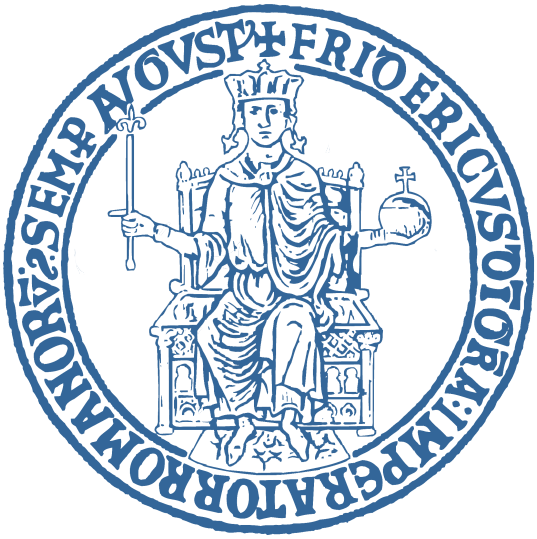


Università degli Studi di Napoli Federico II

Software Testing

2022/2023

Professore	*Studente*
Porfirio Tramontana	Vincenzo Tramo

Indice

Testing Theory

1 Testing Theory

1.1 Definizioni IEEE di "testing"

- **Standard IEEE 729 - 1983**: Il processo di analizzare un elemento software per rilevare le differenze tra condizioni esistenti e richieste (cioè, bugs) e per valutare le caratteristiche dell'elemento software.
- **Standard IEEE 610.12 - 1990**: Il processo di funzionamento di un sistema o di un componente in condizioni specifiche, l'osservazione o la registrazione dei risultati e la valutazione di alcuni aspetti del sistema o del componente.
- **Standard IEEE 829 - 2008**: Il processo di testing fornisce prove oggettive che il sistema software-based e i prodotti ad esso associati:
  - soddisfano i requisiti di sistema assegnati
  - risolvono il giusto problema
  - soddisfano l'uso previsto e le esigenze degli utenti

  Definizione di Testing: un'attività nella quale un sistema o un componente è eseguito sotto condizioni specifiche, i risultati sono osservati o registrati, e viene effettuata una valutazione di alcuni aspetti del sistema o componente.
- **Standard IEEE 29119 - 2013** (revisionata e confermata nel 2019): il testing sono un insieme di attività condotte per facilitare la scoperta e/o la valutazione delle proprietà di uno o più elementi di prova. Le attività di testing includono pianificazione, preparazione, esecuzione, reporting, e managment delle attività, nella misura in cui sono diretti al testing. Un **test item** è un prodotto di lavoro che viene testato (esempi: un sistema, un oggetto software, una documentazione dei requisiti, una specifica di design, una guida per utenti).

Altre definizioni:

- **Test case**: processo/prova/giudizio da eseguire
- **Test suite**: un set di test cases

E' importante capire le differenze tra *fault*, *error* e *failure*:

- **Software Fault** (*difetto*): un difetto statico nel software
- **Software Error** (*stato di errore*): uno incorretto stato interno che è la manifestazione di fault
- **Software Failure** (*fallimento*): esterno, incorretto comportamento rispetto ai requisiti o un'altra descrizione del comportamento atteso

Nel dettaglio:

- **Fault** (*defect* o *bug*): è un difetto all'interno del codice causato da un errore umano che potrebbe causare il fallimento (*failure*) del sistema software
- **Error** (*humar error*): errori umani nella programmazione, a causa di interpretazioni sbagliate o conoscenza insufficiente del problema da risolvere, oppure errore materiale o qualsiasi altro problema
- **Failure**: una esecuzione di un software che non fornisce il risultato e il comportamento aspettato. I Failures possono essere dinamici: succedono in un dato tempo con un dato input e può essere osservato solo attraverrso l'esecuzione del software

1.2 Obiettivi del Testing Software

1.2.1 Definizione di *Validation* e *Verification*

- **Verification**: il processo per determinare se i prodotti di una fase del processo di sviluppo software soddisfa i requisiti stabiliti durante la fase precedente
- **Validation**: il processo di valutare il software alla fine dello sviluppo per assicurare conformità con l'uso previsto

Verify e Validation (V & V) mostrano se il software soddisfa i suoi requisiti (Verification) e se questo soddisfa i requisiti utente (Validation).

- *Verification*: are we making the product right?
- *Validation*: are we making the right product?

La *verification* è solitamente un'attività più tecnica che utilizza la conoscenza dei singoli artefatti software, dei requisiti e delle specifiche. La *validation* dipende solitamente dalla conoscenza del dominio, cioè dell'applicazione per la quale il software è stato scritto. Ad esempio, la validazione di un software per un aereo richiede la conoscenza di ingegneri aerospaziali e piloti.

Potrebbe accadere che il software soddisfi le specifiche ma fallisce nella convalida (validation). Questo può succedere perché le specifiche non rispecchiano le reali esigenze degli utenti. La validazione espone i difetti delle specifiche.

1.3 Model-Driven Test Design

> Designers are more efficient and effective if they can raise their level of abstraction

Il processo Model-Driven Test Design (MDTD) divide il testing in una serie di piccoli tasks che semplificano la generazione di test.

Il lavoro di sviluppo dei test può essere suddiviso in quattro attività distinte: progettazione del test (*Test Design*), automazione del test (*Test Automation*), esecuzione del test (*Test Execution*) e valutazione del test (*Test Evaluation*).

1.3.1 Software Testing Foundations

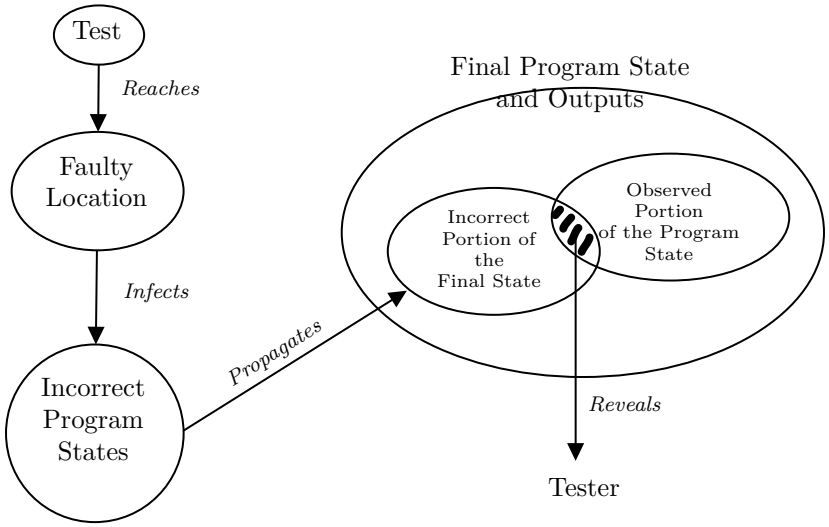
Una delle cose più importanti che un software testers deve conoscere è che i test possono solo dimostrare la *presenza di failures*, ma **non possono mostrare l'assenza di failures**. Il problema di trovare tutti i failures in un programma è *indecidibile* (non è possibile trovare algoritmi che risolvono tale problema).

La definizione di fault e failure ci permette di sviluppare il modello *reachability* (raggiungibilità), *infection* (infezione), *propagation* (propagazione) and *revealability* (rivelabilità) **RIPR Model**. Prima differenziamo il testing dal debugging:

- **Testing**: valutare software osservando la sua esecuzione
- **Test Failure**: l'esecuzione di un test che risulta in un fallimento software
- **Debugging**: il processo di trovare un fault dato un failure

Certamente il problema principale è che dato un fault, non tutti gli input "trigghereranno" il fault nel creare un output incorretto (failure). Anzi, è spesso difficile trovare il corrispondente fault associato al failure.

Sono necessarie quattro condizioni per osservare un failure.



La figura illustra le condizioni.

Prima, un test deve raggiungere la posizione o posizioni del programma che contiene il fault (**Reachability**).

Dopo che l'istruzione è stata eseguita, lo stato del programma deve essere incorretto (**Infection**).

opo, lo stato infettato deve propagare attraverso il resto dell'esecuzione un output o uno stato finale del programma incorretto (**Propagation**).

Alla fine, il tester deve osservare la parte incorretta dello stato finale del programma (**Revealability**). Se il tester osserva solo parti corrette dello stato finale del programma, il failure non è rilevato.

Queste quattro condizioni sono conosciute come il **fault/failure model**, o anche **RIPR model**.

Il modo migliore per affrontare con la complessità nell'ingegneria è usate l'**astrazione**, astraendo da dettagli complicati che possono essere ignorati innocuamente modellando il problema con delle strutture matematiche.

- **Test Design**: è il processo di progettazione di valori di input che testeranno efficacemente il software:
  - *Criteria-based test design*: progettazione di valori che soddisfano un determinato criterio (come ad esempio coperatura);
  - *Human-based test design*: progettazione di valori in base alla conoscenza del dominio e alla conoscenza umana.
- **Test Automation**: è il processo di incorporamento dei valori dei test negli script eseguibili.
- **Test Execution**: è il processo di esecuzione dei test sul software e la registrazione dei risultati;
- **Test Evalution**: è il processo di valutazione dei risultati dei test. La segnalazione dei risultati agli sviluppatori è spesso automatizzata se i test sono completamente automatici, ma non sempre è così.

Il Model-Driven Test Design (MDTD) consente ai progettisti di test di "aumentare il loro livello di astrazione".

Il punto di partenza nella figura è un artefatto software. Potrebbe trattarsi di codice sorgente o di un diagramma UML. Una progettazione basata sull'uomo utilizza l'artefatto per creare un modello astratto del software sotto forma di un domonio di input o di un grafico. Successivamente, l'artefatto viene utilizzato per considerare probabili problemi nel software e, in base ad un criterio, vengono creati i requisiti per testare tali problemi. I requisiti vengono poi perfezionati in una forma più specifica, denominata **specifica del test**.

Una volta perfezionati i requisiti di test, è necessario definire i valori di input che soddisfano i requisiti. Questo fa scendere il processo dal *livello di astrazione di progettazione* al *livello di astrazione dell'implementazione.*

Vantaggi Model-Driven Test Design (MDTD):

- Le definizioni e le applicazioni dei criteri di test sono indipendenti dal livello di test (unit, integration, system, ecc...)
- L'astrazione semplifica i test

La separazione tra la progettazione dei test basata sull'uomo e quella basata sui criteri è una distinzione importante e il riconoscimento che si tratta di attività complementari, non in competizione, è la chiave di questo libro.

1.4 Testare l'adeguatezza

Lo scopo dell'attività di testing è rilevare i failures.

- Un test suite T *rileva un failure* se il programma P è **incorretto** per *almeno* un test case di T
- Un test suite T **ha successo** se rileva uno o più failures del programma P
- Un test suite T è **inadeguato** se esistono faults di P che nessun test cases di T è capace di rilevare
- Un test suite T è **ideale** se l'assenza di failures implica l'assenza di faults

1.5 Assenza della proprietà di continuità e testing esaustivo

Quando parliamo di software, stiamo parlando di *sistemi discreti* (e non continui). Piccole variazioni dei valori di input possono portare a risultati incorretti. La proprietà di continuità non vale.

Il testing esaustivo (*exhaustive testing* - caso ideale) esegue tutti i possibili comportamenti di un sistema software. Se un test esaustivo non mostra alcun errore, il programma è corretto.

È possibile testare in maniera esaustiva?

- Se il programma non ha rami e non ha input, allora la test suite esaustiva esiste ed è composta da un solo test case;
- Se il programma invece ha condizioni e loops, allora sarebbe impossibile testare tutte le combinazioni, perché condizioni annidate in loop generano percorsi infiniti.

1.6 The Seven Principles of Testing

- **Principio 1**: Testing mostra la presenza di difetti, non la loro assenza
- **Principio 2**: Exhaustive testing è (generalmente) impossibile
- **Principio 3**: Il Testing anticipato salva tempo e soldi
- **Principio 4**: I difetti tendono a formare ammassi
- **Principio 5**: Attenzione al paradosso dei pesticidi. Il codice "tende ad adattarsi" al testing: in altre parole, per trovare nuovi difetti introdotti nel tempo è sempre necessario progettare nuovi test
- **Principio 6**: Testing is context-dependent. Non ci sono strategie di testing universali applicabili in ogni contesto
- **Principio 7**: L'assenza di errori è una falsa credenza (conseguenza del Principio 1 e del Principio 2)

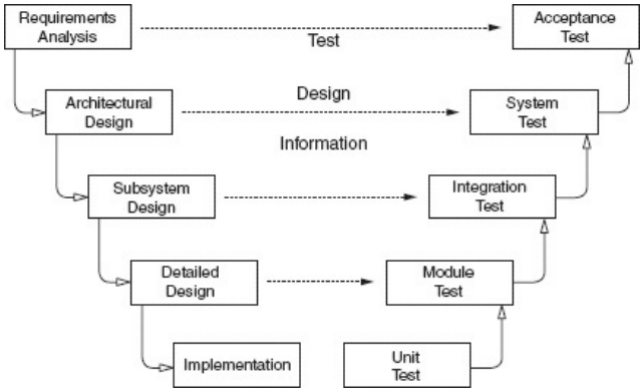
1.7 Livelli di testing basati sui Software Activity

I test possono essere derivati da requisiti e specifiche, artefatti di design o il codice sorgente.
Un diverso livello di test accompagna ogni distinta attività di sviluppo del software:

- **Acceptance Testing** (*test di accettazione*): valutare il software rispetto ai requisiti o alle esigenze degli utenti
- **System Testing**: valutare il software rispetto alla progettazione architettonica e al comportamento generale
- **Integration Testing**: valutare il software rispetto alla progettazione del sottosistema
- **Module Testing**: valutare il software rispetto alla progettazione dettagliata
- **Unit Testing**: valutare il software rispetto all'implementazione

1.7.1 The V Model

Il V Model evidenzia le relazioni tra le attività di sviluppo software e i livelli di test.
Le informazioni per ogni livello di test sono tipicamente derivate dall'attività di sviluppo associata. È una buona pratica progettare i test in maniera concorrente per ogni attività di sviluppo, anche se il software non sarà in una forma da permetterne l'esecuzione fino alla fase di implementazione. La progettazione anticipata dei test permette di rilevare i difetti nelle decisioni di design in anticipo.



- La fase dell'**analisi dei requisiti** cattura le esigenze del cliente. L'**acceptance testing** è progettato per determinare se il software completato soddisfa effettivamente questi requisiti. I test di accettazione devono coinvolgere utenti o altre persone che hanno una forte conoscenza del dominio;
- La fase di **progettazione architettonica** individua componenti e connettori che insieme realizzano un sistema le cui specifiche sono destinate a soddisfare i requisiti precedentemente individuati. Il **System Testing** è progettato per determinare se il sistema assemblato soddisfa le sue specifiche. Presuppone che i pezzi funzionino individualmente e chiede se il sistema funziona nel suo insieme;
- La fase di **progettazione del sottosistema** specifica la struttura e il comportamento dei sottosistemi, ognuno dei quali ha lo scopo di soddisfare alcune funzioni nell'architettura complessiva. L'**integration testing** è progettato per valutare se le interfacce tra i moduli in un sottosistema comunicano correttamente. Il test di integrazione presuppone che i moduli funzionino correttamente;
- La fase di **design dettagliato** determina la struttura e il comportamento dei singoli moduli. Un **modulo** è una raccolta di unità correlate. Il **Module Testing** è progettato per valutare i singoli moduli in isolamento;
- La fase di **implementazione** è la fase dello sviluppo software che produce codice. Una unità di programma è una funzione, procedura, metodo o classe. Lo **Unit Testing** è progettato per valutare le unità prodotte dalla fase di implementazione;
- I **test di regressione** (non mostrato nella figura del V Model) viene eseguito dopo che sono state apportate delle modifiche al software, per garantire che il software aggiornato possieda ancora la funzionalità che aveva prima degli aggiornamenti.

Test Automation

2 Test Automation

Definizione di automazione dei test:

- **Test automation**: l'uso di software per controllare l'esecuzione del test, il confronto dei risultati effettivi con i risultati previsti, l'impostazione delle condizioni preliminari del test e altre funzioni di controllo del test e report del test.

Vantaggi dei test automatici:

- Riduce il costo di implementare i test;
- Riduce l'errore umano;
- Semplifica i test di regressione consentendo l'esecuzione ripetuta di un test.

Differenza tra *excise tasks* e *revenue tasks*:

- **Excise task**: task che non contribuiscono alla soluzione di un problema.
  - compilare codice
  - eseguire un test

  Questi task **devono essere automatizzati**
- **Revenue task**: task che contribuiscono direttamente alla soluzione di un problema
  - determinare quali metodi sono appropriati per una classe Java

L'automazione dei test è un insieme di tecniche e tecnologie che ti permettono di automatizzare (anche parzialmente) alcune attività del processo di testing. L'automazione dei test può coinvolgere:

- La progettazione e la definizione di test case
- La preparazione e l'esecuzione dei test
- La valutazione dei risultati dei test case
- La valutazione dell'efficacia di una test suite o di tecniche di testing

2.1 Software Testability

Un requisito di qualità del software è la sua **testabilità**: quant'è facile testare il software. Definizione:

- **Testabilità**: il grado in cui un sistema o un componente facilita la definizione di criteri di test e l'esecuzione dei test per determinare se tali criteri sono stati soddisfatti.

La testabilità è in gran parte determinata da due problemi pratici: come fornire i valori del test al software (*controllabilità*) e come osservare i risultati dell'esecuzione del test (*osservabilità*).

- **Osservabilità del software**: quant'è facile osservare il comportamento di un programma in termini di output, effetti sull'ambiente o altri componenti software o hardware;
- **Controllabilità del software**: quant'è facile fornire ad un programma gli input, in termini di valori, operazioni e comportamenti (interfaccia grafica, dipendenza temporale, sincronizzazione remota sono motivi di scarsa controllabilità).

La testabilità è fondamentale per l'automazione dei test perché gli script di test devono controllare l'esecuzione del componente sottoposto a test e osservare i risultati del test.

2.2 Componenti di un test case

Un test case, in generale, è un artefatto costituito da più parti.

Una **specifica di test case** è un insieme minimale di informazioni capace di descrivere una specifica di un test case.

Le parti di un test case non sono standardizzate, ma le seguenti sono le più comuni:

- **ID**: identifica univocamente un test case
- **Precondizioni**: precondizioni necessarie affinché il test possa essere eseguito
- **Input**: input del test case
- **Output atteso**: output atteso al termine dell'esecuzione
- **Postcondizioni**: ipotesi che devono essere verificate dopo l'esecuzione del test

Dopo l'esecuzione, abbiamo:

- I **valori di output**
- **Risultato** diviso in:
  - `Success`: se le precondizioni e le postcondizioni sono vere e l'output è uguale all'output aspettato
  - `Failure`: se le precondizioni e le postcondizioni sono vere e l'output NON è uguale all'output aspettato
  - `Not Applicable (N/A)`: se almeno una precondizione o postcondizione non è vera

L'input e l'output possono essere descritti in varie forme.

2.3 Oracolo

L'**oracolo** conosce esattamente l'output atteso per ogni test case.

Nella specifica di un test case, l'oracolo è rappresentato dalla colonna "Output atteso".

Tipi di oracolo:

- **Oracolo umano**: valuta il successo dei casi di test sulla base dei requisiti e del suo personale giudizio
- **Oracolo software**: è un software che ha esattamente lo stesso comportamento del software che deve essere testato
  - Una versione di bubblesort può essere usato per testare un quicksort
  - Una versione precedente del software può essere l'oracolo per valutare il comportamento di un nuovo software offrendo la stessa funzionalità
- **Oracolo implicito**: testing contro i crash, l'oracolo trova un fallimento ogni volta che un crash accade
- **Oracolo estratto dalle specifiche formali**: se i requisiti sono espressi in un modo formale, allora l'oracolo può essere automaticamente derivato da queste ultime.

2.3.1 Il comparatore - comparare l'oracolo con il risultato

Non sempre è facile confrontare il risultato con l'output atteso. Il comparatore confronta il risultato con il valore prodotto dall'oracolo.

- Se l'oracolo fornisce valori esatti, il comparatore deve semplicemente fare un confronto di bit.
- Se l'oracolo fornisce un insieme di valori validi, il comparatore deve controllare se il risultato appartiene a questo insieme.
- Se l'oracolo fornisce valori non esattamente rappresentati dal computer (ad esempio π) il comparatore deve confrontare la differenza tra il risultato ottenuto e una rappresentazione approssimativa del valore dell'oracolo.
- Se il valore atteso è una immagine, il comparatore deve confrontare l'immagine di output con un insieme di immagini per argomentare se l'immagine è sufficientemente simile all'immagine attesa.

2.3.2 Scrivere efficacemente oracoli di test

Un test deve codificare i risultati attesi, comunemente noti come *test oracle*, affinché possa essere automatizzato.

- Gli oracoli di test devono mostrare un'efficacia *rivelabilità*, trovando un equilibrio tra il controllare troppo e il controllare troppo poco.
- Gli oracoli di test devono risolvere *problemi di osservabilità* per ottenere valori che potrebbero non essere facilmente disponibili.

2.3.3 Cosa dovrebbe essere controllato?

Lo stato di uscita del software da testare è tutto ciò che viene prodotto o modificato dal software in fase di test (output sull schermo, file, database, messaggi e segnali) e non solo le dichiarazioni di ritorno esplicite.

Quanto stato di uscita deve controllare un test automatizzato? Una **strategia dell'oracolo di test** è una regola o un insieme di regole che specificano quali stati del programma controllare. Una strategia dell'oracolo di test presenta due caratteristiche generali:

- *Precisione*: quanta parte dello stato di uscita deve essere controllata
- *Frequenza*: si riferisce a quando e quanto spesso deve essere controllato lo stato di uscita

È sempre e comunque importante **controllare alcuni output** e **controllare gli output giusti**. Bisogna controllare gli output che sono direttamente influenzati dal test. Inoltre, **non è necessario controllare molti output** (una bassa precisione va bene). I ricercatori hanno rilevato che non solo è necessario controllare l'output giusto, ma che la verifica di ulteriori output aumenta di pochissimo la rivelabilità. Ancora, **non è necessario controllare più volte lo stato dell'uscita** (una bassa frequenza va bene).

2.4 Alcuni requisiti di qualità dei test

- **Accuracy (precisione)**: il successo di un esecuzione di un test case dovrebbe essere valutato in un modo obiettivo
- **Repeatability (ripetibilità)**: i casi di test devono essere ripetibili
- **Fault localization**: il testing dovrebbe aiutare a **localizzare i fault**
- **Tracciabilità**: la tracciabilità dei test permette di collegare tra lodo i casi di test, le funzionalità del software, codice testato, report di test precedenti e così via...
  - si ottiene imponendo attività di etichettatura e con un database di tracciabilità
  - aiuta a rilevare fault perché limita le porzioni di codice interessate dai casi di test non riusciti
  - permette di prevedere il possibile impatto di una modifica
- **Efficacia**: l'obiettivo di un test è di scovare il maggior numero di fallimenti. Una suite di test `T1` è più efficace di una suite di test `T2` se rileva più differenti faults.
- **Efficienza**: una test suite `T1` è più efficiente di una test suite `T2` se riesce a trovare lo stesso numero di differenti fallimenti con meno sforzo. Lo sforzo di testing può dipendere da:
  - numero di casi di test
  - complessità totale dei casi di test
  - tempo necessario per eseguire i casi di test
  - costo delle risorse (umane e macchine) necessarie ad eseguire i casi di test

2.4.1 Efficienza vs Efficacia

Aumentare il numero di test case di una test suite può aumentarne l'efficacia ma diminuirne l'efficienza. I due requisiti di qualità sono opposti. Si dà importanza a uno dei due fattori in base alle proprie esigenze (spesso in base al tempo a disposizione e in base al costo).

2.4.2 Misurare efficienza ed efficacia

- **Efficacia**: numero di faults trovati / numero di faults esistenti
- **Efficienza**: numero di casi di test che hanno riscontrato errori / numero di casi di test eseguiti

Il numero di faults esistenti è, generalmente, sconosciuto.

Il numero di casi di test eseguiti non è una misura dello sforzo relativo alla loro esecuzione.

Tali misure possono essere prese in considerazione per misure relative. Dati due test suite `T1` e `T2`, due possibili misure di efficacia ed efficienza sono:

- **Efficacia relativa** (`T1`, `T2`): numero di differenti faults trovati da `T1` / numero di differenti faults trovati da `T2`
- **Efficienza relativa** (`T1`, `T2`): numero di casi di test di `T1` che trovano faults / numero di casi di test di `T2` che trovano faults

2.4.3 Misure approssimative: code coverage

Sono necessarie misure approssimative di efficacia ed efficienza utili nei casi in cui ci sono pochi o nessun bug. Una misura approssimativa di efficacia è la **copertura di codice**:

- copertura delle linee di codice
- copertura dei metodi
- copertura delle classi

Se una LOC contiene un fault e non viene eseguita, allora *sicuramente* il fault non sarà trovato.

Se una LOC contiene un fault e qualche test la esegue, allora *è possibile* che il fault causerà un fallimento.

In generale, più è la quantità di codice che eseguiamo, più possiamo sperare di trovare i faults.

Se due test suite `T1` e `T2` trovano lo stesso numero di faults (anche zero), e `T1` copre più LOCs di `T2`, allora possiamo dare più confidenza alla test suite `T1`.

2.5 Unit Testing

Un'**unità** per definizione è la parte più piccola del software da testare.

Le unità possono essere funzioni, metodi, classi, packages o l'intero sistema.

Il **test delle unità** può essere fatto in modalità **black box** o in modalità **white box**.

Definizione di *test automation framework*:

- **Test Framework**: un insieme di assunzioni, concetti e strumenti che supportano l'automazione dei test.

Black Box Testing

3 Black Box Testing

3.1 Specification-based testing

Il **testing basato sulle specifiche** è una tecnica di validazione dove vari test sono progettati per ogni requisito.

In generale, il testing basato sulle specifiche è incluso nella grande famiglia del black box testing.

Il black box testing è una tecnica di testing nel quale i casi di test sono progettati senza nessuna conoscenza interna riguardo il software da testare: è basato solo ed esclusivamente sulle sue specifiche.

Esistono due approcci complementari:

- Nel **testing basato sui requisiti**, i test cases sono progettati sulla base delle analisi dei comportamenti che vuoi che si verifichino. La difficoltà sta nel trovare i valori di test che sono capace di triggherare questi comportamenti
- Nelle **tecniche di partizione del dominio di input** i casi di test vengono progettati per coprire combinazioni di valori di input. La difficoltà consiste nell'essere capace di coprire, possibilmente con il minimo numero di casi di test, i comportamenti dell'applicazione

3.2 Data Driven Testing

Nel **Data Driven Testing** i test hanno una struttura comune e la strategia di testing si concentra sulla scelta dei dati di input.

3.3 Input Domain Model (IDM)

Un modo comune per partizionare l'input space è iniziare a considerare il dominio di ogni parametro separatamente, partizionando il dominio del parametro in *blocchi* e, successivamente, combinare i blocchi per ciascun parametro. Ogni blocco contiene un insieme di valori.

Questo processo viene chiamato **input domain modeling**.

Ogni partizione è basata su una *caratteristica* C degli input del programma. Ogni caratteristicha C permette di definire una partizione che rispetta la proprietà di completezza e disgiunzione dei blocchi. Dal punto di vista della caratteristica scelta C, tutti i valori che appartengono ad uno stesso blocco sono considerati equivalenti.

L'input domain modeling permette quindi di rappresentare lo spazio di input del sistema da testare in un modo astratto.

Esistono due approcci differenti all'input domain modeling:

- *Approccio basato sull'interfaccia*: le caratteristiche vengono scelte direttamente dai parametri di input del programma da testare. Meccanico. È facile identificare le caratteristiche. I casi di test astratti così ottenuti sono facili da implementare perché ogni caratteristica si limita a un singolo parametro. Le precondizioni e le postcondizioni sono ottime sorgenti per progettare le caratteristiche.
- *Approccio basato sulle funzionalità*: le caratteristiche vengono scelte in base alle funzionalità. La modellazione è basata sui requisiti e non sull'implementazione. Questo permette di introdurre della semantica o conoscenza del dominio all'interno dell'Input Domain Model. Rispetto all'approccio basato sull'interfaccia, progettare le caratteristiche con questo approccio può risultare arduo, soprattutto quando le specifiche sono informali e incomplete. L'idea è suddividere lo spazio di input utilizzando il *dominio di conoscenza* del problema.

Una volta scelto l'approccio e una volta costruito l'Input Domain Model, differenti *criteri di copertura* sono disponibili per decidere quale combinazione di valori usare per testare il software.

3.3.1 Scelta dei blocchi dei valori

Una volta progettate le caratteristiche e partizionato l'input space, è necessario scegliere i blocchi e i valori da questi blocchi.

Scegliere più blocchi produce più casi di test.

3.3.2 Combination Strategies Criteria

- **All Combinations Coverage (ACoC)**: devono essere utilizzate tutte le combinazioni di blocchi di tutte le caratteristiche. Il numero di casi di test è il prodotto del numero di blocchi per ogni partizione.
- **Each Choice Coverage (ECC)**: un valore da ogni blocco di ogni caratteristiche deve essere usato in almeno un caso di test.
- **Pair-Wise Coverage (PWC)**: un valore da ogni blocco di ogni caratteristica deve essere combinato con un valore di ogni blocco di ogni altra caratteristica.
- **T-Wise Coverage (TWC)**: un valore da ogni blocco per ogni gruppo di t caratteristiche deve essere combinato.

4 Graph Coverage

Uno dei criteri di copertura del test è la **copertura del grafo**.

- **Graph Coverage Definition**: dato un insieme di *requisiti di test* (TR) per un criterio del grafo C, un insieme di test T soddisfa C sul grafo G se e soltanto se per ogni test requirement tr in TR, esiste almeno un percorso di test p in $path(T)$ tale che p soddisfa tr.

4.1 Structural Coverage Criteria

Un criterio di copertura del grafo viene definito specificando un insieme di requisiti di test, TR. I requisiti di test vengono prodotti da un criterio di copertura del grafo. I requisiti di test contengono predicati.

- **Node Coverage (NC)**: TR contiene ogni nodo raggiungibile in G. Un test set T soddisfa TR se ogni nodo è raggiungibile.
- **Decision Coverage (Branch Coverage or Edge Coverage)**: richiede che ogni arco del grafo G sia incluso almeno una volta in un test case path. Deve esistere almeno un test case in cui la decisione è `true` e almeno un test case in cui la decisione è `false`.

4.2 Percorsi linearmente indipendenti

Un percorso in un Control Flow Graph (CFG) è una esecuzione dal nodo iniziale al nodo finale. Un percorso è detto **linearmente indipendente** (rispetto all'insieme dei percorsi) se introduce almeno un nuovo insieme di nodi o un nuovo arco.

Un percorso in un CFG è linearmente indipendente se attraverso almeno un arco non ancora attraversato. L'insieme di tutti i percorsi linearmente indipendenti di un programma formano i percorsi di base; tutti gli altri percorsi sono generati da una combinazione dei percorsi di base.

Dato un programma, l'insieme di percorsi di base non è unico.

Il numero di percorsi linearmente indipendenti di un programma è uguale al **numero ciclomatico di McCabe**:

- $V(G) = E - N + 2$ dove E è il numero di archi e N il numero di nodi
- $V(G) = P + 1$ dove P è il numero di predicati
- $V(G) =$ numero di cicli chiusi $+ 1$

Le test suite che eseguono i percorsi di base garantiscono l'esecuzione di ogni istruzione e di ogni decisione almeno una volta. Se non consideriamo il numero di volte che un ciclo può essere eseguito, allora anche tutti i percorsi.

4.3 Path Coverage Criteria

Il numero di test case necessari per coprire tutti i percorsi indipendenti è uguale alla complessità ciclomatica.

Il numero di possibili percorsi è generalmente illimitato. La presenza di un loop può causare l'esistenza di un numero di percorsi illimitato che differiscono per il numero di esecuzioni del loop.

Alcuni percorsi sono **infattibili**. Ma dimostrare che un percorso è infattibile è un problema non decidibile.

Il criterio di copertura dei percorsi non è possibile.