Autore: Gabriel Rovesti

Indice

1. Issue Tracking System
2. Version Control System
3. GitHub
4. Git
5. Il Framework Scrum
6. Build Automation
7. Maven
8. Software Testing
9. Test di Unità
10. Analisi Statica del Codice
11. Continuous Integration
12. Continuous Delivery
13. Artifact Repository
14. Configuration Management
15. Container
16. Ansible

Issue Tracking System

Definizione

Un Issue Tracking System (ITS) è un sistema software che permette di gestire e mantenere liste di criticità (issue) all'interno di un'organizzazione. Un'issue rappresenta un'attività o un evento da gestire. L'ITS consente di tracciare queste attività, lasciando una "traccia" (tracking) del loro ciclo di vita.

Caratteristiche principali

- **Condivisione delle informazioni** con team, PM e cliente
- **Repository unico** per tutte le informazioni
- **Sistema di notifica** per informare gli stakeholder
- **Dashboard** per visualizzare lo stato del progetto
- **Processo misurabile** per valutare la qualità del progetto
- **Stato del progetto istantaneo**: attività da fare, in corso, completate

- **Gestione delle release** e pianificazione
- **Assegnazione e prioritizzazione** delle attività
- **Consuntivazione del tempo** impiegato (memoria storica)
- **Chiara assegnazione delle responsabilità**
- **Memoria storica** di tutti i cambiamenti

Work Item

Un Work Item rappresenta un'attività minima all'interno del progetto e contiene:

- **Progetto**: a cui si riferisce
- **Codice**: identificativo univoco
- **Riepilogo**: descrizione breve
- **Descrizione**: dettaglio esaustivo
- **Tipo**: categoria del Work Item (Bug, Task, Epic, ecc.)
- **Stato**: posizione nel workflow
- **Priorità**: importanza relativa
- **Versione di riferimento**: rilascio target
- **Componente**: parte del progetto coinvolta
- **Tag/Etichette**: classificazione ulteriore
- **Collegamenti**: relazioni con altri Work Item
- **Assegnatario**: responsabile dell'attività
- **Segnalante**: chi ha richiesto l'attività
- **Date**: creazione, ultimo aggiornamento, risoluzione
- **Stime**: originaria, a finire, tempo speso
- **Allegati**: documentazione aggiuntiva

Workflow

Il workflow definisce l'insieme di stati e transizioni che un Work Item attraversa durante il suo ciclo di vita. Implementa il processo da seguire per completare l'attività. Può essere associato a uno o più tipi di Work Item, permettendo di registrare tutte le transizioni e i cambi di stato.

Funzionalità di un ITS

1. **Gestione**:
    - Integrazione con SCM
    - Integrazione con ambiente di sviluppo
    - Ricerca avanzata
    - Esportazione
    - Reporting

2. **Condivisione**:
    - Notifiche
    - Bacheche/Board
    - Dashboard
    - Roadmap e Release Notes

Tipologie di ITS

Tra i più diffusi:

- Atlassian Jira
- GitHub
- GitLab
- Azure DevOps
- Redmine
- Trac
- Mantis
- Bugzilla
- Trello

Benefici dell'ITS

- Implementare e verificare l'adozione di un processo
- Migliorare e misurare la qualità del progetto
- Misurare la soddisfazione del cliente
- Definire responsabilità chiare
- Migliorare la comunicazione nel team
- Aumentare la produttività
- Gestire tempo e produttività personale
- Ridurre sprechi

Version Control System

Definizione

I "Version Control Systems" (VCS) o "Source Code Management Systems" (SCM) sono sistemi software che registrano le modifiche a un insieme di file nel tempo, permettendo di recuperare versioni specifiche in un secondo momento. Facilitano la condivisione e la collaborazione su progetti software.

Caratteristiche principali

- **Registrazione delle modifiche**: Mantengono uno storico completo delle modifiche

- **Tracciabilità**: Ogni modifica è associata a un autore, data e motivazione
- **Branching**: Permettono di lavorare su rami di sviluppo separati
- **Collaborazione**: Facilitano il lavoro simultaneo sugli stessi file
- **Merging**: Permettono di unire modifiche provenienti da diversi sviluppatori
- **Ripristino**: Consentono di tornare a versioni precedenti dei file

Tipologie di VCS

VCS Locali

- **Caratteristiche**:
  - Mantengono la storia dei cambiamenti
  - Utilizzano un database locale per tracciare le modifiche
  - Non gestiscono la condivisione
- **Esempi**: SCCS, IDE (Eclipse, IntelliJ)

VCS Centralizzati (CVCS)

- **Caratteristiche**:
  - Server centrale che contiene il repository
  - Gli sviluppatori hanno una sola versione del codice
  - Singolo punto di rottura (il server centrale)
  - Facili da apprendere
- **Esempi**: CVS, Subversion (SVN), Perforce, TFS

VCS Distribuiti (DVCS)

- **Caratteristiche**:
  - Ogni sviluppatore ha una copia completa del repository
  - Possibilità di lavorare offline
  - Migliore risoluzione dei conflitti
  - Supporto per workflow più flessibili
  - Maggiore resilienza (nessun singolo punto di rottura)
- **Esempi**: Git, Mercurial, Bazaar, Darcs

Terminologia base dei VCS

- **Diff**: Insieme di righe modificate su un file
- **Commit**: Insieme di diff convalidate
- **HEAD**: Ultimo commit sulla storia
- **Branch**: Puntatore a un singolo commit
- **Merge**: Integrazione di un branch nell'altro

- **Pull Request**: Richiesta di merging di un branch nel master

Workflow Patterns

Centralized Workflow

- Simile ai sistemi CVCS
- Facile da capire e usare
- La collaborazione è bloccata se il server è offline
- Le merge avvengono nel server centrale

Feature Branch Workflow

- Ogni feature è sviluppata in un branch dedicato
- L'incapsulamento consente di lavorare senza disturbare il codice principale
- Facilita la collaborazione
- I conflitti di merge mappano i conflitti concettuali

GitFlow Model

- Mantiene il branch master come codice rilasciato
- Utilizza il branch develop per lo sviluppo corrente
- Prevede branch dedicati per feature, release e hotfix
- Definisce ruoli precisi per ogni branch

GitHub Flow

- Più semplice del GitFlow
- Adatto per piccoli team e progetti Agile
- Il branch master è sempre pronto per il deploy

GitLab Flow

- Intermedio tra GitHub Flow e GitFlow
- Aggiunge branch di ambiente (staging, production)
- Adatto a team con processi di QA strutturati

Fork Workflow

- Ogni sviluppatore ha una copia (fork) del repository
- Ideale per progetti open source
- Maggiore autonomia degli sviluppatori
- Permette permessi restrittivi

CVCS vs DVCS

CVCS:

- Più semplice da apprendere
- Permette il lock dei file
- Operazioni di commit più lente
- Single point of failure
- Limitato al Centralized Workflow

DVCS:

- Più recente e standard de facto
- Architettura distribuita (maggiore resilienza)
- Supporta più workflow
- Operazioni locali rapide e offline
- Apprendimento più complesso
- Non permette il lock dei file

GitHub

Definizione

GitHub è una piattaforma di hosting per progetti software che utilizza Git come sistema di controllo versione. Offre funzionalità di controllo versione distribuito e gestione del codice sorgente, oltre a caratteristiche di collaborazione come bug tracking, feature request, task management, e wiki per ogni progetto.

Caratteristiche principali

- Servizi di hosting per progetti software
- Funzionalità di controllo versione basate su Git
- Strumenti di collaborazione (issue tracking, pull requests)
- Wikis e documentazione per i progetti
- Integrazione con CI/CD

Issue Tracker in GitHub

GitHub include un sistema di issue tracking che permette di:

- Creare issue con titolo e descrizione
- Assegnare etichette (labels) per categorizzare
- Assegnare le issue a membri specifici del team
- Associare le issue a milestone

- Tracciare lo stato dell'issue (aperto/chiuso)
- Commentare e discutere le issue

Workflow in GitHub

GitHub utilizza un workflow semplificato rispetto ad altri ITS:

- Gli stati sono solo "Open" e "Closed" (con possibilità di riaprire)
- Non è possibile definire workflow personalizzati
- È possibile utilizzare le etichette per indicare lo stato più specifico

Project Board

Le bacheche di progetto (project board) in GitHub consentono:

- Visualizzazione Kanban dei task
- Organizzazione automatica delle issue in colonne (To Do, In Progress, Done)
- Trascinamento delle issue tra le colonne
- Monitoraggio del progresso del progetto

Milestone

Le milestone permettono:

- Raggruppare issue correlate
- Impostare date di scadenza
- Tracciare il progresso verso rilasci specifici
- Monitorare lo stato complessivo delle issue associate

Collaborazione in GitHub

GitHub facilita la collaborazione attraverso:

- Fork di repository
- Branch per funzionalità specifiche
- Pull request per proporre modifiche
- Revisione del codice
- Discussioni sui commit e sulle issue
- Menzioni (@username) per notificare altri utenti

Integrazione con Git

GitHub estende Git con funzionalità aggiuntive:

- Interfaccia web per operazioni Git

- Visualizzazione grafica delle modifiche
- Merge automatico delle pull request
- Collegamenti tra commit e issue

Git

Definizione

Git è un sistema di controllo versione distribuito creato da Linus Torvalds nel 2005. È progettato per gestire progetti di qualsiasi dimensione con velocità ed efficienza, ed è diventato lo standard de facto per il controllo versione.

Caratteristiche principali

- **Branching and Merging**: Facilita lo sviluppo su branch diversi
- **Velocità**: Operazioni locali rapide, sviluppato in C
- **Distribuito**: Ogni clone contiene l'intera storia del repository
- **Integrità**: Ogni commit ha un checksum SHA-1 basato sul contenuto
- **Staging Area**: Area intermedia per preparare i commit
- **Open Source**: Rilasciato sotto licenza GNU GPL v2

Differenze con altri VCS

Git memorizza i dati come snapshot del filesystem, non come differenze. Per efficienza, se i file non cambiano, Git memorizza solo un riferimento al file identico precedente.

Aree locali in Git

1. **Working Directory**: File estratti dal repository
2. **Staging Area** (Index): File pronti per il commit
3. **Repository Locale**: Database degli oggetti Git

Stati di un file in Git

- **Untracked**: File nuovo non versionato
- **Unmodified**: File non modificato rispetto alla versione precedente
- **Modified**: File modificato nella working directory
- **Staged**: File aggiunto alla staging area
- **Committed**: File salvato nel repository locale

Comandi base di Git

Configurazione

```
git config --global user.name "Nome Cognome"
git config --global user.email nome@esempio.com
git config --list
```

Creazione/Clonazione repository

```
git init                        # Crea nuovo repository locale
git clone url [directory]       # Clona repository remoto
```

Modifiche e commit

```
git status                      # Mostra lo stato dei file
git add filename                # Aggiunge file alla staging area
git add .                       # Aggiunge tutti i file modificati
git commit -m "messaggio"       # Crea un commit con i file in staging
git commit -a -m "messaggio"    # Aggiunge tutti i file modificati e crea
commit
```

Visualizzazione modifiche

```
git diff                        # Mostra modifiche non staged
git diff --staged               # Mostra modifiche staged
git diff HEAD                   # Mostra tutte le modifiche
git log                         # Mostra la storia dei commit
```

Branch e merge

```
git branch                      # Lista branch locali
git branch nome                 # Crea nuovo branch
git checkout nome               # Passa a un branch
git merge nome                  # Unisce branch corrente con 'nome'
```

Repository remoti

```
git remote                      # Lista repository remoti
git remote -v                   # Lista dettagliata repository remoti
git fetch origin                # Recupera modifiche remote senza merge
git pull origin branch          # Recupera e unisce modifiche remote
git push origin branch          # Invia modifiche locali al repository remoto
```

Workflow tipico

1. Aggiornare il repository locale (`git pull`)

2. Creare o passare a un branch (`git checkout -b feature`)
3. Apportare modifiche ai file
4. Verificare le modifiche (`git status` , `git diff`)
5. Aggiungere i file alla staging area (`git add`)
6. Creare un commit (`git commit`)
7. Pushare le modifiche al repository remoto (`git push`)
8. Creare una pull request (su GitHub o equivalente)

Il Framework Scrum

Definizione

Scrum è un framework agile per la gestione di progetti complessi. Si basa sull'idea di sviluppo iterativo ed incrementale, dove i requisiti e le soluzioni evolvono nel tempo attraverso la collaborazione di team auto-organizzati.

Origini

Scrum è stato formalizzato da Ken Schwaber e Jeff Sutherland nei primi anni '90, presentato ufficialmente a OOPSLA nel 1995. Il nome deriva dal rugby, dove "scrum" indica la formazione ordinata in cui la squadra lavora come unità per avanzare.

Caratteristiche principali

- **Leggero**: Minima burocrazia e cerimonie
- **Facile da capire**: Regole semplici
- **Difficile da padroneggiare**: Richiede esperienza per l'implementazione efficace

Pilastri di Scrum

1. **Trasparenza**: Tutti gli aspetti del processo sono visibili
2. **Ispezione**: Regolare verifica degli artefatti e del progresso
3. **Adattamento**: Aggiustamenti in base ai risultati delle ispezioni

Sprint

Lo Sprint è un periodo di tempo fisso (generalmente 2-4 settimane) durante il quale il team sviluppa un incremento di prodotto potenzialmente rilasciabile. Ogni Sprint ha:

- Una durata costante
- Un obiettivo chiaro (Sprint Goal)
- Nessuna modifica che metta a rischio lo Sprint Goal
- Non può essere cancellato se non in casi estremi

Ruoli in Scrum

Product Owner

- Definisce le caratteristiche del prodotto
- Rappresenta gli interessi del cliente
- Decide date e contenuto dei rilasci
- Responsabile del ROI del prodotto
- Definisce le priorità del Product Backlog
- Accetta o rifiuta i risultati del lavoro

Scrum Master

- Facilita l'adozione e applicazione di Scrum
- Rimuove gli ostacoli al progresso del team
- Assicura produttività e operatività del team
- Favorisce la cooperazione tra ruoli e funzioni
- Protegge il team da interferenze esterne
- "Servant leader" per il team

Development Team

- Tipicamente 5-9 persone
- Auto-organizzato e cross-funzionale
- Responsabile della realizzazione del prodotto
- Stima gli item del Product Backlog
- Decide autonomamente come realizzare lo Sprint Backlog

Eventi Scrum

Sprint Planning

- Definisce cosa verrà realizzato nello Sprint
- Crea lo Sprint Backlog
- Identifica i task e le stime
- Time-boxed (~8 ore per Sprint di 1 mese)

Daily Scrum

- Incontro quotidiano di 15 minuti
- Allineamento del team sulle attività
- Risponde a tre domande:
    1. Cosa ho fatto ieri?

2. Cosa farò oggi?
3. Ci sono impedimenti?

Sprint Review

- Presentazione dell'incremento realizzato
- Feedback dagli stakeholder
- Informale, demo del prodotto
- Time-boxed (~4 ore per Sprint di 1 mese)

Sprint Retrospective

- Analisi di cosa ha funzionato e cosa migliorare
- Definizione di azioni di miglioramento
- Time-boxed (~3 ore per Sprint di 1 mese)

Artefatti Scrum

Product Backlog

- Lista ordinata di tutto ciò che serve al prodotto
- Unica fonte di requisiti
- Gestito dal Product Owner
- Ordinato per valore, rischio, priorità
- Dinamico, evolve con il prodotto

Sprint Backlog

- Set di item del Product Backlog selezionati per lo Sprint
- Piano per realizzare l'incremento
- Scomposto in task (stime in ore)
- Aggiornato quotidianamente
- Visibile a tutto il team

Incremento

- Somma di tutti gli item completati nello Sprint
- Deve essere "Done" (pronto all'uso)
- Potenzialmente rilasciabile

Concetti chiave

Definition of Done

- Definizione condivisa di cosa significa "completato"
- Varia da team a team
- Garantisce qualità e uniformità

Sprint Goal

- Obiettivo di business dello Sprint
- Fornisce coerenza e focus al team
- Guida le decisioni durante lo Sprint

User Stories

- Descrizione di funzionalità dal punto di vista dell'utente
- Formato: "Come [ruolo], voglio [funzionalità], per [beneficio]"
- Accompagnate da criteri di accettazione

Burndown Chart

- Grafico che mostra il lavoro rimanente nel tempo
- Strumento visivo per monitorare il progresso
- Aiuta a prevedere se lo Sprint verrà completato in tempo

Build Automation

Definizione

La Build Automation è il processo di automatizzazione della creazione di una build software e dei processi associati, tra cui: compilazione del codice sorgente, packaging del codice binario ed esecuzione di test automatizzati.

Problematiche risolte

- Errori di build manuali
- Integrazioni difficoltose e tese
- Tempo sprecato in operazioni ripetitive
- Inconsistenze tra ambienti diversi

Tipi di strumenti

Build-automation utility

Strumenti il cui scopo principale è generare artefatti di build attraverso attività come compilazione e linking del codice sorgente:

- Make (e Makefile)

- Apache Ant
- Gradle
- Rake
- Apache Maven
- NPM
- Grunt

Build-automation server

Strumenti web-based che eseguono utility di build automation in modo pianificato o su trigger:

- Jenkins
- Travis CI
- TeamCity
- CircleCI
- GitHub Actions
- GitLab CI/CD

Processo di Build

Il processo di build trasforma script di build, codice sorgente, file di configurazione, documentazione e test in un prodotto software distribuibile attraverso una serie di fasi:

1. Compilazione del codice
2. Esecuzione dei test
3. Creazione del pacchetto
4. Documentazione
5. Reporting

Caratteristiche (CRISP)

- **Completo**: Indipendente da fonti non specificate nello script di build
- **Ripetibile**: Una esecuzione ripetuta dà lo stesso risultato
- **Informativo**: Fornisce informazioni sullo stato del prodotto
- **Schedulabile**: Può essere programmato ed eseguito automaticamente
- **Portabile**: Indipendente il più possibile dall'ambiente di esecuzione

Maven

Definizione

Apache Maven è uno strumento di gestione e comprensione di progetti software. Basato sul concetto di Project Object Model (POM), Maven può gestire build, reporting e

documentazione di un progetto da un elemento centrale di informazione.

Filosofia

Maven si basa sul paradigma "Convention over Configuration", che prevede una configurazione minima o assente per il programmatore, obbligandolo a configurare solo gli aspetti che si differenziano dalle implementazioni standard.

Caratteristiche principali

- **Build Tool**: Definisce cicli di vita (lifecycle) per configurare ed eseguire il processo di build
- **Dependency Management**: Gestione automatica delle dipendenze
- **Remote Repositories**: Repository remoti per librerie e plugin
- **Universal Reuse of Build Logic**: Plugin riutilizzabili per diversi aspetti della build

Build Lifecycle

Maven si basa sul concetto di build lifecycle, un processo chiaro e definito per la costruzione e distribuzione di un artefatto. I principali lifecycle sono:

Default Lifecycle

Fasi principali:

1. **validate**: Verifica che il progetto sia corretto
2. **compile**: Compila il codice sorgente
3. **test**: Esegue i test unitari
4. **package**: Crea il pacchetto (JAR, WAR, ecc.)
5. **verify**: Esegue controlli sui risultati dei test
6. **install**: Installa il pacchetto nel repository locale
7. **deploy**: Copia il pacchetto nel repository remoto

Clean Lifecycle

Gestisce la pulizia del progetto, eliminando i file generati.

Site Lifecycle

Gestisce la creazione della documentazione del sito del progetto.

Project Object Model (POM)

Il POM è un file XML che contiene informazioni e configurazioni del progetto:

- ID del progetto (groupId, artifactId, version)

- Dipendenze
- Plugin e goal
- Profili di build
- Informazioni generali (sviluppatori, licenze, ecc.)

Archetypes

Gli archetypes sono template di progetti che permettono di generare rapidamente nuovi progetti con una struttura predefinita.

Plugin e Mojo

- I plugin sono i veri esecutori delle azioni in Maven
- Un Mojo (Maven Old Java Object) è un goal in Maven
- I plugin possono avere più goal
- È possibile sviluppare plugin personalizzati

Repository

- **Repository locale**: Cache locale delle dipendenze scaricate
- **Repository centrale**: Repository pubblico gestito dalla community
- **Repository interni**: Repository aziendali per artefatti privati o mirror

Vantaggi

- Struttura di progetto standardizzata
- Gestione dipendenze centralizzata
- Riuso di configurazioni comuni
- Integrazione con CI/CD
- Ampia adozione nella community Java

Software Testing

Definizione

Il software testing è un'indagine condotta per fornire informazioni sulla qualità del software sotto test. Comprende il processo di esecuzione di un programma con l'intento di trovare bug e verificare che il prodotto sia idoneo all'uso.

Principi fondamentali

1. **Il test rileva i difetti**, non la loro assenza
2. **Test esaustivi non esistono** (tranne per casi molto semplici)
3. **Testare il prima possibile** riduce i costi del progetto

4. **Clustering dei difetti**: la maggior parte dei difetti si concentra in pochi moduli
5. **Paradosso del pesticida**: ripetere sempre gli stessi test trova sempre meno difetti
6. **I test dipendono dal contesto** (applicazioni diverse richiedono approcci diversi)
7. **L'assenza di errori** non garantisce l'utilità del software

Categorie di test

Test funzionale vs non funzionale

- **Test funzionale**: Verifica cosa fa l'applicazione
- **Test non funzionale**: Verifica come l'applicazione risponde (performance, sicurezza, usabilità)

Test statico vs dinamico

- **Test statico**: Verifica senza esecuzione del codice (analisi codice, revisione documenti)
- **Test dinamico**: Verifica durante l'esecuzione del software

Verifica vs validazione

- **Verifica**: Il prodotto è stato realizzato secondo le specifiche tecniche
- **Validazione**: Il prodotto soddisfa i requisiti dell'utente

Livelli di test (V-Model)

Component testing (Unit testing)

- Verifica la più piccola unità testabile separatamente
- Test veloci ed indipendenti
- White box testing

Integration testing

- Verifica l'integrazione tra moduli
- Test più lenti e complessi
- White box testing

System testing

- Verifica il comportamento dell'intero sistema
- Verifica rispetto alle specifiche tecniche
- White box o black box testing

Acceptance testing

- Verifica rispetto ai requisiti utente
- Svolto con l'utente finale/cliente
- Black box testing

Casi di test

Un caso di test è un insieme di:

- Precondizioni
- Input
- Azioni (se applicabili)
- Risultati attesi
- Postcondizioni

Per ogni requisito testabile deve esistere almeno un caso di test.

Processo di test

1. **Test planning**: Definizione del piano di test
2. **Test control**: Azioni correttive se il piano non viene rispettato
3. **Test analysis**: Cosa testare
4. **Test design**: Come testare
5. **Test implementation**: Preparazione all'esecuzione
6. **Test execution**: Esecuzione dei test
7. **Checking result**: Verifica dei risultati
8. **Evaluating exit criteria**: Verifica dei criteri di uscita
9. **Test results reporting**: Reportistica
10. **Test closure**: Chiusura del processo

Test di Unità

Definizione

Il testing d'unità (unit testing) è l'attività di testing di singole unità software. Per unità si intende il minimo componente dotato di funzionamento autonomo, come una funzione o un metodo. I test di unità vengono sviluppati dal programmatore per verificare il comportamento atteso delle unità.

Proprietà desiderabili (A TRIP)

Automatic

- Esecuzione automatica senza intervento umano
- Rapidi da eseguire

- Autonomi nel rilevare i fallimenti

Thorough (esaustivi)

- Copertura adeguata del codice
- Verifica di tutti i possibili percorsi di esecuzione
- Misurabili con tool di code coverage

Repeatable

- Producono sempre lo stesso risultato
- Indipendenti dall'ordine di esecuzione
- Indipendenti dall'ambiente di esecuzione

Independent

- Isolati da elementi esterni
- Verificano un solo aspetto della funzionalità
- Non dipendono da altri test

Professional

- Scritti e mantenuti con la stessa cura del codice di produzione
- Documentano il comportamento atteso delle unità
- Spesso hanno volume simile o superiore al codice di produzione

Caratteristiche

- Test di **verifica** (non validazione)
- Test **dinamico** (richiede esecuzione)
- Test **funzionale**
- Approccio **white box** (accesso al codice sorgente)

Framework per unit testing

Un framework per test di unità fornisce:

- Un modo per configurare l'ambiente di test
- Un modo per selezionare test da eseguire
- Un modo per analizzare valori attesi
- Un modo standard per esprimere il risultato

JUnit

JUnit è il framework di test più diffuso per Java. Offre:

- **Annotations**: `@Test`, `@Before`, `@After`, `@BeforeClass`, `@AfterClass`
- **Assertions**: `assertEquals`, `assertTrue`, `assertFalse`, ecc.
- **Test Suites**: Raggruppamento di test
- **Categories**: Classificazione dei test
- **Parametrized Tests**: Test con diversi set di dati

Cosa verificare nei test di unità

Are the Results Right?

Verificare che i risultati prodotti siano corretti rispetto all'input.

Boundary Conditions (CORRECT)

- **Conformance**: Valori conformi al formato atteso
- **Ordering**: Valori che seguono o non un ordine
- **Range**: Valori entro i limiti min/max
- **Reference**: Riferimenti a dati esterni
- **Existence**: Valori esistenti (non null)
- **Cardinality**: Quantità corretta di valori
- **Time**: Ordine temporale corretto

Check Inverse Relationship

Verificare la funzione attraverso la sua inversa (es. radice quadrata → quadrato).

Cross-check Using Other Means

Usare un oracolo esistente per verificare il comportamento.

Force Error Conditions

Verificare il comportamento in condizioni di errore.

Performance Characteristics

Verificare che i test siano veloci (essenziali per CI).

Test Driven Development (TDD)

TDD è un approccio che prevede la scrittura dei test prima del codice:

1. **Red**: Scrittura di un test che fallisce
2. **Green**: Implementazione minima per passare il test
3. **Refactor**: Miglioramento del codice mantenendo i test verdi

Analisi Statica del Codice

Definizione

L'analisi statica del codice è l'analisi del software eseguita senza effettuare l'esecuzione dei programmi, in contrasto con l'analisi dinamica. Si tratta di un tipo di test:

- Statico: non richiede l'esecuzione
- White box: è presente il codice sorgente
- Non funzionale

Motivazioni

Secondo la "teoria delle finestre rotte", piccoli problemi non risolti possono portare a un deterioramento generale della qualità. L'analisi statica aiuta a identificare e correggere piccoli problemi prima che si accumulino.

Funzionalità degli strumenti di analisi statica

- Imporre il rispetto di convenzioni e stili
- Verificare la congruità della documentazione
- Controllare metriche e indicatori (complessità ciclomatica, ecc.)
- Ricercare codice duplicato
- Identificare errori comuni
- Misurare la copertura dei test
- Ricercare parti incomplete

Principali strumenti

Checkstyle

- Verifica aderenza agli standard di codifica
- Controlla layout e formattazione del codice
- Disponibile come plugin Maven

FindBugs / SpotBugs

- Analizza bytecode Java per cercare bug
- Identifica pattern di codice problematici
- Disponibile come plugin Maven

PMD

- Ricerca difetti di programmazione comuni

- Identifica codice inutilizzato, duplicato, complesso
- Supporta più linguaggi (principalmente Java e Apex)

SonarQube

- Piattaforma completa per ispezione continua della qualità
- Storicizza l'andamento della qualità nel tempo
- Classifica le problematiche in:
  - Vulnerabilità (security)
  - Bug (reliability)
  - Code Smell (maintainability)
- Permette di definire quality profile e quality gate
- Offre dashboard e reportistica avanzata

Benefici dell'analisi statica

- Identificazione precoce di problemi
- Miglioramento della qualità del codice
- Standardizzazione delle pratiche di sviluppo
- Riduzione del debito tecnico
- Educazione degli sviluppatori sulle best practice

Continuous Integration

Definizione

La Continuous Integration (CI) è una pratica di sviluppo software in cui i membri del team integrano frequentemente il proprio lavoro, di solito più volte al giorno. Ogni integrazione viene verificata da una build automatica che include test per rilevare errori il più rapidamente possibile.

Motivazioni

- Evitare l'**integration hell** (difficoltà di integrare codice sviluppato separatamente)
- Rilevare problemi tempestivamente
- Ridurre il tempo dedicato al debugging
- Aumentare la visibilità del processo di sviluppo

Processo di CI

1. Gli sviluppatori inviano frequentemente le modifiche al VCS
2. Un server CI monitora il repository per cambiamenti
3. Quando rileva un cambiamento, il server esegue:

   - Build del codice
   - Esecuzione dei test automatici
   - Analisi statica del codice
4. Il server notifica il team in caso di problemi
5. In caso di fallimento, il team corregge immediatamente

Prerequisiti

1. Gestione del codice in un VCS
2. Processo di build automatizzato
3. Test automatici (unità, integrazione, analisi statica)
4. Adozione corretta della pratica da parte del team

Best Practices

Integrare frequentemente

- Più di una volta al giorno
- Modifiche piccole e facili da gestire
- Suddividere le attività in task brevi

Creare test automatici

- Test di unità
- Test di integrazione (subsystem interni)
- Analisi statica del codice

Processo di build veloce

- Evita build lente che scoraggiano l'integrazione
- Ottimizza per individuare errori il prima possibile
- Mantieni sotto i 10 minuti il tempo di build

Gestire il workspace

- Ogni sviluppatore deve poter eseguire build e test localmente
- Mantenere nel VCS tutto il necessario per il processo di build
- Essere pronti a ripristinare la versione precedente

Correggere immediatamente i problemi

- "Nobody has a higher priority task than fixing the build" (Kent Beck)
- Ripristinare la versione precedente se non è possibile correggere rapidamente
- Non commentare i test falliti

Visibilità dello stato della build

- Dashboard accessibile a tutto il team
- Notifiche in caso di fallimento o successo
- Tracciabilità di chi ha introdotto l'errore

Strumenti di CI

Jenkins

- Server open source per l'automazione
- Pipeline configurabili con Jenkinsfile
- Ampia libreria di plugin
- Supporto per esecuzione distribuita

GitHub Actions

- Integrato in GitHub
- Workflow definiti in file YAML
- Esecuzione automatica in base a eventi GitHub
- Facile integrazione con repository GitHub

Differenze tra Jenkins e GitHub Actions

Jenkins	GitHub Actions
Funziona con diversi VCS	Integrazione nativa con GitHub
Configurato via Jenkinsfile o UI	Configurato via file YAML
Più flessibile e personalizzabile	Più semplice e diretto
Scalabile con nodi distribuiti	Limitato dall'infrastruttura GitHub
Più adatto a progetti complessi	Adatto a progetti di piccola/media scala

Continuous Delivery

Definizione

La Continuous Delivery (CD) è una disciplina di sviluppo software in cui il software viene costruito in modo tale da poter essere rilasciato in produzione in qualsiasi momento. È l'estensione naturale della Continuous Integration.

Caratteristiche principali

- Il software è sempre in uno stato rilasciabile

- Il team prioritizza il mantenimento della rilasciabilità rispetto alle nuove funzionalità
- Feedback rapido e automatizzato sulla production readiness
- Deploy a pulsante di qualsiasi versione in qualsiasi ambiente

Differenza con Continuous Deployment

- **Continuous Delivery**: Il rilascio in produzione richiede approvazione manuale
- **Continuous Deployment**: Il rilascio in produzione è completamente automatizzato

Problemi risolti

- Attese tra team (DEV, TEST, OPS)
- Documentazione insufficiente per i rilasci
- Ritardi nel feedback sulle funzionalità
- Scoperta tardiva di problemi architetturali

Deployment Pipeline

La Deployment Pipeline è la modellazione del processo di rilascio tramite una successione di fasi (stages) e verifiche (gates):

- Ogni fase verifica specifici aspetti del software
- I gate determinano il passaggio alla fase successiva
- Le fasi possono essere eseguite in parallelo o in sequenza
- L'obiettivo è fallire il prima possibile (fail-fast)

Fasi tipiche di una pipeline CD

1. **Commit Stage**: Build, unit test, analisi codice
2. **Automated Acceptance Testing**: Test funzionali automatizzati
3. **Manual Testing**: Test manuali, usability testing
4. **Performance Testing**: Test di carico e stress
5. **Staging**: Verifica in ambiente simile a produzione
6. **Production**: Rilascio in ambiente di produzione

Pratiche fondamentali

Only Build Your Binaries Once

- Costruire l'artefatto una sola volta e riutilizzarlo in tutte le fasi
- Garantisce che ciò che viene testato è ciò che va in produzione
- Richiede un Artifact Repository per memorizzare gli artefatti

Deploy the Same Way to Every Environment

- Utilizzare lo stesso script per tutti gli ambienti
- Separare il codice dalle configurazioni specifiche per ambiente
- Collaborare con il team operativo per definire gli script

Smoke-Test Your Deployments

- Verificare rapidamente il corretto funzionamento post-deploy
- Includere verifiche sui sistemi esterni (DB, servizi, ecc.)
- Fallire rapidamente in caso di problemi

Deploy into a Copy of Production

- Testare in ambienti il più possibile simili alla produzione
- Configurazione di rete, sistema operativo, stack applicativo equivalenti
- Utilizzo di Configuration Management per garantire uniformità

Each Change Should Propagate Through the Pipeline Instantly

- Ogni modifica al codice sorgente deve avviare la pipeline
- Verifiche precoci per fallire il prima possibile
- Tracciabilità delle modifiche che causano fallimenti

If Any Part of the Pipeline Fails, Stop the Line

- Prioritizzare i controlli più veloci
- Notificare immediatamente i fallimenti
- Correggere prima di procedere con altre attività

Benefici

- Riduzione del rischio di deploy
- Accelerazione del time-to-market
- Maggiori feedback dagli utenti
- Progressi tangibili e misurabili
- Costi di sviluppo ridotti
- Prodotti di qualità superiore
- Team meno stressati e più collaborativi
- Maggiore documentazione implicita nei processi automatizzati

Requisiti

- Continuous Integration funzionante
- Build automation

- Unit testing
- Artifact Repository
- Configuration Management
- Continuous Testing
- Orchestratore per la pipeline

Artifact Repository

Definizione

Un Artifact Repository (o Binary Repository Manager) 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 generati e utilizzati dall'organizzazione.

Funzionalità principali

- **Archiviazione di artefatti**: Memorizza i binari prodotti nel processo di build
- **Gestione delle dipendenze**: Facilita l'accesso a librerie e componenti esterni
- **Proxy per repository esterni**: Cache locale di artefatti da repository remoti
- **Controllo di accesso**: Gestione di permessi su artefatti e repository
- **Metadati**: Informazioni aggiuntive sugli artefatti (versione, dipendenze, licenze)
- **Verifica di vulnerabilità**: Scansione di componenti con problemi di sicurezza
- **Controllo licenze**: Verifica della compatibilità delle licenze

Tipologie di artefatti

- JAR (Java Archive)
- WAR (Web Archive)
- EAR (Enterprise Archive)
- ZIP/TAR (archivi generici)
- NPM (JavaScript/Node.js packages)
- Docker images
- NuGet (pacchetti .NET)
- RPM/DEB (pacchetti Linux)
- Ruby Gems
- Python Wheels/Eggs

Caratteristiche degli artefatti

- **Identificativo univoco** (GAV: Group, Artifact, Version)
- **Metadati** (licenze, dipendenze, riferimenti al VCS)
- **Checksum** (MD5, SHA1) per garantire integrità

- **Documentazione** (pom.xml, package.json, ecc.)

Maven Repository Management

- **Proxy Remote Repositories**: Cache locale di repository esterni
- **Hosted Internal Repositories**: Artefatti interni o di terze parti con restrizioni
- **Release Artifacts**: Artefatti immutabili e persistenti
- **Snapshot Artifacts**: Artefatti di sviluppo che possono essere aggiornati

Vantaggi dell'utilizzo

- **Velocità**: Riduzione dei tempi di build grazie a cache locali
- **Stabilità**: Indipendenza da repository esterni
- **Controllo**: Governance su componenti utilizzati
- **Collaborazione**: Condivisione standardizzata di artefatti
- **Tracciabilità**: Collegamento tra artefatti e codice sorgente

Strumenti principali

- **Sonatype Nexus**: Supporta tutti i formati principali, open source e commerciale
- **JFrog Artifactory**: Soluzione completa per DevOps, supporto universale
- **Apache Archiva**: Repository manager Java leggero
- **GitHub Packages**: Integrato con GitHub
- **GitLab Package Registry**: Integrato con GitLab
- **Docker Hub**: Specializzato in container images
- **Azure Artifacts**: Parte di Azure DevOps

Casi d'uso principali

- Gestione di librerie proprietarie
- Caching di dipendenze esterne
- Archiviazione di build per deployment
- Integrazione in pipeline CI/CD
- Distribuzione di software a clienti o partner

Configuration Management

Definizione

Il Configuration Management (CM) è un processo di ingegneria dei sistemi per stabilire e mantenere la coerenza delle prestazioni, delle funzionalità e degli attributi fisici di un prodotto con i suoi requisiti, design e informazioni operative durante tutto il suo ciclo di vita.

Obiettivi

- **Riproducibilità**: Capacità di eseguire il provisioning di qualsiasi ambiente in modo completamente automatizzato e identico
- **Tracciabilità**: Capacità di determinare rapidamente le versioni di ogni dipendenza utilizzata e confrontare versioni precedenti

Concetti chiave

Configuration Item (CI)

Un'unità di configurazione che può essere gestita individualmente:

- Computer
- Router
- Server
- Software
- Servizi

Configuration Management Database (CMDB)

Database utilizzato per tracciare tutti i Configuration Items e le relazioni tra di loro.

Infrastructure as Code (IaC)

Gestione dell'infrastruttura IT tramite file di configurazione invece di processi manuali:

- Definizione dichiarativa dello stato desiderato
- Versionabile con VCS
- Riproducibile e scalabile

Benefici

- **Coerenza**: Ambienti identici e ripetibili
- **Controllo**: Gestione precisa delle modifiche
- **Automazione**: Riduzione degli errori manuali
- **Documentazione**: Configurazioni auto-documentate
- **Scalabilità**: Facilità di replicare ambienti
- **Ripristino**: Recupero rapido in caso di problemi

Processi di CM nel software

1. **Identificazione della configurazione**: Definizione di cosa gestire
2. **Controllo della configurazione**: Gestione delle modifiche
3. **Accounting dello stato**: Tracciamento di versioni e modifiche

4. **Audit della configurazione**: Verifica della correttezza

Strumenti di CM

Ansible

- Agentless (non richiede software sui nodi gestiti)
- Basato su YAML e Python
- Push-based (il controller invia comandi)
- Semplice da apprendere e usare

Chef

- Agent-based
- Linguaggio DSL basato su Ruby
- Pull-based (i nodi richiedono configurazioni)
- Flessibile ma con curva di apprendimento ripida

Puppet

- Agent-based
- Linguaggio DSL dichiarativo
- Pull-based con opzioni push
- Forte focus sull'enterprise

Terraform

- Specializzato in provisioning dell'infrastruttura
- Linguaggio HCL (HashiCorp Configuration Language)
- Gestisce provider diversi (AWS, Azure, GCP)
- State-based per tracciare modifiche

Salt (SaltStack)

- Agent-based (minions)
- Supporta sia push che pull
- Basato su YAML e Python
- Veloce e scalabile

Approcci di CM

- **Pull-based**: I nodi richiedono periodicamente configurazioni
- **Push-based**: Il controller invia attivamente configurazioni
- **Masterless**: Configurazioni locali senza server centrale

- **Master-agent**: Server centrale coordina nodi con agenti
- **Agentless**: Nessun software sui nodi gestiti

Integrazione con CD

Il CM è un componente essenziale della Continuous Delivery:

- Garantisce ambienti coerenti per testing e produzione
- Facilita il deployment automatizzato
- Supporta strategie di rilascio come blue-green deployment

Container

Definizione

Un container è un'unità standard di software che pacchettizza il codice e tutte le sue dipendenze per far funzionare l'applicazione in modo rapido e affidabile in ambienti diversi. A differenza delle macchine virtuali, i container condividono il kernel del sistema operativo host.

Evoluzione delle infrastrutture

1. **Era delle macchine fisiche**: Un'applicazione per server, utilizzo inefficiente
2. **Era delle macchine virtuali**: Hypervisor per eseguire più OS su stesso hardware
3. **Era dei container**: Isolamento a livello di processo, condivisione del kernel

Concetti fondamentali

Virtualizzazione

- **Hypervisor**: Software che crea e gestisce macchine virtuali
- **Tipi di Hypervisor**:
  - Type 1 (bare metal): Direttamente sull'hardware (ESXi, Hyper-V)
  - Type 2 (hosted): Su sistema operativo (VirtualBox, VMware Workstation)

Isolamento dei container

I container sono isolati attraverso:

- **Namespaces**: Isolamento delle risorse di sistema
- **Control Groups (cgroups)**: Limitazione dell'uso di risorse
- **Union File Systems**: Gestione efficiente dei filesystem a strati

Docker

Definizione

Docker è una piattaforma open-source che automatizza il deployment, il scaling e la gestione delle applicazioni containerizzate. Offre un modo standard per eseguire il software, indipendentemente dall'infrastruttura.

Principi fondamentali

- **Build**: Creazione di immagini da Dockerfile
- **Ship**: Distribuzione delle immagini attraverso registry
- **Run**: Esecuzione dei container in qualsiasi ambiente

Architettura Docker

- **Docker Engine**: Runtime per container
- **Docker Client**: Interfaccia a riga di comando
- **Docker Daemon**: Servizio che gestisce oggetti Docker
- **Docker Registry**: Repository per immagini Docker

Concetti chiave

Docker Image

Un template di sola lettura con istruzioni per creare un container Docker. Contiene:

- Sistema operativo di base
- Dipendenze
- Applicazione
- Configurazione
- Metadati

Dockerfile

File di testo con istruzioni per costruire un'immagine Docker:

- `FROM` : Immagine base
- `RUN` : Eseguire comandi durante la build
- `COPY` / `ADD` : Aggiungere file all'immagine
- `WORKDIR` : Impostare directory di lavoro
- `EXPOSE` : Dichiarare porte
- `CMD` / `ENTRYPOINT` : Definire comandi di avvio

Docker Container

Istanza in esecuzione di un'immagine Docker, isolata ma che condivide kernel e risorse:

- Leggero e portabile
- Standardizzato
- Sicuro
- Effimero (stateless)

Docker Registry

Servizio che memorizza e distribuisce immagini Docker:

- Docker Hub (pubblico)
- Registry privati
- Cloud provider registries (ECR, ACR, GCR)

Vantaggi rispetto alle VM

- **Efficienza**: Avvio in secondi vs minuti
- **Densità**: Più container per host
- **Leggerezza**: MB vs GB
- **Portabilità**: Funziona ovunque sia installato Docker
- **Isolation**: Isolamento a livello processo vs hardware

Orchestrazione di container

Definizione

L'orchestrazione automatizza il deployment, la gestione, il scaling e il networking dei container.

Problematiche risolte

- Distribuzione container su nodi diversi
- Scalabilità automatica
- Service discovery
- Load balancing
- Rollback automatico
- Self-healing

Docker Swarm

- Soluzione nativa di Docker
- Facile da configurare e usare
- Cluster di nodi Docker
- Servizi distribuiti con replica
- Limitato per deployment complessi

Kubernetes

- Originato da Google, ora gestito da CNCF
- Standard de facto per orchestrazione
- Architettura modulare e flessibile
- Concetti chiave:
  - Pod: Unità base di deployment
  - Node: Macchina fisica o virtuale nel cluster
  - Cluster: Insieme di nodi
  - Control Plane: Componenti di gestione cluster
  - Deployment: Definisce applicazioni e replica
  - Service: Astrazione di rete per pod

Altri orchestratori

- Amazon ECS/EKS
- Azure AKS
- Google GKE
- Nomad (HashiCorp)
- OpenShift (Red Hat)

Ansible

Definizione

Ansible è uno strumento di Configuration Management e IT Automation che permette di definire lo stato di uno o più server in modo prevedibile, replicabile e consistente. È agentless, basato su Python e utilizza YAML per la configurazione.

Caratteristiche principali

- **Non richiede agent**: Opera dal controller tramite SSH
- **Semplice da usare**: Curva di apprendimento bassa
- **Multipiattaforma**: Funziona su diverse piattaforme
- **Leggero e performante**: Minimo overhead
- **Idempotente**: Garantisce risultati consistenti
- **Approccio graduale**: Può essere adottato incrementalmente

Concetti chiave

Control Node (Controller)

Qualsiasi macchina con Ansible installato da cui lanciare i comandi ansible e ansible-playbook.

Managed Nodes

Server gestiti con Ansible, chiamati anche 'hosts'. Sui sistemi Linux è richiesto Python.

Inventory

Lista di server, opzionalmente raggruppati, sui quali Ansible può operare.

Tasks

Singola unità di esecuzione in Ansible, che può essere eseguita con comando "ad hoc".

Handlers

Istruzioni da eseguire quando un task provoca una modifica al sistema.

Playbooks

Insiemi di Tasks e Handlers organizzati in un file YAML.

Modules

"Verbi" del linguaggio Ansible che eseguono operazioni specifiche (installazione pacchetti, copia file, ecc.).

Formati di Inventory

Ansible supporta diversi formati per gli inventory:

- INI-like format
- YAML
- Inventory dinamici

Comandi principali

- **ansible**: Esegue un singolo task (ad hoc)
- **ansible-playbook**: Esegue playbook completi
- **ansible-inventory**: Gestisce gli inventory
- **ansible-doc**: Accede alla documentazione dei moduli

Ad-hoc commands

I comandi ad-hoc permettono di eseguire rapidamente un'operazione senza dover creare un playbook:

```
ansible <host-pattern> [options]
```

Opzioni principali:

- `-u` : Utente per la connessione
- `-k` : Chiede password SSH
- `-K` : Chiede password sudo
- `-b` : Privilege elevation (sudo/become)
- `-m` : Modulo da eseguire
- `-a` : Argomenti del modulo
- `-i` : Specifica inventory

Playbooks

I playbook sono file YAML che definiscono una serie di tasks da eseguire su host specifici:

```yaml
---
- hosts: webservers
  tasks:
    - name: Installa Apache
      apt:
        name: apache2
        state: present
    - name: Avvia servizio Apache
      service:
        name: apache2
        state: started
```

Handlers

Gli handlers vengono eseguiti solo quando un task provoca una modifica e invia una notifica:

```yaml
tasks:
  - name: Configura Apache
    template:
      src: apache.conf.j2
      dest: /etc/apache2/apache.conf
    notify: Riavvia Apache

handlers:
  - name: Riavvia Apache
    service:
      name: apache2
      state: restarted
```

Roles

I roles permettono di organizzare playbook in unità riutilizzabili:

```
roles/
  common/
    tasks/
    handlers/
    files/
    templates/
    vars/
    defaults/
  webserver/
    tasks/
    defaults/
```

Utilizzo:

```
---
- hosts: webservers
  roles:
    - common
    - webserver
```

Vantaggi di Ansible

- **Semplice**: Non richiede conoscenze di programmazione
- **Agentless**: Nessun software da installare sui nodi gestiti
- **Sicuro**: Comunica tramite SSH
- **Idempotente**: Applicare la stessa configurazione più volte produce lo stesso risultato
- **Dichiarativo**: Descrivi lo stato desiderato, non come raggiungerlo
- **Modulare**: Organizzazione in roles e playbooks riutilizzabili