Concetti generali

VERIFICA VS VALIDAZIONE

- **Verifica: sto costruendo bene il prodotto? è conforme alle specifiche?** il nostro obiettivo è il controllo di qualità delle attività svolte nello sviluppo, il prodotto è conforme alle specifiche? (confronto l'input di una fase con l'output della stessa fase)
- **Validazione: sto costruendo il il giusto prodotto? è conforme alle richieste del committente?** controllo qualità rispetto ai requisiti del committente, il prodotto è conforme alle attese del committente? la 'somma delle validita' è conforme?
- con V&V rassicuriamo l'utilizzatore che il sistema è adatto allo scopo, basato su una 'confidenza' dipendente da diversi fattori.

Come dimostro che il mio programma rispetti le specifiche?

- *verifica formale*: se il mio programma P è una funzione, ed S è la specifica, allora dati input del dominio D e output del codominio C, P(d) = c = S(d)
  ❌ è logico-matematica, dovrei dimostrarlo per ogni valore!
- *sw test/debug*: cerco di scoprire se P ha qualche imperfezione, concentrandomi sulle differenze tra P ed S, individuandole e rimuovendole

DEFINIZIONI

- *Nell'ambito di sw è sbagliato parlare di 'correttezza',* poichè una cosa corretta è senza errori, ma non è definito quali siano gli errori.
- Un **errore/error** causa **difetto/fault** che causa **malfunzionamento/failure**.
- **Failure**: comportamento non corretto ed osservabile del sistema, es: sist. non risponde, risponde diversamente. Con il testing cerco di farle emergere.
- **fault/bug** : non implica per forza una failure, ma è sintomo della presenza di un errore.
- **error**: causa un fault, è un errore umano concettuale/interpretativo/etc., Con il Debugging cerco di evidenziarle.
- con **Failure containment** applico **failure prevention**, nascondo all'user il malfunzionamento (es. andare a piedi su un'isola)
- con **Fault detection** opero sul fault.
- con **prevention** opero sull'errore.

Software Testing

- esecuzione di 'esperimenti' in ambiente controllato al fine di acquisire fiducia sul suo funzionamento, su aspetti funzionali di sollto. Proviamo se la Nostra rappresentazione va bene.
  ✅ vedo se sistema risponde alle esigenze, o genera malfunzionamenti.
  ❌ dimostro che un malfunzionamento esiste, ma no che non esista. Inoltre dominio D è vasto, non posso provare tutto, devo usare un 'sample' di dati. Questo vuol dire fare un **test**.
- Normalmente si usano Assert per confrontare risultato atteso (da specifiche, storici) con ottenuto, mediante un oracolo.
- Non si parla più di correttezza (che è assoluta) ma di **reliability**, che è una stima della probabilità di successo a partire da un insieme estratto casualmente dal dominio (poi può dipendere da budget o altro). E' un attributo statico, non asserzione di correttezza. Reliability = 1 - P(difetto)
- L'insieme di input che do sono profili operazioni, stimati attraverso requisiti, specifiche, o effettivamente misurati (log/report). La reliability dipende da questi profili, sia cambiando gli elementi, che le probabilità.

Test automation & Continuos

- Il processo error->fault->failure è di tipo VERIFICA.
- Se l'oracolo fosse l'utente potrei parlare di validazione.
- **Software quality Assurance** : consiste nella definizione di 'momenti di controllo' in cui, associando dei 'punti di controllo' valuto se posso procedere o prendere altre decisioni. Insieme alle attività sistematiche di routine, mi aiutano a capire l'evoluzione del progetto.
- Resta sempre il problema della definizione di 'confidenza', inoltre il **software testing** è solo un sottoinsieme del SQA.
- Abbiamo il **QUALITY CONTROL**, che effettua delle monitorazioni, mediante quality plan (goal + strategie, ovvero regole generali), quality assurance (v&v, come realizzo le strategie), e post-QA (misure,analisi, feedback).
- **SW CONFIGURATION MANAGEMENT**, dove definiamo i componenti del sw, sviluppo/building automatizzato e controllo sui cambiamenti (cambio x, che parti tocca del sw?).
- Nell'approccio tradizionale ho preAlpha - alpha - beta - rc - gold. Ogni passaggio è definito mediante quality gate, più vado avanti più sarà difficile tornare indietro per dei controlli (aumento distribuzione).

- pratiche atte a trovare l'intersezione tra DEV, QA, OPS (parte operazionale del sw), basate sull'idea che ogni commit sia l'ultimo, automatizzandolo (build) e testandolo (QA). Ciò fa parte del **Continuous deployment**, che è così siffatto:
- *CI*: include buil + test locali, ottengo risultati
- *CDdelivery:* deploy + test su macchine diverse, in modo automatizzato. (Include CI)
- *CDeployment:* deploy su ambiente di lavoro
- Questi processi devono essere automatizzabili e ripetibili, nel DEV modifico il codice, in OPS i target dei test, in QA gate di qualità. Si 'autano' tra loro.
- CIntegration << CDelivery << CDeploy

- Se ho loro allora parlo di **Continuous Testing**, in cui i test vengono eseguiti per ogni evoluzione della code base, integrando i processi di QA (testing)

AUTOMATED TESTING

- parola chiave automazione, per i test, per la prioritizzazione, per la bontà, in modo da migliorare efficienza/efficacia del QA, riducendo costi e durata test

FRAMEWORKS

- **GIT** è version control.
- **Maven**: sistema di build per automatizzazione, gestisce le dipendenze dirette e indirette. Un progetto MAVEN è strutturato in directory, in particolare fa parte di una 'struttura' (archetipo), ed è divisibile in moduli, descritto da un pom.xml, presenta una cartella target di output, che è buona norma non mettere nei commit. Le dipendenze vengono acquisite da maven central.
- **JUNIT** : framework per implementare i test, ma non ci da info o consigli su che fare. Presenta comandi `@BeforeAfterBeforeClass` etc. per setup o teardown, ma non posso specificare l'ordine.
- **CI** : esistono vari framework, tipo github Actions, con specifici setup

Maven include le seguenti fasi:

- validate: tutte le info sono disponibili?
- compile: compilo sorgente
- test: provo i test
- package: creo JAR
- integration test: faccio package in ambiente dove test integrazione eseguibili
- verify: vedo se package rispetta i criteri di qualità
  install: installo package per usarlo
- deploy: copio in repo remota
- clean: pulizia artefatti/target
- site: genero documentazione

DOMAIN PARTITIONING

- Con approccio devOps ogni commit può essere l'ultimo.
- Come costruisco un insieme di test buono per rivelare errori?
- **Obiettivo** : ho un programma da testare SUT, con un dominio di input D, di cui considero un insieme più piccolo Tx che identifica errori Ex, in funzione di un budget B.
  Voglio creare Tg (minimale) che rientri nel budget, e rispetto altri gruppi di test trovi più errori e sia di dimensioni uguali/minori.

Come trovo Tg?

- **RANDOM**
  ✅ semplice
  ✅ utile quando mi approccio alla prima volta ad un test
  ✅ può rilevare errori grossolani
  ❌ ricerca senza metodo
  ❌ devo fare più test per avere una certa confidenza.
- **CATEGORY**
  individuo classi di equivalenza, ed eleggo un rappresentante. Queste classi sono influenzate dalla documentazione, familiarità col SUT, etc., a volte è il nome di un elemento che mi suggerisce (ma non è detto sia valido!)

TIPI DI APPROCCIO

- UNIDIMENSIONALE: Ogni parametro è indipendente, quindi mi basta che sia preso almeno una volta, non mi interesse se da solo o con altri.
- MULTIDIMENSIONALE: prodotto cartesiano dei parametri multidimensionali.
- In ogni caso, nei miei test ho sempre un output atteso, e anche se uso approccio unidimensionale potrei mettere più test che per me sono significativi. Devo sempre fornire risultato atteso.

CONCETTI DI UNIT/INTEGRATION TESTING

V-MODEL

- Test e requisiti vengono usati per validare il prodotto, per vedere se sistema funziona e fa ciò che serve.
  In realtà nulla mi vieta di analizzare i singoli pezzi.
- Nel V-Model abbiamo le due 'linee' componenti la V che indicano un approccio sequenziale da parte del designer e del tester, fino al merge per il codice finale. In realtà ad ogni 'altezza' del v-model avviene uno scambio di informazioni.
- Il designer specifica ed implementa le funzionalità, rilasciandole al tester, il quale specifica ed implementa test che poi eseguirà. Ciò normalmente avviene quando al rilascio di nuove funzionalità/versioni, ma anche ad ogni commit o pull request
- Un'altra prospettiva ribalta il modello, partendo dal basso ho Unit, Integration, System, Acceptance (cosa chiede l'utente)

Test di unità

- Lo scopo è rivelare malfunzionamenti ed acquisire confidenza, possono essere scritti anche prima del codice (Test Driven Develop), mentre l'adeguatezza è vista o in base al numero di funzionalità controllate, o in base a metriche di *'copertura'*.

Strategie per Integration Testing

- Innanzitutto si assume di operare in un modulo del SUT isolato, trovare soluzione ottima è problema NP completo. Esistono varie strategie:
- *big-bang*: testo quando ho tutti i moduli disponibili, quindi testo tardi perchè aspetto tutti.
- *top-down* : utile soprattutto per integrare interfacce attese/previste, usando funzionalità più generali. Se voglio che C1 (under-test) mandi msg a C2, e lavoro su C1 creo uno 'stub/mock' di C2, che sono come delle 'marionette' passive (non stimolano). Qui è C1 che chiama C2.
- *bottom-up* : qui parto dai bordi, non servono stub ma test driven (attivo) per lo stimolo, testo unità elementari. Parto dalle foglie e salgo. Qui è c2 che viene chiamata da C1.

Plugin Maven

- *Surefire Plugin*: per gli unit test, utile per lanciare test in un unico modulo maven (Junit sta 'sotto'). Le classi di test devono includere 'Test' nel nome, e i report sono nella cartella target
- *Failsafe Plugin*: utile per test su più progetti/moduli. Attivato con verify maven.

ADEGUACY CONTROL FLOW

COME SONO I MIEI TEST?

- Esistono approcci funzionali black-box e strutturali white-box.
- I *funzionali* richiedono documentazioni, che ci informa del comportamento di un'entità. Mi concentro su COSA deve essere fatto per stabilire l'adeguatezza, senza percorrere tutti i rami, inoltre sono sempre 'sotto' un determinato contesto/sistema, se cambiato il criterio perde valore.
- Gli *strutturali* sono basati sulla copertura del codice, non funzionalità, sono più automatizzabili, ma di minor interesse all'utente, non evidenzia funzioni mancanti.

CRITERI CONTROL FLOW

In generale ci preoccupiamo che i nostri test coprino i nodi del grafo risultante dal codice.

- *Statement coverage*:
  rapporto tra linee codice coinvolte in un test/(istruzioni totali - istruzioni non raggiungibili dal SUT).
- *Block coverage* :
  rapporto tra blocchi coinvolti in un test / (blocchi totali - blocchi irraggiungibili)
- *Decision coverage* :
  rapporto tra insieme degli if percorsi (per ogni alternativa) / (punti if totali - non raggiungibili).
- *Condition coverage*:
  insieme di condizioni validate (devo coprire ogni singola componente) / (totali - non raggiungibili dal sut). Vedo sia condizioni semplice T/F che complesse. A causa della *Lazy evalution* non uso AND ma OR

- La differenza tra gli ultimi due è che nel branch mi interessa il caso: *entro nell'if /non entro nell'if*, mentre nel condition mi interessa che ogni componente dell'if sia true e poi false (se poi non entro non mi interessa). Nessuna delle due implica l'altra.

- *branch condition coverage*:
  insieme delle precedenti (metto insieme numeratori e denominatori), rispetto punti di scelta E espressioni semplici/atomiche, insieme di condizioni e decisioni.

- *multiple condition coverage* :
  utile se mi interessano le condizioni complesse, senza esplorare totalmente, ho insieme condizioni validate/(combinazioni totali - non raggiungibili)

MC/DC

copro tutti i punti di scelta e tutte le espressioni evitando l'esplosione combinatoria degli stati, richiede che:

- ogni blocco sia coperto
- condizioni *semplici* sempre coperte da T e F
- ogni decisione è coperta per ogni ramo
- ogni condizione complessa nel SUT è composta da condizioni semplici e indipendenti.

Queste 3 condizioni implicano block coverage = condition coverage = branch decision coverage = 1 allora
MCDC = numero di condizioni semplici indipendenti/ (sommatoria di condizioni semplici tot - semplici non soddisfacibili)

DATA-FLOW

- Si basa sul concetto di <def,use>. Una variabile può essere usata in un predicato come if (p-use), o per computazioni (c-use).
- Ispezionando il grafo definiremo poi insieme DCU per c-use (coppia di nodi raggiungibile in modo computazionale), e l'insieme DPU per p-use (che opera per i predicati).
- Anche qui esistono metriche:
  C-USE coverage : c-use coperte da tutte le variabili / (dcu - c-use non raggiungibili).
  P-USE coverage : analogo

  Posso fare anche *all coverage* mettondo insieme numeratori e denominatori di entrambe

X-MEN

Per individuarlo deve essere:

- raggiungibile: esiste path di esecuzione che ci arriva
- infezione stato: M != SUT (ovvero il sut deve essere infettato per far esistere il mutante)
- propagazione stato: l'impatto del mutante si protrae fino alla fine dell'esecuzione considerata.

M = SUT se non esiste almeno un caso di test che rispetta le 3 condizioni. Esiste strong mutation (stessi risultati del SUT) e weak mutation (implica strong, presenta stessi stadi intermedi).
Ad esempio M porta stesso risultato di SUT, ma segue percorso diverso (strong ma non weak).

- Esiste anche qui mutation score:

mutanti rivelati/(mutanti vivi + morti)
oppure
mutanti rivelati/(tutti mutanti - equivalenti al sut) , più preciso ma più difficile.