Contents

1 Modulo I - Machine Learning for Software Engineering

2 Ripasso: introduzione a SVN e Github

Un sistema di controllo delle versioni tiene traccia dei cambiamenti ad un file ed inoltre permette di tornare indietro nelle versioni. Ci sono due tipologie:

- Centralizzata: SVN, utenti condividono una repository che è su un solo server centralizzato
- Distribuito: Git, ognuno ha una copia della repo.

Differenze con classici sistemi di storage cloud sono varie:

- SVN e Git sincronizzano solo se c'è richiesta, mentre per sistemi cloud avviene in automatico
- I merge sono a grana fine per SVN e Git, per Dropbox/Drive è a grana più spessa.
- La storia delle versioni è mantenuta da SVN e Git, mentre poterebbe non esserlo per Dropbox/Drive

Working copy: versione su cui è possibile lavorare in locale. Non si lavora mai sulla risorsa condivisa, bensì su quella locale.
Revisione: particolare stato della risorsa condivisa, su cui è possibile tornare indietro. La versione è spesso una revisione particolare, che può essere offerta agli utenti (es versione 1.0 può corrispondere alla revisione 150).

2.1 Issue tracking systems

Sistema che permette di creare, assegnare e tenere traccia dei problemi (issues). Tutto nasce da Bugzilla (per progetti open source), un bug in un codice è molto simile a descrivere un requisito: nasce il concetto di ticket. Un ticket è un informazione rilevante al progetto, come un requisito da implementare o un bug.

Molti sistemi sono gestiti attraverso i ticket (medie-piccole dimensioni), ogni ticket ha un workflow: creato, assegnato, sviluppato, testato, approvato, chiuso. Tra gli esempi di sistemi per issue tracking ci sono Jira, Github, Redmine.

3 Tecniche di Machine Learning ed analisi software a supporto della quality assurance e del testing

I bug software costano circa 2.84$ dollari ogni anno. Il codice viene scritto in diversi linguaggi, da tantissime persone, per poter fixare bug, aggiungere nuove feature e migliorare la qualità del codice. Il software è rilasciato con grande velocità, si vuole prevenire di avere bug in modo da avere technical debt basso.
Failure: comportamento osservato dall'utente e che non corrisponde alle specifiche del sistema. Un difetto software è quella parte di codice che può dare luogo ad una failure, questo non avviene sempre, ma solo sotto determinate condizioni. Per evitare le failure si cerca di individuare i bug prima che questi possano essere eseguiti dagli utenti.
È importante capire, avendo tempo limitato, come poter priorizzare le risorse di analisi. Si parla di software analytics come analisi di dati che riguardano progetti software.
Un aspetto importante è ML per poter predire ed evitare i bug futuri.
In generale:

- Misuro i dati
- Analizzo i dati e creo il modello di ML
- Identifico quale classe/metodo è buggy

I dati sono gestiti da i version control systems ed issue tracking systems.
Importanti i commenti dei commit: invece di descrivere la modifica effettuata, faccio riferimento al ticket sul sistema. Dovrebbe esserci

una relazione 1-a-1 tra ticket e commit: prendo il ticket e lo sviluppo per intero, in modo poi da fare il commit delle mie modifiche.
Se ho il tracciamento preciso tra ticket e codice che ho implementato per quel ticket, posso vedere se l'ammontare di linee di codice richieste per realizzare il ticket è maggiore o minore di quelle richieste per rimuovere il bug. Posso differenziare i ticket in base a bug e requisito e vedere il numero di righe cambiate.
Una volta estratti i dati, posso collezionare molte metriche a riguardo:

- Quante righe di codice modificate
- Chi le ha modificate

3.1 Metriche per individuare bug

Posso usare metriche per fare delle stime su quale classe è buggy:

- relative al codice: classe con molte righe è potenzialmente buggy
- processi: se file è stato affetto da molti cambiamenti
- fattori umani: se file è stato toccato da sviluppatori esperti o non.

Posso leggere tutte queste info tal ticket di JIRA, ogni commit avrà un suo identificativo.
Si cerca di andare a capire quali file di una release sono defective o non defective. Avrò quindi dei dati che darò in pasto a modelli di machine learning black-box, con cui potrò capire dato un nuovo file quanto questo sarà defective. È anche importante capire perché un file è difettoso, ci sono anche regole di normativa per cui se si usa una predizione bisogna anche fornire il perché della predizione. È possibile anche vedere il perché una classe ha avuto una certa probabilità di essere difettoso.

3.2 Process control chart

Chart che mostra la stabilità degli eventi nel tempo: è importante che sull'asse verticale ci sia l'elemento di cui si vuole controllare la stabilità, mentre sull'asse orizzontale ci deve essere qualcosa che possa essere misurato.
Voglio vedere se un certo progetto ha dei difetti per in base al numero di revisione: metto su asse x il tempo (es settimane, o le revisioni), asse y avrò i difetti. Mi chiedo se in un progetto i difetti sono lineari in base alle revisioni. Dobbiamo:

- Selezionare i dati sull'asse y ad esempio il numero di commits o requirements
- Seleziono dati per asse x, asse temporale. Può essere giorni/mesi o anche release
- Colleziono i dati
- Calcolo media e varianza, imposto poi degli assi di limiti superiori ed inferiori (ad esempio):
  - limite superiore (upper control limit): media + 3·deviazione standard
  - limite inferiore (lower control limit): media - 3·deviazione standard. Può andare sotto 0, alle volte conviene andarci ma altre volte no (quindi lo si fa fermare a 0).

Visualizzando il chart, posso capire se qualcosa è andata particolarmente male: quando un punto è al di fuori del limite, è significativamente diverso da tutti gli altri punti. Quindi in questo caso si analizza il perché per una determinata release c'erano dei valori così estremi, ad esempio perché c'è stato un numero così elevato di difetti.

3.3 Ripasso: continous integration e travis

Pratica per cui i membri del team integrano il loro lavoro continuamente. Ogni integrazione è verificata con un build del progetto,

viene fatto per evitare errori nell'integrazione del progetto.
È molto importante che il built sia automatico, in modo che chiunque possa effettuarlo, e non solo alcuni membri del team. Importante anche che la built sia self-testing, e che inoltre i commit avvengano quotidianamente o almeno per ogni feature.
I test vanno effettuati in un ambiente che sia il più possibile simile all'ambiente di production: ad esempio, se ho sviluppato un app, il production environment sarà il dispositivo dei miei utenti.

3.4 Ripasso: technical debt

Applicazione di concetti finanziari al dominio del software engineering. Code smell e regole di qualità: una regola di qualità è un principio che detta come il codice dovrebbe essere, ad esempio:

- alta densità di commenti
- bassa densità di codice
- if statement non difficile da leggere

Una violazione o code smell è una porzione di codice non perfetto. È importantissimo soffermarsi sulla differenza fra difetto e code smell: difetto può dare vita ad un failure (sotto determinate condizioni), mentre il code smell è qualcosa non osservabile dall'utente (ad esempio la lunghezza di un if statement). Il problema è che i code smell impattano gli sviluppi futuri.
Il debito tecnico può essere visto come un qualcosa che darà problemi a sviluppi futuri (impatta la prossima release), può emergere organicamente come molti sistemi aumentando di complessità, aumentano la loro complessità di gestione. Inoltre, può anche essere scelto in maniera opportunistica, ovvero decidere deliberatamente di avere una certa quantità di debito tecnico.
Il technical debt consiste in due parti:

- principio: costo per eliminare i code smells

- interesse: la penalità da pagare in futuro per il debito non eliminato

Come gestisco i code smell attraverso i ticket: tutto ciò che bisogna fare deve essere scritto in un ticket. Non c'è altro mezzo di informazione che il progetto può usare, linee guida e processi aziendali possono essere documentati su un altra piattaforma (confluence, per completare JIRA).Se c'è un code smell, **dipende dal processo aziendale**: può essercene uno che dice che chi ha introdotto lo smell lo toglie senza ticket, un altro che dice che i problemi rilevati da Sonar Cloud verranno analizzati a parte da specifiche persone che avranno il ruolo di aprire i ticket appositi. È possibile impostare i repository in modo che rigetti i commit che violano alcune regole di qualità (configurando Travis/Jenkins), così saprò che il codice è sempre smell free.

3.5 Misure di analisi

Voglio rispondere alla domanda: le linee di codice per le feature, sono state di più o di meno di quelle usate per risolvere i bug? Quando gestisco i dati, posso farlo per due motivi:

- guardare al passato: perché spendo molte LOC per risolvere i bug? Vado a fare delle analisi introspettive sul progetto
- avvento del ML, predico il futuro: prendo delle decisioni nel presente, in modo che l'impatto dei cambiamenti futuri sia controllato

4 Merging JIRA con Git

4.1 Git: comandi utili

Una volta effettuato il clone di una repository, è possibile avere accesso al log: **git log <options>**, da accesso ai cambiamenti mostrando

- autore del commit
- cambiamento
- commento

se non viene specificato il branch, git ritorna il log di quello corrente. Una cosa interessante può essere comparare due commit fra loro, per vedere i cambiamenti: tramite il comando **git diff <commit_id>..<altro commit_id>**. Alcune visualizzazioni sono human readable, ma nel codice che fa l'analisi automatica non è importante.
Data la revisione di un file, per ogni riga è possibile risalire al commit che ha inserito quella determinata riga: ho un file di 10 righe, mi chiedo chi ha inserito una riga, quando e perché. **git blame <filename>** (con -w ignora gli spazi bianchi). Fornisce la data dell'ultima modifica con la relativa modifica (ci sono anche meccanismi per andare indietro) e chi l'ha effettuata. È possibile fare il grep sul testo, in modo da cercare solo specifiche parti di interesse del log.

4.2 Cercare ticket in JIRA

Una volta trovato un progetto, è possibile selezionare il tipo di ticket da cercare, lo sviluppatore etc... (numerosi filtri), in advanced è anche possibile creare delle query SQL-like, con cui è possibile creare delle ricerche sofisticate e metterle da parte.

5 Snoring and proportion

Quando vedo un commit per un ticket di tipo bug, che ha toccato 3 classi, so che quelle 3 classi prima del commit erano buggate prima del commit. Con il blame so qual è il commit preciso che ha prodotto un determinato stato di una riga. In generale, la predizione dei difetti cerca di individuare quale artefatto software è più propenso a mostrare difetti: ci mettiamo nei panni di chi deve testare la release finale (molte classi) per capire quale classe è opportuno testare. Vorremmo una stima di quanto una classe è probabilmente buggy, per

ordinarle in modo crescente (per probabilità).
Prima di poter fare delle stime, è necessario misurare cosa stimare: quando si fa una predizione: un conto è farla binaria, un altro è se vanno predetti dei numeri. Quest'ultima è estremamente più complessa di quella binaria.
esempio: supponiamo di dover predirre il meteo, a seconda del fatto che io dica che piove o no, o che piove 20 cl. Non ho alcune informazioni, ad esempio se piove più o meno di 20 cl o in media 20 cl. Potrei avere predizioni diverse per giorni diversi. Bisogna fornire **un grado di confidenza** (e quindi intervallo di confidenza), in modo da rendere la previsione più comprensibile. L'approccio del Machine Learining consiste nell'utilizzo dello storico dei dati: ad esempio per il meteo posso avere delle misure storiche che mi aiutano a stimare le previsioni per un determinato giorno, in base a misure passate che hanno avuto lo stesso risultato conseguente. Quindi per predirre il comportamento di una classe in futuro, devo poter conoscere il suo comportamento passato e quindi le misure passate.
Il progetto è un insieme di classi, ho varie revisioni e varie release (o versioni): alcune classi evolvono durante il tempo, che è diviso in versioni, ed ogni tot revisioni (che corrisponde ad un commit) ho delle release.

5.1 Difetti dormienti

Per definire un classe difettosa, questa deve contenere un bug. Il problema è che per capire se una classe è o meno difettosa bisogna avere come input i bug: in base a questo, troviamo il commit che ha eliminato il bug, che tocca delle classi che quindi erano bug. È possibile che un difetto sia scoperto ed eliminato diverse release dopo la sua introduzione: è stato introdotto nella release 1.0 ed eliminato nella 3.2, ma quando ero in release 3 non avendo fatto il fix del bug non ne conoscevo l'esistenza e non conoscevo le classi che lo contenevano.

L'esistenza di un difetto non può essere nota prima che questo sia eliminato, abbiamo diverse versioni.

le affected version (AV) sono quelle affette dal bug, la AV è definita come la injected version (IV) inclusa e la fixed version (FV) esclusa, ho anche la open version (OV), ovvero quando il ticket è stato aperto. Il problema è capire qual è la IV, ovvero quando il bug è stato introdotto.
Se ho creato il ticket in un certo intervallo temporale, non sapevo quale classe aveva bug. Abbiamo un commit, che ha come commento il fix di un bug relativo ad un ticket e possiamo scoprire quale riga aveva un bug; inoltre, sappiamo che la versione successiva a questa non abbiamo più il bug.
Ci accorgiamo di due problemi:

1. se non avessimo cambiato la riga di codice, non avremmo mai scoperto che la classe aveva un bug e quindi fino a quel periodo temporale ci sembrava che la classe fosse bug free.

2. supponiamo che la versione successiva non sia buggy, il problema è sapere quando questo bug è stato iniettato: sappiamo che la classe aveva un bug e adesso non lo ha più, ma non so per quante revisioni/versioni passate la classe aveva il bug

per poter fare delle stime occorre conoscere tutte le revisioni in cui la classe era buggy, dobbiamo poter andare indietro nel tempo per poter etichettare la classe come buggy finché serve. Quindi, usando il **blame**, posso scoprire quale commit ha introdotto il bug e posso anche scoprire in quale versione ricade il commit ed in quale versione viene risolto. Per ogni commit ho una data: se so che le versioni sono ordinate temporalmente (ad esempio se so che ogni tot viene rilasciata una versione) posso sapere, in base alla data del commit, a quale versione appartiene, in quale versione era buggy e quando no

5.2 SZZ

Metodo che consiste nell'usare il blame per capire qual è la prima versione in cui il bug è stato introdotto. Andiamo indietro nelle versioni, fino a trovare quando la LOC buggy è stata creata, un altro approccio è quello di vedere le AV in Jira: quando l'utente elimina il bug, fa anche una stima di quali versioni erano affette dal bug, quindi possiamo leggere i ticket Jira, in particolare il filed, ma non è detto che ci sia. L'approccio ha comunque dei problemi:

- se per introdurre una feature cambio qualcosa che funzionava ed ora non funziona più, sono costretto a cambiare quello che funzionava, ed SZZ mi direbbe che il bug è nella classe che funzionava, ma prima di introdurre la feature il problema non c'era, ovvero introduco un regression bug[1].
- il blame va una sola volta indietro e non sappiamo quante volte occorre andare indietro (anche se è possibile settare fin quando andare indietro)
- se dei cambiamenti in realtà non aggiungono nulla (ad esempio refactoring di codice) possono essere colpiti dal blame, quando in realtà la modifica è non funzionale. Ci sono tool che vanno indietro finché non trovano una modifica funzionale, scartando quelle non funzionali

Per poter predirre il futuro, devo utilizzare i dati del passato: ci stiamo orientando per poter predirre quali classi sono buggy, deve essere possibile quali erano buggy.

5.3 Prediction dataset

La predizione di difetti cerca di identificare gli artefatti software che hanno dei difetti, lo scopo è quello di ridurre l'effort di testing e code review. L'affidabilità della predizione dipende dalla qualità dataset,

[1] facendo una modifica di una parte di codice, andiamo a rompere un'altra parte di codice, senza rendercene conto, in quanto non possiamo testarla

esistono diversi sorgenti di rumore nel dataset (in-accuratezze), ad esempio la mail classificazione dei difetti, ovvero difetti taggati in Jira che non lo sono, o anche la difficoltà di andare indietro col blame etc... È possibile predirre:

- commit
- classe
- altro

e dire se questi sono buggy. Il nostro contesto sarà predirre la buggyness di una classe, avranno delle probabilità assegnate. La stessa classe appare in diverse revisioni e diverse release: ci mettiamo nei panni di chi è all'ultima revisione e vuole testare lo stato della classe prima di fare la nuova release. Il problema di capire l'origine dei difetti è che alcuni difetti possono essere scoperti solo dopo che sono stati introdotti: esistono difetti che vengono introdotti e risolti nella stessa release, marchiamo con:

- I: difetto iniettato
- F: difetto fixato
- N: nulla

è possibile inserire più di un difetto in una release

	r_1	r_2	r_3
C_1	IF	N	F
C_2	I	N	F
C_3	II	F	F

Può accadere che sono in una release in cui non ho fatto il fix di un difetto introdotto in quella precedente, quindi non ne conosco l'esistenza. Quindi, quando parte il balme, questo mi classifica la classe come non buggy. Tutte e tre le classi della tabella, in release 2 sembrerebbero non buggy se la misurazione venisse fatta in release

II, diverso se fatta nelle altre release: quindi, cambia lo stato a seconda di quando avviene la misurazione.
Più la release è recente, più c'è il problema che alcune classi mi sembrano non buggy anche se lo sono. Quindi, le release che è più probabile che abbiano bug dormienti: se sono in release i, le ultime mi sembrano tutte non buggy, perché magari li troverò dopo mesi. C'è il problema di capire se si possono usare i dati di queste release. Il numero di bug nelle release è decrescente, in quanto c'è il problema dei bug dormienti; la predizione fa sempre riferimento alla coppia (classe, release), NON VANNO CONSIDERATI QUELLI PRECEDENTI

5.4 Snoring

Concetto importante nella misurazione dei difetti nelle classi: ho positivi e negativi, il positivo è la classe buggy. Parlo di falso positivo quando la classe mi sembra buggy e non lo è. Il problema è un falso negativo, in quanto la classe mi sembra non defective, quando in realtà lo è. Questo perché la misurazione avviene (ad esempio tramite SZZ) considerando i bug non ancora eliminati.
Una classe è snoring se la classe ha uno o più bug di tipo dormiente, una classe in realtà è snoring se sembra non buggy,ma in realtà lo è. Con riferimento alla tabella sopra, nella release 1 le prime due classi mi sembrano non defective, la classe 3 mi sembra defective, in quanto per fare la misurazione sto nella release successiva, quindi avendo un fix la release precedente mi sembra defective. (vedi tabelle slides)
Domande di ricerca che provano a capire:

- per quanto tempo le classi dormono
- per quanto tempo le classi sono snoring
- lo snoring impatta la misura delle stime fatte?

Vogliamo vedere quante release passano in media dall'injection al fix: per ogni bug, per quante release questo è dormiente.

la linea centrale rappresenta la mediana, quindi possiamo dire che la maggior parte dei difetti in *Connectors* ha dormito per almeno 25 release.
qui abbiamo la percentuale (delle release di sleep). Quindi, se non volessimo il rumore nei dati, dovremmo scartare questa percentuale di release (altrimenti me perderei quei bug, che sono dormienti).
Se io misuro la buggyness di una classe, il fatto che alcuni bug dormano non è un problema, l'importante è trovarne almeno uno per far si che la misurazione sia corretta.

5.4.1 Misurare lo snoring

Supponiamo di avere un tot di release, considerarle tutte farà si di avere un certo livello di snoring, soprattutto nelle ultime: meno ne considero, meno snoring avrò ma serve avere un compromesso rispetto alla taglia del dataset, quindi non è possibile dare massima priorità all'accuratezza del dataset.
Quindi, dobbiamo misurare come cambia l'accuratezza in base alle release tolte: per capire come l'in-accuratezza varia al variare del tempo, il *missing rate* è la percentuale di classi che sono falsi negativi diviso tutte le classi positive, ovvero $\frac{FN}{FN+TP}$, la percentuale di classi positive che perdo nella misurazione.
Per capire se varia nel tempo, devo vedere quando un set di bug vengono trovati nel tempo: se prendessi le prime release, avrei un certo grado di inaffidabilità rispetto a quelle ancora prima, devo vedere quando sono stati trovati i primi bug, quindi mi soffermo sul primo 5% del dataset. Misuro cosa accade al primo 5%, misurato con diversi punti di osservazione, che variano nel tempo (misurato col numero di release), ovvero come cambia lo stato delle classi se le osservassi in futuro.
Quindi, in base al missing rate, so quante release devo scartare: ad esempio, per la maggioranza delle release, il missing rate è più del 50% a meno che non venga tolto il 20% delle release.

5.4.2 Impatto dello snoring sull'accuratezza della perdizione

Ci son due concetti importanti in Machine Learing:

- training set: dati con cui viene costruito il modello
- testing set: dati che vengono usati per misurare la correttezza del modello

una parte dei dati non verranno usati (unused data). L'accuratezza della stima, sullo stesso modello migliora considerando le release sempre più avanti nel futuro, ma non è possibile aspettare troppo. Lo step successivo è chiedersi, dato che si sa che lo snoring cresce al crescere delle release, se il modello non prende in considerazione le ultime release migliora la qualità della perdizione, ovvero avere meno dati è meglio che avere dati con snoring: togliere l'ultima release da dei dati talmente snoring che è conveniente non usarla, ma dipende dal tipo di modello che si usa: dipendentemente dal fatto che si vuole aumentare la precisione di uno o più modelli, occorre togliere più o meno release. Esistono poi classificatori che sono più o meno robusti allo snoring.
Ultime considerazioni

- togliere una release è sempre meglio che non toglierne per l'accuratezza
- rimuovere 3-4 release è sempre peggio che toglierne una

rimane come questione se toglierne una o due.

6 Modulo II - Software Testing

7 Introduzione e concetti generali per software testing

La progettazione di un sistema software è complessa: in primo luogo per l'interazione fra le persone che hanno un interesse nello sviluppo del sistema (ingegneri, programmatori, analisti, committenti etc...).
È importante cercare di avere degli strumenti che guidino nel processo di sviluppo del software in modo da poter definire delle garanzie che tutti i membri del team siano sulla strada giusta.
È necessario stabilire dei checkpoint per poter dire che il processo di progettazione e sviluppo sta procedendo nel modo corretto: Boehm definisce e distingue i due concetti di verifica e validazione:

- le attività di verifica servono per poter verificare che stiamo facendo le cose nel modo giusto, cerchiamo di rispondere alla domanda: stiamo costruendo bene il prodotto? Dove per bene si intende in maniera conforme alle sue specifiche e con un certo livello di qualità
- le attività di validazione servono per rispondere alla domanda: stiamo costruendo il prodotto giusto? In questo caso, l'enfasi è sul fatto che il prodotto sia o meno conforme a quanto richiesto dal committente, quindi si strutturano una serie di attività con riferimento alle richieste di quest'ultimo.

I due obiettivi ci impongono due modi diversi di relazione con il prodotto software, potrebbero anche entrare in conflitto fra loro ed in tal caso occorrerà risolvere o quanto meno mitigare tale conflitto.
In un contesto più generale, l'obiettivo è rassicurare l'utente che il sistema è adatto allo scopo per cui è stato progettato.
Chiamiamo le operazioni di verifica e validazione più brevemente $V\&V$: con tali attività, vogliamo tranquillizzare gli attori coinvolti nel processo di design e sviluppo del software, mostrando che ciò che viene fatto è conforme alle specifiche.

Le tecniche di $V\&V$ tendono a dare un'indicazione di confidenza sulla bontà del prodotto software.
In generale, il livello di fiducia è un concetto non assoluto, in quanto dipende da vari fattori:

- è funzione del software nell'organizzazione
- è funzione delle attese dell'utente
- è funzione delle politiche di mercato
- etc...

Il peso attribuito alle operazioni di verifica o di validazione dipende dal contesto e dalle fasi in cui ci si trova, il punto è capire in base a cosa frasi guidare per arrivare al livello di qualità del software desiderato, a quel punto si sceglie se porre l'enfasi sulla verifica o sulla validazione.

Definizione: sia S una specifica, ovvero una funzione che prende elementi da un certo dominio D e li mappa su un certo codominio C. Un programma P è una particolare implementazione di S, che lega gli elementi di D e di C. Il programma P è corretto $\leftrightarrow \forall d \in D, P(d) = S(d)$.
Se S è una specifica data ad uno dei partecipanti del progetto per essere realizzata, andare a vedere se P è corretto è un'operazione di verifica, mentre se S si basa sule necessità dell'utente, allora è una validazione.
Affermare che P sia corretto equivale a dire che P soddisfa le specifiche, ma per dire che è corretto bisognerebbe in pratica andare a testare tutti i valori $d \in D$ e mostrare che $P(d) = S(d)$ e tale operazione può essere infinita (se ad esempio $D = \mathbb{R}$.
In generale, questo problema è difficile, gli approcci possono essere due: intanto si distingue la specifica fra una specifica di progetto ed una utente e poi come risolvere l'uguaglianza $\forall d$

- 1° metodo: verifica formale. Si dimostra formalmente, con la

logica-matematica, che S effettivamente rispetta la specifica per ogni possibile input.

- 2° metodo: software testing e debugging. Si cerca di scoprire se P nasconde delle imperfezioni.

Ci focalizzeremo sul 2° metodo (anche perché chi cazzo è capace a fare una verifica formale, figuriamoci dimostrare formalmente che sia valida); il 1° metodo ha innumerevoli benefici, quindi quando è possibile bisogna validare formalmente.
Nel 2° approccio, si ammette di avere una discrepanza fra P ed S, la si accetta e si cerca di trovare i punti in cui sussiste tale discrepanza, andando a cercare le porzioni di dominio critiche per trovare tali punti.

7.1 Cos'è il software testing

Per raggiungere una adeguata confidenza sulla presenza o meno di mismatch fra P ed S si può:

- pianificare una strategia che faccia vedere potenziali differenze fra P ed S. Questo è ciò che si fa col software testing (ci focalizzeremo su questo)
- cercare di capire il perché abbiamo delle difformità e come possiamo rimuoverle. Questo è ciò che si fa con il debugging

Nel software testing, l'obiettivo principale è far saltare fuori le differenze possibili fra P ed S, quindi bisogna individuare le parti su cui focalizzarsi per trovare queste differenze. Se si scopre anche il motivo dell'errore è un plus, ma non è l'obiettivo principale. Invece, nel debugging sappiamo di avere un errore e vogliamo scoprire il perché di tale errore, in modo da poterlo risolvere.
Dire che P deve essere corretto rispetto ad S, vuol dire che P rispetta la formula introdotta precedentemente, ovvero P è privo di errori. Ma P privo di errori intende molteplici cose, più formalmente possi-

amo definirlo facendo la distinzione fra errore, difetto e malfunzionamento:

1. un errore è ciò che genera un difetto
2. un sistema software è pieno di difetti (o fault o bug) ma non è detto che un utilizzatore del software incappi in tutti. Quando questo accade, il fault genera un malfunzionamento (o failure), che viene percepito dall'utente.
3. un malfunzionamento è il risaluto di un fault

Possiamo cercare di evitare i malfunzionamenti pure mantenendo il sistema buggato, ad esempio isolando il bug ovvero facendo in modo che nessun utente vi si imbatta; è possibile organizzare una campagna di $V\&V$ per nascondere tale bug.
Sempre nella sfera del $V\&V$, vorremo cercare di omettere il fault: per afre ciò, applichiamo delle politiche di fault detection o removal, e questa è la parte più grossa del software testing: si pianificano attività il cui scopo è quello di esporre i fault.
Molto più complessa è l'attività di errore removal, in quanto occorre capire perché il progettista/analista/sviluppatore ha avuto una percezione non corretta del software che andava analizzato e sviluppato.
La parte che riguarda l'analisi e rimozione di errori e fault fa parte del debugging, mentre la ricerca di failure del software testing.
esempio:
(metti codice) il metodo *makeItDouble* mi fa intendere che il risultato sia il doppio del parametro che ho passato come input, ma se passo 3 ottengo 9. Il risultato 9 è la failure, causata dal fault che è il prodotto del parametro con se stesso. Probabilmente, tale fault è dovuta ad un errore umano, come ad esempio un typo di "*" al posto di "+".
Ma ad esempio, se passo 2 come input, non mi accorgo mai del problema e se il codice che utilizzo passa sempre 2, l'utente non sperimenterà mai la failure perché il bug non sussisterà.

Il software testing è composto da attività che vengono eseguite in un ambiente controllato e sul sistema software così come verrà usato dagli utenti finali, mentre nelle verifiche formali si ragiona su un modello astratto dall'originale.
Il lavoro sul sistema vero e proprio ha come vantaggio il fatto che si prova se il software originale risponde alle esigenze per cui è stato creato, ma come svantaggio il fatto che il test non può dimostrare l'assenza di errori, bensì solo la presenza di essi.
Il mood del test engineer è starno: devi essere contento se trovi errori, non se non li trovi (cit*).
Contesto tipico: ho il mio system under test, che viene stimolato da un'attività di test, che è composta da 4 fasi:

- inizializzazione
- esercizio
- verifica: mi chiedo se il system under test si è comportato come mi aspettavo. Verifico delle asserzioni che avevo fatto, se sono verificate il test passa, altrimenti fallisce. Non esiste un test in cui non ci sono asserzioni
- teardown dell'ambiente, ripulisco il system under test da eventuali evoluzioni di stato causate dai test.

esempio: scrivo un programma che calcola il fattoriale su tutti gli interi, su un numero negativo è da intendersi come il fattoriale del suo opposto. ho prodotto *myBuggyFact*: comincio a pormi delle domande:

- quanto è ampio il dominio di test?
- quanto è costoso esplorare un sotto-dominio di input: anche se avessi avuto un enumerazione finita, ma per l'esecuzione del test ci avessi messo molto tempo(es 4 giorni), mi sarei posto dei problemi anche se il dominio è piccolo

- potrei considerare una sottoparte? Potrei considerare se ci sono input equivalenti fra loro
- tutti i possibili valori di input sono equivalenti fra loro?

vogliamo verificare se la nostra implementazione P è conforme ad S, ma nel dominio considerato solo una piccolissima parte causerà difformità fra P ed S: dovrei mettere in piedi una strategia che cerchi di focalizzarsi su quei piccoli cluster del dominio che mi causano la difformità. Quindi ci sono domande fondamentali da porsi quando ci si trova davanti ad un'attività di pianificazione di software testing:

- quali e quanti input selezionare?
- quando posso smettere di testare? All'inizio avrò molti errori, ma poi mi aspetto di avere meno errori avanzando nel tempo. Quindi quando possiamo ritenerci soddisfatti della campagna di testing che h effettuato
- come facciamo a sapere se il risultato di un'operazione è corretto o meno? Ogni test deve prevedere una fase di verifica del comportamento, quindi cosa mi aspetto dalle interazioni di servizio. Non è una cosa semplice da fare, oracle problem: avere una sorgente di informazione che mi dice che risultato aspettarmi in base al comportamento, la risoluzione dell'oracolo può essere effettuata in vari modi come ad esempio dedurre i valori attesi da specifiche, storici, sistemi equivalenti in esecuzione etc...

da qui in avanti, bisognerà porsi queste domande.
Fare software test significa definire delle strategie per eseguire un programma con un campione di dati in input.

7.1.1 Correttezza e reliability

Se non riusciamo a testare esaustivamente il dominio, non riusciamo ad essere certi dell'assenza di errori in un prodotto software, ha ancora senso definire un sistema corretto come un sistema senza

errori? La correttezza di un programma asserisce la sua completa aderenza alle specifiche ($P(d) = S(d) \forall d$) e questo può essere effettuato solo tramite prove formali, quindi la validazione mediante testing richiederebbe l'esplorazione esaustiva del dominio di input. Introduciamo un concetto di correttezza relativo, **reliability**: il sistema è corretto nel senso che non è esente da errori, è possibile che abbia errori. Stimiamo la probabilità di interazione con successo verso il programma, a partire da un insieme di parametri in ingresso estratti casualmente dal dominio di input. La reliability è un attributo statistico di un sistema, non un asserzione di correttezza.
Consideriamo ancora il metodo *myBuggyFact*, in pratica configuriamo il sistema in modo che il metodo sarà invocato in maniera equiprobabile con uno solo dei parametri [-5; -3; 0; 2; 4]; la reliability è la probabilità di corretto funzionamento, quindi possiamo considerare il complementare: $1 - P(fail\_on\_demand)$, ho $\frac{1}{5}$ di probabilità di fallimento e $\frac{4}{5}$ di probabilità di successo. Stiamo legando il concetto di reliability a quello di operational profile, che è una descrizione numerica di come il sistema sarà utilizzato; avrò differenti profili di esecuzione in base al tipo fi utenti. Viene stimato mediante l'analisi e le considerazioni di requisiti e specifiche, oppure in maniera effettiva misurando le richieste al sistema ed l'analisi di log etc...

8 Test automation e Continous test

Facciamo software test per contribuire a dare qualità al prodotto software che stiamo sviluppando: attività sistematiche che mettono in evidenza gli obiettivi per l'utilizzo del prodotto software. Abbiamo dei punti chiave:

- systematic activities: processo
- providing evidence: punti di controllo
- fitness: obiettivi misurabili che si vogliono raggiungere

La software quality assurance prevede un insieme di piani di attività per il monitoraggio ed il controllo del processo di sviluppo software, l'obiettivo è assicurare una sufficiente confidenza che un prodotto software soddisfi le specifiche, sia utilizzabile nel contesto considerato e venga sviluppato secondo le modalità stabilite.
Un piano di software quality assurance dovrebbe considerare anche altri aspetti di software testing? Sicuramente la parte di testing è importante, gli aspetti della software quality assurance sono abbastanza ortogonali al software testing, ma esistono altri aspetti importanti:

- ci deve essere una esplicita gestione del processo di qualità
- quali sono le fasi del processo di qualità
- aderenza standard a procedure o a convenzioni

se testiamo senza nessun tipo di aderenza agli obiettivi da raggiungere o senza strumenti automatici, l'efficacia della campagna di testing viene impattata: come riesco a fare in modo efficiente le attività di software testing.

8.1 Quality control

Gli obiettivi sono:

- monitorare il processo di sviluppo e rilascio
- controllare che i requisiti siano soddisfatti

si utilizzano vari strumenti:

- processo di QA
- metodi d monitoring
- controlli di QA

In una visione di insieme, definire un quality control vuol dire:

- avere un quality plan, con degli obiettivi e con una strategia. Non c'entra nulla con il software testing

- fare quality assurance, viene messo in atto il quality plan usando tecniche di $V\&V$, quality standards...
- post QA, valutiamo tutti i risultati

le attività di software testing devono andare di pari passo con un piano di quality control, quindi dire che il prodotto è di qualità perché è stato effettuato software testing = syntax error (cit*)

8.2 Software configuration management

Per raggiungere un software di qualità, dobbiamo aver strutturato in un certo modo la linea di progettazione e sviluppo. Ho vari componenti:

- identificare un sistema modulare
- sistema di progettazione e/o di sviluppo che consenta il controllo del versioning: che cosa vul dire release x.xx e cosa cambia con la y.yy, quindi non solo il sistema automatico che fa il versioning
- un sistema per la configurazione di ambienti di build

normalmente ciò che accade è che vengono definiti due macro-ambienti:

- pre-production: regole sostanzialmente per gli ambienti di sviluppo
- production: definisco dei QA gates, dei punti di controllo per cui vogliamo vedere se ciò che abbiamo definito va bene.

Tradizionalmente, le release sono associate a vari momenti:

1. software in sviluppo
2. versione per-alpha
3. versione alpha: usato in ambienti controllati e da utenti che appartengono a determinati profili
4. versione beta: può essere usato dagli utenti in ambienti reali, ci si aspetta una qualità che va migliorata

5. versione release candidate: abbiamo una determinata confidenza sulla beta
6. release gold: passati determinati check, abbiamo la nostra release

abbiamo definito dei quality gate, in alcuni le attività di software testing sono perdonatati, altri sono per ambienti di production, in cui sono gli utenti a fare le scelte e quindi i software tester non hanno più un'importanza così alta.
Il problema dell'approccio è che se non abbiamo strutturato un piano di software configuration management, tutto andava bene prima dell'operation e una volta uscito dall'ambiente controllato non funziona più nulla. Questo perché c'è un passaggio netto dagli ambienti di pre-production a quelli di production, più il gap fra i due ambienti è grande e più sono i rischi che tutto vada a finire male. Vogliamo far si fin dall'inizio che questo salto sia il più "dolce" possibile.
Fare software testing quindi non basta per poter determinare la QA: se trovo errori è bene, altrimenti non ho nulla da dire. Devo quindi relazionare il software testing ad un piano in cui ho vari pillars

8.3 QA and release cycle - DevOps

Una serie di pratiche, più o meno recenti, legate alle pratiche di continous integration, continous testing etc... L'approccio a cascata introduce problemi vari, sopratutto nel passaggio dall'ambiente di pre-production a quell di production per i più svariati motivi. L'approccio al concetto di release cycle hanno portato al concetto di DevOps (Development and Operations): ci sono sempre 3 categorie di attori:

- ambiente di development: ingegneri e sviluppatori
- operation: team che prendono una versione gold di un software e si occupano di installarla sulle risorse dei clienti, di fare deployment etc... Le persone che sono nel mondo Ops hanno una visone

molto chiara delle infrastrutture e possono avere meno chiaro del perché nel software le cose funzionano in un certo modo

- in parallelo ai due mondi c'è il QA: ingegneri che devono verificare che ciò che viene rilasciato matchi con un certo quality plan

l'approccio tradizionale inserisce l'intervento dei QA nei vari gate, in modo agnostico rispetto al scelte di Dev o Ops. Il suggerimento dell'approccio DevOps è quello di vedere queste 3 entità come strettamente legate fra loro: ogni entità deve sapere qualcosa rispetto agli altri, i team devono essere strutturati in modo collaborativo, così che tutti abbiano una visone d'insieme completa.
In pratica, si passa da un approccio a fasi, ad un concetto ciclico in modo continuo, dove le varie fasi di continous integration o build del software, vanno in commistione con le vasi di continuous delivery e continuous deployment del software: ciò che un'azienda doverebbe fare è mettere su una pipeline/framework dove ogni modifica scatena in modo automatico la compilazione del sistema, la verifica del sistema, la validazione automatica del test e proponga la versione che ha passato tutti i test ad un ambiente simile al deployment. A questo punto, vengono proposti dei test di accettazione, che se vanno a buon fine scaturiscono nel deployment nell'ambiente effettivo di production.
Ricorrono sempre i concetti di automazione e deployment, punti chiave del meccanismo DevOps. Quindi un'azienda che opera seguendo le continous practicings, ha strutturato una catena di framework tali che per ogni commit ricevuto sul src code è possibile abilitare l'intero meccanismo di build, delivery e deployment in modo che sia integrato nella gold release.
Quindi, una volta scritta una nuova funzionalità del codice, in modo automatico questa viene compilata, testata, deployata in un ambiente simile a quello di production, lanciare test di validazione per requisiti utente. Se tutto va bene viene proposta una release da de-

ployare e poi deployarala effettivamente.
Per fare tutto, è necessario strutturare i processi di DevOps perché siano ripetibili e cercare di limitare al minimo l'intervento umano sul framework definito. Sarebbe interessante se ogni ambito avesse a disposizione dei tool per poter automatizzare tutto il processo.

8.3.1 Testing automatizzato

Fare test in un contesto DevOps vuol dire fare test attraverso l'uso di infrastrutture o framework che eseguano le attività per noi. Come test engineer occorre capire quali test fare e stabilire le policies per tali test, ma poi pensare a strumenti per la selezione e la prioritizzazione dei test, o anche per stabilire la bontà dell'insieme dei casi di test sviluppati. Tutte queste attività si contrappongono al manual testing, ancora oggi molto diffuso e causa della nota di spesa tra le più alte delle aziende di software. I principali obiettivi dell'automated testing sono:

- migliorare l'efficienza dei passi di QA
- migliorare l'efficacia dei passi di QA

i benefici saranno:

- costi ridotti
- ridurre la durata dei test
- aumentare la ripetibilità dei test
- aumentare l'attendibilità statistica dei responsi delle fasi di QA
- migliorare la documentazione delle failures

In particolare all'interno del mondo DevOps, così come si fa continous integration, allo stesso modo si cerca di stimolare il mondo del continous testing: ogni volta che uno sviluppatore propone un cambiamento, vorremmo attivare una catena di processi software che

eseguano i test. I test vengono automaticamente eseguiti sulla versione attuale del sistema, ma è possibile anche configurarli su degli ambienti con caratteristiche fornite dagli altri team (ad esempio con altri SO, librerie etc...) e non solo su ambienti locali allo sviluppo. Se dovessi fare a mano il passaggio da ambiente locale a configurazione custom avrei dei costi, che vengono abbattuti se tutto avviene automaticamente.
L'approccio ha diversi vantaggi:

- i problemi vengono subito alla luce
- lo sviluppatore deve solo preoccuparsi di mandare la nuova feature in analisi, non deve preoccuparsi dell'ambiente di esecuzione etc... Si preoccuperà solo del risultato finale, che sarà booleano: passa o non passa

se immagino di avere un version control system, ogni volta che faccio un commit/PR sul codice partono una serie di attività in modo automatico che verificano se i vari needs configurati sulla pipe sono stati rispettati.

8.3.2 CI + CT

Stiamo parlando di strutturare un ambiente di sviluppo che:

1. non dipenda dalle singoli abitudini del membro del team
2. possa tenere conto agilmente di aspetti che vengono dal mondo Dev ed aspetti che vengono dal mondo Ops e QA

I pezzi fondamentali di cui abbiamo bisogno:

- ambiente di sviluppo locale allo sviluppatore, che deve essere configurato in maniera indipendente ed in modo che nessuna scelta impatti quelle degli altri o che impatti sulla qualità del prodotto software
- ogni sviluppatore deve inviare il contributo su un version control system che deve armonizzare i contributi ricevuti da tutti

- in un ottica CI, ogni potenziale commit sul verison control system deve generare un processo di build che almeno compili il codice (su una macchina astratta diversa da quella degli sviluppatori). Chi può dirci qual è un sotto-insieme possibile di macchine su cui andare a compilare il codice: sicuramente il project leader dell'ambiente Dev, ma ci fa Ops sa dove andrà testato il sistema e quindi potrà fornire tale informazione. Nel momento in cui viene committato un contributo nella code base occorre che il codice compili sul determinato numero di macchine. Occorre un build server che si occupi di fare la compilazione sulle macchine richieste, fornendo un report con i risultati.; questo è strettamente legato al concetto di software testing. A questo punto, è possibile agganciare un pezzo delle attività di QA proprio sul build server: il report ottenuto dal build server può includere i risultati dei test, riportando quelli passati e non

quindi, cerchiamo di capire come tirare su un ambiente che ci introduca alle pratiche almeno di CI e CT.

8.4 Verison control, automated build e test

Git: version control system open source e distribuito, di cui esistono versioni on-line, in particolare per tracciare le versioni useremo Github: ogni volta che viene proposta un cambiamento alla code base, vorremmo che vengano triggerate attività di CI e CT.
Quindi, dobbiamo affidarci ad un framework che ci aiuto almeno a compilare il src code. Per automatizzare la build: Maven, che è un sistema di build principalmente ad ambienti Java. Permette automaticamente di gestire le dipendenze sia esplicitamente, tramite la dichiarazione di librerie (con versioni) da considerare e sia transitiva, ovvero dichiarando solo quello che si usa esplicitamente ed al resto ci pensa il sistema di build. Maven struttura il build in una serie di fasi, l'errore in una fase i interrompe il processo di build, ogni step del processo deve identificare la fase di appartenenza. Quindi, sap-

piamo come rendere il build automatico, in modo che la code base scateni azioni maven su qualche build server, ora vorremmo rendere il processo di build serlf-testing. Il framework di rifermento è JUnit, ha consentito lo sviluppo di pratiche di test-driven development: scrivo un test che interagisce col sistema, dopo aver scritto il test scrivo la funzionalità finché il test non passa. ATTENZIONE: Junit consente di scrivere ed organizzare test program, non dice assolutamente la strategia di test, quali valori usare etc... è un linguaggi che permette di facilitare la scrittura dei test cases, è un DSL che ha delle astrazioni che consentono di gestire i cicli di vita dei test in modo più semplice. La fase di test viene strutturata attraverso diverse fasi, come agganciamo la codifica dei casi di test all'esecuzione ed al continous testing: con Maven, nella fase di test Maven va alla ricerca di tutte le classi che hanno un determinato nome, se le trova lancia un'istanza di JUnit e le passa in ingresso.
Tutto ciò avviene in locale, ma serve un build server, in qualche modo collegato con la codebase: Travis CI.

9 Modulo III: Enterprise IT

10 Introduzione

IBM: azienda nel mercato da 110, nata per cerare dispositivi per il business. Non punta al mercato di largo consumo, bensì alla commercializzazione con altre aziende che acquistano prodotti hardware o software. Enterprise IT: cos'è l'IT di una grande società e quali sono i requisti di una grande società, ovvero adottati da un cliente di tipo enterprise.
Enterprise: impresa che può essere:

- azienda privata
- ente governativo
- etc...

Obiettivi: partendo da un background di sviluppo e progettazione software, come posso metterlo in esercizio in un ambiente enterpire. Vogliamo capire:

- i tipici Enerprise level per ambienti IT
- quali sono i tipici requisiti non funzionali e l'order of magnitude di essi
- introdurre considerazioni base riguardanti il Total Cost of Acquisition (TCA) o Total Cost of Ownership (TCO)

10.1 Cos'è un Enterprise

Un Enterprise è una grande impresa che potrebbe essere un'azienda, un ente pubblico, etc... in generale in ente sofisticato con certe caratteristiche:

- grande compagnia o organizzazione
- gestisce numero elevato di client

- ha milioni di impiegati
- richiede disponibilità 24x7: se erogo un servizio, questo deve essere sempre attivo (365 giorni l'anno)
- operano in una indistry strettamente regolata: in una industry ci sono molte regole, che permettono alle aziende di cooperare. Industry è il mercato in cui si trova ad operare un'azienda, ma è anche l'insieme di regole di business: per entrare nel mercato esiste un insieme di regole a cui bisogna sottostare, non si può aprire, ad esempio, una banca dal nulla: questa è una industry. Nelle grandi Enterprise esistono delle regole, ad esempio di sicurezza per industry del modo delle compagnie aeree, o anche la replicazione dei server che mantengono i dati dei clienti, in modo da far fronte a disastri naturali etc..., magari distanti tot km, per quello che riguarda un industry del mercato bancario
- fornisce servizi sotto degli SLA restringenti
- deve essere in grado di gestire picchi di carico, spesso anche 1000x il solito

esempio: aziende del mondo tel&co. Lavorare e gestire un numero di clienti così elevato porta benefici all'azienda, ma anche ai clienti, che possono avere i servizi a costi minori perché ammortizzati sul numero totale dei clienti. Una Enterpirse level solution è una soluzione in larga scala che supporta tali concetti, che richiede la cooperazione di esperti IT.
Nel classico ciclo di vita dell'applicazione, ci si focalizza sul develop del codice, eventualmente pensando al middleware, ad esempio utilizzo di DB, application server (che è un server che permette operazioni di tipo transazionale, per avere operazioni ACID) e SO, o per ambienti di virtualizzazione, containers.
Esistono tutta un'altra serie di aspetti per un'applicazione enterprise:

- data replication, partitioning

- monitoraggio dell'ambiente di esecuzione
- sistemi per gestione di sicurezza
- orchestrazione, load balancing

tutto ciò si trova sopra la parte hardware, che può essere on premise o in cloud.
Organizzare un business di tipo Enterprise conviene anche per piccole aziende: se scalo nel numero di prodotti consegnati, il costo di spedizione etc... è più ammortizzato. Lo stesso vale per applicazioni Enterprise: posso partire col classico server tower, passare poi ad un data center con rack (cambia la dimensione a seconda del server, nomenclatura x-unit).
Nelle applicazioni frontali esistono due tipo di sistemi sempre presenti nel back-end:

- system of engagement
- system of record

coinvolgo un grande numero di sistemi Enterprise per ogni transazione, se qualcuno di questi va giù è un problema.
Sistemi-Z IBM (nell'ambito dei systems of record): i vecchi mainframe, che hanno sempre la caratteristica di poter mandare in esecuzione software datati, ma anche modelli.
Per accedere ai sistemi si usano le classiche API, i sistemi hanno degli use case ben chiari e definiti. esempio: use case di acquisto di un biglietto aereo

1. scelgo la meta, la data, l'aeroporto di partenza e di arrivo etc...
2. l'applicazione gira, mi fornisce la lista di voli possibili, i posti disponibili. Hunder the hood: ho "parlato" con un system of engagement, che ha contattato un system of record, avranno contattato DB con una query specifica per filtrare i risultati etc... Una serie di ingaggi sempre più sofisticati verso il system of record, il cui poi risultato viene sempre mandato al mio system of engagement e sul front end della mia applicazione.

11 Requisiti non funzionali Enterprise

I fondamentali sono:

- reliability ad availability
- serviceability
- security
- performances
- scalability

In una Enterprise, contano anche gli ambienti, che sono diversi ed hanno diversi requisiti:

1. sviluppo
2. test
3. pre-production aka QA (Quality Assurance): ambiente copia del production env, per testare che tutto sia ok
4. produzione: ambiente fondamentale, è quello che riceve le query per la mia applicazione etc... Se va giù anche per poco tempo, è complicato
5. high availability: serve per assicurarsi che se un ambiente cade, ce ne sia un altro che ha e stesse informazioni
6. distaster recovery

11.1 Reliability ed availability

Spesso reliability = affidabilità, se è al 99% è alta? Manco per nulla, anche 99.9999%, per ad esempio un volo aereo non è alta.
Un affidabilità del 99.99999% (i cinque "9") è considerata il minimo (rivedi appunti SDCC), a seconda del tipo di applicazione, crescerà l'affidabilità.
Se ho due sistemi che hanno un'affidabilità del 99.9%, l'affidabilità

del sistema dato dai due che cooperano decresce (è il prodotto). È importante lavorare sull'affidabilità, quando si parla di affidabilità occorre specificarne il livello, non basta dire che è up 24x7.
Molte industry hanno dei requisiti di affabilità dettati dagli SLA e che sono dettati ed imposti a chi sta nell'industry, mentre altre scelgono i loro requisiti privatamente.
Al crescere del numero di sistemi, cresce la probabilità di failure: avere una macchina che ha affidabilità de 99.999% è diverso da averne 1000 che hanno la stessa. Il sistema avrà un costo diverso, ma anche il servizio erogato sarà diverso e quindi bisognerà eseguire l'analisi economica.
La reliability è invece una misura della ridondanza dei componenti, in modo da permette al sistema di continuare a fornire il servizi anche in presenza di fallimenti. In presenza di ridondanza in un sistema, l'uptime viene calcolato come 100 - il prodotto dei failure rate dei componenti ridondati.

11.2 Serviceability

Si riferisce all'abilità del supporto tecnico di poter risolvere problemi in real-time, invece di fermare il sistema ho delle copie di backup. L'obiettivo è ridurre il costo operazionale e mantenere la continuità di business.
Ad esempio, la possibilità di poter fare disk swapping, se uno si rompe.

11.3 Security

Un altro requisiti non funzionale importante, elemento più critico che può esistere nell'Enterprise. Ormai gli attacchi sono presenti e costanti, bisogna poter gestire la sicurezza dei sistemi e spesso viene dato per scontato. È anche necessario per poter essere compliant a determinati requisiti: GDPR multa del 4% se un data breach non viene denunciato entro 24h.

Un altro requisito importante è l'encryption dei dati:

- sia perché chi arriva e copia i dati li trova cifrati
- sia per il valore dei dati stessi

internamente solo le persone autorizzate possono accedervi. Ma quali dati cifrare e quando cifrarli, inoltre dove vanno salvati i dati cifrati e che si occupa della cifratura? Problemi con cui le aziende hanno a che fare, l'approccio tipico era quello del cifrare i dati personali. Selected encryption: se cifro solo quei dati, sto dicendo all'attacker che i dati sensibili sono lì. Un altro metodo può essere il full disk encryption, ma non risolve tutti i problemi: cifrare/decifrare grava sulla CPU, l'operazione può essere costosa ed inoltre i dati passano in chiaro appena escono dal disco. L'approccio Enterprise è la pervasive encryption: cifro tutto, evito le modifiche dell'applicazione senza inserire la cifratura nell'applicazione. Diversi benefici:

- si cifra tutto e tutto viaggia cifrato nel sistema (i dati non sono solo cifrati nel disco)
- l'impatto di processamento dei dati viene fatto su hardware specifico e solo quando necessario
- tutto in container/servizi che permette di amministrare bene le chiavi di cifratura

Se ho i dati anche sulle repliche per ridondanza, il sistema di cifratura permette di recuperarli e di recuperare le chiavi di de-cifratura

11.4 Performances

La performance di un computer è l'ammontare di lavoro utile compiuto da un sistema. Ci sono diverse metriche di performance, per i system of records la metrica più importante è data dalle Transactions per Seconds (TPS): le computazioni DB centriche sono il cuore di questi sistemi. Ad esempio:

- transazioni con carte di credito: milioni di TPS
- acquisti di tickets: centinaia di migliaia di TPS
- chiamate telefoniche loggate: fino a milioni di TPS

al crescere del business, cresce anche di conseguenza il numero di TPS: ad esempio una catena di supermercati, che deve tenere traccia oltre che del fatturato, anche del numero di prodotti acquistati. Ho due approcci possibili per scalare, in modo da poter distribuire il carico di lavoro sui vari server:

- scale up (verticale)
- scale out (orizzontale)

tipicamente lo scale up ha una crescita dei costi esponenziali, rispetto a quelli lineari dell'approccio scale out.
In un ambiente Enterprise, oltre al costo dell'hardware devo anche tenere conto dei costi di software, del networking (servono più hub e switch per più server), serve anche uno spazio più ampio e di manutenzione dello spazio (aria condizionata, IT admin) etc...
I costi, nella realtà dell'Enterprise, crescono nel modo inverso: oltre all'ambiente di produzione ho un ambiente di test, uno di alta affidabilità, uno per disaster recovery etc..., quindi se il mio business cresce dovrei aumentare il numero di server in tutti gli ambienti (non solo in quello di production). Quindi, seguendo una logica di scale out, in sistemi di dimensioni medie per una singola applicazione si arrivano ad avere 10k server e quindi la manutenzione non è banale. La maggior parte dei data center tende a seguire l'approccio di scale out, in quanto è più semplice: partendo da un ambiente piccolo, si vanno ad aggiungere altri server.

12 Costi per Enterprise

Consideriamo un DB come software da dover gestire, l'open source nel mondo Enterpirse viene usato ma non è gratis: occorre il technical

support, in modo che in caso di problemi sia possibile avere subito una soluzione.
Solitamente c'è una licenza, che può essere legata a diversi aspetti; alcuni sono commerciali:

- IBM D2
- Microsoft SQL
- Oracle DB
- Dynamo DB

se ad esempio la licenza è legata al numero di core (spesso bei DB), ho bisogno di tante licenze quanti sono i core e quindi costi importanti.

12.1 TCO vs TCA

Quando si parla del costo di una soluzione è importantissimo parlare del TCO (Total Cost Of Ownership), che è diverso dal TCA:
ogni applicazione ha dei costi legati ad hardware, software, servizi

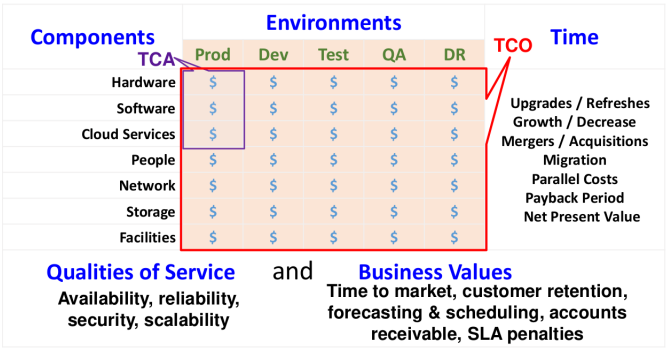

Figure 1: TCO vs TCA

cloud e questi sono i TCA (Total Cost of Acquisition). Vanno anche considerati i costi di gestione del server e dell'applicazione, tipicamente per ogni 30 server occorre un IT admin (anche se non è una sola persona, possono essere anche di più che poi completano un full time equivalent, ovvero 40h/sett).

Bisogna anche considerare altre voci di costo, che sono presenti anche per tutti gli altri ambienti e di tutti i requisiti introdotti prima, come anche dei business values e del tempo: una cosa è il costo di acquisto, ma poi nel tempo dovrò fare degli upgrades software, refresh hardware e quindi tipicamente il TCO si fa su una finestra di 5 anni; spesso il refresh cycle viene stabilito nell'azienda: sul totale dei server, se ne cambieranno periodicamente (non per forza tutti insieme) un certo numero. Spendere decine di milioni per mettere su un ambiente Enterprise è abbastanza comune: a seconda della scelta di avere scale up o scale out ci saranno costi diversi, ma tipicamente il costo di scale out è maggiore; inoltre, la parte che ormai ha un costo predominante del TCO è quella del software come anche il costo del disater recovery (che deve mettere insieme i costi di hardware, software etc... che copre non per forza tutti gli stessi servizi dell'ambiente main site).

12.2 Rack - z15

In un ambiente Enterprise, rispetto ad un ambiente x86, posso avere un singolo rack (evoluzione dei mainframe), ne esistono due versioni:

- z15, ultima versione. Possono girarvi n sistemi operativi
- LinuxONE, mainframe dedicata la mondo Linux

12.2.1 Hardware z15

L'hardware viene visto come un data-center in una scatola, composto da n server più i sistemi di storage, networking etc..., è stato progettato per virtualizzare e condividere le risorse: sopra il firmware di virtualizzazione gira lo z/VM, che crea VM dal punto di vista hardware (hypervisor di tipo 1).
Su ogni z/VM gira un z/OS e possono essere differenti, o anche un altro hypervisor.Ci sono diverse tipologie di processori:

- di tipo generale (equivalente delle CPU)

- alcuni dedicati a fare dei compiti specifici (ad esempio per la gestione dell'I/O)
- processori dedicati all'accoppiamento di due macchine, per farla sembrare una singola
- zaaP, ottimizzati per eseguire il codice Java (ottimizzare la garbage collection etc...), sono una configurazione particolare di quelli generali
- spare processor, di scorta. In questo modo se un'altra delle CPU fallisce, questi entrano in gioco

LPAR suddivisione del sistema z in una macchina logica che viene configurata su misura, viene fatta anche dal punto di vista hardware per efficienza. Le LPAR sono isolate dal punto di vista hardware, in modo che in caso di guasti le altre LPAR rimangono su. Siccome il server z è configurato con diversi LPAR, riesce a stare su per decine di anni (senza fare reboot), possono condividere hardware, ma la memoria deve essere isolata.

RAIM - memory sparing si usano memorie RAIM (Redudant Array Indipendent Memory): ci sono più banchi di memoria, in modo che se uno si rompe ce n'è un altro che ha i dati disponibili. Il RAIM prende il 20% della memoria diposnibile, e versioni non RAIM non sono disponibili su macchine simili.
Lo z15 contiene fino a 108 core a 5.2GHz, fino a 40TB di memoria configurabile, offre anche retro compatibilità fino agli anni 60. Si parla di un "data center in a box", i costi saranno molto elevati.

12.2.2 Confronto x86

Sicuramente per avere le stesse prestazioni devo avere molteplici server, e abbiamo visto che al crescere dei server crescono anche i

problemi in termini di availability, security etc...

12.3 Gestione dei picchi di carico

Un altro problema importante da gestire è calibrare un sistema per gestire i picchi di carico: fare over-provisioning è una soluzione tipica, ma c'è un grande spreco di risorse, quindi prezzi più alti così come anche consumi più alti.
Uno degli approcci è quello di usare un ambiente cloud, per sfruttarne l'elasticità, tipicamente per una Enterprise si configura un ambiente dedicato in base al tipo di applicazione. L'utilizzazione tipica dei server sta intorno al 20%, sotto alto utilizzo siamo introno al 30%, questo in base allo statistical multiplexing model: in base alla media, posso avere picchi differenti in base all'applicazione. Esistono diversi studi dello statistical multiplexing per cui se mettiamo su un servizio su un server, la probabilità che tutti i server abbiano un picco di richieste è molto bassa (utilizzazione pari ad 1), ci sono diversi server di tipo Enterprise in cui la distanza fra picco e media è bassa.
Queste macchine inoltre sono state pensate per lavorare ad utilizzazione al 100%, il sistema operativo è abituato a girare all' 80-90% e tutto ciò porta ad avere un utilizzo delle risorse molto ottimizzato.

12.4 Case study - TCO su 5 anni

Come abbiamo visto, per un analisi completa del TCO, dobbiamo avere una visione completa su tutti i componenti dell'applicazione, non solo in termini hardware e software ma anche per le persone etc... e questo per tutti i possibili ambienti.
A seconda degli anni, ci sono costi più preponderanti rispetto ad altri etc...
Per effettuare un TCO, occorrono 4 passi principali:

- Analizzare lo UC e capire il workload, solitamente si effettua parlando con il cliente. Occorre capire sia la media che i picchi

del workload, misurato in TPS (transaction per seconds): non avviene così spesso che l'analisi venga fatta prima di mettere in esercizio un sistema, nel caso in cui l'ambiente sia già in esercizio, i valori si possono inferire dal sistema già in esercizio. Si ottengono dei report su una finestra minima di 1 mese per ricavare i valori, altrimenti occorrerà fare delle stime o delle ipotesi.

- Valutazione e dimensionamento della piattaforma per permettere di supportare il workload, quindi ad esempio capire la taglia del server su cui deployare l'ambiente:
  - taglia dell'ambiente di produzione, considerando i requisiti di HA
  - sizing di tutti gli altri ambienti: Dev e Test, Pre-prod, DR

  il sizing del distaster recovery è molto interessante, bisogna trovare il compromesso giusto. Occorre inoltre confrontare scale up con scale down
- valutazione dei costi, analisi del TCO
- descrizione con grafici, dati, tabelle etc.. facendo riferimento ai dati di input

12.4.1 Sizing

Fare un sizing preciso è complesso: occorre analizzare i dati, facendo analisi tecnica dell'hardware e del software, la tipologia di VM e di hypervisor etc...
Partiamo da un'ipotesi semplificata: il numero di core di un ambiente Enterprise è almeno 17 volte minore di un ambiente x86. Partiamo da una ipotesi consolidata (slides)
A questo punto, andrebbe progettata l'architettura tecnica: spesso esiste già, quindi va modificata ed evoluta (N.B: l'ambiente di pre-prod dovrebbe avere la stessa taglia del prod). Si parla più che altro di core: è possibile avere un server con 12 core, o uno equivalente

che ne ha 24 l'importante è la capacità elaborativa.
esempio 1: situazione active passive, dove mantengo dei core per il DR da parte, per sopperire a disastri.
esempio 2: active active (vedi video).
In LinuxONE:
esempio1: posso tenere nel sito 24-75 CBU (core di backup)
esempio 2: ho abbastanza CBU per poter coprire la pre-prod e la prod, con un ambiente di sviluppo e test più piccoli. LinuxONE: una delle due architetture z della IBM, su cui girano solo software di tipo Linxu[2].

12.4.2 Costi

Occorre valutare diversi costi:

- Il software deve girare su un software stack, il tipico prevede:
  - OS
  - VM
  - Application server: oltre al classico software, permette di poter gestire differenti transazioni in parallelo. Ho il software che coordina le chiamate alle transazioni, da smistare sui vari server.
  - DB
  - data replication tool
  - tool di monitoring
  - security tools

  per tutti, occorre conoscere diversi prezzi.

Gli altri costi di cui si tiene conto sono:

- costi per le persone, che devono manutenere i server. Un full-time equivalent (una persona assegnata a tempo pieno) può fornire

[2]distro supportate: Red Hat, Suse, Ubuntu

supporto fino a 30 server. Occorre considerare anche i fully loaded cost, non solo il salario: oltre a quello ed alle tasse, ci sono una serie di costi come l'ufficio, il manager del team etc...

- costi di rete: tipicamente, per ambiente Enterprise è in media 7000$ per server
- costi di facilities: affitto del data center, costi di elettricità

un altro elemento che sta diventando importante è il carbon footprint, ovvero l'emissione di CO_2 del data center.

12.5 Use case: transazione con carta di credito di una banca

Ogni acquisto del cliente va registrato sui server, occorre verificare un sacco di cose per poter completare e validare una transazione. Una transazione concettuale è fatta di molteplici transazioni IT, assumiamo che un core x86 possa effettuare 20 TPS, intese come transazioni concettuali.
Importante: il costo per core conta anche quelli spare, quindi se non attivi dono soldi "buttati".
Per il software: a seconda del utilizzo può cambiare il prezzo: ad esempio per l'ambiente di production può avere un costo maggiore di quello di dev.
Tipicamente: sul LinuxONE è possibile avere diversi carichi sulle CPU, perché sono progettati per lavorare al 90-95 % di utilizzazione, mentre per x86 non è così (se il carico cresce troppo, si rischia di violare lo SLA).