***Scritti da Gabriel: seguono l’ordine delle lezioni e comprendono anche accenni o completamenti dei laboratori del corso***

***28/02/2022: Introduzione ed Issue Tracking System/ITS***

Il corso tratta principalmente la condivisione e creazione di software gestito in maniera condivisa. In questo senso, il programma deve essere controllato nel suo codice sorgente e poi integrato attraverso la fase di build. Lo sviluppo avviene attraverso una modalità di integrazione continua, al fine di verificare se sia effettivamente funzionante in vari ambienti di sviluppo.

In linea di massima si segue il principio della *continuous delivery pipeline*, che è un'implementazione del paradigma continuo, in cui build, test e distribuzioni automatizzati sono orchestrati come un flusso di lavoro di rilascio. Tutto ciò fa parte della creazione Agile dei progetti, al fine di gestire vari processi tradizionali e fornire molteplici funzionalità. Tutto ciò viene garantito attraverso vari principi, prendendo ad esempio l’esplorazione continua dei bisogni del mercato/clienti attraverso feedback, integrazione continua di caratteristiche, creazione e consegna di fasi di produzione e rilascio a richiesta, rendendo il prodotto disponibile una volta correttamente pronto.

Gli esempi di *build tracking* sono il bugtracker di Ubuntu, con diverse migliaia di bug giornalmente segnalati. Segue *Jira*, software molto utilizzato, dove si riportano le segnalazioni, menù e a destra la versione del software in cui un certo bug è stato sistemato. Anche GitHub presenta una serie di tag ed è un buon issue tracking system (trova quindi le criticità, mettendo nero su bianco quante più cose possibili, utile per tutti gli attori di un progetto software). Questi sistemi mantengono una lista di problemi (issues), permettendo di trovare o localizzare una serie di problemi e permettendo di risolvere anche a catena i problemi (per esempio tramite servizio clienti). Un esempio di questo sono i bugtrackers, vedendo se c’è già un bug di quel tipo. GitHub e Jira si presentano come gli strumenti più usati in generale.

Abbiamo per esempio *l’ITS per la gestione di Progetto*, che facilita il management, lo sviluppo e il ciclo di vita di un certo progetto, per esempio anche in base alle richieste del cliente, segnate come requisiti. Quando nasce una certa attività, dopo una fase di analisi, si usa una stima per capire quanto tempi si impiega per certe attività. Si devono usare il più possibile, per agevolare la vita di tutti, sviluppatori e project manager, che deve essere bravo a non far perdere tempo al suo team, confrontandosi poi direttamente col cliente nelle riunioni apposite di avanzamento lavori, dette *SAL/Stato Avanzamento Lavori*. Sicuramente tutto questo è una misura della qualità, capendo dove effettivamente si è verificata una certa problematica/bug e avendo quindi istantanee. È importante dare una giusta priorità alle singole attività, quantificando il tempo impiegato singolarmente e capendo l’effettiva efficacia.

*I Work Item* sono degli strumenti utili per capire come sviluppare e lavorare su un progetto, magari dal punto di vista del cliente facendo capire le sue esigenze e come usarlo. Una cosa utile in questi strumenti è capire la situazione di un bug, riproducendolo e capendo i passi che hanno portato alla sua scoperta/segnalazione/risoluzione. Sempre in merito ai work item, esistono vari campi identificativi, come il progetto di riferimento, un codice univoco, un tipo (campi, stati, schermate, workflow), stato nel workflow, priorità, stato di risoluzione, versione di riferimento dove è stato trovato il bug.

Vi sono poi i campi come le etichette di classificazione, collegamenti tra i vari Work Item, assegnatario/responsabile, segnalante, date di ultimo aggiornamento/risoluzione/stima delle attività, tempo speso ed eventuali allegati.

Per portare a compimento le attività, l’insieme di stati e transizioni dei Work Item e del loro tempo di vita è il *Workflow*, insieme di stati e transizioni di un Work item e che, associato ad un progetto, ne ricostruisce lo storico delle segnalazioni, tracciandone soluzione/risoluzione. Determinate esigenze del cliente sono documentate nei *SLA (Service Level Agreement)*, per esempio la risoluzione di una certa problematica X entro un certo tempo Y. Ciascun requisito può essere categorizzato e classificato in *macrorequisito*, *requisito* e *sottorequisito*, suddividendo e specificando l’individuazione e l’utilizzo delle sottoattività.

Importanti in questo senso sono i *collegamenti*, definendo le relazioni tra i Work item, solitamente bidirezionali e utilizzate come possibile criterio di ricerca, capendo o meno le relazioni tra i componenti.

Altro strumento utile sono le *notifiche*, che segnalano la risoluzione e fungono da tracciamento. Citiamo anche i *filtri*, i quali possono essere salvati ed esportati; in generale sono quindi una base per creare report, board e dashboard. I report sono suddivise in tipologie e caratterizzate graficamente da vari tipi di grafico (torta, istogramma, etc.). Utili anche le *board/bacheche*, visualizzando i work item di uno o più progetti e visualizzandone l’insieme e le interazioni tramite possibili filtri e interagendo velocemente con ognuno di essi.

Un buon *ITS* deve essere configurato identificando i *processi*, quindi possibili best practices definiti dai framework usati oppure vincoli dati dallo stesso cliente e anche identificare/configurare gli strumenti utili, definendo tipi, campi custom, work item e collegamenti, creando uno strumento di tracking e risoluzione facile ed efficace.

Un progetto normalmente ha una fase di inizio, ne viene pianificata una versione di rilascio, vengono risolte e create nuove attività (fase di sprint, nell’esempio del caso d’uso Jira delle slide) e poi rilasciata una versione. Nel team, in base all’ITS il *manager* (amministratore) definisce, una volta creato il progetto, il processo da seguire, con relativi work item, eseguendo e censendo un modello di stima e aggiungendo gli specifici utenti, assegnando loro i giusti permessi. Nel team di sviluppo gli *utenti* (intesi come team di sviluppo) selezionano i work item, avviano e completano la lavorazione documentando le attività e programmando il rilascio.

Il manager monitora l’avanzamento dei lavori sulla base dei report degli utenti, definendo le nuove versioni ed iterazioni, dando report specifici al cliente, magari appunto anche con le notifiche.

Tra i benefici di utilizzo di un ITS si ha l’implementazione di processi, verificandone l’adozione, misura di qualità e soddisfazione del cliente, produttività del team riducendo sprechi e spese.

***04/03/2022: VCS, tipi di VCS e workflow patterns***

La repository di deposito del codice sorgente e conseguente evoluzione di un progetto è cosiddetto *Version Control System (VCS)* gestendo i cambiamenti di documentazione o codice sorgente documentando anche chi possa aver eseguito una certa modifica. Ogni revisione può essere confrontata con le altre tramite lo storico e fare eventuale *rollback*, tornando ad una versione precedente, oppure svolgere una attività di *merge* tra versioni presenti, quindi unendole.

Prendiamo anche gli SCM, quindi *Source Code Management systems (SCM)/Version Control System (VCS)*, nati per permettere la collaborazione.

Caratteristiche principali sono la tracciabilità di ogni modifica, avendo un percorso completo di un prodotto e della sua creazione e facilitando il ripristino nell’eventualità, nonché individuazione e risoluzione di conflitti e condivisione di commenti e documentazione.

Segnaliamo i *benefici*, mantenendo la storia completa dei cambiamenti ai file e lavorando senza interferenze in differenti rami di sviluppo (*branching*), verificando se le modifiche fatte possono corrispondere ed essere sensate sulla base di ciò che è presente. Ogni progetto ha un ramo principale, chiamato *trunk/master*, da cui ne si diramano tutti gli altri. La tranciabilità ci dà la conoscenza anche dei changelog tra le varie versioni, differenziando le singole modifiche tra le successive versioni dei progetto.

Molte persone gestiscono le singole versioni tramite dei *VCS locali/Local VCS*, quindi tra directory dello stesso PC; ovviamente è semplice ma difficile, specie in ambiente condiviso e recuperando versioni precedenti. Utilizzando un *Version Database* abbiamo modo di vedere le patch tra le versioni, ricreando lo stato di qualsiasi file in qualsiasi momento. Essi non gestiscono la condivisione (es. IDE vari) e quindi possono risultare inadatti.

Altri sistemi sono i *Centralized SCM (CVCS),* meno vecchi e molto diffusi sperabilmente avendo una versione di backup in cui i singoli computer hanno una sola versione; ogni volta che viene eseguita una modifica, essa si ripercuote anche nel nodo centrale, verificando se sia i due rami siano allineati o meno. Il version database viene gestito da un server centrale e ogni sviluppatore è un client che ha nel suo spazio di lavoro solo una versione alla volta del codice. Il più famoso di questi è SVN (descritto meglio sotto quest’ultimo).

Nel caso di conflitti, essi vengono risolti in locale, una volta magari scaricata la versione corretta se disponibile da sistemi vari oppure online.

Un *Distributed VCS/DVCS* è distribuito per duplicazione in ogni nodo; quando il nodo centrale non è disponibile, è possibile continuare a lavorare e registrare i cambiamenti. La fase di registrazione avviene attraverso commit e singole push, risolvendo i singoli conflitti anche grazie all’impostazione di diversi tipi di flussi di lavoro che non sono possibili in sistemi centralizzati.

Strumenti più interessanti e presenti oggi sono i *Cloud-Based SCM*, che sono dei “VCS As a Service” il Version Database è gestito in un servizio cloud, delegando quindi la gestione ad un servizio esterno. In questo caso il codice sorgente non è nell’infrastruttura aziendale.

Forniscono altri servizi oltre al VCS di tipo grafico/visuale.

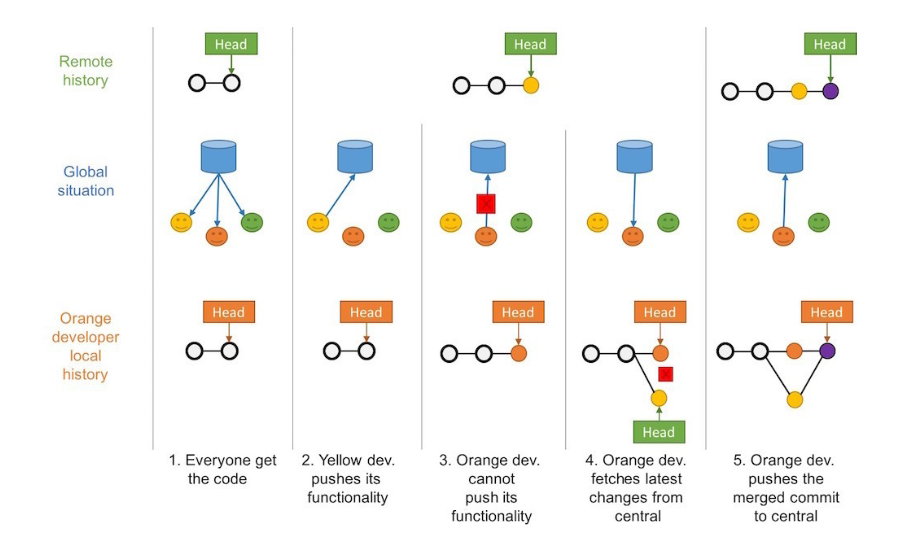
Parliamo quindi di *terminologia*:

* *commit*, quindi cambiamenti sul database, nuove versioni e differenze varie. Questi permettono di versione capire se una sia più aggiornata o meno. Essi possono essere locali o remoti e l’ultima commit cronologicamente è la HEAD.
* *branch*, puntatore verso un commit specifico, salvando uno *snapshot*, quindi un’istantanea della situazione fino a quel momento. Per integrare un branch si deve eseguire un’operazione di merge.
* *pull request*, un modo di gestire branch e di unire questi al ramo master. Essa può richiedere di inoltrare le modifiche sul server centrale. Prima di fare il merge si ha la possibilità di rivedere le modifiche apportate; a quel punto viene poi eseguita la modifica, eseguendo il merge nel branch di destinazione.

Ora vediamo i singoli *workflow patterns* utilizzati.

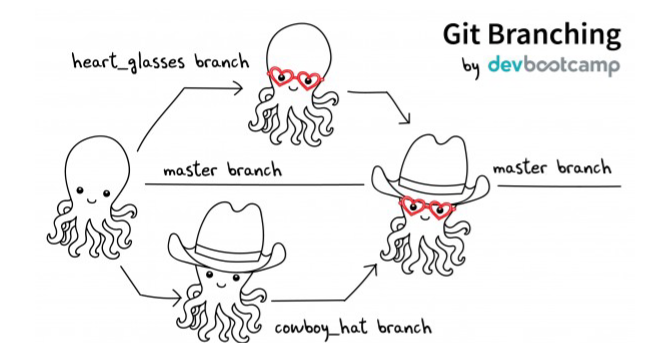
Un primo esempio è di tipo *centralized*, in cui, se il nodo principale è in down o la coronologia è corrotta, la collaborazione è bloccata. Se la versione non fosse allineata rispetto ad una dei colleghi, la modifica non viene presa. Dovrà eseguire una pull quindi per la versione corretta, facendo commit e pushando la versione corretta una volta fatto il merge delle modifiche.

L’esempio grafico segue:



Parliamo poi di *feature branch*, inserendo un branch per caratteristica. Esempio esplicativo è il polipo con gli occhiali e col cappello. Le modifiche vengono mantenute separate, senza inquinare le modifiche eseguite dagli altri branch. La collaborazione quindi è più facile e permette di lavorare senza disturbare la base di codice principale. I conflitti di merge, si intende, sono più facili da tracciare in questo caso e agevola la collaborazione.

Ecco lo schema, modalità che sarà vista negli assignment:



Altra modalità è il *GitHub Flow*, dove si ha una pull request per avvertire che avviene la revisione. Ciascuna commit agisce in modo singolo e se possibile viene rinominare una commit, riutilizzata, salvata, permettendo anche un backup. Si intende che questo pattern voglia essere semplice e collaborativo senza dare vincoli particolari; similmente a questo ci sta anche la *GitLab Flow* che implementa questo principio su GitLab e la differenza principale implica che GitLab permette di avere grafici per verificare lo stato di avanzamento del progetto e GitHub permette di eseguire già le modifiche se in quel momento ci si trova nel ramo master. Entrambe sono molto simili e molto veloci.

Poi segue il *Gitflow*, estendendo le funzionalità del Feature Branch Workflow, operante come target a larga scala. Dal ramo master stacchiamo il ramo di *develop*, ramo di sviluppo e gestione del progetto. Ad esso, parallelamente vengono implementati dei *release branches*, qualora si voglia ideare un rilascio.

In esso vengono integrate tutte le attività. Nel ramo master si possono attivare dei meccanismi per avere le release notes, capendo tutte le attività dei vari ITS e rilasciandole nel modo migliore. In questo caso i singoli branch hanno ciascuno un proprio ruolo.

Altro workflow molto utile è quello di tipo *fork*, pattern ereditato da GitHub e utilizzato spesso nei progetti open source. Qui ognuno esercita, una volta preso il controllo della directory principale, delle pull requests che possono essere integrate su autorizzazione di chi gestisce il master branch.

*Un confronto quindi tra CVCS/DVCS* (si intende per lock il fatto di non poter eseguire modifiche)

Immagine che contiene testo

Descrizione generata automaticamente

***07/03/2022: Laboratorio 1: GitHub come strumento di ITS***

In generale GitHub risulta essere uno strumento valido di versionamento e di tracking system, offrendo funzionalità di collaborazione e gli account free riescono ad avere collaboratori illimitati e repository con un certo numero di azioni (2000 azioni al minuto al mese); ne si può permettere facilmente il fork, quindi un utente fa una sorta di copia personale della repo per poterci poi lavorare. Attenzione alla licenza di un certo codice o libreria, di solito specificato, per poter meglio comprendere l’utilizzo.

Ci occupiamo di aprire una *segnalazione* (tramite la sezione *Issue*, cliccando su *New*). Eventualmente ad una segnalazione è possibile associare un commento, nel caso chiudendo anche la issue tramite il commento. È possibile sottoscriversi alle singole segnalazioni, verificando se la segnalazione viene risolta o meno. Ad una segnalazione è possibile associare delle etichette (*labels*, e.g. Bug, Documentation, Enchancement, etc.) oppure crearne facilmente di personalizzate.

Altra sezione importante sono le *Milestones*, equivalente delle versioni. All’atto della creazione, oltre al titolo, si ha anche una data entro cui si intende rilasciarla (*due date*). Ad ogni milestone è associata una percentuale di completamento.

È inoltre possibile associare issues alle milestones, sapendo l’ordine delle attività e organizzando al meglio il lavoro.

Andando nella repository e cliccando *Settings*, nella sezione *General*, scorrendo verso il basso, si ha *Set up templates*, di cui ci sono alcuni esempi come *Bug Report* (contenente già un formato della segnalazione dell’utente, come degli screen, desktop, screen, etc.). i template vengono aggiunti proprio sotto forma di commit, cliccando su *Propose Changes* e poi su *Commit Changes*.

A questo punto viene creata nella repo una cartella chiamata *Issue Templates*, contenente dei file .md (markdown) che possiede i template precedenti, poi eventualmente editandoli e customizzandoli.

A questo punto cliccando su *New Issue*, si vede che si facilita la compilazione della issue, avendo le categorie scelte dal template. Eventualmente si può creare anche un form, magari con un template custom, avendo ancora più campi e rendendo quindi la segnalazione agevola e anche esteticamente carina.

Andando a *Close with comment*, la segnalazione può essere chiusa (passerà in stato *Closed*). Naturalmente le varie segnalazioni possono essere *filtrate*, ad esempio a seconda dello stato.

Tutto questo fa parte del workflow, composto di vari stati (*Open/Closed/Reopen*, l’ultima nel caso si riapra la segnalazione).

La parte di notifiche viene gestita come prima, entrando sulla segnalazione e cliccando *Subscribe*. Nella repo, cliccando su *Settings* e poi *Collaborators*, posso aggiungere delle persone che collaborano ad un certo progetto con *Add People*. A quel punto l’altra persona riceve una notifica con cui deciderà o meno di accettare la collaborazione.

Per poter conoscere lo stato di un progetto, si ha la sezione *Board*, all’interno della sezione *Projects*. Noi vedremo le board di tipo *Agile*.

Creato il progetto, con un template, un nome ed una descrizione, si possono associare segnalazioni in modalità *drag and drop*, dando segnalazioni aperte oppure chiuse, aggiornandone facilmente lo stato.

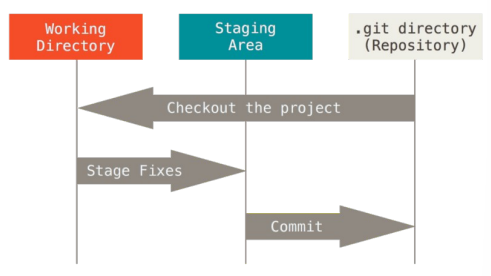
Questo strumento è un simil report e segnala l’avanzamento delle segnalazioni.

Per sopperire alla mancanza di campi custom, possono creare gerarchie di etichette associando vari stati di avanzamento, negli esempi reali di uso di GitHub come ITS.

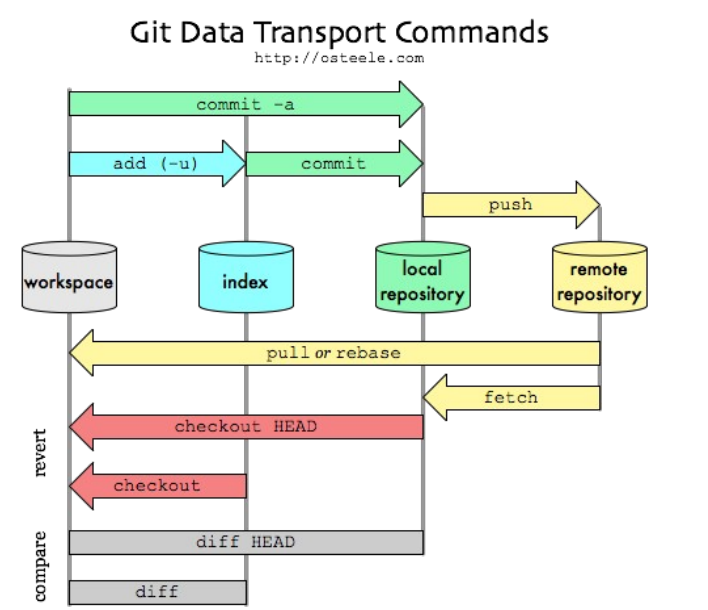
***11/03/2022: Git e SVN vs Git***

Oggi andremo a vedere la possibilità di storicizzazione e lo stesso Git, software di controllo distribuito utilizzabile anche in CLI, ispirandosi a strumenti proprietari analoghi per facilitare lo sviluppo del kernel Linux. Le caratteristiche sono lo sviluppo su branch diversi (locali o condivisi), con eventuale merge (*branching and merging)*, operazioni fatte in locale e la possibilità di avere backup multipli tramite distribuzione delle directory, adottando vari workflow. Ogni commit viene identificato da un ID (checksum SHA-1) che ne garantisce l’integrità; non è quindi possibile cambiare un commit senza modificarne l’ID, altrimenti Git ferma l’operazione per evitare di perdere i commit precedenti.

Altra caratteristica è la *staging area*, dove vengono validati i file modificati che potranno essere versionati da un commit. Qui aggiungiamo tutto ciò che interessa ai fini del commit.



Git rimane strumento open-source e pensa il filesystem come delle snapshot in cui, ogni volta che si fa un commit o si salva lo stato del progetto, si ha un’istantanea del progetto e dello stato in quel momento dei file salvandone, un riferimento. Se il file è già presente, non verrà di nuovo memorizzato, ma si inserirà solo un link a file precedenti già memorizzati. I file delle copie locali in Git possono essere nella *Working directory,* quindi modificati ma non ancora validati, nella *Staging Area,* quindi validati ma non ancora committati, nel *Repository locale,* in stato *Committed*.

Un file in Git può quindi avere vari stati:

* *untracked* (solo locale), in cui Git non ha ancora archiviato il file e non ne sa lo stato;
* *unmodified*, ancora non modificato in Git;
* *modified*, dunque aggiunto ed editato il file;
* *staged*, salva lo snapshot nella Staging Area;
* *committed*, storicizza l’operazione di modifica prendendo il file dalla Staging Area e salvandolo nella repo locale;a prima configurazione di Git avviene con i seguenti comandi:

*git config --global user.name "inserire username qui"*

*git config --global user.email inserire email qui*

Si invocano poi i comandi con*: git config –list*

Le configurazioni possono essere fatte a vari livelli:

● system: per l’intero sistema per tutti gli utenti

● global: per il singolo utente

● local (di default): per singolo repository

Si noti anche il comando *rebase*, che recupera le modifiche di un branch mettendole in HEAD ad un altro.

*Lista dei comandi* (nell’ordine, creazione/clonazione repo, add/commit, stato/ripristino modifiche, branching/merging/repo remoti):

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

*SVN VS GIT*

Prima di tutto diciamo che *SVN* è un sistema centrale di controllo di gestione a cui possono accedere tutti gli utenti e le modifiche apportate non possono essere unite insieme, facendo in modo due utenti non elaborino contemporaneamente un file. Esso è organizzato in cartelle, offrendo la possibilità di scaricare ed elaborare qualsiasi sottopercorso indipendentemente dal resto del percorso ad albero. I branch stessi sono creati come directory.

All’interno della macchina locale viene salvato solo l’albero su cui si sta lavorando e gli ultimi cambiamenti sono salvati in locale.

Abbiamo poi *Git*, che punta direttamente ai commit e permette il salvataggio in locale di tutta la cartella. Essendo molto veloce sulle medio/grandi operazioni risulta versatile; SVN se la cava su file binari e di grosse dimensioni. La semantica di Git è chiara, dicendo con ogni singolo comando cosa viene fatto.

*Sotto un confronto:*



***14/03/2022: Laboratorio 2: Git Work Flow e GitHub ITS***

Si crea una nuova repository; il prof consiglia di dare l’accesso in SSH, dando un’apposita chiave.

Ciò che viene fatto è la creazione di un file readme che viene committato.

Importante: occorre aggiungere una chiave SSH all’account per poter lavorare correttamente.

Questo viene fatto:

* tramite terminale (<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>)

Attenzione che Windows presenta il comando ls nella guida; ovviamente non è supportato dal caro Windows. Consiglio nel caso Windows il caso sottostante; per Linux basta il link precedente, trovando chiave SSH e poi aggiungendola.

* tramite GUI (aprendosi Git GUI, cliccando *Show SSH Key* e poi cliccando *Generate SSH Key.* Successivamente si va in Settings dell’account GitHub e poi si aggiungere la chiave in copia/incolla, eventualmente aggiungendo una passphrase). Eventualmente:
* <https://docs.github.com/articles/generating-an-ssh-key/>
* <https://support.automaticsync.com/hc/en-us/articles/202357115-Generating-an-SSH-Key-on-Windows#:~:text=On%20the%20Start%20Menu%20of,and%20select%20Show%20SSH%20Key>

Con quanto visto si vede che si integra una modifica, la si annulla e poi reintegrarla successivamente.

Successivamente cerchiamo di risolvere una segnalazione da noi creata e svogliamo l’attività, chiudendola in linea di comando.

Similmente, volendo aggiungere un commento, basta ancora referenziare (anche nel caso precedente) tramite uno sharp (#) la segnalazione e poi inoltrare la modifica.

Si ha poi un esempio di conflitto tra pull requests, in questo caso non sulla repo locale o su quella remota, ma sulla working copy. Una volta risolti i conflitti, viene aggiunto all’area di staging, committato e pushato.

***18/03/2022: Visione degli altri workflow: GitFlow e Fork Workflow; inizio framework SCRUM***

Parliamo del GitFlow ancora, dove si accenna ad uno sviluppo in parallelo rispetto al ramo di develop, bloccando tramite il ramo di release eventuali sviluppi di feature per la milestone principale.

Quindi andremo a fare un’attività di merge verso un ramo master, contenente l’ultima versione rilasciata.

Taggando poi una nuova versione, è possibile trovare il ramo con il tag corretto e fare una hotfix.

Proseguiamo vedendo il Git Flow (fine lab scorso), seguendo tutto quello che sta al link:

<https://danielkummer.github.io/git-flow-cheatsheet/>

La parte la aggiungo per totale completezza (non essendo presente con tutti i comandi a differenza delle altre nelle guide/pdf del prof).

*git flow init*

(Mettiamo il ramo per tutte le funzionalità di creazione)

Segue l’inizializzazione tipo così (per casino mio ho i branch diversi, il branch di develop si chiamava *develop*, ramo master è *master*).

Immagine che contiene testo

Descrizione generata automaticamente

Implementiamo un branch per una nuova feature:

Immagine che contiene testo

Descrizione generata automaticamente

*Immagine che contiene testo

Descrizione generata automaticamente*

Seguono gli altri comandi:

*git checkout develop*

*git flow feature start cappello*

*git add cappello.txt*

*git commit -m “aggiunto il file cappello”*

*git flow feature finish cappello*

Andiamo quindi a rilasciare la funzionalità presente aprendo un ramo di release (release branch) rilasciando la funzionalità richiesta.

Lo schema delle modifiche si vede con *git log*.

Avviamo poi:

*git flow release start v.1.0*

*echo “README V.1.0” >> README.txt*

*git flow feature finish*

Sul ramo di “develop” si sta continuando a lavorare, con il ramo master che con conclude mettendo insieme le feature dei rami di develop, ad esempio.

*git checkout release/v.1.0*

*git flow release finish cappello*

*git flow feature publish*

Il ramo occhiali è solo su develop, invece il ramo master avrà solo cappello il readMe.

Similmente switcha a develop (*git checkout develop*)

e incorporare altre funzionalità (*git feature start pipa*)

Noi quindi abbiamo visto la pubblicazione e la storicizzazione; inoltre con *feature publish*, viene pubblicata la release dal branch, eventualmente può essere chiusa e convalidata.

Vediamo quindi meglio fork workflow, completando la visione di tutti i workflow.

Vado quindi a lavorare su un repo pubblico su cui non ho i permessi con il fork, spostandosi nella mia working area e quindi modificarlo, clonarlo o altro.

Ad esempio:

*git clone* (indirizzo di un progetto, che inizia con “git@github”…)

A questo punto un semplicissimo esempio:

modifica di un file readMe, aggiunta (git add), commit (git commit “”) e si vede subito la modifica.

Creo quindi una pull request, dove all’altro progetto arriverà una notifica di modifica, che potrà essere integrata nell’originale.

Esempio reale con uso di repo distribuiti sono i fork su repo del cliente (caso consulenza di progetto) e inoltrazione della push su un branch con alias dell’originale.

Cominciamo a parlare del *framework SCRUM*, avendo un approccio a staffetta atto allo sviluppo dei prodotti, con obiettivi di massima velocità e flessibilità.

Di fatto magari i requisiti iniziali non soddisfano più l’evoluzione storica del mercato, prodotto ed eventuali controversie. Scomponiamo quindi il progetto, cercando di realizzare passo passo tutte le attività.

Esso è un processo agile (progetto AGILE), per sviluppo di progetti complessi e difficili da risolvere, realizzandoli possibilmente nel più breve tempo possibile.

Le priorità sono date dal business, avendo i team che si organizzano scegliendo il percorso migliore per consegnare le funzionalità con priorità maggiore.

Ogni 2 settimane oppure ogni mese, chiunque può vedere il software funzionante, decidendo se rilasciarlo nello stato attuale oppure proseguendo con un altro sprint, aggiungendo altre funzionalità al backlog del prodotto e successivamente riutilizzabili.

Esso è intuitivo, leggero, difficile da padroneggiare.

Principalmente è *trasparente* (con un linguaggio comune), *controllo* (pianificando ispezioni e strumenti per impedire variazioni inutili), *adattamento* (aggiustamenti per minimizzare ulteriori deviazioni dal focus progettuale).

L’insieme delle feature è composto dal *product backlog*, comprendendo tutte le attività conosciute ed implementabili nel prodotto. All’inizio di ogni sprint, il product owner scrive queste attività, dando priorità in base alle esigenze del business. Ciò forma lo *sprint backlog*, assegnando caratteristiche per gli sprint e poi espandendole gradualmente.

L’insieme di funzionalità quindi sarà implementata in una demo, per prevedere la preparazione di caratteristiche e compiendo una retrospettiva capendo le esigenze ed usi particolari dopo gli sprint.

I gruppi si auto-organizzando, dove il prodotto evolve attraverso “sprint” mensili. Il processo di creazione è quindi iterativo/incrementale, ottimizzando il controllo dello sviluppo e del rischio.

La loro durata è fissa e abbastanza breve, prevedendo un certo rischio che siamo disposti ad avere.