

QUALITA DEL SOFTWARE

Sommario

INTRODUZIONE.....	1
FINITE MODELS.....	4
DEPENDANCE AND DATA FLOW ANALYSIS.....	6
DATA FLOW ANALYSIS	9
SYMBOLIC EXECUTION.....	9
GENERALIZED SYMBOLIC EXECUTION & DYNAMIC SYMBOLIC EXECUTION	11
STRUCTURAL TESTING AND DATA FLOW TESTING.....	14
MODEL BASED TESTING.....	17
FAULT BASED TESTING AND MUTATION ANALYSIS.....	21
TEST EXECUTION.....	23
PLANNING AND MONITORING	25
INTEGRATION, SYSTEM AND REGRESSION TESTING	26

INTRODUZIONE

Nel corso verranno trattati dei metodi per ricavare informazioni di un programma analizzandone la struttura come:

- Test case selection
- Adequacy criteria
- Structural testing
- Model based testing
- Data flow testing
- Fault based testing

Andando ad eseguire questi test possiamo ricavare informazioni sulla bontà di un programma.

Il primo assignment riguarda la lettura del report del 2017 e fare un piccolo riassunto dei suoi punti chiave.

La qualità del software è importante in quanto un software di bassa qualità può portare a perdite finanziarie consistenti, la presenza di bug software può causare errori critici durante l'utilizzo dello stesso e infine questi errori in alcuni contesti possono portare al rischio della vita di qualcuno.

Eseguire i test e le analisi è un processo complicato in quanto:

- Data l'impossibilità di verificare la correttezza di ogni segmento di codice è necessario individuare tecniche di analisi e testing corrette per avere un'idea globale del funzionamento del software
- La correttezza di un software spesso è indecidibile ovvero che non si può determinare una correttezza assoluta

Il libro del professore non va scaricato ma il prof ti fornisce le slide degli spezzoni che devi studiare.

Puoi mancare 2 consegne di assignment e avere un massimo di 10% di assenze alle lezioni.

Ci sono degli homework che vengono discussi in incontri serali online dalle 17:30-18:30.

Quindi da fare ci sono assignment da consegnare in tempo e homework da consegnare e discutere online.

In generale ogni argomento viene presentato con una **lezione breve da mezz'ora che riassume il contenuto del libro**. Il PDF del capitolo del libro usato viene caricato con la lezione. Poi vengono dati dei materiali extra di lavoro come **esercizi che vengono eseguiti e consegnati generalmente nella settimana stessa della lezione**. A fine settimana c'è una lezione online dove ci si focalizza sugli esercizi; ovviamente questa lezione avviene prima della consegna degli esercizi stessi e serve per chiarire dubbi sullo svolgimento degli stessi.

ATTENZIONE GLI ASSIGNMENT SONO VALUTATI MENTRE GLI HOMEWORK NON SONO VALUTATI.

ATTENZIONE GLI ASSIGNMENT VANNO FATTI IN INGLESE!

A FRAMEWORK FOR TEST AND ANALYSIS – CAPITOLO DEL LIBRO

L'obiettivo dell'analisi e test del software è sia di **valutare le qualità del software, sia migliorarne le capacità individuandone dei difetti**. Non esistono tecniche perfette per il testing e analisi del software, ma esistono tecniche che con dei trade-off vanno a specializzarsi in determinati tipi di analisi o testing.

Dare una **valutazione a come un software esegua correttamente le richieste dell'utente viene definita validazione**. Dato che le richieste sono scritte da persone, è possibile che queste contengano errori, quindi un sistema rispetta la richiesta correttamente se riesce a produrre un output desiderato dall'utente nonostante non riceva in ingresso un input impreciso. **La validazione quindi va a confrontare le specifiche iniziali con il prodotto finale per identificare alcuni errori in fase di sviluppo del software e controllare se il software rispetti le esigenze dell'utente**, tuttavia per quanto riguarda quest'ultimo controllo si può dire che per la validazione sia necessario un giudizio umano per ottenere tale scopo.

La **verifica** d'altro canto va a **controllare la consistenza di una implementazione con una specifica**. In particolare la **specifica** può essere identificata come un **design generico** mentre un'**implementazione** come un **design specifico**. La verifica quindi svolge la funzione di controllo di consistenza tra le due descrizioni a differenza della validazione che confronta una descrizione con le effettive necessità dell'utente.

Per quanto si voglia cercare di definire precisamente se un programma sia in grado di soddisfare delle **proprietà specifiche o meno, questa operazione non è possibile**. Se per sistemi estremamente semplici è possibile determinare il comportamento dello stesso in ogni situazione possibile, per la maggior parte dei sistemi questo non è possibile, lasciando quindi sempre un margine di potenziale errore che non può essere verificato da nessun tipo di tecnica. **Occorre quindi eseguire delle approssimazioni** da testare per avere una verifica approssimata delle proprietà del sistema.

Una **tecnica per verificare una proprietà** può essere inaccurata/approssimata in due modi:

- **Inaccuratezza/approssimazione pessimistica**: non è garantito di accettare un programma anche se il programma possiede le proprietà analizzate. In questo caso gli algoritmi vanno ad astrarre le proprietà del software, come ad esempio la Data flow analysis, per poi analizzarle e cercare di individuare delle proprietà generali del sistema.
- **Inaccuratezza/approssimazione ottimistica**: possono essere accettati anche dei programmi che non possiedono le proprietà analizzate. **Si intende che una volta scelti degli input in maniera ponderata ed eseguito un testing del sistema sugli stessi, se il sistema non produce errore su tali input allora viene messo in produzione.**

In sostanza il **primo approccio** cerca di **osservare l'intero sistema andando a generalizzare tramite delle astrazioni le sue funzioni per individuarne delle proprietà generali**, mentre col **secondo approccio** si crea un **sottoinsieme di esecuzioni che verrà testato** e se il risultato è positivo allora si reputa l'intero sistema come corretto.

Un terzo approccio possibile nell'eseguire il testing e analisi di un sistema consiste **nell'andare a semplificare la proprietà che si vuole analizzare oltre il livello di semplificazione ottenuto dall'approccio pessimistico**.

In generale si può dire che la scelta migliore sia quella di **bilanciare** tecniche di verifica ottimistiche e pessimistiche in modo tale da ottenere risultati accettabili e ridurre i problemi.

SOFTWARE FAILURE CAUSED \$1.7 TRILION IN FINANCIAL LOSSES – APPROFONDIMENTO

Le tecniche di sviluppo del software sono cambiate drasticamente negli ultimi anni, tuttavia non si può dire lo stesso per la fase di testing del software. I report di analisti evidenziano come **le metodologie di testing dei software non riescono a mantenere il passo con quelle di sviluppo** e soprattutto come, a causa di questa disparità, il numero di bug rilasciati con i software stia continuamente incrementando.

Questo articolo ha come obiettivo quello di mettere in luce come l'utilizzo di software sia radicato nella nostra quotidianità e come il malfunzionamento degli stessi causi disagi non solo agli utenti ma anche cospicue perdite finanziarie ai proprietari dell'applicativo.

Le statistiche riportate da questo articolo mostrano come tramite un conteggio conservativo **il numero di persone afflitte da problemi di software failure sono 3.7 su 7.4 bilioni**. Le stime inoltre suggeriscono che le **perdite finanziarie** causate da queste failure nel 2017 sono pari a **circa 1,715,430,778,504 di dollari**. Infine tali problematiche hanno causato dei **ritardi nel rilascio di prodotti o delle necessarie manutenzioni agli stessi per un totale di più di 268 anni**.

Le industrie maggiormente affette da questa problematica sono i servizi pubblici, la sanità, le aziende di vendita al dettaglio e più in generale quelle aziende che fanno uso di applicativi mobile. Nel 2017 per queste tipologie di aziende si è vista un'impennata nel numero di software failures che però è stata ridimensionata negli ultimi mesi dell'anno. Ovviamente la maggior parte degli articoli riguardanti questi errori riguardano applicativi pubblici che hanno comportato la perdita di dati personali degli utenti in caso di hacking.

Le principali tipologie di software affette dalla software failure sono gli applicativi embedded come ad esempio quei software che gestiscono il sensore dell'airbag di una macchina, **i software mobile o web** e infine le **applicazioni enterprise** ovvero gli applicativi aziendali.

Le principali categorie di errore in cui le software failures sono riconducibili sono i software bugs, le vulnerabilità di sicurezza e i glitch di usabilità. I primi definiscono una situazione in cui il software non funziona come programmato, le vulnerabilità evidenziano potenziali rischi di hacking, mentre gli ultimi sono dei problemi che rendono difficoltoso l'utilizzo dell'applicazione ma non ne bloccano completamente il funzionamento.

Tra i vari disagi causati al pubblico dalle software failures abbiamo:

- **Fallimenti di software pubblici** che causano **perdita di dati personali** di utenti e disservizi in campo sanitario, economico e politico
- **Fallimenti di software di intrattenimento** come applicativi social e giochi

- **Fallimenti di software di trasporto** causando disagi di ritardi, annullamenti o addirittura incidenti mortali
- **Fallimenti di software di telefonia e web**
- **Fallimenti di software cloud** come i servizi di cloud storage che impossibilitano l'accesso ai dati agli utenti per un certo tempo o la perdita definitiva degli stessi

Oltre ai danni finanziari di un'azienda che sono facilmente calcolabili abbiamo anche dei danni di immagine ad un brand che sono difficilmente quantificabili. L'articolo struttura un **Brand Erosion Index** in grado di eseguire una **stima del danno di immagine** basandosi su una ricerca di articoli che riferiscono la presenza di un software fail per un determinato brand. **In generale i brand più afflitti di danno all'immagine sono le compagnie che vendono prodotti tech** come smartphone ad esempio.

FINITE MODELS

I Finite Models sono molto utilizzati nella software engineering nelle fasi di testing e analisi.

Le caratteristiche che i models dovrebbero avere sono:

- **Compattezza:** i modelli devono essere ragionevolmente piccoli sulla base dei compiti a loro affidati
- **Predittivi:** i modelli devono essere in grado di catturare alcune caratteristiche che saranno utili per il software
- **Semanticamente significativi:** i modelli devono fornire informazioni significative riguardo il sistema
- **Sufficientemente generali:** i modelli devono essere applicabili ad un insieme di sistemi e non ad un singolo

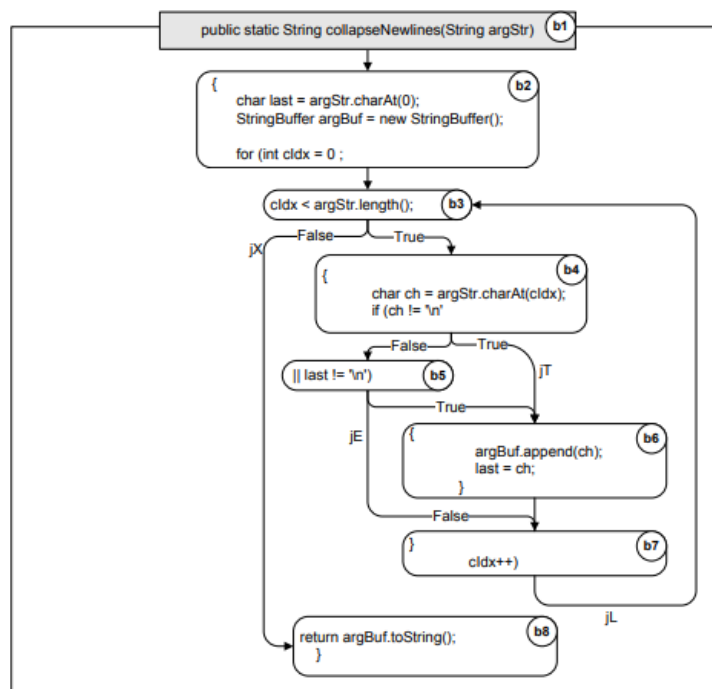
I modelli creano **un'astrazione del sistema**, ovvero che omettono alcuni dettagli del sistema e rappresentano il comportamento generico dello stesso. Le tecniche di astrazione sono la **soppressione dei dettagli e il compattare gli stati**, ovvero considerare solo una parte dei vari dettagli che costituiscono gli stati di un sistema in modo tale da ridurre potenzialmente anche il numero di stati stesso in quanto considerare meno dettagli potrebbe rendere uguali degli stati.

Tra i modelli più famosi per l'analisi e il testing abbiamo la **Intraprocedural Control Flow Graph**. L'idea alla base è quella di **mappare ogni singola istruzione o blocchi di istruzioni in degli stati che possono essere attraversati**. Quindi ad esempio se abbiamo una IF avremo due stati che partono dalla stessa che identificano lo stato in cui la IF è TRUE e lo stato in cui la IF è FALSE. Segue un **esempio di utilizzo** di per rappresentare del codice:

```

7  public static String collapseNewlines(String argStr)
8  {
9      char last = argStr.charAt(0);
10     StringBuffer argBuf = new StringBuffer();
11
12     for (int cldx = 0 ; cldx < argStr.length(); cldx++)
13     {
14         char ch = argStr.charAt(cldx);
15         if (ch != '\n' || last != '\n')
16         {
17             argBuf.append(ch);
18             last = ch;
19         }
20     }
21
22     return argBuf.toString();
23 }

```



Un altro modello è la **Linear Code Sequence and Jump (LCSJ)**. Il modello rappresenta ogni sequenza di codice del programma con un nodo e ogni salto che collega un nodo ad un altro con un arco. In pratica quindi la sequenza di codice identifica tutti quegli stati che possono essere attraversati senza fare una decisione. Quando bisogna fare una decisione (ad esempio con una IF) allora si avrà un salto. Segue un esempio di rappresentazione LCSAJ del codice precedente:

From	Sequence of Basic Blocks							To
entry	b1	b2	b3					jX
entry	b1	b2	b3	b4				jT
entry	b1	b2	b3	b4	b5			jE
entry	b1	b2	b3	b4	b5	b6	b7	jL
jX								b8 return
jL			b3	b4				jT
jL			b3	b4	b5			jE
jL			b3	b4	b5	b6	b7	jL

Entrambi questi modelli eseguono operazioni all'interno di una singola procedura del sistema. Ma spesso le procedure coinvolte in un sistema sono multiple. In questo caso dobbiamo usare un **Inter procedural control flow graph**. In questa rappresentazione i nodi sono le classi e le funzioni e gli archi sono le chiamate che possono essere fatte. Questo tipo di modello può essere context sensitive o insensitive; nel primo caso se ci fossero più funzioni o classi che chiamano una funzione non abbiamo idea di quale classe abbia chiamato quella che stiamo considerando, nel caso sensitive invece ci sono rappresentazioni multiple dello stesso stato che identificano da quale nodo questo stato sia stato invocato. Ovviamente i modelli context sensitive sono più precisi ma computazionalmente i costi scalano esponenzialmente rispetto ad una rappresentazione insensitive.

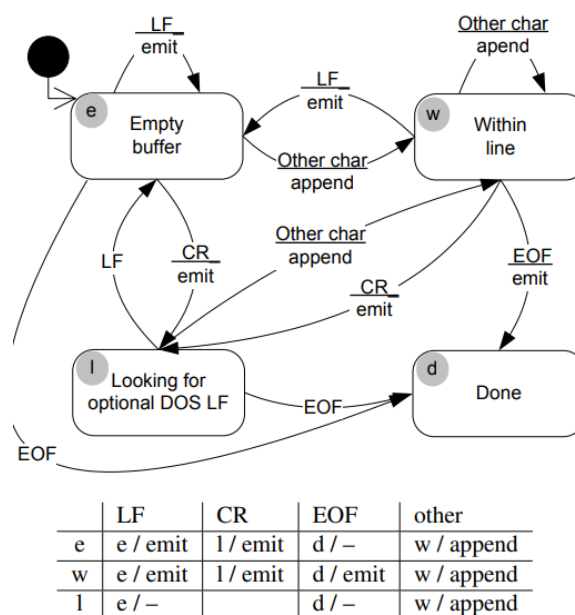
Un altro modello è la **Finite States Machines**. La rappresentazione è simile alle precedenti solo che in questo caso abbiamo i nodi che rappresentano gli stati e gli archi che rappresentano le transizioni da uno stato ad un altro.

L'utilizzo di questi modelli è vario e uno degli scopi principali del loro uso è ragionare sulle proprietà del sistema. Il modello crea un'astrazione di un programma che cerca di verificare una determinata proprietà. Se posso verificare che l'astrazione del programma rappresenti correttamente le sue proprietà allora posso dire che se la proprietà è valida nel modello di astrazione allora questa lo sarà anche nel modello originale.

FINITE MODELS – CAPITOLO DEL LIBRO

L'utilizzo dei modelli consente di **eseguire operazioni di analisi e testing prima che l'intero sviluppo del software sia finito** e durante tutta la sua fase di sviluppo. Un modello è una rappresentazione semplificata del sistema che ne approssima gli attributi principali.

I **Finite State Machine (FSM)** sono dei modelli che vengono generalmente costruiti prima del codice sorgente che li identifica. La rappresentazione di questi modelli consiste nell'individuare degli stati in cui il sistema può vertere e degli archi che collegano i vari stati che definiscono le transizioni tra uno stato e un altro. Di **solito gli archi vengono etichettati per indicare quale sia l'operazione, condizione o evento che ha causato la transizione da uno stato ad un altro.**



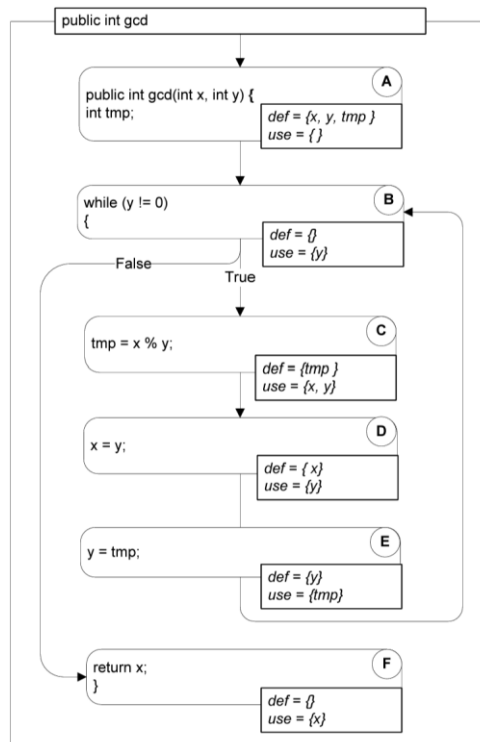
Oltre alla rappresentazione con nodi e archi, la FSM può essere **rappresentata con una tabella delle transizioni degli stati**. Nella tabella appena mostrata si ha nelle righe ogni stato del sistema e nelle colonne tutti gli eventi di input che possono causare una transizione da uno stato ad un altro.

DEPENDANCE AND DATA FLOW ANALYSIS

DEPENDENCE AND DATA FLOW MODELS – CAPITOLO DEL LIBRO

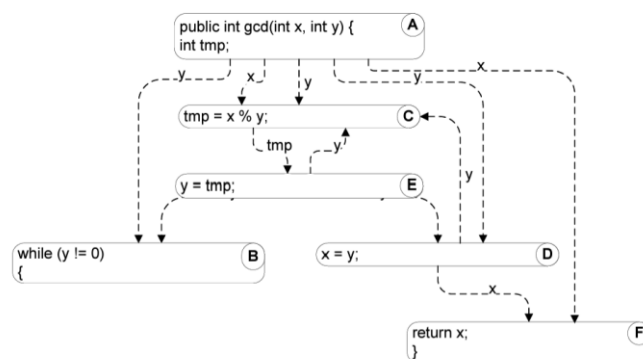
I data models sono molto utilizzati nella software engineering. L'idea che sta sotto i data flow models è quella di creare un modello che rappresenta le dipendenze all'interno di un sistema, come ad esempio le dipendenze tra due o più statement/pezzi di codice.

Un primo esempio è quello delle **Definition-Use Pairs**. Con **definizione** si intende quel pezzo di codice dove si va ad assegnare un valore ad una variabile (es. $X = 1$), mentre con **uso** si intende l'utilizzo di una variabile precedentemente definita (es. $Y = X + 1$); il **percorso** che porta dalla definizione di una variabile al suo utilizzo viene definito **Def-Use path**. In generale dato un pezzo di codice possiamo dire che il suo **control graph** è composto da dei nodi che possiedono due set distinti: il set di definition e il set degli uses.

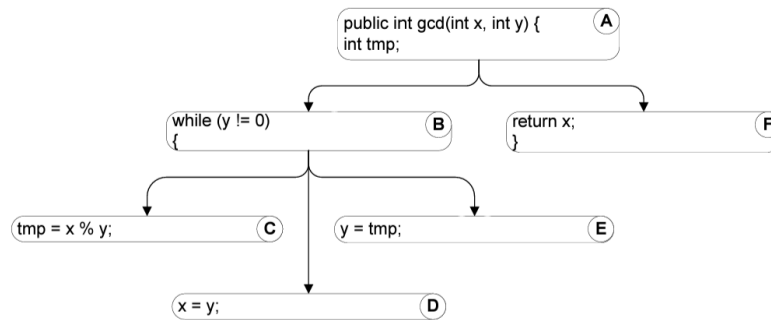


Il **Definition-Clear o Killing** indica una **ridefinizione di una variabile** che va a sovrascrivere un valore precedentemente assegnato ad una variabile.

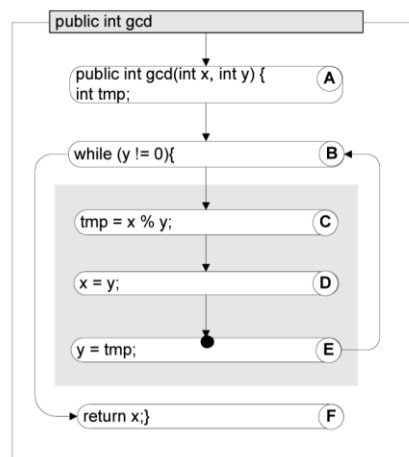
Dal concetto di definition e use possiamo elaborare un nuovo tipo di grafico, ovvero il **Data Dependence Graph**. In questo grafo i **nodi corrispondono agli statement** del codice e gli **archi collegano i nodi tra di loro** se esiste una data dependence tra questi, ovvero **se all'interno di un nodo c'è l'uso di una variabile definita nell'altro nodo**. Se la variabile viene ridefinita in un successivo statement, allora **anche questo statement** sarà collegato con un arco a tutti gli altri statement che fanno uso di tale variabile.



La data dependence è diversa dalla control dependence, infatti i Control Dependence Graph hanno come nodi gli statement del codice, ma gli archi definiscono un collegamento tra uno statement e il codice che può essere eseguito immediatamente dopo. **DA RIVERDERE**



La data e control dependence introducono il concetto di **Dominator**. Il Dominator è il nodo che deve essere attraversato per raggiungere un altro nodo. Quindi dato un Control Dependence Graph possiamo individuare delle relazioni di dominazione. Nella seguente immagine possiamo vedere come per raggiungere il nodo E sia necessario passare prima per i nodi A e B, questo indica che E ha una relazione di control dependance con A e B.



Calcolare le Def-Use Pairs vorrebbe dire prendere per ogni nodo del control graph tutti i possibili path andando ad identificare le definition e uses utilizzati e infine si identificano tutti i definition use path nel grafo; tutto ciò in un pezzo di codice articolato risulta essere impossibile perché il calcolo diverrebbe esponenzialmente complesso. Esiste tuttavia un algoritmo in grado di eseguire questo calcolo chiamata **Reach Equation**. Questo algoritmo dato un nodo X del grafo che si sceglie va ad identificare quali nodi possono raggiungere il nodo X e quali nodi possono essere raggiunti dal nodo X. Se una variabile venisse ridefinita (ovvero killata) nel nodo X oppure se tale variabile venisse definita nel nodo X, allora nel calcolo dei nodi che possono raggiungere X bisognerà togliere quelle definizioni di variabili.

Questa computazione può essere utilizzata per raggiungere una fixed point computation, ovvero definire in generale la Reach di un nodo X (ovvero i nodi che possono raggiungere un nodo) come l'unione della Reach Out di tutti i predecessori di tale nodo (ovvero i nodi che possono raggiungere i nodi che precedono il nodo X obiettivo). Per quanto riguarda la Reach Out del nodo X, questa viene calcolata come la Reach del nodo X alla quale sono state sottratte tutte le definizioni che sono state Killate dal nodo X e tutte le definizioni che sono generate nel nodo X.

In conclusione questo algoritmo è molto utile perché riduce notevolmente lo spazio di codice da analizzare andando a prendere solo i segmenti in cui una variabile viene definita e utilizzata senza essere killata.

Un altro tipo di equazione è la **Avail Equation**. Questa equazione è simile alla precedente solo che la Avail di un nodo è definita come l'intersezione delle Avail Out dei nodi che la precedono. L'avail out quindi è

costituita da ciò che è available prima del nodo X andando ad eliminare tutte le variabili killate e create dal nodo stesso.

La differenza quindi tra Reach e Avail equation è che la prima tiene in considerazione quello che succede per almeno uno dei predecessori del nodo preso in considerazione, mentre la seconda tiene in considerazione quello che occorre almeno una volta in tutti i nodi precedenti al nodo preso in considerazione.

L'ultima equazione considerata è la **Live Variabile equation**. La Live di questa equazione è una **unione della LiveOut del nodo SUCCESSIVO a quello preso in considerazione**.

Andando quindi a classificare i tipi di analisi possibili appena mostrati possiamo creare la seguente tabella:

NOTE DI LEZIONI ESERCIZI

Quando si fa un live analysis che è per forza una backward any-path analysis, la gen è l'utilizzo di una variabile che è stata definita precedentemente mentre la kill è la riassegnazione del valore di una variabile che poteva ad esempio non esistere prima oppure esistere prima ma si va a sovrascrivere il valore. Le analisi di tipo anypath vanno sempre ad inizializzare un insieme vuoto di live e liveout, mentre le analisi di tipo allpath l'insieme iniziale conterrà tutte le variabili gen. Per le analisi live quando si vanno a calcolare i vari passaggi si parte dal fondo del codice. La live di una riga di codice è composta dall'unione di tutti i successori (next statement nell'esercizio) ma nel caso di un'analisi live dato che si parte dal fondo i successori non ci sono per il primo record preso in considerazione. Per calcolare il liveout invece si prendono gli elementi in live e si tolgono le kill e le gen. **In generale possiamo dire che la live è calcolata come l'unione delle liveout del successore dello statement preso in considerazione, mentre la liveout si calcola come la live – le kill dello statement preso in considerazione + le gen di questo statement.** I vari pass1 e pass2 da fare sono fatti nella stessa identica maniera con l'unica differenza che nei passaggi successivi al primo si va a prendere la liveout dal passaggio precedente.

Per la availability le gen vengono create ogni volta che c'è un'espressione (sia quello che sta a destra dell'=', sia quello che viene usato nei for, if, ecc) mentre la kill va ad applicarsi quando si va ad usare la variabile che è stata riassegnata.

DATA FLOW ANALYSIS

SALTATA IN QUANTO BESTIA DI SATANA. ASSIGNMENT IMPOSSIBILE FATTO ALLA CAZZO. DA RECUPERARE PIÙ AVANTI. IL CONTENUTO È LO STESSO DEL CAPITOLO PRECEDENTE MA CON QUALCHE AGGIUNTA.

SYMBOLIC EXECUTION

SYMBOLIC EXECUTION – VIDEOLEZIONE

La **Symbolic execution** è un concetto semplice ma molto utilizzato per l'analisi delle applicazioni attualmente. In sintesi la funzione della symbolic execution è quella di **eseguire un programma utilizzando simboli al posto dei valori**.

Per le funzioni semplici ad esempio $x = y + z$ dove $y=1$ e $z=3$ possiamo semplicemente sostituire i valori di y e z con dei simboli tipo low, high e, infine il valore di x viene determinato dall'espressione low + high.

Per blocchi di programma più complessi il discorso cambia; dato il seguente codice:

low = 0

high = $\frac{H-1}{2} - 1$

mid = $\frac{H-1}{2}$

```
while(high>=low){
mid=(high+low)/2;
```

In questo caso abbiamo un ciclo while che potrebbe complicare le cose; analizzando il codice possiamo dire che lo statement che fa entrare o meno nel ciclo while non va a modificare il valore delle variabili precedentemente analizzate in alcun modo, tuttavia lo **statement del while può essere true o false**. In questo caso quindi avremo una **diramazione che prende in considerazione le due eventualità** ottenendo quindi: $(H-1)/2-1 \geq 0$ nel caso di true e $\text{not}((H-1)/2-1 \geq 0)$ nel caso di false. **I percorsi che portano all'esecuzione di una delle due diramazioni viene chiamato path expressions.**

Spesso **le condizioni imposte dal codice possono essere semplificate** utilizzando delle weaker condition come segue:

low = L; high = H; mid = M

$M = (L+H)/2$

Questa condizione completa può essere semplificata andando a definire M in maniera più debole ma sempre corretta rispetto alla versione completa originale:

$L \leq M \leq H$

Le path execution così create possono essere utilizzate per verificare un pezzo di codice. Per fare ciò si utilizzano le **pre e post conditions**. Per verificare un pezzo di codice devo dimostrare che il programma soddisfa le sue specificazioni che sono date sia dalle precondition che dalle postcondition. **Le prime sono delle asserzioni fatte all'inizio del codice che impongono le condizioni che devono essere vere affinché si abbia un'esecuzione del codice, mentre le seconde sono delle asserzioni che si presentano alla fine del blocco di codice che identificano cosa deve essere vero alla fine dell'esecuzione del blocco di codice.**

Per utilizzare pre e post condition in congiunta alla symbolic execution per verificare la correttezza di un programma usiamo il seguente codice:

```
char * binarySearch( char *key, char *dictKeys[ ], char *dictValues[ ],
                    int dictSize) {

    int low = 0;
    int high = dictSize - 1;
    int mid;
    int comparison;

    while (high >= low) {
        mid = (high + low) / 2;
        comparison = strcmp( dictKeys[mid], key );
        if (comparison < 0) {
            /* dictKeys[mid] too small; look higher */
            low = mid + 1;
        } else if ( comparison > 0 ) {
            /* dictKeys[mid] too large; look lower */
            high = mid - 1;
        } else {
            /* found */
            return dictValues[mid];
        }
    }
    return 0; /* null means not found */
}
```

Vediamo come le **precondition** del programma siano che **innanzitutto nell'input l'array dictKeys sia ordinato**, nel senso che i valori al suo interno siano messi in ordine crescente. La **postcondition** che può essere chiamata anche **invariant in quanto è un pezzo di codice che deve essere vero la possiamo identificare con il loop while**. L'invariant dice che, **a prescindere da quello che viene fatto durante la computazione, il valore dell'elemento key preso dall'array dictKeys deve essere compreso tra il valore di low e high**. Una volta definite precondition e postcondition possiamo passare ad una **esecuzione simbolica del frammento di codice $mid = (high + low)/2$** . Una volta fatto **avremo computato un nuovo stato** del quale dobbiamo **verificare se soddisfa l'invariant**. In conclusione per verificare la correttezza di un programma dobbiamo andare a verificare che la precondition sia soddisfatta, l'invariant definita dal loop while sia soddisfatta e che all'uscita del loop anche le postcondition siano soddisfatte.

GENERALIZED SYMBOLIC EXECUTION & DYNAMIC SYMBOLIC EXECUTION

GENERALIZED & DYNAMIC SYMBOLIC EXECUTION – VIDEOLEZIONE

I due argomenti toccati saranno l'esecuzione simbolica generalizzata che, come visto nel capitolo precedente, tiene in considerazione le reference come parte dello spazio di input del programma e la esecuzione simbolica dinamica. Iniziando dalla **esecuzione simbolica generalizzata con delle reference in input possiamo dire che l'idea di base sia che** gli input non siano semplicemente dei valori interi che possiamo rappresentare con dei simboli, ma che **il programma riceva in input una struttura dati** che sarà stata allocata nello heap. **Le path condition del programma che stiamo analizzando devono tenere conto delle varie forme delle strutture dati in input** (vedi esercizio dove possiamo avere una lista vuota o diversi elementi).

Prendiamo in considerazione il seguente codice:

```
void toAnalyze(Node n) {  
  if (n.first != null)  
    n.first.next.traverse(); ... }  
  
public class Node {  
  //represents a tree  
  private int info;  
  private Node first; //children  
  private Node next; //sibling  
  /* methods, e.g., constructor, addChild, addSibling, ... */ }
```

Il codice crea una struttura di un albero non binario dove **ogni nodo ha un figlio che a sua volta può avere diversi fratelli**. Possiamo notare quindi che **l'input di questo programma è una reference** in quanto si va a ricevere diversi nodi in ingresso.

Partendo quindi con l'esecuzione simbolica consideriamo che **i nodi in input** sia un input numerico e quindi lo **identifichiamo con un simbolo n che rappresenta tutti i valori possibili della reference: $n = n_0 \rightarrow ?$**

Con questa rappresentazione **se durante l'esecuzione tale input non venisse mai utilizzato, allora questo simbolo sarebbe già sufficiente per rappresentare l'esecuzione** del programma dato che i valori della reference non vengono coinvolti nell'esecuzione.

Nel nostro caso tuttavia abbiamo un accesso ad n subito alla seconda riga di codice. Al momento dell'accesso ad una variabile reference dobbiamo identificare quali siano tutti i suoi possibili valori per cui il programma avrebbe dei comportamenti diversi. In questo caso i valori possibili di cui dobbiamo tenere conto sono: $n = \text{NULL}$, $n.\text{first} = \text{NULL}$, $n.\text{first} \neq \text{NULL}$.

Per quanto riguarda il primo valore di cui tenere conto possiamo dire che quando si chiama il metodo $n.\text{first}$ dobbiamo identificare quali siano i possibili valori di n prima di considerare l'intera chiamata al metodo. In generale si vanno ad identificare tre possibilità, ovvero: $n = \text{NULL}$, n è un alias di una reference precedentemente utilizzata ($n_0 = n_1$), n sia una reference che punta ad un oggetto da analizzare. Dato che è la prima reference che stiamo prendendo in considerazione abbiamo solo le possibilità in cui n sia null oppure sia una reference da analizzare.

Possiamo ora andare ad analizzare $n.\text{first}$ e i suoi possibili valori: null, alias di se stesso (quello con il loop nel grafico), puntatore ad un'altra reference. Infine andando a considerare la terza riga di codice dobbiamo prendere in considerazione i possibili valori di next allo stesso modo eseguito per i precedenti.

Abbiamo quindi in generale 4 forme possibili di input che portano a diverse esecuzioni ciascuno.

Confrontando queste esecuzioni rispetto all'obiettivo del programma possiamo dire che il caso in cui abbiamo

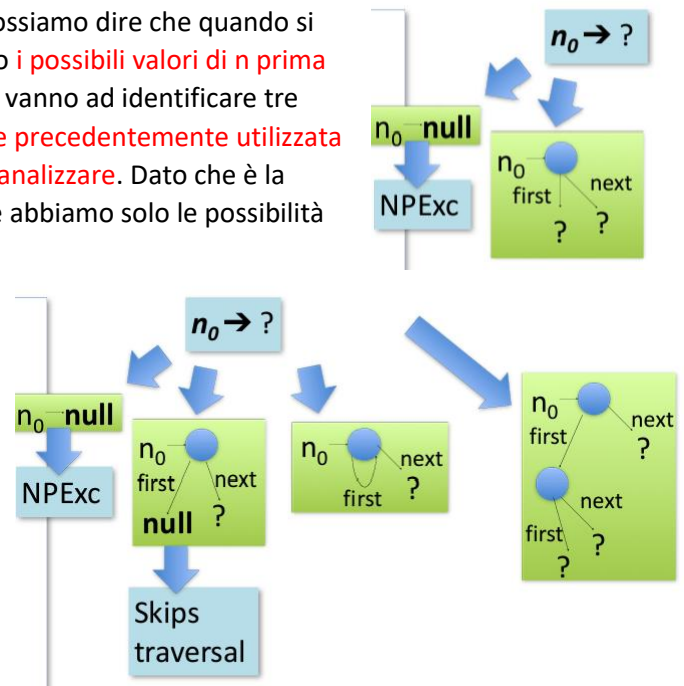
il loop identificerebbe un nodo che punta a sé stesso come nodo successivo, il che non coincide con una struttura ad albero. In questo caso se volessi creare un esempio con degli input precisi però non sarei in grado di ottenere un albero di questo tipo, quindi possiamo dire che questa incongruenza è in realtà un falso allarme. È importante quindi andare ad analizzare le precondizioni del programma affinché non si creino dei falsi allarmi durante l'esecuzione simbolica che rendono l'analisi sbagliata.

Passiamo ad un esempio sulla Dynamic Symbolic Execution. L'esecuzione simbolica permette di analizzare il comportamento di un programma in base al suo spazio di input. L'obiettivo di quest'analisi è quello di o derivare delle proprietà andando ad analizzare le possibili esecuzioni del programma sulla base del suo input verificando che tali proprietà siano soddisfatte per ogni cammino del programma oppure analizzare i vari cammini del programma ed identificare degli input concreti che portino all'esecuzione di quei cammini. Come nell'esempio precedente partiamo da un codice:

```
int f(int x, int y) {
    if (x < 0) {
        x = abs(x);
    } else if (y < 0) {
        y = abs(y);
    }

    int k = 0, max = 0;
    if (x > y) {
        k = x - y; max = x;
    } else {
        k = y - x; max = y;
    }
    int[] a = new int[max + 10];

    if (k >= max + 10) {
        abort("Buffer overflow");
    }
    a[k] = 0;
    return a[0];
}
```



In questo caso nello spazio di input abbiamo solo dei valori interi, ma in realtà è possibile che lo spazio di input sia come quello dell'esempio precedente ovvero delle reference di oggetti. Il nostro obiettivo andando ad eseguire una DSE è quello di **generare degli input X e Y che permetta di eseguire ognuno dei cammini del programma.**

Se dovessimo in primis **identificare quanti cammini dobbiamo andare ad analizzare** in questo caso il calcolo è semplice; **basta vedere quanti if/else abbiamo** e contare 2 cammini per ognuno di essi, prestando attenzione però al primo if che in realtà ha 3 cammini in quanto se $x=0$ allora non si entra né nell'if né nell'else if. **Una volta contati tutti i cammini dobbiamo moltiplicare i numeri tra di loro ottenendo $3 \times 2 \times 2 = 12$.** Dei cammini così identificati **alcuni di questi potrebbero essere infeasible, ovvero che non esistono degli input che consentono di percorrerli.**

La DSE guida la sua struttura utilizzando i casi di test concreti. Il primo caso di test che utilizziamo è un caso in cui si considera una generazione randomica di X e Y ottenendo $X = 1234$ e $Y = -567$. In questo caso l'esecuzione sarebbe: $x>0$ quindi non entro nel primo if, $y<0$ quindi entro nell'else if, $x>y$ quindi entro nell'if successivo, $k < \max + 10$ quindi non entro nell'if. **Ora possiamo calcolare la path condition del cammino che abbiamo appena percorso, ovvero:**

```
path condition =  
X >= 0 && Y < 0 &&  
X > -Y && X + Y < X + 10
```

Notare come abbiamo $X > -Y$ dato che eseguendo $\text{abs}(y)$ andiamo ad invertire di segno Y (IMPORTANTE PER ESERCIZI).

Una volta calcolata la path condition dobbiamo chiederci se questa sia o meno feasible. Essendo però partiti da un caso di test sappiamo già che il cammino è feasible.

Sappiamo che ci sono altri cammini da analizzare e ora possiamo **farci guidare dal caso di test appena analizzato nello scegliere quale sia il prossimo cammino da analizzare.** Risulta piuttosto semplice dire che **possiamo scegliere tutti quei cammini che eseguono una decisione diversa quando raggiungiamo i branch (ovvero gli if) e per fare ciò posso in maniera molto semplice calcolare le loro path condition andando a modificare la path condition che ho identificato in precedenza.** Per ottenere questo risultato **basta andare ad invertire una delle condizioni che ho nella path condition precedente; considerando quindi le 4 possibili variazioni delle 4 condizioni si ottiene:**

```
alternative (sub-)paths:  
1) X < 0  
2) X >= 0 && Y >= 0  
3) X >= 0 && Y < 0 &&  
   X <= -Y  
4) X >= 0 && Y < 0 &&  
   X > -Y && Y >= 10
```

Possiamo notare come nel primo caso se $X<0$ allora si entra in un cammino totalmente nuovo e non percorso, nel secondo caso invece abbiamo modificato $Y \geq 0$ quindi abbiamo un cammino che sappiamo inizia con $X>0$ e poi quando arriva ad $Y \geq 0$ non sappiamo cosa fa poi. Allo stesso modo il terzo e quarto caso viene strutturato analogamente.

Se ora generassimo dei casi di test che rispettano queste 4 condizioni allora avremmo 4 cammini differenti da percorrere e analizzare. In questo caso però sarà necessario invocare il solver per verificare che siano dei **cammini feasible.** Notiamo quindi come la condizione 4 non sia percorribile dato che $Y<0$ non ammette che $Y \geq 10$.

Una volta fatto ciò basterà creare dei casi di test che soddisfino le condizioni dei path alternativi e iterare il processo precedente per andare ad individuare le nuove path condition del cammino selezionato. Ovviamente quando iteriamo il processo non andiamo a definire delle variazioni di condizioni che ci ricondurrebbero a quelle già definite in precedenza.

STRUCTURAL TESTING AND DATA FLOW TESTING

STRUCTURAL TESTING AND DATA FLOW TESTING – VIDEOLEZIONE

I criteri utilizzati dalla structural e data flow testing si basano sul codice, ovvero che si basano sull'utilizzo della struttura del codice oppure della struttura dei dati. La structural code-based testing si fa in quanto identifica cosa manca nella test suite del codice ed è complementare al functional testing; questi sono complementari in quanto ci sono dei casi che solo la functional testing può identificare e casi che solo la structural testing può identificare.

I più importanti structural testing criteria sono vari, il primo è la **Statement testing**. Questo test prevede che ogni statement (nodo nel grafico CFG) sia eseguito almeno una volta. Esiste una misura di coverage che divide il numero di statement eseguiti per il numero totale di statement nel programma. Per dare un esempio quando si fa questo tipo di testing in generale si ha un programma e il suo CFG, si ha anche in input un set di valori di test e sulla base di questi input si va a verificare quanti nodi del grafo CFG vengono esplorati dall'input. Infine si fa una media tra i nodi esplorati e i nodi totali.

Un altro structural testing criteria è il **Branch testing**. Questo test prevede che ogni ramo del grafico CFG venga eseguito almeno una volta. Anche in questo caso esiste una misura di coverage che divide il numero di branch eseguiti con il numero di branch totali. Allo stesso modo dello statement testing avremo degli input dei quali dovremmo valutare quanti rami attraversano del grafico CFG per poi stimarne la media. Sappiamo che attraversare tutti i rami implica necessariamente attraversare tutti i nodi ma non il viceversa. Attraversare tutti i rami non implica necessariamente che vengano esplorate tutte le possibili condizioni, basti pensare alla presenza di un OR che viene soddisfatto solo nella prima condizione e mai nella seconda; in questo caso il ramo viene attraversato ma viene esplorata solo una condizione dell'OR.

Un altro structural testing criteria è la **Basic condition testing**. Questo test prevede che ogni condizione base venga eseguita almeno una volta. Questo significa che ogni condizione del programma deve essere Vera e Falsa per almeno un caso di test. Anche qui abbiamo una misura di coverage che divide il numero di valori di True e False presi da ogni basic condition con il doppio del numero delle basic condition totali. Possiamo avere casi in cui la basic condition adequacy viene soddisfatta e al tempo stesso non soddisfare le branch condition adequacy; infatti sempre nel caso dell'OR di due condizioni se abbiamo degli input che ottengono T, F e F,T in questo caso abbiamo soddisfatto le basic condition ma non la branch in quanto il risultato della OR con questi input sarà sempre True e quindi non viene esplorato il branch del False.

Una possibile combinazione della branch condition e la basic condition è chiamata branch and condition adequacy che ovviamente è definito come un test che prevede che vengano soddisfatte sia la branch che la basic condition. Quindi avremo che tutte le condizioni di un programma siano coperte e tutti i suoi rami vengano esplorati.

Un'estensione più completa che include sia la basic che branch condition è la **compound condition adequacy**. A differenza della condizione precedente questa si occupa di coprire tutte le possibili valutazioni delle compound condition (ovvero coprire tutti i possibili valori della truth table) e di coprire tutti i rami di un decision tree.

Questi approcci tuttavia sono difficili da implementare in quanto la crescita delle possibilità da prendere in considerazione per la creazione della truth table è esponenziale. Pertanto esiste un approccio più intelligente ovvero quello della **MC/DC (Modified Condition/Decision Coverage)**. Questo approccio prende

ogni singola condizione del programma e inserirla in una truth table avendo cura che tra tutti i casi di test ce ne sia almeno uno per cui:

- 1- Tale condizione sia **vera**
- 2- Tale condizione sia **falsa**
- 3- Se la condizione è **vera** allora questa avrà un **impatto sull'output finale** (ovvero se è vera allora output vero)
- 4- Se la condizione è **falsa** allora questa avrà un **impatto sull'output finale** (ovvero se è falsa allora output falso)

Per valutare se una condizione avrà un impatto sull'output finale consideriamo la truth table, prendiamo in considerazione che i valori "dont care" (ovvero in generale quelli messi negli OR) possano assumere qualsiasi valore vogliamo, quindi vediamo se al variare della condizione in considerazione allora varia anche l'output finale nonostante vengano mantenute invariate le altre condizioni fisse ad eccezione di quelle "dont care". Per capire meglio questo tipo di

approccio vediamo come per la seguente tabella per la condizione a i test case 1 e 13 siano quelle che impattano sull'output finale.

La tabella viene definita dalle seguenti condizioni $((a \mid b) \& c) \mid d) \& e$. I valori sottolineati identificano quei valori che hanno un impatto sull'output finale.

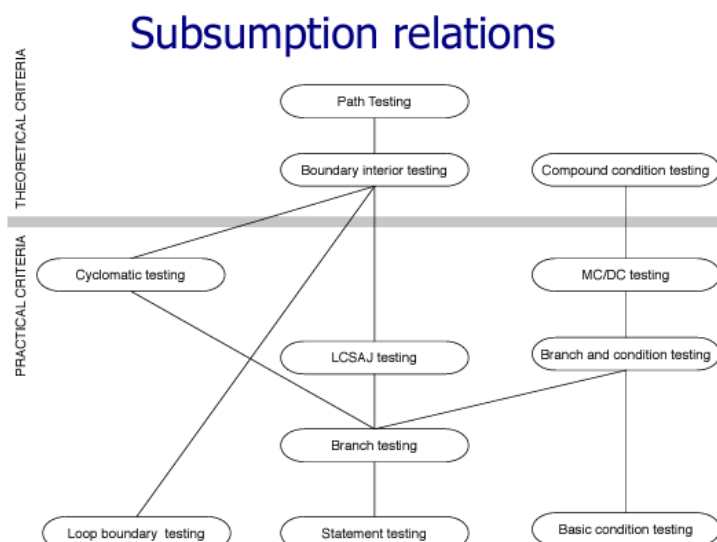
	a	b	c	d	e	Decision
(1)	<u>True</u>	–	<u>True</u>	–	<u>True</u>	True
(2)	<u>False</u>	<u>True</u>	<u>True</u>	–	<u>True</u>	True
(3)	<u>True</u>	–	<u>False</u>	<u>True</u>	<u>True</u>	True
(6)	<u>True</u>	–	<u>True</u>	–	<u>False</u>	False
(11)	<u>True</u>	–	<u>False</u>	<u>False</u>	–	False
(13)	<u>False</u>	<u>False</u>	–	<u>False</u>	–	False

Un altro structural testing criteria è la **Path adequacy testing**. Questo test prevede che ogni cammino venga percorso almeno una volta. Anche qui abbiamo una misura di coverage che divide il numero di path eseguito con il numero totale di path possibili.

Il più comune tra queste tipologie di testing è il **Boundary interior path coverage**. Questa tecnica prevede di prendere in considerazione tutti i path del programma che non entrano più di una volta all'interno di un loop.

Un altro criterio è la **LCASJ (Linear Code and Sequence Jump) adequacy**. Questo test va ad analizzare tutti i subpath sequenziali all'interno di un CFG.

Per poter confrontare questi criteri di testing abbiamo la **subsumption relation**. Il grafico che identifica la subsumption dei criteri è il seguente:



Possiamo notare come il **branch testing** subsuma lo **statement testing** come abbiamo detto prima a livello teorico. È importante però dire che nonostante graficamente **Basic condition testing** sembrerebbe possa subsumere la **statement testing**, in realtà **non subsume lo statement testing in quanto eseguire tutte le condizioni con tutte le combinazioni possibili di vero e falso non sempre ci porta a seguire tutti i branch oppure tutti gli statement del codice (basti pensare ad un OR dove se usi T,F e F,T hai valutato tutte le condizioni ma non hai valutato il caso in cui c'è FF che porterebbe ad un altro branch o statement)**. **Notiamo anche la divisione del grafico in sezione pratica e teorica.** I criteri pratici sono dei criteri che generano un numero non esponenziale di casi mentre i criteri teorici ne creano un numero esponenziale.

MC/DC – HOMEWORK

The following program fragment adds an object to the right branch of a tree structure.

```
if((o != null) && !tree.contains(o) && (tree.isBalanced() || (tree.right().size() < tree.left().size()))) {  
  
    tree.addRight(o);  
  
}
```

Derive a set of test cases that satisfy the MC/DC adequacy criterion. The solution should indicate the elementary conditions, specify the combinations of truth values that satisfy the MC/DC criterion, and provide a set of corresponding concrete test cases.

Per procedere alla risoluzione di questa domanda **come prima cosa bisogna eseguire MC/DC** e successivamente andare a **definire i test case che identificano i vari scenari possibili**.

Per questione di **leggibilità** andiamo a identificare le varie condizioni del programma nel seguente modo:

`if((o != null) : A`

`!tree.contains(o) : B`

`tree.isBalanced(): C`

`(tree.right().size() < tree.left().size() : D`

Possiamo quindi **riscrivere la porzione di programma** nel seguente modo:

IF (A && B && (C || D))

Da qui possiamo creare dei test case che soddisfano la MC/DC:

	A	B	C	D	OUT
1	T	T	T	-	T
2	F	-	-	-	F
3	T	F	-	-	F
4	T	T	F	F	F
5	T	T	F	T	T

Possiamo notare come **non siamo partiti da un test esaustivo** dove andiamo a prendere in considerazione tutte le possibili combinazioni di valori di condizioni, ma **siamo andati direttamente a creare la tabella MC/DC inserendo dove opportuno i "dont care" (-)**. In questo modo abbiamo individuato il numero minimo di casi di test possibili da utilizzare ovvero 5.

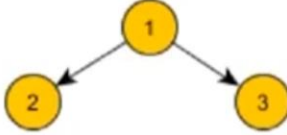
Ora dobbiamo andare a **creare un set di test case CONCRETI** corrispondenti a ciò che abbiamo individuato nella tabella MC/DC. Sarebbe quindi opportuno utilizzare dello pseudo codice per crearli in ottica **assignment**, quindi in pratica scrivere del codice molto base che va a costruire l'oggetto che soddisfa quelle condizioni specificate nella tabella MC/DC tenendo ovviamente conto che essendo un test case concreto dobbiamo associare un valore anche a quelle condizioni che nella tabella sono "dont care". Ad esempio per il primo test case della tabella dovremmo creare un albero, successivamente inizializzare un albero che non contiene un oggetto, bilanciare l'albero e poi decidere se la dimensione dell'albero di destra sia maggiore di quello di sinistra o meno (dato che è dont care posso scegliere cosa fare succedere). Segue un immagine che rappresenta come si potrebbe implementare un test case concreto:

Test case 1

Riga della tabella MC/DC per la quale si costruisce il test case concreto TC1. (Copiata dalla tabella MC/DC sopra creata così da verificare facilmente se il test case creato è corretto).

<u>o != null</u>	<u>!tree.contains(o)</u>	<u>tree.isBalanced()</u>	<u>tree.right().size() < tree.left().size()</u>
<u>T</u>	<u>T</u>	<u>T</u>	F

Test case: TC1
Object o=4;
Tree tree=new Tree();
tree.addRoot(1);
tree.addLeft(2);
tree.addRight(3);



```

graph TD
    1((1)) --> 2((2))
    1 --> 3((3))
  
```

Questa immagine rappresenta l'esecuzione del test case 1 della tabella MC/DC. I valori T sono sottolineati in quanto influiscono sul risultato finale, infatti F non è sottolineato in quanto nella tabella MC/DC è un "dont care" (lo si poteva mettere anche a True che non cambiava nulla). Poi nella tabella sotto abbiamo un test case concreto dove andiamo ad assegnare un valore non nullo a o, poi inizializziamo un nuovo albero, poi ci aggiungiamo una radice e due nodi. Verifichiamo come questo esempio soddisfi tutte le condizioni: o non è nullo, l'albero non contiene o, l'albero è bilanciato, i nodi di destra non sono minori di quelli di sinistra. Abbiamo quindi rispettato le condizioni imposte e quindi abbiamo creato un test case adeguato.

MODEL BASED TESTING

MODEL BASED TESTING – VIDEOLEZIONE

In questo capitolo si parlerà di come generare dei test cases da dei model. Prima di procedere è necessario avere un'immagine più grande di come si vanno a produrre questi test cases: si inizia sempre dalle **specification di un programma**, le quali possono essere di dimensione elevata e non utili se considerate tutte insieme. Di queste specification si va ad identificare quali siano le feature che possono essere testate in maniera indipendente dagli altri feature del programma. Infine si vanno ad identificare le test case specification, ovvero una descrizione delle caratteristiche di ogni test case, per poi andare a definire nel concreto i test cases.

Per poter passare da feature che possono essere testate in maniera indipendente alla generazione dei test cases abbiamo due possibili strade:

- 1- **Representative values**: identificare le representative values.
- 2- **Model**: derivare un modello da cui poi estrarre le test specifications.

In questo capitolo si parla della strada che prevede l'utilizzo dei Models.

Il primo modello che prendiamo in considerazione è la **Finite state Machine**. Per quanto riguarda le sue **specifiche informali** (vuol dire che stiamo dando un esempio su cui poi eseguire un FSM model) possiamo dire che **esiste uno slot** che rappresenta lo slot di un computer model che **può essere bound o unbound**. Gli slot bound sono assegnati ad un componente mentre quelli unbound sono vuoti. Gli slot offrono diversi servizi tra cui **installare** un modello, funzione di **binding** con un componente, funzione di **unbinding** e **verifica di binding**. Da queste informazioni possiamo identificare gli stati e le transizioni che consentono di passare da uno stato ad un altro; in particolare **gli stati sono not installed, unbound e bound**, mentre **le transizioni sono install, bind, unbind e isbound**. Da qui quindi possiamo creare il grafico FSM e quindi generare il modello.

Un altro modello da prendere in considerazione sono le **Decision Structures**. Per quanto riguarda le **specifiche informali** possiamo dire che abbiamo **una funzione di pricing** che determina il prezzo di un prodotto sulla base della tipologia di utente che sta eseguendo l'acquisto. Ci sono **tre tipologie di utenti che sono Buisiness, educational e individual**, ognuno delle quali ha delle caratteristiche di pricing diverse. Per risolvere questi tipi di modelli bisogna andare a **definire una decision table** che verifica che cosa succede all'output finale se il valore delle condizioni varia. Per ottenere una maggiore precisione nella rappresentazione **possiamo aggiungere dei constraints** che definiscono delle regole (come ad esempio non può esistere un utente sia student che business). Una volta che abbiamo la decision table e i constraints possiamo andare a definire **tre tipologie di test cases**:

- **Basic condition coverage**: genera una **test case specification** per ogni colonna della tabella
- **Compound condition coverage**: genera una test case specification **per ogni combinazione di truth values delle basic condition** (ovvero ogni condizione deve essere valutata almeno una volta quando è vera e falsa per ogni combinazione possibile di condizioni)
- **Modified condition/decision coverage (MC/DC)**

Un altro modello da prendere in considerazione sono le **Flowgraph based testing**. Anche in questo caso abbiamo delle **specifiche informali** che descrivono un problema di spedizione di un ordine di un prodotto comprato. Tra le varie informazioni fornite dal problema abbiamo: il costo del prodotto, l'indirizzo di spedizione, il metodo di spedizione, ecc... **Da tutte queste specifiche possiamo andare a creare un grafico CFG che rappresenti il problema indicato**.

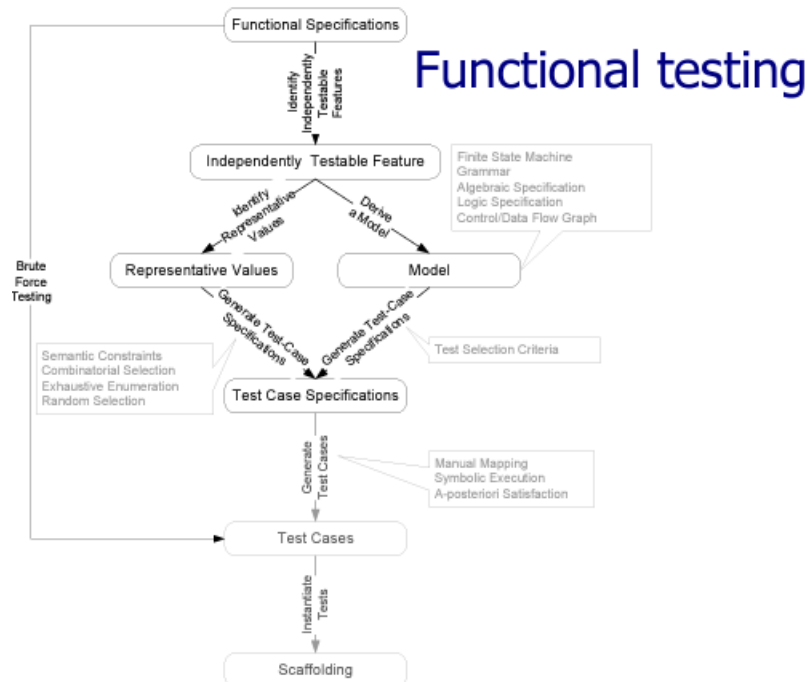
L'ultimo modello che andiamo a vedere è il **Grammar Based testing**. Di questo non si capisce un cazzo di niente quindi spera di non utilizzarlo mai. Guarda caso è il terzo punto dell'homework mai fatto, che spero venga visto nella lezione dopo.

MODEL BASED TESTING – VIDEOLEZIONE 2

In questa lezione si vede nel dettaglio alcune caratteristiche del model testing. In generale si può dire che gli **obiettivi principali** del testing sono quelli di **rivelare dei problemi nel software rivelandone gli errori** e, dopo aver corretto gli errori rivelati, **avere un software che superi tutte le prove di test** e si dimostri quindi corretto. La tecnica model based va a definire un **criterio di adeguatezza** che ci indica **se è vero o falso che abbiamo eseguito un numero adeguato di test case con risultati positivi per essere sicuri della correttezza del software stesso**. L'obiettivo quindi sarà quello di **andare ad elencare un insieme di obiettivi di test che riteniamo esaustivi per la correttezza del software e poi cercare di eseguirli tutti**. Per ottenere questo insieme di obiettivi di test ci sono diverse strade che sono generalmente raggruppate in **test strutturali (basati sul codice)**, **test funzionali (basati sulle specifiche del sistema)** e **test basati sui modelli**.

Il processo di test design segue lo schema che segue. In questo schema abbiamo come primo punto di partenza **le specifiche del software**; da queste bisogna **derivare delle feature che possono essere testate indipendentemente**. Successivamente si va ad **elencare gli obiettivi di test attraverso i modelli che**

rappresentano in maniera più formale una specifica. Da qui possiamo derivare le **specifiche di test**, **costruire i test cases** e concludere il processo con lo **scaffolding**.



MODEL BASED TESTING – HOMEWORK

Il compito ti dà delle **specifiche informali** dove ti spiega il funzionamento di un **sito di vendita di libri**. Da queste specifiche bisogna **estrapolare delle feature che NON sono le singole funzioni ma le macrofunzioni**. Una volta **identificate queste feature** bisogna andare **ad identificare un metodo per generare dei test case e derivare un modello di rappresentazione della feature**.

Quindi in questo caso abbiamo 3 feature: sconto, acquisto e modalità di salvataggio dei libri nella piattaforma. Di queste 3 feature ci **copiamo la parte di specifiche informali** che fanno riferimento a queste, poi **descriviamo il modello che abbiamo deciso di applicare** e, infine, **appliciamo il modello**.

Per la **prima** feature dello sconto si utilizza il modello della **decision table**; in questa tabella ogni riga **conterrà una condizione** (es. avere fidelity card, avere più di 100 libri comprati) **mentre nella riga finale avremo l'output ottenuto dai valori di queste condizioni**. Avremo quindi una tabella composta da combinazioni di veri, falsi e dont care e in fondo l'output finale ottenuto. In questo caso bisogna prestare attenzione alla condizione (nascosta) che definisce che lo sconto dei 100 libri comprati sia cumulabile e quindi bisogna interpretarlo non come un dont care ma valutarlo con vero o falso in diverse combinazioni possibili.

Per la **seconda** feature si utilizza un **modello CFG**. Leggendo le specifiche di acquisto del libro possiamo **creare un grafico che valuta le possibili situazioni che possono verificarsi durante l'acquisto**. In generale si andrà a creare un semplice **grafico con delle biforcazioni ottenute generalmente dalla presenza di if**. Si poteva anche eventualmente **utilizzare una macchina a stati** anche se non spiega come.

La **terza** feature usa un **grammar based testing**. Il codice fornito nella specifica è il seguente:

The books are stored on the platform according to the following DTD scheme:

<!ELEMENT books (book+)>

<!ELEMENT book (title, author+,year,rating+,review+,price-new,price-pre-owned)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT author (#PCDATA)>

<!ELEMENT year (#PCDATA)>

<!ELEMENT rating (0|1|2|3|4|5)>

<!ELEMENT review (#PCDATA)>

<!ELEMENT price-new (#PCDATA)>

<!ELEMENT price-pre-owned (#PCDATA)>

La soluzione corretta proposta da uno studente dice che è partito dalle strutture più complesse ovvero da books che è una lista di oggetti di tipo book e successivamente ha analizzato le strutture più semplici. Segue la soluzione:

<books> ::= <bookSequence>

<bookSequence> ::= <book> <bookSequence> | <book>

<book> ::= <title><author+><year><rating+><review+><price-new><price-pre-owned>

<author+> ::= <author> <author+> | <author>

<review+> ::= <review> <review+> | <author>

<rating+> ::= <rating> <rating+> | <author>

<title> ::= STRING

<author> ::= STRING

<year> ::= STRING

<rating> ::= 0|1|2|3|4|5

<review> ::= STRING

<price-new> ::= STRING

<price-pre-owned> ::= STRING

Possiamo notare alcune informazioni particolari dal codice DTD iniziale:

- **#PCDATA accetta qualsiasi condizione possibile** quindi STRING va bene ma potevo mettere anche un INT o DATA se opportuno ad esempio nel caso di <year> che poteva essere DATA.
- **Il + identifica una o più elementi quindi non è contemplata la presenza di un EMPTY**. In questo caso abbiamo per books, author, review e rating i due casi divisi dalla | che identificano il caso in cui ci sia un solo elemento e il caso in cui ci siano più elementi. Nel caso in cui ci fosse stata una * allora bisognava anche prendere in considerazione la possibile presenza di un EMPTY aggiungendolo con la | dopo i due appena menzionati.

Infine l'esercizio chiedeva di andare a definire un metodo per andare a definire i casi di test a partire dai tre definiti dal libro che sono:

- 1- Production coverage criterion
- 2- Boundary condition grammar based criterion
- 3- Probabilistic grammar based criterion

Quello più adeguato è il **production coverage criterion** dato che sono unbound e quindi è sufficiente per creare i casi di test.

QUESTE ULTIME RIGHE NON SONO CHIARE GUARDA IL PDF

FAULT BASED TESTING AND MUTATION ANALYSIS

FAULT BASED TESTING – VIDEOLEZIONE

Abbiamo visto negli scorsi capitoli che i test case possono essere derivati da diverse tecniche tra cui:

- **Functional testing**: deriva dalle **specifiche** o dalle informazioni che determinano il comportamento di un programma
- **Structural testing**: deriva dal **codice** e dalla struttura del programma
- **Model based testing**: deriva da dei **modelli** sul comportamento del programma

L'ultima tecnica è il **fault based testing**. Questa **tecnica fa uso di quello che pensiamo sia un punto debole del programma dove potrebbero esserci degli errori** per generare dei test case ad hoc.

Per spiegare il funzionamento di questa tecnica partiamo da un esempio: come posso contare **il numero di pesci in un lago**? La risposta è semplice: **non posso, devo fare una stima**. Ma come possiamo fare una stima? Possiamo **prendere ad esempio 100 pesci e segnarli con un punto rosso**, successivamente ci mettiamo a **pescare altri 100 pesci e vediamo quanti di questi sono segnati di rosso**. In questo modo se ad esempio vediamo che **solo 20 pesci sono segnati**, posso andare a **stimare che il numero di pesci totale nel lago sia all'incirca 5 volte 100 pesci ovvero 500**.

Allo stesso modo posso trovare gli errori in un programma. Ci sono due tecniche per trovare questi errori, una delle quali è la **Mutation Testing**. In questo tipo di testing un **Mutant è una copia di un programma con una mutazione**, mentre una **mutazione è un cambiamento sintattico generato da noi (seeded bug)** come ad esempio **cambiare un > in un >=**. Una volta definiti mutant e mutation **vado ad eseguire una test suite su tutti i mutant programs**. I mutant vengono **killati se falliscono almeno una volta durante l'esecuzione dei test**. Il numero i mutant killed in questo processo mi indicano **l'efficienza del mio test suite nell'individuare bug reali**. Le assunzioni che stanno alla base di questo tipo di testing sono:

- **Competent programmer hypothesis**: gli errori reali sono delle **piccole variazioni del programma corretto**
- **Coupling effect hypothesis**: **se individuo degli errori semplici, allora individuo anche gli errori complessi** in quanto questi sono una composizione degli errori semplici.

Per ottenere le mutazioni che danno il via a questo processo si utilizzano i **Mutation Operators**. Questi sono delle piccole variazioni del codice come ad esempio:

- **Constant replacement**: cambia il valore di una costante come ad esempio **X > 2 trasformato in X > 13**
- **Relation operator replacement**: cambia l'operatore come ad esempio **X > 2 trasformato in X < 2**

- **Variable initialization elimination**: elimino il valore di inizializzazione di una variabile come ad esempio `int X=2` trasformato in `int X`

I mutant che **non vengono killati nel processo di esecuzione** del programma vengono chiamati **Live Mutants**. Questi non vengono eliminati generalmente perché:

- Sono equivalenti al valore iniziale pre-mutazione (es. $X > 0$ e $X \geq 0$ possono essere equivalenti se X non vale mai 0)
- Il test non è adeguato

Il problema principale che questo tipo di testing porta con sé è il **numero considerevole di mutant che vengono generati e che quindi devono essere testati**. Ci sono **due modi per testare questi mutant** che sono:

- 1- **Weak Mutation**: non vengono eseguiti tutti i mutant ma dei meta-mutant che contengono diversi mutant (quindi **non valuto ogni singolo mutant alla volta ma di più in contemporanea**) e durante l'esecuzione faccio la **kill dei mutant non appena trovo una singola variazione** nell'esecuzione del programma.
- 2- **Statistical Mutation**: creo un **sample randomico di mutant** selezionati per l'esecuzione

TEST EXECUTION – LEZIONE LUNGA (SEZIONE DI FAULT-BASED TESTING)

La lezione nella sezione della test execution in realtà parla di fault-based testing per la maggior parte del tempo quindi non ha senso che stia lì ma qui.

Il **motivo per cui si fa fault-based testing** non è quello di semplicemente iniettare dei fault e riuscire ad identificarli ma bensì iniettare dei **fault che consentono di elencare dei comportamenti del programma e sapere che il nostro test suite è sufficiente per poter distinguere tutti quei fault rispetto a quel criterio dà un'idea della forza della test suite**.

La relazione che c'è tra mutation analysis e fault-based testing è che **la mutation analysis è una delle possibili implementazioni della fault-based testing**. Se quindi **idealmente il nostro programma avesse errori che appartengono esclusivamente ad una singola tipologia di mutazione**, allora implementando diversi test case che utilizzano delle mutazioni di quel tipo potremmo andare a dire che **il mutation testing è effettivamente sufficiente**. Per creare le mutazioni è possibile prendere spunto dalla seguente tabella di **mutazioni di operandi**:

<i>Operand Modifications</i>			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$

Ad esempio il primo modello presentato dove si **rimpiazza la costante C1 con C2** può rappresentare una **svista del programmatore che sbaglia a passare la costante ad un determinato metodo**. Questo meccanismo di inserimento di diverse mutazioni ad esempio rimpiazzando le costanti costituisce l'implementazione del fault-based testing.

La **test adequacy** viene definita secondo la mutation analysis come il rapporto tra mutazioni iniettate e mutazioni individuate, ovvero il **mutation score**. Ogni volta che un mutante viene identificato dalla test suite si dice che avviene una **KILL** del mutante.

Come possiamo definire un **non-valid mutant** e un **non-useful mutant**? Un **mutante non valido**, dato che stiamo iniettando dei dati, è un mutante che **produce un codice che non compila**, ovvero che dà errore in esecuzione. Un **mutante non utile** è un mutante che **produce un codice che compila, ma che non ha particolari utilità per valutare la bontà della test suite in quanto produce un errore troppo facile da trovare** (ad esempio modificare l'entry point in modo tale che dia `NullPointerException`).

I **mutanti equivalenti** non sono lo stesso dei mutanti non utili, in quanto sono dei mutanti che non consentono di distinguere dal programma originale la test suite, **ovvero le mutazioni iniettate non generano nessun cambiamento rispetto all'esecuzione originale del programma**.

Le kill delle mutazioni possono essere di due tipi: **weak kill** e **strong kill**. Sappiamo che le kill avvengono se durante l'esecuzione del programma originale non ottengo nessun fault e durante l'esecuzione del programma con un mutante ottengo un fault. Questo tipo di kill che è quello che generalmente si incontra è la **strong kill**, quindi la kill che avviene se il codice originale ottiene un risultato diverso dal codice col mutante. Le **weak kill** invece si verificano quando si osservano i risultati intermedi e il programma originale ottiene dei risultati intermedi diversi dal programma con la mutazione; **in pratica si guarda lo stato del programma immediatamente dopo aver eseguito la linea dove viene iniettata la mutazione e si verifica se l'output coincide**.

TEST EXECUTION

TEST EXECUTION – LEZIONE BREVE

Nel capitolo precedente abbiamo visto come un problema fondamentale del testing sia quello di **campionare lo spazio di esecuzione**, ovvero **selezionare un set definito di input di esecuzioni da testare basandoci sul codice, specifiche, errori ecc...** Questi metodi vanno a definire dei **test cases**. Una volta ottenuti i test cases bisogna andare ad eseguire effettivamente il testing attraverso una **struttura che possa eseguire il programma**. Spesso i test cases vengono eseguiti non sulla versione deployed e completa del programma, ma bensì su delle **unità del programma che possono avere delle dipendenze da dei sistemi o da altre componenti del programma**.

Per eseguire il programma incompleto abbiamo quindi bisogno dello **SCAFFOLDING**, ovvero un qualcosa che **non necessariamente farà parte del prodotto finale del programma ma che serve per testare ciò che effettivamente sarà parte del programma finale**. Nello scaffolding i dati forniti in input prendono il nome di **DRIVER**. Se ad esempio una unità di un programma che vogliamo testare utilizza la **funzione compare()** che confronta due elementi, ma questa funzione **non è stata ancora definita al momento del testing** nella sezione che stiamo testando, allora abbiamo bisogno di **un qualcosa che simuli la funzione che viene chiamato STUB o MOCK**. Se il programma che stiamo analizzando **utilizza dei dati provenienti da un database** che al momento del testing non è disponibile abbiamo bisogno di qualcosa che lo sostituisca e questo è il cosiddetto **HARNESS**. Infine un programma potrebbe avere la necessità di verificare la **se il risultato prodotto sia accettabile o meno** e questo viene chiamato **ORACLE**.

Per creare uno scaffolding abbiamo bisogno di rispettare la fondamentale **proprietà di controllability & observability**, ovvero avere la possibilità in fase di testing di **controllare l'esecuzione anche in assenza di**

componenti che risultano essere indispensabili per l'esecuzione dell'unità che si sta testando e di osservare i risultati prodotti dall'esecuzione del test senza necessariamente avere le componenti adibite a questo compito. Per ottenere questo risultato si possono percorrere due strade: o si utilizza un programma ad-hoc che vengono generati con JUnit oppure si usa un programma generico che utilizzano vecchie funzionalità o costruiscono delle funzionalità nuove applicabili in diverse situazioni. In generale si creano degli STUB specifici per i casi che hanno particolare importanza, mentre si utilizzano degli STUB più generici per il resto.

Abbiamo detto che l'ORACLE è quella componente adibita a prevedere quale sia il risultato accettabile del programma. Per costruire gli oracoli ci sono due approcci diversi il primo è il Comparison-Based oracle. Questo tipo di oracolo esegue il programma che si vuole testare con i test inputs e si confrontano i risultati così ottenuti con un set di output corretti per determinare se i risultati ottenuti siano accettabili o meno. Il secondo approccio adottato sono i Self-Checking oracle. Questi oracoli non vanno più a confrontare i risultati a fine esecuzione ma bensì durante l'esecuzione del programma attraverso delle sezioni di self checking che verificano la correttezza fino a quel punto di esecuzione.

Quando si vanno ad utilizzare gli oracoli negli object oriented systems in generale abbiamo una serie di oggetti e una serie di inputs che sono una serie di invocazioni di metodi e infine abbiamo dei risultati che sono sia un output che uno stato in cui verte il programma. Dobbiamo quindi avere degli oracoli che non solo controllano che l'output sia accettabile ma che anche lo stato del programma lo sia. Per fare ciò ci sono due metodi di osservazione dello stato di output che sono gli intrusive approaches e gli equivalent scenarios. I primi utilizzano degli specifici costrutti come ad esempio gli inspectors che possono individuare lo stato della classe sotto test. Il secondo approccio anziché utilizzare l'esatta sequenza di metodi invocati dalla unità che si sta testando va ad utilizzare una sequenza di metodi equivalente e una sequenza di metodi non equivalenti per poi confrontarne il risultato. Per generare degli scenari equivalenti in generale andiamo ad escludere la sequenza di invocazioni di metodi che riporterebbe il programma allo stato iniziale e considereremo quindi solo quella porzione di invocazione di metodi che effettivamente altera lo stato del programma. Per generare invece degli scenari non equivalenti in generale si va ad eliminare (abbastanza a caso) alcune invocazioni dei metodi originali, spostare alcune invocazioni di metodi o crearne delle nuove senza andare a stravolgere troppo la sequenza di invocazioni di metodi iniziale. Infine per verificare l'equivalenza degli stati si possono utilizzare degli inspectors, esaminare i risultati ottenuti applicando un set di metodi e verificare se questi eseguiti singolarmente produrrebbero lo stesso risultato o meno, utilizzare un metodo compare che consente di verificare se l'oggetto prodotto sia equivalente ad un altro oggetto di cui si conosce la correttezza.

TEST EXECUTION – LEZIONE LUNGA

Parte della lezione lunga che parla effettivamente del test execution.

Il test driver è il sostituto di alcune parti di programma, come ad esempio il suo main, che ne consente il suo funzionamento complessivo.

Lo stub è il sostituto di alcune sotto funzionalità del programma come ad esempio dei metodi che vengono invocati durante l'esecuzione del programma.

L'oracolo è il valore atteso che viene utilizzato per valutare se un caso di test è accettabile oppure no. Gli oracoli possono essere parziali, ovvero non determinare uno specifico output ma verificare che l'output rispetti alcune proprietà ritenute importanti del programma. In generale l'oracolo parziale viene implementato inserendo delle asserzioni in certi punti del codice che vanno a convalidare che il valore delle variabili rispettino le attese oppure no.

PLANNING AND MONITORING

PLANNING AND MONITORING– VIDEOLEZIONE

Si può dire che il costo di individuare e riparare dei fault in un programma cresca al crescere del lasso di tempo che intercorre tra il commettere l'errore nel codice e l'individuazione dei fault causati dall'errore stesso. Questo avviene perché non accorgersi subito di un errore può compromettere tutte le funzionalità sviluppate dopo rendendole da modificare o riscrivere completamente.

Il quality process in generale include tre tipologie di verification steps:

- 1- **Internal consistency checks**: controlla che il sistema sia consistente internamente come ad esempio verificare che le classi non siano troppo grandi a livello di linee di codice
- 2- **External consistency checks**: controlla gli elementi specifici del sistema
- 3- **Generation of correctness conjectures**: fare uso di test cases per verificare la correttezza di un sistema

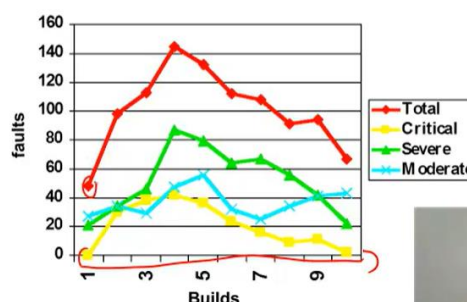
In generale il quality process è composto da diverse attività che variano tra internal e external consistency check e anche la generazione di test cases i quali sono pianificati durante un lungo lasso di tempo. Per fare ciò si fa uso del quality plan e delle strategies. Possiamo dire che le strategies hanno come obiettivo l'organizzazione di metodi di controllo generici applicabili a diversi scenari, mentre i quality plan hanno come focus principale il progetto stesso che si sta considerando. Per quanto riguarda la struttura delle strategies possiamo dire che avendo un focus generico queste vanno ad individuare metodi di approccio tramite l'utilizzo di esperienze passate di progetti passati, invece per quanto riguarda i quality plan si ha una struttura standard per lo specifico progetto in oggetto. Infine si può concludere che l'evoluzione delle strategies è un processo lento che si evolve assieme all'organizzazione stessa, mentre l'evoluzione dei quality plan è molto rapida e si adatta alle esigenze del singolo progetto.

Guardando nel dettaglio come avvengono i test e le analisi tramite l'utilizzo delle strategies possiamo dire che queste vengono eseguite sulla base di esperienze passate evitando di commettere gli stessi errori commessi nel passato. Per quanto riguarda i test e le analisi utilizzano un plan invece ha una strategia ben definita che prevede di: verificare oggetti e features, le attività vengono organizzate e le risorse vengono allocate alle attività in base alla loro importanza e complessità, segue degli approcci specifici di sviluppo e infine utilizza dei criteri ben definiti per valutare i risultati ottenuti.

Un'attività molto importante che fa parte del quality process è il risk planning. Questa attività fa fronte a diverse criticità che si possono riscontrare durante lo sviluppo che possiamo suddividere in

- **Generici management risk**: gestione del personale, delle tecnologie a disposizione, scheduling delle attività in visione delle deadline, ecc..
- **Quality risk**: gestione della fase di development, execution e individuare i requisiti quanto prima

Dato che lo sviluppo di un progetto avviene attraverso diverse release, è possibile monitorare l'andamento del verificarsi di criticità tramite analogia. Questa pratica in sostanza va a contare quanti faults e di quali tipologie si presentano nelle varie release per monitorarne l'andamento.



I valori in questo grafico non sono da prendere come negativi se i faults sono tanti, perché se ho tanti faults all'inizio dello sviluppo del programma vuol dire che ho implementato dei metodi efficienti per individuarli e il decrescere costante degli errori determina anche un corretto approccio per la loro risoluzione.

Minuto 25 mi sono rotto le palle di sta roba

INTEGRATION, SYSTEM AND REGRESSION TESTING

INTEGRATION AND SYSTEM TESTING– VIDEOLEZIONE

A differenza di tutte le modalità precedentemente viste di testing che eseguivano il test di una parte di codice, l'**integration and system testing** va a testare i diversi elementi, che singolarmente potrebbero già essere stati testati con i metodi visti nei capitoli precedenti, unendoli tra di loro andando quindi a comporre il programma completo dove le varie unità interagiscono tra di loro. Questa interazione può portare a dei problemi che non sarebbero visibili tramite il testing isolato delle varie unità che compongono il sistema. In particolare possiamo dire che l'**integration testing** va a testare delle unità integrate tra di loro ma non l'intero sistema, mentre l'intero sistema è testato tramite il system testing.

I faults che possono presentarsi nelle integrazioni sono diversi da quelli che solitamente si incontrano quando si analizzano le singole unità. Alcuni dei più comuni faults che si possono individuare nelle integrazioni sono:

- **Inconsistent interpretation of parameters or values**: ad esempio quando due moduli integrati tra loro utilizzano diverse unità di misura per i valori utilizzati come ad esempio i metri e le yards.
- **Violations of value domains, capacity or size limits**: ad esempio un buffer overflow
- **Side effects on parameters resources**: ad esempio situazioni di conflitto tra file temporanei
- **Omitted or misunderstood functionality**: ad esempio interpretazioni inconsistenti di web hits
- **Nonfunctional properties**: ad esempio integrare due componenti che lavorando assieme ci mettono troppo tempo e ottengono un timeout
- **Dynamic mismatches**: ad esempio incompatibili chiamate a metodi polimorfi

Ci sono differenti strategie per eseguire l'integration testing:

- **Big Bang**: mettere assieme tutte le unità e testare l'intero sistema
- **Top-down**: strategia strutturale che inizia da un modulo top e si crea uno STUB per i moduli che sono utilizzati dal top module. Successivamente si analizzano i moduli STUB creati prima e i suoi sotto moduli fino ad arrivare al punto in cui non ci sono più sotto moduli.
- **Bottom-up**: strategia strutturale che parte da uno stub e risale al suo driver e così via fino a tornare al top module.
- **Sandwich or Backbone**: strategia strutturale che integra la strategia top-down e bottom-up in base alle esigenze di testing.
- **Threads**: strategia funzionale che testa l'intero sistema, ma di ogni modulo di questo ne testa un singolo thread alla volta che riguarda una specifica funzionalità e va avanti finché non testa tutte le funzionalità che ci sono

Per quanto riguarda le strategie possibili per eseguire il system testing abbiamo:

- **System testing**: testing che va a verificare la correttezza e la completezza, ovvero la capacità del sistema di soddisfare le specifiche richieste.
- **Acceptance testing**: testing che va a verificare la utilità e soddisfazione, ovvero la capacità del sistema di fornire le funzionalità chieste dagli utenti per cui è stato creato.
- **Regression testing**: testing che va a verificare che le varie versioni di release del programma mantengano le stesse funzionalità delle precedenti.

In generale il system testing viene eseguito per **testare le proprietà globali** che non sarebbero possibili da analizzare tramite le singole unità come le **performance, la latenza e la reliability**.

Un'altra caratteristica generale su cui fare dei test è **l'usabilità**. Le tipologie di usability testing sono:

- **Exploratory testing**: fornire diverse versioni delle interfacce del programma agli utenti e individuare quale sia la più apprezzata
- **Comparison testing**: confrontare il proprio lavoro con quello dei competitors per individuare lacune e punti di forza
- **Usability validation testing**: creare delle nuove interfacce ed utilizzare una porzione di utenti per testarne le funzionalità

Quando si parla di regression test selection si tratta di andare ad **analizzare quali siano le conseguenze di una modifica ad una porzione di programma e di eseguire dei test adeguati**. In sostanza si va ad **individuare tutti quei test cases** che andavano a testare le funzionalità che sono state modificate e li si flagga come **obsoleti**, successivamente **si generano dei nuovi test cases** utili a testare sezione di codice modificata.