7 - Debugging

Il **debugging** è il processo di riconoscimento e rimozione dei bug
Un **bug** è un errore presente internamente nel software
Gli errori nel software possono essere di due tipi:

- **sintattici** $\rightarrow$ sono errori rilevati sempre dal compilatore in fase di **compilazione**, come un `;`
- **semantici** $\rightarrow$ sono errori **non rilevabili**, come l'uso di un `=` al posto di `==`
  I bug sono presenti anche nei programmi più semplici e l'attività di debug è altamente complessa e richiede un tempo imprevedibile e a volte anche molto lungo, per questo è fondamentale adottare dei metodi che **riducano la presenza di bug** e l'attività prolissa di **debugging**.
  La grandezza del problema è **direttamente proporzionale** alla difficoltà di ricerca degli errori.

Il debugging si concentra sulla **rimozione degli errori semantici**, i vari errori semantici che si possono incontrare sono:

- **Interruzione inattesa** del programma
- **Loop** del programma
- termine con **risultati errati**

Il trasferimento di un bug nei passi successivi del ciclo di sviluppo software **fa crescere il costo di debugging esponenzialmente**.

Come attuare il debugging

La fase di debug si basa su tre fasi:

- trovare le istruzioni che causano il bug
- scovarne il motivo
- correggere il codice
  La prima fase è la più **complessa**, prima di adottare il debugging si fa riferimento a delle **linee guida** che sono fondamentali da **seguire** per individuare le istruzioni che generano il bug.

Linee guida:

1. **SUPPORTO DEL COMPILATORE**:
   Molti compilatori emettono un **warning** per analizzare una possibile zona soggetta a bug, anche se apparentemente i warning non sono un problema, la maggior parte dei

bug si celano dietro essi; un altro accorgimento e **non accettare** i consigli dell'IDE senza capirne le direttive proposte.

2. **PATTERN FAMILIARI**:
   Se si stanno usando comandi già utilizzati o noti basterà **confrontare il pattern** con la nostra istruzione per comprende l'errore.
3. **ESAMINARE UN CODICE SIMILE**:
   Se un bug si verifica in una porzione di codice allora è altamente probabile che si trovi un altro bug annidato all'interno di un codice simile, questo spesso avviene nei cicli e nelle condizioni di uscita, questo problema viene chiamato **problema del copy-and-paste**. Una buona progettazione del codice elimina la presenza di questi bug, dato che porzioni di codice simili possono essere richiamate attraverso una singola **funzione**, se il problema persiste sarà presente solamente nella singola funzione.
4. **BACKWARD REASONING**:
   Quando viene individuato un bug, occorre pensare in maniera **bottom-up**, partendo dal risultato si risale alla catena della cause che possono verificarlo. Una possibile causa di questa catena sarà errata. Un buon metodo per il **backward reasoning** è la stesura di un codice **leggibile**.
5. **SVILUPPO INCREMENTALE**:
   Testare il programma man mano che venga ultimato, se i test all'istante t sono funzionanti, questo vuol dire che il problema si annida nel codice tra la fine dell'istante t e l'istante $t+1$. La progettazione modulare ci permette di individuare meglio i bug in questo caso.
6. **LEGGERE E SPIEGARE IL CODICE**:
   Il codice dev'essere compreso sia dalla macchina che da chi lo scrive, la leggibilità del codice e la sua conoscenza è fondamentale, conoscendo il nostro codice saremo in grado di **spiegarlo** e capire cosa si sta facendo, se non riusciamo a spiegarlo vuol dire che è presente un indice di complessità e quindi è altamente probabile che il bug si trovi nella zona di codice non spiegabile. E' consigliabile quindi comprendere sempre il codice e cercare di spiegarlo ad altri.
7. **RENDERE RIPRODUCIBILE UN BUG**:
   Bisogna individuare tutti i campi che comportano alla fuoriuscita del bug, se il bug non si presenta ad ogni **generazione** del codice sarà ancora più **complesso** riuscire a capirne il motivo.
8. **DIVIDE ET IMPERA**:
   Individuare tutte le condizioni **minimali** che permettono la nascita del bug, quindi è fondamentale attuare i **casi limite dei test**, poiché come sappiamo il test dei casi limite ci permette di individuare le situazioni che portano il programma in **error**.
   La condizioni minimali **facilitano** nella localizzazione di un bug, per questo bisogna conoscere a priori le condizioni minimali del problema prima di scriverlo sotto forma di codice.
9. **RICERCA DI REGOLARITA'**:
   Alcuni bug si presentano con una **determinata regolarità**, in questo caso occorre capire

qual è il meccanismo con cui si genera questa frequenza regolare, ad esempio dando determinati input, solo input negativi, o stringhe più lunghe di tot.
Comprendere la regolarità aiuta a comprendere la **natura del problema**.

10. **STAMPE AUSILIARIE**:
    I così noti messaggi **trappola**, ci permettono di tenere traccia di come **cambiano le variabili** nel corso del programma stampandole ad ogni utilizzo, utile specialmente nelle situazioni in cui non è possibile attuare il debugger, come nei sistemi distribuiti. Si adotta un metodo di ricerca **binaria**, ovvero se dopo un determinato output la variabile non presenta errori, il problema sarà presente nella metà di programma successiva a quell'output, ripetere questo procedimento **iterativamente**.
    Le stampe ausiliarie devono essere poste come **commenti** dopo la rivelazione del bug, non devono essere presenti nel codice visibile. Nelle situazioni di massima complicanza si richiede anche l'uso del **logging (log)** il `log` è la **registrazione** di tutte le operazioni effettuate dal programma.

Debugger

Se l'uso di queste linee guida non porta a una risoluzione del problema si **deve usare** uno strumento denominato **debugger**.
Un debugger riesce a guardare all'**interno del programma** durante l'esecuzione:

- **Tracing** del programma: esecuzione istruzione per istruzione
- Visualizzazione del contenuto delle **variabili**
- **Valutazione dinamica** delle espressioni
- **Breakpoint**
- **Stack trace**: sequenza di chiamate a funzione effettuate dal programma
  Bisogna imparare ognuno di questi strumenti alla perfezione per aumentare la **produttività** nella programmazione.

Debug in VSCode

Per eseguire l'azione di debug in VSCode si usa l'opzione *"Start Debugging"*, richiamabile con **F5**.
Per controllare nella fase di debug **l'esecuzione del programma** usiamo i vari pulsanti che ci presenta la barra di ispezione fornita da VSC:

- **Continua e Pausa**: Esegue il debug dal punto corrente riga per riga ed esegue il programma fino al breakpoint
- **Step Over**: Esegue la subroutine (funzione) in un singolo passo senza entrarci dentro e passerà al blocco di istruzioni successivo
- **Step Into**: Entra nella funzione per eseguire il debug riga per riga

- **Step Out**: : Quando ci si trova all'interno di una subroutine, ritorna al contesto di esecuzione precedente completando le righe rimanenti nella subroutine corrente come se fosse un singolo comando
- **Restart**: Termina l'esecuzione e o ricomincia il debug o termina totalmente l'esecuzione

Breakpoint

L'esecuzione del programma **Step-by-Step** richiede un dispendio di tempo eccessivamente lungo, per questo si usa una valida alternativa, ovvero i **breakpoint**. I breakpoint identificano i punti del programma che vogliamo **monitorare** e si usano in corrispondenza di espressioni **critiche**.
Il programma viene eseguito normalmente fino alla riga segnata dal breakpoint, dopo di che il debugger si attiva e comincia a monitorare lo stato della macchina e delle variabili coinvolte.
Per **inserire un breakpoint** è sufficiente un **doppio-click** sul numero che identifica la riga dell'istruzione, o eventualmente tasto destro sulla riga→*AddBreakpoint*. Le istruzioni segnate da un breakpoint vengono **evidenziate** accanto al numero della riga. Esistono vari tipi di breakpoint in base alle necessità e alle complessità, come il *Conditional Breakpoint* o il *Triggered Breakpoint*.

Conditional breakpoint

Imposta condizioni basata sulle **espressioni**, ovvero il breakpoint viene raggiunto ogni volta che un espressione restituisce **true**, **conteggi di hit**, un controllo di quante volte un punto di breakpoint deve essere raggiunto prima che finisca l'esecuzione, o una combinazione di entrambi.

Triggered breakpoint

Il punto di breakpoint viene attivato quando viene raggiunto un prossimo punto di interruzione.