INTRODUZIONE

Per lo svolgimento della tesi ne “Continuous Integration”, ho dovuto appoggiarmi sulla creazione di una applicazione chiamata Nota Spese.

I tool che sono stati utilizzati sono i seguenti:

-Linguaggio di programmazione Java/JavaFX

-Installazione della JDK e JRE -🡪 si collega con hypersqldb perché è una precondizione per poter usare questo database

-Utilizzo del pattern MVC

-IDE di sviluppo: IntelliJ

-Installazione di Maven, con compilazione del Pom.xml

-Hibernate (scelto un database embedded, HSQLDB) (parlare delle annotazioni e del many to many)

-HSQLDB (che è un RDBMS), link <http://www.html.it/articoli/introduzione-a-hypersql/>

-GIT <https://git-scm.com/book/it/v1/Basi-di-Git-Repository-Git>

(problemi git: non so come introdurre i comandi)

-JUNIT + file che ho creato per JUNIT

<https://examples.javacodegeeks.com/core-java/junit/junit-before-and-beforeclass-example/>

-Jenkins

“[**Jenkins**](http://jenkins-ci.org/) is an open source automation server. With Jenkins, organizations can accelerate the software development process through automation. Jenkins manages and controls development lifecycle processes of all kinds, including build, document, test, package, stage, deployment, static analysis and many more.

You can set up Jenkins to watch for any code changes in places like SVN and Git, automatically do a build with tools like Ant and Maven, initiate tests and then take actions like rolling back or rolling forward in production.

**PARLARE DI GIT**

Git è uno strumento nato dall’esigenza di lavorare in team (strumenti che già esistevano) ma che allo stesso tempo doveva avere dei requisiti che avrebbero facilitato al massimo il lavoro dei programmatori. I requisiti voluti e raggiunti sono:

-Velocità

-Design semplice

-Ottimo supporto allo sviluppo non-lineare (migliaia di rami paralleli)

-Completamente distribuito

-Capacità di gestire, in modo efficiente (velocità e dimensione dei dati), grandi progetti

**COME FUNZIONA GIT**

Git non immagazzina i dati e non li registra, li considera come delle istantanee, per cui ogni volta che viene salvato un certo momento del progetto, lui fa un'immagine di tutti i file in quel momento, salvando un riferimento all’ istantanea. Per efficienza, non risalva i file non modificati che stanno nell’istantanea, ma crea un riferimento al file precedentemente “scattato” (salvato).



Figura x.

Come si può vedere nella figura X, nella versione 1 si salvano tre rami diversi, nella versione due vengono modificati A e C, mentre B rimane invariato, per cui avrà un riferimento alla versione 1.

Al confronto con gli altri VCS (version control system), Git esegue la maggior parte delle sue operazioni in locale, non occorre collegarsi alla rete e puoi cambiare da un ramo a un altro in maniera istantanea. Git si ricorda di tutte le modifiche fatte nei mesi, e può mostrare la differenza tra le due versioni in maniera locale. Quindi possiamo lavorare in maniera offline e caricare i file modificati (tramite upload) una volta che avremo la connessione.

Una qualità di Git è l’integrità, ovvero, qualsiasi cosa che viene salvata non passa inosservata perché una checksum si intromette ogni volta e sarà un riferimento a quello che salviamo. Inoltre difficilmente possiamo cancellare erroneamente tutto in maniera definitiva. Se carichiamo preventivamente in remoto è pressochè impossibile.

**GLI STATI DI GIT**

I file di Git sono classificati e diramati in vari strati, i primi due sono tracciato (tracked) e non tracciato (untracked).

I file non tracciati sono quelli che non sono stati salvati nell’istantanea o nell’area di stage.

I file tracciati sono tutti quelle che sono nelle istantanee e ha sotto di sé tre stati: committed (committati), modified (modificati) e staged (in stage).

Committato significa che il file è al sicuro nel database locale. Modificato significa che il file è stato modificato, ma non è ancora stato committato nel database. In stage significa che hai contrassegnato un file, modificato nella versione corrente, perché venga inserito nell’istantanea della prossima commit.

Questo ci porta alle tre sezioni principali di un progetto Git: la directory di Git, la directory di lavoro e l'area di stage.



Figura x1. Directory di lavoro, area di stage e directory di Git.

La directory di Git è dove Git salva i metadati e il database degli oggetti del tuo progetto. Questa è la parte più importante di Git, ed è ciò che viene copiato quando si clona un repository da un altro computer.

La directory di lavoro è un checkout di una versione specifica del progetto. Questi file vengono estratti dal database compresso nella directory di Git, e salvati sul disco per essere usati o modificati.

L'area di stage è un file, contenuto generalmente nella directory di Git, con tutte le informazioni riguardanti la tua prossima commit.

Il flusso di lavoro (workflow) di base in Git funziona così:

1. Modifica i file nella tua directory di lavoro
2. Mettere tutto in stage, facendo le istantanee che andranno all'area di stage
3. Eseguire il comando commit, ovvero salva i file nell'area di stage in un'istantanea permanente nella tua directory di Git.

Se una particolare versione di un file è nella directory git, viene considerata già eseguito il commit. Se il file è stato modificato, ma è stato aggiunto all'area di staging, è in stage. E se è stato modificato da quando è stata estratto, ma non è in stage, è modificato.

Il ciclo di vita dello stato dei file viene riassunto in questa figura:



Figura y.

**COMANDI GIT**

Git dispone di tantissimi comandi, per ogni azione che abbiamo descritto o per ogni azione che vogliamo far fare a git, ci sono dei comandi che si possono scrivere in un prompt. Nel mio caso ho installato il plugin di Git sull’IDE IntelliJ, e ora vi elencherò i comandi più usati e importanti.

**status** mostra i file non tracciati

**add** per tracciare un nuovo file

**commit** salva il tuo elaborato, dandogli un checksum, aggiungendo eventuali messaggi (scrivere cosa si ha modificato), crea l’istantanea.

**Rm** (remove) rimuove un file dalla directory di lavoro.

**Mv** (rename) rinonima un file

**Log** mostra tutte le informazione del progetto (autore, messaggi, data creazione)

**Fetch** scarica i dati da un progetto remoto

**Pull** scarica e unisce il ramo indicato in quello su cui stai lavorando

**Push** carica il lavoro fatto nel repository remoto

**Branch** crea un nuovo ramo

**Checkout** cambio il ramo in cui sono in un altro che specifico e sposta l’HEAD\*

**Merge** unisce due rami in uno unico

\*HEAD: puntatore a una istantanea, se faccio una commit, sarà figlia del corrente HEAD.

**DIRAMAZIONE IN GIT**

Riprendiamo il discorso della commit: quando la eseguiamo, facciamo una istantanea che possiede autore, messaggi di versionee la checksum.

La prima commit che facciamo genera un padre, ovvero l’istantanea salvata è un padre. Dalle prossime commit che faremo, si genereranno dei figli (rami)

Ramo: è un puntatore a uno dei suoi padri, il ramo principale viene chiamato MASTER.

Diramare: dirigere il flusso principale in altre direzioni senza pasticciare il flusso principale. La diramazione e il cambio di un ramo all’altro è veramente leggera. Questo mi garantisce una grande fluidità del lavoro, ad esempio se sto lavorando sul ramo Sviluppo, ma il mio capo mi chiede di modificare del codice dal progetto originale, posso cambiare ramo senza problemi, effettuare la modifica, e poi ritornare sul mio ramo Sviluppo. Quest’ultimo ramo, non avrà con sé la modifica dell’altro ramo, occorrerà fare una **rebase**, una volta ultimato il ramo Sviluppo.

**Workflows distribuiti**

Ci sono vari modi per distribuire il codice su cui lavorare e come condurre il lavoro, modi che cambiano anche dalla quantità di persone su cui lavorano sul progetto. Analizzo le metodologie di utilizzo dei workflows:

**Repositorio singolo**

Tutti gli sviluppatori scaricano il progetto e Git non permette la sovrascrizione del lavoro altrui, per cui se il lavoratore A fa una PUSH e dopo qualche tempo anche il lavoratore B tenta di fare una PUSH, questa azione non causa sovrascrizione del lavoro di A, perché Git non permette di fare una push al lavoratore B, se esso non è aggiornato all’ultima modifica di A.

**Repositorio con manager d’integrazione**

Si ha un repositorio ufficiale e ogni sviluppatore tiene un proprio repositorio dove fa le proprie push e chiederà al manager di prendere le sue push con una pull. Lavoreremo con uno o più workflow in questo caso.

**Repositorio con Dittatore e Tenenti**

Variante del repositorio con manager, usata solitamente da progetti con centinaia di collaboratori (esempio: Kernel Linux).

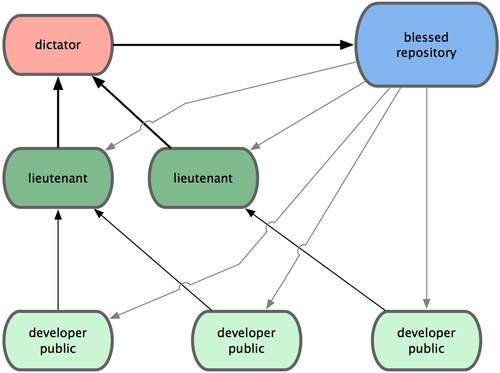
Molti manager d'integrazione sono responsabili di certe parti del repositorio e vengono chiamati tenenti. Tutti i tenenti hanno un manager d'integrazione conosciuto come "dittatore benevolo". Il repository del dittatore benevolo è il repositorio di riferimento dal quale tutti i collaboratori prendono il codice. Il flusso di lavoro è il seguente:

-Sviluppatori normali lavorano sul loro branch ed eseguono un rebase del proprio lavoro sul master. Il branch master è quello del dittatore.

-I tenenti uniscono il lavoro degli sviluppatori nel proprio branch master.

-Il dittatore esegue l'unione dei branch master dei tenenti nel proprio branch master.

-Il dittatore esegue una push del proprio ramo master nel repository di riferimento, cosicché gli sviluppatori possano accedervi.



Questo tipo di workflow non è comune ma può essere utile in progetti molto grandi o in ambienti con una gerarchia forte, perché consente al leader del progetto (il dittatore) di delegare molto del lavoro e raccogliere vasti sottoinsiemi di codice in punti diversi prima di integrarli.

**JUNIT**

Uno degli aspetti più importante della programmazione è il testing, perciò per il mio progetto ho utilizzato JUnit, che è un framework di unit testing per il linguaggio di programmazione Java.

Lo ho aggangiato al mio IDE, creato una cartella di test e designata come cartella dei tests.

Creazione test: adesso potrete andare in una qualsiasi classe del progetto, che volete testare, basta andare dentro alla classe, cliccare sul nome della classe e premere ALT+ENTER -> [create test]

Esempio: public class NoteSpese {🡨ALT+ENTER su NoteSpese in questo caso

Nel primo label: testing library, impostate JUNIT4; se si accende una lampadina che vi dice che non riesce a trovarlo, potete cliccare sul bottone FIX, e potete impostare JUNIT nel caso che lo abbiate ancora importato nel progetto; fatto ciò, sempre nella finestrella, potete selezionare le funzioni/metodi che volete testare. Vedrete che cliccando OK, vi si creerà una classe dentro la cartella *tests.*

Come notate, sopra alla funzione che volete testare, c’è l’annotazione @*test,* indica che andremo a fare un test su quel metodo.

A fianco di @*test* possiamo aggiungere due parametri che possono essere:

1. @*Test*(expected=ParseException.class), che si aspetta che il test si aspetta che lanci una eccezione di parsing.
2. @*Test*(timeout=100), che controlla che non sia troppo lunga l’esecuzione della funzione, in caso che passi troppo tempo (settato attraverso il timeout), richiamerà un fallimento del test (utile per vedere se si creano dei loop nella funzione).

Annotazioni:

possiamo dare un ordine di esecuzione nel caso che abbiamo tanti test. Mettiamo sopra ad ogni test una certa annotazione che ora andiamo a vedere:

@Before: questa annotazione viene eseguita prima di ogni test. Solitamente usata per preparare l’ambiente di test (leggere un file, inizializzare una classe con tutti i dati)

@After: questa annotazione viene eseguita dopo ogni test.

@BeforeClass: questo metodo è eseguito una volta, prima di tutti i test. Usato per alcune attività che può essere ad esempio la connessione a un database. I metodi marchiati con questa annotazione vanno definiti in maniera statica per lavorare con **JUNIT**.

@AfterClass: questo metodo è eseguito una volta, dopo che di tutti i test siano finiti. Usato per vedere se tutto sia terminato in maniera corretta e pulita, per esempio si può annotare sulla chiusura di di un database. I metodi marchiati con questa annotazione vanno definiti in maniera statica per lavorare con **JUNIT**.

@Ignore o @Ignore("sto disabilitando il test perché…"): alcune volte volete temporaneamente disabilitare un test o un gruppo di test. Ci basterà appore @Ignore sopra @Test:

Esempio: @Ignore  
 @Test  
 public void modificaNoteSpese() { ... }

Così non verrà eseguito quel test, tuttavia **JUNIT** riporterà quanti test il numero di test ignorati.

*Assert*: è il metodo che ti dice la risposta del test (se correct, se failure, se error ) e il più importante nel test. I più usati sono *assertEquals*(valore che mi aspetto, valore che la funzione provata mi da)

*assertTrue*(verifica se il risultato del test sia true)

*assertFalse*(verifica se il risultato del test sia false)

**JENKINS**

Jenkins è un server automatizzato open source. Possiamo accellerare l’organizzazione dello sviluppo software attraverso l’automatizzazione. Jenkins gestisce e controlla lo sviluppo del ciclo di vita di tutti i processi, includendo documenti, pacchetti, test, distribuzione, analisi statistica e altro ancora.

La cosa che riteniamo più importante ai fini della tesi è il poter impostare Jenkins come un “osservatore” che al momento di un cambiamento di codice di un VCS (nel nostro caso, Git), automaticamente inizia a fare dei test e fa azioni preimpostate. Questa è la continous integration.