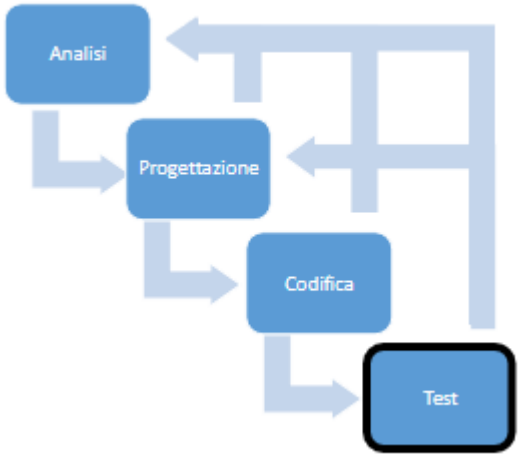
# 6 - Testing

Il **testing** è la fase di **verifica sistematica** della correttezza di un software ed è parte integrante dei processi di sviluppo di un software. La scala delle azioni da eseguire durante la creazione di un software è la seguente:




Il test ci permette di controllare solo la correttezza di un software, in caso dovessimo risolvere degli errori entreremo in una nuova attività chiamata **debugging**, nella quale è preferibile non entrare, un buon programmatore deve cercare di risolvere i problemi nella fase di testing, nella *debugging fase* i tempi di sviluppo si allungano vertiginosamente.
Il **debugging** è l'attività di rimozione di eventuali problemi emersi dopo il testing.

La **verifica della correttezza** non è l'unico aspetto che si può valutare con il testing, possiamo anche valutare:

- Le **prestazioni**: se il codice svolge i suoi compiti in un tempo **sufficiente**
- L'**usabilità**: se l'interazione con utente avviene in modo chiaro
- **Portabilità del codice**: se il codice è eseguibile su **macchine diverse** senza problemi
- **Accettazione**: se il codice risponde perfettamente alle richieste dell'utente

## Casi di test

Il problema principale che si affronta nel testing del programma è la definizione dei **casi di test**, ovvero le situazioni in cui il programma può presentare degli errori. Da questo problema nascono due teoremi fondamentali.

1. **TEOREMA DI HOWDEN**: Non esiste un algoritmo che, dato un programma P qualsiasi, generi un piano di test ideale e finito
2. **TEOREMA DI DIJKSTRA**: Il test di un programma può rilevare la presenza di malfunzionamenti, ma **non dimostrarne** l'assenza.

## Metodologie

Il modo in cui andiamo a testare il nostro programma si divide in due grandi classi:

- **TEST FUNZIONALE** → Test Black Box
- **TEST STRUTTURALE** → Test White Box
  Nel primo caso, la selezione dei casi di test dipende dagli **I/O**, mentre nel secondo caso dipendono dalla **struttura del programma**.

# Blackbox

Questa metodologia di test verifica la correttezza del programma **valutando l'output prodotto**, senza entrare nel merito di come ottenere il risultato, si verificano i **risultati attesi**.
Esistono tre tecniche principali:

- Verifica delle **condizioni limite**
- Definizione di **classi di equivalenza**
- **Error Guessing**

## Verifica delle condizioni limite

Prima di verificare le condizioni limite bisogna analizzare il programma e capire **quali sono i possibili valori limite** e valutare **tutte le possibili condizioni**.
*Esempio condizioni limite*:

```
int i = 0;
char s[MAX];
while ((s[i] = fgetc(file)) != '\n' && i < MAX-1)
      i++;
s[--i] = '\0';
```

**Quali sono le condizioni limite?**
**Tipicamente le condizioni limite corrispondono a:**

- Input vuoto
- Input «pieno» (numero massimo di caratteri disponibili, es.)
- Input errato
- Condizioni di uscita dai cicli

In un codice bisogna testare prima:

- Le parti più semplici
- Le parti più frequentemente utilizzate

## Classi di equivalenza

**Non è possibile** testare tutti i possibili valori di un programma.
In questo caso vengono usate le classi di equivalenza:
Una **classe di equivalenza** è un insieme di valori di input per i quali l'algoritmo si comporta in modo analogo.
Ovvero l'algoritmo con una data fascia di valori ci farà capire che risultato otterremo con

qualsiasi dato di quella fascia se il risultato è coerente con ciò che l'algoritmo dovrebbe risolvere, mentre i valori al di fuori della fascia possono produrre comportamenti differenti.
Un algoritmo deve avere **almeno due classi di equivalenza**.
Questo metodo ci conferma la buona gestione dei problemi e creazione del codice ma bisogna sempre gestire i **casi limiti** come prima cosa.

Per **progettare** le classi di equivalenza di assegnano dei nomi alle diverse classi, successivamente per ogni classe di equivalenza, si definiscono dei valori di esempio che appartengano al range. Si testa l'algoritmo con un valore valido e con uno non valido e si verifica che l'esito sia quello **atteso**, in caso non sia un ottimo risultato bisogna fare debugging.
Normalmente si è abituati a testare il codice solamente con i valori validi, è fondamentale invece verificare il comportamento d'innanzi ai **valori non validi**, perché tipicamente è proprio lì che sono presenti i bug.

### Programmazione difensiva

L'utilizzo della programmazione **difensiva** è un metodo efficace per gestire i casi limite e i comportamenti **anomali**. Capire prima i casi limite e scrivere del codice che prevenga eventuali valori errati può aiutare a limitare i bug del programma.

```
if(vote<0 || vote>MAX_VOTE){
    exam=NOT_VALID;
} else if(vote<=18) exam= NOT_PASSED;
...
```

### Random Guessing

Il random guessing **non è una tecnica** vera e propria poiché dipende dall'intuito del programmatore, l'errore viene rilevato con l'**esperienza**, durante la scrittura del codice si individua il codice normalmente senza eseguire un test vero e proprio.

## White box

Verifica la correttezza del programma **analizzando la strutturazione del codice sorgente**, ovvero la selezione dei casi di test non dipende dai valori di I/O ma serve a verificare il corretto funzionamento del programma in tutti i **percorsi possibili** che possono verificarsi, si studiano tutte le possibili soluzioni che possono accadere tra i vari cicli, if e funzioni e le loro possibilità.
Il concetto sui cui è basato il testing white box è detto **concetto di copertura**.

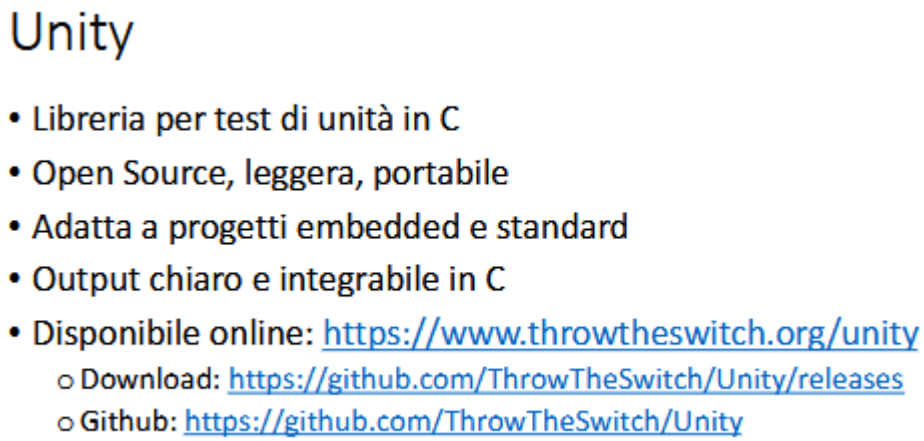
# Accorgimenti generali

A prescindere della metodologia scelta, bisogna seguire delle precise regole generali, detta **verifica incrementale**:

- **Test di pari passo con l'implementazione**
- **Test di unità elementari**, ovvero prima testare le funzioni e successivamente le parti di programma che usano quella funzione, grazie alla programmazione modulare si può testare ogni funzione separatamente dalle altre.
  Ogni programma deve essere consegnato con un **piano di test**, ovvero la descrizione di tutti i test eseguiti per la risoluzione del programma e dei suoi problemi.

# Test di unità

Essa è una tecnica di progetto e sviluppo del software, serve ad evidenziare che le singole unità software sviluppate siano corrette e pronte all'uso, testando singole porzioni di codice alla volta.
Un **unità software** è una procedura o una funzione, per eseguire questo test di unità si scrivono degli **unit test** ovvero degli ipotetici "contratti scritti" che la porzione di codice testata deve **assolutamente soddisfare**.
In principio, essendo un framework fresco, è stato creato per Java sotto nome di **JUnit**, è stato proposto successivamente negli altri linguaggi, dando vita all'ecosistema **xUnit**. Un framework di testing automatizza l'esecuzione dei test, verifica i risultati attesi e segnala gli errori in modo sistematico. In *C* per poter usare questi tipi di test si usa la libreria **Unity**.
*Come usare Unity*:




## Utilizzo di Unity

Creare un file sorgente *C* che verrà utilizzato per eseguire i test.

```c
#include "unity.h"
#include "unitu.c"
#include "unity_internals.h"

void setUp(void){
    //set stuff up here -- Questa funzione viene eseguita prima dei test
}
```

```
void tearDown(void){
    //clean stuff up here -- Questa funzione viene eseguita al termine dei test
}
```

Queste funzioni **devono essere dichiarate ma possono rimanere anche vuote**.
Possono essere create diverse funzioni **Test** che verranno chiamata dalla funzione main.
Nel main si possono usare quanti si vogliano della direttiva `RUN_TEST(<nome funzione da testare>)`, per poter testare le varie funzioni da testare.

## Scrivere i metodi di test

Ciascun metodo di test contiene al suo interno delle **asserzioni**.
Un'**asserzione** è una **funzione che verifica una condizione logica** e restituisce Vero o Falso.
*SINTASSI IN UNITY*:

```
TEST_ASSERT(espressione)
TEST_ASSERT(0==0) //true
TEST_ASSERT(0==1) //false

TEST_ASSERT_EQUAL(espressione, valore) -> //Restituisce true se il valore è quello che indichiamo
TEST_ASSERT_NOT_EQUAL(espressione,valore) -> //Restituisce true se il valore calcolato NON è quello che indichiamo
```

*Esempio Asserzione*:
Supponiamo di avere una funzione calcolaBMI che calcola il BMI:
`TEST_ASSERT_EQUAL(calcolaBMI(200,100), 25)`
Equivale a dire: Asserisco che il BMI di un individuo con altezza pari a 200 e peso pari a 100 è uguale a 25.
Equivale a dire che se l'asserzione restituisce **true** il valore restituito dall'espressione/funzione è funzionante.

I parametri con cui testare la funzione sono le **condizioni limite e le classi di equivalenza** trattate in precedenza, mentre il valore atteso è il comportamento che ci aspettiamo. Per una funzione il formato che segue è il seguente:
`TEST_ASSERT_EQUAL(nomeFunzione(parametri),valoreAtteso)`.
Queste funzioni di Test non sono le uniche di Unity, ma c'è un intera lista di test specifici per array, interi, stringhe, strutture ecc... sul seguente link UNITY TEST

## Esempio Setup e Teardown

Creiamo un nuovo file sorgente "tunity_math.c" che conterrà l'esecuzione del piano di test.

```
static int *buffer;

void setUp(void){
    buffer=malloc(10*sizeof(int));
    for(int=0;i<10;i++) buffer[i]=i;
}

void tearDown(void){
    free(buffer);
}

void test_valore_ceentrale(void){
    TEST_ASSERT_EQUAL(5,buffer[5]);
}

void test_somma_buffer(void){
    int somma=0;
    for(int i=0; i<10; i++) somma+=buffer[i];
    TEST_ASSERT_EQUAL(45,somma);
}

int main(void){
    UNITY_BEGIN();
    RUN_TEST(test_valore_centrale);
    RUN_TEST(test_somma_buffer);
    return UNITY_END();
}
```