***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.

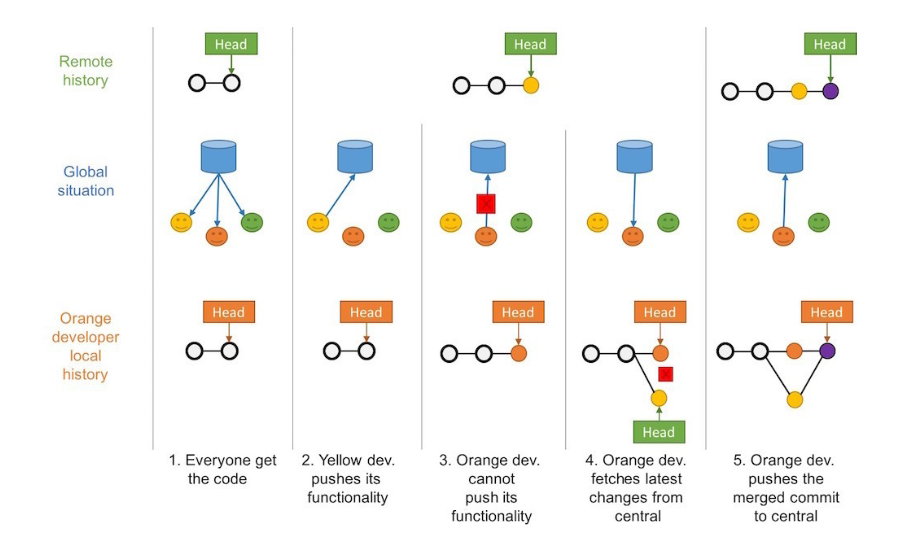
Parliamo quindi di *terminologia*:

* *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.

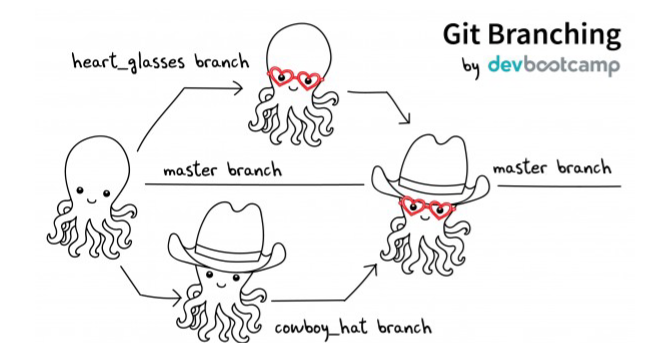
Un primo esempio è di tipo *centralized*, in cui, se il nodo principale è in down o la coronologia è corrotta, la collaborazione è bloccata. Se la versione non fosse allineata rispetto ad una dei colleghi, la modifica non viene presa. Dovrà eseguire una pull quindi per la versione corretta, facendo commit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

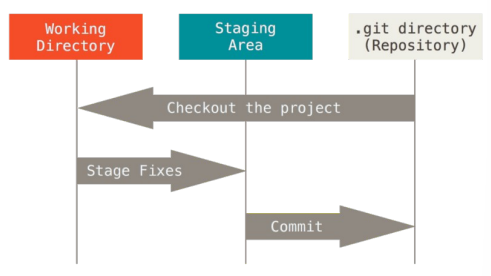
Questo strumento è un simil report e segnala l’avanzamento delle segnalazioni.

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.

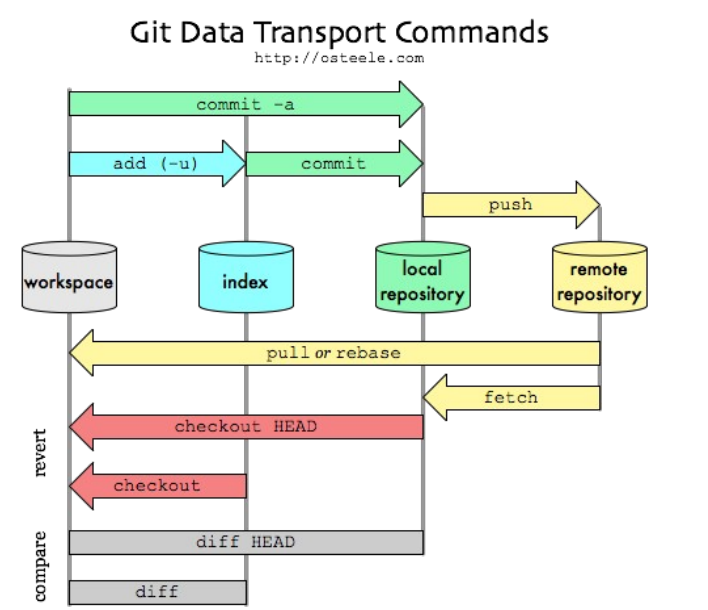
***11/03/2022: Git e SVN vs Git***

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.

SCRUM è intuitivo, leggero, difficile da padroneggiare.

Immagine che contiene testo

Descrizione generata automaticamente

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.

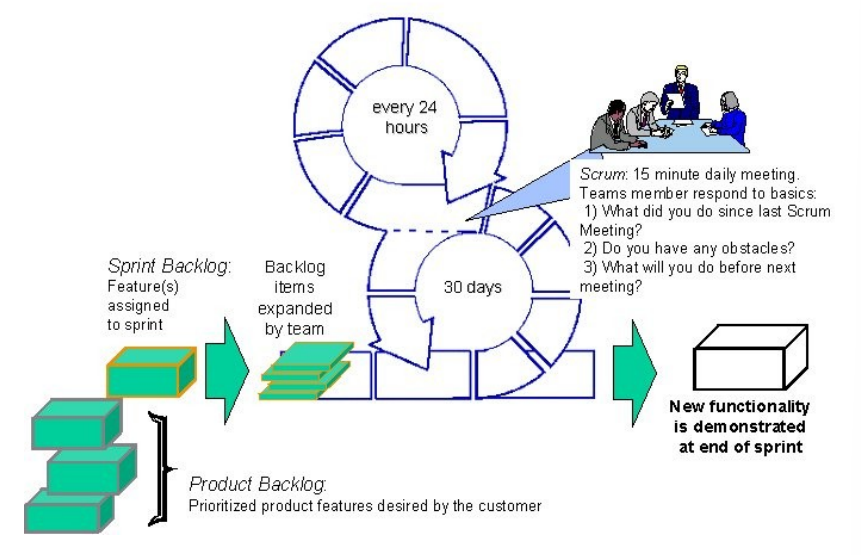
La loro durata è fissa e abbastanza breve, prevedendo un certo rischio che siamo disposti ad avere.

***21/03/2022: SCRUM: disamina completa***

Il focus è quindi di concentrarsi sulle piccole funzionalità in ogni sprint, ricevendo anche i feedback dell’utente e adattando sulla base di esso i propri requisiti. Tutti gli stakeholders decidono quindi se rilasciare il progetto così com’è o se implementare in successivi sprint.

Per adottare questo framework è disponibile una documentazione, gestendo le singole pratiche e facilitando l’adozione dei compiti, bilanciando le parti.

Di fatto, citando rapidamente i principi dell’altra volta, tutto viene gestito con un linguaggio semplice, vedendo facilmente quando le attività vengono considerate completate (*trasparenza*), ispezioni per capire se le attività siano state implementate o meno sulla base della pianificazione fatta *(controllo)*, aggiustamenti per migliorare/minimizzare deviazione tramite feedback continuo (*adattamento*).



Il *product backlog* rappresenta l’insieme delle funzionalità che dovranno essere implementate per un prodotto, stimandole e prendendone un sottoinsieme utile (*sprint backlog*), confidando di riuscire ad implementarlo entro lo stesso sprint e possibilmente completandolo entro le 2-4 settimane.

Ogni giorno si ha un evento (*daily meeting*), dove il development team discute di quello che ha fatto il giorno prima e cosa intende fare nel giorno stesso; normalmente hanno una durata di 15 minuti. Tutti possono vedere/monitorare le attività all’interno di uno sprint.

In base alla Definition of Done, si capirà se le attività corrispondono alle esigenze espresse in precedenza, definite con formalismi prima della creazione di attività.

I gruppi si autoorganizzano, cercando di dare il maggior valore di business ad una serie di requisiti implementati in maniera incrementale. Il maggior rischio è la durata dello sprint, perché si cerca di implementare le features entro la fine dello sprint.

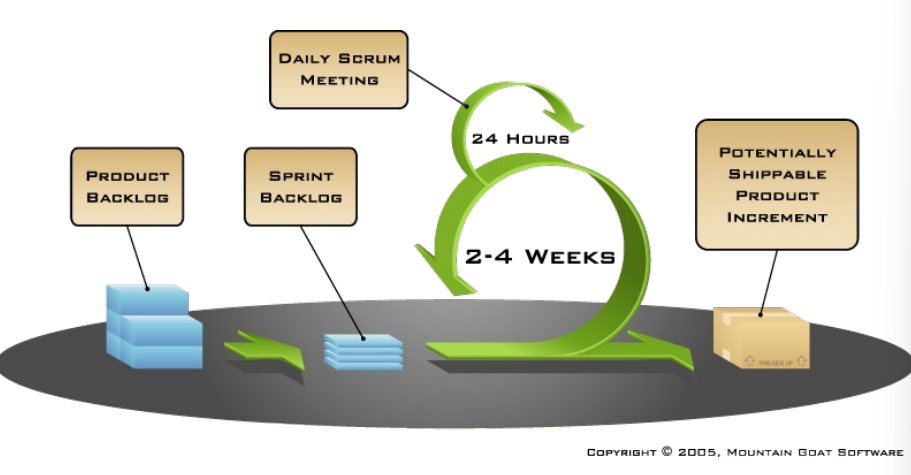
Tutti questi principi in linea teorica seguono l’Agile Manifesto, con questi principi:

Immagine che contiene testo

Descrizione generata automaticamente

Nel framework Agile è meno strutturata rispetto ai classici processi essenziali, formalizzando e rispettandoil contratto, mettendo tutto nero su bianco ed evitando eventuali contenziosi.

Descriviamo graficamente lo sprint e come si compone:



Una durata costante dello sprint favorisce un ritmo migliore, dove il prodotto è progettato e testato continuativamente attraverso una serie di sprint. Alla fine di questi vengono celebrati tutti gli eventi. Quindi piuttosto che fare tutto insieme, si agisce singolarmente.

Una volta stabilita la durata, si cerca di capire quanto possiamo mantenere i cambiamenti all’esterno di un singolo sprint. Lo sprint backlog può quindi essere chiarito/rinegoziato tra Product Owner e team di sviluppo, arrivando poi con più dettagli possibili e maggiore chiarezza.

Se non si riuscisse a completare l’attività entro la fine di quello sprint, vengono spostate al successivo.

Uno sprint può essere cancellato se lo Sprint Goal diventa obsoleto, con durata limitata; questo raramente ha senso.

Parliamo ora di *ruoli*:

Immagine che contiene testo

Descrizione generata automaticamente

*Product owner*: Definisce le caratteristiche del prodotto, rappresentando il desiderio del committente (cioè il cliente), decidendo date e contenuto del rilascio e capendo la redditività del prodotto (ROI/Return of Investments, rapporto tra risultato operativo e capitale investito). Adegua poi le caratteristiche per ogni iterazione, secondo quanto necessario, accettando o rifiutando i risultati del lavoro.

*Scrum Master*: Conduce il progetto e fa ragionare il product owner in merito a come le attività funzionano, adottando il tutto a livello di gestione dei valori e delle pratiche Scrum. Rimuove gli ostacoli, assicurandosi poi che il gruppo di lavoro sia pienamente operativo e produttivo. Previene il working team da interferenze esterne, coadiuvando il lavoro tra Product Owner e Team di sviluppo (ruolo di *servant leader*).

*Development Team*: Generalmente composti da 5-9 persone, responsabili dell’incremento in conformità alla Definition of Done, lista di obiettivi. Naturalmente questo racchiude una serie di competenze trasversali (programmatori, tester, progettisti di user experience, ecc.). Tutti lavorano full-time, ma possono esserci eccezioni (es. amministratori di database). Generalmente si auto-organizza.

Definiamo poi gli *eventi*:

Immagine che contiene testo

Descrizione generata automaticamente

*Sprint planning*: La stima delle attività viene fatta dal development team, in un ambito time boxed (8 ore per Sprint di 1 mese), selezionando dal product backlog gli item che può impegnarsi a completare. Viene creato quindi lo Sprint backlog in maniera collaborativa per tutto il team, definendo i tasks ed eseguendo una stima per ciascuno. Vengono poi decomposte le User Story (cioè le specifiche da condividere con il team di sviluppo capendo cosa sviluppare), tipo in questo modo:

Immagine che contiene testo

Descrizione generata automaticamente

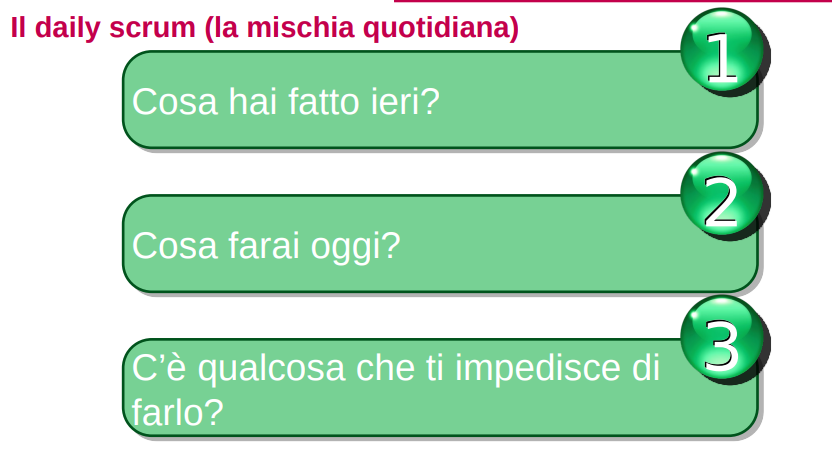
Definiamo poi il meeting dello sprint, eseguendo il planning come segue:

Immagine che contiene testo, segnale

Descrizione generata automaticamente

La stima dello sprint backlog si stima non in ore ma in storypoint, capendo le singole attività in base alla “pesantezza” (level of effort), presupponendo siano ognuna scomposte in sottoattività.

*Daily scrum meeting (stand up meeting)*: Incontro giornaliero di 15 minuti circa fatto in piedi, perché si pensa venga fatto in fretta (prima della pausa caffè/pausa pranzo). Si cerca di sincronizzarsi capendo lo stato delle attività di tutti, aggiornando la scrumboard. Ad esso può anche partecipare il product owner. Non è un SAL, ma è un impegno assunto tra pari.



*Sprint review:* Rappresenta un recap e una organizzazione di gruppo su quanto fatto fino a quel momento. È time boxed, 4 ore per sprint di 1 mese, realizzato senza slide, con la regola delle 2 ore per preparazione. Viene validato/accettato quanto realizzato durante lo sprint, con un’idea informale di partecipazione. Generalmente coinvolge tutto il gruppo e anche esterni.

*Sprint retrospective*: Fatta dopo la Sprint review e prima del prossimo Sprint Planning, anche qui Time boxed (3 ore per Sprint di 1 mese), valutando ciò che funziona e ciò che non funziona. Si cerca quindi di capire se il progetto qualitativamente è convincente, definendo le attività possibilmente migliorabili dal team. A questo partecipa tutto il gruppo di lavoro.

Esistono vari template per poter organizzare le caratteristiche volute, possibili miglioramenti, ecc.

Un modo molto semplice di vedere la cosa:

Immagine che contiene testo

Descrizione generata automaticamente

Parliamo poi di *artefatti*:

Immagine che contiene testo

Descrizione generata automaticamente

*Product backlog:* L’insieme di requisiti e funzionalità, miglioramenti, fix composta dal Product Owner con l’aiuto dello Scrum Master, rivalutando frequentemente le attività con il Development Team. Viene raffinato in maniera dinamica e continuativa. Un esempio di definizione di attività:

Immagine che contiene testo

Descrizione generata automaticamente

Qui invece descriviamo le caratteristiche delle User Stories (comunque scomposte da parte del development team sulla base delle esigenze lavorative):

Immagine che contiene testo

Descrizione generata automaticamente

Si cerca poi di dare una breve indicazione dell’obiettivo principale dello sprint, dando tutte le funzionalità all’utente (user stories mica per niente, come si vede quindi) entro la fine di quello specifico *sprint goal*.

*Sprint backlog*: Ogni componente del Development Team sceglie cosa fare e la stima delle attività viene gestita durante lo stesso giorno, dato che ogni membro può aggiungere/cancellare/modificare parti dello sprint backlog, facendo “emergere” i lavori da svolgere durante gli sprint. Il lavoro deve essere chiaro, decomponendolo mano a mano che il progetto avanza. Tutto deve essere massimamente visibile a tutti attraverso la scrumboard. Un esempio:

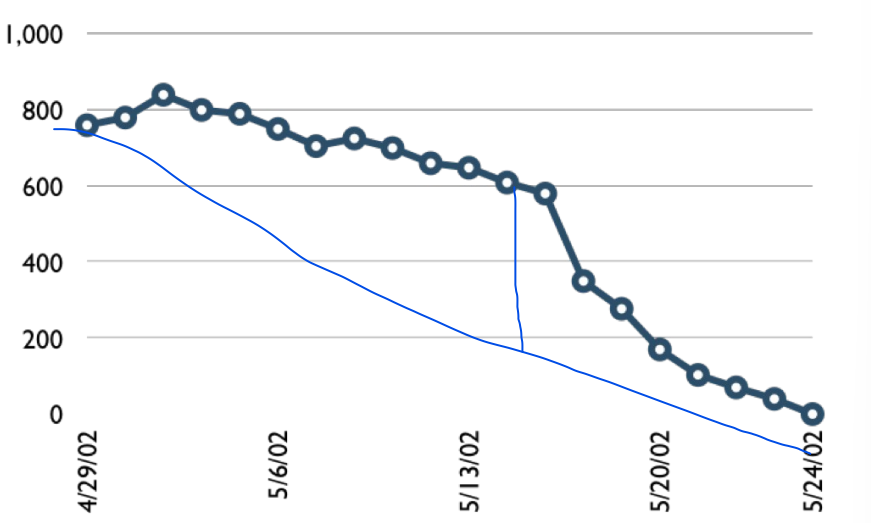
Immagine che contiene tavolo

Descrizione generata automaticamente

*Definition of Done*: Definisce il significato di completamento (”Done”) per ogni Sprint Item, definendo il minimo set di attività che definisce l’attività completata. Varia per ogni gruppo di lavoro e deve essere ben chiaro per ogni membro del gruppo di lavoro.

*Acceptance criteria*: Definisce se una storia sia stata completata come voluto, con frasi semplici condivise tra Product Owner e Development Team. Possono essere incluse con la User Story. Rimuovono l’ambiguità dei requisiti. Un esempio di grafico che capisce le attività (vedendo poi lo scostamento delle attività dall’andamento ipotizzato, come si vede disegnato brutalmente da me).

Esso viene definito come *sprint burndown chart*:



***25/03/2022: Presentazione primo assignment e Build Automation***

Si parla per l’appunto di *build automation*, quindi rendere automatica la compilazione del software, compilando codice sorgente e impacchettandolo in codice binario, usando test automatici.

Storicamente si esegue tramite dei makefile, oggi tuttavia vi sono due categorie:

* *build-automation utility*, con lo scopo di creare un artefatto (frammento comprendente codice+ambiente di sviluppo ed indicazione) di un progetto;
* *build-automation servers,* eseguendo build-automation utilities, schedulate in maniera automatizzata.

Ci focalizziamo su quelli utility; solitamente gli errori si hanno nelle dipendenze e compilazione dei programmi. Nello sviluppo di progetti, quindi, il momento più tosto era proprio l’integrazione, non sapendo se le singole parti sviluppate potessero funzionare, diminuendo il rischio di scaricare le responsabilità su una singola persona. Le esigenze sono, chiaramente, la collaborazione ad un progetto, tramite compilazione, utilizzo e modifica di bug presenti nelle dipendenze, generando e provando gli artefatti.

Ci sono vari tipi di tool, per esempio di *scripting*, tramite utilizzo di apposito linguaggio o automatismo (tramite file script, makefile, Gradle), oppure *artifact oriented*, configurando correttamente il process di build (Apache Maven, che vedremo e NPM). Il *processo di build* si compone quindi di un insieme di passi che trasformano gli script, codice, file di configurazione, documentazione e test su un prodotto.

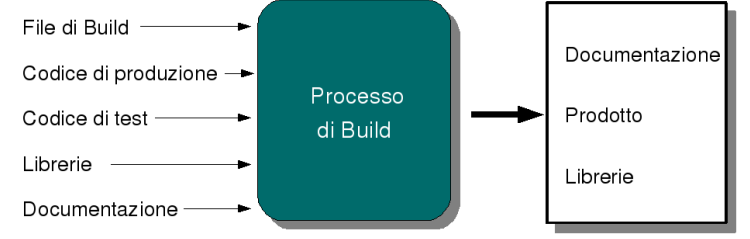


Immagine che contiene testo

Descrizione generata automaticamenteSempre dal libro *Pragramatic Programmer*, deriva la descrizione delle caratteristiche Agile come *CRISP* quindi:

* *completo*, indipendente dalle fonti di build
* *ripetibile*, per cui un’esecuzione ripetuta dà sempre lo stesso risultato, accedendo ai file contenuti nel sistema di gestione del codice
* *informativo*, descrivendo lo stato del prodotto
* *schedulabile*, programmato ad una certa ora ed eseguibile automaticamente
* *portabile*, indipendente il più possibile dall’ambiente di esecuzione

Da questo vi sono i repo degli artefatti (*artifact repository*), mantenendo dati e documentazioni.

A tale scopo approfondiamo proprio *Maven*, gestendo la build di un progetto, creazione di reportistica e documentazione, con una configurazione minima e il programmatore che usa il framework configurerà solo le peculiarità del progetto, descrivendo ciò che si differenzia dalle implementazioni standard.

Alcune delle principali caratteristiche di Maven sono:

* *build tool,* definendo delle lifecycle di build, configurando ed eseguendo il build process;
* *dependency management*, dipendenze di progetto con file di configurazione (esempio, JUNIT con sue dipendenze inserite nel CLASSPATH, caso Java);
* *remote repositories*, definendo repo remote con gran parte delle librerie e plugin usati da Maven per implementare ed estendere la lifecycle di build;
* *Universal Reuse of Build Logic*, con plugin usati per tutte le parti di processo, esecuzione di framework, ecc. Tutti questi plugin sono documentati e configurabili, filtrando per keyword progetti preesistenti, usati come plugin (definiti come mojo)

Maven permette anche la gestione degli *archetipi*, quindi permettendo la creazione di un progetto avendo già un template utilizzabile partendo da processi preesistenti e successivo comando di test (spostandosi nella cartella di progetto).

Esempio con Maven: *mvn archetype: generate (…) mvn test*

Esso è basato sul concetto centrale di build lfecycle, quindi costruendo e distribuendo un particolare artefatto chiaramente definito. Per ogni persona che costruisce il progetto è necessario solo un piccolo set di comandi per poter utilizzare Maven, per qualsiasi tipo di progetto.

Vi sono tre tipi di build lifecycles:

* *default*, consegna del progetto
* *clean*, pulizia del progetto
* *site lifecycle*, creando il sito di documentazione del progetto

Per il default si strutturano le seguenti fasi:

Immagine che contiene testo

Descrizione generata automaticamente

L’idea quindi è la definizione delle fasi appena descritte, implementandole singolarmente con l’utilizzo di mojo. Si ha poi il *POM/Project Object Model*, unità fondamentale di lavoro, con un file XML che contiene informazioni sul progetto e sulle sue configurazioni.

Quando si esegue un obiettivo o task, Maven cerca il POM, lo legge e ne prende la configurazione, come si vede dall’immagina (da cui poi si esegue il comando *mvn install*):

Immagine che contiene testo

Descrizione generata automaticamente

Un toolkit utile di Maven è *Artifact*, definito come modello o pattern originale, quindi fornendo una base per progetti, generando versioni parametrizzate. Di fatto Gradle si pone come strumento migliore per la gestione della build lifecycle, riprendendo a piene mani da Maven.

Di fatto quindi è un framework collezione di una serie di plugin, eseguendo azioni sul codice e implementando le azioni sotto forma di Mojo.

***28/03/2022: Software Testing***

Il *software testing* è uno strumento di investigazione del prodotto, creando informazione sulla qualità del prodotto/progetto, fornendo una vista indipendente sul software e permettendo al business di apprezzare e comprendere i rischi e discuterne correttamente con gli stakeholders. Tutto ciò deve rendere un prodotto ed un software adatto all’uso. Le linee guida/terminologie e casi di test in maniera corretta/oggettiva viene fornita dalla ISTQB.

Definiamo il processo di *testing*, sia statico che dinamico, concerne la preaparazione, pianificazione e valutazione dei prodotti software, determinando la soddisfazione di determinati requisiti. Ciò parte anche dalla stessa progettazione del manuale utente. L’approccio scettico nei confronti dei test è il non fidarsi se, a seguito delle esecuzioni di test, non si trovino bug o problematiche.

Lo sviluppo software è un processo umanistico, proveniente dall’esperienza reale, parte tutto dal nostro cervello e stato d’animo, portando ed errori.

I bug possono esistere nel codice, ma magari le condizioni che verificano l’errore non si sono mai realizzate. Esempio famoso: Millennium Bug, quindi il campo dell’anno conteneva solo 2 caratteri (70/80/90…). All’avvento degli anni 2000, si poteva avere confusione tra gli anni del secolo prima e del secolo dopo.

La buona scrittura di test è la rappresentazione di tutte le situazioni di errore/sviluppo del programma. Di fatto abbiamo:

* il programmatore, (45% di possibilità inserimento difetti) che fa un *mistake* durante lo sviluppo ed esso genera un comportamento inatteso e, quando eseguito, si ha un comportamento inatteso nel programma, dando una *failure*;
* l’analista, (20% analisi requisiti, 25% progettazione) che ascolta il cliente e ne interpreta le volontà, eventualmente immettendo difetti se non hanno una corretta comprensione di quanto richiesto.

Naturalmente tutto ciò può pesantemente influenzare la qualità del prodotto.

Si devono prevedere anche condizioni ambientali nella creazione del prodotto (ad es. un app che sfrutta anche la funzionalità GPS; naturalmente mi aspetto che funzioni anche senza aver attivato questa specifica funzionalità).

Abbiamo i categorie di testing:

* *funzionale*, funzionalità richieste come esigenze reali dall’utente finale, presenti esplicitamente;
* *non funzionale,* cioè non funzionalità ma requisiti pratici (usabilità, facilità di utilizzo, sicurezza, accessibilità). La stessa legge italiana prevede l’erogazione di servizi e progettazione degli stessi si considerino tutte le possibili categorie di utenti (non vedenti, daltonici, ecc.);
* *statico,* che si occupa di test senza esecuzione di codice (dovuti anche in merito alla complessità del codice sorgente scritto, in maniera tale che implementando modifiche il progetto sia solido e stabile);
* *dinamico,* codice in esecuzione da qualche parte;
* *verifica,* un software fatto bene, fatto a regola d’arte, progettando un sistema secondo i canoni giusti e vincoli di condizionamento e si sa che funziona correttamente;
* *validazione,* rispettando le specifiche dell’utente e i requisiti stabiliti.

Definiamo una serie di casi di test, descritti da IEEE/IQSTB

* *caso di test*, una serie di valori di input, precondizioni, risultati attesi (es. caso d’uso in un sito, scaricando del materiale; precondizione, essere registrati ed avere un profilo), postcondizioni, sviluppate per un particolare obiettivo con uno specifico requisito. Devono essere eseguite più prove per poterne scrivere di buoni e migliorano l’ideazione degli stessi requisiti, risolvendo le ambiguità;
* *condizione di test*, un aspetto testabile o un oggetto verificato da un sistema con specifici casi di test.

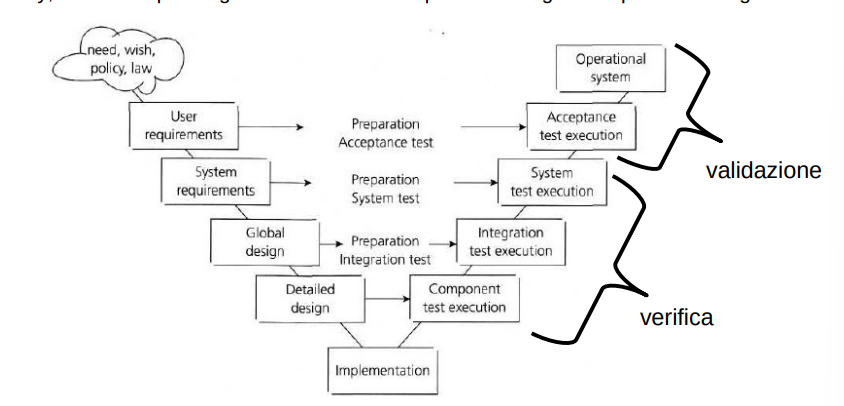
I requisiti sono espressi nel linguaggio naturale, dunque foriero di ambiguità, parlando infatti di *testable requirements*, progettando un certo requisito secondo le catene CI/CD (continuous integration/contnuous development). Se i casi di test sono scritti bene, posso essere automatizzati, quindi usati da una macchina. Essi devono essere spiegati in maniera comprensibile sia a chi lo progetta sia a chi lo riceve.

Diciamo inoltre che i test, oltre ad essere pianificati, controllato e disegnati (*test planning, test control, test analysis*), segue l’implementazione, l’esecuzione, controllo dei risultati (checking result), nonché valutazione dei criteri di uscita (*evaluating exit criteria*, risoluzione dei bug più gravi, eventualmente reggendo i meno gravi) e chisuura di test (*test closure*), con una documentazione (*test results reporting*) solida e che certifica la qualità del prodotto.

SI ha quindi una serie di principi (sette, cosiddetti *seven testing principles*):

1. *testing show presence of defects,* Il test evidenza e dimostra la presenza di difetti, non ne evidenza l’assenza;
2. *exhaustive testing is impossibile, t*estare tutto è impossibile, a meno di avere un’applicazione con input limitato e struttura logica molto semplice. Per questo motivo è importante valutare il rischio di malfunzionamento, capendo cosa serve testare. Ci si può concentrare sui casi più importanti da risolvere (*risk based testing*), oppure partendo dagli stessi requisiti (*requirement based*);
3. *early testing*, avviando la fase di test il prima possibile, avviato parallelamente al processo di sviluppo stesso
4. *defect clustering,* capendo che la maggior parte degli errori si estende ad un numero limitato di moduli; statisticamente circa il 20% di questi contiene circa l’80% degli errori;
5. *pesticide paradox*, quindi adattando i casi di test allo sviluppo software altrimenti disponiamo di test inefficaci e in grado di risolvere problemi vecchi ma non attuali;
6. *testing is context dependent*, ovviamente il testing viene fatto in base alle specifiche richieste ed applicazioni del software (ad esempio i software medici hanno test basati sul rischio, diversi da un’applicazione/sito online popolare in cui si richiedono rigorosi test prestazionali;
7. *absence of error fallacy*, cioè l’idea che se non si riscontrano difetti nel software non significa che sia perfetto, anzi, al contrario. Si deve capire se tutto corrisponde alle esigenze del cliente.

Il test è dipendente dal contesto (ad es. dispositivi medici richiede test basati sul rischio, sito web popolare richiede rigorosi test di prestazione, ecc.). Si può anche commettere l’errore di pensare il sistema non abbia errori (*absence of errors fallacy*). Si ha poi la correlazione tra fasi di sviluppo e fasi di test ed esecuzione dgli stessi (*v-model*), come ad esempio:



I singoli pezzi di un codice devono colloquiare nel modo giusto, con l’*integration testing*, con dei subsystem, verificati dalle unità di test (interni), oppure tramite DB/file system (esterni). Sto quindi esaminando il sistema più in alto, verificando il comportamento di tutto il sistema e verificando tutte le specifiche tecniche (*system testing*). Un utile test è lo *smoke testing/sanity checking*, verificando che il sistema funzioni almeno per quanto concerne le funzioni base.

L’*acceptance testing*/*user acceptance testing (UAT)*, ricevendo determinati feedback dell’utente che portano alla chiusura di un progetto o all’ulteriore prosecuzione.