Concetti

Andrea Comar

November 16, 2024

Contents

1 Automazione dei Test nell'ambito del Software Development

L'automazione dei test, in ambito di sviluppo software, è la pratica di utilizzare strumenti e script per eseguire automaticamente i test su un'applicazione, anziché farli manualmente. L'obiettivo è garantire che il software funzioni come previsto, rilevando eventuali bug e problemi in modo rapido e sistematico. Ecco alcuni aspetti principali dell'automazione dei test:

1.1 Tipologie di test automatizzati

- **Test unitari**: verificano la correttezza delle singole unità di codice, come metodi o funzioni, e sono di solito eseguiti dai programmatori.
- **Test di integrazione**: verificano che diversi componenti dell'applicazione interagiscano correttamente tra loro.
- **Test end-to-end (E2E)**: simulano il comportamento reale degli utenti, testando il sistema completo dall'inizio alla fine per verificare il funzionamento di tutto il flusso.
- **Test di regressione**: assicurano che nuove modifiche al codice non abbiano introdotto bug o rotto funzionalità esistenti.

1.2 Vantaggi dell'automazione dei test

- **Efficienza**: i test automatizzati possono essere eseguiti più velocemente dei test manuali e su larga scala, riducendo i tempi di rilascio.
- **Ripetibilità**: uno script di test automatizzato può essere eseguito quante volte necessario, utile nei casi di modifiche frequenti.
- **Affidabilità**: riduce l'errore umano, dato che ogni test viene eseguito sempre nello stesso modo.
- **Feedback rapido**: permette ai team di ottenere velocemente informazioni sulla qualità del codice dopo ogni cambiamento, supportando pratiche agili come l'integrazione continua (CI).

1.3 Strumenti di automazione dei test

Esistono molti strumenti per l'automazione dei test, e la scelta dipende dal tipo di applicazione e dalle specifiche esigenze del team. Alcuni strumenti popolari includono:

- **JUnit** e **TestNG** per i test unitari in Java.
- **Selenium** per l'automazione di test di interfaccia utente web.

- **Cypress** e **Playwright** per i test di applicazioni web.
- **Appium** per l'automazione di test di app mobili.
- **Jenkins** o **GitLab CI** per l'integrazione continua e la gestione dei test su vari ambienti.

1.4 Best practices

- **Scegliere i test da automatizzare**: non tutto è adatto all'automazione, quindi è fondamentale selezionare test critici, ripetitivi e che offrono valore su larga scala.
- **Mantenere il codice dei test**: come il codice dell'applicazione, anche gli script di test vanno mantenuti, aggiornati e rivisti periodicamente.
- **Implementare il testing progressivo**: usando una piramide di test, con più test unitari alla base, test di integrazione al centro e meno test E2E alla sommità.

In sintesi, l'automazione dei test è una componente cruciale per la qualità del software moderno e permette ai team di mantenere un elevato livello di affidabilità e di efficienza nel ciclo di sviluppo.

2 Sviluppo Agile in Programmazione

Lo sviluppo agile è un approccio al software development caratterizzato da un processo iterativo e incrementale che enfatizza la collaborazione tra team di sviluppo e stakeholder, l'adattamento alle modifiche e il rilascio frequente di versioni funzionanti del prodotto. L'obiettivo principale è fornire valore rapidamente e rispondere efficacemente ai cambiamenti, migliorando continuamente il prodotto in base al feedback degli utenti.

Principi dello Sviluppo Agile

Lo sviluppo agile si basa su un manifesto pubblicato nel 2001, che include quattro valori fondamentali e dodici principi:

- **Valori fondamentali**:
  1. Individui e interazioni più che processi e strumenti.
  2. Software funzionante più che documentazione esaustiva.
  3. Collaborazione con il cliente più che negoziazione dei contratti.
  4. Rispondere ai cambiamenti più che seguire un piano.
- **Principi** (alcuni dei più rilevanti):
  - Rilasciare software funzionante frequentemente.

- Accogliere i cambiamenti, anche in fasi avanzate del progetto.
- Collaborare quotidianamente con il cliente.
- Mantenere un ritmo sostenibile di sviluppo.
- Misurare i progressi attraverso il software funzionante.

2.1 Metodologie Agile

Esistono diverse metodologie agile, ognuna con le proprie caratteristiche e pratiche, ma tutte condividono i principi fondamentali dell'agilità. Tra le più popolari ci sono:

- **Scrum**: basato su iterazioni chiamate "sprint", in cui un team lavora su un insieme di funzionalità definite e ha incontri quotidiani ("daily standup") per coordinare il lavoro. Al termine di ogni sprint, viene presentato il lavoro completato in una "sprint review" e viene pianificato il successivo.
- **Kanban**: focalizzato sul miglioramento continuo e sul controllo del flusso di lavoro tramite una bacheca visuale. Gli elementi di lavoro sono rappresentati su una bacheca Kanban, che permette di monitorare il progresso e limitare il numero di attività in corso ("work in progress").
- **Extreme Programming (XP)**: enfatizza pratiche tecniche come il testing continuo, il refactoring e la programmazione in coppia, per migliorare la qualità del codice e rispondere rapidamente ai cambiamenti.

2.2 Vantaggi dello Sviluppo Agile

- **Risposta rapida ai cambiamenti**: l'approccio iterativo permette di adattare le priorità in base ai feedback, minimizzando il rischio di sviluppare funzionalità non desiderate.
- **Collaborazione continua**: la comunicazione frequente tra team e stakeholder garantisce che tutti siano allineati e che le aspettative siano gestite in modo efficace.
- **Qualità e feedback continuo**: i rilasci frequenti permettono di ottenere un feedback rapido sul prodotto, facilitando il miglioramento costante della qualità.

2.3 Sfide dello Sviluppo Agile

Nonostante i vantaggi, l'approccio agile presenta anche alcune sfide:

- **Dipendenza dalla collaborazione**: richiede un alto livello di comunicazione e coinvolgimento da parte di tutti i membri del team e degli stakeholder.

- **Gestione delle priorità**: in ambienti dinamici, stabilire le giuste priorità e rispettarle può essere complesso.
- **Scalabilità**: per team o progetti di grandi dimensioni, mantenere l'agilità richiede strumenti e adattamenti specifici.

2.4 Conclusione

Lo sviluppo agile ha rivoluzionato il mondo della programmazione, consentendo una maggiore flessibilità e reattività rispetto agli approcci tradizionali. Implementare l'agile in modo efficace può migliorare significativamente il prodotto finale e soddisfare meglio le aspettative degli utenti, ma richiede anche impegno nel mantenere una buona comunicazione e nel migliorare continuamente il processo.

3 DevOps in ambito software

DevOps è un approccio alla gestione del ciclo di vita del software che combina sviluppo (Dev) e operazioni IT (Ops) per migliorare la collaborazione, l'integrazione e la consegna continua di software. L'obiettivo principale di DevOps è ridurre il divario tra i team di sviluppo e quelli operativi, migliorando l'efficienza, la qualità e la velocità di rilascio del software, e garantendo una gestione solida del ciclo di vita delle applicazioni. Ecco i principali aspetti di DevOps:

3.1 Principi di DevOps

DevOps si basa su alcuni principi fondamentali:

- **Collaborazione**: sviluppatori, operatori e altri stakeholder lavorano a stretto contatto, eliminando i silos tra team e promuovendo una cultura di responsabilità condivisa.
- **Automazione**: le attività ripetitive, come il testing, la distribuzione e il monitoraggio, vengono automatizzate per velocizzare i processi e ridurre gli errori umani.
- **Integrazione continua (CI)** e **consegna continua (CD)**: pratiche che permettono di integrare frequentemente il codice e rilasciarlo in produzione in modo rapido e controllato.
- **Monitoraggio continuo**: utilizzo di strumenti per monitorare costantemente l'applicazione e l'infrastruttura, raccogliendo dati per rilevare problemi e migliorare le prestazioni.

3.2 Pipeline DevOps

La pipeline DevOps rappresenta il flusso di lavoro automatizzato che copre l'intero ciclo di vita del software, dalle modifiche al codice fino alla distribuzione. Essa include le seguenti fasi:

- **Integrazione continua (CI)**: ogni modifica del codice viene integrata automaticamente in un repository centrale e sottoposta a test automatizzati.
- **Consegna continua (CD)**: una volta passato il test, il software viene preparato per essere rilasciato automaticamente in ambienti di staging o di produzione.
- **Distribuzione continua**: se l'organizzazione adotta questa pratica, il software viene rilasciato automaticamente in produzione senza intervento umano, purché superi tutti i test e le verifiche.

3.3 Strumenti DevOps

Gli strumenti DevOps sono fondamentali per implementare l'automazione e l'integrazione tra i diversi team. Alcuni dei più utilizzati includono:

- **Git e GitHub/GitLab**: per il controllo di versione e la gestione del codice.
- **Jenkins, GitLab CI/CD e CircleCI**: per l'automazione dell'integrazione e distribuzione continua.
- **Docker** e **Kubernetes**: per la gestione di container e orchestrazione, favorendo la portabilità e la scalabilità delle applicazioni.
- **Ansible, Chef e Puppet**: strumenti di gestione della configurazione e provisioning automatico.
- **Prometheus, Grafana e ELK Stack** (Elasticsearch, Logstash, Kibana): per il monitoraggio e l'analisi delle prestazioni.

3.4 Vantaggi di DevOps

- **Velocità e frequenza di rilascio**: DevOps permette rilasci rapidi e continui, accelerando l'innovazione.
- **Qualità e affidabilità**: l'automazione dei test e la pipeline CI/CD riducono il rischio di errori, migliorando la qualità del software.
- **Efficienza operativa**: elimina le attività manuali e ripetitive, aumentando la produttività del team.
- **Scalabilità e portabilità**: l'utilizzo di container e strumenti di orchestrazione come Kubernetes facilita la scalabilità e la gestione delle risorse.

3.5 Sfide di DevOps

Implementare DevOps può presentare delle sfide:

- **Cultura e resistenza al cambiamento**: richiede un cambiamento culturale verso la collaborazione e la condivisione della responsabilità tra i team.
- **Integrazione complessa**: DevOps richiede l'adozione di molti strumenti e pratiche nuove, che possono richiedere formazione e adattamento.
- **Sicurezza**: l'automazione e i rilasci frequenti possono esporre il sistema a rischi di sicurezza, quindi è necessario adottare un approccio DevSecOps, che integra la sicurezza fin dalle prime fasi.

3.6 Conclusione

DevOps è più di un insieme di strumenti: è una filosofia che unisce team e processi per aumentare l'efficienza e la qualità dei rilasci software. Quando implementato correttamente, DevOps permette alle aziende di rispondere rapidamente ai cambiamenti, migliorando la produttività e la soddisfazione degli utenti finali.

Continuous Integration in ambito software

L'integrazione continua (Continuous Integration, o CI) è una pratica di sviluppo software che prevede l'integrazione frequente delle modifiche al codice in un repository condiviso e l'automazione dei test per rilevare eventuali errori al più presto possibile. Il concetto di CI è parte integrante del movimento DevOps ed è fondamentale per migliorare la qualità del software e ridurre i tempi di rilascio.

Principi dell'Integrazione Continua

- **Integrazione frequente**: i team di sviluppo effettuano l'integrazione del proprio codice più volte al giorno in un repository centrale (ad esempio, utilizzando Git).
- **Automazione dei test**: ogni volta che viene integrato nuovo codice, vengono eseguiti automaticamente test unitari, di integrazione o altri test automatici per assicurarsi che il codice sia privo di errori.
- **Feedback rapido**: la CI fornisce feedback immediato ai programmatori sugli errori presenti nel codice, facilitando una correzione tempestiva e limitando la proliferazione di bug.

Pipeline di Integrazione Continua

La pipeline CI è l'insieme di strumenti e processi automatizzati che controllano il codice per verificarne la qualità e, se tutto è in ordine, passano il codice per ulteriori verifiche o distribuzioni. Essa include solitamente:

- **Controllo di versione**: l'integrazione continua richiede un sistema di controllo di versione (come Git) che permette ai team di gestire in modo collaborativo le modifiche al codice.
- **Build automatizzate**: ad ogni modifica o "commit" al repository, la pipeline di CI costruisce automaticamente il codice in un ambiente di build isolato.
- **Test automatizzati**: una volta costruito, il codice è sottoposto a una serie di test per verificarne il corretto funzionamento. Questo può includere test unitari, test di integrazione, test funzionali e altri.
- **Report e notifiche**: se i test falliscono, la CI invia notifiche immediate al team, segnalando quali parti del codice hanno causato il problema.

Strumenti di Integrazione Continua

Esistono vari strumenti per implementare l'integrazione continua, ciascuno con funzionalità specifiche. Alcuni dei più popolari sono:

- **Jenkins**: uno dei primi strumenti CI open source, altamente configurabile e con numerosi plugin per estenderne le funzionalità.
- **GitLab CI/CD**: offre una pipeline CI/CD integrata direttamente nella piattaforma di gestione del codice GitLab.
- **Travis CI**: una soluzione CI cloud-based che si integra facilmente con GitHub.
- **CircleCI** e **GitHub Actions**: strumenti CI molto popolari tra gli sviluppatori che utilizzano GitHub per il controllo di versione.

Vantaggi dell'Integrazione Continua

- **Rilevamento rapido dei problemi**: la CI aiuta a identificare i bug e gli errori non appena il codice viene modificato, riducendo il tempo necessario per risolverli.
- **Rilasci più veloci**: le organizzazioni che adottano la CI possono rilasciare nuove funzionalità e miglioramenti al prodotto con maggiore frequenza e affidabilità.
- **Qualità del codice migliorata**: i test automatizzati e il feedback immediato permettono di mantenere alta la qualità del codice, riducendo la probabilità di regressioni.

- **Collaborazione più efficace**: la CI facilita il lavoro di team numerosi, poiché riduce i conflitti di integrazione e promuove un flusso di lavoro più collaborativo.

Sfide dell'Integrazione Continua

- **Tempo e risorse iniziali**: impostare una pipeline CI ben funzionante richiede tempo e sforzi iniziali per configurare gli strumenti e creare una suite di test adeguata.
- **Dipendenza dai test**: la CI richiede una buona copertura dei test; se i test non sono sufficientemente completi, problemi e bug possono comunque passare inosservati.
- **Manutenzione**: man mano che il progetto cresce, anche la pipeline CI richiede una manutenzione continua, inclusi aggiornamenti, ottimizzazioni e correzioni di eventuali errori nei test automatizzati.

Conclusione

L'integrazione continua è una pratica essenziale nello sviluppo moderno del software, specialmente per team che lavorano su progetti complessi o distribuiti. La CI permette di migliorare la qualità del software, ridurre i tempi di rilascio e favorire una collaborazione efficace tra i membri del team. Quando combinata con la consegna continua (Continuous Delivery), la CI può dare origine a una pipeline DevOps completa che automatizza l'intero ciclo di vita del software, dal codice al rilascio.

4 Test-Driven Development in ambito software

Il Test-Driven Development (TDD) è una metodologia di sviluppo software che enfatizza la scrittura di test prima della creazione del codice che li soddisferà. Questa pratica, parte integrante delle metodologie agili, mira a migliorare la qualità del software e a garantire che ogni componente del codice soddisfi i requisiti specifici fin dall'inizio. Ecco i concetti chiave del TDD:

4.1 Ciclo di Sviluppo TDD

TDD si basa su un ciclo iterativo composto da tre fasi principali:

- **Red**: scrivere un test per una nuova funzionalità o una modifica del codice. Dato che il codice che soddisfa questo test non esiste ancora, il test fallirà.
- **Green**: scrivere il codice minimo necessario per far passare il test, senza preoccuparsi della pulizia o dell'ottimizzazione.
- **Refactor**: migliorare e ottimizzare il codice appena scritto, mantenendo i test verdi (ossia, che continuano a superare il test).

Questo ciclo viene ripetuto per ogni nuova funzionalità o modifica, consentendo agli sviluppatori di costruire il software pezzo per pezzo, garantendo al contempo che ogni nuovo blocco funzioni correttamente.

4.2 Vantaggi del TDD

- **Qualità del codice migliorata**: il TDD porta gli sviluppatori a concentrarsi sulla progettazione prima della scrittura del codice, riducendo gli errori e i difetti.
- **Documentazione automatica**: i test stessi fungono da documentazione, fornendo agli sviluppatori una comprensione chiara di ciò che il codice deve fare.
- **Meno debug e meno difetti**: i problemi vengono rilevati subito e risolti prima che il codice si evolva, riducendo il tempo necessario per il debug in fasi successive.
- **Facilità di manutenzione e refactoring**: i test garantiscono che le modifiche successive non compromettano il funzionamento del codice, rendendo il codice più facile da modificare e ottimizzare nel tempo.

4.3 Tipi di Test in TDD

TDD include diversi tipi di test, che coprono vari aspetti del software:

- **Test unitari**: verificano il comportamento di singole funzioni o metodi. Costituiscono la maggior parte dei test in TDD.
- **Test di integrazione**: verificano che diverse parti del software funzionino correttamente insieme.
- **Test funzionali**: verificano il comportamento del software dal punto di vista dell'utente finale, assicurandosi che i requisiti siano rispettati.

4.4 Strumenti di TDD

Molti strumenti supportano il TDD, rendendo più facile scrivere e gestire i test. Tra i più utilizzati:

- **JUnit**: una libreria di test unitari per Java, ampiamente utilizzata per scrivere test in ambito TDD.
- **pytest**: un framework di test per Python, semplice da utilizzare per creare test unitari e di integrazione.
- **RSpec**: uno strumento di testing per Ruby, particolarmente adatto a un approccio orientato al comportamento (BDD), simile al TDD.
- **Jest** e **Mocha**: per il testing di applicazioni JavaScript, sono strumenti popolari che permettono di eseguire test unitari in ambiente web.

4.5 Sfide del TDD

Implementare il TDD può presentare delle difficoltà:

- **Curva di apprendimento iniziale**: richiede una certa esperienza per scrivere test efficaci e per sapere come strutturare il codice per supportare il TDD.
- **Investimento di tempo**: la scrittura dei test aumenta il tempo di sviluppo iniziale, anche se riduce il tempo speso in debug successivamente.
- **Dipendenza dai test**: test mal scritti o non sufficientemente coprenti possono portare a una falsa sicurezza sulla qualità del software.

4.6 Conclusione

Il Test-Driven Development è una pratica potente che, se utilizzata correttamente, può migliorare significativamente la qualità del software e ridurre il debito tecnico. Sebbene richieda un investimento iniziale e un cambiamento di mentalità per molti sviluppatori, TDD si è dimostrato efficace nel creare applicazioni più robuste, facili da mantenere e ben documentate.

5 JUnit

JUnit è un framework di testing open-source ampiamente utilizzato in ambito Java per la scrittura e l'esecuzione di test unitari. È uno strumento fondamentale per garantire la qualità del software, supportando lo sviluppo orientato ai test (Test-Driven Development, TDD) e promuovendo la pratica del Continuous Integration (CI).

5.1 Cosa sono i test unitari?

I test unitari sono piccoli test che verificano il corretto funzionamento di una specifica parte del codice, come una singola funzione o un metodo di una classe. L'obiettivo è isolare quella porzione di codice e confermare che si comporti come previsto in diverse condizioni.

5.2 Caratteristiche principali di JUnit

- **Facilità d'uso**: JUnit è semplice da configurare e utilizzare, integrandosi facilmente in ambienti di sviluppo come Eclipse, IntelliJ e NetBeans.
- **Annotazioni**: Utilizza annotazioni per definire metodi di test, cicli di vita del test, e condizioni di esecuzione.
- **Assert**: Fornisce una serie di metodi `assert` per verificare le condizioni dei test, come `assertEquals`, `assertTrue`, `assertFalse`, e così via.

- **Report di Test**: Genera report dettagliati sull'esito dei test, indicando quali test sono passati, falliti o sono stati ignorati.
- **Integrazione con CI/CD**: Si integra perfettamente con strumenti di integrazione continua come Jenkins, GitHub Actions, e GitLab CI.

5.3 Le principali annotazioni di JUnit

JUnit utilizza annotazioni per definire i metodi di test e il ciclo di vita del test. Ecco alcune delle annotazioni più importanti:

1. `@Test`: Indica che il metodo è un caso di test.

```
@Test
public void testSomma() {
    assertEquals(5, 2 + 3);
}
```

2. `@BeforeEach`: Eseguito prima di ogni metodo di test (prepara il contesto).

```
@BeforeEach
public void setUp() {
    // Codice di inizializzazione
}
```

3. `@AfterEach`: Eseguito dopo ogni metodo di test (pulizia risorse).

```
@AfterEach
public void tearDown() {
    // Codice di pulizia
}
```

4. `@BeforeAll`: Eseguito una sola volta prima di tutti i metodi di test (deve essere statico).

```
@BeforeAll
public static void initAll() {
    // Inizializzazione globale
}
```

5. `@AfterAll`: Eseguito una sola volta dopo tutti i metodi di test (deve essere statico).

```
@AfterAll
public static void cleanUpAll() {
    // Pulizia finale
}
```

6. `@Disabled`: Disattiva un test, utile per test non ancora pronti.

```
@Disabled
@Test
public void testNonPronto() {
    // Questo test non verra eseguito
}
```

5.4 Esempio pratico di test con JUnit 5

Supponiamo di avere una classe `Calcolatrice` con un metodo `somma()` che vogliamo testare:

Classe da testare (`Calcolatrice.java`):

```
public class Calcolatrice {
    public int somma(int a, int b) {
        return a + b;
    }
}
```

Classe di test (`CalcolatriceTest.java`):

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class CalcolatriceTest {
    private Calcolatrice calcolatrice;

    @BeforeEach
    public void setUp() {
        calcolatrice = new Calcolatrice();
    }

    @Test
    public void testSomma() {
        assertEquals(5, calcolatrice.somma(2, 3), "La somma
            dovrebbe essere 5");
    }
}
```

5.5 Esecuzione dei test

È possibile eseguire i test direttamente dal proprio ambiente di sviluppo (IDE) o tramite linea di comando utilizzando strumenti come Maven o Gradle:

- **Con Maven**:

```
mvn test
```

- **Con Gradle**:

```
gradle test
```

5.6 Benefici dell'uso di JUnit

- **Rilevamento precoce dei bug**: I test unitari rilevano bug prima che il software venga distribuito.
- **Refactoring sicuro**: I test esistenti agiscono come una rete di sicurezza durante le modifiche al codice.
- **Documentazione viva**: I test ben scritti fungono da documentazione eseguibile del comportamento del codice.
- **Integrazione continua**: Facilita l'integrazione continua automatizzata per migliorare la qualità del software.

5.7 Differenze tra JUnit 4 e JUnit 5

JUnit 5 è la versione più recente e introduce diverse novità rispetto a JUnit 4:

- Modularità: JUnit 5 è suddiviso in tre moduli (`JUnit Platform`, `JUnit Jupiter`, e `JUnit Vintage`).
- Lambda e metodi `assertAll`, `assertTimeout` più moderni.
- Supporto migliorato per le estensioni (`@ExtendWith`).

Esempio di `assertAll` in JUnit 5:

```
@Test
void testMultiplo() {
    assertAll("Multiplo di assert",
        () -> assertEquals(2, 1 + 1),
        () -> assertTrue(5 > 3),
        () -> assertNotNull("Ciao")
    );
}
```

5.8 Conclusione

JUnit è un pilastro fondamentale nello sviluppo di software Java, favorendo pratiche di sviluppo orientate ai test e contribuendo a una maggiore affidabilità del codice. La sua semplicità, combinata con una potente serie di funzionalità, lo rende uno strumento irrinunciabile per sviluppatori e team di sviluppo.

6 Refactoring

Il **refactoring** è una pratica fondamentale nello sviluppo software che consiste nel migliorare il design interno del codice esistente senza modificarne il comportamento esterno. L'obiettivo principale è rendere il codice più leggibile, manutenibile e facile da modificare, riducendo al contempo la complessità e migliorando la struttura.

6.1 Perché fare refactoring?

Il refactoring è importante per vari motivi:

- **Migliorare la leggibilità**: Un codice più leggibile è più facile da comprendere per i membri del team e per i futuri sviluppatori che potrebbero lavorare sul progetto.
- **Facilitare la manutenzione**: Con un codice ben strutturato, è più facile individuare e correggere bug, nonché aggiungere nuove funzionalità.
- **Ridurre il debito tecnico**: Il debito tecnico è il risultato di decisioni di progettazione affrettate o compromessi nel codice. Il refactoring aiuta a ridurlo, migliorando la qualità del codice nel lungo periodo.
- **Promuovere il riutilizzo**: Un codice modulare e ben organizzato facilita il riutilizzo di componenti in altre parti del progetto o in progetti futuri.
- **Prevenire la degenerazione del codice**: Con il tempo, il codice tende a diventare disordinato (fenomeno noto come "software rot"), specialmente in progetti a lungo termine. Il refactoring aiuta a mantenere il codice in buone condizioni.

6.2 Quando fare refactoring?

Il refactoring può essere eseguito in vari momenti dello sviluppo:

- **Prima di aggiungere nuove funzionalità**: Pulire il codice esistente rende più facile l'integrazione di nuove funzionalità.
- **Dopo aver corretto un bug**: Il refactoring può aiutare a prevenire futuri bug e migliorare la stabilità del sistema.
- **Durante le revisioni del codice**: Se il codice esaminato è difficile da comprendere o modificare, è un buon candidato per il refactoring.
- **Continuamente**: Molti team praticano il refactoring continuo, apportando miglioramenti incrementali durante il normale ciclo di sviluppo.

6.3 Tecniche comuni di refactoring

Esistono diverse tecniche di refactoring che possono essere applicate a seconda delle necessità:

1. **Rinomina di variabili, metodi e classi**: Rendi i nomi più descrittivi per migliorare la comprensibilità del codice.

```
    // Prima
    int a = 5;
    int b = 10;
    int c = a + b;

    // Dopo
    int numero1 = 5;
    int numero2 = 10;
    int somma = numero1 + numero2;
```

2. **Estrai metodo (Extract Method)**: Suddividi metodi lunghi in metodi più piccoli e riutilizzabili.

```
    // Prima
    void stampaFattura() {
        // Logica per calcolare il totale
        double totale = calcolaTotale();
        System.out.println("Totale: " + totale);
    }

    // Dopo
    void stampaFattura() {
        double totale = calcolaTotale();
        stampaTotale(totale);
    }

    void stampaTotale(double totale) {
        System.out.println("Totale: " + totale);
    }
```

3. **Sostituisci codice duplicato (Eliminate Duplicated Code)**: Identifica blocchi di codice ripetuti e spostali in un metodo comune.

```
    // Prima
    double calcolaAreaQuadrato(double lato) {
        return lato * lato;
    }

    double calcolaAreaCerchio(double raggio) {
        return Math.PI * raggio * raggio;
    }

```

```
    // Dopo (usando polimorfismo o una strategia comune)
    interface Forma {
        double calcolaArea();
    }

    class Quadrato implements Forma {
        double lato;
        public double calcolaArea() { return lato * lato
            ; }
    }

    class Cerchio implements Forma {
        double raggio;
        public double calcolaArea() { return Math.PI *
            raggio * raggio; }
    }
```

4. **Semplifica le condizioni**: Riduci la complessità di istruzioni condizionali.

```
    // Prima
    if (status == 1 || status == 2 || status == 3) {
        // Logica
    }

    // Dopo
    if (status >= 1 && status <= 3) {
        // Logica
    }
```

5. **Introdurre Oggetti Parametro**: Quando un metodo ha troppi parametri, considerare la creazione di un oggetto per contenerli.

```
    // Prima
    void creaOrdine(String nome, String indirizzo,
        String telefono) { ... }

    // Dopo
    void creaOrdine(Cliente cliente) { ... }

    class Cliente {
        String nome;
        String indirizzo;
        String telefono;
    }
```

6.4 Strumenti di supporto al refactoring

Esistono vari strumenti che supportano il refactoring del codice in modo sicuro:

- **IntelliJ IDEA**: Offre potenti funzionalità di refactoring automatizzato come estrazione di metodi, rinomina, e altro.
- **Eclipse**: Include strumenti di refactoring integrati, come la conversione di blocchi di codice in metodi e la modifica del nome di variabili e classi.
- **Visual Studio Code**: Supporta il refactoring per diversi linguaggi di programmazione tramite estensioni.
- **ReSharper**: Un'estensione per Visual Studio che fornisce strumenti avanzati di refactoring.

6.5 Buone pratiche per il refactoring

- **Scrivi test prima del refactoring**: I test automatizzati ti aiutano a garantire che il comportamento del codice non cambi durante il refactoring.
- **Apporta cambiamenti incrementali**: Evita grandi refactoring in un'unica volta. Migliora il codice passo dopo passo per ridurre il rischio di introdurre bug.
- **Usa il version control**: Il controllo di versione (come Git) ti permette di annullare facilmente modifiche se qualcosa va storto.
- **Mantieni la semplicità**: Segui il principio KISS (Keep It Simple, Stupid) e cerca di mantenere il codice il più semplice possibile.

6.6 Conclusione

Il refactoring è una pratica cruciale per mantenere la qualità del software nel tempo. Aiuta a migliorare la manutenibilità, riduce i rischi associati alle modifiche e rende il codice più comprensibile per gli sviluppatori attuali e futuri. Sebbene richieda tempo e risorse, i benefici a lungo termine superano di gran lunga i costi iniziali.

7 Principi SOLID di programmazione

I principi **SOLID** sono un insieme di linee guida per progettare software più comprensibile, manutenibile e flessibile. Sono stati definiti da Robert C. Martin (conosciuto anche come *Uncle Bob*) e sono considerati fondamentali nello sviluppo software orientato agli oggetti.

7.1 Single Responsibility Principle (SRP)

Principio di singola responsabilità: Ogni classe dovrebbe avere una, e una sola, responsabilità. In altre parole, una classe dovrebbe avere un solo motivo per cambiare.

```
// Prima: violazione del SRP
class ReportGenerator {
    void generateReport() { /* Generazione report */ }
    void printReport() { /* Logica di stampa */ }
}

// Dopo: applicazione del SRP
class ReportGenerator {
    void generateReport() { /* Generazione report */ }
}

class ReportPrinter {
    void printReport() { /* Logica di stampa */ }
}
```

7.2 Open/Closed Principle (OCP)

Principio aperto/chiuso: Il software dovrebbe essere aperto all'estensione, ma chiuso alla modifica. Ciò significa che possiamo aggiungere nuove funzionalità senza modificare il codice esistente.

```
// Prima: violazione dell'OCP
class Rectangle {
    double width, height;
}

class AreaCalculator {
    double calculateArea(Rectangle rect) {
        return rect.width * rect.height;
    }
}

// Dopo: applicazione dell'OCP usando polimorfismo
interface Shape {
    double calculateArea();
}

class Rectangle implements Shape {
    double width, height;
    public double calculateArea() {
        return width * height;
    }
}

class Circle implements Shape {
    double radius;
    public double calculateArea() {
        return Math.PI * radius * radius;
```

```
    }
}
```

7.3 Liskov Substitution Principle (LSP)

Principio di sostituzione di Liskov: Le classi derivate devono poter essere sostituite alle loro classi base senza alterare il corretto funzionamento del programma.

```
// Prima: violazione del LSP
class Bird {
    void fly() { /* Volare */ }
}

class Ostrich extends Bird {
    void fly() { throw new UnsupportedOperationException();
        }
}

// Dopo: applicazione del LSP
interface Flyable {
    void fly();
}

class Sparrow implements Flyable {
    public void fly() { /* Volare */ }
}

class Ostrich { /* Nessuna implementazione di fly() */ }
```

7.4 Interface Segregation Principle (ISP)

Principio di segregazione delle interfacce: È meglio avere molte interfacce specifiche piuttosto che una singola interfaccia generica. Le classi non dovrebbero essere obbligate a implementare metodi che non usano.

```
// Prima: violazione dell'ISP
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    public void work() { /* Lavorare */ }
    public void eat() { /* Non applicabile per i robot */ }
}

// Dopo: applicazione dell'ISP
```

```java
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() { /* Lavorare */ }
    public void eat() { /* Mangiare */ }
}

class Robot implements Workable {
    public void work() { /* Lavorare */ }
}
```

7.5 Dependency Inversion Principle (DIP)

Principio di inversione delle dipendenze: Le classi di alto livello non dovrebbero dipendere da classi di basso livello, ma entrambe dovrebbero dipendere da astrazioni. Inoltre, le astrazioni non dovrebbero dipendere dai dettagli, ma i dettagli dovrebbero dipendere dalle astrazioni.

```java
// Prima: violazione del DIP
class LightBulb {
    void turnOn() { /* Accendere la lampadina */ }
}

class Switch {
    LightBulb bulb;
    void operate() { bulb.turnOn(); }
}

// Dopo: applicazione del DIP
interface Switchable {
    void turnOn();
}

class LightBulb implements Switchable {
    public void turnOn() { /* Accendere la lampadina */ }
}

class Switch {
    Switchable device;
    void operate() { device.turnOn(); }
}
```

7.6 Conclusione

I principi SOLID sono fondamentali per la progettazione di un software robusto e flessibile. L'applicazione di questi principi riduce la complessità del codice, aumenta la sua riusabilità e migliora la capacità di adattarsi ai cambiamenti. Mantenere il codice orientato a questi principi porta a sistemi più manutenibili e scalabili nel lungo termine.

8 Clean Code in ambito software

Il concetto di **Clean Code** (codice pulito) è stato reso popolare da *Robert C. Martin* nel suo libro *Clean Code: A Handbook of Agile Software Craftsmanship*. Si riferisce a un insieme di pratiche e principi che mirano a scrivere codice chiaro, semplice, leggibile e manutenibile. Il clean code è facile da capire, sia per chi lo scrive che per chi dovrà lavorarci in futuro, e aiuta a ridurre i costi di manutenzione e sviluppo nel lungo termine.

8.1 Caratteristiche del Clean Code

Il clean code presenta una serie di caratteristiche fondamentali:

- **Leggibile**: Il codice dovrebbe essere facilmente comprensibile. Ogni parte del codice dovrebbe essere chiara e intuibile senza richiedere troppo contesto.
- **Semplice**: Un buon codice è quello che risolve il problema nel modo più semplice possibile. Ciò significa evitare complessità inutili e preferire soluzioni chiare e dirette.
- **Manutenibile**: Il codice dovrebbe essere facile da modificare e adattare. Ciò implica un buon design, la modularità, e un basso accoppiamento tra componenti.
- **Testabile**: Un codice pulito dovrebbe essere facile da testare. L'uso di funzioni pure, classi ben definite e separazione delle responsabilità contribuisce a rendere il codice testabile.
- **Elegante**: Il clean code dovrebbe essere bello da vedere. Questo si traduce in un uso efficace delle convenzioni del linguaggio e in una buona organizzazione del codice.

8.2 Principi fondamentali del Clean Code

8.2.1 Nomina significativa

Usare nomi chiari e descrittivi per variabili, funzioni e classi è una pratica fondamentale.

```
// Esempio di codice non pulito
int d; // cosa rappresenta 'd'?
if (d == 1) { ... }

// Esempio di clean code
int giorniRimanenti;
if (giorniRimanenti == 1) { ... }
```

8.2.2 Funzioni piccole e focalizzate

Le funzioni dovrebbero essere brevi e fare una sola cosa. Funzioni lunghe e complesse sono difficili da leggere e testare.

```
// Esempio di codice non pulito
void processaOrdine() {
    // Calcolo sconto
    // Generazione fattura
    // Invio email di conferma
}

// Esempio di clean code
void applicaSconto() { ... }
void generaFattura() { ... }
void inviaEmailConferma() { ... }
```

8.2.3 Evita i commenti inutili

Il codice dovrebbe essere auto-esplicativo. I commenti dovrebbero essere usati solo per chiarire parti veramente complesse, non per spiegare cosa fa un blocco di codice semplice.

```
// Esempio di codice non pulito
int x = 10; // Inizializza x con 10

// Esempio di clean code
int tentativiMassimi = 10;
```

8.2.4 Principio DRY (Don't Repeat Yourself)

Il principio **DRY** incoraggia a evitare la duplicazione del codice. Blocchi di codice ripetuti dovrebbero essere estratti in funzioni o classi riutilizzabili.

```
// Esempio di codice non pulito
double calcolaAreaRettangolo(double larghezza, double
    altezza) {
    return larghezza * altezza;
}
```

```
5
6  double calcolaAreaQuadrato(double lato) {
7      return lato * lato;
8  }
9
10 // Esempio di clean code
11 interface Forma {
12     double calcolaArea();
13 }
14
15 class Rettangolo implements Forma {
16     double larghezza, altezza;
17     public double calcolaArea() { return larghezza * altezza
           ; }
18 }
19
20 class Quadrato implements Forma {
21     double lato;
22     public double calcolaArea() { return lato * lato; }
23 }
```

8.2.5 Principio KISS (Keep It Simple, Stupid)

Mantieni il codice il più semplice possibile. La semplicità porta a una maggiore leggibilità e a un minor numero di bug.

```
1  // Esempio di codice non pulito
2  if (utente != null && utente.getRuolo() != null && utente.
       getRuolo().equals("admin")) {
3      // Logica per amministratore
4  }
5
6  // Esempio di clean code
7  if (utenteEAmministratore(utente)) {
8      // Logica per amministratore
9  }
```

8.2.6 Evita i Magic Numbers

I **Magic Numbers** sono numeri senza un significato evidente. È meglio usare costanti con nomi significativi.

```
1  // Esempio di codice non pulito
2  double area = 3.14 * raggio * raggio;
3
4  // Esempio di clean code
5  final double PI_GRECO = 3.14;
6  double area = PI_GRECO * raggio * raggio;
```

8.3 Refactoring per ottenere Clean Code

Il **refactoring** è il processo di ristrutturazione del codice esistente senza alterarne il comportamento esterno. Serve a migliorare la leggibilità, ridurre la complessità e mantenere la qualità del codice.

```
// Prima del refactoring
class Cliente {
    void processa() {
        // Calcola sconto
        // Genera fattura
        // Invia email
    }
}

// Dopo il refactoring
class Cliente {
    void processa() {
        calcolaSconto();
        generaFattura();
        inviaEmail();
    }

    private void calcolaSconto() { ... }
    private void generaFattura() { ... }
    private void inviaEmail() { ... }
}
```

8.4 Conclusione

Adottare il clean code è essenziale per mantenere un codice di alta qualità. Seguendo le linee guida per un codice pulito, gli sviluppatori possono ridurre i costi di manutenzione, migliorare la leggibilità e facilitare la collaborazione tra i membri del team. Scrivere clean code non è solo una questione di stile, ma una buona pratica che porta a sistemi software più robusti e sostenibili nel lungo termine.

9 git

10 Continuous Deployment in ambito software

Il **Continuous Deployment** (CD) è una pratica avanzata di sviluppo software che fa parte delle metodologie DevOps. Consiste nell'automazione del processo di rilascio del software, in modo che le modifiche al codice vengano distribuite automaticamente negli ambienti di produzione senza necessità di intervento manuale. Questa pratica consente alle aziende di rilasciare rapidamente

nuove funzionalità, migliorare la qualità del software e rispondere in modo agile alle esigenze del mercato.

10.1 Caratteristiche del Continuous Deployment

Le caratteristiche principali del Continuous Deployment includono:

- **Automazione del rilascio**: Il codice scritto dagli sviluppatori passa automaticamente attraverso un processo di integrazione e distribuzione senza interruzioni manuali, fino ad arrivare in produzione.
- **Feedback immediato**: Gli sviluppatori ricevono feedback in tempo reale su come le loro modifiche influenzano il sistema, grazie all'uso di test automatizzati e monitoraggio continuo.
- **Rilascio frequente**: Consente rilasci frequenti e regolari, aumentando la velocità con cui le nuove funzionalità, miglioramenti e correzioni di bug raggiungono gli utenti finali.
- **Riduzione del rischio**: Automatizzando i test e i processi di deployment, si riducono gli errori umani e si minimizza il rischio di bug introdotti durante il rilascio.
- **Affidabilità**: Attraverso l'automazione, è possibile avere un processo di distribuzione affidabile e ripetibile, garantendo che ogni release segua lo stesso flusso.

10.2 Differenze tra Continuous Integration, Continuous Delivery e Continuous Deployment

È importante distinguere tra i concetti di **Continuous Integration (CI)**, **Continuous Delivery (CD)** e **Continuous Deployment (CD)**:

- **Continuous Integration (CI)**: Si riferisce all'automazione del processo di integrazione del codice nel repository condiviso, eseguendo test automatici per verificare che le modifiche non introducano errori.
- **Continuous Delivery (CD)**: Estende la CI aggiungendo l'automazione dei test e della distribuzione negli ambienti di staging o pre-produzione. Tuttavia, il rilascio in produzione richiede ancora un'approvazione manuale.
- **Continuous Deployment (CD)**: È il livello successivo al Continuous Delivery, dove ogni modifica che supera i test viene distribuita automaticamente in produzione senza interventi manuali.

10.3 Architettura del Continuous Deployment

Un tipico flusso di lavoro di Continuous Deployment comprende i seguenti passaggi:

1. **Commit del codice**: Gli sviluppatori scrivono e inviano il codice al repository di controllo versione (ad es. Git).
2. **Build automatizzata**: Un sistema di integrazione continua (ad es. Jenkins, GitLab CI, Travis CI) rileva il nuovo commit, avvia una build automatizzata e verifica che il codice possa essere compilato correttamente.
3. **Test automatici**: Vengono eseguiti una serie di test automatici, tra cui test unitari, test di integrazione, test funzionali e test di sicurezza.
4. **Deployment automatico**: Se tutti i test passano, il sistema di Continuous Deployment esegue automaticamente il deployment della nuova versione del software negli ambienti di produzione.
5. **Monitoraggio e feedback**: Dopo il rilascio, vengono attivati strumenti di monitoraggio per rilevare eventuali problemi in tempo reale. Se viene rilevato un problema critico, è possibile eseguire un rollback automatico.

10.4 Esempio di Pipeline di Continuous Deployment

Di seguito un esempio di pipeline di Continuous Deployment definita in YAML per un sistema CI/CD come GitLab:

```yaml
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - echo "Compilazione del progetto..."
    - mvn clean package

test:
  stage: test
  script:
    - echo "Esecuzione dei test..."
    - mvn test
  allow_failure: false

deploy:
  stage: deploy
  script:
    - echo "Deploy in ambiente di produzione..."
```

```
    - kubectl apply -f deployment.yaml
  environment: production
  only:
    - main
```

Listing 1: Esempio di pipeline CI/CD in YAML

10.5 Vantaggi del Continuous Deployment

- **Riduzione del Time-to-Market**: Permette di rilasciare nuove funzionalità e correzioni di bug più rapidamente.
- **Miglioramento della qualità del software**: I test automatici garantiscono che solo il codice di alta qualità venga rilasciato in produzione.
- **Feedback continuo**: Permette di ottenere feedback immediato dagli utenti finali, facilitando miglioramenti rapidi e iterativi.
- **Scalabilità**: È particolarmente utile per ambienti cloud, dove le risorse possono essere scalate automaticamente in base al carico.

10.6 Sfide del Continuous Deployment

- **Elevata copertura dei test**: È necessario un insieme completo di test automatici per garantire che il codice sia privo di bug.
- **Monitoraggio e rollback**: È essenziale implementare un monitoraggio robusto e una strategia di rollback automatica per gestire eventuali problemi post-rilascio.
- **Cultura del DevOps**: Il Continuous Deployment richiede una stretta collaborazione tra sviluppatori, tester e team operativi, oltre a una cultura di miglioramento continuo.

10.7 Conclusione

Il Continuous Deployment rappresenta un approccio moderno ed efficiente per il rilascio del software, che consente alle aziende di innovare rapidamente e migliorare la soddisfazione degli utenti finali. Tuttavia, per implementare correttamente questa pratica, è necessario investire in strumenti di automazione, test approfonditi e una cultura DevOps consolidata.

11 unit testing

12 maven

13 make

14 build/packaging/bundle