facciano fallire il **prima possibile**.

11.4.6 If Any Part of the Pipeline Fails, Stop the Line

(Def.) 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.

12 Configuration Management

(Def.) La **Configuration management** (CM) è un processo di ingegneria dei sistemi per **stabilire** e **mantenere** la coerenza delle prestazioni, degli attributi funzionali e fisici di un prodotto con i suoi **requisiti**, la **progettazione** e le **informazioni** operative per tutta la sua vita.

12.1 Definizione ITIL

(Def.) L'obiettivo del **Configuration Management** è di fornire un modello **logico** dell'infrastruttura attraverso l'**identificazione**, il **controllo**, la **gestione** e la **verifica** di tutte le **versioni** di "Configuration Items" esistenti. Il configuration item (CI) è un unitá di configurazione che puo' essere gestita individualmente (es. computer, routers, servers, software, etc...).
Un elemento chiave del processo è il **Configuration Management** Database (CMDB), che viene utilizzato per tracciare tutti i CI e le relazioni tra di loro.

12.2 Componenti del CM

I principali componenti del Configuration Management moderno sono:

- **GIT** e i suoi branch.
- **Package managers** e **task managers**.
- Tool di **configurazione**.
- Tool di **testing**.

- Script di **configurazione**.
- Pattern per l'update dell'**infrastruttura**.

12.3 Infrastructure update pattern

12.3.1 Living infrastructure

(Def.) La **living infrastructure** è un'infrastruttura mutabile, che può essere modificata direttamente nel tempo, senza essere completamente ricreata. È l'approccio tradizionale, dove i server vengono aggiornati, riconfigurati o modificati "in loco".
I cambiamenti avvengono sugli ambienti **esistenti**, sono incrementali e progressivi. Spesso viene utilizzata in ambienti **legacy** o dove l'automazione è **limitata**.
I principali vantaggi di questo pattern sono:

- Più **economica** per ambienti piccoli o non automatizzati.
- **Non** richiede di ricreare tutto da zero per ogni modifica.

Al contrario gli svantaggi sono:

- Rischio maggiore di **errore** (modifiche manuali o non tracciate).
- **Incoerenze** tra ambienti (es. sviluppo $\neq$ produzione) e **difficoltá** di rollback.

12.3.2 Immutable infrastructure

(Def.) La **immutable infrastructure** è un'infrastruttura dove ogni modifica comporta la creazione di una nuova istanza o ambiente, piuttosto che aggiornare quello esistente. I vecchi componenti vengono eliminati e sostituiti.
Le risorse (server, container, ecc.) non vengono **mai** modificate dopo il deploy e ogni aggiornamento comporta il **provisioning** di nuove risorse (come una nuova VM o container).
I principali vantaggi di questo pattern sono:

- Maggiore **coerenza** e prevedibilità (nessuna deriva di configurazione).
- Rollback **semplice**: basta rilanciare la versione precedente.

- Perfetta **integrazione** con container (es. Docker) e orchestratori (es. Kubernetes).

Al contrario gli svantaggi sono:

- Richiede un **investimento** iniziale in automazione e strumenti (CI/CD).
- Può avere costi **maggiori** in ambienti grandi se non ben ottimizzata.

12.4 Ansible

(Def.) **Ansible** è uno strumento di Configuration Management e IT Automation: permette di definire lo stato di uno o piú server in modo **prevedibile**, **replicabile**, **consistente**. Le sue caratteristiche principali sono:

- Non richiede l'installazione di agent sulle macchine da gestire, si opera dal controller.
- Semplice ed immediato da usare, non richiede conoscenza di linguaggi di programmazione.
- Multipiattaforma e portatile.
- Leggero e performante.
- Idempotente, ossia si ottiene sempre il risultato atteso.
- Permette un approccio graduale: si può iniziare ad utilizzare con script già fatti.
- Curva di apprendimento bassa: in pochi minuti dalla lettura del manuale si è già operativi.

12.4.1 Concetti chiave

I concetti principali di Ansible sono:

- **Control Node o Controller**: una qualsiasi "macchina" ove sia installato Ansible, si possono eseguire i comandi: ansible e ansible-playbook; si può usare un qualsiasi server linux come Controller.
- **Managed Nodes**: tutti i server gestiti con Ansible, anche chiamati hosts.

- **Inventory**: é una lista di server, opzionalmente raggruppati, sui quali ansible può operare.
- **Tasks**: la singola unità di "esecuzione" su Ansible; un singolo task può essere eseguito come comando "ad hoc" con il comando ansible.
- **Handlers**: Sono istruzioni che possono essere eseguite quando un task opera una modifica al sistema. Ad esempio, dopo l'installazione di una applicazione potremmo voler avviare il servizio.
- **Playbooks**: sono insiemi di Tasks e Handlers.
- **Modules**: Sono i "verbi" del nostro linguaggio. Ci sono moduli per installare pacchetti, per copiare file, per eseguire comandi remoti.