**TEST**

Il test è un metodo empirico per verificare la correttezza di un software, un processo mirato a mostrare il comportamento di un software su un input specifico.

Ci sono due categorie principali di testing:

* **Black Box (scatola nera)**
* **White Box (scatola bianca)**

Il **Black Box Testing** verifica se il software produce il risultato atteso, ignorando come il software funziona internamente.

Il **White Box Testing,** invece testa singole porzioni di codice sorgente.

Il Unit Testing è una metodologia di White Box Testing che testa le piccole porzioni di codice che possono essere testate.

*Perché fare Unit Testing ?*

* Per verificare la correttezza del codice sorgente, seguendo un approccio Divide et impera.

Si suddivide il sistema in unità, ognuna delle quali viene debuggata separatamente per ridurre la probabilità di presentare bug.

Questo tipo di testing verifica che l’applicazione funzioni come specificato anche dopo le modifiche apportate ad essa, garantendo che la funzionalità originale continui a lavorare correttamente e che le modifiche non abbiano introdotto nuovi bug.

**JUNIT**

JUnit è un framework di testing per Java, progettato per eseguire test di unità in modo efficiente.

Le sue caratteristiche principali includono:

* Facilità di creazione di test
* Una vasta gamma di facilitazione per le asserzioni (confrontro tra risultato previsto e risultato atteso)
* Un test runner per eseguire i test e strutture per l’aggregazione di test.

I concetti fondamentali di JUnit includono:

* **Test Case:** Una singola unità di verifica, che è un metodo che verifica una specifica funzionalità di un’unita di codice.
* **Test Suite:** Una raccolta di test di unità.

Convenzioni di denominazione:

* I nomi dei metodi dei casi di test dovrebbero indicare il comportamento atteso. Ad esempio nomi come “TestGetPassword”, “TestCreazioneUtente” sono preferibili rispetto a nomi generici come “test1” o “mytest”.
* Inoltre, le classi di test generalmnete terminano con “Test”. Ad esempio “MathTest”.

I casi di test sono basati su annotazioni, non è necessario estendere alccuna classe speciale. I metodi dei casi di test sono di tipo **void** e non accettano parametri. Alcuni esempi di annotazioni includono:

* **@Test(timeout = 10) 🡪** Il test riesce se termina entro 10 secondi.
* **@Test(expected = IllegalArgumentExceprion.class) 🡪** I test riesce se viene lanciata un’eccezione di tipo IllegalArgumentExcpetion.
* **@Ignore(“reason”) 🡪** Ignora un test con una data ragione.
* **@BeforeEach** (Before in JUnit 4) **🡪** Indica un metodo che verrà chiamato prima di ogni caso di test. E’ utilizzato per preparare l’ambiente prima dell’esecuzione di ciascun test.
* **@AfterEach** (After in JUnit 4) **🡪** Indica un metodo che verrà chiamato dopo ogni caso di test. Viene utilizzato per rilasciare risorse o effettuare azioni di pulizia dopo l’edecuzione di ciascun test.
* **@BeforeAll** (BeforeClass in JUnit 4) 🡪 Indica un metodo che verrà chiamato all’inizio del test, prima di qualsiasi caso di test. Spesso utilizzato per eseguire operazioni di setup che devono avvenire una sola volta per l’intera suite di test.
* **@AfterAll** (AfterClass in JUnit 4) 🡪 Indica un metodo che verrà chiamato alla fine del test, dopo l’esecuzione di tutti i casi di test. Usato per eseguire operazioni di cleanup che devono avvenire una sola volta alla conclusione dell’intera suite di test.
  + *BeforeAll e AfterAll devono essere statici (public void static) e devono essere presenti al massimo una volta nell’intera suite di test.*

**Parametrized Tests**

* Utilizzando **@RunWith(Parameterized.class)** e un parametro contrassegnato con **@Parameters,** possiamo eseguire un test su più valori del parametro.

**Immagine che contiene testo, schermata, software

Descrizione generata automaticamente**

**Test Suite:**

* I Test Suite raggruppano i test in gerarchie.

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

**Asserts:**

* **assertEquals(expected, actual)** 
  + Funziona con oggetti, interi, long, byte, stringhe ecc..
  + Object: richiama object.equals(object) per verificare l’uguaglianza
* **assertEquals(expected, actual, epsilon)**
  + Per float e double
* **assertTrue / assertFalse(bool)**
* **assertNull / assertNotNull(object)**
* **assertSame / assertNotSame(object, object)**
* **assertArrayEquals(object[], object[])**

**Mock:**

* I Mock sono oggetti simulati che vengono utilizzati durante i test.

Nei contesti di testing, i mock sono spesso creati per sostituire parti del sistema che non sono oggetto del test stesso.

L’obiettivo principale di un mock è simulare il comportamento di un oggetto reale, consentendo ai test di concentrarsi su una parte specifica del codice senza coinvolgere completamente tutte le dipendenze reali

**MAVEN**

Maven è uno strumento di gestione di progetti per Java basato sul concetto di un Modello di Oggetto di Progetto (POM). Gestisce il processo di costruzione di un progetto, dalla sorgente al programma, supportando nativamente il riutilizzo del codice e l’integrazione tra progetti diversi.

Il ciclo di vita di compilazione di Maven suddivide il processo di costruzione in diverse fasi, tra le più comuni ci sono:

* **validate 🡪** Verifica che il progetto sia corretto e che tutte le informazioni siano disponibili
* **compile 🡪** Compila le sorgenti del progetto
* **test 🡪** Esegue un test sul codice compilato utilizzando un framework di testing unitario (ad esempio, JUnit)
* **package 🡪** Impacchetta il codice sorgente compilato in un pacchetto distribuibile (ad esempio, come file JAR o WAR)
* **verify 🡪** Esegue controlli per verificare che il pacchetto sia valido e rispetti i criteri di qualità
* **install 🡪** Installa il pacchetto nel repository locale per poterlo utilizzare in altri progetti locali.
* **Deploy 🡪** Copia il pacchetto in un repository remoto, consentendo di condividere il progetto con altrii sviluppatori e altri progetti esterni.
* **Site 🡪** Gestisce la creazione del sito web del progetto.
* **Clean 🡪** Gestisce la pulizia del progetto.

Il ciclo di vita della compilazione è una catena di fasi, e quando una fase viene eseguita, tutte le faso che la precedono nel ciclo di vita saranno eseguite. Ad esempio, l’esecuzione della fase “package” implica l’esecuzione di “validate”, “compile” e “test”.

**POM:**

* Tutte le informazioni relative a un progetto Maven sono centralizzate nel file pom.xml

**Coordinate:**

* Un progetto Maven è identificato da una tripla di valori: <groupid, artiffactid, versione>

**Dipendenze:**

* Maven consente di dichiarare le dipendenze del progetto in modo dichiarativo nel file pom.xml
* Non è necessario scaricare manualmente i file JAR e includerli nel progetto.
* Maven adotta un sistema di repository. Un repository può essere locale, remoto o centrale.

**Archetipi:**

* Gli archetipi sono modelli di progetto da cui un programmatore può partire quando crea un nuovo progetto Maven.
* Crea una struttura di cartelle e un file POM in base all’archetipo scelto.

**GIT**

Git è uno strumento per il controllo di versione che consente a gruppi di sviluppatori di lavorare contemporaneamente su più file. I file sono memorizzati in un repository condiviso (tipicamente online), e le copie sono conservate localmente sui client.

**Operazioni dei client:**

* Creare una copia di lavoro dei file come sono nel repository online.
* Modificare la copia locale dei file
* Eseguire il commit delle modifiche con un messaggio che specifica cosa è stato cambiato.
* Spostare la propria copia nel repository centrale

I commit sono come istantanee del codice accompagnate da un messaggio. I commit possono essere annullati, e gli utenti possono selezionare quali file includere nel commit e quali escludere.

**Git Branching:**

* Il concetto di “branching” in Git è fondamentale per la gestione del lavoro parallelo e lo sviluppo di funzionalità isolate. Un “branch” rappresenta una linea di sviluppo separata, che consente a più persone di lavorare contemporaneamente su diverse funzionalità senza interferire l’una con l’altra.

Quando si inizia a lavorare su un progetto Git, il branch predefinito è generalmente chiamato “main”. Questo rappresenta il punto di partenza principale del progetto.

**Gitflow:**

* Gitflow è un modello di branching alternativo che coinvolge l’uso di branch di funzionalità e branch principali multipli. I branch possono essere remoti o locali.

**Comandi:**

* **git pull 🡪** Aggiorna la tua copia di lavoro con eventuali modifiche del repository remoto.
* **git add 🡪** Aggiunge uno o più file all’area di staging, preparandoli per il successivo commit.
* **git commit 🡪** Esegue il commit delle tue modifiche alla tua copia di lavoro locale.
* **git push 🡪** Carica le modifiche locali confermate nel repository remoto.
  + *Nota: Se qualcuno altro ha caricato il suo codice nel repository remoto dopo l’ultima volta che abbiamo eseguito un pull, allora dobbiamo eseguire un altro pull, altrimenti non possiamo effettuare il push.*
* **git status 🡪** Visualizza lo stato della directory di lavoro e dell’area di staging.
* **git branch 🡪** Elenca, crea o elimina branch
* **git checkout *name* 🡪** Si sposta su un altro branch o su un commit specifico. Può anche essere utilizzato per creare e spostarsi su un nuovo branch in un’unica operazione.
* **git merge *name* 🡪** Fonde un branch dato con il branch corrente, creando un commit di merge.
* **git init 🡪** Crea e inizializza un nuovo repository git locale
* **git clone *remote url* 🡪** Crea un repository git locale clonando un repository remoto esistente.